Informatics

Extending Cross-Blockchain Token Transfers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Kühne, BSc
Matrikelnummer 01426065

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 6. Mai 2020

Matthias Kühne	Stefan Schulte

Informatics

Extending Cross-Blockchain Token Transfers

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Matthias Kühne, BSc
Registration Number 01426065

to the Faculty of Informatics

at the TU Wien

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

Vienna, 6th May, 2020

Matthias Kühne　　　　Stefan Schulte

Erklärung zur Verfassung der Arbeit

Matthias Kühne, 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, 6. Mai 2020

Matthias Kühne

Acknowledgements

First of all, I would like to thank my advisor Associate Prof. Dr.-Ing. Stefan Schulte, who provided me with valuable technical aspects and constructive feedback. Furthermore, his excellent response time to all requests enabled me to finish this thesis within reasonable time. I would also like to thank Michael Borkowski, who always responded to all my technical questions in a very fast and detailed manner, which helped me a lot during the elaboration of this thesis.

Additionally, I am very grateful for to support of my family throughout this whole period of time, especially for the encouragement by my wonderful wife Julia and my son Johann.

Kurzfassung

Blockchain-Technologien und Kryptowährungen haben in den letzten Jahren einen großen Aufschwung erlebt. Die erste Blockchain, Bitcoin, wurde im Jahr 2008 vorgestellt und war auf die Anwendung für dezentrale Kapitaltransaktionen fokussiert. Später folgten sogenannte Blockchains der zweiten Generation, wie z. B. Ethereum. Diese ermöglichten das Ausführen von Code auf Blockchains mittels Smart Contracts. Durch neue Forschungsarbeiten werden immer mehr Anwendungsfälle für Blockchains gefunden, was aufgrund der verschiedenen Anforderungen in der Erstellung neuer Blockchains und Kryptowährungen resultiert.

Dies führt zu einer starken Fragmentierung des Forschungsfeldes und zu oftmals inkompatiblen Technologien. Dem entgegenwirkend ermöglicht das Deterministic Cross-Blockchain Token Transfers (DeXTT) Protokoll Interoperabilität zwischen Blockchains. Es erlaubt den Nutzern das Übertragen von Tokens auf mehreren Blockchains durch sogenannte Token-Transfers. Momentan gibt es bereits eine Implementierung von DeXTT für Ethereum-basierte Blockchains. Das Ziel dieser Arbeit ist es, das DeXTT-Protokoll um eine Implementierung für eine weitere Blockchain-Technologie zu erweitern.

In dieser Arbeit führen wir eine formale Definition für die Anforderungen des DeXTT-Protokolls ein. Des Weiteren erstellen wir eine Erhebung der technischen Aspekte von aktuellen Blockchain-Technologien in Bezug auf diese Anforderungen. Basierend auf dieser Erhebung wurde Bitcoin als Basistechnologie für eine Erweiterung des DeXTT-Protokolls ausgewählt

Zusätzlich schlagen wir ein Designkonzept für DeXTT auf der Bitcoin-Blockchain vor, in welchem wir DeXTT-Daten in Null Data Outputs von Bitcoin inkludieren und Bitcoin Core zur Kommunikation mit der Blockchain nutzen. Eine konkrete Implementierung dieses Designkonzepts wird mittels der Programmiersprache Java erstellt. Wir evaluieren diese Implementierung in Bezug auf die erforderliche Gültigkeitsdauer von Token-Transfers und der dabei anfallenden Kosten. Diese Evaluierung liefert eine minimale Gültigkeitsdauer der Transfers von vier Blöcken (3000 Sekunden) bei der Nutzung von unbestätigten Transaktionen bzw. fünf Blöcken (2400 Sekunden) für die ausschließliche Nutzung von bestätigten Transaktionen. Des Weiteren erbrachte die Evaluierung folgende konkrete Kosten für den durchschnittlichen Bitcoin-Preis und die durchschnittlichen Transaktionengebühren vom April 2020: 1,1451 USD für den Empfänger eines Transfers und mindestens 0,2395 USD für jeden Zeugen des Transfers pro Blockchain.

Abstract

Blockchain and cryptocurrency technologies have emerged rapidly in recent years. The first proposed blockchain was Bitcoin in 2008, with the primary usage for decentralized monetary transactions. Later, so-called second generation blockchains like Ethereum emerged, introducing the concept of smart contracts that enable code execution within blockchain. More recent blockchain research includes also completely other directions and use cases. Since each use case yields potentially different requirements, current blockchain technology is exposed to a fast pace in change, including the creation of completely new blockchains and cryptocurrencies.

This results in a big fragmentation of the field and mostly incompatible blockchain technologies arise. The Deterministic Cross-Blockchain Token Transfers (DeXTT) protocol ensembles an approach that provides means of blockchain interoperability and therefore tackles the problem of blockchain fragmentation. It enables users to record the transfer of tokens on an arbitrary number of blockchains simultaneously. Currently, an Ethereum prototype for the DeXTT protocol exists. The aim of this thesis is to extend the DeXTT protocol by implementing the protocol on another suitable blockchain technology.

Within this thesis, we formally define the requirements on blockchains to support a DeXTT implementation. Furthermore, we introduce a survey on current blockchain technologies in regards to their technical aspects concerning the DeXTT requirements. Based on this survey, Bitcoin was chosen as a base technology to extend the DeXTT protocol.

In addition, we propose a design for DeXTT on the Bitcoin blockchain, where we embed DeXTT payloads into null data outputs of Bitcoin transactions and utilize the Bitcoin Core client to access the blockchain. A concrete implementation of this design concept is created using the Java programming language. We evaluate the implementation in regards of the required transfer validity period and costs of DeXTT token transfers. This evaluation yields a minimum validity period of four Bitcoin blocks (2400 seconds) when using unconfirmed transactions, and five blocks (3000 seconds) for the usage of confirmed transactions only. Furthermore, the evaluation runs yield the concrete cost of DeXTT transfers for the average Bitcoin transaction fees and price in April 2020: 1.1451 USD for the receiver of a transfer and at least 0.2395 USD for each witness of the transfer per blockchain.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Since the introduction of Bitcoin [Nak08], blockchain and cryptocurrency technologies have emerged rapidly and even gained more attention in recent years [Zoh15]. While in the early stages of blockchain research and development, the utilization for decentralized monetary transactions was primarily considered, second generation blockchains, like Ethereum [Woo14], additionally introduced the concept of smart contracts, enabling code execution within blockchains [TS16]. To achieve this, these blockchains also introduce a quasi Turing-complete language, such as Solidity for Ethereum, and an execution environment for their smart contracts, such as the Ethereum Virtual Machine (EVM) for Ethereum [Dan17].

Recent blockchain research also follows completely other directions, having in mind novel use cases in different fields. Research includes general industrial applications [AM19], various economic and engineering fields [FF18], but also more specialized directions such as Business Process Management (BPM) [Pry+20; Men+18a], healthcare [LXS19] or applications for anti automotive counterfeiting [Lu+19]. Generally speaking, blockchains could potentially be applied anywhere where there is a need to execute transactions and store data in a decentralized way [Sch+19].

Because each blockchain use case has potentially different requirements for its implementation and features, current blockchain technology is exposed to a fast pace in change. This change includes the creation of completely new blockchains and cryptocurrencies or the addition of changes to existing blockchains that enable new functionalities [Yli+16]. Another possible way to introduce new features is the introduction of additional layers on top of an existing blockchain [Wil+19]. These practices inevitably lead to a big fragmentation in research and development, resulting in mostly incompatible technologies.

The number of listed blockchains or cryptocurrencies respectively on CoinMarketCap[1] can be adducted as a way to illustrate this fragmentation. As of March 2020, there are more than 5167 different cryptocurrencies listed on this platform.

As a result of this fragmentation, additional problems for users arise, as they must decide which blockchain and cryptocurrency they want to use. New blockchain technologies need to build a large user base first and also have a higher risk of containing undiscovered potential security issues, leading to potential loss of funds [Nof+17], but they offer novel features and technologies. In contrast, old blockchains do not offer novel features, but are instead more likely to be secure, as they have already been analyzed more deeply [Li+17]. For developers of blockchain-based applications, there also arises the question of which blockchain to base their applications on, as interoperability of blockchains is mostly not given.

The size of a blockchain's user base is also a crucial factor for the choice of an appropriate blockchain. Not only does it define the number of users that can be interacted with, but it is also an important part for the proper and secure functionality of a blockchain due to the distributed consensus rules [Nar+16].

Because of this fragmentation in research and development of blockchain technologies and also the competition for a high user base, there should be a different approach for users to choose between blockchain technologies.

This leads to the conclusion that the ability to interoperate between different blockchains needs to emerge, enabling users to select blockchains dynamically according to new trends and needs. Currently available approaches for blockchain interoperability provide a very limited set of possible interactions, which are represented mostly in the form of atomic swaps [Her18], where assets of different cryptocurrencies are atomically swapped. These swaps can happen without the need for a trusted third party but they still do not allow real interoperability, as transactions on one chain do not affect other blockchains.

Due to these limitations, interoperability of blockchains is therefore still actively researched [Bor+19b; Liu+19; Zam+19; ZAA19; Zam+18; JDX18; Sir+19; KP19] and new projects [Met18; Woo16; ZW17] currently arise in the field.

Nevertheless, as of today, the following actions can still only be performed within a single blockchain [Sch+19]:

- the sending of tokens between participants,
- the execution of smart contracts,
- the storing of data with guaranteed validity in a blockchain.

The ability to carry out these actions across multiple blockchains would ultimately lead to less fragmentation and easier use of blockchain technologies.

[1] `https://coinmarketcap.com/all/views/all/`

1.2 Aim of the Work

There is quite some research being done and ready-to-use software available, regarding atomic swaps between blockchains where different protocols exist to swap funds on diverse blockchains. Regarding further approaches of blockchain interoperability, some novel research directions are currently explored, resulting in new protocols for interactions between blockchains. One very promising candidate is the Deterministic Cross-Blockchain Token Transfers (DeXTT) protocol [Bor+19b], which enables users to record the transfer of tokens on an arbitrary number of blockchains simultaneously, while being completely decentralized without the involvement of an untrusted third party. The DeXTT is described in detail in Section 2.4.

There already exists a prototype[2] of the DeXTT protocol developed to use with Ethereum-based blockchains, which is conceptually portable to other blockchains. The main aim of this thesis is to extend the DeXTT protocol by implementing the protocol on another suitable blockchain technology, evaluate the implementation and compare the outcome with the already proposed implementation and its evaluation on the Ethereum blockchain. The following aspects describe the problems to be solved within this thesis.

Requirements for DeXTT Implementations. To choose an appropriate blockchain for a new DeXTT implementation, the protocol requirements have to be formally defined. The exact technically specified demands of the protocol for an underlying blockchain technology allows the comparison of the requirements with the specifications and functionalities of possible blockchain candidates. This corresponds to the research questions: “What are the technical requirements on a blockchain to support the DeXTT protocol?”.

Suitable Blockchain Candidate. Within this thesis, a suitable blockchain candidate to serve as a platform for a new DeXTT implementation is chosen. For this, it is necessary to take into account the formal requirements of the DeXTT protocol and also the properties and features of different blockchain technologies currently available. This can be formulated as the following research question: “Which blockchain is currently best suited and yields the most research value for a new DeXTT implementation?”.

DeXTT Design for Chosen Blockchain. A new implementation of the protocol requires the creation of an appropriate design for the chosen blockchain. The details of the newly created design are highly depending on the chosen blockchain and yield a theoretical point of view of how DeXTT works within the bounds of the blockchain’s features. The corresponding research question can be formulated as: “What design choices are required to integrate the DeXTT protocol into the chosen blockchain?”.

[2]https://github.com/pantos-io/dextt-prototype

DeXTT Implementation for Chosen Blockchain. The main outcome of this thesis is an implementation of the DeXTT protocol. This artifact will be developed for the blockchain platform that has been determined to be best suitable and will be based on the already existing prototype and the elaborated design. The according research question reads as follows: “How can the DeXTT protocol be implemented to work with the chosen blockchain technology?”.

Evaluation and Comparison of Implementation. The newly implemented software needs to be evaluated in detail. The evaluation includes a quantitative analysis of the required DeXTT transaction durations and the actual execution cost on the blockchain and also how the implementation compares to the already existing prototype on Ethereum. This part corresponds to the following research questions: “What DeXTT transaction durations are required for the new implementation? How high are the cost for the DeXTT protocol on the chosen blockchain? How do these values compare to the DeXTT Ethereum prototype?”.

1.3 Methodology and Approach

The used methodology and approach for this thesis can be roughly broken down into the following parts.

Breakdown of Requirements for DeXTT Implementations. To define a detailed collection of formal requirements for a DeXTT implementation, the protocol itself and the currently available prototype have to be analyzed. The availability of a certain possible computational complexity for programs and also the existence of sufficient hash function implementations on a smart contract platform are important parts for an implementation to be feasible. Furthermore, the requirements must be distinguished between pure blockchain implementations, meaning that all of the protocol’s logic runs on the blockchain, and implementations where parts or all of the protocol’s logic run on the client side.

Survey on different blockchains. To be able to select a suitable and reasonable blockchain technology for the DeXTT protocol implementation, a detailed survey of qualified candidate blockchains have to be created. As there exists a high number of different blockchains, it is crucial to already preselect promising candidates, which are then analyzed concerning their technical aspects. The survey will ultimately lead to the decision of which blockchain to use for the DeXTT implementation. The main factor for the preselection of blockchains is their rank (based on their market cap) on platforms such as CoinMarketCap[3], as the significance of the chosen blockchain is an important factor.

Design, Creation and Description of New DeXTT Implementation. The main part of this work is the design and implementation of a new DeXTT prototype

[3]https://coinmarketcap.com/

for the chosen blockchain platform. For this, the currently existing Ethereum prototype, the technology behind DeXTT and also the technical aspects of the chosen blockchain need to be analyzed and understood precisely. The prototype will then be built making use of best practices in software engineering and solutions specialized for the given platform and the chosen programming language. The implementation will be designed to act as close to the prototype for the Ethereum platform as possible, to ensure its compatibility and comparability.

Evaluation of New Implementation. For the evaluation of the newly created implementation of the DeXTT protocol on the chosen blockchain platform, private blockchains will be deployed to ensure a high level of controllability of the evaluation environment. The use of private blockchains allows a high level of reproducibility and controllability for the evaluation runs and does not introduce any additional cost other than for the hardware on which the blockchains and the DeXTT clients are executed. The data raised from the evaluation will then be compared to the data of the already existing Ethereum prototype. Additionally, both data sets can be joined to allow a more general perception and conclusions about the DeXTT protocol, regarding its cost and transaction duration.

1.4 Structure

The structure of this thesis is constructed as follows:

Chapter 2 introduces all important background knowledge required for the main part of this thesis. This includes a basic description of blockchain technology, followed by the fundamentals about Bitcoin and Ethereum. The last part of the chapter presents the DeXTT protocol in detail. Within Chapter 3, an overview about the current state of the art concerning different blockchain interoperability techniques is presented. In Chapter 4, requirements for the DeXTT protocol are presented and properties and features of different currently available blockchain technologies are analyzed concerning their technical aspects. Last, the selection rationale of the blockchain candidate for the DeXTT protocol implementation is described.

Chapter 5 presents the design approaches for a DeXTT implementation for the Bitcoin blockchain. The introduced design takes all specific implementation choices of the Ethereum prototype into account. Within Chapter 6, details about the concrete implementation of the DeXTT protocol on the Bitcoin blockchain are discussed. In Chapter 7, the evaluation of the DeXTT-Bitcoin implementation is presented. The evaluation includes a quantitative analysis of the required DeXTT transfer validity period and the actual execution cost on the blockchain. Additionally, the results are compared and combined with the results of the evaluation of the Ethereum prototype. The thesis is concluded by Chapter 8, where the work and results are summarized and relevant future work is discussed.

CHAPTER 2

Background

This chapter gives an introduction to fundamental knowledge required for the later parts of this thesis. First, the basics of blockchain technologies are described briefly, including a short description of smart contract platforms. Furthermore, the basics of Bitcoin are presented together with all relevant aspects of Bitcoin that are used in the following chapters.

Next, the fundamentals of Ethereum are described, including all parts necessary for a good understanding of the DeXTT Ethereum prototype and comparisons of the results with the results on Ethereum. This part focuses on the aspects that differentiates Ethereum from Bitcoin.

The chapter is concluded by a detailed breakdown of all relevant parts of the DeXTT protocol.

2.1 Blockchain Basics

The principle of a blockchain as it is used in today's cryptocurrencies was first introduced by Nakamoto [Nak08] as part of the unveiling of the Bitcoin blockchain and cryptocurrency in 2008. The blockchain itself in its core is actually a type of distributed data structure. The basic ideas behind the technology will be presented in the following sections.

2.1.1 Hash Pointers

An important concept for blockchains is the use of hash pointers, which are simply pointers to the location of stored information, as a regular pointer, but they additionally provide a cryptographic hash of the information pointed at. This hash can be used to check whether the information the hash pointer references, has changed [Nar+16].

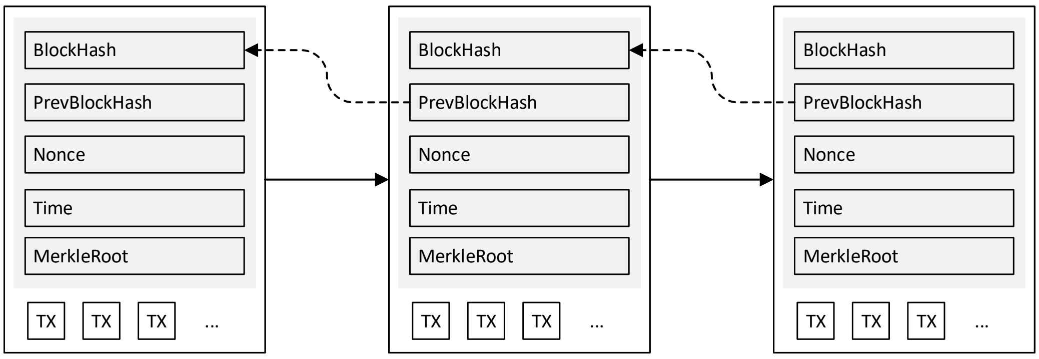

Figure 2.1: Simplified blockchain structure [TS16].

A hash function is a function that produces a fixed size output from an input of arbitrary size and is efficiently computable [Nar+16]. A cryptographic secure hash function additionally provides three additional properties: *collision-resistance*, *hiding* and *puzzle-friendliness*. Details about these properties can be found in [Nar+16].

Blockchains make use of such hash pointers in different ways, including the usage for linking blocks as in a linked list and to build Merkle Trees, which are binary trees that use hash pointers to reference the tree nodes.

2.1.2 Blockchain Structure

A blockchain can be seen as a ledger, containing all transactions of the corresponding currency in a totally ordered manner. Contrary to traditional banking systems, there is no centralized party where this ledger is stored. The ledger is distributed among all participants, called nodes, each of them storing a local copy of the blockchain, making a blockchain a distributed ledger [TS16].

The basic structure of a blockchain is a linked list of blocks, but utilizing hash pointers instead of ordinary pointers. This means that each block also contains a cryptographic hash value of the block it points to, which allows to check if the block pointed at has changed values. Due to this verification ability, any attempts to tamper blocks in the blockchain can be detected, thus enabling blockchains to be utilized as a tamper-evident log [Nar+16]. Because existing blocks cannot be changed anymore, the only way of adding data to the blockchain is to add new blocks, it can therefore be seen as a write-only log [Zoh15].

A simplified illustration of the basic structure of a blockchain is shown in Figure 2.1, where each block also holds the hash value of the previous block.

2.1.3 Distributed Consensus

A typical problem for a distributed currency system is achieving distributed consensus among the participating nodes, which is required because there exists no trusted third party like a bank for the consensus. Blockchains face two types of problems when trying to achieve distributed consensus: The imperfection of the network that connects the nodes and attempts of adversaries to subvert the protocol [Nar+16].

Distributed consensus is achieved in blockchain systems by two simple rules in their consensus algorithm [Zoh15]:

1. The creator of a new block gets chosen at *random* at each round [Nar+16]. For instance in Bitcoin, this is achieved by making block creation difficult by design, called Proof of Work (PoW). The block creation in Bitcoin is called mining [Zoh15].
2. The adoption of the longest valid chain. If nodes receive conflicting blocks that make up a longer consistent chain than before, they adopt to the longer chain and abandon the blocks in the shorter branch of the blockchain [Zoh15].

The rule to adopt the longest valid chain is effectively preventing the double spending of coins, where two transactions spending the same coins are created. Ultimately, provided by the consensus rule, only one of these two conflicting transactions will end up in the longest chain. The probability that the chain changes to the other transaction in a possible longer chain decreases exponentially by the number of new blocks created after the block that contains one of the two double spending transactions [Nar+16].

2.1.4 Transactions

The transactions that make up the content of the distributed ledger of a blockchain are the mechanism of cryptocurrencies to change ownership rights of coins. Coins, on blockchains like Bitcoin, are not ordinary coins, but rather a chain of transactions making up the amount of coins that can be spent, called Unspent Transaction Output (UTXO) [Nar+16]. In contrast, blockchains like Ethereum use the concept of account balances as coins [AW18].

Each participant of a blockchain needs at least a public/private key pair to interact with the blockchain. Using this key pair, the participant can create transactions and cryptographically sign them to prove that they are allowed to spend the referenced coins [TS16].

A public/private key pair consists of matching asymmetric keys that can be used for encryption and decryption of data. As their names suggests, the public key is meant to be shared with others, whereas the private key should be kept secret by its owner.

Data that is encrypted by the public key of the key pair yields a message that can only be read by the owner of the corresponding private key. Data that is encrypted by the private key of the key pair yields a message, called *digital signature*, that can be read and verified by anyone who knows the public key, but it could have only been produced by the owner of the key pair knowing the private key [Her19]. More details about digital signatures can be found in [Nar+16].

2.1.5 Smart Contracts

Bitcoin and other first generation blockchains already support smart contracts to some extent through a limited scripting language that is used to unlock UTXOs. More details about its limitations are given in Section 2.2. Second generation blockchains additionally introduce the concept of quasi Turing-complete smart contracts. The first blockchain to utilize that feature was Ethereum [Woo14].

A blockchain that supports smart contracts in the style of second generation blockchains provides the ability to deploy a compiled contract to the blockchain. Such a contract can be seen as a program that lives on the blockchain and can be invoked by a user of the blockchain or even by other smart contracts. If a contract is invoked, it is executed by each blockchain node in its blockchain-specific execution environment, e.g., the EVM for Ethereum [Men+18b].

Because such a smart contract resembles an object from an object-oriented perspective, it also has a state, which can change after each invocation. Each new state of a contract is recorded and therefore saved on the blockchain, after the contract has been executed by the node creating a new block [Her19; Dan17].

Smart contracts are usually written in a high-level language, such as Solidity for the Ethereum blockchain, which is then compiled to a low-level byte code that can be executed in the given execution environment. Only the compiled byte code is deployed to the blockchain.

Such quasi Turing-complete smart contracts can be used to model complex scenarios, but there are also some pitfalls, mainly through software vulnerabilities in a contract's code. Because such contracts can be used to manage funds, vulnerabilities can lead to a total loss of funds [Her19].

2.2 Bitcoin

Bitcoin, introduced by Nakamoto [Nak08], is the first peer-to-peer electronic cash system that utilizes blockchain technology for its distributed ledger as described in Section 2.1. Bitcoin is also by far the most popular cryptocurrency as of March 2020, being ranked on first place based on its market cap on CoinMarketCap[1].

Bitcoin is used to store payment information in a decentralized way in its blockchain in the form of Bitcoin transactions. It also offers a stack-based scripting language called Script,

[1]`https://coinmarketcap.com/currencies/bitcoin/`

Table 2.1: Different Bitcoin address types [bit18a].

Address type	**Chain**	**Leading Symbol(s)**	**Example**
P2PKH	Mainnet	1	**1**7VZNX1SN5NtKa8UQFxwQbFeFc3iqRYhem
P2SH	Mainnet	3	**3**EktnHQD7RiAE6uzMj2ZifT9YgRrkSgzQX
Bech32	Mainnet	bc1	**bc1**qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4
P2PKH	Testnet	m *or* n	**m**ipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn
P2SH	Testnet	2	**2**MzQwSSnBHWHqSAqtTVQ6v47XtaisrJa1Vc
Bech32	Testnet	tb1	**tb1**qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx

which is utilized to determine if funds, or rather UTXOs, can be unlocked by a given transaction. In contrast to smart contract languages of second generation blockchains like Ethereum, Script is not Turing-complete and rather limited in its features [Nar+16].

2.2.1 Addresses

In Bitcoin, identities of its users are generated in a decentralized way by the users themselves. Because the addresses that serve as identities are derived from a public key, creating a new address requires only the generation of a new public/private key pair using the given digital signature scheme, i.e., no central authority is necessary. Bitcoin users can generate as many addresses as they desire. It is actually recommended to use a fresh receiving address for each transaction to enhance the user's privacy [Nar+16; bit20d].

Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] as its digital signature scheme, which is an U.S. government standard. Moreover, the standardized *secp256k1* elliptic curve [Hes00] is used for its signature scheme [Nar+16].

Addresses in Bitcoin are derived from a public key by first hashing it with the SHA-256 hashing algorithm and then the RIPEMD-160 algorithm subsequently. Then, a version number is prepended and a checksum is appended. A base58-encoding is then used to eliminate any ambiguous characters from the address string. The address derivation is done to shorten and obfuscate the public key and also to improve its readability [TS16]. Bech32 type addresses use base32-encoding instead of base58-encoding [WM17].

The version number prefix of addresses affects the leading symbol in the encoded address string, different address types and Bitcoin blockchain networks use different leading symbols as shown in Table 2.1.

There is a distinction between Pay to Public Key Hash (P2PKH) [bit20d], Pay to Script Hash (P2SH) [And12] and Bech32 [WM17] addresses in Bitcoin, indicated by their prefix. The distinction is needed because the transaction mechanism for these different types differs from each other. More on different transaction types is presented in Section 2.2.5.

Bitcoin address prefixes are also different for different blockchain networks, if using another network than the Bitcoin mainnet, such as the Bitcoin testnet [bit19h; bit19i]. Additionally, there exists the *Regtest mode* [bit19i], which is similar to the testnet, but uses a private blockchain. For Bech32 addresses, the address prefix for the *Regtest mode* is different from the testnet prefix, having a value of `bcrt1`[2]. More details on different Bitcoin blockchain networks are discussed in Section 2.2.7.

2.2.2 Block Creation

In Bitcoin, there are two incentives that make it desirable to create new blocks that include valid transactions. One incentive is a block reward, which allows the creator of a block to include a special coin-creation transaction in the block. The recipient address can be chosen arbitrary. Typically, it is the address of the node that created the block. This can be seen as payment to the node for creating and validating a new block. The block reward started with a value of 50 Bitcoins and halves every 210,000 blocks [Nar+16].

The second incentive to create blocks including valid transactions are *transaction fees*. This is achieved by spending less coins on the total value of transaction outputs than on the total value of transaction inputs, i.e., the sender of a transaction pays the fee. The change is given to the creator of the block that includes the transaction [Nar+16]. More details about transaction fees are discussed in Section 2.2.6.

The creator of the next new block is determined through a cryptographic hash puzzle, which approximates the selection of a random node by selecting nodes in proportion to their computing power. This process is called PoW [Nar+16]. This hash puzzle requires that the hash of each block header is smaller than a threshold value called *target*. By changing the *nonce* field in the block header, which can contain arbitrary data, the hash of the header can be recomputed until the hash suffices to the target value [Zoh15].

The target is defined by the number of leading zeros for the binary hash value. The SHA-256 function is used for the hash calculation. For n leading zeros for the hash target, on average 2^n attempts are needed to solve the hash puzzle. The number of leading zeros therefore represents the *difficulty* to solve the cryptographic hash puzzle and to create a new valid Bitcoin block [TS16]. The process of repeatedly attempting to create new valid blocks is called *mining* and the participating nodes are called *miners* [Nar+16].

There also exist other techniques to select random nodes in proportion to other means than computing power, for instance when using Proof of Stake (PoS), the selection is done in proportion to ownership of the currency [Nar+16].

When a transaction is included in a valid block, it is considered a confirmed transaction (1 confirmation), otherwise it counts as unconfirmed (0 confirmations) and can generally not be considered valid. The confirmation number of a transaction is defined by the number of blocks in the blockchain after the block that includes the transaction (including

[2]`https://github.com/bitcoin/bitcoin/issues/12314`

the block that includes the transaction itself). A confirmation number of six is a common heuristic to be sufficiently convinced that the transactions is included in the blockchain [Nar+16].

To provide a certain stability and reasonable waiting times for transaction confirmations, the target valid for the cryptographic hash puzzle for the block creation is adjusted every 2,016 blocks. It gets adapted to approximately meet a block creation rate of one block every 10 minutes. This means, the target is recalculated every 2 weeks ($2{,}016 \cdot 10$ minutes) on average [TS16].

2.2.3 Block Structure

Transactions in Bitcoin are grouped in blocks as an optimization, because it is faster for the participants to reach consensus on a block of transactions than on each transaction individually. Furthermore, a hash pointer chain made up of blocks containing multiple transactions is shorter than chaining up individual transactions, which makes it easier to verify the data structure [Nar+16].

To store all transactions of a block, a Merkle tree is utilized. This is a binary tree that uses hash pointers and allows to create a digest of all transactions in a block in an efficient way. It also allows to check if a transactions is included in a block by searching a path in the Merkle tree. The length of such a path is logarithmically bound by the number of transactions in a block [Nar+16].

A Bitcoin block consists of a block header bundled together with all transactions of the block, arranged in a tree structure [Nar+16]. The block header contains the following data fields:

Block Version. The block version specifies which version of validation rules have to be followed for the block. Rule changes can be introduced by soft forks. Currently, version 4 is used [bit20c].

Previous Block Hash. Contains the SHA-256 hash of the block header of the previous block and functions as a hash pointer for the blockchain structure. The hash ensures that previous blocks cannot be changed without also changing the header of this block [bit20c].

Merkle Root Hash. The root of the Merkle tree data structure used for the transactiosns in the block. This hash value is derived from all hashes of the transactions that are included in this block. This ensures that no transaction in the block can be changed without also changing the block header [bit20c].

Time. Contains the block time as an Unix epoch timestamp, meaning it is represented as seconds since *1970-01-01T00:00 UTC*. It is set usually to the time the miner started hashing the header [bit20c]. To be valid, it must be strictly greater than the median timestamp value of the last 11 blocks and at most 2 hours in the future compared to the network-adjusted time. The network-adjusted time of a node is the median time of all nodes that are connected to this node. This means that the timestamps in block headers are not accurate, allowing an error of a few hours [bit19b].

NBits. This field encodes the target threshold that the hash of this blocks header must be less or equal to in order to be valid. It therefore defines the difficulty for the block creation. The value is encoded in a less precise format as the field only has a size of 32 bit to represent a 256 bit value [bit20c].

Nonce. An arbitrary number that can be changed in the mining process to find a block header hash that is less or equal to the current target threshold [bit20c].

A Bitcoin block is limited in its size. Before the Segregated Witness (SegWit) soft fork [LLW15], the block size was limited to 1MB (1,000,000 *bytes*) in total size. SegWit introduced a new unit called *block weight*, which is defined as $Basesize \cdot 3 + Totalsize$, where $Basesize$ is defined as the block size in bytes using the original block serialization without any witness-related data. The $Totalsize$ is the block size in bytes where transactions are serialized including witness data as defined in [LW16].

The new rule after the SegWit soft fork is defined as $blockweight \leq 4{,}000{,}000$ [LLW15].

2.2.4 Transactions

The transactions on the Bitcoin blockchain do not make up an account-based currency system, because that would mean if someone wants to verify if a transaction is valid, they would have to keep track of these account balances. That would require to not only save transactions that transfer coins themselves, but also to use more efficient data structures that track balances after each transaction or block. Without additional data structures, one would need to look backwards in time to scan all previous transactions of an account to verify a new transaction [Nar+16].

The overcome these problems with account-based systems, Bitcoin uses another approach for its ledger, that only requires to keep track of the transactions themselves. The transactions specify a certain number of inputs and outputs, the inputs consume coins of previous outputs and the outputs create new coins that can be consumed later, called UTXOs (see Section 2.1.4). To reference other outputs, each transaction has a unique identifier, the outputs are then indexed beginning with 0. Transactions that create new coins are an exception that do not consume other coins of previous outputs [Nar+16].

Because previous outputs can only be used once as transaction inputs, the entire amount of each input needs to be assigned to the new output of a transaction. Any amount that is not consumed in an output will be collected by the miner of the block as a transaction fee. Because it is often necessary to only transfer parts of the coins of a transaction input, the remainder of the coins need to be assigned to a new output, effectively sending the amount of Bitcoins back to the sender itself, creating a new UTXO that can be consumed again. It is considered best practice to actually use a fresh change address for this new output each time a transaction is made [Nar+16; bit20d].

To check whether a transaction output has been spent already, it is only necessary to scan all transactions between the referenced output and the transaction that uses that output as an input, as it is impossible to spend an output before it is created. Therefore, it is not necessary to check to entire blockchain to verify a transaction [Nar+16].

Listing 2.1: A Bitcoin transaction [Nar+16].

```
1  {
2    "hash":"5a42590fb..."
3    "ver":1,
4    "vin_sz":2,
5    "vout_sz":1,
6    "lock_time":0,
7    "size":404,
8    "in":[
9      {
10       "prev_out":{
11         "hash":"3be4ac972...",
12         "n":0
13       },
14       "scriptSig":"30440b6a7..."
15     },
16     {
17       "prev_out":{
18         "hash":"7508e6ab2...",
19         "n":0
20       },
21       "scriptSig":"3f3a4ce81..."
22     }
23   ],
24   "out":[
25     {
26       "value":"10.12287097",
27       "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b... OP_EQUALVERIFY
           ↪ OP_CHECKSIG"
28     }
29   ]
30 }
```

In Listing 2.1, a low-level example of an actual legacy Bitcoin transaction is shown in human-readable format, which gets converted to a compact binary format on the blockchain. A transaction is composed of three parts:

Metadata. This part of a transaction (line numbers 2-7 in Listing 2.1) is mainly used for housekeeping. The metadata contains a hash value of the entire transaction, the *transaction ID*, serving as a way to uniquely identify the transaction. Furthermore, a transaction contains information about its input and output sizes and the transaction size. The *lock_time* is used to delay the publication of the transaction by the miners until either a specific block number or a certain point in time [Nar+16].

Inputs. The inputs of a transaction (line numbers 8-23 in Listing 2.1) form an array, each entry representing one transaction input. An input references a previous transaction via its hash value and uses an index value to specify the referenced output of that transaction. Each output can be used and therefore referenced only once as an input, otherwise it is an attempt to spend the same coin twice, called *double spending*, which is forbidden. An output can therefore only be either an UTXO that has not been referenced yet or a Spent Transaction Output (STXO) that has already been spent [TS16]. Each input also contains a field *scriptSig*, where in the simplest case, a signature is specified, that is used to claim the referenced output. But this field actually contains a redeem script using Bitcoins *Script* language and can therefore be more complex than just a single signature [Nar+16]. More details about these scripts are given in Section 2.2.5.

Outputs. Just as the inputs, the outputs (line numbers 24-29 in Listing 2.1) also form an array of transaction outputs. Each of them contains a *value* that specifies the amount of coins that output represents. The sum of the values of all outputs have to be less or equal to the sum of the values of all referenced outputs of the transaction inputs. If the output sum is less than the input sum, the difference represents the transaction fee that can be claimed by the miner [Nar+16]. The outputs additionally specify a field called *scriptPubKey*, which contains again a Bitcoin *Script* that is used to specify how the output can be claimed. In the case of the transaction in Figure **??**, the script in the given output specifies a P2PKH script, effectively only allowing the claiming of this output to the owner of a given public key that is given in its hashed form inside the script [Nar+16]. More details on how *Script* works are given in Section 2.2.5.

SegWit transactions [LLW15] that were introduced through a soft fork in 2017 [Fry17] specify additional fields in the transaction structure. SegWit was proposed to solve the malleability problem, doing so by redesigning the structure of transactions. Additionally, through SegWit the maximum block size was also increased. The idea of SegWit is to decouple the information needed for transaction verification from the rest of the transaction. This detached data includes scripts and signatures that are than placed in a new structure called *witness* [Pér+19]. Through the SegWit soft fork, the following new fields for SegWit transactions are defined [LLW15]:

Marker. Must be a zero byte, having a value of $0x00$.

Flag. Must be a single byte that is not zero. Currently, $0x01$ must be used.

Witness. This field includes the serialization of all the witness data of the transaction, where each transaction id can be associated with one entry in the witness field.

Additionally, a new hash value, called *wtxid* is included, which is the SHA-256 hash of the traditional transaction data combined with the newly defined fields [LLW15]. *Bech32* addresses represent native output addresses for SegWit transactions (see Section 2.2.5) [WM17].

2.2.5 Bitcoin Script

The scripting language that Bitcoin uses is called *Script* and was specially designed for the use within Bitcoin, but is very similar to a language called *Forth*, which is a simple stack-based programming language [Nar+16]. Script is also a stack-based language, which means that a stack is used as memory layout and that each instruction is only executed once. More precisely, Script does not support the concept of loops and therefore the number of instructions gives an upper bound for the runtime and memory consumption of a script [Nar+16]. This also means that the language is not designed to be Turing-complete, making it easier to handle and avoiding unintended side effects [TS16].

Functionality

The scripting language of Bitcoin gives a certain amount of programmability to what a transaction actually does. The most common form of a Bitcoin transaction is to redeem a previous output through a signature using the correct key. This is done by specifying the public key or its hash respectively within the transaction output that is to be redeemed. To encode such behavior, each output of a Bitcoin transaction contains a script called *scriptPubKey*, that expects some arguments. These arguments are given by a script called *scriptSig* inside the input within the transaction that wants to spend the output [Nar+16; TS16].

To validate whether a transaction redeems the output of a previous transaction correctly, the input script (`scriptSig`) of the new spending transaction is combined with the output script (`scriptPubKey`) of the previous output by simply concatenating them. The resulting combined script, `{<scriptSig> <scriptPubKey>}`, must run successfully, yielding the output value *true*, for the spending transaction to be valid. Generally, there are only two possible outcomes of a script execution, either it runs successfully yielding (output *true*), making the transaction valid. Or the scripts execution yields an error, in that case the transaction is invalid and must not be included in a block [Nar+16].

The Bitcoin Script language only allows for 256 different instructions, because such opcodes are encoded within a single byte value. But not all of those 256 possible values are actually assigned to a valid instruction, some of them are reserved, meaning that they do not have any specific meaning yet, but might be added to the protocol in later versions. There are also some instructions that existed in early versions of the Bitcoin protocol, but were removed in later versions, because the clients might have a bug in their implementations. The motivation of disabling those instructions comes from bugs such as the one found in the implementation of the *OP_LSHIFT* instruction, that could crash any Bitcoin node if exploited [bit19g].

Script includes many basic instructions that can be expected for a programming language, such as basic arithmetics, flow control, although without the support for loops, throwing errors or returning early. But there are some crucial limitations. Currently, all bitwise logic operators except for the equality check are disabled and additionally, all multiplication, division and shift instructions are disabled. The language also supports some important cryptographic instructions for hashing and signature verification. The following hash functions can be used natively via a single instruction: *RIPEMD-160*, *SHA-1* and *SHA-256*. There are also instructions for ECDSA [JMV01] signature verification using the *secp256k1* elliptic curve [Hes00], including the *OP_CHECKMULTISIG* instruction that checks multiple signatures using a single instruction [Nar+16; bit19g]. The signatures must be encoded using strict *DER* encoding [Wui15].

Standard Transactions

There are currently five script templates that can be used to create standard transactions. Any non-standard transaction that contains anything besides a standard *scriptPubKey* will neither be accepted, broadcast nor mined by peers and miners using the default Bitcoin Core settings. If they are already included in a block, transaction will avoid the *IsStandard* test and are therefore handled normally [bit20d]. Currently the standard *scriptPubKey* script types are as follows [bit20d]:

Pay to Public Key Hash (P2PKH). This is probably the most common and essential form of transaction script. It transfers coins to one or multiple Bitcoin addresses [TS16; bit20d]. The script template used for that type of transactions is shown in Listing 2.2.

Listing 2.2: P2PKH script template [TS16].

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
    ↪ OP_CHECKSIG
scriptSig: <sig> <pubkey>
```

The output script specifies a public key hash `<pubKeyHash>` together with the instructions of the script. Within the input script of its spending input, a signature `<sig>` together with its public key `<pubkey>` are specified. The script checks first, whether the `<pubKeyHash>` is indeed the hashed version of the `<pubkey>`, followed by checking if the `<pubkey>` matches the given signature `<sig>`, meaning that the according private key of the key pair was used to create the signature [TS16].

Pay to Script Hash (P2SH). Because of how Bitcoin Script works, the sender of coins has to specify the script to redeem the coins later. This can be counter-intuitive. For the sender it would be the easiest way to only specify an address to send coins to. This led to the introduction of P2SH [And12] transactions [Nar+16]. They move the responsibility for supplying the redeem condition from the sender to the receiver. The sender merely has to specify the hash value of the script that is used to redeem the coins [And12]. The template of a P2SH script is shown in Listing 2.3.

Listing 2.3: P2SH script template [TS16].

```
scriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL
scriptSig: [<sig> ...] <redeemScript>
```

The P2SH script first hashes the given `<redeemScript>` and checks whether the result matches the specified `<redeemScriptHash>`. If it matches, a special second step of validation is performed, the `<redeemScript>` is reinterpreted as a sequence of instructions and executed, with the rest of the stack (`[<sig> ...]`) as its input [Nar+16].

Multisig. Multisig, or m-of-n multi-signature transactions, require m valid out of n possible signatures for a transaction to be redeemed successfully. Because of an off-by-one error in the original Bitcoin implementation, `OP_CHECKMULTISIG` consumes one more element from the stack than indicated by m, therefore an extra value of 0 is added on the stack [TS16; bit20d]. The m-of-n multi-signature script template is shown in Listing 2.4.

Listing 2.4: m-of-n multi-signature script template [TS16].

```
scriptPubKey: <m> <pubKey> [<pubkey> ...] <n>
   ↪ OP_CHECKMULTISIG
scriptSig: 0 [<sig> ...]
```

The output script specifies the number of required signatures `m`, followed by n possible public keys `<pubKey>` and the number of possible signatures `n`. To redeem the coins in the output, the input script must first specify a zero value due to the bug in the opcode, followed by m signatures that each match a different public key listed in the output script [TS16].

Pay to Public Key (P2PK). These transactions are a simplified form of P2PKH transactions, where instead of a public key hash, the whole public key of the receiver is specified in the output script [bit20d]. This allows for simpler scripts as shown in the transaction template in Listing 2.5, but it comes with two downsides. First, the information that has to be shared between receiver and sender to make a transaction is significantly longer and second, the actual public key is revealed in advance, providing less protection in case the used ECDSA becomes vulnerable [bit19g].

Listing 2.5: Pay to Public Key (P2PK) script template [bit19g].

```
scriptPubKey: <pubKey> OP_CHECKSIG
scriptSig: <sig>
```

Null Data. This transaction type can be used to add arbitrary data to the blockchain using a provable unspendable output script. There exists no input script that can redeem coins in transaction outputs using Null Data. This also means that the output can be pruned from the UTXO set although being unspent, i.e., the output does not bloat the UTXO set as do other means of storing data on the blockchain [bit20d; bit19g]. Because this transaction outputs can not be spent, they usually contain a zero Bitcoin amount and there is a limit of one Null Data output per transaction [Ant14]. The size limit for the Null Data transaction to be relayed and mined in the network is currently 83 bytes in size for the whole *scriptPubKey* as of *Bitcoin Core 0.12.0*. This means that a maximum of 80 bytes of raw data can be included, because the Null Data opcode `OP_RETURN` (1 byte) and also a data push (2 bytes) have to be included [bit20d]. The script template is shown in Listing 2.6.

Listing 2.6: Null Data script template [TS16].

```
scriptPubKey: OP_RETURN <data>
```

Segregated Witness transactions

Through the introduction of SegWit transactions, there are now also adapted versions of the standard scripts to support the new transaction structure. *Bech32* addresses represent native SegWit output addresses and allow the usage of native SegWit transactions. It is also possible to nest SegWit transactions into P2SH transactions, using them non-natively with P2SH addresses [LLW15]. Beginning with *Bitcoin Core version 0.19.0.1*, Bech32 addresses and therefore also native SegWit transactions are the default format when using its GUI[3]. The following two adapted standard script templates are used for native SegWit transactions.

Pay to Witness Public Key Hash (P2WPKH). The template for the version 0 Pay to Witness Public Key Hash (P2WPKH) script is shown in Listing 2.7. The 0 in the *scriptPubKey* indicates that this is a version 0 witness program, the type is given by the length of the following data. The semantics stays the same as for P2PKH transactions, but the signature and public key are now given in the *witness* part of the spending transaction, i.e., there is no *scriptSig* anymore [LLW15].

Listing 2.7: P2WPKH script template [LLW15].

```
witness: <sig> <pubkey>
scriptSig: (empty)
scriptPubKey: 0 <20-byte pubKeyHash>
```

Pay to Witness Script Hash (P2WSH). Listing 2.8 shows the version 0 Pay to Witness Script Hash (P2WSH) script template. Again, the 0 in the *scriptPub-Key* indicates that this is a version 0 witness program, the transaction type is

[3]`https://bitcoin.org/en/release/v0.19.0.1`

distinguished by the length of the following data, being a 32 byte hash. The semantics is the same as for the P2SH script without SegWit, but the input for the `<witnesScript>` and the script itself are now included in the *witness* part of the spending transaction, i.e., *scriptSig* is not needed anymore [LLW15].

Listing 2.8: P2WSH script template [LLW15].

```
witness: [<sig> ...] <witnesScript>
scriptSig: (empty)
scriptPubKey: 0 <32-byte witnessScriptHash>
```

2.2.6 Transaction Fees

An incentive for miners besides the block reward, are transaction fees (see Section 2.2.2). Each transaction can give a certain amount of its input value to the miner of the block that includes the transaction, by not spending the sum of all input amounts within its outputs. The difference between the sum of Bitcoins of all inputs and the sum of Bitcoins spent in all outputs can be claimed by the miner as a transaction fee [Nar+16]. Because Bitcoin is designed to include transaction fees in that way, fees are defined and payed by the sender of a transaction [bit19e].

Transactions compete for the limited available space in each block, therefore market forces should eventually set the fee rate [Zoh15]. The minimum transaction fee that is necessary for a transaction to confirm within a given number of blocks varies over time and is determined through supply and demand of block space in Bitcoin's free market. The maximum block size is currently 1 million vbytes and as blocks are not produced on a fixed schedule (see Section 2.2.2), the effective maximum block size varies over time. This varying effective maximum block size defines the supply of block space. The demand of block space is given by the number of spenders, how much they are willing to pay as transaction fee and how long they are willing to wait for a confirmation. The demand varies according to patterns, such as the current day of the week [bit19e].

One vbyte or virtual byte, is equal to 4 weight units. Each weight unit represents a $1/4,000,000th$ of the maximum block size. The amount of vbytes a transaction needs in a block depends on the type (SegWit or legacy) and the actual size of the transaction [bit18c].

Miners want to maximize the total amount of transaction fees they can earn by creating one block. Because the size of a block is limited, they try to include the transactions with the highest fee per size unit (vbyte), called fee rate. By increasing the fee rates, the probability that the transaction is included in a more recent block is also increased. Many Bitcoin clients such as Bitcoin Core support some means for dynamic fee rate estimates for a given amount of blocks that the sender can bear to wait, longer wait time means a lower fee rate [EOB19; bit19e]. There are also online services that analyse and estimate fee rates over time, such as *Feesim*[4].

[4] `https://bitcoinfees.github.io/`

2.2.7 Blockchain Networks

Experimenting and testing with the Bitcoin blockchain is often needed, but when doing so using the Bitcoin mainnet, i.e., the live Bitcoin network, real Bitcoins are needed and therefore interactions are limited and expensive. One way to avoid testing applications using the mainnet is to use the *Bitcoin testnet* [bit19h], which is essentially a global playground to experiment with the Bitcoin protocol and its capabilities [TS16].

The testnet uses a distinct blockchain that is independent from the mainnet. Testnet coins are supposed to never have any value and can be obtained from so-called *faucets*[5] for free, although often the amount of coins that can be received in a certain amount of time is limited. The Bitcoin testnet blockchain mimics the mainnet and runs the same code as the peers on the mainnet, apart from slight changes in its parameters. These changes include different port numbers and other address prefixes. The most important change is done to the block creation difficulty, which resets back to the minimum value for one block, if no block has been found in *20 minutes* [TS16; bit19h]. Additionally, in testnet some restrictions are relaxed, such as standard transaction checks, meaning that all transactions, standard or not, are treated equally [bit19i].

Another way of experimenting is the use of local testing environments, meaning that a private blockchain is run locally, where there is no interaction with random peers and blocks. *Bitcoin Core* has a built-in *regression test mode (regtest mode)* that can be used as a local testing environment. It provides more control over the blockchain, blocks can be generated instantly and coins, of course without having any value, can be created [TS16]. Apart from these differences, it follows the same basic rules as the testnet [bit19i]. It is also possible to run multiple independent regtest mode instances and therefore blockchains in parallel on the same machine[6].

2.3 Ethereum

Ethereum [Woo14] is a second generation blockchain that utilizes the concept of quasi Turing-complete smart contracts, which are programs that live on the blockchain (see Section 2.1.5). These programs can be invoked by users or other contracts by sending transactions to the blockchain. The effects of those invocations are then validated by the network [BP17].

2.3.1 Cryptographic Primitives

In Ethereum, the *Keccak-256* hash function is used throughout its whole design. Instead of the final SHA-3 specification, the version 3 of the winning entry to the SHA-3 contest by Bertoni et al. [Ber+11] is utilized [Woo14].

[5] e.g. `https://coinfaucet.eu/en/btc-testnet/`
[6] `https://bitcoin.stackexchange.com/a/39168`

Just as Bitcoin, Ethereum uses the ECDSA [JMV01] as its digital signature scheme together with the standardized *secp256k1* [Hes00] elliptic curve. The cryptographic signatures that are produced by this signature scheme do not use DER encoding as in Bitcoin, which only includes the r and s value of the signature [Bro09]. In Ethereum, the signatures include the three values v, r and s of the signature, allowing the recovery of the public key that was used to create the signature [Woo14; AW18].

2.3.2 Accounts

In contrast to Bitcoin, balances or coins in Ethereum are account-based instead of the usage of UTXOs. Ethereum has the concept of two different account types, externally owned accounts (EOAs) controlled by their corresponding private keys and contract accounts that have smart contract code. Contract accounts do not have a private key. They are solely owned and controlled by their own logic within the contract on the blockchain [AW18].

Each account is comprised of the following data [AW18]:

Ether balance. Represents the number of ether (native coins on Ethereum) the account owns.

Nonce. Indicates the number of successfully sent transactions for EOAs or the number of created contracts for contract accounts.

Accounts storage. A permanent data store, used by contract accounts to persist their state. Is always empty for EOAs.

Program code. Includes the byte code of a smart contract if the account is a contract account. For an EOA, it is always empty.

Both account types have addresses that are used for interactions. The address format of Ethereum accounts is designed in a simple way. Ethereum addresses are represented as hexadecimal numbers. They are derived from the last 20 bytes of the Keccak-256 hash of the corresponding public key [AW18].

2.3.3 Ethereum Smart Contracts

A smart contract in Ethereum is essentially a program that lives on the blockchain. This is achieved by deploying its byte code. The smart contract code is executed in its own execution environment called EVM. A deployed contract has its own ether balance and other users or contracts can make procedure calls through the Application Binary Interface (ABI) that the contract exposes. An essential feature is the ability of a contract to send and receive ether [Nar+16].

Contracts are usually written in a high-level language that is easier to write and read than the byte code. The code of the high-level language is then compiled into EVM byte code before being deployed. There exist several compatible high-level languages. The most established and important one is *Solidity*. Solidity is a procedural programming language that utilizes a syntax that is very similar to JavaScript's syntax [AW18].

In contrast to Bitcoin Script, Ethereum smart contracts support the concept of loops, meaning that the execution of a contract is not bound by time or space. Furthermore, it is generally undecidable if a contract will run forever, known as the Halting Problem. To limit the execution of contracts, Ethereum uses a mechanism called *gas*. Each instruction that is executed costs a small amount of ether (called gas in this context). The cost depend on the given instruction. For instance, the computation of a hash value or persisting data costs more than primitive instructions such as basic arithmetics and logic. Each contract call must specify the maximum amount of gas that can be spent, called *gas limit* [Nar+16].

2.3.4 Ethereum Blockchain

Ethereum uses the same consensus model as Bitcoin: PoW is used to create blocks and determine the longest chain and therefore the current state of the Ethereum state machine. There are plans for the near future to change to the PoS voting system. As a result of the account-based balances and other state data on the blockchain, Ethereum uses a serialized hashed data structure called *Merkle Patricia tree* to store the system state, which is a specialized form of a Merkle tree [AW18].

Just as in Bitcoin, there is also a difficulty for block creation that is adjusted regularly to meet the aimed block creation time of *15 seconds*. This time is much shorter than in Bitcoin and allows for faster transaction confirmations [Dan17].

Blocks in Ethereum contain similar data as Bitcoin blocks, such as the previous block hash, transactions, difficulty, a nonce and a block timestamp. As a result of the state-based approach, smart contract support and the shorter block creation time, Ethereum includes additional data fields into each block, such as the hash of its uncle block, state changes and gas limit and usage [Woo14]. Ethereum has stricter rules for valid blocks regarding the block timestamp. In addition to other rules, for a block to be valid, the timestamp has to be greater than the one on the previous block and less than two hours into the future. This means that the time of a block must always be strictly greater than the time of its preceding block, which is not the case in Bitcoin [But14].

2.4 DeXTT: Deterministic Cross-Blockchain Token Transfers

Within this section, the DeXTT protocol [Bor+19b] is presented in a detailed manner, as the main aim of this thesis is to extend the DeXTT protocol by implementing the protocol on another suitable blockchain technology.

The DeXTT protocol can be used to record the transfer of tokens on multiple blockchains simultaneously, while being completely decentralized. It enables blockchain interoperability in the sense of cross-blockchain asset transfers, where tokens are not locked within an individual blockchain. The tokens that can be transferred through the use of the protocol are not native currencies (such as Bitcoin on the Bitcoin blockchain), but are a token type that can exist on a given number of blockchains simultaneously. They can be traded and are synchronized across all participating blockchains. In theory, DeXTT supports the use of any number of blockchains and autonomously synchronizes its transactions across them in a fully decentralized manner. The DeXTT protocol has its own means to prevent double spending and to deal with the *cross-blockchain proof problem*. More details about the protocol are described in the following sections [Bor+19b].

2.4.1 Claim-First Transactions

To enable interaction between blockchains, as in asset transfers, consistency between the blockchains is required. This consistency implies that the presence of data on one blockchain must be a reliable indication that on another blockchain, related data is contained. In the scenario of token transfers, in an approach where strict consistency between blockchains would be supported, a token transfer on one blockchain could be undoubtedly detected on another blockchain, enabling easy synchronization. In practice, strict consistency is not possible between blockchains, meaning it is not possible to verify on one blockchain whether specific data has been recorded on another blockchain. This fact is called the *cross-blockchain proof problem* and poses a challenge for cross-blockchain collaborations [Bor+18b; Bor+19b].

Because strict consistency between blockchains is not possible in practice, DeXTT only requires eventual consistency for synchronizing data, accepting temporary disagreements between the blockchains. This is a feasible approach, because blockchains themselves only provide eventual consistency. The DeXTT protocol proposes to achieve eventual consistency by using *claim-first transactions*. In contrast to traditional blockchain transfers, where tokens can only be claimed after they have been marked as spent, *claim-first transactions* allow the reversal of temporal order. This means that tokens can be claimed before they have been spent, in the case of DeXTT for a certain period of time, namely until the information has been propagated to the other blockchains. Within this time window, tokens can exist on both the balance of the sender and receiver of a transfer [Bor+19b].

Eventual spending of the tokens of the sender is enforced within DeXTT. To ensure eventual consistency, the protocol relies on parties that observe transfers and propagate them to other blockchains. These observers are given a monetary incentive called *witness reward*, to ensure propagation. This is done through the *witness contest*, where one observer is selected deterministically per transfer to receive a witness reward. For this reward, part of the transferred tokens are used, therefore the transaction sender pays for it [Bor+19b].

2.4.2 Initiation of a Token Transfer

A wallet $\mathcal{W}$ in DeXTT consists of a pair of corresponding keys, a public/private key pair. For a token transfer from the sending wallet $\mathcal{W}_s$ (source) to the receiving wallet $\mathcal{W}_d$ (destination), $\mathcal{W}_s$ signs the intent of the transfer to the destination using its private key, confirming that the given amount of tokens are to be transferred to $\mathcal{W}_d$. A validity period for the transfer is additionally defined, denoting the time period for the witness contest. The sender's intent is shown in (2.1), where x denotes the amount of tokens to be transferred, including the *witness reward*, $[t_0, t_1]$ expresses the validity period and α stands for the signature of the entire content inside the brackets [Bor+19b].

$$\left[\mathcal{W}_s \xrightarrow{x} \mathcal{W}_d,\, t_0,\, t_1\right]_\alpha \tag{2.1}$$

All this data of the sender's intent is transferred to the receiving wallet. The transfer itself can happen either on any of the participating blockchains, or using an off-chain channel. The transfer does not need to be secure, because the data is meant to be published later anyhow. Wallet $\mathcal{W}_d$ countersigns the data of the sender's intent after receiving it using its private key. The result is the entire Proof of Intent (PoI) that is shown in (2.2), where β is the newly created signature.

$$\left[\mathcal{W}_s \xrightarrow{x} \mathcal{W}_d,\, t_0,\, t_1,\, \alpha\right]_\beta \tag{2.2}$$

The PoI includes all data necessary to prove that the transfer is authorized by both the sender and accepted by the receiver. The receiver of the transfer can post this PoI on any blockchain within the DeXTT ecosystem by using a transaction called *CLAIM*. This transaction allows the receiver to publish the PoI in order to later claim it. It does not need to be posted on more than one blockchain. The purpose of this transaction is to publish the PoI to enable its propagation across all participating blockchains through the later described *witnesses*. The *CLAIM* transaction is shown in (2.3). Wallet $\mathcal{W}_d$ on the left hand side denotes the wallet that posts and signs the transaction, $CLAIM$ indicates the transaction type [Bor+19b].

$$\mathcal{W}_d : CLAIM \left[\mathcal{W}_s \xrightarrow{x} \mathcal{W}_d,\, t_0,\, t_1,\, \alpha\right]_\beta \tag{2.3}$$

There are four preconditions for the *CLAIM* transaction [Bor+19b]:

1. The PoI needs to be valid, meaning that both signatures α and β are correct.
2. The balance of the source wallet must be sufficient for the amount to be transferred.
3. The PoI must not be expired, meaning that the time t_1 has not been reached yet.

4. No other PoI with overlapping validity period and the same source wallet $\mathcal{W}_s$ may be known on the blockchain on which the new PoI is posted. This means a wallet must not sign an outgoing PoI while still having another pending outgoing PoI. This precondition is put in place to prevent the double spending of tokens.

2.4.3 The Witness Contest

After the *CLAIM* transaction has been posted on a blockchain, the information about the transfer is only available on that blockchain. To ensure consistency between blockchains, the intent about this transfer must be propagated to all participating blockchains. This propagation is achieved by a mechanism called *witness contest*.

Any party that observes the *CLAIM* transaction can become a contestant in this contest by propagating the PoI to all blockchains in the ecosystem and therefore becoming a candidate for receiving the reward. The transaction that is used for this propagation is called *CONTEST* and is shown in (2.4). The specified wallet $\mathcal{W}_o$ stands for an arbitrary wallet that signs the message of the transaction, resulting in the signature ω [Bor+19b].

$$\mathcal{W}_o : CONTEST\left[\mathcal{W}_s \xrightarrow{x} \mathcal{W}_d,\, t_0,\, t_1,\, \alpha,\, \beta\right]_{\omega} \tag{2.4}$$

The PoI must be valid and is not allowed to violate any PoI's validity period. The *CONTEST* transaction can be posted multiple times by various contestants. The transaction is eventually posted to all blockchains by every participating party, as contestants are interested in participating on all blockchains in order to maintain consistency of their own balances [Bor+19b].

2.4.4 Selection of the Winning Witness

Once the witness contest ends after the expiration of t_1, a winning witness of the contest has to be selected and be therefore rewarded with the *witness reward*. This is done by the *FINALIZE* transaction, which is purely time-based and must be triggered after t_1. It can be posted by any party, but in the current approach it is assumed that the destination wallet $\mathcal{W}_d$ is posting the *FINALIZE* transaction to every blockchain. The transaction is shown in (2.5). It only requires the parameter α that identifies the corresponding PoI.

$$FINALIZE\left[\alpha\right] \tag{2.5}$$

The *FINALIZE* transaction concludes the contest of the referred PoI and the winning witness $\mathcal{W}_w$ gets awarded the reward, currently being defined as *1 token*. Additionally, the actual transfer of tokens is performed, increasing the balance of $\mathcal{W}_d$ by $(x-1)$ tokens and decreasing the balance of $\mathcal{W}_s$ by x tokens. Because *FINALIZE* is posted on all blockchains, these actions are also performed on each of them [Bor+19b].

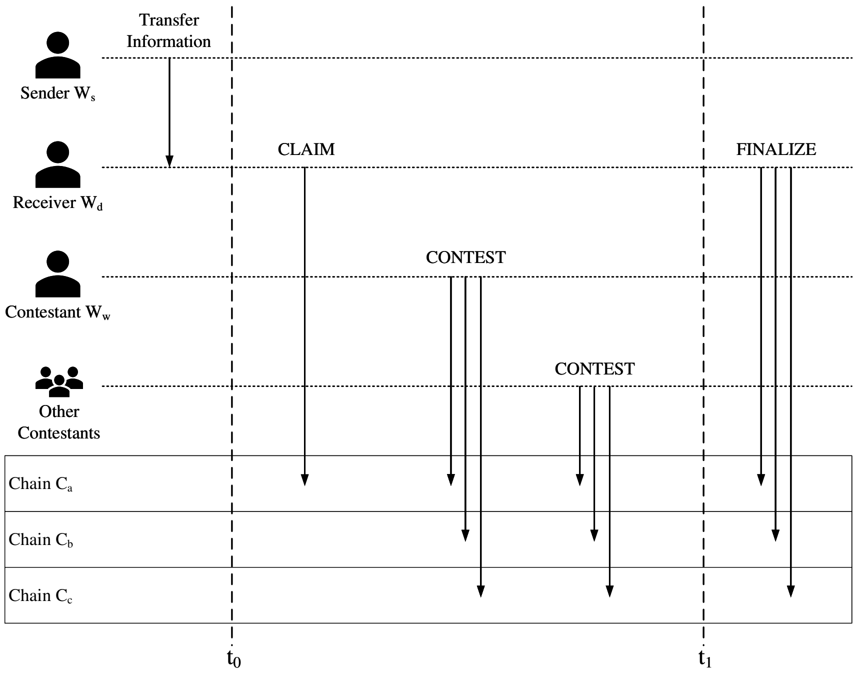

Figure 2.2: Sequence of transactions of a DeXTT Transfer [Bor+19b].

The winner of the contest is deterministically assigned, being the contestant with the lowest signature ω, meaning the one with a value closest to zero. Since the signature is only formed from the data of the PoI and the contestant's private key, there is no way in increasing the chance of winning, except for creating a large amount of wallets, which is computationally expensive and therefore does not violate the protocol [Bor+19b].

In Figure 2.2, an overview of the different transactions posted by each party on various blockchains for a DeXTT token transfer are illustrated. The winner of the contest is shown separately as $\mathcal{W}_w$. The sender $\mathcal{W}_s$ must first provide the destination of the transfer $\mathcal{W}_d$ with the sender's intent, then after t_0 has expired, $\mathcal{W}_d$ posts a *CLAIM* transaction to one blockchain. All contestants post *CONTEST* transactions to all blockchain after observing the *CLAIM*. This is done in no particular order, but before t_0. Finally, after t_1 is expired, $\mathcal{W}_d$ posts the *FINALIZE* transaction to all blockchains to conclude the transfer [Bor+19b].

2.4.5 Double Spending Prevention

An attempt of double spending tokens by a malicious party is executed by signing two different PoIs that are conflicting with each other. Executing both transfers could result in a negative balance for the malicious sender. Such double spending attempts are prevented by the *VETO* transaction, which can be used by any party that notices two conflicting PoIs with the same source wallet and overlapping validity period. The same incentive technique as for the witnesses, a contest, is used for this transaction. Any party can report the conflicting PoIs by posting the *VETO* transaction, as shown in (2.6), to all blockchains in the ecosystem. The participant with the lowest signature ω is assigned a reward after the veto validity period. The original PoI is referred to by α, which is already known on the blockchain. The remaining data describes the new conflicting PoI [Bor+19b].

$$\mathcal{W}_w : CONTEST \left[\alpha,\, \mathcal{W}_s \xrightarrow{x'} \mathcal{W}_{d'},\, t'_0,\, t'_1,\, \alpha'\right]_\omega \tag{2.6}$$

Because on different blockchains, the order of PoIs might be different, the VETO transactions do not have to be the same on each blockchain, but the following preconditions must hold: α must refer to a PoI already known on the blockchain and the PoIs must actually be conflicting with each other. The *VETO* then has the following effects [Bor+19b]:

1. The balance of the sender is set to zero to penalize the creation of conflicting PoIs.
2. Every PoI of the sender that is not yet expired is canceled and therefore no *FINALIZE* transaction for any of them is permitted anymore, meaning that the transfers are not executed.
3. A new contest, called *VETO* contest, is started.

The *VETO* contest is very similar to the regular witness contest. Its purpose is the propagation of conflicting PoIs. Currently, the same reward of 1 token is awarded to its winner, the validity period of the contest is defined as shown in (2.7) [Bor+19b].

$$t_{VETO} = max\,(t_1,\, t'_1) + max\,(t_1 - t_0,\, t'_1 - t'_0) \tag{2.7}$$

This makes the *VETO* contest valid until t_{VETO}, which is defined by adding the later expiration time of the conflicting PoIs to the longer validity period of them, ensuring that there is sufficient time for the contest. Similar to the regular witness contest, the *VETO* contest is concluded by the *FINALIZE-VETO* transaction as defined in (2.8) [Bor+19b].

$$FINALIZE - VETO \left[\alpha,\, \alpha'\right] \tag{2.8}$$

The effect of the *FINALIZE-VETO* transaction is essentially the same as for the *FINALIZE* transaction, assigning the reward to the winning veto contestant, i.e., the participating wallet with the lowest signature ω. The *FINALIZE-VETO* transaction can potentially be posted by anyone. The winner of the contest has again monetary incentive to do so [Bor+19b].

2.4.6 Ethereum Prototype

The change of state of blockchains through the transactions of the DeXTT protocol can be implemented by using smart contracts, such as they are provided by the EVM on the Ethereum blockchain. Borkowski et al. [Bor+19b] implemented a reference implementation of DeXTT for evaluation purposes using the Solidity language for Ethereum. The prototype is freely available as Open Source software on Github[7]. They evaluated the protocol in regards to its functionality, performance and cost on a blockchain ecosystem by performing repeated token transfers using their prototype implementation [Bor+19b].

For the evaluation, the implemented Solidity smart contracts were deployed on three private Ethereum-based blockchains using *geth* nodes. A testing client software was used to perform the transfers. The block times of the private blockchains were configured to mimic the value of the real Ethereum blockchain as of January 2019, being 13s on average. There were two experiment series, each one using ten clients that constantly and simultaneously initiate transfers within the given blockchain ecosystem. The first series of this experiment was used to evaluate the impact of the transfer validity period, the second series was used to measure the average cost of a DeXTT transfer [Bor+19b].

The results of the experiment series of the DeXTT prototype implementation on the private Ethereum blockchains show that it is sufficient, in order to avoid corrupted transfers, to use a validity period of at least 4 blocks (52 seconds) when using the reference implementation. Corrupted transfers are transfers that lead to permanently inconsistent balances due to insufficient time to post all required DeXTT transactions to all blockchain in the ecosystem. Eventual consistency is not guaranteed anymore if corrupted transfers occur. Additionally, the experiment runs yield an upper bound for DeXTT transfer cost on Ethereum, which depends on the number of participating blockchains. For 10 blockchains, the total transaction cost for the receiver of a transfer is 0.59 USD. Each observer posting contest transactions pays 0.94 USD, as of January 2019. Potential optimizations in the smart contract code could further reduce this upper bound for the cost [Bor+19b].

[7]`https://github.com/pantos-io/dextt-prototype`

CHAPTER 3

State of the Art

In this chapter, the state of the art of different blockchain interoperability techniques are presented and discussed. The most relevant contribution in the context of this thesis is the DeXTT protocol [Bor+19b] proposed by Borkowski et al., which is presented in detail in Section 2.4. The presented techniques are therefore discussed in regard of the DeXTT protocol, highlighting the differences in the approaches.

First, in Section 3.1 current approaches for exchanging and swapping assets across different blockchains are discussed. Section 3.2 presents techniques that implement a more generalized cross-blockchain communication approach, not being bound to the swapping of assets in their functionality. They allow, among others, use cases similar to the token transfer of the DeXTT protocol.

3.1 Exchange of Assets

This section introduces blockchain interoperability approaches that focus on the swapping or exchanging of assets among its users. The assets are therefore not transferred between the different blockchains, only their ownership is transferred amongst the blockchains. The section is concluded by a comparison of the techniques and the DeXTT protocol.

3.1.1 Atomic Cross-Chain Swaps

Atomic cross-chain swaps as proposed by Herlihy [Her18] are distributed coordination tasks, where assets are exchanged between multiple parties across multiple blockchains, i.e., Alice wants to give Bob some assets on Blockchain A in exchange of assets on Blockchain B that belong to Bob. The exchanged assets are traded and not actually moved between blockchains. Such a trade introduces some risks, as the user who gives the assets to the other one first, faces the risk that the other party does not obey the agreement and keeps both assets [Sir+19].

To prevent such risks, atomic swaps involve the publishing of transactions that utilize *hash-locks* and *time-locks* via Hashed Time-Lock Contracts (HTLCs) to ensure the atomicity of the trade [Sir+19]. These contracts ensure that either both transactions of a swap or neither take place, without the need that the involved users trust each other [Sir+19]. Hash-locks are cryptographic locks that can be unlocked by revealing a secret s such that $h = H(s)$, where h is the hash value used in the lock. Time-locks prevent the redeeming of assets from a transaction until the specified time interval has passed [Sir+19]. HTLCs make us of both kind of locks and require that the receiver claims the payment by revealing the according hash before the time-lock expires. The assets that were locked inside the contract can be refunded to the sender after the expiration of its time-lock [bit19d].

Herlihy provides an analysis of atomic cross-chain swaps [Her18] for the cases where multiple parties exchange assets. The presented atomic swap protocol guarantees the following [Her18]:

1. All swaps take place if all parties conform to the protocol.
2. If a party disobeys the protocol, none of the conforming parties ends up worse off, except that their assets are temporarily locked up.
3. There is no incentive for any party to deviate from the protocol.

A cross-chain swap is modeled as a *directed graph* $\mathcal{D} = (V, A)$ in the work of Herlihy. The vertices V of $\mathcal{D}$ represent the involved parties, its arcs A are proposed asset transfers. Herlihy shows that no atomic swap protocol is possible, if graph $\mathcal{D}$ is not strongly connected. Further, it is shown that the protocol has time complexity of $O(diam(\mathcal{D}))$ and a space complexity, as in bits stored on all blockchains, of $O(|A|^2)$ [Her18].

The protocol can introduce unwanted side effects, where coins might be potentially lost. This can happen, if the contracts of the participants are deployed in a wrong order, making it possible for some parties to redeem the coins without deploying their own contracts to lock their assets. Another case, where coins might be lost, happens if the time values of the time-locks expire almost at the same time. One party could then reveal their secret s at the very last moment, leaving no time for the others to redeem their assets before the time expires [Her18].

3.1.2 Atomic Commitment Across Blockchains

A decentralized *all-or-nothing* atomic cross-chain commitment protocol [ZAA19] was introduced by Zakhary, Agrawal, and Abbadi as an improvement on existing atomic cross-chain swapping protocols. They model the redeeming and refunding in the smart contracts that exchange the assets as conflicting events. Additionally, a permissionless network of witnesses is utilized, guaranteeing that such conflicting events can never occur simultaneously. The witnesses also ensure that all smart contracts in an atomic cross-chain transaction are either redeemed or refunded, therefore ensuring the property named *all-or-nothing* [ZAA19].

The protocol overcomes the drawbacks by current atomic cross-chain transfer protocols described in Section 3.1.1. One drawback is that such swaps do not guarantee the all-or-nothing atomicity principle, making it possible that only one party can redeem their assets, if a time-lock expired due to a crash failure or network delay of an honest participant. Another drawback of previous approaches is the requirement of sequentially publishing the smart contracts to ensure that no malicious participant can take advantage of the protocol by declining the publication of an according contract [ZAA19].

The all-or-nothing atomic cross-chain commitment protocol overcomes these drawbacks through the utilization of an open *witness network*. The protocol allows all participants to publish their smart contracts for the asset exchange concurrently, resulting in a drastic decrease in latency of the swap. The witness network is represented as a permissionless blockchain, which decides whether a transaction should be committed or aborted. The miners of this blockchain together represent the witnesses on the cross-chain transfers. It is recommended that the witness blockchain network is chosen from the set of involved blockchains. The proposed protocol reduced the latency of a transaction from $O(diam(\mathcal{D}))$ to $O(1)$ as it allows parallel execution of sub-transactions, decoupling the latency from the size of the transaction graph $\mathcal{D}$ [ZAA19].

3.1.3 Republic Protocol

The *Republic protocol* [ZW17] proposed by Zhang and Wang represents a decentralized dark pool exchange that provides atomic swaps. A dark pool refers to the concept that the trades of its users are placed on a hidden order book, meaning that the details about asset exchanges are kept secret. The trades are then automatically matched through a computation protocol. The Republic protocol enables trades through atomic swaps. The trade orders are matched through a decentralized network of nodes and can not be reconstructed in a feasible way. The order matching nodes are organized by an Ethereum smart contract called *Registrar* into a network topology that makes it infeasible for an adversary to reconstruct a trade order [ZW17].

3.1.4 Discussion

All three presented approaches for blockchain interoperability enable its users to swap and exchange coins across blockchains. The *atomic cross-chain swap* protocol by Herlihy [Her18] performs the exchange in an atomic way, but it is still possible to violate the *all-or-nothing* property. In contrast, the *atomic cross-chain commitment protocol* by Zakhary, Agrawal, and Abbadi [ZAA19] focuses on the overcoming of this drawback and satisfies the *all-or-nothing* property while also reducing the latency of the asset exchange. These advances come with a more complex protocol architecture, where a witness network is used. The *Republic protocol* by Zhang and Wang [ZW17] has laid its focus on a completely different target, namely keeping the details about the exchanges of users secret.

Compared to the *DeXTT protocol* by Borkowski et al. [Bor+19b], all three techniques for blockchain interoperability can be distinguished from DeXTT in the same way. All of them only enable the swapping or exchanging of assets, where only the owner of assets changes, not the blockchain where they reside. In contrast, asset transfers in DeXTT are synchronized and executed across all participating blockchains. The assets therefore exist on all blockchains of the ecosystem.

3.2 Cross-Blockchain Transfers

Within this section, current blockchain interoperability approaches that are more advanced and universally applicable than simple exchange protocols are presented. They allow not only a change of ownership of assets across blockchains, but also the transfer of actual coins from one blockchain to another. After presenting different interoperability techniques, this section is concluded by a discussion, where the different approaches and the DeXTT protocol are compared.

3.2.1 Leveraging Blockchain Relays for Cross-Chain Token Transfers

The DeXTT protocol emerged from the Token Atomic Swap Technology (TAST)[1] research project, which also led to the publication of additional white papers [Bor+18a; Bor+18b; BRS18; Bor+19a; Fra+19; Sig+19a; Sig+19b; Fra+20]. The most recently pursued approach of the research project by Sigwart et al. aims to eliminate the drawbacks of the DeXTT protocol. The drawbacks to be eliminated include the high synchronization cost for balance changes, no means of adding a new blockchain later on and not being able to hold different amounts of tokens on different blockchains [Fra+19].

Within the approach currently being researched, a token transfer only incorporates the blockchains that are directly involved in a cross-blockchain token transfer. For a transfer from blockchain A to blockchain B, only these two blockchains need to communicate with each other to finalize the transaction [Sig+19a]. This is achieved by burning the tokens that are meant to be transferred on the source blockchain A and recreating the same amount on the destination blockchain B. For this, blockchain B needs a way to verify that the tokens have actually been burned. The authors use the concept of blockchain relays to achieve this verification, where the state of a source blockchain is replicated on the destination blockchain. This replication happens in a decentralized way without the need for a trusted third party [Fra+20].

To verify the existence of certain pieces of state on the source blockchain, Simplified Payment Verification (SPV) is used to proof that a certain transaction is part of the source blockchain. For this verification, the destination chain performs a *light validation*

[1] https://dsg.tuwien.ac.at/projects/tast/

using only block headers. The relaying of these block headers is done through the *relayers*, which regularly submit block headers of the source blockchain to the destination blockchain. Clients dispute any submitted illegal block to the destination chain [Sig+19a]. The submission of this information creates a cost for the relayers and clients that dispute blocks, therefore an incentive structure is put in place to compensate these cost [Sig+19a; Sig+19b]. The potential reward through this incentive structure motivates relayers to submit block headers and clients to dispute illegal block headers [Sig+19a].

A first prototyp implementation of this protocol for Ethereum-based blockchains is available on Github[2]. The concepts of the protocol are not only applicable to cross-blockchain token transfers, but could potentially be utilized as a basis for arbitrary cross-blockchain applications [Sig+19a].

3.2.2 XCLAIM: Trustless, Interoperable Cryptocurrency-Backed Assets

An alternative to atomic cross-chain swaps are so-called *cryptocurrency-backed assets*. This approach allows the locking of an asset x on blockchain B_x while unlocking a corresponding representation $y(x)$ of the locked asset on blockchain B_y. Asset y can then be used natively on B_y and can be redeemed on B_x again in the same way later on [Zam+19].

XCLAIM [Zam+18] is a generic framework that achieves trustless and efficient cross-chain exchanges through the use of cryptocurrency-backed assets presented by Zamyatin et al. They introduce protocols to issue, transfer, swap and redeem such assets in a secure and non-interactive manner throughout existing blockchains. *XCLAIM* utilizes publicly verifiable smart contracts together with *chain relays* to achieve cross-chain state verification. Chain relays are used to provide external blockchain data from the blockchain B, where assets are locked, to the blockchain I that issues the backed assets. These relays are used to make the issuing of assets non-interactive. They are utilized in a smart contract component on the issuing blockchain I and interpret the state of B using functionality comparable to a SPV client, only using block headers of blocks from B [Zam+18].

XCLAIM uses these techniques to construct a publicly verifiable audit log of user actions on both participating blockchains. It enforces the correct behavior of its participants through the utilization of collateralization and punishment through a *proof-or-punishment* approach. This means that its users must proactively prove their adherence to the protocol [Zam+18].

3.2.3 Metronome

Metronome [Met18] is a blockchain interoperability approach that claims to enable cross-blockchain transfers of Metronome tokens (MET), which is an additionally in-

[2] https://github.com/pantos-io/go-testimonium

troduced cross-chain currency. To transfer MET tokens from one blockchain A to another blockchain B, the user first has to destroy and therefore remove the tokens on blockchain A, receiving a *proof of exit* receipt. The destroyed tokens can then be claimed on blockchain B by presenting this receipt to the according Metronome smart contract, which then yields the correct amount of tokens on the target blockchain [Met18; JRB19].

The Metronome user manual only provides a high level description of the protocol, describing the following components [Met18]:

Exporting. When a user calls the export function, it takes the provided MET of the call and burns them on the local chain and issues an *ExportReceipt*. The user which burns the tokens will pay a small fee to be claimed by the validator.

Importing. Any user of the protocol can provide an *ExportReceipt* to the Metronome smart contract on the destination blockchain. After the import is processed, the tokens are delivered to the recipient. The party that completes the Metronome import will be rewarded with a transaction fee.

Validation. For the export and import functionality, validators are required. Among others, they are needed to provide additional information for validation of particular imports. They play a role in the Metronome ecosystem similar to the concept of *miners*, verifying and authenticating Metronome's distributed source of truth.

As of April 2020, Metronome is already deployed on *Ethereum* [Woo14] and *Ethereum Classic*[3], the support for more blockchains is currently under research and development[4].

3.2.4 Polkadot

The *Polkadot* project [Woo16] proposed by Wood represents a more generic multi-blockchain framework. Its functionality is aimed to provide a framework for blockchain interoperability through the use of a central relay blockchain for its maintenance [Woo16; Sch+19]. It allows cross-blockchain communication between heterogeneous blockchains, called *parachains*. The architecture additionally consists of a central *relay blockchain* that manages transaction consensus and delivery [QAN19]. To include existing blockchains that can not be included as a parachain directly, as they do not provide the according interface, a so-called *bridge blockchain* has to be used for each of the existing blockchains to integrate them [Sch+19]. There are four basic roles involved in the upkeep of the Polkadot network [Woo16]:

Validators. A validator helps to seal new blocks on the network, its role is contingent to a sufficient bond of tokens or by being nominated for the role. The validators of the network run a relay blockchain client implementation. They must be ready at each

[3] `https://ethereumclassic.org/`
[4] `https://www.metronome.io/roadmap/`

block to be nominated to ratify a new block on a parachain, including receiving, validating and republishing candidate blocks. A validator gives the taks of creating new parachain blocks to another party called *collator*. After a new parachain block has been created, they must ratify the new relay blockchain block themselves.

Nominators. They are simply a stake-holding party contributing to the security bond of validators. They place risk capital to signal their trust on a particular validator.

Collators. Collators are used to assist the validators by producing parachain blocks. Each collator maintains a *full node* for a particular parachain, enabling them to have all available information to create new valid blocks and execute transactions.

Fishermen. The task of fishermen is to provide a proof that at least one bonded party acted illegally, for which they get a reward. Such illegal actions include the signing of two blocks with the same ratified parent block or in the case of parachains, helping to ratify an invalid block.

3.2.5 Blockchain Router: A Cross-Chain Communication Protocol

Wang, Cen, and Li introduce *blockchain router* [WCL17], a network that empowers blockchains to connect and communicate with each other. In this network, one blockchain plays the role of the blockchain router, which has the task to analyze and transmit communication requests. It also dynamically maintains a topology structure of the network. The architecture of the protocol is inspired by how routing is done in the Internet. A simple routing network consists of routers and terminal devices. In this proposed protocol, the blockchains that want to communicate with each other correspond to the terminal equipment, called *sub-chains*. Sub-chains can send messages to other sub-chains via a chain router and receive messages from the chain router, but they can not communicate directly with each other. Sub-chains are connected to a chain router by following the cross-chain communication protocol [WCL17; JRB19]. The blockchain router protocol distinguishes between four different types of participants [WCL17]:

Validators. The tasks of the validators consist of the verification, concatenation and forwarding of blocks to the correct destination. They must run a *full client* of the blockchain router and collect and ratify blocks from the sub-chains.

Nominators. A nominator contributes its own funds to validators. In doing so, nominators obtain a corresponding payoff or punishment, based on how the funded validators perform.

Surveillants. The surveillants are monitoring the blockchain router's behavior, reducing the incidents of malicious behavior.

Connectors. The task of a connector is to link blockchain routers with the sub-chains. They are responsible for sending information between sub-chains and routers and form a consensus system for each sub-chain. A connector maintains a *full node* for its sub-chain and can therefore execute transactions on the sub-chain. The connectors provide data about executed transactions to the validators.

3.2.6 HyperService

The HyperService platform [Liu+19] proposed by Liu et al. aims to extend the functionality of blockchain interoperability from asset transfer to distributed computation [Zam+19]. It allows the building and execution of Decentralized Applications (DApps) across heterogeneous blockchains via a virtualization layer on top of the blockchains. The design of HyperService consists of a developer-facing programming framework, which enables the building of cross-blockchain applications in a unified programming model. Additionally, a secure blockchain-facing cryptography protocol is used to realize those applications on blockchains [Liu+19]. The architecture of HyperService consists of four components [Liu+19]:

DApp Clients. The clients are the gateway for DApps to interact with the HyperService platform.

Verifiable Execution Systems. These systems can be seen as *blockchain drivers.* They compile high-level DApps programs, which are the runtime executables in HyperService, into transactions that are executable on the individual blockchains.

Network Status Blockchain. This part can be seen as the blockchain of blockchains, providing a view onto the execution status of the DApps.

Insurance Smart Contracts. The correctness and violations of DApp executions are arbitrated in a trust-free manner by these contracts. If an exception occurs, they financially revert all executed transactions.

3.2.7 Distributed Cross-Blockchain Transactions

Zhao and Li propose two distributed commit protocols [ZL20] that are still in an early research state. Their approaches enable nonblocking distributed commits for multi-party cross-blockchain transactions. They mainly focus on the protocol specification and assume that the participating blockchains either have an effective programmable way to communicate via smart contracts or that a proxy to enable this communication is available [ZL20]. The two protocols are designed as follows [ZL20]:

Synchronous Cross-Blockchain Transactions Protocol This protocol is very similar to the *two-phase commit protocol* and designed to strictly enforce the *Atomicity, Consistency, Isolation and Durability (ACID)* properties of cross-chain transactions, resulting in a higher latency. It delays the global commit until none of the participating blockchains can unilaterally rollback the transaction. A specific blockchain, called *coordinator*, initiates a transaction by sending the *precommit* message to all blockchains and then waiting for their *ready* replies. In the second phase of the protocol, the initiating blockchain broadcasts a *commit* message. Each blockchain then carries out the local action and waits for a specified amount of time before returning the required *done* message. The wait time makes sure that the local transaction has enough confirmations.

Redo-Log-Based Blockchain Protocol This approach follows the spirit of *redo-logs*. The key change to the *Synchronous Cross-Blockchain Transactions Protocol* is the omission of the wait time before the *done* message, replying instantly. If a transaction is included in a blockchain branch that is later cut off, the protocol takes according action and returns the transaction back to the request pool. This results in eventual consistency.

3.2.8 Discussion

All of the approaches presented withing this section feature more universal and capable blockchain interoperability techniques than the protocols for assets swaps and exchanges shown in Section 3.1. The *blockchain relay* approach of Sigwart et al. [Sig+19a], *XCLAIM* by Zamyatin et al. [Zam+18] and Metronome [Met18] implement similar approaches, where tokens in a cross-blockchain asset transfer are first destroyed or locked on the source blockchain and then re-issued on the target blockchain after their destruction has been proofed. All three approaches rely on some sort of relay mechanism for the proof of the destruction on the source blockchain. In comparison to the DeXTT protocol, they all have in common that token transfers happen across blockchains. The difference lies in the synchronization that is used in DeXTT, which achieves that all tokens and transfers are recorded on all participating blockchains, whereas in the three relaying approaches, tokens are transfered between two blockchains. This removes the need of synchronization, which comes with a high cost, but also requires the use of some sort of relay for its proofs.

The approaches of *Polkadot* by Wood [Woo16] and the *blockchain router* by Wang, Cen, and Li [WCL17] use another way to achieve cross-blockchain communication. Instead of relays, they utilize a central blockchain that connects and communicates with the participating blockchains. Both are explicitly designed to enable general cross-blockchain communication instead of just token transfers, as it is the case for the DeXTT protocol.

HyperService by Liu et al. [Liu+19] takes blockchain interoperability to another level by providing a platform for distributed computation, whereas the DeXTT protocol only supports transferring tokens. HyperService can of course also be used to transfer tokens across blockchains by implementing this functionality as a DApp, although this comes with the overhead of running the whole platform.

The *distributed cross-blockchain transaction approach* of Zhao and Li [ZL20] focuses only on the protocol specifications and not on the cross-blockchain communication. They also do not describe what token transfers or general transactions using their protocols look like. The nature of their proposed protocols implies that the blockchains are synchronized as it is the case for the DeXTT protocol, although this does not necessarily mean that tokens exist on all blockchains simultaneously as in DeXTT, as the synchronized transactions can contain arbitrary data.

CHAPTER 4

Requirements and Blockchain Selection

Within this chapter, the requirements for the DeXTT protocol (see Section 2.4) are collected and compared to the features of current blockchain technologies to select a suitable and reasonable blockchain technology for the DeXTT protocol implementation.

First, the requirements for an underlying blockchain technology to support the adaption of the DeXTT are analyzed and specified in Section 4.1. This is done with regards to different approaches of where the protocol logic resides, on the blockchain itself or on the DeXTT client-side. In Section 4.2, the properties and features of different currently available blockchain technologies are analyzed concerning their technical aspects. This is done in the form of a survey of preselected blockchains. At last, Section 4.3 describes the selection rationale of the blockchain candidate for the DeXTT protocol implementation.

4.1 DeXTT Requirements

This section defines the formal requirements a blockchain technology must fulfill to support an implementation of the DeXTT protocol for its blockchain. An implementation of the DeXTT protocol can be described as a DApp. Each DApp solution is composed of at least two elements [But14]:

1. The *smart contract* or generally speaking, data and logic residing on the blockchain.
2. The *client-side application*, used to interact with the blockchain.

It is possible to choose different distributions for the logic and data of a DApp among these two components [AW18]. This applies also for a DeXTT implementation.

Having more parts of a DApp on the blockchain comes with the advantage, that is is far easier to create the according client implementation, which is beneficial when there are multiple different client implementations. The client-side is easier to understand and develop and therefore less error-prone. A disadvantage of this approach is that the underlying blockchain technology must potentially support more diverse features.

In contrast, having more parts of a DApp in the client-side application comes with the downside that the clients are harder to understand and implement and therefore more error prone. Additionally, if some parts of the logic are wrongfully implemented differently among multiple client implementations, the clients might not be compatible to each other. An advantage of moving DApp logic to the client-side is the potentially lower requirements of features for the used blockchain technology. There are two edge cases of the distribution among the parts of a DApp:

1. All of the logic and data of a DApp reside on the according blockchain. The client-side application is solely used to manage the wallets and to interact with the blockchain, e.g., calling methods of a smart contract.
2. The blockchain itself contains no logic and is solely used to store some required data for the DApp. All of the logic concerning verifications, transactions and other parts of the DApp are implemented in the client-side application.

The existing Ethereum implementation[1] of DeXTT resembles an implementation close to the first edge case, having all its relevant logic and data reside on the blockchain. Only the creation of DeXTT transactions and the interactions are handled off-chain. The specification of requirements are based on the existing literature of the DeXTT protocol and on the given Ethereum implementation. The main focus will be put on the Ethereum prototype, as this gives a better inside of how DeXTT can be implemented.

Because of the implied variations in the possible implementations of the DeXTT protocol, the requirements analysis in the following sections is split into two parts, resembling the two edge cases of how a DApp can be implemented. The specified requirements must not be seen as proven minimal features a blockchain must provide for DeXTT, but rather as an attempt to approach a small set of requirements that are sufficient to enable a correct implementation of the DeXTT protocol for a blockchain technology.

4.1.1 Blockchain-Side Logic - Smart Contract Solution

This section describes the requirements a blockchain needs to fulfill in order to support a DeXTT implementation, if all of the logic and data resides on the given blockchain as a smart contract (see Section 4.1). This is also done by the already existing Ethereum implementation. The requirements are split into two parts, the ones that apply for the DeXTT protocol in general and additional requirements to also ensure the compatibility with the Ethereum prototype.

[1] `https://github.com/pantos-io/dextt-prototype`

General Requirements

The generally applicable requirements for a blockchain or smart contract platform to support DeXTT (see Section 2.4), without providing guaranteed compatibility with the Ethereum Prototype can be defined as follows:

Signature Verification. Some DeXTT transactions contain a signature that needs to be verified in order to classify it as valid. To support the use of the concept of claim-first transactions, a functionality to verify cryptographic signatures is required. The used signatures are not bound to a specific algorithm, any digital signature algorithm that provides sufficient cryptographic security can be applied [BRS18].

Cryptographic Hash Function. In practice, data is hashed to reduce it to a fixed size before it is being signed. There exist also certain limitations on the size of data to be signed, e.g., for ECDSA [Nar+16]. Therefore, the availability of an according *cryptographic hash function* is required, such that arbitrary data can be verified via the available signature verification scheme.

Timestamp. The DeXTT protocol relies on specific timings of transactions. To verify these transaction times, some means of accessing the execution time of a certain transaction or block on the blockchain are required. This is often achieved by the inclusion of a timestamp in the block header of each block (see Section 2.2.3). This timestamp must be available during the execution of the according smart contract. Without the availability of such timing information, it is not possible to deterministically process and verify DeXTT transactions.

Dynamic Data Structures. A DeXTT implementation relying purely on a smart contract has to *store* all relevant data within the blockchain and *access* and *manipulate* the data via the contract. This is demonstrated in the according smart contract of the DeXTT Ethereum implementation[2]. The stored data includes among others the current balances, all relevant information of ongoing transactions, contests and veto-contests. These types of data are bound in their size to the number of users of the DeXTT protocol, which is not constant and can grow over time. To store such data, some means of dynamically adding new data entries must exist. This can be done by *dynamic data structures* such as *arrays* or *mapping types* in Solidity [Fou19a]. Furthermore, to access and manipulate the dynamically sized data by a given *key* or *value* in the smart contract, there must exist at least one of two options:

1. Providing direct access by the data structure itself through the use of a given access parameter such as a *key*. This is for instance the case for *mapping types* [Fou19a] in Solidity.

[2]`https://github.com/pantos-io/dextt-prototype/blob/master/truffle/contracts/PBT.sol`

2. The existence of some form of loop to iterate over all data entries until the desired one is found.

Control Structures. The smart contract must be able to compare values and make decisions that are needed to follow the DeXTT protocol, e.g. checking signatures and taking different actions according to the outcome of the verification. To provide this functionality, some means of control structures are required. It must be at least possible to introduce conditional branches, as possible through e.g. an *if* statement in Solidity [Fou20f].

Operators. Another requirement is the availability of certain operators. For comparisons, at least one variation of a *smaller than* (“ $<$ ”) or *bigger than* (“ $>$ ”) must exist and be able to compare the contest signatures of DeXTT, certain time constraints and token values. Additionally, there must be some means of equality operator (“ $==$ ”) for verifications and checks to enable conditional branching. Basic arithmetic operators, at least in the form of addition (“ $+$ ”) and subtraction (“ $-$ ”) are also required. This is among others needed for the calculation of the validity period of *veto contests*. All of these operators are for instance available for Solidity [Fou19a] on Ethereum.

Concluding from these requirement specifications, a language for smart contracts on a blockchain must not be Turing-complete [BRS18]. More specifically, no forms of loops are required if certain dynamic data structures such as *mapping types* in Solidity [Fou19a] are available as described above.

Additional Compatibility Requirements

The following specific requirements additionally need to be fulfilled to ensure full compatibility between DeXTT on the given blockchain and the existing Ethereum implementation:

Signature Verification. To be able to verify the same signatures as the ones used with the Ethereum implementation, the ECDSA [JMV01] digital signature scheme using the standardized *secp256k1* elliptic curve [Hes00] must be supported for signature verifications within smart contracts on the blockchain. The need to provide the same signature verification scheme is not required to be able to re-use the same DeXTT transactions with the identical signatures. Instead, it is crucial to support the same signatures, because they are used for the witness selection within the DeXTT protocol. The selection of witnesses can only be deterministic across blockchains that use the same signature scheme, otherwise the signature values would be different for the same transactions, leading to different winners of the witness contests and therefore to inconsistencies.

Cryptographic Hash Function. In the existing DeXTT implementation, different data is hashed using the *Keccak-256* hash function before its being signed. As in

Ethereum itself, not the final SHA-3 specification, the version 3 of the winning entry to the SHA-3 contest by Bertoni et al. [Ber+11] is used. Again, it is crucial to support the same hash function to enable the protocol to select the same winners for witness contests deterministically across the blockchains.

Converting Data Fields to Raw Bytes. Within the Ethereum prototype, data that is hashed and signed is first encoded and packed using the `abi.encodePacked()` function of Solidity [Fou20e]. To verify signatures for deterministic witness selection, it is therefore necessary to support the same encoding functionality for the used data types. In the case of the implementation using Solidity, the used data types are: `bytes`, `address`, `uint256` and `bytes32` [Fou19a]. The encoding and packing itself does not include any complex computation, simple byte conversions and manipulations suffice to achieve the same functionality for the given use case [Fou20e].

4.1.2 Client-Side Logic

This section discusses the requirements a blockchain needs to fulfill in order to support a DeXTT implementation, if all logic and relevant data of the protocol resides on the client-side application. This resembles the second edge case for DApp implementations described above.

The requirements for this type of implementation are more relaxed than for a pure smart contract implementation. Because every part of logic, computation and handling of data is done on the client-side of the DApp, the only involvement of the blockchain in the protocol is to provide a secure and tamper-evident way of logging the used DeXTT transactions. This can be achieved by including the DeXTT transactions as blockchain-specific transactions into the blocks of the blockchain, because blockchains can be utilized as a tamper-evident log (see Section 2.1). The specific requirements can be defined as follows:

Secure On-Chain Storage. There must exist a secure way of storing arbitrary data within the blocks of the given blockchain. The minimum required size to store DeXTT transactions themselves depends on how their data is encoded and what signature scheme is used. It is also possible to only store a hash of DeXTT transactions on the blockchain to reduce the size of the stored data. A drawback of this approach is that the actual transactions must be distributed among the client-side applications in another way, as only the hash values are shared via the blockchain. Additionally, the used hash function should be secure with an approximately uniform distribution, such as it is the case for the Keccak-256 [GM11] or SHA-256 [GH04] hash functions. It is also possible to split the DeXTT transactions into multiple parts and put them into multiple blockchain-specific transactions to overcome possible size limits.

Table 4.1: Top-17 ranked entries on CoinMarketCap.

Rank	**Platform**
1	Bitcoin
2	Ethereum
3	XRP
4	Tether
5	Bitcoin Cash
6	Litecoin
7	EOS
8	Binance Coin
9	Bitcoin SV
10	Stellar
11	Tron
12	Cardano
13	UNUS SED LEO
14	Monero
15	ChainLink
16	Tezos
17	Neo

Timestamp. As for an implementation that uses smart contracts for its logic, this approach also requires the inclusion of timing information bundled with the DeXTT transaction, because the protocol relies on specific timings of transactions. This timing information can be given in the form of timestamps that are included in the blocks of the blockchain.

4.2 Blockchain Survey

This section presents a survey of qualified candidate blockchains for a DeXTT implementation including the properties and features of different blockchain technologies currently available. The technical aspects discussed are focused on details that are relevant in regards to the requirements of the DeXTT protocol. Furthermore, the features are compared to the different requirements for DeXTT implementations.

There exists a vast number of different blockchains, therefore the blockchains to be analyzed are preselected based on their rank (based on the market cap of the according cryptocurrencies) on CoinMarketCap[3] because the significance of the chosen blockchain is an important factor. The ranking order taken into account for the survey resembles the state of CoinMarketCap as of the 23rd of November 2019. The top-17 ranked entries shown in Table 4.1 are discussed below.

[3]https://coinmarketcap.com/

4.2.1 Bitcoin

Bitcoin [Nak08] offers Script, which is a simple and stack-based language. It does not support any means of loops and is therefore not Turing-complete [Nar+16]. The same signature scheme as in Ethereum is used, i.e., ECDSA using the *secp256k1* elliptic curve. Script also offers cryptographic hash functions, but the *Keccak-256* hash function is not included. Additionally, to the best of our knowledge, it is not possible to implement Keccak-256 within Script, because crucial operators such as multiplications, divisions, bit-shifting operations and most bitwise logic operations are disabled [bit19g]. Another limitation of Script is the circumstance that is it not possible to change the blockchain state and therefore store data. The only possible outcome of a Script execution is the failing or succeeding of the execution. This outcome can only be utilized to decide whether the according transaction is valid or not [Nar+16]. Concerning data storage on the blockchain, Bitcoin offers a standardized way to store arbitrary data of a maximum size of *80 bytes* within one transaction. More details about Bitcoin Script are given in Section 2.2.5.

Due to the various limitations of Bitcoin Script, Bitcoin does not fulfill the requirements for a DeXTT implementation using *blockchain-side logic*. The biggest limitation is given by the lack of the ability to store, access and manipulate data on the blockchain via Script and to achieve any other output than the validity of a Script run. Because of these limitations, a possible DeXTT implementation for Bitcoin is limited to a *client-side logic* approach. This is possible, because the blockchain supports both *secure on-chain storage*, although limited to 80 bytes per transaction, and timestamps (see Section 2.2.3) of blocks and therefore transactions.

4.2.2 XRP

XRP is a digital asset native to the *XRP Ledger* blockchain [SYB14; CM18]. Its technology is utilized to power the *RippleNet* payments network[4]. The XRP Ledger is an open-source blockchain technology that focuses on fast transaction times but does not support the concept of universal smart contracts, as only predefined contracts can be utilized [Pro19d; Jam18]. The blockchain supports the storing of arbitrary data up to a size of 1kB natively via the *Memo field* included in transactions [Pro19c]. The ledger of the blockchain also contains a timestamp for each version of the ledger via the *close_time* data field [Pro19b]. From the given limitations and features of the blockchain, it can be concluded, that for the XRP ledger, solely a DeXTT implementation using *client-side logic* would be possible.

4.2.3 EOS

EOS is the native token on the *EOSIO* blockchain platform [Blo18]. The EOSIO blockchain utlizes the C++ programming language together with a smart contract Application Program Interface (API) for its smart contract platform. As an execution

[4]https://ripple.com/ripplenet/

environment for smart contracts, a WebAssembly Virtual Machine (VM) called *EOS VM* is used [Blo20b].

The smart contract API offers support for various hash functions and signature verification using ECDSA with the *secp256k1* elliptic curve[5], but the *Keccak-256* hash function is not natively supported [Blo19]. It is of course possible to implement the *Keccak-256* hash function using the C++ programming language and therefore utilize it in a smart contract. An example of how such an implementation could look like can be found on Github[6]. The blocks of EOSIO contain a timestamp that is accessible through smart contracts [Blo20a]. The timestamp is not allowed to lie more than 7 seconds in the future[7].

Due to the capabilities of the C++ programming language [Str13], it is therefore possible to implement the DeXTT protocol for the EOSIO blockchain using the *blockchain-side logic* approach. Furthermore, all requirements for an implementation that provides compatibility with the Ethereum prototype are fulfilled by EOSIO.

4.2.4 Binance Coin

The Binance Coin (BNB) is the cryptocurrency of the Binance platform [Bin17] and native to the *Binance Chain* blockchain [Bin20a]. The blockchain provides decentralized exchange features, but does not support smart contracts [Bin20a; Bin20b]. The Binance Chain includes a timestamp in its block headers, but does not include a native way to store arbitrary data in its blocks [Bin20a]. It is still possible to store data, using encoding approaches as the ones that are utilized by the *Omni Protocol* that builds a protocol layer on Bitcoin [Wil+19]. It can therefore be concluded, that only a *client-side logic* approach of the DeXTT protocol would be possible for the Binance blockchain.

4.2.5 Stellar

The *Stellar* blockchain technology [Maz16] is based on the XRP Ledger platform, but introduces a few changes, such as a different consensus algorithm [BS18]. Smart contracts in Stellar are not given by conventional programming languages, but are expressed as compositions of transactions. These compositions are connected and executed by various constraints. The following constraints can be implemented by Stellar smart contracts: *Multisignature*, *Batching/Atomicity*, *Sequence* and *Time Bounds* [Fou19b; BP17].

Stellar allows the storage of arbitrary data in the *memo* data field inside transactions. It can hold up to 28 bytes of a string encoded in ASCII or UTF-8 or a 32 byte raw hash value [Fou19c]. Just as the XRP ledger, Stellar also contains a timestamp for each ledger version, called *close time* [Fou16].

[5]https://github.com/EOSIO/eosjs-ecc/blob/master/src/key_public.js
[6]https://github.com/ethereum/ethash/blob/master/src/libethash/sha3.c
[7]https://github.com/EOSIO/eos/issues/7447

Due to the given constraints on Stellar smart contracts and the best of our knowledge, the smart contract concept of Stellar does not allow a DeXTT implementation with more blockchain involvement than in the *client-side logic* approach, for which all requirements are fulfilled.

4.2.6 Cardano

The Cardano blockchain uses a research-first driven approach for its development[8,9] [Kia+17; Dav+18; DDL18; CD17; KMZ17]. It is planned that it becomes a smart contract platform utilizing a purely functional programming language called Plutus [Fou20d]. The support for smart contracts is currently under development and not documented extensively [Fou20b]. Therefore, no precise statement about DeXTT support via a *blockchain-side logic* approach can be made.

In Cardano, it is possible to encode arbitrary data into addresses [Fou20a]. Therefore, data can be added to the blockchain, e.g., by sending a small amount of coins to such an address in a transaction[10].

Cardano divides times into *epochs*, which are further divided into *slots*. Both are numbered, starting with zero and included in each block [Fou20c]. This concludes that at least a *client-side logic* approach for a DeXTT implementation is possible for Cardano. It is likely that in the near future, the currently developed smart contracts will allow for an implementation utilizing *blockchain-side logic*.

4.2.7 Monero

The Monero blockchain is built to enhance the privacy of its users, including the concept of untraceable transactions[11]. The blockchain technology does not provide any means of smart contracts, therefore no *blockchain-side logic* approach of the DeXTT protocol is possible [Pro19a].

Monero transactions include a field called *extra*, which usually includes the *transaction public key* or an extra nonce. Because the field is not verified on the blockchain, it can be utilized to include arbitrary data [Alo18]. Additionally, blocks on the Monero blockchain contain timestamps which are added by the miners [Alo18]. Therefore, Monero fulfills the requirements for a *client-side logic* DeXTT protocol implementation.

4.2.8 Tezos

Tezos is a blockchain technology that comes with support for Turing-complete smart contracts [Goo14]. Tezos supports the writing of such smart contracts in multiple different

[8]`https://www.cardano.org/en/home/`

[9]`https://www.cardano.org/en/academic-papers/`

[10]`https://forum.cardano.org/t/can-i-associate-a-message-with-a-wallet-or-transaction-on-the-blockchain/17892`

[11]`https://web.getmonero.org/`

languages, including Pascal, OCaml, Python, Haskell and a domain-specific language called Archtype. Such contracts can then be compiled into the smart contract language of Tezos, called *Michelson* [Tez20b].

Michelson supports access to the timestamp of the current block, includes a *Map* data type natively and also provides signature verification through ECDSA with the *secp256k1* elliptic curve. The *Keccak-256* hash function is not provided, but can be implemented due to Michelson's Turing-completeness [Tez20a].

The capabilities of Tezos fulfill all requirements for a *blockchain-side logic* DeXTT implementation approach including means to provide compatibility to the already existing Ethereum implementation.

4.2.9 Neo

The *Neo* blockchain [Neo20b] offers the support of a vast variety of different programming languages for its smart contract platform. Languages such as C#, Java, C++, GO or JavaScript can be utilized and compiled for Neo's execution environment, the Turing-complete *NeoVM* [Neo20b; Neo20c]. To limit resources within a smart contract, GAS tokens are charged for the operation and storage of smart contracts [Neo20b].

Within a Neo smart contract, the timestamp of the current block can be obtained via a system-call [Neo19]. Although some hash functions are included natively, the *Keccak-256* hash function is not among them, but can be implemented [Neo20a]. Neo utilizes ECDSA with the *secp256r1* elliptic curve. Therefore, signature verification for the *secp256k1* must be implemented by hand to provide compatibility between a new DeXTT implementation and the Ethereum DeXTT prototype [Neo20d; Neo20a].

Due to Neo's features, all requirements for a *blockchain-side logic* DeXTT implementation including compatibility to the existing prototype are fulfilled, although some cryptographic functions must be reimplemented as they are not provided natively.

4.2.10 Other Blockchains Inside Top-17 Ranking

Not all top ranked entries are discussed and presented in detail in the previous sections, because some are not relevant in the context of this thesis or are too similar to already discussed blockchain technologies.

Ethereum [Woo14] is not discussed in detail concerning DeXTT compatibility, because such a DeXTT implementation already exists. More details of Ethereum are presented in Section 2.3.4. The codebase of the *Tron* blockchain was originally a fork form Ethereum. Tron is generally very similar to Ethereum, also utilizing Solidity as a smart contract language and offering similar features, such as a Turing-complete virtual machine [Fou18]. It is therefore implied that DeXTT could be implemented for Tron in a similar fashion.

Some Blockchains are very similar to Bitcoin and therefore the same conclusions concerning DeXTT implementations are implied. *Bitcoin Cash*[12] was forked from Bitcoin and thereby the block size limit was increased to *8MB*. *Litecoin*[13] also emerged from the Bitcoin code base. Its changes compared to Bitcoin comprise a different PoW algorithm called *scrypt* and a shorter targeted block creation time of 2.5 minutes [Pro18]. Another derivate of Bitcoin is *Bitcoin SV*[14], which claims to be the only Blockchain that follows the original vision of the Bitcoin white paper by Nakamoto [Nak08].

Some top ranked entries on CoinMarketCap contain cryptocurrencies that are not coupled to an own blockchain and are therefore not relevant in the context of this survey. The following entries are not discussed any further: *Tether* [Tet16], *UNUS SED LEO* or *LEO* [Bit19] and *ChainLink* [EJN17].

4.3 Discussion of Blockchain Selection

In this section, the results of the blockchain survey in Section 4.2 are discussed and the aspects for the selection of a blockchain technology for a new DeXTT implementation are presented.

The previous section provides a survey of 17 technologies. Three of those are not relevant in regards to a new DeXTT implementation, as they do not provide an own blockchain. Out of the discussed blockchains, *EOS*, *Tron*, *Tezos* and *Neo* provide all means to implement DeXTT for their platforms using a *blockchain-side logic* approach (see Section 4.1.1) while also providing compatibility to the existing Ethereum implementation. Additionally, *Bitcoin*, the *XRP Ledger*, *Bitcoin Cash*, *Litecoin*, the *Binance Chain*, *Bitcoin SV*, *Stellar*, *Cardano* and *Monero* fulfill all requirements for a DeXTT implementation following a *client-side logic* approach (see Section 4.1.2). More details about the fulfillment of the DeXTT requirements by the different blockchains are shown in Table 4.2.

The ranking of blockchains according to their market cap is an important factor, as highly ranked blockchains are typically more popular and used throughout the community. Therefore it is preferable to implement DeXTT on a highly ranked blockchain platform rather than on a less known and used platform. Furthermore, because an implementation for DeXTT using a *blockchain-side logic* approach on Ethereum already exists, creating an implementation utilizing the opposite approach of using *client-side logic* would generally be more valuable in regards of the expected results. This comes from the fact that another *blockchain-side logic* approach will most likely be built in a similar fashion to the existing one. In contrast, for a *client-side logic* solution, other novel design choices have to be made, revealing valuable results of how DeXTT can be implemented in this way and how the implementation differs from the previous approach.

[12]`https://www.bitcoincash.org/`
[13]`https://litecoin.org/`
[14]`https://bitcoinsv.com/`

Table 4.2: Fulfillment of the DeXTT requirements by the surveyed blockchains.

	Signature Verification	Cryptographic Hash Function	Timestamp	Dynamic Data Structures	Control Structures	Operators	Secure On-Chain Storage	Blockchain-Side Approach	Client-Side Approach
Bitcoin	✓	✓	✓	x	✓	✓	✓	x	✓
Ethereum	✓	✓	✓	✓	✓	✓	✓	✓	✓
XRP	x	x	✓	x	x	x	✓	x	✓
Bitcoin Cash	✓	✓	✓	x	✓	✓	✓	x	✓
Litecoin	✓	✓	✓	x	✓	✓	✓	x	✓
EOS	✓	✓	✓	✓	✓	✓	✓	✓	✓
Binance Coin	x	x	✓	x	x	x	✓	x	✓
Bitcoin SV	✓	✓	✓	x	✓	✓	✓	x	✓
Stellar	x	x	✓	x	x	x	✓	x	✓
Tron	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cardano	x	x	✓	x	x	x	✓	x	✓
Monero	x	x	✓	x	x	x	✓	x	✓
Tezos	✓	✓	✓	x	✓	✓	✓	x	✓
Neo	✓	✓	✓	x	✓	✓	✓	x	✓

Due to these observations, we choose *Bitcoin* as a blockchain platform for the DeXTT implementation. Most crucially, because Bitcoin is by far the most popular cryptocurrency as of March 2020, being ranked on first place based on its market cap on CoinMarketCap[15]. Another important factor for the selection process is not only that Bitcoin requires a *client-side logic* implementation (see Section 4.2), but also that many very similar derivatives of Bitcoin exist, such as Bitcoin Cash and Litecoin. A DeXTT implementation for Bitcoin can therefore be applied in a very similar fashion to many other existing blockchains.

[15] https://coinmarketcap.com/currencies/bitcoin/

CHAPTER 5

DeXTT-Bitcoin Design

In this chapter, the design approaches for a DeXTT implementation for the Bitcoin blockchain are presented. The overall design and concept is generally applicable and forms an abstraction from the actual implementation details, which are later described in Chapter 6. The concept of our design takes into account all specific implementation choices of the Ethereum prototype[1] that are required to provide compatibility between the two DeXTT protocol designs of both blockchains.

The chapter starts with Section 5.1, presenting the concept of embedding the various DeXTT transactions in the Bitcoin blockchain, as it is required for a *client-side logic* DeXTT approach as described in Section 4.1.2. To the best of our knowledge, this is the approach with the most blockchain involvement possible for Bitcoin (see Section 4.2.1). Thereafter, the handling of the communication between the client-side application and the Bitcoin blockchain is discussed in Section 5.2. The subsequent Section 5.3 presents the different aspects of the participation in witness contests. Next, an approach of including unconfirmed transactions in the design concept is introduced in Section 5.4. This chapter is concluded by Section 5.5, where the client logic regarding the timing of sending transactions in multi-blockchain environments is presented.

5.1 Embedding DeXTT Transactions in Bitcoin

Within this section, the solution approach for the inclusion and embedding of DeXTT transactions in the Bitcoin blockchain is presented. Such functionality is required to store and log all DeXTT transactions via the blockchain in a secure and tamper-evident way. To achieve this, a suitable data inclusion method for the blockchain has to be designed and a way to embed the different data needed by DeXTT has to be found.

[1]`https://github.com/pantos-io/dextt-prototype`

To embed arbitrary DeXTT payloads in Bitcoin, first, a way of generally storing data in the blockchain has to be defined. Such an approach is presented in Section 5.1.1. Next, Section 5.1.2 is comprised of a discussion of how the different transactions needed for the DeXTT protocol can be embedded and included in Bitcoin utilizing our presented method. Furthermore, the precise structure of our solution approach is presented in the same section.

5.1.1 Bitcoin Transaction Format

This section discusses the inclusion of arbitrary data in the Bitcoin blockchain, which is required for a *client-side logic* DeXTT implementation.

Problem Statement

The DeXTT protocol defines multiple different transactions that are utilized for its execution (see Section 2.4). To adopt the protocol for the Bitcoin blockchain, these transactions have to be included in some way in the blockchain, to allow the secure and tamper-evident storage and access of these transactions. Each Bitcoin block automatically includes the timestamp of its creation (see Section 2.2.3). Therefore, together with the inclusion of the various DeXTT transactions, such a storage mechanism of transaction logs that allows the deterministic execution of the transactions can be created.

Solution Approach

In our design approach, the *null data* standard transaction of Bitcoin is utilized (see Section 2.2.5). Such transactions allow the storage of arbitrary data in a transaction output. No further parts of the Script language of Bitcoin is used for our approach of integrating the DeXTT protocol into Bitcoin. The main reason to choose this way of including arbitrary data is that is the recommended way to do so [bit20d]. Using a null data transaction, the UTXO database of Bitcoin does not include the according output, immediately marking the output as not spendable. Apart from storing data off-chain, null data transactions represent the best approach for data inclusion regarding the blockchain ecosystem [bit20d; bit19g]. A similar approach of including a protocol merely through data inclusion in Bitcoin is used by e.g. the *Omni Protocol* [Wil+19] or *Counterparty* [Cou19], effectively building a protocol layer on Bitcoin.

One downside of using null data transaction outputs is the size limit of *80 bytes* for its payload [bit20d]. This limitation can easily be overcome by simply using multiple null data outputs. The data can then be assembled off-chain. Because null data outputs are limited to one per transaction, multiple transactions have to be utilized to write more than *80 bytes* of data onto the blockchain [Ant14].

Every Bitcoin transaction needs to include at least one input to be considered valid [bit17a]. Furthermore, for a transaction to be included in a new block with higher probability, an according transaction fee has to be paid [EOB19; bit19e]. Therefore, the sum of all

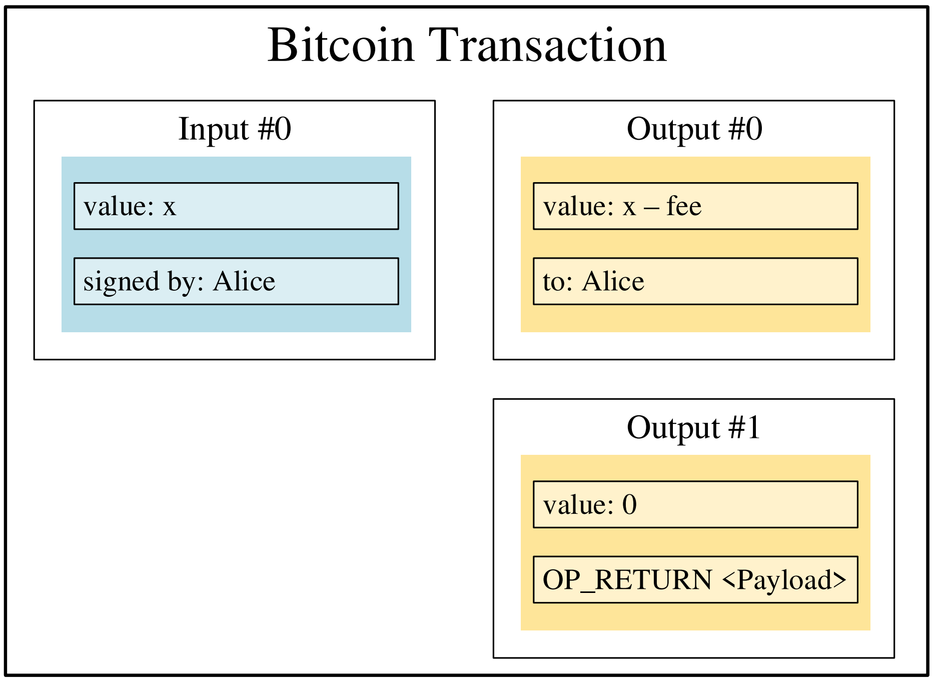

Figure 5.1: Illustration of the format of a Bitcoin transaction in DeXTT.

inputs of a Bitcoin transaction with embedded DeXTT data has to be high enough for the required transaction fee. In our approach, the number of inputs is fixed at one to reduce the complexity of searching UTXOs and building the transactions. Additionally, this simplification reduces the size of the actual transaction, effectively lowering the fee. As a result of how Bitcoin transactions are designed (see Section 2.2.4), the change of coins (input value − transaction fee) has to be put in an additional transaction output.

The overall Bitcoin transaction structure that is used for DeXTT transactions is illustrated in a simplified form in Figure 5.1. The transaction consists of one input that pays for the transaction fee and two outputs. The first output returns the change of coins to the sender. The second output represents the null data output used to embed DeXTT transactions. It has an output value of zero and utilizes the OP_RETURN opcode in its output script as described in Section 2.2.5.

The Bitcoin transaction structure that handles the DeXTT payload is designed to utilize native SegWit transactions for the transaction input and the change output. Therefore *Bech32 addresses* and P2WPKH standard transactions are used (see Section 2.2.5). This method is also used as the default format for transactions since *Bitcoin Core version 0.19.0.1*[2].

[2]https://bitcoin.org/en/release/v0.19.0.1

By using native SegWit transactions, the size and therefore the required fee for such a Bitcoin transaction is reduced (see Section 2.2.3). The size of the Bitcoin transactions to handle DeXTT payloads averages out at about 121 vbytes without any included data. The actual size is variable, because the length of the used signatures in the DER format is not fixed and variable length integers are used to describe different length fields in transactions [ITU15; bit20c]. The base size of about 121 vbytes is composed of 109 vbytes for a transaction with one SegWit input and one SegWit output and 12 vbytes for an empty null data output[3] [bit20c; LLW15].

Alternative Approaches

There are also other means of including arbitrary data in Bitcoin transactions to store the data on the blockchain. A simple approach is to encode data in fake Bitcoin addresses that are then used in transaction outputs [Wil+19]. This solution bloats the UTXO database by creating unspendable outputs and therefore should be avoided. A more advanced way of storing data in transactions without bloating the UTXO database of Bitcoin can be achieved by using 1-of-n multi-signature transactions (see Section 2.2.5) and encode arbitrary data in $n - 1$ fake public keys. The n^{th} public key can be used to redeem the transaction output later on [Wil+19]. This approach is more complex compared to the simple and recommended approach of using null data outputs.

5.1.2 DeXTT Transaction Format

Within this section, the data format of the different DeXTT transactions is presented. This representation is used to embed the transactions in the Bitcoin blockchain, utilizing Bitcoin transactions as described in the previous section. Therefore, for each individual DeXTT transaction the protocol is comprised of, a suitable data representation has to be defined, which can then be included in the blockchain. In our design, the sender's intent of a DeXTT token transfer is handled off-chain (see (2.1) in Section 2.4.2). Therefore, no data format for blockchain inclusion of this data has to be defined. The details about the off-chain sharing of sender intent data is implementation specific and described in Section 6.2.2.

In our design approach, the entire data of DeXTT transactions themselves is included in the blockchain via the specified format within this section. Another approach to securely include data in the blockchain to allow deterministic execution of those specific protocol transactions would be to only store hash values of the actual transactions on the blockchain [Wil+19]. The hashes would still allow to verify which transaction was included and therefore executed on the blockchain in which block. Furthermore, this approach only requires a small and fixed size of data to be stored on the blockchain, e.g., a 32 byte hash value. This can potentially lower the actual transaction size and cost. The downside of this alternative solution is that the actual transaction data has to be

[3]https://www.reddit.com/r/Bitcoin/comments/7m8ald/how_do_i_calculate_my_fees_for_a_transaction_sent/

distributed among all participating clients, effectively building a custom network for data exchange between all clients, similar to the one used for the actual blockchain [Wil+19]. This brings the overhead of building, managing and maintaining this additional network. Therefore, this alternative approach is not considered in the context of this thesis and prototype implementation.

Addresses

The DeXTT protocol requires the encoding of the source and destination wallet of a token transfer in its transactions. This can be achieved by providing the public part of the used cryptographic public/private key pair or by deriving some form of address from this public key as it is done in Bitcoin or Ethereum (see Chapter 2). For compatibility reasons, ECDSA [JMV01] with the *secp256k1* elliptic curve [Hes00] is used for the public/private key pair. Furthermore, the same key pair that is used to interact with the Bitcoin blockchain is also used for DeXTT by each participant. This enables the concept of utilizing the sender of Bitcoin transactions as a form of signature (see Section 5.3.1).

In the already existing DeXTT Ethereum prototype implementation[4], the source and destination wallets are encoded using Ethereum addresses that are derived from the according public key. These addresses are also used to derive the contest signatures for the witness contests in the DeXTT Ethereum implementation[5]. Therefore it is required for a new implementation to utilize the same address format to encode the wallets to reach compatibility with the Ethereum prototype. If another format is used, the contest signatures are different across the implementations, leading to inconsistencies between the blockchains.

Compatible addresses must therefore use the address format of Ethereum, being derived from the last 20 bytes of the Keccak-256 hash of the corresponding public key [AW18]. In addition to the achieved compatibility, these addresses also come with the advantage of needing only 20 bytes of storage space, effectively reducing the transaction cost compared to an approach where the whole public key is included [JMV01; Hes00].

Signature Format

The signatures that are used in various DeXTT transactions also need to be encoded using the same format as the Ethereum prototype to provide compatibility between the implementations. A public/private key pair for ECDSA [JMV01] with the *secp256k1* elliptic curve [Hes00] is used to create the digital signatures for the protocol. Furthermore, the signatures are composed of the concatenation of the three values v, r and s of the signature.

In contrast to, e.g., the DER signature encoding used in Bitcoin, the utilized signatures allow the recovery of the public key that was used to create it. For this key recovery,

[4]`https://github.com/pantos-io/dextt-prototype`
[5]`https://github.com/pantos-io/dextt-prototype/blob/master/truffle/contracts/Cryptography.sol`

Algorithm 5.1: Signature Verification

```
  Input: Signature $sig$, Signed data $msg$, Address $addr$
  Result: Validity of Signature, either $True$ or $False$
1 $recoveredKey = recover(signature, msg)$;
2 $recoveredAddress = ethereumAddress(recoveredKey)$;
3 if $recoveredAddress == addr$ then
4     return $True$;
5 else
6     return $False$;
7 end
```

the signature and the signed data is needed. This property allows the omitting of the public keys from DeXTT transactions, if the according addresses of the signers of the signatures are included. The signature verification can then be achieved as illustrated in Algorithm 5.1. First, the public key that was used to sign is recovered from the given signature in line number 1. Next, its corresponding address is calculated in line number 2. The calculation of the corresponding address is done by taking the last 20 bytes of the Keccak-256 hash of the recovered key. The recovered address is then compared to the actual address of the party who supposedly signed the data in line number 3. If the addresses match, the signature is considered valid and `True` is returned (line number 4), if they do not match the algorithm returns `False` (line number 6).

Transaction Prefix

Each DeXTT transaction that is embedded in Bitcoin needs to be marked in a way to recognize it as a DeXTT payload among all existing null data outputs that are included in the Bitcoin blockchain. This allows for easy filtering of DeXTT payloads without the need to try to fully parse each individual null data output. To achieve this, a *marker prefix* is included as the first bytes of each DeXTT transaction, containing the word *"DeXTT"* as bytes using the according American Standard Code for Information Interchange (ASCII) representation. This yields the raw byte values in hexadecimal format of `0x4465585454`, consisting of five bytes [INC17]. Furthermore, each DeXTT payload contains one byte to indicate the *protocol version*. This allows to upgrade the specification of the DeXTT-Bitcoin design later on, while being able to provide compatibility with older versions. For the current design, the hexadecimal byte value `0x01` to indicate version 1 is used. The prefix of each embedded transaction additionally contains one byte that indicates the DeXTT *transaction type* that is represented by it.

Each DeXTT payload data fragment therefore contains a prefix consisting of seven bytes, leaving room for additional 73 bytes of data per null data output. The remaining data of the payloads depend on the used transaction type and differ in their size and structure. Through the encoding of the type, the remaining structure of transactions can be different for each type. The different transaction types and how they are encoded in our design approach to be included in the Bitcoin blockchain are presented in the next sections.

Claim Transaction

The *claim* transaction as defined in the DeXTT protocol (see (2.3) in Section 2.4.2) contains a PoI that consists of the following data elements: source ($\mathcal{W}_s$) and destination wallet ($\mathcal{W}_d$), the token amount (x) that is transferred, start (t_0) and end time (t_1) of the validity period and the signatures created by the source (α) and destination wallet (β) [Bor+19b].

Both source and destination wallet are represented by using Ethereum addresses, therefore each of them requires 20 bytes of storage space. Each of the two signatures of the *claim* transaction use the format as described in the previous section. Storing a signature in this way requires a fixed size storage space of 65 bytes [Woo14]. The signatures are produced the same way as in the Ethereum prototype. The source wallet's signature α is created by signing the output of the *Keccak-256* hash of the data comprosed of [“a”, $\mathcal{W}_s$, $\mathcal{W}_d$, x, t_0, t_1]. The destination wallet's signature β is generated by signing the output of the *Keccak-256* hash of [“b”, α].

The inclusion of destination and source addresses and the two signatures alone requires 170 bytes ($2 \cdot 20 + 2 \cdot 65$) of blockchain storage space. Because one null data output in Bitcoin can at most hold 80 bytes of data, at least three such outputs are required to fit one DeXTT *claim* transaction in the blockchain. The structure and encoding of the remaining data for a claim transaction is not strictly given by compatibility constraints. When using three null data outputs, after considering the 170 bytes for addresses and signatures together with 21 bytes ($3 \cdot 7$ *bytes*) for the transaction prefixes, 49 bytes ($240 - 170 - 21$) out of the 240 totally available bytes are free to be occupied by the remaining parts of the *claim* transaction.

In our design approach, we therefore choose to split each *claim* transaction into three individual parts to be stored in the Bitcoin blockchain. The data is split into the following three DeXTT-Bitcoin Claim transactions:

DeXTT-Bitcoin Claim data transaction. The first partition of the *claim* transaction contains the actual data of the token transfer. The actual structure of the payload data is shown in Table 5.1. The *DeXTT-Bitcoin Claim data transaction* contains the transaction prefix with the according transaction type in the field `Type`. Furthermore, both the source and destination addresses are included. The amount of tokens of the transfer is represented by a 16 byte value, interpreted as an unsigned integer. This allows token transfers of up to $2^{128} - 1$ tokens in one transaction which should be sufficient for must use cases. The start and end times of the validity period are represented using 4 bytes per timestamp. These 4 bytes are used to encode a 32 bit unsigned integer value that is interpreted as a Unix epoch timestamp. This is the same format as is used for timestamps in the Bitcoin blockchain (see Section 2.2.3). Therefore, the used format does not introduce any

Table 5.1: DeXTT-Bitcoin Claim data transaction.

	Marker	Version	Type	Source	Destination	Amount	t_0	t_1	Hash
Value	`0x4465585454`	`0x01`	`0x01`	$< addr >$	$< addr >$	$< uint >$	$< t >$	$< t >$	$< hash >$
Size	5 bytes	1 byte	1 byte	20 bytes	20 bytes	16 bytes	4 bytes	4 bytes	8 bytes

Table 5.2: DeXTT-Bitcoin Claim Signature α transaction.

	Marker	Version	Type	Source Signature	Hash
Value	`0x4465585454`	`0x01`	`0x02`	$< signature\alpha >$	$< hash >$
Size	5 bytes	1 byte	1 byte	65 bytes	8 bytes

Table 5.3: DeXTT-Bitcoin Claim Signature β transaction.

	Marker	Version	Type	Destination Signature	Hash
Value	`0x4465585454`	`0x01`	`0x03`	$< signature\beta >$	$< hash >$
Size	5 bytes	1 byte	1 byte	65 bytes	8 bytes

loss of accuracy or range for the timings of DeXTT transactions. In addition to the transaction prefix and the data values needed by the *claim* transaction, an 8 byte hash value is included in each DeXTT-Bitcoin Claim data transaction. The usage of this hash value is discussed later on. The total size of a *DeXTT-Bitcoin Claim data transaction* therefore amounts to 79 bytes $(7 + 20 + 20 + 16 + 4 + 4 + 8)$.

DeXTT-Bitcoin Claim Signature α transaction. The second partition of the *claim* transaction is comprised of the signature α of the source of the token transfer. The structure of the transaction part is shown in Table 5.2. In addition to the transaction prefix that contains the according transaction type, the 65 byte signature α of the transaction sender is included. The last data field in this transaction again includes an 8 byte hash value that is further discussed later on. The *DeXTT-Bitcoin Claim Signature α transaction* therefore requires 80 bytes $(7 + 65 + 8)$ of storage.

DeXTT-Bitcoin Claim Signature β transaction. The last partition of the *claim* transaction comprises the second needed signature β created by the receiver of the token transfer. The transaction structure is presented in Table 5.3. First, the transaction prefix with the according transaction type value is included. Besides the included signature β, again an 8 byte hash value is contained in the transaction. The usage of this hash is discussed in the following paragraph. This transaction structure results in a total size of 80 bytes $(7 + 65 + 8)$ for the *DeXTT-Bitcoin Claim Signature β transaction.*

A *claim* transaction can only be executed by the client-side application after all partitions of the same claim are combined again. To optimize this assembly process, all three transactions that are used to include a *claim* transaction in Bitcoin contain the same 8 byte hash value. The hash value is created by first calculating the Keccak-256 [Ber+11] hash of the data that is also used to create the signature α. This data is comprised of the following parts: [“a”, $\mathcal{W}_s$, $\mathcal{W}_d$, x, t_0, t_1]. Next, the last 8 bytes of the 32 byte Keccak-256 hash are taken to form the desired hash value. The size of 8 bytes is chosen in our design as it is the maximum size that could be included in both the *DeXTT-Bitcoin Claim Signature α transaction* and the *DeXTT-Bitcoin Claim Signature β transaction* without the need to reduce any other parts of the transaction in its size. Furthermore, although an 8 byte hash does not provide the same strength as the whole 32 byte hash value [GM11], there exist 2^{64} different possible values, therefore a hash collision is still expected to occur very rarely. The likelihood of a collision reaches 50% after around 2^{32} transactions [Sma16].

This 8 byte hash is utilized in our design approach to identify the according *claim* transaction. Through this identification, the complexity to assemble the whole *claim* transaction from the three DeXTT-Bitcoin transactions is massively reduced for the client-side application. Instead of matching transaction parts by trying to verify the signatures of all currently unassembled transaction parts with each other, only the parts with matching hash values have to be considered for the signature verification. This reduces the complexity to constant time, if no hash collisions occur. A hash collision does simply increase the matching effort slightly, raising the need to try to validate the signatures of multiple transaction parts with each other. More details of how the partitions of a *claim* transaction are combined in our client-side application implementation is presented in Section 6.3.

Contest Transaction

The *contest* transaction as it is formally defined in the DeXTT protocol (see (2.4) in Section 2.4.3) consists of the whole PoI data of the token transfer as it is included in the claim transaction [Bor+19b]. Additionally, the signature ω of the contestant is included. This additional signature results in the need for an additional DeXTT-Bitcoin transaction. To reduce the cost and transactions needed for DeXTT on the Bitcoin blockchain, we choose to reuse existing claim transactions on the blockchain and introduce one additional DeXTT-Bitcoin Contest transaction that takes advantage of the existing data. More details on how the reuse of claim transactions is done in our design is discussed in detail in Section 5.3.2.

Because it is assumed for the *DeXTT-Bitcoin Contest transaction* that the entire PoI data is already present on the blockchain, the contest transaction only needs to incorporate a signature of the contestant that verifies that the contestant indeed wants to participate in the contest. In our design of DeXTT for Bitcoin, we aim to provide full compatibility with the Ethereum prototype. In contrast to the formal protocol specification, this prototype does not use the cryptographic signature ω for the selection of the winning witness, but rather a hash value is utilized for the contest signature for the selection. This hash is

Algorithm 5.2: Calculation of contest signature.

```
    Input: Address $contestantAddress$, PoI data: $[\mathcal{W}_s, \mathcal{W}_d, x, t_0, t_1]$
    Result: Contest Signature of contestant
1 $poiHash = keccak256(“a”, \mathcal{W}_s, \mathcal{W}_d, x, t_0, t_1)$;
2 return $keccak256(contestantAddress, poiHash)$;
```

Table 5.4: DeXTT-Bitcoin Contest transaction.

	Marker	Version	Type	Full Hash
Value	`0x4465585454`	`0x01`	`0x04`	$< hash >$
Size	5 bytes	1 byte	1 byte	32 bytes

calculated from the contestant's address and the PoI data as shown in Algorithm 5.2[6]. The participant with the smallest contest signature wins the contest.

Because the contest signature is created without the usage of the signature ω, another way of providing a proof that the contestant wants to participate in the contest can be used. In our approach, we take advantage of the fact that a Bitcoin transaction needs to be signed by the sender, therefore the data included in the transaction is also cryptographically signed by the sender. More details on how this information is embedded and can be used to verify the transaction sender are shown in Section 5.3.1. Because the contest transaction is posted to the blockchain by the contestant and therefore is also signed by her, only some data to identify the according claim transaction must be included. In our approach, the Keccak-256 hash of $[“a”, \mathcal{W}_s, \mathcal{W}_d, x, t_0, t_1]$ is embedded in the contest transaction, which effectively references the according PoI data [GM11]. The data that is hashed is the same as for the signature α and the 8 byte hash value in the DeXTT-Bitcoin Claim transactions. The structure of the *DeXTT-Bitcoin Contest transaction* is shown in Table 5.4. This approach of only including a 32 byte hash value instead of a 65 byte signature reduces the transaction size and therefore also the cost of a contest participation. The total size of a DeXTT-Bitcoin Contest transaction consists of 39 bytes $(7 + 32)$.

Finalize Transaction

The *finalize* transaction that is defined in the DeXTT protocol (see (2.5) in Section 2.4.4) formally only consists of the signature α of the sender of the token transfer [Bor+19b]. The purpose of this signature is to uniquely identify the corresponding token transfer and claim transaction. To reduce the size of the *DeXTT-Bitcoin Finalize transaction*, the same mechanism as in the DeXTT-Bitcoin Contest transaction to identify the PoI data is utilized. Therefore, instead of a 65 byte signature, a 32 byte hash value of

[6]`https://github.com/pantos-io/dextt-prototype/blob/master/truffle/contracts/Cryptography.sol`

Table 5.5: DeXTT-Bitcoin Finalize transaction.

	Marker	**Version**	**Type**	**Full Hash**
Value	`0x4465585454`	`0x01`	`0x05`	$< hash >$
Size	5 bytes	1 byte	1 byte	32 bytes

Table 5.6: DeXTT-Bitcoin Finalize-Veto transaction.

	Marker	**Version**	**Type**	**Conflicting Sender**
Value	`0x4465585454`	`0x01`	`0x06`	$< addr >$
Size	5 bytes	1 byte	1 byte	20 bytes

the PoI is included in the finalize transaction. The structure including the according transaction type byte value is presented in Table 5.5. Just as the contest transaction, the *DeXTT-Bitcoin Finalize transaction* takes up 39 bytes $(7 + 32)$ of storage on the blockchain.

Veto Transaction

To embed *veto* transactions (see (2.6) in Section 2.4.5) in the Bitcoin blockchain, the same approach as in the Ethereum prototype implementation is utilized. Veto transactions are implicitly included by using the ordinary contest transactions. The client-side application automatically interprets the contest transaction as a veto transaction if a conflicting PoI exists. Therefore, protocol users that participate in all possible witness contests automatically participate in the veto contests.

Finalize-Veto Transaction

In the *finalize-veto* transaction of the DeXTT protocol (see (2.8) in Section 2.4.5), both signatures α and α' by the sender that created both conflicting PoIs are included [Bor+19b]. The finalize transaction is utilized in the design in our client-side application, just as in the Ethereum implementation, to identify the sender's address of the conflicting token transfers. Therefore, it is sufficient to only include the address of the sender of the conflicting PoIs instead of the signatures. This effectively reduces the size and complexity of the transaction. The structure of the DeXTT-Bitcoin Finalize-Veto transaction is shown in Table 5.6. The resulting transaction occupies 27 bytes $(7+20)$ on the blockchain.

Additional Transaction: Mint Transaction

The formal specifications of the DeXTT protocol do not provide any means of token creation or minting [Bor+19b]. Such functionality is required in the design of the Bitcoin implementation, because there must exist a way of minting tokens to excessively test and evaluate the protocol implementation (see Chapter 7). Therefore an additional

Table 5.7: DeXTT-Bitcoin Mint transaction.

	Marker	Version	Type	Receiver	Amount
Value	`0x4465585454`	`0x01`	`0x06`	$< addr >$	$< uint >$
Size	5 bytes	1 byte	1 byte	20 bytes	16 bytes

transaction for the minting of tokens is introduced. The same concept is utilized within the Ethereum implementation.

Instead of such an additional transaction, it would also be possible to hard-code certain amounts of tokens into the testing environment in the client-side application. But that approach does not allow the same properties as transactions that are included in the blockchain. Most importantly, the minting of tokens would not be stored and distributed through the blockchain, making it impossible to dynamically mint additional coins and synchronize the information between all clients.

The mint transaction contains the same transaction prefix as the other transactions and additionally includes two data fields. Its structure is shown in Table 5.7. First, the address of the receiver of the minted tokens is specified. Furthermore, the amount of tokens that is created through the mint transaction is included, again as a 16 byte unsigned integer value. When the mint transaction is executed, the specified amount of tokens is added to the balance of the wallet that is indicated through the receiver address.

To prevent every participating client to mint an arbitrary amount of tokens, within the client-side application of the DeXTT implementation for Bitcoin a special address is hard-coded. This address represents the only participant that is allowed to mint tokens, therefore only mint transaction that are posted by this participant are executed. All other minting attempts are ignored. The same concept of minting transactions is used in the Ethereum implementation of DeXTT, including the same restrictions on its sender[7]. Details of how the sender of the transaction is verified are discussed in Section 5.3.1.

5.2 Blockchain Interaction

This section presents the means of communication between the client-side application for DeXTT and the Bitcoin blockchain. To include DeXTT transactions in the blockchain and read and execute them later on, a way of sending valid transactions to the blockchain and accessing transactions that were included in the blockchain is required. Section 5.2.1 presents and discusses the chosen way of communicating with the blockchain. Furthermore, Section 5.2.2 shows how the general interaction sequence of the client-side application with the blockchain is designed to achieve the intended functionality of the client.

[7]`https://github.com/pantos-io/dextt-prototype/blob/master/truffle/contracts/PBT.sol`

5.2.1 Communication Approach

In this section, the communication with a blockchain is discussed. This communication is mainly comprised of writing and reading transactions to and from the blockchain.

Problem Statement

The client-side application of the DApp that represents the DeXTT implementation for the Bitcoin blockchain needs to communicate with the blockchain to store and access data that is included on the blockchain-side of the DApp. This is achieved by sending valid transactions to the blockchain and reading transactions that were sent to the blockchain. To create valid transactions that contain DeXTT payloads via null data outputs, information about previous transactions needs to be available to be able to use a valid UTXO for the input of the transaction. Furthermore, to execute DeXTT transactions and to create new ones according to the protocol, the previous DeXTT transactions are needed in the client-side application. Otherwise, it is not possible to recreate the correct state of open token transfers and user balances that is required for future protocol compliance. Therefore, at least a certain subset of blockchain state needs to be accessible via the client, containing UTXOs data of the according Bitcoin wallet and all relevant DeXTT-Bitcoin transactions that are already included on the blockchain.

Solution Approach

To provide the required means of communication between the client-side application and the Bitcoin blockchain, our design approach utilizes the *Bitcoin Core* [bit20b] client software that yields an implementation of a full Bitcoin node [bit19a]. Because Bitcoin Core represents a full node, the program stores and validates the state of the entire Bitcoin blockchain and can also send newly created transactions to the Bitcoin blockchain network [bit19c]. Bitcoin Core can operate on the Bitcoin mainnet or testnet and also allows the usage of privately managed blockchains via its regtest mode (see Section 2.2.7).

The interaction between the client-side application of the DeXTT-Bitcoin DApp and Bitcoin Core is handled by using the Remote Procedure Call (RPC) interface that is offered by Bitcoin Core. This interface provides access to the APIs of Bitcoin Core through the use of Hypertext Transfer Protocol (HTTP), where JavaScript Object Notation (JSON) objects are embedded in the request and response messages. The data format that is used for the JSON-RPC interface is based on version 1.0 of the JSON-RPC specification[8] [bit20c; bit20a].

In our design of the DeXTT protocol for Bitcoin, this JSON-RPC interface it used to communicate with Bitcoin Core and therefore also to interact with the Bitcoin blockchain by using the according methods of the API provided by Bitcoin Core. The overall design approach of the communication means between these components in illustrated in Figure 5.2. It shows that the DeXTT-Bitcoin client only needs to store DeXTT-specific

[8]`https://www.jsonrpc.org/specification_v1`

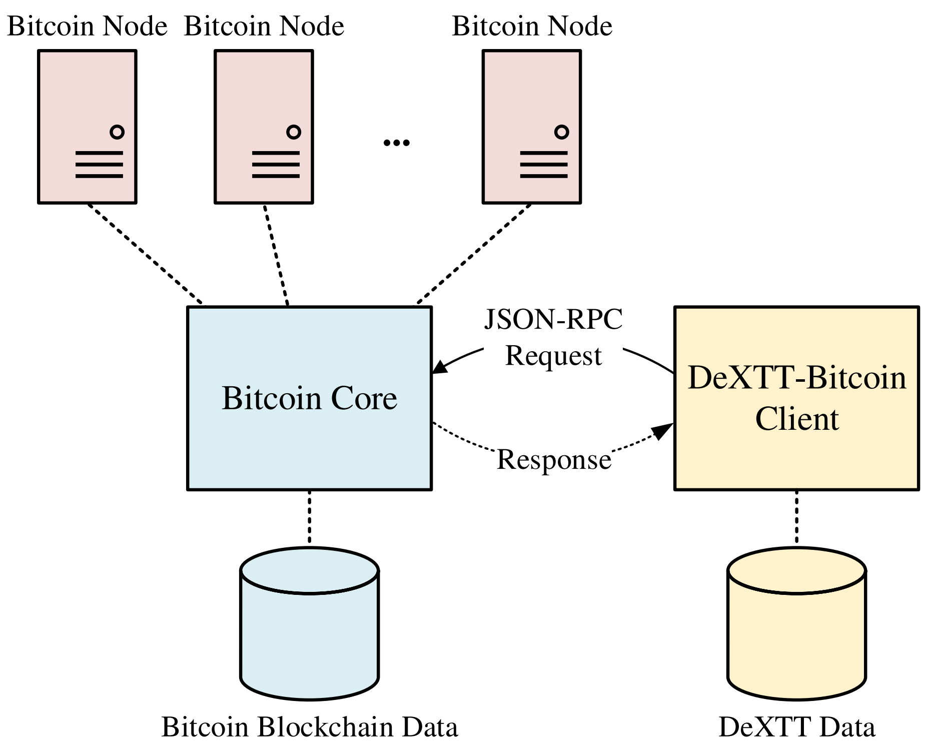

Figure 5.2: Communication between DeXTT-Bitcoin client and Bitcoin Core.

data. All required data from the Bitcoin blockchain is stored and managed by Bitcoin Core and accessed by the DeXTT client via the JSON-RPC interface. Moreover, Bitcoin Core is connected to the actual nodes of the Bitcoin blockchain network and handles the according communication with the blockchain.

The JSON-RPC interface of Bitcoin Core is used for all communication means with the Bitcoin blockchain. Furthermore, Bitcoin Core and its APIs are also utilized to manage the Bitcoin wallets and addresses that are needed to post new transactions to the blockchain. All of the RPC calls that are used throughout our design and implementation of the DeXTT protocol are described as follows [bit20c; Cor19]:

CreateRawTransaction. This RPC creates a new transaction that spends the specified inputs and creates the specified outputs. The return value of the call contains a hex-encoded raw transaction. The inputs of the created transaction do not get signed and the transaction is not transmitted to the Bitcoin network by calling this method. The required arguments include the inputs and outputs of the transaction.

DecodeRawTransaction. By calling this RPC, a given hex-encoded raw transaction is decoded and the serialized representation of the transaction is returned. This returned transaction contains all relevant transaction data in decoded serialized form.

DumpPrivKey. This call reveals the *private key* that corresponds to the specified Bitcoin *address*.

EstimateSmartFee. This function returns an estimation for the current transaction fee per kB that is required for a transaction to be confirmed within the specified number of blocks.

GenerateToAddress. This RPC method immediately mines the specified number of blocks and sends the newly created Bitcoin to the given address. The required arguments of this call comprise the number of blocks to be mined and the Bitcoin address to which the generated amount of Bitcoin is sent to. This RPC is only available in *regtest mode*.

GetBlock. RPC that returns the data of a Bitcoin block. The desired block is specified by providing the according hash value of the block as a hex-encoded string value as an argument to the call.

GetBlockCount. This call returns the number of blocks that are contained in the currently longest blockchain.

GetBlockHash. This RPC returns the hash value of a Bitcoin block. The block is specified via its height in the currently longest blockchain.

GetRawTransaction. Returns a hex-encoded raw Bitcoin transaction that is identified by its *transaction id* as a parameter to the RPC.

ListSinceBlock. This RPC returns all confirmed transactions that were included in the blockchain since the specified block was generated. This block is identified via its block hash that is provided via an argument to the RPC.

ListUnspent. Via this call, a list of UTXOs is provided as an array. The transactions can be filtered to have a certain number of confirmations. This can be specified by providing a minimum confirmation count or a maximum confirmation count or both to which the UTXOs must comply.

SendRawTransaction. This RPC can be utilized to send a serialized hex-encoded raw Bitcoin transaction to the Bitcoin network.

SignRawTransactionWithKey. This call is used to sign the inputs of a serialized hex-encoded raw transaction. Both the transaction that should be signed and the private key to be utilized for the signature are provided via arguments to the RPC. The result of this call includes a hex-encoded raw transaction including the newly created signatures.

WalletPassphrase. This RPC stores the decryption key of the currently used Bitcoin wallet in memory for the specified amount of time. While the decryption key is stored in memory, tasks that require private keys can be performed, e.g., dumping the private key via the *DumpPrivKey* RPC. The *WalletPassphrase* RPC requires two arguments: the passphrase of the wallet and a timeout value.

Discussion of Alternatives

It is also possible to apply other approaches to achieve the desired communication with the Bitcoin blockchain. One possible solution is the usage of a slim node or client that communicates with an external service. The blockchain data itself is therefore not stored by the client but only accessed through a service that stores the data. One example of such a blockchain web service is *BlockCypher*[9]. The advantage of this approach is that less data has to be stored locally. A disadvantage of this method is that the communication and accesses to these services are often limited, e.g., only allowing a certain number of requests within a given period of time. Additionally, it is not possible to use something similar as the regtest mode of Bitcoin Core (see Section 2.2.7). Therefore, this limits the client to use either the Bitcoin mainnet or testnet. Another disadvantage is that the data provided by the blockchain web service must be fully trusted using such an approach.

It would also be possible to go without any additional software or service, effectively implementing a custom Bitcoin node that handles all of the communication with the network itself [bit19c]. This approach results in an implementation overhead through which also additional errors might be introduced. It is therefore a better approach to rely on third party software such as Bitcoin Core, that is used and tested thoroughly.

5.2.2 Blockchain Interaction Overview

This section gives an abstract overview of how the client-side DeXTT application interacts with Bitcoin Core and therefore with the Bitcoin blockchain. The description of interactions is reduced to the core functionality of the client software, namely the reading and execution of DeXTT transactions. The client design of DeXTT for Bitcoin includes the functionality to work on multiple Bitcoin blockchains simultaneously. This ability is needed to allow the interaction and therefore the execution of the DeXTT protocol in an environment that consists of multiple blockchains, such as it is possible by using the regtest mode of Bitcoin Core (see Section 2.2.7).

A DeXTT client in a *client-side logic* implementation must continuously process all newly created Bitcoin blocks and transactions. For this, it must filter each transaction output in search for a null data output that contains a DeXTT payload. All found DeXTT transactions must then be executed, effectively changing the state of DeXTT wallets and open transactions. Furthermore, a client implementation must react to the transactions accordingly, e.g., send *contest transactions* if a new valid *claim transaction* is processed.

[9]`https://www.blockcypher.com/`

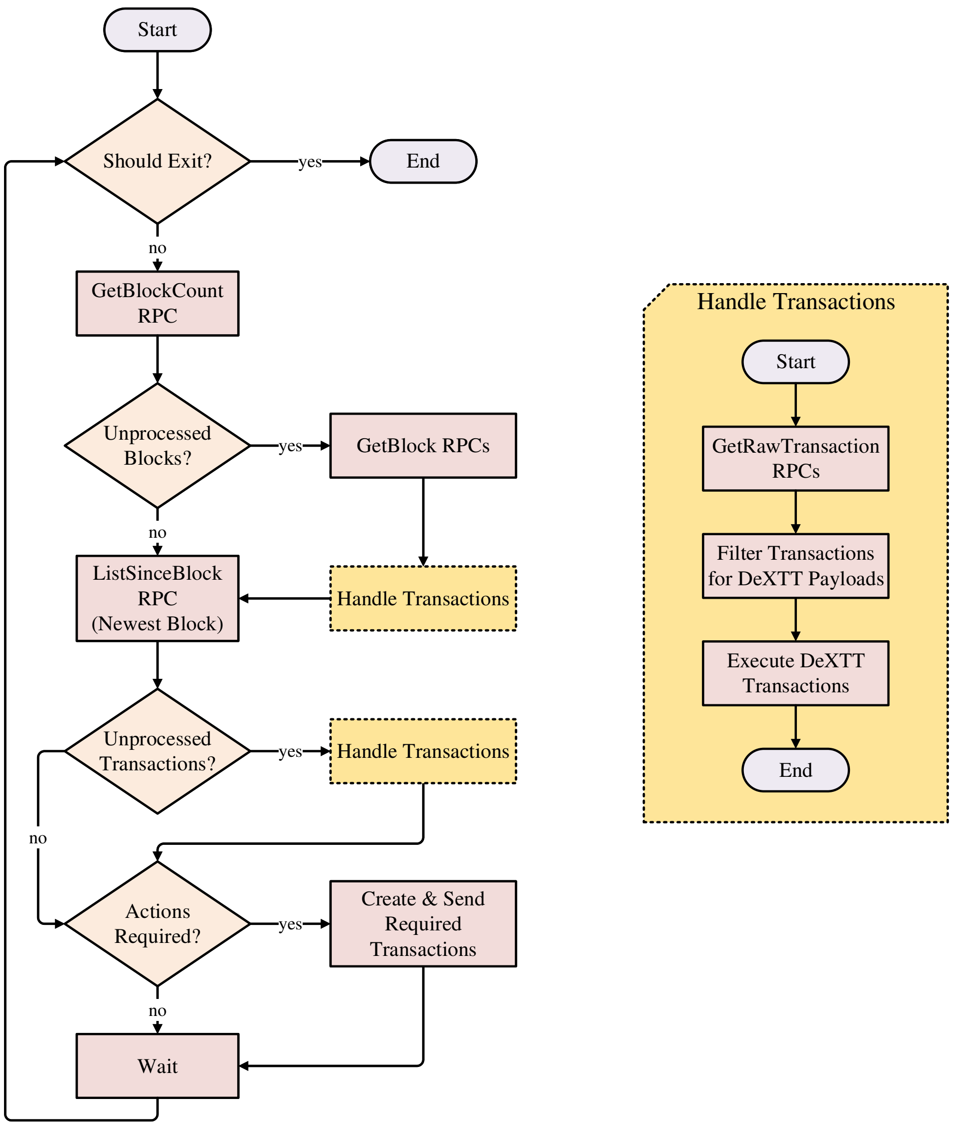

Figure 5.3: Block diagram of interaction between the DeXTT and Bitcoin Core.

Figure 5.3 gives a simplified view on how the client-side application of our DeXTT implementation processes transactions. The shown block diagram of actions represents a loop that is active as long as the client is running. Such a loop is executed for each different blockchain in a multi-blockchain scenario, e.g., by using the regtest mode of Bitcoin Core with multiple blockchains. The body of the loop first checks if a new block was generated, by using the *GetBlockCount* RPC. The returned number of blocks is then compared to the number of already processed blocks to determine, if a new block was created that has to be processed by the client. The number of already processed blocks can be initially set to the genesis block of DeXTT to prevent the scanning of the whole blockchain (see Section 7.1.1). If new blocks were found, each transaction of each block is accessed through the according RPCs. These transactions are all confirmed with at least one confirmation. Every new transaction is then searched for null data outputs that contain DeXTT payloads. If this search yields DeXTT payloads, they are parsed and executed by the client, effectively updating its state.

To also make use of existing transactions that are not yet included in a block, i.e., unconfirmed transactions, the *ListSinceBlock* RPC is utilized to get all transactions since the most recent block. This call therefore only returns unconfirmed transactions. These transactions are then again parsed and executed. Generally, unconfirmed transactions are treated differently by our DeXTT client to still ensure determinism for their execution. More details about the handling of unconfirmed transactions is presented in Section 5.4.

After all new transactions have been processed, the resulting state of the DeXTT protocol is checked to determine if any actions must be taken, e.g., finalizing a token transfer or participating in a witness contest. Details about the logic to check if actions are required are discussed in Section 5.3.3 and Section 5.5. After all necessary actions such as the sending of transactions have been taken, the client waits a specified amount of time before repeating the polling of data from the blockchain. This is done to reduce the communication effort and the Central Processing Unit (CPU) workload. Because the expected time between new blocks in Bitcoin is ten minutes, it is expected that a wait time of a few seconds does not introduce any disadvantages for the client, while still lowering the communication and CPU effort. The lower communication load also reduces the workload of Bitcoin Core, which is utilized for the communication.

The blockchain interaction design presented in this section enables the DeXTT client to deterministically build and maintain the state of DeXTT protocol execution on the given blockchains by using a polling approach via the RPCs provided by Bitcoin Core. By utilizing this state information, the client can act accordingly, e.g., finalizing a transaction after the validity period of a token transfer expired.

5.3 Contest Participation

In this section, different aspects of the participation in witness contests are elaborated. First, the verification process of signatures of Bitcoin transactions is presented in Section 5.3.1. This is required to determine the sender of a transaction, who thereby shows her agreement with the content of the transaction through this signature. In the next section, the concept of regarding claim transactions also as contest participations is discussed. Followed by details on how claim transactions are reused for contest transactions. The last section discusses the introduction of different waiting periods by the client between the observation of a valid claim and the sending of a contest participation.

5.3.1 Contest Sender Verification

The commitment to the content of a Bitcoin transaction and therefore also to the contained DeXTT payload is assumed to be given by the sender of a transaction. In our design, the sender of a transaction is determined through the identity that was used to unlock to input of the Bitcoin transaction through its signature. This identity is given in the form of the public key of the participant, which has to be extracted from the transaction. From this public key, the according address for DeXTT transactions can be constructed. It can then be guaranteed that the according DeXTT transaction inside the Bitcoin transaction originates from the calculated address. This information can be regarded as equivalent to a signature of the DeXTT payload by the same party.

In our approach, we use P2WPKH standard transactions for all transactions that embed DeXTT payloads. P2WPKH transactions include a witness program in its inputs to unlock outputs to be spent. The currently used *version 0* witness program that is present in each input of the transaction consists of two data items, a signature and a matching public key that are used to unlock the referenced transaction output [LLW15].

In our design, the DeXTT client therefore parses the witness program of each input to extract the given public key and therefore determines the sender of each DeXTT transaction. The sender is required to be known for contest participations through both claim and contest transactions (see Section 5.3.2) and for the mint transaction.

The utilized approach only works for P2WPKH transactions that use the current version 0 witness program. For future versions of the witness program, the public key might be included in some other ways. Therefore, an alternative solution to determine the sender of a transaction would consist of extracting the information from the spent transaction output[10]. The downside of that approach is that for each transaction, also the transaction that contains the referenced output must be retrieved from the blockchain, resulting in more communication overhead.

5.3.2 Claim Transaction as Contest Participation

In our DeXTT-Bitcoin design approach, a claim transaction automatically also counts as a contest participation. This is possible because the formal definitions of a claim and a contest transaction (see Section 2.4) contain the same data relevant for the participation, a PoI [Bor+19b]. The same concept is applied in the Ethereum prototype of the DeXTT protocol.

The conceptional merging of the transactions brings the advantage that the receiver of a token transfer who posts the claim on the blockchain does not need to send another separate contest transaction. The contest participation is done through the claim itself. As a result, in contrary to the formal definition of DeXTT, the claim must be posted on all participating blockchains, because otherwise inconsistencies could occur [Bor+19b]. This

[10] `https://bitcoin.stackexchange.com/questions/88526/find-senders-public-key-in-segwit-transaction`

fact does not introduce any disadvantages, because otherwise the contest transactions would have been posted on all blockchain. Furthermore, it can be assumed that the receiver of a transfer wants to participate in the contest anyhow, because otherwise she would miss out on a potential witness reward.

Because a claim is automatically considered as a contest participation, it is possible to design the protocol without an explicitly defined contest transaction. Instead, only claim transactions can be used for both purposes. This brings the disadvantage of posting the same data multiple times per blockchain. Furthermore, each claim transaction consists of three null data outputs, resulting in a higher cost than for a single null data output (see Section 5.1.2).

The concept of the contest transaction that consists only of one null data output, effectively containing only a reference to the claim data in the form of a hash, tackles those drawback of the claim transaction (see Section 5.1.2). In our design, it is possible to use this contest transaction on any blockchain that has already seen a claim transaction containing the data of the PoI. The transaction then has the same effect as posting the claim transaction, participating in the according witness contest. By using this approach, less data is written to the blockchain, resulting in lower cost for the participant.

5.3.3 Waiting Period for Contest Participation

One concept of DeXTT is that the protocol participants do only participate in a witness contest, if they still have a chance to win it. That information can be determined by comparing their own contest signature (see Section 5.1.2) with the signatures of parties that already participate in the contest. This results in a theoretical average of $log_2(n)$ candidates posting contest participations [Bor+19b].

This result would require that each participant knows the signatures of some contestants in advance. To enable such a behavior, in the design of our DeXTT client a waiting period for contest participations is introduced. Instead of posting contest participations immediately after observing a claim on a blockchain, each client waits a random amount of time out of a specified time interval. This potentially results in a variation of when clients want to participate in contests, therefore part of the contestants and their contest signatures are already known by the clients that wait longer than the others. These clients can then first check if they still have a chance to win, as they already have some information about other contestants.

When only confirmed Bitcoin transactions are considered for the execution of DeXTT, clients learn about other transactions and therefore also contest participations only if a new block is generated. Therefore, the waiting period is expressed in the number of blocks to wait when only confirmed transactions are used. Our design also comprises the concept of processing unconfirmed transactions (see Section 5.4). If this concept is utilized, a traditional waiting period can be applied, because clients learn about other transactions as soon as they are sent to the Bitcoin network, no newly generated block is necessary.

5.4 Greedy Approach: Utilizing Unconfirmed Transactions

This section presents the utilized concept of handling and processing unconfirmed transactions, without introducing indeterminism in the execution of DeXTT transactions. Indeterministic execution can generally occur by executing unconfirmed transactions, because they are not yet included in a block and therefore have no timestamp associated with them. For instance, a contest transaction might be executed as an unconfirmed transaction, but is included later in a block that contains a timestamp after the expiration of the validity period of the according PoI. The same transaction will therefore be processed differently after it is included in such a block. The only timing information that can be used for unconfirmed transaction is given by the block timestamp of the last generated block. In our approach, it is assumed that timestamps in blocks are always greater or equal to the timestamp of the previous block, otherwise no meaningful timing reasoning can be made. To provide determinism for transaction execution, the different DeXTT-Bitcoin transactions are treated as follows:

Claim Transaction A claim transaction needs to occur after the finalize transaction of the previous token transfer by the same sender. Furthermore, the start time t_0 of the validity period must have already passed and the end time t_1 must not be reached yet. This means that for the transaction execution time t, $t_0 \leq t < t_1$ must suffice for the transaction to be valid regarding its timing [Bor+19b]. The constraint of $t_0 \leq t$ can be enforced for unconfirmed transactions. Its validity can be achieved by requiring that the timestamp t_{newest} of the newest known block suffices to $t_0 \leq t_{newest}$. Therefore, any newly generated block will have a timestamp greater or equal to t_0. The second timing constraints regarding the end time t_1 of the transaction validity period can not be enforced for the usage of unconfirmed transactions. Although it can be checked if $t_{newest} < t_1$ and therefore it is possible that the transaction gets included in a block before t_1, the next block could still be generated with a timestamp greater or equal to t_1. Therefore, it can not be determined with certainty if a claim transaction is valid considering its timing by only utilizing the unconfirmed claim transaction. The same reasoning about the timings applies for the contest participation, which is automatically included in the execution of a claim transaction in our design approach.

The execution of unconfirmed claim transactions would yield an advantage for clients, because there is more time to react to the claim by sending a contest participation and therefore a higher chance that the contest participation is included in a block with a timestamp before t_1. Therefore, another approach for handling unconfirmed claim transactions is introduced. Instead of executing the claim transaction and therefore changing the state of the DeXTT wallets and data, the clients only use the information to react to the claim by sending contest participations, before the claim is confirmed. This results in still utilizing the advantage of unconfirmed claim transactions, without introducing additional indeterminism.

Contest Transaction The necessary timings to execute contest transactions are the same as for claim transactions in our approach. Therefore, it is not possible to execute contest participations provided by unconfirmed transactions, without introducing the possibility of indeterminism of transactions executions. As a result, we utilize unconfirmed contest transactions solely for the purpose of determining the contest signatures of other contestants before a client sends its own contest participation to the blockchain. This allows the observation of other contestant's transactions without waiting for the generation of new blocks. Details about when contest participations are sent are discussed in Section 5.3.3. Unconfirmed contest transactions are therefore not executed in the client and do not change the DeXTT wallets and other state information.

Finalize Transaction A finalize transaction must occur after the end time t_1 of the corresponding token transfer has expired. This can only be checked for unconfirmed finalize transactions, if the newest block of the blockchain has a timestamp t for which $t_1 \leq t$ holds. However, because a finalize transaction must occur before the claim for another new token transfer by the same sender, an indeterminism of transaction execution might occur, if such a claim is added to a new block before the previous finalize. By executing the finalize as an unconfirmed transaction, the claim is considered valid, whereas without such a premature execution, the claim is considered invalid. Furthermore, executing unconfirmed finalize transactions does not yield additional value. Therefore, we did not implement any means of executing unconfirmed finalize transactions.

Finalize-Veto Transaction The same reasoning as for finalize transactions applies for finalize-veto transactions with regards to the veto contest end time t_{VETO} instead of t_1. Therefore, unconfirmed finalize-veto transactions do not get executed in our DeXTT client.

Mint Transaction The newly introduced mint transaction that is utilized to create tokens for testing means, does not introduce any additional timing constraints. Due to this and the fact that it is only utilized for testing purposes, unconfirmed mint transactions get executed instantly within our client.

The handling and execution of unconfirmed transactions can also be designed in a way that eagerly executes every observed unconfirmed transaction. That could result in inconsistencies and indeterminisms. Therefore, there must be a way of undoing already executed transactions and reevaluate the correct state of the DeXTT client if such transactions are included in blocks in a way that do not fulfill the presumptions made by the unconfirmed execution. This alternative approach therefore introduces an implementation overhead together with the introduction of potentially temporary wrong state information. As a result, we decided to design the Bitcoin DeXTT client-side application without eager execution of unconfirmed transactions.

5.5 Multi-Blockchain Client Logic

This section gives an overview of how our DeXTT client is designed with regards to its logic for handling multiple Bitcoin blockchains that are used for the DeXTT protocol. Such a multi-blockchain scenario can be achieved, e.g., by utilizing the regtest mode of Bitcoin core. The client-side application of the DeXTT DApp must therefore consider the different timings of DeXTT transactions on multiple blockchains. An important factor for the design of how the client reacts to different events are the different timings across the blockchains. Generally, blocks are not generated at the same time on all blockchains. Additionally, the timestamps across the blockchains may differ. The only assumption that is made about a timestamp t that is included in a block, consists of the expectation that for each block i and its subsequent block $i+1$ the constraint of $t_{b_i} \leq t_{b_{i+1}}$ holds, i.e., the timestamps progress in a monotonic manner.

Timings and therefore the execution of transactions differ across the different blockchains. Therefore, the logic of when to react to the current DeXTT state and send according transactions, e.g., participate in a contest, is handled considering a global view of the blockchain ecosystem. For this, the timestamps of blocks of all participating blockchains are taken into account to determine if certain actions should be taken. This is done to prevent inconsistencies across the blockchains, e.g., by sending contest participations that are not valid on all blockchains, due to time constraints of the corresponding PoI.

In Section 5.5.1, definitions for two concepts of global time constraints are presented. The subsequent sections discuss approaches to handle client logic in regards to those global timing concepts.

5.5.1 Global Timings Across Blockchains

To handle different timing constraints within the logic that determines when to send which transactions, two concepts of global timing information across the blockchains are introduced. Figure 5.4 illustrates both global time concepts by an example using three blockchains, C_a , C_b and C_c. The block B_{a_i} represents the i^{th} block on blockchain C_a. In this example, the timestamp of block B_{b_i+1} represents the *earliest blockchain time* $t_{earliest}$ and the timestamp of block B_{c_i+4} represents the *latest blockchain time* t_{latest}. The global time concepts are further discussed below.

Latest Blockchain Time

The concept of the *latest blockchain time* is needed to check if a certain point of time t has been reached on any of the participating blockchains, e.g., to check if a claim transaction might be already invalid on any of the blockchains before sending it. Therefore, the client can decide to not send the transaction to any of the blockchains, effectively preventing a certain inconsistency across the blockchains.

The *latest blockchain time* is defined to be represented by the timestamp t with the latest value of all timestamps across all blockchains, i.e., the timestamp of the most recently

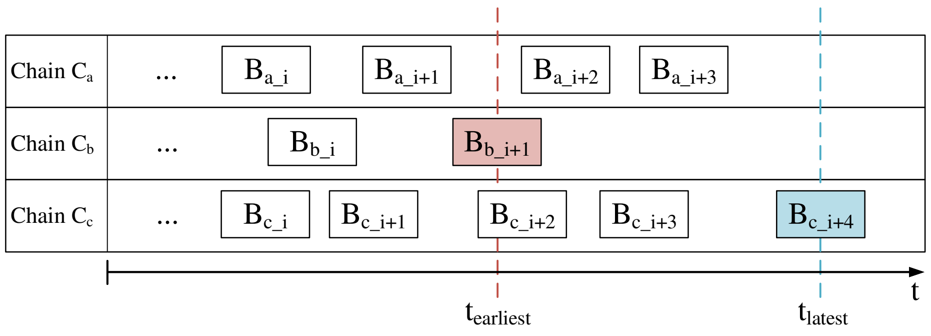

Figure 5.4: Global timings across multiple blockchains.

created block out of all blockchains. Therefore, for all blocks b with the timestamp t_b, $t_b \leq t$ holds.

Earliest Blockchain Time

The concept of the *earliest blockchain time* is required to check if a point in time t has passed on all participating blockchains, e.g., to check if the validity of the start time of a PoI has been reached on all blockchains and therefore the according claim transaction does not end up invalid due to its start time on any of the blockchains.

The *earliest blockchain time* is defined as the latest timestamp t for which it holds that each blockchain has blocks with greater or equal timestamps. Therefore, it can be regarded as the timestamp of the blockchain with the globally earliest block as its latest block. Therefore, t is the latest timestamp of any block for which it holds, that on any blockchain there exists a block b with the timestamp t_b such that $t \leq t_b$.

5.5.2 Initial Claim Transactions

Before a receiver of a token transfer sends a claim transaction based on the given PoI, the time constraint of the start time of the validity period of the transfer must be checked first. Otherwise, there exists the possibility to introduce inconsistency across blockchains. This can happen, if on one blockchain the claim transaction gets included in a block with time $t < t_0$ whereas on another blockchain, the same transaction is included in a block with $t \geq t_0$.

To comply with the time constraint regarding t_0, for the *earliest blockchain time* t_e it must hold that $t_e \geq t_0$ when the initial claim transaction of a token transfer is sent to the blockchains. This infers that the claim transaction is included in a block with timestamp $t \geq t_0$ on all blockchains.

5.5.3 Finalize and Finalize-Veto Transactions

For the logic to determine if a finalize or finalize-veto transaction has to be sent due to the expiration of the according validity period, an additional assumption is introduced. It is expected that if on one blockchain, a block with a timestamp t_i is generated, the next block generated by any participating blockchain has a timestamp t_{i+1} such that $t_i \leq t_{i+1}$. This assumption holds for, e.g., the regtest mode of Bitcoin Core, where newly generated blocks are created with the current local time as their timestamp. Furthermore, if the client is used solely on the mainnet or testnet of Bitcoin, this assumption does not introduce any drawbacks, as not more than one Bitcoin blockchain is used.

After each iteration of transaction executions (see Section 5.2.2), the DeXTT client checks if any token transfer needs to be finalized or veto-finalized by comparing the according end time t_{end} to the current *latest block time* t_l. For each not yet finalized token transfer for which $t_{end} < t_l$ holds, the according finalize transactions are then sent to each blockchain.

By using the *earliest blockchain time* instead of the *latest blockchain time*, the additional timing assumption could be dropped, but it would take a longer time until finalize transactions could be sent, waiting for each blockchain to generate a new block first.

5.5.4 Contest Participations

If a client receives a claim transaction on any blockchain for a new token transfer, it first waits for a certain amount of time as discussed in Section 5.3.3. After the waiting period, the validity end time t_1 is compared to the *latest blockchain time* t_l. Only if $t_l < t_1$, the DeXTT client sends its contest participation to all blockchains.

This is done due to possible inconsistencies across blockchains if there exists a block on any blockchain B with a timestamp t_B greater than t_1. This could potentially result in the contest transaction being treated as valid on all blockchains except on blockchain B, yielding a different witness contest winner on B and therefore being inconsistent with the other blockchains.

CHAPTER 6

Implementation Details

This chapter presents details about the concrete implementation of the DeXTT protocol on the Bitcoin blockchain. The focus of this chapter lies on specific architectures and concepts that are used within the implementation. The source code of the DeXTT Bitcoin prototype that was created in the scope of this thesis is freely available as Open Source software on Github[1].

The chapter is started by the presentation of the used technologies in Section 6.1. The subsequent Section 6.2 gives an overview about the general architecture of the prototype implementation. Section 6.3 elaborates how claim transactions are assembled from the three DeXTT-Bitcoin claim transaction parts.

6.1 Technology Stack

This section presents the utilized technologies for the client-side application of the DeXTT-Bitcoin implementation. For the communication with the Bitcoin blockchain, the *Bitcoin Core* [bit20b] client software is used as described in Section 5.2. We decided to implement the Bitcoin prototype of DeXTT using Java together with several Java libraries that enable us to realize the client-side application efficiently while applying best practices of software engineering. The details of all used technologies within the design and implementation of DeXTT for Bitcoin are listed below.

Bitcoin Core. As described in detail in Section 5.2, the client-side utilizes the Bitcoin Core [bit20b][2] client software to communicate and interact with the Bitcoin blockchain. For our implementation, Bitcoin Core version 0.19.0.1 is employed.

[1]`https://github.com/MatthiasKuehne/dexxt-bitcoin-prototype`
[2]`https://bitcoincore.org/`

Java. The DeXTT client is developed using the Java language with the Java SE 12.0.1 Oracle JDK[3]. Java was chosen because it allows object-oriented programming, provides a big community, many third-party libraries and can be run on any system that runs the Java VM.

Maven. To manage the different dependencies for third-party Java libraries and the build lifecylce of the client implementation, Apache Maven[4] 3.6.1 is used. Maven automatically loads the specified libraries and includes them in the software project and the build artifacts.

log4j. Apache log4j[5] is a logging framework that enables global settings for the format, output stream and categories for different log messages. It is used in version 2.12.1 within the DeXTT client.

bitcoin-rpc-client. This library is used to access the JSON-RPC interface of Bitcoin Core from the Java client implementation. It offers wrapper functions to access the API of Bitcoin Core and therefore takes care of parsing, type conversions and the communication via HTTP. There exist several Java libraries to offer similar functionality [bit20a]. We choose the bitcoin-rpc-client[6] in version 1.1.1 because it offers wrappers for all utilized RPCs and is still actively developed.

Web3j. Web3j[7] that is used in version 4.5.14 is a Java library to work with smart contracts and blockchains such as Ethereum. In our implementation, it is utilized for cryptographic functions such as signature creation and verification, the Keccak-256 hash function and the derivation of Ethereum addresses from public keys.

bitcoinj. The bitcoinj[8] 0.15.6 library offers means to work with the Bitcoin protocol. Although it includes sophisticated features such as maintaining a Bitcoin wallet or sending and receiving transactions, in our implementation it is only used to create a public/private key pair from a given Wallet Import Format (WIF) private key [bit17b] as it is returned by the *DumpPrivKey* RPC of Bitcoin Core.

Picocli. Picocli[9] 4.1.4 is included in the DeXTT client to provide an efficient way within the application source code to offer a Command Line Interface (CLI). Annotations are used to specify commands and the usage help for the interface is automatically generated by this library.

Guava. Guava[10] by Google offers a wide variety of core libraries to enhance the functionality provided by Java. Within the DeXTT implementation, Guava 28.2-jre is

[3]https://www.oracle.com/java/
[4]https://maven.apache.org/
[5]https://logging.apache.org/log4j/2.x/
[6]https://github.com/Polve/bitcoin-rpc-client
[7]https://www.web3labs.com/web3j
[8]https://bitcoinj.github.io/
[9]https://picocli.info/
[10]https://github.com/google/guava

included for the usage of its event bus implementation[11]. Such an event bus offers publish-subscribe communication between different components without explicitly registering to each other and therefore allows loosely coupled components.

JUnit. JUnit[12] 5.6.0 is used as a testing framework. The library offers functionality to write unit tests for the DeXTT client application.

AssertJ. The AssertJ[13] 3.15.0 Java library provides a rich set of assertions that are used for writing unit tests.

6.2 Software Architecture

This sections gives an overview of the software architecture of the DeXTT-Bitcoin prototype.

The client implementation is designed to not only feature all details that are needed to run the DeXTT protocol together with the Bitcoin blockchain via Bitcoin Core, but also includes all necessary code to excessively run the software for testing and evaluation means. Because this prototype is currently solely utilized for such evaluation means, the actual core to run DeXTT including all logical details is wrapped by parts that are responsible to execute evaluation runs. Therefore, our implementation represents an all in one solution that bundles the DeXTT protocol execution together with the evaluation runs.

This allows a simpler design of the overall system to evaluate the DeXTT implementation. Another approach would be to provide an extensive interface by a DeXTT protocol application and utilize that interface by a separate software artifact that executes the evaluation runs. This approach comes with a decent implementation overhead and is not necessary in the context of this thesis.

The DeXTT client application is structured into different functional responsibilities through the introduction of various packages. The Java packages that make up the implementation are described below.

6.2.1 Communication.Bitcoin Package

The `Communication.Bitcoin` package only contains one Java class called `Bitcoin-Communicator` that is responsible to handle all direct communication with Bitcoin Core through the API calls that are offered by the *bitcoin-rpc-client* library. The class offers functions that can be utilized to send a DeXTT payload to the blockchain or to retrieve all DeXTT transactions of specified blocks. Therefore, the `BitcoinCommunicator` yields all implementations to build and embed data into Bitcoin transactions. In addition, it includes all relevant code to parse Bitcoin transactions and find DeXTT

[11]`https://github.com/google/guava/wiki/EventBusExplained`
[12]`https://junit.org/junit5/`
[13]`https://assertj.github.io/doc/`

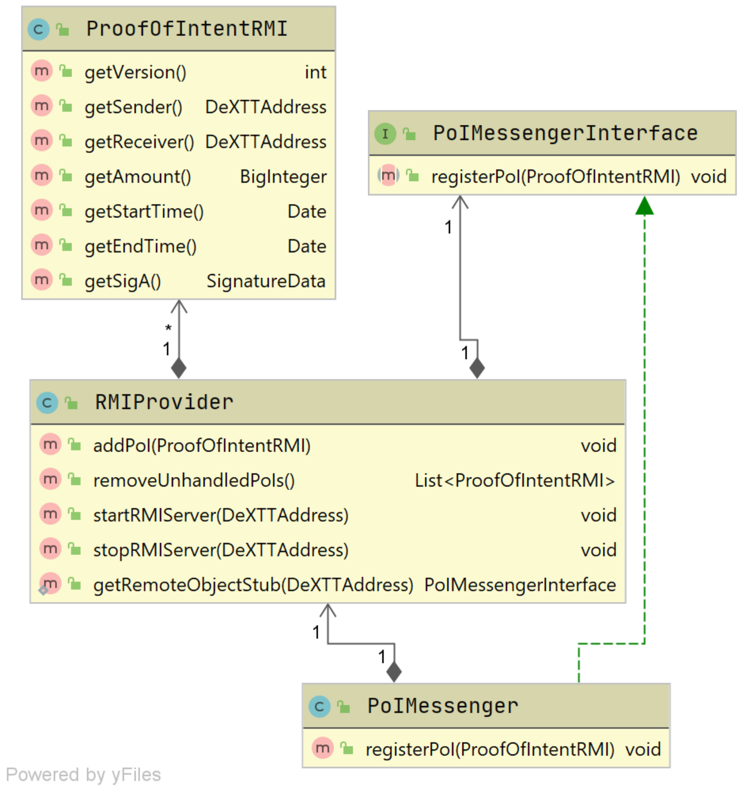
ProofOfIntentRMI
getVersion() int
getSender() DeXTTAddress
getReceiver() DeXTTAddress
getAmount() BigInteger
getStartTime() Date
getEndTime() Date
getSigA() SignatureData
PoIMessengerInterface
registerPoI(ProofOfIntentRMI) void
RMIProvider
addPoI(ProofOfIntentRMI) void
removeUnhandledPoIs() List<ProofOfIntentRMI>
startRMIServer(DeXTTAddress) void
stopRMIServer(DeXTTAddress) void
getRemoteObjectStub(DeXTTAddress) PoIMessengerInterface
PoIMessenger
registerPoI(ProofOfIntentRMI) void
Powered by yFiles

Figure 6.1: Communication.RMI Package.

payloads and corresponding meta-data such as the public key of the sender of the transaction. All relevant data that is parsed from Bitcoin transactions is packed into a `RawBitcoinTransaction` for each DeXTT payload.

6.2.2 Communication.RMI Package

The `Communication.RMI` package includes all necessary implementation details for the usage of the Java Remote Method Invocation (RMI) API [Ora19f]. We use RMI for the off-chain sharing of the sender's intent of a DeXTT token transfer. In contrast to the Ethereum prototype[14], the data of the sender's intent is not shared via the used blockchain. The usage of RMI allows each running client to call methods of an exposed interface of other clients. This allows the sending of a sender's intent as an call argument to such an exposed method.

The structure of the classes and methods inside the `Communication.RMI` package are shown in Figure 6.1. The `ProofOfIntentRMI` class is used to encapsulate all data of a sender's intent. The `PoIMessengerInterface` interface is exposed by all clients

[14]`https://github.com/pantos-io/dextt-prototype`

to allow the calling of the `registerPoI` method, which is used to transfer the data. The `PoIMessenger` implements this interface by saving the received data inside the `RMIProvider` class. In addition to storing all received RMI data, the `RMIProvider` class also provides all required functionality to start and stop the RMI server. The currently pending received sender's intents are processed by the client by polling the `removeUnhandledPoIs` method regularly.

Because the actual approach to share the data of a sender's intent is not specified in the DeXTT protocol, there exist also other means to achieve this off-chain data transfer besides the usage of Java RMI. One possible alternative approach would be to utilize network transfers by using sockets to establish a raw communication link between the DeXTT clients [Ora19g]. Compared to our RMI approach, the usage of sockets introduces additional implementation overhead, because more details of sending and receiving data have to be dealt with. Additionally, it does not offer a real benefit over the RMI implementation. Although sockets allow for more control over the communication and server parts, it is also possible to introduce new errors within a socket implementation.

6.2.3 Configuration Package

The `Configuration` package bundles all relevant parts that are used to represent the current configuration and constant data parts of the DeXTT client. Inside the `ConfigCommand` class, all logic and functionality to provide the CLI for the application are included by utilizing the *Picocli* library. The current configuration data given by the command line call of the program is then saved inside the `Configuration` class, which is globally available as a *singleton* object [Gam+94]. Details about the possible configurations via the CLI are presented in Chapter 7.

Furthermore, all data that can be considered constant, meaning that it stays the same for all application runs of the clients, are available through the `Constants` class. This constant data includes among others the byte values for the different DeXTT-Bitcoin transaction types (see Section 5.1.2).

6.2.4 DeXTT Package

Inside the `DeXTT` package, all relevant parts for the execution of the DeXTT protocol are contained. Its sub-packages are presented in the following sections. Figure 6.2 shows the classes and sub-packages of the `DeXTT` package without details about their public interface to reduce the complexity of the illustration. The `BitcoinParser` class provides the functionality to parse any given DeXTT payload, given as a `RawBitcoinTransaction` and to create the according parsed version of the DeXTT-Bitcoin transaction called a `BitcoinTransaction` (see Section 6.2.8). Inside the `Cryptography` class, all necessary cryptographic functionality of the application is included, such as signature creation and verification. The `Helper` class is a collection of small functions that are used throughout the DeXTT client. Most of these functions provide means to copy data into byte buffers.

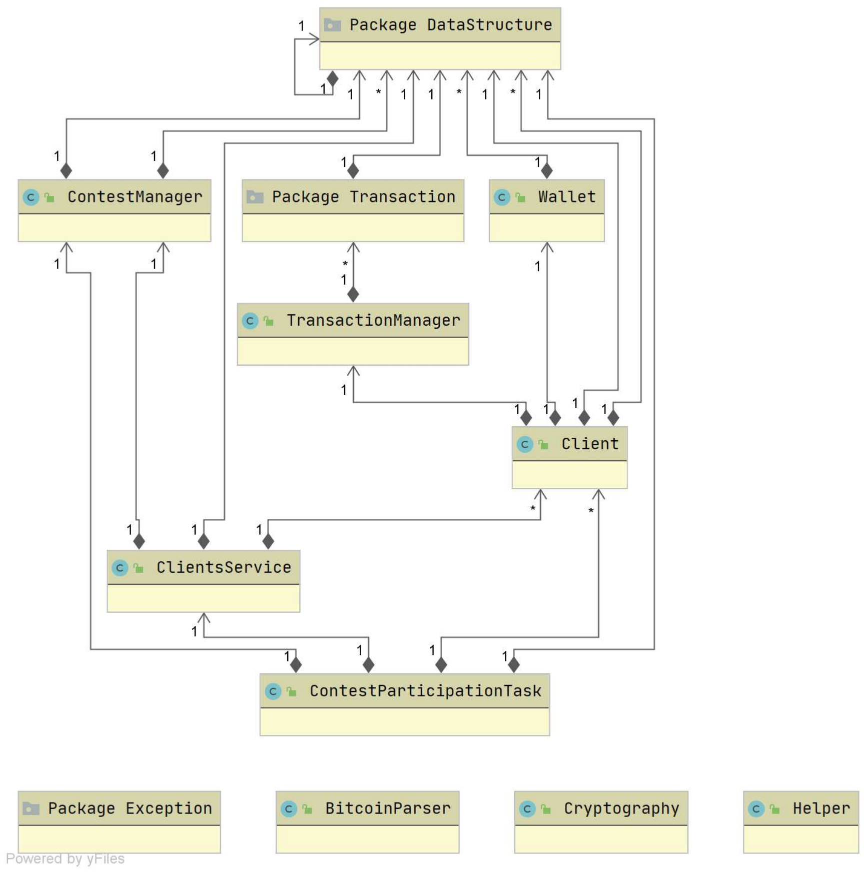

Figure 6.2: DeXTT Package.

The class called `ClientsService` represents a service that contains the multi-blockchain logic as described in Section 5.5. It contains the method `loopIteration` which represents one iteration of the loop for the DeXTT logic and execution (see Figure 5.3 in Section 5.2.2). This function should therefore be called repeatedly to achieve the polling approach of the main loop of the DeXTT client (see Section 6.2.10). In addition to polling blockchain data and applying appropriate actions, it also checks if any new sender intents were received via RMI (see Section 6.2.2) and sends the according claim transactions if necessary. To maintain a global state about ongoing DeXTT witness contests and their end times and participants, the `ContestManager` class is utilized. If a contest participation is required by the rules of the multi-blockchain logic, a new instance of the `ContestParticipationTask` is created and scheduled to run and send the participation after a random wait time through a `ScheduledExecutorService` [Ora19d].

The `ClientsService` also encapsulates an instance of the `Client` class for each participating Bitcoin blockchain. The `Client` handles all data that are specific to one blockchain in the ecosystem for which the instance is responsible. It also controls the communication to the blockchain via a `BitcoinCommunicator` instance and is responsible to process blocks and transactions of the specified blockchain. For managing the state of processed transactions for the blockchain, a `TransactionManager` instance is used. More details on the usage of the `TransactionManager` are given in Section 6.3. The actual execution of DeXTT transactions and the handling of the DeXTT protocol state is done through an instance of the `Wallet` class for each blockchain. The `Wallet` class is designed to represent and mimic the `PBT.sol` smart contract of the Ethereum implementation[15]. It therefore stores DeXTT state such as wallet balances or current winners of ongoing witness contests. The `Wallet` class triggers events (see Section 6.2.9) through the event bus provided by the *Guava* library for each relevant successful change of DeXTT state, such as a new contest participation. There are multiple event buses used within the application: One global event bus to forward events to the global state managing class `ClientsService` and one additional event bus for each blockchain to deliver events from the `Wallet` of a specific blockchain to the corresponding `Client` instance.

6.2.5 DeXTT.DataStructure Package

Inside the `DeXTT.DataStructure` package, DeXTT-specific data structures are defined. Theses additional types include a `DeXTTAddress` class that is based on the Ethereum address type of the *Web3j* library and several Data Transfer Objects (DTOs), e.g., to encapsulate data of PoIs or elements that are used within a `SortedSet` [Ora19e].

6.2.6 DeXTT.Exception Package

Additional checked exceptions [Ora19a] to add more semantics to thrown exceptions within the DeXTT execution are specified inside the `DeXTT.Exception` package. These exceptions include among others the `FullClaimMissingException`, which is thrown if a contest transaction is tried to be executed without a previous claim transaction on the given blockchain.

6.2.7 DeXTT.Transaction Package

The `DeXTT.Transaction` package contains classes to represent the different types of transactions of the DeXTT protocol. Its sub-package `DeXTT.Transaction.Bitcoin` is described in the following sections. The structure of the various DeXTT transaction classes is shown in Figure 6.3. The abstract class `HashReferenceTransaction` is introduced to bundle parts of the implementation of contest and finalize transactions, which both utilize references to a PoI through a hash (see Section 5.1.2). All transactions implement the `Transaction` interface, which offers all necessary functionality to handle transactions. The concrete transactions overwrite the methods of the interface accordingly

[15]`https://github.com/pantos-io/dextt-prototype/blob/master/truffle/contracts/PBT.sol`

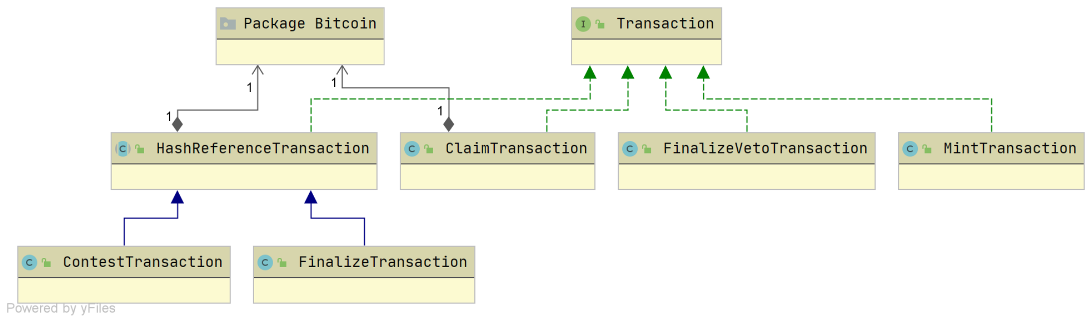

Figure 6.3: DeXTT.Transaction Package.

to enable their usage through dynamic binding [Boo07] by only utilizing functionality provided by the interface.

A transaction is executed calling its `tryToExecute` method, which takes a `Wallet` instance as its argument. This enables the execution of all transaction types through the same method call of their interface. The transaction implementations choose the according method of the provided `Wallet` themselves. By using this dynamic binding approach, the caller, in this case a `Client` instance, does not have to know and differentiate the type of the transaction to be executed. This differentiation is handles by the dynamically bound method implementation.

By using an approach, where dynamic binding is not used in such a way, additional implementation overhead to execute transactions is required inside the `Client`. Such an approach could be to store different types of transactions individually or by checking the concrete type of a transaction and casting its type to use a method of the transaction implementation explicitly. Both of those ways of handling transactions introduce unnecessary overhead.

6.2.8 DeXTT.Transaction.Bitcoin Package

Inside the `DeXTT.Transaction.Bitcoin` package, the different types of DeXTT-Bitcoin transactions as defined in Section 5.1.2 are represented through different classes. The structure of these classes is shown in Figure 6.4. The abstract classes `BitcoinHash-ReferenceTransaction` and `BitcoinClaimTransaction` are introduced to bundle functionality that is required by multiple transaction implementations. The `Raw-BitcoinTransaction` class represents a DeXTT-Bitcoin payload that is extracted from a Bitcoin transaction. It contains the raw bytes of the payload together with metadata. Such raw transactions are parsed by the `BitcoinParser` to create an according `BitcoinTransaction`. The `BitcoinTransaction` interface is implemented by all DeXTT-Bitcoin transactions and provides methods to build corresponding DeXTT transactions from them or to convert them back into a raw DeXTT payload to be included in a Bitcoin transaction. Again, dynamic binding [Boo07] is applied to utilize those con-

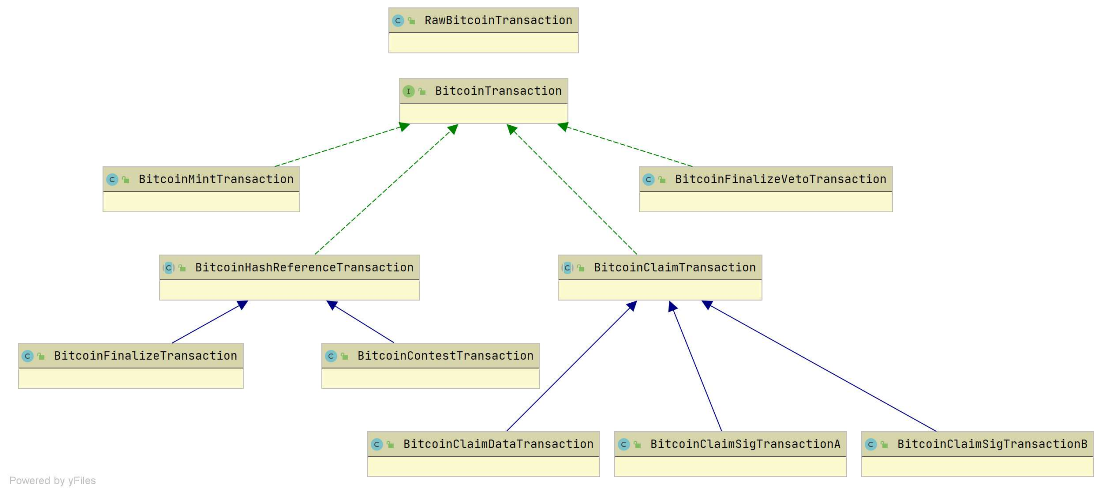
RawBitcoinTransaction
BitcoinTransaction
BitcoinMintTransaction
BitcoinFinalizeVetoTransaction
BitcoinHashReferenceTransaction
BitcoinClaimTransaction
BitcoinFinalizeTransaction
BitcoinContestTransaction
BitcoinClaimDataTransaction
BitcoinClaimSigTransactionA
BitcoinClaimSigTransactionB
Powered by yFiles

Figure 6.4: DeXTT.Transaction.Bitcoin.

version methods without the need to differentiate between the different DeXTT-Bitcoin transaction implementations inside the caller. More details about the conversion to DeXTT transactions are given in Section 6.3.

6.2.9 Events Package

The `Events` package contains classes to represent different event types and also defines a globally usable event bus as a *singleton* object [Gam+94] represented by the `GlobalEventBus` class. The events themselves are built as DTOs, only encapsulating the required data to handle the according event. Furthermore, events are differentiated by the utilized *Guava* library simply by the type of object that is handled in the event. Therefore, for each individual event, a different type must be utilized. The class definitions include events for:

1. contest participations,
2. started witness contests,
3. new processed blocks within a main loop iteration,
4. finalized token transfers,
5. new unconfirmed DeXTT-Bitcoin claim transactions,
6. new unconfirmed DeXTT-Bitcoin contest transactions,
7. started veto contests and
8. finalized veto contests.

All these events are triggered within the `Wallet` class. Each event is sent to the global event bus to be processed by the cross-blockchain client logic. In addition, the events are also transmitted to the according event bus of the blockchain to which the `Wallet` instance corresponds and can be processed by the `Client` instance that handles the state of the given blockchain.

Another approach of distributing such event data could be achieved by manually calling methods of all event listeners providing the relevant data as arguments. This would require the emitter of events to know all listeners for all its emitted event types and therefore introduces an implementation overhead and additional coupling between the different components of the application. To counteract theses drawbacks, an event bus is utilized for such communication means. To further reduce the implementation effort and the potential introduction of errors, the event bus implementation of the *Guava* library is used.

6.2.10 Runners Package

The `Runners` package contains classes that are used to run the application. The `EvaluationRunner` class simulates a client for the implementation evaluation by repetitively sending token transfers to other participants. More details about the evaluation are given in Chapter 7. The `BlockGenerateRunner` runs the application in block generation mode for the regtest mode of Bitcoin Core. It therefore only periodically generates new blocks on the participating blockchains. The `MintRunner` sends a specified mint transaction to all blockchains to create tokens that can be used for the evaluation runs.

6.3 Matching DeXTT-Bitcoin Claim Transactions

This section shows how different *DeXTT-Bitcoin claim transactions* are matched by their included hash values (see Section 5.1.2) to form an executable claim transaction.

The matching of different claim transaction parts is not as trivial as putting together transactions with the same hash value. The main problem for such a simple approach is the fact that this hash value can only be considered as a way to simplify the matching. It is not secured in any way, meaning that transactions can possibly contain any byte sequence as the hash value, e.g., if a malicious party sends a wrong hash value in a transaction intentionally. Another possibility is the occurrence of a hash collision, where two different PoIs happen to have the same 8 byte hash value. In both cases, the hash values of unrelated claim transaction parts match. The wrongful matching can only be detected by validating the signatures contained in the *DeXTT-Bitcoin Claim Signature* α and *DeXTT-Bitcoin Claim Signature* β transactions.

Due to the data used for creating the signatures α and β in a claim transaction, the α signature can only be verified via the data provided by a *DeXTT-Bitcoin claim data* transaction. The verification of the β signature requires to know the α signature, as α is used to create the β signature. This introduces dependencies between the three parts of the claim transactions that have be verified and matched, e.g., a *DeXTT-Bitcoin claim data* and a *DeXTT-Bitcoin Claim Signature β* can can only be matched by their included hash values, because the verification through the β signature is not possible without knowing the corresponding α signature.

In our implementation, storing incomplete claim transactions is done through an instance of the `TransactionManager` class that is contained within the `Client` instance of the corresponding blockchain. After a DeXTT-Bitcoin transaction has been parsed within the `processRawTransaction` method in the `Client`, the `TransactionManager` is queried for already processed incomplete transactions that have the same hash value as the newly parsed transaction. If the returned list of transactions is not empty, it is tried to match the newly parsed DeXTT-Bitcoin transaction with the already created incomplete DeXTT transactions.

The matching process is illustrated in a simplified form in Figure 6.5. In our implementation design, all transactions are only known by the type of their interface, not the concrete transaction implementation. Therefore, we utilize a dynamic binding approach [Boo07] to still match transactions correctly according to their type. As shown in Figure 6.5, the list of transactions with the same hash value is given as an argument to the `putIntoDeXTTTransaction` method of the newly parsed `BitcoinTransaction` instance. The type of the `BitcoinTransaction` is not known within the `Client`. The `putIntoDeXTTTransaction` method is given as a default method [Ora19b] inside the `BitcoinTransaction` interface, therefore a dynamically bound method named `tryToAddDeXTTBitcoinTransaction` is called for each transaction within the transaction list. Inside this method, the current object of a concrete type of the `BitcoinTransaction` is given through the `this` keyword [Ora19h]. The keyword is used as a parameter to the `tryToAddDeXTTBitcoinTransaction` method of the current `Transaction` that is used to try to match the `BitcoinTransaction`. This method is overloaded [Ora19c], therefore the concrete method implementation is defined by the type of the given argument. This allows to differentiate between different types of `BitcoinTransaction` instances.

The `tryToAddDeXTTBitcoinTransaction` method then performs the actual matching of transactions utilizing signature verifications and returns the matched transactions. It is necessary to allow the returning of multiple transactions, because it is possible that a split of a currently incomplete claim transaction is required. This can happen if it contains the data part and β signature part of the claim, which are added to the same transactions if their hash matches, without verifying any signatures. If in such a case, the according α signature transaction is tried to be matched, it is possible that the signature verification utilizing α is only valid for one of the two parts, therefore requiring the splitting of the claim into two separate claim transactions. If no match is found for a new `BitcoinTransaction`, a new `Transaction` is created from it. All matched or created transactions are then returned to the calling `Client`, where all newly completed transactions are executed.

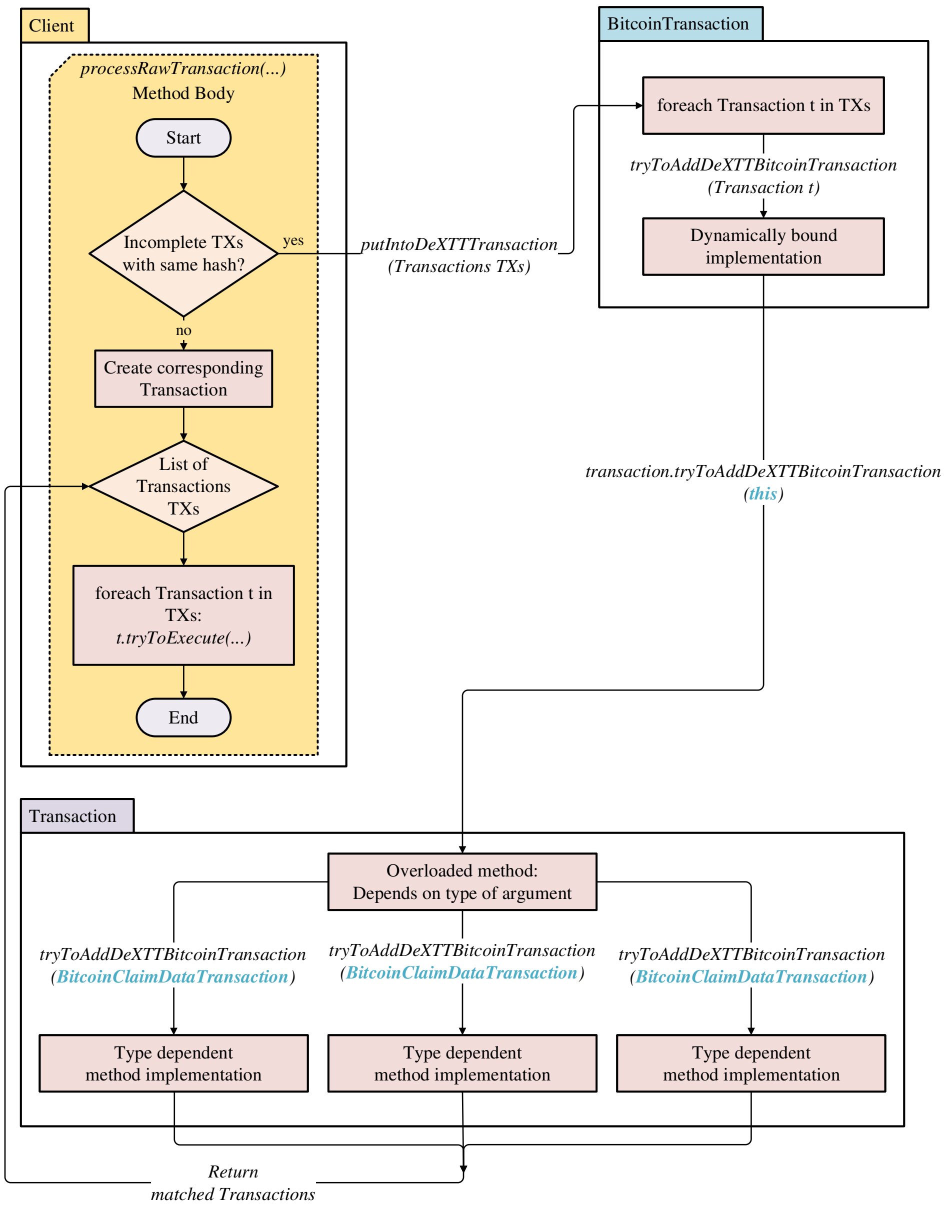
Client
processRawTransaction(...)
Method Body
Start
Incomplete TXs with same hash?
yes
no
Create corresponding Transaction
List of Transactions TXs
foreach Transaction t in TXs: t.tryToExecute(...)
End
putIntoDeXTTTransaction (Transactions TXs)
BitcoinTransaction
foreach Transaction t in TXs
tryToAddDeXTTBitcoinTransaction (Transaction t)
Dynamically bound implementation
transaction.tryToAddDeXTTBitcoinTransaction (this)
Transaction
Overloaded method: Depends on type of argument
tryToAddDeXTTBitcoinTransaction (BitcoinClaimDataTransaction)
tryToAddDeXTTBitcoinTransaction (BitcoinClaimDataTransaction)
tryToAddDeXTTBitcoinTransaction (BitcoinClaimDataTransaction)
Type dependent method implementation
Type dependent method implementation
Type dependent method implementation
Return matched Transactions

Figure 6.5: DeXTT-Bitcoin claim transaction matching.

CHAPTER 7

Evaluation

This chapter presents the evaluation of the DeXTT-Bitcoin implementation described in the preceding chapters. The evaluation includes a quantitative analysis of the required DeXTT transfer validity period and the actual execution cost on the blockchain. The required data for the evaluation analysis is gathered through the usage of the private blockchains offered by the regtest mode of Bitcoin Core (see Section 2.2.7). This approach allows a high level of reproducibility and controllability for the evaluation runs and does not introduce any additional cost other than for the hardware on which the blockchains and the DeXTT clients are executed.

The first section gives an overview of the different running modes and CLI parameter of the DeXTT client implementation. The succeeding Section 7.2 presents the computational environment that is used to run the evaluation of the client application together with the different run configurations. Section 7.3 presents the analysis of the evaluation runs using multiple private blockchains through the regtest mode of Bitcoin Core. In Section 7.4, details of a single evaluation run of the DeXTT client on the Bitcoin testnet are presented. The last part of this chapter is presented in Section 7.5, including the comparison of the evaluation with the Ethereum implementation[1] of DeXTT.

7.1 Client Application Modes

To enable a seamless evaluation of the DeXTT-Bitcoin client design and the DeXTT protocol on Bitcoin, the client implementation is built to support the configuration of evaluation runs through a CLI. Through this interface, the mode of the application run can be chosen and all relevant parameters are set. The running of the evaluation is bundled together with the core DeXTT-Bitcoin implementation, yielding one software

[1]https://github.com/pantos-io/dextt-prototype

application that includes all functionality. It is currently designed to only support its usage for evaluation runs.

For the evaluation runs, the client application supports multiple running modes, each designed to handle a different task (see Section 6.2.10) and each featuring its own set of CLI parameters. The relevant parameter that are required to be provided to the CLI for all modes and the subcommands to choose the application mode are presented below:

Client Address. The address that is specified via the `-a` or `--address` option represents the Bitcoin address of the client. This address is used for all communication with the Bitcoin Core API. The private/public key pair that is utilized throughout the DeXTT client application is the one that corresponds to that Bitcoin address. Its private key is requested by the *DumpPrivKey* RPC.

Blockchain Mode. The used Bitcoin blockchain is specified by the `-c` or `--chain` option. The three values `REGTEST`, `TESTNET` and `MAINNET` are allowed. This option is only relevant for internal calculations regarding Bitcoin addresses and keys. The actual blockchains that are used for the DeXTT protocol are specified by their according Uniform Resource Locator (URL) of the Bitcoin Core RPC interface as described below.

Bitcoin Core RPC URLs. The `-u` or `--urlrpc` option is used to define the URLs that are used to access the RPC interface of Bitcoin Core. At least one URL must be specified. If multiple URLs are given as arguments to the client, each of the given URLs is considered to belong to a different instance of Bitcoin Core and therefore to different blockchains. This enables the specification for multi-blockchain evaluation runs. An URL to access the Bitcoin Core API must be formed as `http://<user>:<pw>@<host>:<port>/`, whereas `<user>` and `<pw>` are used for authentication means and `<host>` and `<port>` specify the location of the API. All of these values can be configured in a *configuration file* for Bitcoin Core called `bitcoin.conf` [bit19f].

Subcommands. To choose a running mode for the DeXTT client application, an according subcommand must be given to the CLI. The subcommand must be one out of `evaluationrun`, `generateblocks` or `mint`. The former two subcommands are described in more detail in the following sections. The `mint` subcommand enables the minting of tokens (see Section 5.1.2) for use by the DeXTT protocol by sending the according transactions to the specified blockchains.

7.1.1 Evaluation Mode

The evaluation mode of the DeXTT client implementation is specified by the subcommand `evaluationrun` of the CLI. It is used to execute evaluation runs by simulating a client that is sending and receiving tokens. Such an evaluation run includes the execution of the DeXTT protocol as described in Chapter 5, together with the periodical creation of new PoIs, which are sent to their destination through Java RMI. The evaluation runs are therefore intended to be run with multiple clients and instances of the application simultaneously. After the specified amount of time for the evaluation has passed, the application stops executing the protocol and saves the gathered evaluation data before exiting. The evaluation mode of the DeXTT client can be parameterized through additional options and parameters. The relevant configuration options are listed below.

Client Addresses. The DeXTT addresses (see Section 5.1.2) of all participating clients are specified through the `-c` or `--clientAddresses` option. These addresses are used for the periodical creation and sending of PoIs via Java RMI.

DeXTT Genesis Block. The height of the first block that contains DeXTT payloads on a blockchain can be declared by the `-g` or `--genesisBlock` option. This enables the client to skip the retrieving and parsing of all previous blocks that are known to not contain any DeXTT transactions. For large blockchains, this will result in a much faster DeXTT protocol initialization for the client.

Contest Participation Mode. By specifying the `-m` or `--contestMode` option, the mode for contest participations can be chosen. The value must be either `FULL` or `HASHREFERENCE`. The `FULL` mode sends all contest participations through the usage of DeXTT-Bitcoin claim transactions, whereas the `HASHREFERENCE` mode utilizes the DeXTT-Bitcoin contest transactions if possible as specified in Section 5.3.2.

Evaluation Runtime. The time set via the `-r` or `--runtime` option defines the total runtime for the evaluation run.

Transfer Validity Period. The time span for the validity period of a DeXTT transaction is given by the `-t` or `--transactionTime` option. This validity period defines the time period for the witness contest of a DeXTT transfer (see Section 2.4).

Unconfirmed Transactions. The `-u` or `--processUnconfirmed` option defines if the application should utilize unconfirmed transactions as described in Section 5.4.

Contest Participation Waiting Period. The maximum waiting period for clients until they participate in a witness contest (see Section 5.3.3) is split into two options, depending on whether unconfirmed transactions are utilized or not. If only confirmed transactions are used, the maximum waiting period is specified by the `-b` or `--contestBlocksWait` option and given in the number of blocks to wait. For the usage of unconfirmed transactions, the maximum waiting period can be defined in milliseconds by using the `-w` or `--contestTimeWait` option.

Force Veto Transactions. By default, an evaluation run only creates valid transactions, i.e., no veto contests are needed. To also evaluate veto contests, the forcing of double spending attempts needs to be enabled by the `-f` or `--forceVetos` option. This creates two instead of one token transfer periodically. These PoIs are

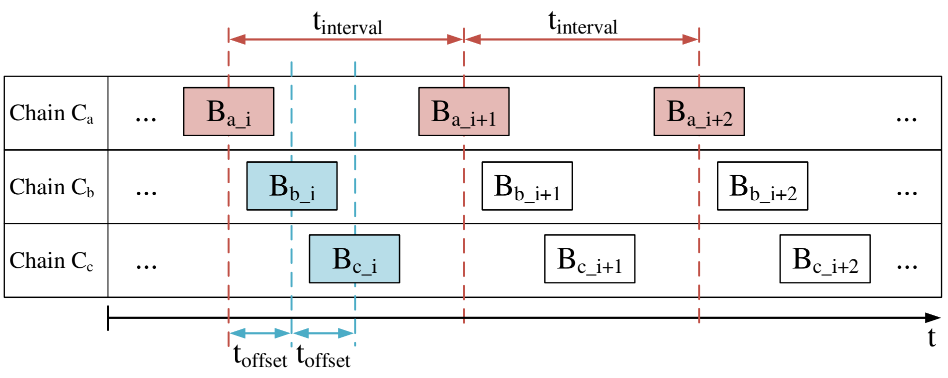

Figure 7.1: Block generation mode timings.

therefore invalid because they feature the same sender and overlapping validity periods and trigger a veto contest. In our implementation, clients that receive a sender's intent check it for its validity first, before sending an according claim transaction. Therefore, the checking for double spendings of tokens must also be disabled by specifying the `--allowDoubleSpend` option. Because the sender of double spending token transfers is locked from further transfers by default in our implementation, the `--enableAutoUnlocking` must be specified for multiple double spending token transfers that trigger veto contests by the same sender.

7.1.2 Block Generation Mode

The block generation mode of the DeXTT client enables the periodical creation of new Bitcoin blocks when using Bitcoin Core in regtest mode. This is achieved through the `GenerateToAddress` RPC of Bitcoin Core which is only enabled for regtest mode. Just as the evaluation mode, the block generation mode can operate on multiple instances of Bitcoin Core and blockchains simultaneously. The periodical generation of blocks is required for the evaluation runs, because without the creation of blocks, no transaction will ever be confirmed and therefore the DeXTT protocol implementation can not be evaluated correctly. The block creation times can be configured by CLI parameters as described below.

Block Generation Runtime. The time set via the `-r` or `--runtime` option defines the total runtime for the block generation run.

Block Creation Interval. The block generation mode creates new Bitcoin blocks at a fixed rate. The time between each newly created block on each blockchain is specified as $t_{interval}$ via the `-i` or `--blockInterval` option. The block creation interval timing is illustrated in Figure 7.1.

Blockchain Offset. When multiple blockchains are used for an evaluation, the block generation mode must generate blocks for all participating blockchains. To enable an approach where blocks are not created simultaneously across all blockchains, an additional timing information is introduced. The time t_{offset} defines how much time lies between the creation of a block on blockchain C_a and the creation of a new block on blockchain C_b. The concept can be applied to multiple blockchains. Each blockchain still maintains the same $t_{interval}$ time, but the block creation is shifted by t_{offset} into the future compared to the previous blockchain. If for a number of blockchains n, $(n-1) \cdot t_{offset} \geq t_{interval}$ holds, a wrap around occurs and the creation of blocks across blockchains overlaps. The concept of the different timings of block creations across blockchains is shown in Figure 7.1.

To achieve the biggest possible time between each newly generated block, the blockchain offset must be specified as $t_{offset} = \frac{t_{interval}}{n-2}$. To create blocks on one blockchain with the biggest possible gap to another blockchain, the offset must be specified as $t_{offset} = \frac{t_{interval}}{2}$. This timing implies of course that the big offset only applies to half of the blockchains, whereas the other half will have an offset of zero to each other.

7.2 Evaluation Setup and Environment

This section gives an overview of the setup including settings and timings for the evaluations runs. In addition, the evaluation environment and different configurations are discussed. For the actual evaluation runs, a suitable execution setup and environment have to be specified. The evaluation setup is aimed to be as similar as possible to the setup that was applied for the evaluation of the Ethereum implementation [Bor+19b]. The Bitcoin blockchain has an expected median block creation rate of one block every ten minutes (see Section 2.2.2), whereas the block creation time that was used for the Ethereum evaluation was fixed at 13 seconds. Therefore, the timings and durations have to be adopted and scaled accordingly to be both similar to the Ethereum evaluation but also using the according timings that are suitable for the Bitcoin blockchain.

7.2.1 General Evaluation Setup

In our evaluation, we use a fixed block creation time of ten minutes to mimic the aimed median block time of Bitcoin. This results in a factor of about 46.15 between the 13 seconds of the Ethereum evaluation and our approach of ten minutes (600 seconds). Because we aim to have evaluation runs with the same relative duration as it was done for Ethereum, the run time has to be scaled up using this factor between the block creation times. This is required because the run time defines the number of blocks that will be created during the evaluation run and therefore also the number of DeXTT transactions that are possible. To run the evaluation with the same amount of created blocks, the run time is increased from 30 minutes in Ethereum to 83,077 seconds or 23 hours 4 minutes and 37 seconds in our Bitcoin evaluation. Just as in the Ethereum evaluation, the clients create and send new token transfers at a random time out of an interval after the previous transfer of the sender has been finalized and therefore terminated successfully. This interval is defined to be between 15 and 30 seconds in Ethereum [Bor+19b]. Therefore, we scale it up by about 46.15 in our approach, which results in a time interval between 692 and 1385 seconds.

For the rest of the evaluation that is done through private Bitcoin blockchains that are provided by the regtest mode of Bitcoin Core, the same setup as for the Ethereum evaluation is used. The evaluation runs consists of *ten clients*, each running an instance of the DeXTT-Bitcoin implementation. In addition, three private Bitcoin blockchains are used simultaneously for each run to create a multi-blockchain setup. This is done by using three instances of Bitcoin Core in regtest mode, where each is started using a different *data directory* [bit19f] to enable the differentiation between the blockchains of the Bitcoin Core instances. Furthermore, each Bitcoin Core instance must be configured by its own configuration file, to assign different network ports to the instances. Additionally, for each evaluation run, one instance of the DeXTT-Bitcoin implementation has to be started to periodically create blocks by using its block generation mode.

7.2.2 Evaluation Configurations

Throughout the evaluation process, different evaluation runs utilizing different sets of configurations are used. The main parameter to be adjusted for different runs is the transfer validity period to allow the evaluation of its impact on the success of token transfers. The validity period is adjusted for runs with different configurations and the amount of successful and corrupt token transfers is measured. A corrupt token transfer occurs, if the transfer results in inconsistencies across the participating blockchains.

The block creation interval that defines the time between the generation of new blocks on a blockchain is set to be fixed at ten minutes as described above. Additionally, the parameter for the blockchain offset time t_{offset} (see Section 7.1.2) is adjusted across different runs. Three different values for the offset are used, all representing a certain edge case. For a blockchain offset of 0 seconds, blocks across blockchains are generated at about the same time, an offset value of 200 seconds represents the edge case of the maximum time between each generated block across the blockchains. At last, an offset value of 300 seconds is used to achieve the edge case of the maximum time between blocks of the first and second utilized blockchains. As a result, the third blockchain generates new blocks at about the same time as the first one. For the transaction period evaluation, the last configuration parameter that is adapted for different runs is the usage of unconfirmed transactions. All evaluation runs are once executed with and without utilizing unconfirmed transactions.

In addition to the evaluation of the impact of the transfer validity period, the impact of the maximum wait time for contest participations (see Section 5.3.3) is also evaluated. For these evaluations, the transfer validity period is fixed and only the maximum wait time and the usage of unconfirmed transactions is adapted throughout the evaluation runs.

Throughout all evaluation runs, the relevant sizes of the underlying Bitcoin transactions for all DeXTT-Bitcoin transactions are measured. These transaction sizes are proportional to the transaction costs of the transactions, because for each vbyte, a certain amount of transaction fee has to be paid (see Section 2.2.6). In our evaluation, it is assumed that transactions get included in the next generated block, therefore for a later cost analysis, the according fee rate has to be used. Because no veto contest transactions occur in the evaluation runs described above, an additional evaluation run that forces veto contests is used to measure the transaction sized.

In the following paragraphs, all configurations of evaluation runs that are executed for the evaluation of the DeXTT-Bitcoin client implementation are listed. For the transfer validity period evaluation, runs that include all combinations of the following parameters are executed:

- The usage of unconfirmed or only confirmed transactions, resulting in two different configurations.
- Three different values for the blockchain offset as described in Section 7.1.2 are used throughout the configuration: zero, 200 and 300 seconds. The blockchain offsets enable the gathering of data that is less generic and closer to real blockchain ecosystems than only generating blocks in a synchronized manner.
- The transfer validity period is step-wise increased, starting at 231 seconds and being increased by 231 seconds for each configuration until the value of 3234 seconds is reached. This results in 14 different configuration values. The values are again scaled up by a factor of about 46.15 from the 5 seconds increase interval that was used for Ethereum. Considering the results of the Ethereum evaluation [Bor+19b], 3234 seconds is assumed to be a high enough validity period to only generate successful token transfers.

The total number of evaluation runs for the transaction period evaluation is calculated by combining all above configurations, resulting in 84 ($2 \cdot 3 \cdot 14$) different configurations. All of these runs are executed with the same maximum waiting period for contest participations, namely *one block* when using only unconfirmed transactions and *ten seconds* for configurations that utilize unconfirmed transactions. For the additional runs for the contest participation waiting period evaluation, the following configurations are used:

- Again two different configurations for unconfirmed or only confirmed transactions are applied.
- The same three blockchain offset values of zero, 200 and 300 seconds are used.

- The transfer validity period is fixed at 3,234 seconds throughout these evaluation runs, because the evaluation of this time period is not the aim of these evaluation runs. Rather, the impact on different contest participations is measured.

- For the usage of only confirmed transactions, five different values for the maximum contest participation waiting period are used: zero, two, three, four or five blocks. This results in five different configurations for confirmed transactions. For the usage of unconfirmed transactions, three different maximum waiting periods are considered: zero, 20 or 30 seconds. The configurations for one block and ten seconds are already included in the evaluation runs for the transaction period evaluation.

The total number of additional configurations for the waiting period evaluation is again given by the combination of the above configurations, resulting in 24 configurations. Nine ($1 \cdot 3 \cdot 1 \cdot 3$) for the usage of unconfirmed transactions and 15 ($1 \cdot 3 \cdot 1 \cdot 5$) configurations that only utilize confirmed transactions. One additional evaluation configuration is applied to measure transaction sizes for veto contests. This evaluation run is configured as follows: only confirmed transactions, a block creation offset of zero seconds, a maximum contest participation wait time of one block and a transfer validity period of 3234 seconds. The overall number of different configurations and therefore of required evaluation runs for the specified parameter combinations amount to 109 ($84 + 24 + 1$).

In addition to the 109 different evaluation runs for a multi-blockchain setup using Bitcoin Core's regtest mode, one additional evaluation run is executed on the Bitcoin testnet (see Section 2.2.7), representing a single-blockchain setup that utilizes a real deployed blockchain instead of private ones. The following configuration was used for the single testnet evaluation run: utilize unconfirmed transactions, ten seconds maximum wait time for contest participations and a transfer validity period of 6,468 seconds. The validity period was chosen to be double the value of the maximum period that was used for the evaluation using the regtest mode, because the testnet has an expected block creation time of about 20 minutes (see Section 2.2.7), instead of the used ten minutes for the multi-blockchain setup. Because only a single blockchain is used and blocks are not generated by our application, no blockchain offset must be specified.

7.2.3 Execution Environment

One evaluation run needs more than 23 hours to execute and our evaluation is aimed to consist of many different runs. Therefore, it is not feasible to execute all runs in a sequential order on one machine. A sequential execution order would require a vast amount of time, e.g., 100 different evaluation runs would take up more than 96 days. Because of this limitation, we execute our evaluation runs in a parallel manner, executing all runs at the same time. This approach only requires the specified 83,077 seconds to be executed plus any overhead for initialization of the runs. But the parallel execution of evaluation runs comes with the drawback of requiring more computational resources.

As described in Section 7.2.1, one evaluation run with one single configuration consists of executing eleven DeXTT-Bitcoin client instances and three Bitcoin Core instances simultaneously. These instances not only require a certain amount of CPU power, but additionally a significant amount of memory is needed. To limit the memory usage of the Java client instances, we set the maximum size of the heap for each instance to 256MB [Ora20], which was measured to be sufficient. Each instance of Bitcoin Core was measured to require up to 100MB of memory. Together, all program instances for one evaluation run require at least 3116MB of memory. Therefore, the number of feasible evaluation runs that can be performed in a parallel manner is bound by the memory resources of the underlying machine.

As described in Section 7.2.2, 109 different configurations and therefore evaluation runs should be executed. The parallel execution of these 109 configurations results in a memory requirement of at least 339,664MB ($109 \cdot 3116$MB) or about 331.68GB. To content these memory demands, a VM instance running Debian[2] 10 was used on the Google Cloud Platform[3]. The chosen instance provides 512GB of memory and 64 virtual CPU cores, providing enough margin for higher memory demands. Because by default, the used Debian 10 operating system does not allow enough threads to run all 109 configurations in parallel, its system settings have to be adapted accordingly[4,5].

To start all parallel evaluation runs, *bash* scripts are used[6]. For each configuration, a folder that contains a *data directory* and *configuration file* for Bitcoin Core is created [bit19f]. Additionally, in each folder for the different configurations, an argument file for each DeXTT-Bitcoin client instance is created. These argument files contain all according CLI arguments for the client and can be used to specify the included arguments to the application by using `@<file>` instead of specifying the arguments directly[7]. These brings the advantage that the files can be reused to run the application with the same arguments again.

Before the instances of a configuration are started, initial blocks are generated on the according private blockchains by the `GenerateToAddress` RPC to provide enough Bitcoin funds for each client to post Bitcoin transactions on the blockchains.

7.3 Multi-Blockchain Evaluation Analysis

Within this section, the gathered data of the evaluation runs using a private multi-blockchain setup enabled through regtest mode of Bitcoin Core is analyzed. The different configurations of the evaluation runs are presented in Section 7.2.2. The analysis is

[2]https://www.debian.org/
[3]https://cloud.google.com/
[4]https://askubuntu.com/questions/845380
[5]https://stackoverflow.com/questions/344203
[6]https://github.com/MatthiasKuehne/dexxt-bitcoin-prototype/tree/master/scripts
[7]https://picocli.info/#AtFiles

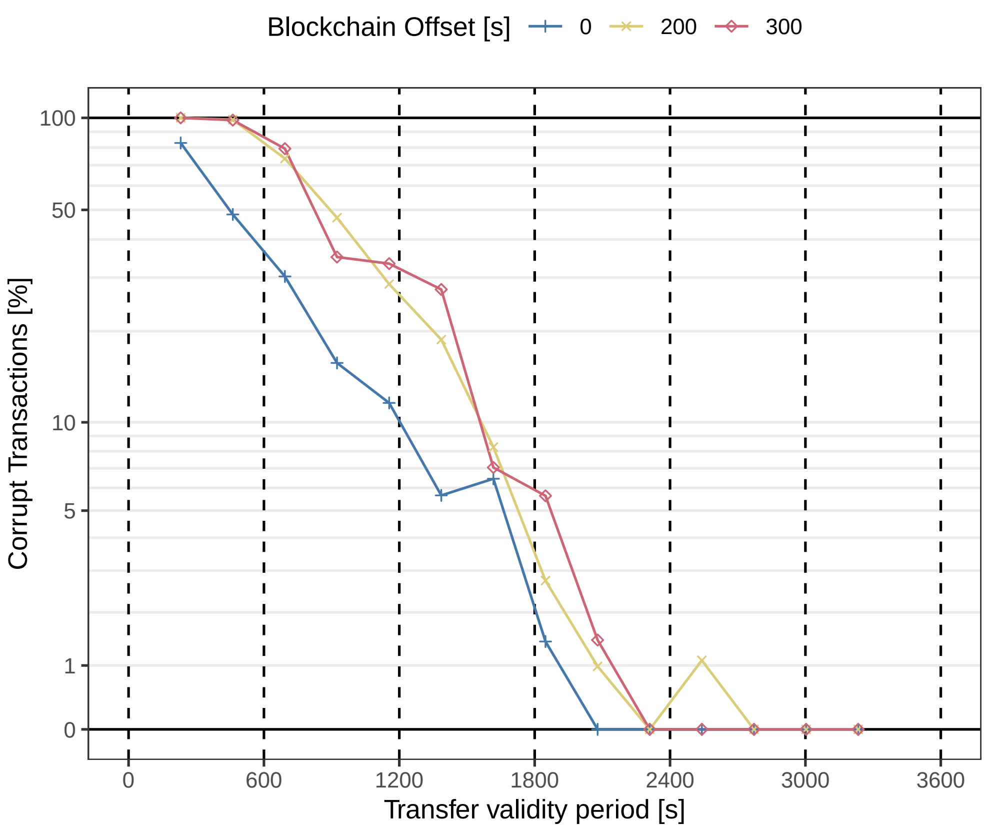

Figure 7.2: Transfer validity period evaluation with confirmed transactions (logarithmic scale).

split into three parts. First, the evaluation of the impact of the transfer validity period of DeXTT transactions regarding the percentage of corrupt transactions is presented. Second, different values for the maximum waiting period for contest participations is analyzed regarding the percentage of total contest participations and the percentage of contest participations through DeXTT-Bitcoin contest transactions. Third, the cost of the different DeXTT transactions are analyzed based on the size of the measured Bitcoin transactions.

7.3.1 Transfer Validity Period Analysis

For the analysis of the impact of the transfer validity period, the percentage of corrupt transactions is measured. Corrupt transactions include all token transfers that lead to inconsistencies across the blockchains and therefore do not achieve eventual consistency. This can happen if not all claim or contest transactions are registered and executed on all blockchains within the transfer validity period. Inconsistencies can then occur if either a different witness contest winner is chosen or if the contest is not even started on some of the blockchains.

Figure 7.2 shows the results of the evaluation of the transfer validity period for the configurations that only utilized confirmed Bitcoin transactions. Each line represents the percentage of corrupt transactions for a different blockchain offset value t_{offset} (see Section 7.1.2). The marked data points that make up the line each represent one evaluation run with the given configuration. Every 600 seconds time interval as marked on the x-axis of the shown diagram represents the time of ten minutes for the creation of one Bitcoin block. It can be observed that the blockchain offset time makes a difference in the percentage of corrupt transactions. For a blockchain offset of zero seconds, the percentage of corrupt transactions stays at zero for transfer validity periods of 2079 seconds and higher values. The corrupt transaction percentage for blockchain offsets of 200 and 300 seconds is very similar and consistently higher than for a zero seconds offset. The difference in the results for the given blockchain offsets can be explained by the delay of the initial claim transaction between the blockchains. Clients on later blockchains can therefore also react later to the claim transaction. Generally, about 5 Bitcoin blocks or 3000 seconds are required for the transfer validity period to only produce successful token transfers that ensure eventual consistency.

In Figure 7.3, the results of the evaluation of the transfer validity period for the configurations that utilized unconfirmed Bitcoin transactions are presented. It can be observed that the difference between the evaluated blockchain offset times is not significantly apparent when using unconfirmed transactions. The reason for this behavior is given by the design of the processing of unconfirmed transactions, which allows clients to react to claim transactions immediately, without the need for a new block to be generated first. Therefore, the timing of block creations is not as relevant as it is for configurations using only confirmed transactions. Generally, for transfer validity periods of about 4 Bitcoin blocks or 2400 seconds, no corrupt and therefore inconsistent transactions occur.

The difference between the usage of only confirmed transactions and the inclusion of unconfirmed transactions is shown in Figure 7.4. The shown percentages of corrupt transactions are aggregated values of the percentages of the different blockchain offset values for each transaction execution type. The data shows that the utilization of unconfirmed transactions (see Section 5.4) within the DeXTT-Bitcoin client implementation gives a significant advantage on the required minimum transfer validity period. The percentage of corrupt transactions is much lower for the same validity periods by using unconfirmed transactions. The transfer validity period and therefore the amount of Bitcoin blocks that are minimally required to generate only consistent token transfers based on our data also differs between the two approaches. When using unconfirmed transactions, an about 600 seconds or one Bitcoin block lower validity period is required compared to only processing confirmed transactions. The reason for this difference is again given by the fact that clients can react to claim transactions faster and therefore sending their contest participations earlier.

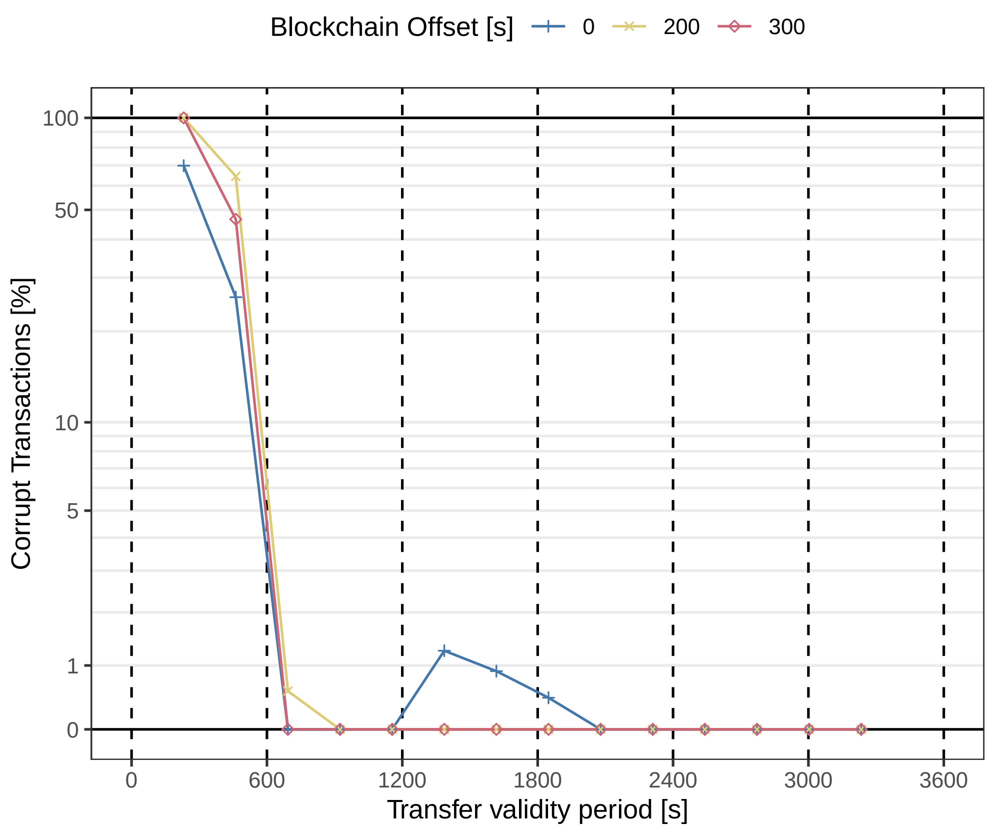

Figure 7.3: Transfer validity period evaluation with unconfirmed transactions (logarithmic scale).

7.3.2 Contest Participation Waiting Period Analysis

The analysis of the impact of the maximum waiting period for contest participations is done by observing how contest participations differ for different waiting periods. Two different values regarding contest participations are gathered:

Contest Participations Represents the percentage of overall contest participations among all clients that take part in the evaluation. The value excludes the receiver of a token transfer who automatically always participates in the according witness contest due to the design for our approach (see Section 5.3.2). In theory, for n overall clients, an average of $log_2(n)$ witness candidates will participate in the witness contests (see Section 5.3.3). The waiting period was introduced in our approach to enable a similar behavior. To analyze the impact on the introduced waiting period, the percentage of all contest participations has to be evaluated.

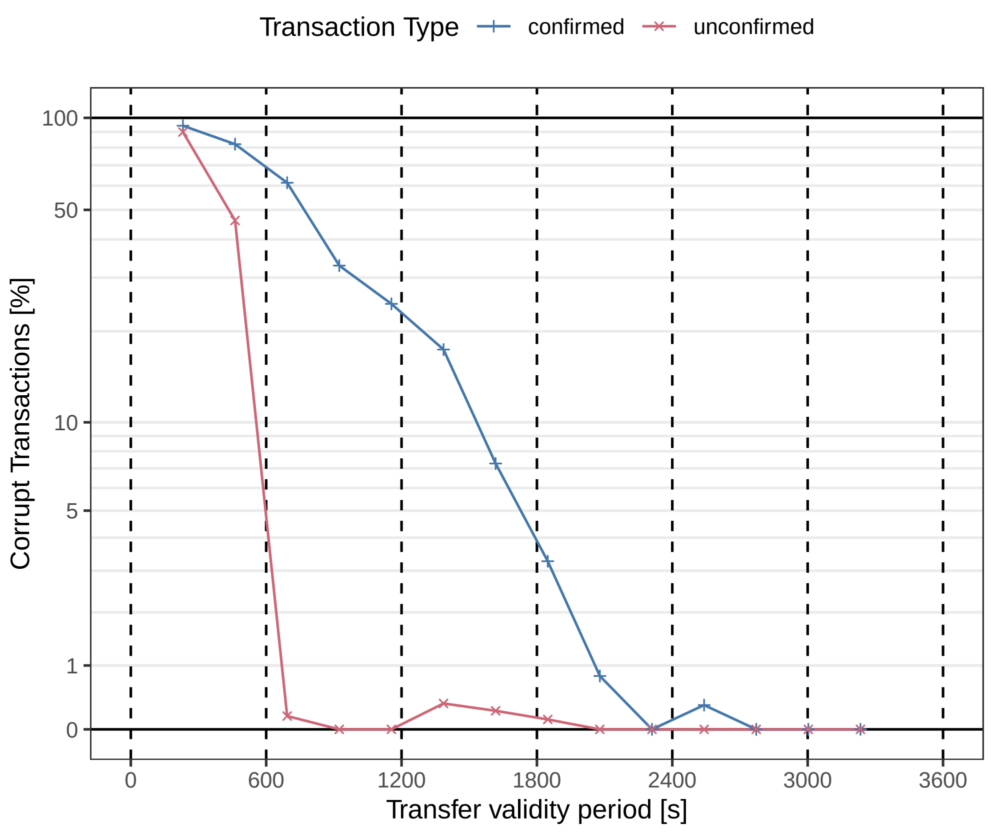

Figure 7.4: Difference between confirmed and unconfirmed transaction regarding the transfer validity period (logarithmic scale).

Contest Transaction Participations The waiting period has also a theoretical impact on the number of contest participations that are executed through DeXTT-Bitcoin contest transactions (see Section 5.3.2). The longer a client waits, the higher the probability that a DeXTT-Bitcoin claim transaction is already present on each blockchain and therefore a DeXTT-Bitcoin contest transaction suffices to participate in the witness contest. To analyze such an impact of the maximum waiting period, the percentage of contest participations through DeXTT-Bitcoin contest transactions out of all contest participations is recorded for the evaluation.

In the context of this evaluation, the theoretical average number of witness contest participations of $log_2(n)$ can be quantified for the given number of participating clients in the evaluation runs. This results in a theoretical number of participations per contest of about 3.32 ($log_2(10)$) clients or 33.2% for the total number of ten clients.

The impact of the maximum waiting period on the number of contest participations as it was observed in our evaluation runs for configurations using only confirmed transactions

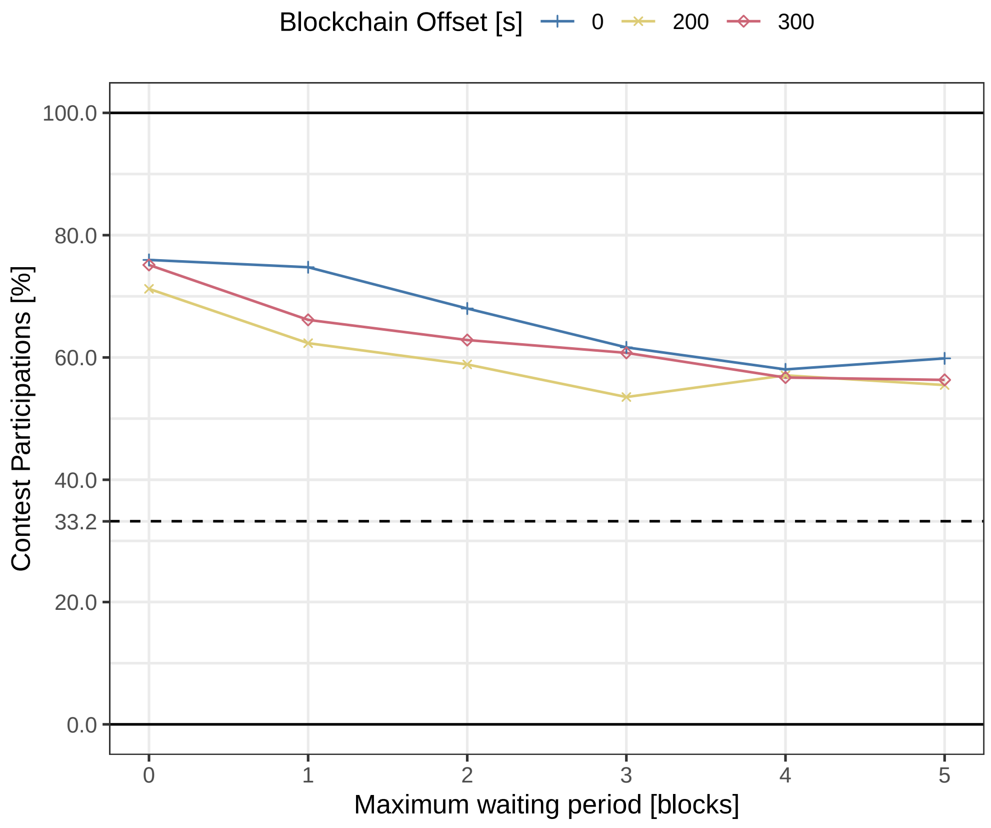

Figure 7.5: Contest participations with confirmed transactions.

is shown in Figure 7.5. Again, each line represents the observed values for the given blockchain offset time. It can be seen that the different blockchain offset times do not have a significant impact on the observed contest participations. Generally, the number of contest participations per witness contest is a lot higher than the theoretical value, but is getting lower for longer maximum waiting periods. The reason for the gap between the theoretical and observed values lies in the nature of the design of DeXTT for the Bitcoin blockchain, where contest participations are not sent in a sequential manner and not all other participations are known by the time of the participation (see Section 5.3.3).

In contrast, in Figure 7.6 the same evaluation is shown for configurations that utilize unconfirmed transactions. Again, the blockchain offset time does not introduce any significant difference in the number of contest participations and the measured values are still higher than the theoretical average of contest participations. The number of contest participations falls once from the maximum waiting period of zero seconds to ten seconds significantly and then remains about the same for the remaining waiting times. This can be explained by the nature of the waiting periods when using unconfirmed transactions. The waiting period is chosen randomly from a big amount of different time

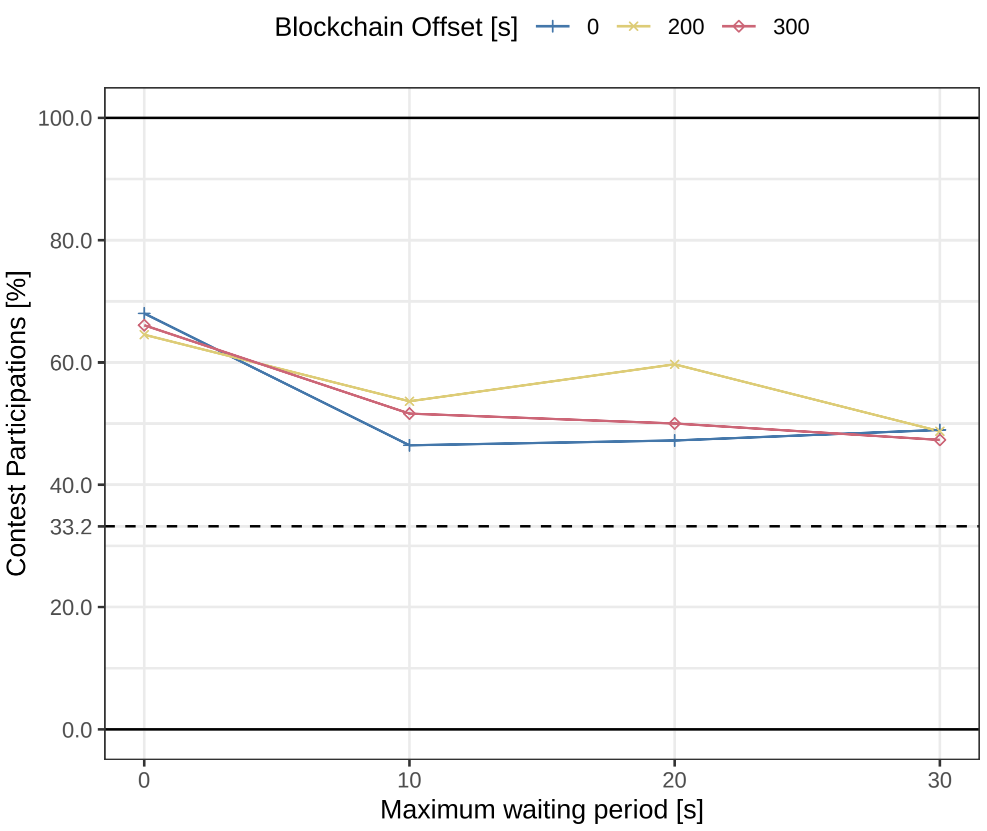

Figure 7.6: Contest participations with unconfirmed transactions.

values, therefore the chosen waiting times are expected to be distributed uniformly across the possible values, leaving enough time between the participations for other clients to observe the previous participations. Therefore, a further increase in the waiting period does not introduce any significant additional gains for a lower contest participation percentage.

When comparing the results for the run using only confirmed transactions in Figure 7.5 with the results of using unconfirmed transactions in Figure 7.6, it can be seen that the number of contest transactions is smaller when using unconfirmed transactions even for the same maximum waiting period of zero seconds or blocks respectively. The reason for this difference again lies in the nature of how unconfirmed transactions are processed in our design approach. It is possible for a client to have already observed other unconfirmed participations at a waiting period of zero because the timing of the execution of clients are not synchronized, therefore waiting period zero still means a small difference in absolute time between the clients.

The impact of the maximum waiting period for contest participations on the percentage

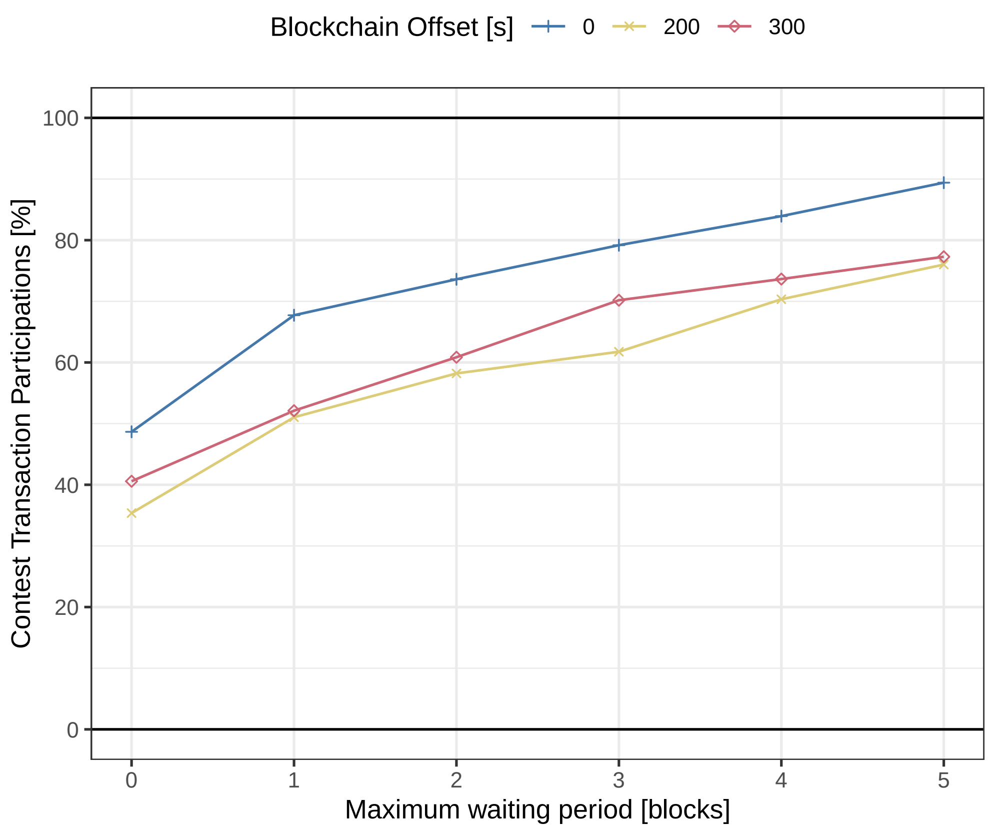

Figure 7.7: Contest transaction participations with confirmed transactions.

of contest participations through DeXTT-Bitcoin contest transactions for configurations using only confirmed transactions is shown in Figure 7.7. It can be observed that the percentage of DeXTT-Bitcoin contest transaction participations significantly differs between different values of the blockchain offset time. The reason for this behavior is given by the design of how contest participations are sent and processed. If a client sends a claim transaction as a contest participation, it needs to be confirmed on the blockchain before it is seen by the other participants. Therefore, for a certain point in time and a given blockchain offset greater than zero, it is more likely that the previous claim transaction is not yet confirmed on all blockchains, resulting in another participation using a claim transactions.

In contrast, the impact of the maximum waiting period on the number of contest participations through contest transactions for configurations using unconfirmed transactions is shown in Figure 7.8. For the usage of unconfirmed transactions, the different blockchain offset times do not introduce any significant change in the number of contest transaction participations in contrast to the evaluation runs using only confirmed transactions. The blockchain offset does not affect the participations, because to determine the way of

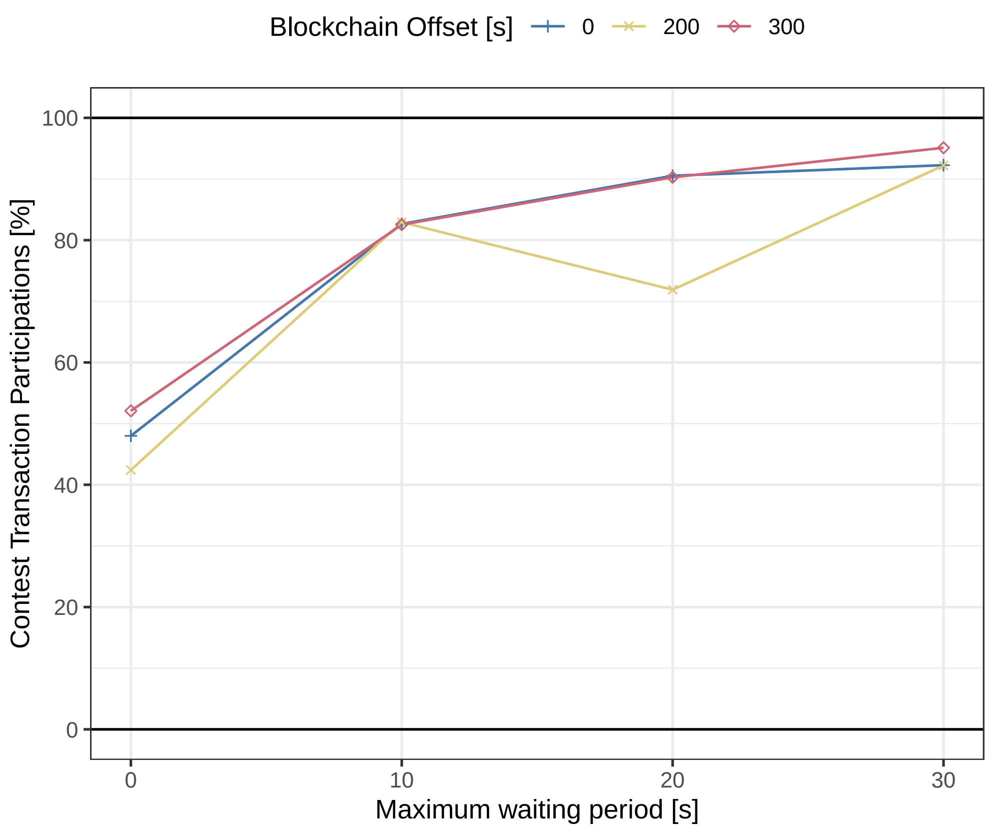

Figure 7.8: Contest transaction participations with unconfirmed transactions.

participation, only unconfirmed claim transactions are needed to be observed. Therefore, it does not matter when the transactions are included in blocks. The contest transaction participation percentage does not reach 100% for higher maximum waiting periods, because the waiting period is chosen randomly between zero and the maximum value. On average, this will always result in a few very small waiting times, where the clients did not yet observe the claim transactions on all blockchains.

In general, it can be concluded that a larger waiting time comes with significant advantages considering the average witness contest participation cost. Both a lower percentage of overall participations and the usage of contest transactions instead of claim transactions lower the amount of data that must be included in the blockchain and therefore reduce the cost to follow the protocol. For the usage of unconfirmed transactions, this advantage can already be achieved by choosing a relatively small waiting period of ten seconds without introducing any disadvantages. When using only confirmed transactions, a higher maximum waiting period does increase the impact of the cost advantages, but comes with relative long waiting times of a few blocks. Therefore, the waiting period can influence the minimal required transfer validity period and must be chosen carefully.

Table 7.1: Measured DeXTT-Bitcoin transaction sizes.

Transaction	**Mean [vbytes]**	σ **[vbytes]**	**Number of Observations**
Claim	604.98	0.150	162975
Contest/Veto	159.99	0.088	141825
Finalize	159.99	0.090	69528
Finalize-Veto	147.99	0.092	234

7.3.3 Cost Analysis

For the analysis of the costs of DeXTT token transfers on the Bitcoin blockchain, the costs of the different DeXTT-Bitcoin transactions that are required for the DeXTT transactions have to be evaluated. Because the cost for posting Bitcoin transactions is based on the size of the transactions, the according sizes for the Bitcoin transactions that embed the various DeXTT-Bitcoin transactions are measured. For this measurement of transaction sizes, no additional evaluation runs are required, because the sizes can be acquired while executing the evaluation runs for the previous analysis as described in Section 7.2.2.

Because we utilize P2WPKH standard transactions (see Section 2.2.5), the size that is taken into account for the fee calculation given in virtual bytes (vbytes) is measured for each transaction. The results of these measurements, given as the mean and standard deviation (σ) of all measurements combined, are presented in Table 7.1 together with the number of observed transactions that the calculation is based on. Because the length difference for Bitcoin transaction varies only on a very small scale (see Section 5.1.1), the standard deviation of the measured transaction sizes is negligible small.

The actual costs for these transactions not only depends on their size. In addition, the current fee rate per vbyte defines the cost for inclusion into the Bitcoin blockchain (see Section 2.2.6). For our evaluation, we assume that the according fee for inclusion into the next generated Bitcoin block is paid. If a fee rate for a later inclusion, defined by the number of blocks, is used, the evaluated transfer validity period has to be adjusted accordingly. Because the fee rate is adopted constantly, for our calculations, we took the average fee rate in April 2020 amounting to 20.93 satoshis per vbyte. A satoshi is the minimum possible transaction value of Bitcoin and represents 0.00000001 Bitcoins [Nar+16; bit18b]. The change in the fee rate for inclusion in the next block from January 2019 until April 2020 is shown in Figure 7.9 based on the Bitcoin transaction fee data from *Billfodl*[8].

The average Bitcoin price in April 2020 was 7150.611 USD per Bitcoin or 0.00007150611 USD per satoshi according to CoinMarketCap[9]. The resulting costs for DeXTT-Bitcoin transactions based on the average fee rate in April 2020 for inclusion in the next block and the

[8]`https://billfodl.com/pages/bitcoinfees`
[9]`https://coinmarketcap.com/currencies/bitcoin/historical-data/`

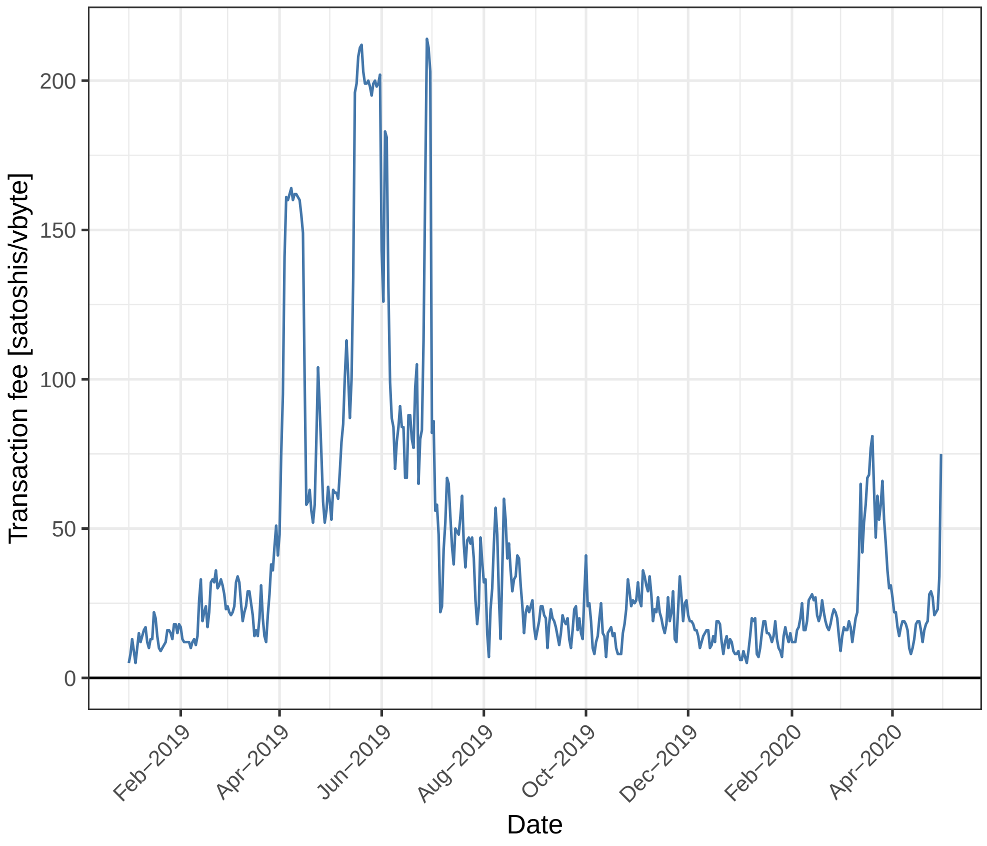

Figure 7.9: Estimated Bitcoin transaction fees for inclusion in next block.

Table 7.2: DeXTT-Bitcoin transaction costs for April 2020.

Transaction	**Mean [satoshis]**	σ **[satoshis]**	**Mean [USD]**	σ **[USD]**
Claim	12665	4	0.9056	0.0003
Contest/Veto	3350	2	0.2395	0.0001
Finalize	3350	2	0.2395	0.0001
Finalize-Veto	3098	2	0.2215	0.0001

average Bitcoin price in April 2020 are shown in Table 7.2. The evaluated costs are given both in satoshis and in USD, each entry consisting of the mean cost and its standard deviation. Note that the entry for contest and veto transactions are combined, because we do not differentiate between the two transaction types in our design.

Theoretical Token Transfer Cost

For a Bitcoin token transfer on m blockchains with n total observers, the following transactions are theoretically required to be posted on the blockchains when using our design of the protocol in an optimal environment and setting: m DeXTT-Bitcoin claim transactions, $m \cdot log_2(n)$ DeXTT-Bitcoin contest transactions and m DeXTT-Bitcoin finalize transactions [Bor+19b]. The receiver of the transfer bears the cost for m claim and m finalize transactions, whereas each of the $log_2(n)$ observers pays for m contest transactions.

Based on the previous cost evaluation, for an ecosystem of ten blockchains and ten observers that participate in the protocol, the token transfer cost for the receiver amounts to 11.4517 USD. The cost for each of the $log_2(10)$ observers is 2.3954 USD. The total amount that is payed by all observers together amounts to 7.9575 USD, resulting in total costs of 19.4092 USD among all parties of the transfer.

The used number of transactions in the above calculations are based on the theoretically best assumptions about the number of contest participations. Additionally, it is assumed that each contest participation as done through a DeXTT-Bitcoin contest transaction and no additional DeXTT-Bitcoin claim transactions are required.

In comparison, a P2WPKH standard Bitcoin transaction with one input and two outputs requires about 140 vbytes of storage that have to be paid by transaction fees[10] [bit20c; LLW15]. 109 vbytes for one input and one output as used for DeXTT payloads (see Section 5.1.1) and additional 31 vbytes for the second output, which is required for the change of the transaction input (see Section 2.2.4). For the average fee rate in April 2020 for inclusion in the next block and the average Bitcoin price in April 2020, this results in a price of 0.2096 USD for a P2WPKH Bitcoin transfer. Compared to the example of ten Bitcoin blockchains as described above, a transfer performed on all blockchains would cost 2.0956 USD to be paid by the transaction sender.

Token Transfer Cost with Confirmed Transactions

The cost calculations in the previous part are based on the theoretically best number of DeXTT-Bitcoin transactions for a token transfer. The transactions that are required when relying on the evaluated data of our implementation for the configuration that uses only confirmed transactions and a maximum waiting period of 5 blocks are as follows: m DeXTT-Bitcoin claim transactions posted by the receiver, $m \cdot 57.25\% \cdot n$ contest participations and m DeXTT-Bitcoin finalize transactions. Out of these contest transactions, 81.19% consist of a DeXTT-Bitcoin contest transaction and the remaining 18.81% are achieved through a DeXTT-Bitcoin claim transactions.

For ten participating blockchains and ten observers, the cost for the receiver amounts to 11.4517 USD. Each of the $57.25\% \cdot n$ contest participants on average pays 3.6485 USD,

[10]https://www.reddit.com/r/Bitcoin/comments/7m8ald/how_do_i_calculate_my_fees_for_a_transaction_sent/

resulting from 1.9448 USD for contest transactions and 1.7037 USD for additional claim transactions. The total amount payed by all observers together amounts to 20.8873 USD. Therefore, the total token transfer cost among all parties is 32.3390 USD.

The token transfer cost based on the data of the evaluation runs for confirmed transactions is significantly higher than the theoretical cost for the best case, because more client participate in contests and some participations require an additional DeXTT-Bitcoin claim transaction.

Token Transfer Cost with Unconfirmed Transactions

The required transactions for a token transfer based on the evaluated data for the configurations that utilize unconfirmed transactions and a maximum waiting period of 30 seconds are as follows: m DeXTT-Bitcoin claim transactions posted by the receiver, $m \cdot 48.36\% \cdot n$ contest participations and m DeXTT-Bitcoin finalize transactions. 93.16% of the contest participations are done through a DeXTT-Bitcoin contest transaction, whereas the other 6.84% require an additional DeXTT-Bitcoin claim transaction.

For a setup consisting of ten blockchains and ten observers, the receiver of the transfer pays 11.4517 USD. Each contestant pays 2.8514 USD on average, resulting from 2.2315 USD for contest transactions and 0.6199 USD for additional claim transactions. Therefore, all contest participations combined cost 13.7893 USD, yielding a total cost for a token transfers among all parties of 25.2411 USD.

This result shows that the usage of unconfirmed transactions not only allows for a shorter transfer validity period (see Section 7.3.1). They additionally introduce cheaper contest participations on average, resulting in a lower cost for token transfers, when compared to the usage of only confirmed transactions.

7.4 Bitcoin Testnet Evaluation

The evaluation run on the Bitcoin testnet is not executed to gather evaluation data, but rather as a proof of concept for the DeXTT implementation to work also on a real deployed blockchain that is more similar to the Bitcoin mainnet as the private blockchains utilized through the other evaluation runs.

The single run using the configurations presented in Section 7.2.2 yielded only successfully finalized token transfers. Because of the single blockchain scenario using unconfirmed transactions, during the execution all contest participations were done through DeXTT-Bitcoin contest transactions. The average contest participation rate was observed to be at 26.05% and therefore significantly lower than for the evaluation runs in multi-blockchain environments (see Section 7.3.2). This is most likely also a result of the single-blockchain environment, meaning that there are no delays across multiple blockchains. Another possible factor might be the low number of only 81 token transfers throughout the evaluation run, possibly skewing the data.

Table 7.3: Ethereum implementation transaction costs for April 2020.

Transaction	**Mean [USD]**	σ **[USD]**
Claim	0.1312	0.0252
Contest	0.1853	0.1459
Finalize	0.1034	0.0002
Veto	0.2985	0.2089
Finalize-Veto	0.1105	0.0039

From the execution of the DeXTT client with the Bitcoin testnet, it can be concluded that the DeXTT-Bitcoin implementation not only works on private predictable blockchains, but is also executing as expected on a deployed blockchain network.

7.5 Comparison with Ethereum Implementation

In this section, the results of the evaluation analysis of the DeXTT-Bitcoin implementation are compared to the raised data for the Ethereum implementation[11] of DeXTT.

7.5.1 Transfer Validity Period

In our evaluation, the minimum transfer validity period of token transfers was found to be five Bitcoin blocks (3000 seconds) for confirmed transactions and four blocks (2400 seconds) for unconfirmed transactions (see Section 7.3.1). The evaluation of the Ethereum implementation resulted in a minimum validity period of four blocks (52 seconds) [Bor+19b]. In terms of the number of required blocks for all transactions to be executed successfully across all blockchains, the results of both evaluations yield similar results, only the Bitcoin implementation that exclusively uses confirmed transactions needs one block more. The reason why a slightly longer validity period is required by our evaluation is most probably attributed to the utilized blockchain offset time. No comparable approach is mentioned in the evaluation of the Ethereum prototype [Bor+19b].

In terms of absolute transfer validity period, due to the nature of the blockchains, Ethereum requires a much shorter minimum validity period. Therefore, in a setting where both blockchains are used, the validity period that is required for Bitcoin determines the overall validity period.

7.5.2 Transaction Costs

To compare the costs of both DeXTT implementations, the measured costs for the Ethereum implementation given in Ethereum Gas were used to calculate more recent costs in USD. For this recalculation, the average values in April 2020 were taken into account,

[11]`https://github.com/pantos-io/dextt-prototype`

Table 7.4: Combined transaction costs of Bitcoin and Ethereum for April 2020.

Transaction	**Bitcoin [USD]**	**Ethereum [USD]**	**Combined [USD]**
Claim	0.9056	0.1312	1.0368
Contest	0.2395	0.1853	0.4248
Finalize	0.2395	0.1034	0.3429
Veto	0.2395	0.2985	0.5380
Finalize-Veto	0.2215	0.1105	0.3320

yielding a Gas price of 13.22 Gwei[12] ($1\ Ether = 10^9\ Gwei = 10^{18}\ wei$) and an Ether price of 171.89 USD[13]. The resulting transaction costs for the Ethereum implementation are shown in Table 7.3. Compared to the cost analysis of the Bitcoin implementation (see Section 7.3.3), it can be seen that the transaction costs are significantly lower for all transaction types except the veto transaction. The much higher cost for a claim transaction in Bitcoin comes from the structure of the transaction, requiring three null data outputs. Otherwise, the general higher costs for transactions in Bitcoin is mainly the result of the fee structures of the underlying blockchains and the higher price of Bitcoin compared to Ether.

The comparison of the costs for the two different blockchain technologies additionally allows the cost analysis of the combination of DeXTT for both blockchains. The combined mean costs for DeXTT transactions to be included in both blockchains simultaneously are shown in Table 7.4 [Bor+19b]. Based on these costs, the combined costs for a token transfer can be calculated for an exemplary scenario of five Bitcoin and five Ethereum blockchains and ten observers as follows. The receiver of the transfer needs to send five Bitcoin claim and five Bitcoin finalize transactions and additionally five Ethereum claim and five Ethereum finalize transactions. This results in a total cost of 6.8985 USD for the receiver. Each of the theoretical $log_2(10)$ observers pays for five Bitcoin and five Ethereum contest transactions, bearing a total cost of 2.124 USD. Therefore, all observers together pay 7.0558 USD for all contest participations. This results in a total token transfer cost of 13.9542 USD.

It can be observed, that due to the significantly lower cost on Ethereum, the transfer costs for an ecosystem consisting of ten blockchains is also significantly lower than for a pure Bitcoin ecosystem (see Section 7.3.3).

[12]https://etherscan.io/chart/gasprice
[13]https://etherscan.io/chart/etherprice

CHAPTER 8

Conclusion

Blockchain research and development have resulted in a big fragmentation of the field and mostly incompatible blockchain technologies. This fragmentation leads to additional problems for users, as they must decide which blockchain and cryptocurrency they want to use. Users must find a balance between novel features of new technologies and the time-proven security of well-established blockchains.

Due to the problems that arise from the fragmentation and incompatibility of blockchain technologies, the ability to interoperate between different blockchains needs to emerge, enabling users to select blockchains dynamically according to new trends and needs.

The DeXTT protocol ensembles an approach that provides means of blockchain interoperability and therefore tackles the aforementioned problems. DeXTT enables users to record the transfer of tokens on an arbitrary number of blockchains simultaneously, while being completely decentralized. There is no untrusted third party involved in executing the protocol. Blockchain interoperability is provided in the sense of cross-blockchain asset transfers, where tokens are not locked within an individual blockchain. These tokens can be traded and are synchronized across all participating blockchains.

A prototype implementation of the DeXTT protocol had already been implemented for Ethereum-based blockchains. Within this thesis, the DeXTT protocol was extended by implementing the protocol on another suitable blockchain technology.

Before choosing a suitable blockchain technology, a detailed requirements analysis for an underlying blockchain technology to support the adaption of the DeXTT protocol was created. The requirements were split into two parts: an approach where the protocol logic resides on the blockchain and one where all logic is handled off-chain on the client-side. Additionally, the requirements on a new implementation to be compatible with the existing Ethereum prototype were defined. It was shown that for a blockchain to support the *client-side logic* approach of DeXTT, it is sufficient for the blockchain to support *secure on-chain data storage* and provide *timestamp* data within its blocks.

To determine the most suitable blockchain in regards of yielding the biggest additional research value, the properties and features of different currently available blockchain technologies were analyzed concerning their technical aspects. This analysis was done in the form of a survey that focused on technical details that are relevant for the requirements of the DeXTT protocol. Within this survey, the top-17 ranked entries on CoinMarketCap[1] were considered and analyzed. The survey yielded that *EOS*, *Tron*, *Tezos* and *Neo* provide all means to implement DeXTT for their platforms using a *blockchain-side logic* approach (see Section 4.1.1). Furthermore, *Bitcoin*, the *XRP Ledger*, *Bitcoin Cash*, *Litecoin*, the *Binance Chain*, *Bitcoin SV*, *Stellar*, *Cardano* and *Monero* fulfill all requirements for a DeXTT implementation following a *client-side logic* approach (see Section 4.1.2). We concluded that a DeXTT implementation for *Bitcoin* would be the best choice regarding its research value.

Our elaborated design for DeXTT on the Bitcoin blockchain utilizes a *client-side logic* approach, where the blockchain is merely used as a secure and tamper-evident storage of DeXTT transactions. The transaction data is included in blocks on the Bitcoin blocks by utilizing the *null data* outputs of Bitcoin. Because null data outputs are limited to a 80 byte payload, we introduced a method of splitting the data of claim transactions into multiple null data outputs. For the communication with the Bitcoin blockchain, our design approach makes use of the Bitcoin Core client software, which is accessed by its JSON-RPC interface.

Within the context of this thesis, we created a concrete implementation of the proposed design of the DeXTT protocol for the Bitcoin blockchain. For this implementation, we decided to employ the *Java programming language* together with several Java libraries that allowed us to realize the client-side application efficiently while applying best practices of software engineering. Among others, the utilized Java library included the *bitcoin-rpc-client* library, which offers wrapper functions to access the JSON-RPC API of Bitcoin Core. Our presented implementation approach was designed to not only feature all details that are needed to run the DeXTT protocol together with the Bitcoin blockchain via Bitcoin Core, but also includes all necessary code to excessively run the software for testing and evaluation means. Therefore, the DeXTT-Bitcoin implementation that was created in the context of this thesis represents a solution that bundles the DeXTT protocol execution together with the evaluation runs.

We utilized the created DeXTT-Bitcoin implementation excessively to execute a variety of evaluation runs to evaluate both the design and implementation of the DeXTT protocol for Bitcoin. The executed evaluation runs were comprised of 109 different run configurations for multi-blockchain setups. Each evaluation run consisted of three privately-managed Bitcoin blockchains via Bitcoin Core in regtest mode, ten DeXTT-Bitcoin client instances that periodically initiated token transfers and one DeXTT-Bitcoin client instance to periodically generate new Bitcoin blocks. The results of these evaluation runs yielded a minimum transfer validity period for DeXTT token transfers of four Bitcoin

[1] `https://coinmarketcap.com/`

blocks (2400 seconds) for configurations where the clients utilized unconfirmed Bitcoin transactions (see Section 5.4). For the configurations that only used confirmed Bitcoin transactions, a minimum transfer validity period of five Bitcoin blocks (3000 seconds) was found to be sufficient.

These results regarding the transfer validity period match the minimum validity period that was found to be sufficient for the Ethereum prototype, where four blocks were required, although the absolute time was much shorter (52 seconds) due to the difference of the block creation times between Bitcoin and Ethereum.

Concerning the costs for DeXTT token transfers on Bitcoin, the gathered data concerning the sizes of the underlying Bitcoin transactions combined with the average Bitcoin fee for inclusion in the next block and the average Bitcoin price in April 2020 resulted in the following values:

- 0.9056 USD for a claim transaction,
- 0.2395 USD for a contest, veto, or finalize transaction and
- 0.2215 USD for a finalize-veto transaction.

Compared to the transaction cost of Ethereum based on the average Gas and Ether prices in April 2020, the contest, finalize and finalize-veto transactions are about a factor of two more expensive on Bitcoin. The claim transaction on Bitcoin costs about seven times as much as on Ethereum. Only the veto transaction was found to be slightly cheaper on Bitcoin compared to Ethereum.

In addition to the evaluation runs using privately-managed blockchain instances, one test run was executed on the Bitcoin testnet, which showed that the DeXTT-Bitcoin client implementation is also suitable to be applied to a real deployed blockchain network.

There are still some open questions and topics for possible future work:

Eager Execution of Unconfirmed Transactions. Within the context of this thesis, the design for DeXTT on the Bitcoin blockchain was created with the execution of confirmed transactions in mind. Additionally, we introduced the processing of unconfirmed Bitcoin transactions in a limited way to gather information about claim and contest transactions before a new block is found. This limited processing of unconfirmed transactions does not include any means of actually executing DeXTT transactions and therefore does not change the state of the client and wallet. The handling and execution of unconfirmed transactions can also be designed in a way that eagerly executes every observed unconfirmed transaction, but this approach could result in inconsistencies and indeterminisms. Therefore, to enable the eager execution of unconfirmed transactions, meaning that all unconfirmed DeXTT transactions are processed and executed leading to a change in state, additional features including the undoing of already executed transactions and the reevaluation of the correct state of the DeXTT client need to be introduced.

One Global Claim Transaction. In our current design of DeXTT for Bitcoin, on each participation blockchain at least one claim transaction for every token transfer is required. This claim can then be referenced by the DeXTT-Bitcoin contest transaction, effectively enabling contest participations for less cost. It is also possible to reduce the required claim transactions to one global claim transaction, which can be included on any of the participation blockchains. Therefore, on the other blockchains, this global claim can be referenced through its hash value inside contest transactions. This approach can further reduce the cost for contest participations and for the receiver of a token transfer, who then only needs to send one claim transaction to one blockchain and only contest transactions to the other blockchains, instead of sending a more expensive claim transaction to all blockchains.

Inclusion of DeXTT Transaction hashes in Blockchain. We designed the inclusion of DeXTT transaction in the transactions of Bitcoin through the insertion of all relevant data of the transactions in null data outputs. An approach which requires less data to be included and therefore reduces the overall cost, is to only include the hash values of DeXTT transactions in null data outputs of Bitcoin. For such an approach to be feasible, there must additionally exist a way of distributing the actual transaction data among the participation clients. This can be achieved by building a custom network for data exchange between all clients.

Additional DeXTT Implementations. Another interesting DeXTT implementation that builds upon both the Ethereum and newly created Bitcoin clients, is a client that combines both implementations into a global approach that can actually run DeXTT on both Bitcoin and Ethereum simultaneously. This enables the evaluation of an approach that bridges token transfers across different blockchain technologies. In addition, a DeXTT design and implementation for a completely different blockchain technology can yield additional results on how the protocol performs on various blockchains. Some possible candidates for future implementations on different blockchain technologies include *EOS*, *Tezos* or *Neo*.

List of Figures

List of Tables

List of Algorithms

Acronyms

ABI Application Binary Interface. 23

ACID Atomicity, Consistency, Isolation and Durability. 38

API Application Program Interface. 47, 65, 66, 80–82, 92

ASCII American Standard Code for Information Interchange. 58

BPM Business Process Management. 1

CLI Command Line Interface. 80, 83, 91, 92, 94, 99

CPU Central Processing Unit. 70, 99

DApp Decentralized Application. 38, 39, 41, 42, 45, 65, 74

DeXTT Deterministic Cross-Blockchain Token Transfers. 3–5, 7, 24–26, 28, 30, 31, 34, 39–51, 53–66, 68–75, 77, 79–97, 99, 100, 102–105, 108–113, 117, 119

DTO Data Transfer Object. 85, 87

ECDSA Elliptic Curve Digital Signature Algorithm. 11, 18, 19, 23, 43, 44, 46, 47, 49, 57

EOA externally owned account. 23

EVM Ethereum Virtual Machine. 1, 10, 23, 30

HTLC Hashed Time-Lock Contract. 32

HTTP Hypertext Transfer Protocol. 65, 80

JSON JavaScript Object Notation. 65, 66, 80

P2PK Pay to Public Key. 19

P2PKH	Pay to Public Key Hash. 11, 16, 18–20
P2SH	Pay to Script Hash. 11, 18–21
P2WPKH	Pay to Witness Public Key Hash. 20, 55, 71, 108
P2WSH	Pay to Witness Script Hash. 20, 21
PoI	Proof of Intent. 26–29, 59, 61–63, 71, 72, 76, 85, 88, 93, 94
PoS	Proof of Stake. 12, 24
PoW	Proof of Work. 9, 12, 24, 50
RMI	Remote Method Invocation. 82–84, 93
RPC	Remote Procedure Call. 65–70, 80, 92, 94, 99
SegWit	Segregated Witness. 14, 16, 17, 20, 21, 55, 56
SPV	Simplified Payment Verification. 34, 35
STXO	Spent Transaction Output. 16
TAST	Token Atomic Swap Technology. 34
URL	Uniform Resource Locator. 92
UTXO	Unspent Transaction Output. 9–11, 14, 16, 20, 23, 54–56, 65, 67
VM	Virtual Machine. 47, 80, 99
WIF	Wallet Import Format. 80

Bibliography

[Alo18] Kurt M. Alonso. *Zero to Monero*. 2018. URL: https://src.getmonero.org/library/Zero-to-Monero-1-0-0.pdf (visited on 05/05/2020).

[AM19] Jameela Al-Jaroodi and Nader Mohamed. “Industrial Applications of Blockchain”. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 550–555.

[And12] Gavin Andresen. *Pay to Script Hash. BIP-0016*. 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (visited on 05/05/2020).

[Ant14] Andreas M. Antonopoulos. *Mastering Bitcoin. Unlocking Digital Cryptocurrencies*. O’Reilly Media, 2014.

[AW18] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum. Building Smart Contracts and DApps*. O’Reilly Media, 2018.

[Ber+11] Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. *The KECCAK SHA-3 submission*. 2011. URL: https://keccak.team/files/Keccak-submission-3.pdf (visited on 05/05/2020).

[Bin17] Binance. *Binance Exchange*. Whitepaper. 2017. URL: https://www.binance.com/resources/ico/Binance_WhitePaper_en.pdf (visited on 05/05/2020).

[Bin20a] Binance. *Binance Chain Documentation*. 2020. URL: https://docs.binance.org/ (visited on 05/05/2020).

[Bin20b] Binance. *Binance Chain FAQ v0.5*. 2020. URL: https://docs.binance.org/faq/faq.html (visited on 05/05/2020).

[bit17a] bitcoin.it. *Protocol rules*. 2017. URL: https://en.bitcoin.it/wiki/Protocol_rules (visited on 05/05/2020).

[bit17b] bitcoin.it. *Wallet import format*. 2017. URL: https://en.bitcoin.it/wiki/Wallet_import_format (visited on 05/05/2020).

[bit18a] bitcoin.it. *List of address prefixes*. 2018. URL: https://en.bitcoin.it/wiki/List_of_address_prefixes (visited on 05/05/2020).

[bit18b] bitcoin.it. *Satoshi (unit)*. 2018. URL: https://en.bitcoin.it/wiki/Satoshi_(unit) (visited on 05/05/2020).

[bit18c] bitcoin.it. *Weight units*. 2018. URL: https://en.bitcoin.it/wiki/Weight_units (visited on 05/05/2020).

[Bit19] Bitfinex. *Initial Exchange Offering of LEO Tokens*. 2019. URL: https://www.bitfinex.com/wp-2019-05.pdf (visited on 05/05/2020).

[bit19a] bitcoin.it. *Bitcoin Core*. 2019. URL: https://en.bitcoin.it/wiki/Bitcoin_Core (visited on 05/05/2020).

[bit19b] bitcoin.it. *Block timestamp*. 2019. URL: https://en.bitcoin.it/wiki/Block_timestamp (visited on 05/05/2020).

[bit19c] bitcoin.it. *Full node*. 2019. URL: https://en.bitcoin.it/wiki/Full_node (visited on 05/05/2020).

[bit19d] bitcoin.it. *Hash Time Locked Contracts*. 2019. URL: https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts (visited on 05/05/2020).

[bit19e] bitcoin.it. *Miner fees*. 2019. URL: https://en.bitcoin.it/wiki/Miner_fees (visited on 05/05/2020).

[bit19f] bitcoin.it. *Running Bitcoin*. 2019. URL: https://en.bitcoin.it/wiki/Running_Bitcoin (visited on 05/05/2020).

[bit19g] bitcoin.it. *Script*. 2019. URL: https://en.bitcoin.it/wiki/Script (visited on 05/05/2020).

[bit19h] bitcoin.it. *Testnet*. 2019. URL: https://en.bitcoin.it/wiki/Testnet (visited on 05/05/2020).

[bit19i] bitcoin.org. *Bitcoin Developer Examples*. 2019. URL: https://bitcoin.org/en/developer-examples (visited on 05/05/2020).

[bit20a] bitcoin.it. *API reference (JSON-RPC)*. 2020. URL: https://en.bitcoin.it/wiki/API_reference_(JSON-RPC) (visited on 05/05/2020).

[bit20b] bitcoin.org. *Bitcoin Core*. 2020. URL: https://bitcoin.org/en/bitcoin-core/ (visited on 05/05/2020).

[bit20c] bitcoin.org. *Bitcoin Developer Reference*. 2020. URL: https://bitcoin.org/en/developer-reference (visited on 05/05/2020).

[bit20d] bitcoin.org. *Transactions*. 2020. URL: https://bitcoin.org/en/transactions-guide (visited on 05/05/2020).

[Blo18] Block.one. *EOS.IO Technical White Paper v2*. Whitepaper. 2018. URL: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md (visited on 05/05/2020).

[Blo19] Block.one. *Chain API*. 2019. URL: https://developers.eos.io/manuals/eosio.cdt/v1.6/group__crypto (visited on 05/05/2020).

[Blo20a] Block.one. *Consensus Protocol*. 2020. URL: https://developers.eos.io/welcome/latest/protocol/consensus_protocol (visited on 05/05/2020).

[Blo20b] Block.one. *Technical Features*. 2020. URL: https://developers.eos.io/welcome/latest/overview/technical_features (visited on 05/05/2020).

[Boo07] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison Wesley, 2007.

[Bor+18a] Michael Borkowski, Daniel McDonald, Christoph Ritzer, and Stefan Schulte. *Towards Atomic Cross-Chain Token Transfers : State of the Art and Open Questions within TAST*. Whitepaper. 2018. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-1.pdf (visited on 05/05/2020).

[Bor+18b] Michael Borkowski, Christoph Ritzer, Daniel McDonald, and Stefan Schulte. *Caught in Chains : Claim-First Transactions for Cross-Blockchain Asset Transfers*. Whitepaper. 2018. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-2.pdf (visited on 05/05/2020).

[Bor+19a] Michael Borkowski, Philipp Frauenthaler, Marten Sigwart, Taneli Hukkinen, Oskar Hladký, and Stefan Schulte. *Cross-Blockchain Technologies : Review , State of the Art , and Outlook*. Whitepaper. 2019. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-4.pdf (visited on 05/05/2020).

[Bor+19b] Michael Borkowski, Marten Sigwart, Philipp Frauenthaler, Taneli Hukkinen, and Stefan Schulte. "Dextt: Deterministic Cross-Blockchain Token Transfers". In: *IEEE Access* 7 (2019), pp. 111030–111042.

[BP17] Massimo Bartoletti and Livio Pompianu. "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2017, pp. 494–509.

[Bro09] Daniel R. L. Brown. *SEC 1: Elliptic Curve Cryptography*. 2009. URL: https://www.secg.org/sec1-v2.pdf (visited on 05/05/2020).

[BRS18] Michael Borkowski, Christoph Ritzer, and Stefan Schulte. *Deterministic Witnesses for Claim-First Transactions*. Whitepaper. 2018. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-3.pdf (visited on 05/05/2020).

[BS18] Thomas Bocek and Burkhard Stiller. "Smart Contracts – Blockchains in the Wings". In: *Digital Marketplaces Unleashed*. Springer Berlin Heidelberg, 2018, pp. 169–184.

[But14] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Whitepaper. 2014. URL: https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (visited on 05/05/2020).

[CD17] Ignacio Cascudo and Bernardo David. “SCRAPE: Scalable Randomness Attested by Public Entities”. In: *Applied Cryptography and Network Security*. Springer International Publishing, 2017, pp. 537–556.

[CM18] Brad Chase and Ethan MacBrough. *Analysis of the XRP Ledger Consensus Protocol*. 2018. arXiv: `1802.07242 [cs.DC]`.

[Cor19] Bitcoin Core. *Bitcoin Core 0.19.0 RPC*. 2019. URL: `https://bitcoincore.org/en/doc/0.19.0/` (visited on 05/05/2020).

[Cou19] Counterparty. *Protocol Specification*. 2019. URL: `https://github.com/CounterpartyXCP/Documentation/blob/master/Developers/protocol_specification.md` (visited on 05/05/2020).

[Dan17] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017.

[Dav+18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. “Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain”. In: *Advances in Cryptology – EUROCRYPT 2018*. Springer International Publishing, 2018, pp. 66–98.

[DDL18] Bernardo David, Rafael Dowsley, and Mario Larangeira. “Kaleidoscope: An Efficient Poker Protocol with Payment Distribution and Penalty Enforcement”. In: *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2018, pp. 500–519.

[EJN17] Steve Ellis, Ari Juels, and Sergey Nazarov. *ChainLink*. Whitepaper. 2017. URL: `https://link.smartcontract.com/whitepaper` (visited on 05/05/2020).

[EOB19] David Easley, Maureen O’Hara, and Soumya Basu. “From mining to markets: The evolution of bitcoin transaction fees”. In: *Journal of Financial Economics* 134.1 (2019), pp. 91–109.

[FF18] Tiago M. Fernandez-Carames and Paula Fraga-Lamas. “A Review on the Use of Blockchain for the Internet of Things”. In: *IEEE Access* 6 (2018), pp. 32979–33001.

[Fou16] Stellar Development Foundation. *Ledger*. 2016. URL: `https://www.stellar.org/developers/guides/concepts/ledger.html` (visited on 05/05/2020).

[Fou18] TRON Foundation. *Advanced Decentralized Blockchain Platform*. Whitepaper. 2018. URL: `https://tron.network/static/doc/white_paper_v_2_0.pdf` (visited on 05/05/2020).

[Fou19a] Ethereum Foundation. *Types*. 2019. URL: `https://solidity.readthedocs.io/en/v0.6.6/types.html` (visited on 05/05/2020).

[Fou19b] Stellar Development Foundation. *Stellar Smart Contracts*. 2019. URL: `https://www.stellar.org/developers/guides/walkthroughs/stellar-smart-contracts.html` (visited on 05/05/2020).

[Fou19c] Stellar Development Foundation. *Transactions*. 2019. URL: https://www.stellar.org/developers/guides/concepts/transactions.html (visited on 05/05/2020).

[Fou20a] Cardano Foundation. *Addresses in Cardano SL*. 2020. URL: https://docs.cardano.org/cardano/addresses/ (visited on 05/05/2020).

[Fou20b] Cardano Foundation. *Cardano Settlement Layer Documentation*. 2020. URL: https://docs.cardano.org/introduction/ (visited on 05/05/2020).

[Fou20c] Cardano Foundation. *Cardano SL Technical Details*. 2020. URL: https://docs.cardano.org/technical/ (visited on 05/05/2020).

[Fou20d] Cardano Foundation. *Plutus Introduction*. 2020. URL: https://docs.cardano.org/technical/plutus/introduction/ (visited on 05/05/2020).

[Fou20e] Ethereum Foundation. *Contract ABI Specification*. 2020. URL: https://solidity.readthedocs.io/en/v0.6.6/abi-spec.html (visited on 05/05/2020).

[Fou20f] Ethereum Foundation. *Expressions and Control Structures*. 2020. URL: https://solidity.readthedocs.io/en/v0.6.6/control-structures.html (visited on 05/05/2020).

[Fra+19] Philipp Frauenthaler, Marten Sigwart, Michael Borkowski, Taneli Hukkinen, and Stefan Schulte. *Towards Efficient Cross-Blockchain Token Transfers*. Whitepaper. 2019. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-5.pdf (visited on 05/05/2020).

[Fra+20] Philipp Frauenthaler, Marten Sigwart, Christof Spanring, and Stefan Schulte. *Leveraging Blockchain Relays for Cross-Chain Token Transfers*. Whitepaper. 2020. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-8.pdf (visited on 05/05/2020).

[Fry17] Shaolin Fry. *Mandatory activation of segwit deployment. BIP-0148*. 2017. URL: https://github.com/bitcoin/bips/blob/master/bip-0148.mediawiki (visited on 05/05/2020).

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GH04] Henri Gilbert and Helena Handschuh. “Security Analysis of SHA-256 and Sisters”. In: *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2004, pp. 175–193.

[GM11] Atefeh Gholipour and Sattar Mirzakuchaki. “A Pseudorandom Number Generator with KECCAK Hash Function”. In: *International Journal of Computer and Electrical Engineering* (2011), pp. 896–899.

[Goo14] L. M. Goodman. *Tezos - a self-amending crypto-ledger*. Whitepaper. 2014. URL: https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf (visited on 05/05/2020).

[Her18] Maurice Herlihy. "Atomic Cross-Chain Swaps". In: *2018 ACM Symposium on Principles of Distributed Computing (PODC '18)*. ACM Press, 2018, pp. 245–254.

[Her19] Maurice Herlihy. "Blockchains from a Distributed Computing Perspective". In: *Commun. ACM* 62.2 (2019), pp. 78–85.

[Hes00] Philipp Hess. *SEC 2: Recommended Elliptic Curve Domain Parameters*. 2000. URL: https://www.secg.org/SEC2-Ver-1.0.pdf (visited on 05/05/2020).

[INC17] INCITS. *INCITS 4-1986[R2017]: Information Systems - Coded Character Sets - 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*. 2017. URL: https://standards.incits.org/apps/group_public/project/details.php?project_id=1829 (visited on 05/05/2020).

[ITU15] ITU-T. *X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. 2015. URL: https://www.itu.int/rec/T-REC-X.690-201508-I/en (visited on 05/05/2020).

[Jam18] Febin John James. *Ripple Quick Start Guide*. Packt Publishing, 2018.

[JDX18] Hai Jin, Xiaohai Dai, and Jiang Xiao. "Towards a Novel Architecture for Enabling Interoperability amongst Multiple Blockchains". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 1203–1211.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1 (2001), pp. 36–63.

[JRB19] Sandra Johnson, Peter Robinson, and John Brainard. *Sidechains and interoperability*. 2019. arXiv: 1903.04077 [cs.CR].

[Kia+17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *Advances in Cryptology – CRYPTO 2017*. Springer International Publishing, 2017, pp. 357–388.

[KMZ17] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. "Non-Interactive Proofs of Proof-of-Work". In: *IACR Cryptology ePrint Archive* 2017.963 (2017), pp. 1–42.

[KP19] Tommy Koens and Erik Poll. "Assessing interoperability solutions for distributed ledgers". In: *Pervasive and Mobile Computing* 59 (2019), pp. 1–10.

[Li+17] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. “A survey on the security of blockchain systems”. In: *Future Generation Computer Systems* (2017).

[Liu+19] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, and Yih-Chun Hu. *HyperService: Interoperability and Programmability Across Heterogeneous Blockchains*. 2019. arXiv: `1908.09343 [cs.CR]`.

[LLW15] Eric Lombrozo, Johnson Lau, and Pieter Wuille. *Segregated Witness (Consensus layer). BIP-0141*. 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki` (visited on 05/05/2020).

[Lu+19] Donghang Lu, Pedro Moreno-Sanchez, Amanuel Zeryihun, Shivam Bajpayi, Sihao Yin, Ken Feldman, Jason Kosofsky, Pramita Mitra, and Aniket Kate. “Reducing Automotive Counterfeiting Using Blockchain: Benefits and Challenges”. In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 2019, pp. 39–48.

[LW16] Eric Lombrozo and Pieter Wuille. *Segregated Witness (Peer Services). BIP-0144*. 2016. URL: `https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki` (visited on 05/05/2020).

[LXS19] Mengyi Li, Lirong Xia, and Oshani Seneviratne. “Leveraging Standards Based Ontological Concepts in Distributed Ledgers: A Healthcare Smart Contract Example”. In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 2019, pp. 152–157.

[Maz16] David Mazières. *The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus*. Whitepaper. 2016. URL: `https://www.stellar.org/papers/stellar-consensus-protocol` (visited on 05/05/2020).

[Men+18a] Jan Mendling, Ingo Weber, Wil Van Der Aalst, Jan Vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, et al. “Blockchains for Business Process Management - Challenges and Opportunities”. In: *ACM Trans. Manage. Inf. Syst.* 9.1 (2018).

[Men+18b] Weizhi Meng, Jianfeng Wang, Xianmin Wang, Joseph Liu, Zuoxia Yu, Jin Li, Yongjun Zhao, and Sherman S. M. Chow. “Position Paper on Blockchain Technology: Smart Contract and Applications”. In: *Network and System Security*. Springer International Publishing, 2018, pp. 474–483.

[Met18] Metronome. *Metronome: Owner’s Manual*. June 2018. URL: `https://www.metronome.io/download/owners_manual.pdf` (visited on 05/05/2020).

[Nak08] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Whitepaper. 2008.

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

[Neo19] Neo. *NeoContract White Paper*. 2019. URL: https://docs.neo.org/docs/en-us/basic/technology/neocontract.html (visited on 05/05/2020).

[Neo20a] Neo. *Enum OpCode*. 2020. URL: https://docs.neo.org/developerguide/en/api/Neo.VM.OpCode.html (visited on 05/05/2020).

[Neo20b] Neo. *Neo White Paper*. 2020. URL: https://docs.neo.org/docs/en-us/basic/whitepaper.html (visited on 05/05/2020).

[Neo20c] Neo. *NeoVM*. 2020. URL: https://docs.neo.org/docs/en-us/basic/technology/neovm.html (visited on 05/05/2020).

[Neo20d] Neo. *Wallets*. 2020. URL: https://docs.neo.org/developerguide/en/articles/wallets.html (visited on 05/05/2020).

[Nof+17] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. "Blockchain". In: *Business & Information Systems Engineering* 59.3 (2017), pp. 183–187.

[Ora19a] Oracle. *Class Exception*. 2019. URL: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Exception.html (visited on 05/05/2020).

[Ora19b] Oracle. *Default Methods*. 2019. URL: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html (visited on 05/05/2020).

[Ora19c] Oracle. *Defining Methods*. 2019. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html (visited on 05/05/2020).

[Ora19d] Oracle. *Interface ScheduledExecutorService*. 2019. URL: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/concurrent/ScheduledExecutorService.html (visited on 05/05/2020).

[Ora19e] Oracle. *Interface SortedSet<E>*. 2019. URL: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/SortedSet.html (visited on 05/05/2020).

[Ora19f] Oracle. *Module java.rmi*. 2019. URL: https://docs.oracle.com/en/java/javase/12/docs/api/java.rmi/module-summary.html (visited on 05/05/2020).

[Ora19g] Oracle. *Package java.net*. 2019. URL: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/net/package-summary.html (visited on 05/05/2020).

[Ora19h] Oracle. *Using the this Keyword*. 2019. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html (visited on 05/05/2020).

[Ora20] Oracle. *Tuning Java Virtual Machines (JVMs)*. 2020. URL: https://docs.oracle.com/cd/E15523_01/web.1111/e13814/jvm_tuning.htm (visited on 05/05/2020).

[Pér+19] Cristina Pérez-Solà, Sergi Delgado-Segura, Jordi Herrera-Joancomartí, and Guillermo Navarro-Arribas. *Analysis of the SegWit adoption in Bitcoin*. Mimeo, 2019. URL: http://deic.uab.es/~guille/publications/papers/2018.recsi.segwit.pdf (visited on 05/05/2020).

[Pro18] Litecoin Project. *Litecoin*. 2018. URL: https://litecoin.info/index.php/Litecoin (visited on 05/05/2020).

[Pro19a] The Monero Project. *Monero Technical Specs*. 2019. URL: https://monerodocs.org/technical-specs/ (visited on 05/05/2020).

[Pro19b] XRP Ledger Project. *Ledger Header*. 2019. URL: https://xrpl.org/ledger-header.html (visited on 05/05/2020).

[Pro19c] XRP Ledger Project. *Transaction Common Fields*. 2019. URL: https://xrpl.org/transaction-common-fields.html (visited on 05/05/2020).

[Pro19d] XRP Ledger Project. *XRP Ledger Overview*. 2019. URL: https://xrpl.org/xrp-ledger-overview.html (visited on 05/05/2020).

[Pry+20] Christoph Prybila, Stefan Schulte, Christoph Hochreiner, and Ingo Weber. "Runtime verification for business processes utilizing the Bitcoin blockchain". In: *Future Generation Computer Systems* 107 (2020), pp. 816–831.

[QAN19] Ilham A. Qasse, Manar Abu Talib, and Qassim Nasir. "Inter Blockchain Communication: A Survey". In: *Proceedings of the ArabWIC 6th Annual International Conference Research Track*. Association for Computing Machinery, 2019.

[Sch+19] Stefan Schulte, Marten Sigwart, Philipp Frauenthaler, and Michael Borkowski. "Towards Blockchain Interoperability". In: *Business Process Management: Blockchain and Central and Eastern Europe Forum*. Springer International Publishing, 2019, pp. 3–10.

[Sig+19a] Marten Sigwart, Philipp Frauenthaler, Taneli Hukkinen, and Stefan Schulte. *Towards Cross-Blockchain Transaction Verifications*. Whitepaper. 2019. URL: https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-6.pdf (visited on 05/05/2020).

[Sig+19b] Marten Sigwart, Philipp Frauenthaler, Christof Spanring, and Stefan Schulte. *Preparing Simplified Payment Verifications for Cross-Blockchain Token Transfers*. Whitepaper. 2019. URL: `https://www.dsg.tuwien.ac.at/projects/tast/pub/tast-white-paper-7.pdf` (visited on 05/05/2020).

[Sir+19] Vasilios A. Siris, Pekka Nikander, Spyros Voulgaris, Nikos Fotiou, Dmitrij Lagutin, and George C. Polyzos. "Interledger Approaches". In: *IEEE Access* 7 (2019), pp. 89948–89966.

[Sma16] Nigel P. Smart. *Cryptography Made Simple*. Springer International Publishing, 2016.

[Str13] Bjarne Stroustrup. *The C++ programming language*. 4th ed. Addison Wesley, 2013.

[SYB14] David Schwartz, Noah Youngs, and Arthur Britto. *The Ripple Protocol Consensus Algorithm*. Whitepaper. 2014. URL: `https://ripple.com/files/ripple_consensus_whitepaper.pdf` (visited on 05/05/2020).

[Tet16] Tether. *Tether: Fiat currencies on the Bitcoin blockchain*. 2016. URL: `https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf` (visited on 05/05/2020).

[Tez20a] Tezos. *Michelson: the language of Smart Contracts in Tezos*. 2020. URL: `https://tezos.gitlab.io/whitedoc/michelson.html` (visited on 05/05/2020).

[Tez20b] Tezos. *Tezos Developer Portal*. 2020. URL: `https://developers.tezos.com/` (visited on 05/05/2020).

[TS16] Florian Tschorsch and Bjorn Scheuermann. "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies". In: *IEEE Communications Surveys Tutorials* 18.3 (2016), pp. 2084–2123.

[WCL17] Hui Wang, Yuanyuan Cen, and Xuefeng Li. "Blockchain Router: A Cross-Chain Communication Protocol". In: *Proceedings of the 6th International Conference on Informatics, Environment, Energy and Applications*. Association for Computing Machinery, 2017, pp. 94–97.

[Wil+19] JR Willett, Maran Hidskes, David Johnston, Ron Gross, and Marv Schneider. *Omni Protocol Specification*. 2019. URL: `https://github.com/OmniLayer/spec` (visited on 05/05/2020).

[WM17] Pieter Wuille and Greg Maxwell. *Base32 address format for native v0-16 witness outputs. BIP-0173*. 2017. URL: `https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki` (visited on 05/05/2020).

[Woo14] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. Yellowpaper. 2014. URL: `https://gavwood.com/paper.pdf` (visited on 05/05/2020).

[Woo16] Gavin Wood. *PolkaDot: Vision for a Heterogeneous Multi-Chain Framework.* Whitepaper. 2016. URL: `https://polkadot.network/PolkaDotPaper.pdf` (visited on 05/05/2020).

[Wui15] Pieter Wuille. *Strict DER signatures. BIP-0066.* 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki` (visited on 05/05/2020).

[Yli+16] Jesse Yli-Huumo, Deokyoon Ko, Sujin Choi, Sooyong Park, and Kari Smolander. “Where Is Current Research on Blockchain Technology?—A Systematic Review”. In: *PLOS ONE* 11.10 (2016), e0163477.

[ZAA19] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. *Atomic Commitment Across Blockchains.* 2019. arXiv: `1905.02847 [cs.DB]`.

[Zam+18] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J. Knottenbelt. *XCLAIM: Trustless, Interoperable Cryptocurrency-Backed Assets.* Cryptology ePrint Archive, Report 2018/643. `https://eprint.iacr.org/2018/643`. 2018.

[Zam+19] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. *SoK: Communication Across Distributed Ledgers.* Cryptology ePrint Archive, Report 2019/1128. `https://eprint.iacr.org/2019/1128`. 2019.

[ZL20] Dongfang Zhao and Tonglin Li. *Distributed Cross-Blockchain Transactions.* 2020. arXiv: `2002.11771 [cs.DB]`.

[Zoh15] Aviv Zohar. “Bitcoin: Under the Hood”. In: *Communications of the ACM* 58.9 (2015), pp. 104–113.

[ZW17] Taiyang Zhang and Loong Wang. *Republic Protocol: A Decentralized Dark Pool Exchange Providing Atomic Swaps for Ethereum-Based Assets and Bitcoin.* Whitepaper. 2017. URL: `https://releases.republicprotocol.com/whitepaper/1.0.0/whitepaper_1.0.0.pdf` (visited on 05/05/2020).