Coincer: The Decentralized Cryptocurrency Exchange

Master Thesis

Bc. Michal Zima

Brno, 2015
Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Eva Hladká, Ph.D.
Acknowledgement

I would like to express my gratitude to my advisor Eva Hladká for her kind guidance, huge patience, and a lot of helpful advice. My gratitude also belongs to Matúš Madzin for his great help with modelling and verification of Coincer’s communication protocol. I am also very grateful to Petr Švenda and Zdeněk Říha for their openness and greatly valuable consultations of the atomic protocol.
Abstract

This thesis focuses on the topic of decentralization of exchange of different kinds of cryptocurrencies. It elaborates on the protocol for a trustless exchange across two separate blockchains, and shows its atomicity. On top of this protocol is built a peer-to-peer platform for decentralized exchange of cryptocurrencies—a P2P overlay and a communication protocol are designed. Several experiments are conducted against an implementation of the proposed platform.
Keywords

Bitcoin, altcoins, cryptocurrencies, blockchain, atomic exchange, cross-chain exchange, decentralized exchange, peer-to-peer overlay, P2P.
Contents

1 Introduction .......................................................... 1

2 Cryptocurrencies .................................................... 2
  2.1 Scripting Language ............................................. 2
  2.1.1 Overview of Basic Script Operations .................... 2
  2.2 Transactions .................................................... 4
    2.2.1 Time Locks ................................................. 4
  2.3 Blocks .......................................................... 4

3 Atomic Cross-chain Exchange ..................................... 6
  3.1 Protocol ......................................................... 6
    3.1.1 Notation .................................................... 6
    3.1.2 Description ............................................... 6
    3.1.3 Atomic Rollbacks ......................................... 10
      1. Before Alice broadcasts tx1 ............................... 10
      2. After Alice broadcasts tx1, but before Bob broadcasts tx3 . . . . . ... 10
      3. After tx1 and tx3 broadcast, but before Alice broadcasts tx5 .......... 10
      4. After Alice broadcasts tx5 .................................. 10
  3.2 Protocol Atomicity ............................................. 10
    3.2.1 Liveness ................................................... 11
    3.2.2 Safety ..................................................... 12

4 P2P Overlay ........................................................ 14
  4.1 Requirements and Design Goals ................................ 14
    4.1.1 Anonymity of Peers ...................................... 14
    4.1.2 Communication Between Peers ............................ 15
    4.1.3 Efficient Routing ......................................... 15
    4.1.4 NAT Traversal ............................................. 16
    4.1.5 Redundancy ................................................. 16
  4.2 Possible Solutions ............................................. 16
    4.2.1 Types of Ad-Hoc Routing Protocols ...................... 17
      Proactive Protocols .......................................... 17
      Reactive Protocols .......................................... 17
      Hybrid Protocols ............................................ 17
    4.2.2 Optimized Link State Routing Protocol (OLSR) ......... 18
4.2.3 Destination-Sequence Distance Vector Protocol (DSDV) ........................................ 18
4.2.4 Babel .................................................. 18
4.2.5 B.A.T.M.A.N. .......................................... 19
4.3 Proposal .................................................. 19
4.3.1 Loops .................................................. 20
4.3.2 Routing Table Updates ................................. 20
4.3.3 Graceful Departure ..................................... 21
4.3.4 Attacks ................................................ 21
   Man-in-the-Middle Attack ................................ 21
   Replay Attack .......................................... 22
   Sybil Attack .......................................... 23
   Denial of Service Attack ................................. 24
5 Communication Protocol ................................. 25
5.1 Protocol Form ........................................... 25
5.2 Basic Protocol Format .................................. 25
5.3 Inner Message Format and Data .......................... 27
   5.3.1 Types of Messages ................................... 27
   5.3.2 Messages of Type p2p ................................. 28
       p2p.hello .............................................. 28
       p2p.bye .............................................. 29
       p2p.iamhere ......................................... 29
       p2p.whereareyou ..................................... 30
       p2p.peers.sol ....................................... 30
       p2p.peers.adv ...................................... 31
   5.3.3 Messages of Type market ............................ 31
       market.order ......................................... 31
       market.cancel ....................................... 32
       market.sol .......................................... 33
       market.adv .......................................... 34
   5.3.4 Messages of Type trade ............................. 34
       trade.proposal ....................................... 35
       trade.accept ......................................... 35
       trade.reject ........................................ 36
       trade.px .............................................. 36
       trade.key ............................................ 37
       trade.sigreq ......................................... 38
       trade.sig ............................................ 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Trading Protocol</td>
<td>39</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Model of a Successful Protocol Run</td>
<td>41</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Model of Erroneous Situation #1: Interruption Before Going into Blockchain</td>
<td>41</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Model of Erroneous Situation #2: Missing Second Commitment</td>
<td>44</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Model of Erroneous Situation #3: No Coins Are Claimed</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Cryptography</td>
<td>44</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Signing</td>
<td>44</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Encryption</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Protocol Extensibility</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>Analysis</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>Testing Environment</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Attachments</td>
<td>55</td>
</tr>
</tbody>
</table>
1 Introduction

Few years after the Bitcoin had started, more and more alternative cryptocurrencies began showing up [1]. In most cases, those were forks of Bitcoin’s original code and basically differed from Bitcoin only in a couple of parameters, such as coin supply cap, shorter interval of blocks, or a hashing function. Nevertheless, they attracted some users and started their life of a cryptocurrency. In contrast to Bitcoin, their userbase was very small, so it was more beneficial to exchange them with highly divisible bitcoins better than with any fiat currency which would be rather impractical.

While handling fiat money acted as a driving force towards higher security of bitcoin exchanges, newly emerging cryptocurrency exchanges that avoided this element in their businesses also lacked this force. As a result, many of them were successfully cracked and big amounts of coins were stolen. These losses account to millions of US dollars [2].

As centralized exchanges continually fail to provide a secure way of exchanging different kinds of coins, it is natural to seek different approaches to this problem. However, traditional methods come with different kinds of problems such as a need for trust between parties. Nonetheless, cryptocurrencies introduce cryptographic features that could help overcome this weak point.

In this thesis I devise a system for a decentralized and trustless exchange of cryptocurrencies. In such a system, users are all the time either the solely holders of their coins possessing a full control over them, or they are (atomically) trading them. The system itself does not create any new attack vectors without providing a safe and effective countermeasure. Therefore, it forms a safer alternative for common centralized exchanges.
2 Cryptocurrencies

Cryptocurrencies are a novel concept devised by Satoshi Nakamoto, an unknown creator of the first cryptocurrency Bitcoin. A cryptocurrency is a distributed transactional system based on a shared public ledger. Transactions in such a system are chained together by links. Each transaction carries a set of rules (called a script) under which it can be spent. It also provides necessary information that proves it can spend its preceding transactions. Most common usage of this mechanism is to require a signature and a public key to a given public key hash.

2.1 Scripting Language

Transactions in cryptocurrencies are very flexible. They are not sent from a certain sender to a specific recipient or account/address. They are rather addressed to a script, and can be spent by whoever is able to supply to the script such data that it evaluates to true.

Compared with standard programming languages, this scripting language is simplified—it does not include cycles nor procedures, but it does include, among other things, a stack, a branching operator and several cryptographic functions. As a result, cryptocurrencies can be used in a wide variety of use cases. One of them is shown in this work.

The scripting language is syntactically based on reverse Polish notation. All data is put on a stack, and operations work only with the data elements stored on the top of the stack. Reading from the stack is destructive. If an operation returns a value, it stores the result on the stack. Some operations could also abort the evaluation of the script.

2.1.1 Overview of Basic Script Operations

This overview covers only these operations that are used later in this work. For a complete reference of script operations used in Bitcoin see [3] or [4].
OP_0  An empty value is pushed onto the stack.

OP_FALSE  This is an alias for OP_0.

OP_TRUE  Pushes the value “1” onto the stack.

OP_IF  Executes following statements if the top of the stack is not FALSE.

OP_ELSE  Executes only if the previous statements were not executed.

OP_ENDIF  Ends the OP_IF, OP_ELSE block.

OP_VERIFY  Checks the top of the stack, halts and invalidates the script unless the value was TRUE.

OP_EQUAL  Pushes TRUE (1) if top two stack items are equal, pushes FALSE (0) otherwise.

OP_EQUALVERIFY  Same as OP_EQUAL, but also runs OP_VERIFY afterward so as to halt unless the result was TRUE.

OP_HASH160  Returns RIPEMD160(SHA256(x)) hash of the top item.

OP_CHECKSIG  Pops a public key and a signature, and validates the signature for the transaction’s hashed data, returns TRUE if matching.

OP_CHECKMULTISIG  Validates the first signature against each of the public keys until it finds a match. Starting with the subsequent public key, validates the second signature against each remaining public key until it finds a match. All signatures need to match a public key. If all signatures are valid, returns TRUE, FALSE otherwise. Due to a bug, one extra unused value is removed from the stack, prefix with OP_0 as a workaround.
2. Cryptocurrencies

2.2 Transactions

Most cryptocurrencies do not use subtractive system for accounting amounts. Instead, they utilize a very specific system of inputs and outputs. Coins that are being spent enter the transaction as inputs. Analogically, coins that leave the transaction do so as outputs. Each output is of a certain amount and carries its own script. Outputs are spent in other transactions as inputs—here each input has to provide data for the script to evaluate it as true.

In 2012, Bitcoin introduced a special construction called P2SH\(^1\) that allowed outputs to include only a hash of a script. The script itself then had to be provided in the following input as the top stack data element [5]. In 2014, P2SH was chosen as a way to enable most of custom scripts as standard [6]. Although it was always possible to use valid non-standard scripts, transactions that include them are not broadcast by most peers. Standardization of P2SH scripts therefore greatly influences practical usability of Coincer.

2.2.1 Time Locks

Each transaction contains an information about its earliest validity time. Normally, there is no such constraint and transactions are valid at any time. If a transaction is locked to a specific time or a block number, it cannot be used before its lock expires. After that moment there is no difference between this transaction and a transaction without a time lock.

2.3 Blocks

Individual transactions are verified and in average every time frame grouped into a block of transactions (in Bitcoin this is every 10 minutes). Each block must satisfy a condition that its double hash is lower than a certain value determined by the network. This way the cryptocurrency self-regulates creation of blocks to said time frames. All that regardless of the computing power of the network, because

---

1. Pay to script hash.
the lower the target hash is, the more times a nonce inside of a block has to be incremented and the block hashed.

Blocks are linked into a *blockchain*. The deeper is a block in the chain, the lower is the probability of occurring of a fork of the chain that would displace the block.
3 Atomic Cross-chain Exchange

This chapter covers the topic of how to actually exchange two different cryptocurrencies (e.g., bitcoins for litecoins) without any third party, utilizing only the blockchain and the scripting language of respective cryptocurrencies. Atomic in this context means that either both trading parties receive exchanged coins or their original coins can be returned to them in a guaranteed fashion.

I build on previous work of TierNolan, who proposed two protocols for a cross-chain exchange [7]—one atomic and one non-atomic [8]. The atomic one is to be elaborated later in this chapter, while the non-atomic protocol is not covered in this thesis at all.

3.1 Protocol

The following protocol allows two mutually untrusted parties to atomically exchange their coins across two separate blockchains.

3.1.1 Notation

Transactions used within the protocol are denoted $tx1$, $tx2$, etc., $x$ is a variable, three capital letters, e.g., $ABC$, denote a cryptocurrency.

Values in scripts which are not prefixed with “OP_” are data elements to be pushed onto the stack. These values also implicitly imply that they are preceded by appropriate push script operations in a binary script representation. This notation is treated as standard in cryptocurrencies [4].

3.1.2 Description

Let $ATC$ and $BTC$ be two cryptocurrencies, $Alice$ and $Bob$ be two trading parties. Then let’s presume:

1. Both cryptocurrencies $ATC$ and $BTC$ support all script operations that are used in this protocol.

2. Semantics of script operations hold.
3. Cryptocurrency transactions are not malleable\(^1\).

4. Alice wants to exchange \(N\) ATC for \(M\) BTC.

5. Bob wants to exchange \(M\) BTC for \(N\) ATC.

6. Alice and Bob generate new key pairs solely for the purpose of the exchange and deliver their public keys to the other party.

7. Alice starts the protocol.

Alice generates a cryptographically secure random number \(x\) and keeps it secret. Then she creates a transaction \(tx1\). Its output script is of P2SH format and is as follows:

\[
\text{OP\_HASH160} \\
<\text{RIPEMD160}(\text{SHA256}(\text{redeem script 1})))> \\
\text{OP\_EQUAL}
\]

The referenced \textit{redeem script 1} is as follows:

\[
\text{OP\_IF} \\
2 <\text{Alice’s public key}> <\text{Bob’s public key}> 2 \\
\text{OP\_CHECKMULTISIG} \\
\text{OP\_ELSE} \\
\text{OP\_HASH160} <\text{RIPEMD160}(\text{SHA256}(x)))> \text{OP\_EQUALVERIFY} \\
<\text{Bob’s public key}> \text{OP\_CHECKSIG} \\
\text{OP\_ENDIF}
\]

If there is true value on the top of the stack, this script does only multisignature validation. For the given two public keys it requires two signatures—one for each of them. Otherwise, if the top stack value is false, i.e., empty, 0, or −0, the script applies hashing functions SHA256 and RIPEMD160 to the next top stack value, and compares the result with hashed \(x\) value. Afterwards, if this check successfully passed, validation of Bob’s signature is performed.

By transaction \(tx1\) Alice commits her ATC into the exchange. But she does not publish it yet.

---

\(^1\) Transaction malleability problem is described in detail in [9, 10].
Alice then creates a transaction \( tx2 \) that takes \( tx1 \) as an input and spends all of its ATC, with the following input script, except for the signatures:

\[
\text{OP}_0 \text{ <Alice’s signature> <Bob’s signature> OP_TRUE}
\]

This script pushes five data elements onto the stack, the output script in transaction \( tx1 \) then checks the serialized script on the top and executes it. \( \text{True} \) value, that comes next on the top of the stack, then triggers execution of the first branch of the script, which will consume the rest of the stack in multiple-signature checking procedure.

Besides the script, this transaction also carries a time lock set to a safely high value. The transaction serves the purpose of an atomic rollback. If the trade does not proceed before the time lock expires, this transaction transfers all coins back to Alice when broadcast.

Alice transmits this unsigned transaction to Bob and asks him to sign it. Bob verifies the contents of the transaction, signs it, and sends the signature back to Alice. Bob also extracts the value of the hashed \( x \) from \textit{redeem script 1}. Alice completes the transaction by signing it herself as well.

Next, Bob creates a transaction \( tx3 \) by which he will commit his BTC into the exchange. Its main output script is as follows:

\[
\text{OP\_HASH160} \\
<\text{RIPEMD160(SHA256(redeem script 2))}> \\
\text{OP\_EQUAL}
\]

With the \textit{redeem script 2} being as follows:

\[
\text{OP\_IF} \\
2 \text{ <Bob’s public key> <Alice’s public key> 2} \\
\text{OP\_CHECKMULTISIG} \\
\text{OP\_ELSE} \\
\text{OP\_HASH160 <RIPEMD160(SHA256(x))> OP\_EQUALVERIFY} \\
<\text{Alice’s public key}> \text{OP\_CHECKSIG} \\
\text{OP\_ENDIF}
\]


In this script Bob uses the hash of $x$ as it was included in the transaction $tx2$. He does not know the actual value of $x$ so he could not hash it himself. For the transaction $tx3$ Bob also creates a rollback transaction $tx4$. It uses $tx3$ as an input and spends all of its BTC, with the following input script, again except for the signatures:

\[ \text{OP}_0 \ <\text{Bob’s signature}> \ <\text{Alice’s signature}> \ \text{OP}_\text{TRUE} \ <\text{redeem script 2}> \]

This transaction, analogically to $tx2$, also carries a time lock. But this time its value is lower than it was in $tx2$—we can consider a half of the value in $tx2$. Setting it safely lower prevents from a possible time race condition.

Bob sends this unsigned transaction to Alice to have it signed by her. Alice checks its contents and returns her signature to Bob. He then completes the transaction by adding also his signature.

After sending the signature to Bob, Alice broadcasts her transaction $tx1$ to the cryptocurrency network. From this moment on no direct communication between Alice and Bob is necessary for finishing the trade. Bob waits till the transaction gets placed into the blockchain and receives a sufficiently safe number of confirmations. Then he broadcasts his transaction $tx3$. Now it is Alice who waits for this transaction getting into the blockchain and having enough confirmations. When this occurs, Alice creates transaction $tx5$ that redeems all BTC from $tx3$ with this input script:

\[ <\text{Alice’s signature}> \ <x> \ \text{OP}_\text{FALSE} \ <\text{redeem script 2}> \]

As opposed to the input script of the transaction $tx4$, this script triggers execution of the second branch of the redeem script 2. By broadcasting this transaction Alice reveals the value of $x$, therefore Bob can also create a transaction ($tx6$) that will redeem ATC from $tx1$. Its input script will be as follows:

\[ <\text{Bob’s signature}> \ <x> \ \text{OP}_\text{FALSE} \ <\text{redeem script 1}> \]

By now, the trade is complete.
3.1.3 Atomic Rollbacks

At any moment the trade can be either finished without cooperation of the other party, or canceled. The whole time frame can be split into the following four cases:

1. Before Alice broadcasts \textit{tx1}
   
   As long as nothing has been committed into the blockchain, canceling a trade is a trivial action, i.e., nothing needs to be done.

2. After Alice broadcasts \textit{tx1}, but before Bob broadcasts \textit{tx3}
   
   If Bob refuses to continue with the trade, Alice has to wait until the time lock of the transaction \textit{tx2} expires. Then she can broadcast it and obtain her ATC coins back.

3. After \textit{tx1} and \textit{tx3} broadcast, but before Alice broadcasts \textit{tx5}
   
   It might occur that despite the fact that Alice committed her coins, she refuses to finish the trade. In such a case, both Alice and Bob must wait till timeouts of their rollback transactions (\textit{tx2} and \textit{tx4}) expire, because Alice did not reveal the value of $x$. Then they can broadcast them and retrieve their coins.

4. After Alice broadcasts \textit{tx5}
   
   In this last case, Alice already claimed Bob’s BTC and revealed $x$. Therefore, the only possibility for Bob is to finish the trade by creating and broadcasting the transaction \textit{tx6}.

3.2 Protocol Atomicity

The protocol should have the following two properties to be atomic and complete (i.e., not allowing any other than specified actions):

\textbf{Liveness} There always eventually exists a step each party can do either to make progress within the protocol, or to revert its effects.
Safety The protocol never deadlocks, and neither party can acquire both parties’ coins from the trade.

3.2.1 Liveness

I want to disprove that one of the parties can perform such an action that the other party cannot perform any action in a finite time. Therefore, I will do an analysis for each step of the protocol.

As the very first thing, Alice generates $x$ and creates a transaction $tx1$. If Alice does not generate a number, the protocol stops—Bob can trivially terminate it—, or continues if Alice provides a value, but Alice may lose her coins if $x$ is not a cryptographically secure random number and Bob can figure its value out\(^2\).

If Alice does not create the transaction $tx1$, Bob trivially terminates the protocol; if she creates a transaction with a different script, Bob trivially terminates the protocol when he verifies the content of her transaction $tx2$, which contains this script. Other parts of $tx1$ Bob verifies as soon as Alice broadcasts it—if something is different from the expected form, Bob trivially terminates the protocol, while Alice has to wait for the time lock of the transaction $tx2$ to expire to claim her coins back.

In the next step, Alice creates the transaction $tx2$. If she deviates in any aspect of the transaction from the expected form, Bob trivially terminates the protocol at the moment he verifies its content. The same applies for cases when Alice does not send Bob anything or sends him something different than the transaction $tx2$. If Bob does not return Alice a valid signature for $tx2$, she also trivially terminates the protocol.

Then, it is Bob’s turn to create transactions $tx3$ and $tx4$. If he does not create transaction $tx3$, Alice trivially terminates the protocol; if he creates a transaction with different script, Alice, again, trivially terminates the protocol when she verifies the content of the transaction $tx4$, which contains this script; other parts of $tx3$ Alice verifies as soon as Bob broadcasts it—if something is different from the expected form, Alice waits for expiration of the time lock of the trans-

\(^2\) By this action, Alice violates the protocol, but as she cannot harm Bob, it does not break the given property.
action $tx2$ to claim back her coins from the trade, and Bob has to wait for expiration of the time lock of his transaction $tx4$.

If Bob’s transaction $tx4$ anyhow deflects from the expected form, Alice trivially terminates the protocol at the moment she verifies its content. The same applies for cases when Bob does not send Alice anything or sends her something different than the transaction $tx4$. If Alice does not return Bob a valid signature for $tx4$, he trivially terminates the protocol.

After sending Bob the signature for the transaction $tx4$, Alice is supposed to broadcast her transaction $tx1$, otherwise Bob trivially terminates the protocol. If she does not broadcast the transaction $tx1$ in the expected form, Bob, again, trivially terminates the protocol.

If Bob does not broadcast the transaction $tx3$ until expiration of the time lock of the transaction $tx4$, or the transaction is different than expected, Alice waits for expiration of the time lock of her transaction $tx2$ and uses it to claim her coins from the trade back.

If then Alice does not claim Bob’s coins from his transaction $tx3$ before elapsing the time lock of the transaction $tx4$, Bob terminates the protocol by broadcasting $tx4$. But, if Bob does not claim Alice’s coins from her transaction $tx1$, he might lose the coins as soon as the time lock of the transaction $tx2$ expires.$^3$

### 3.2.2 Safety

Thanks to liveness, the protocol is deadlock-free. Therefore, it only needs to be shown that no trading party may lose their coins while they conform to the protocol.

Coins are sent to the trade only via transactions $tx1$ and $tx3$, with equivalent output and redeem scripts.

\[
\begin{align*}
\text{OP\_HASH160} \\
<\text{RIPEMD160(SHA256(redeem script 1))}> \\
\text{OP\_EQUAL}
\end{align*}
\]

Spending coins from a transaction requires supplying such data onto the stack that this script evaluates to $true$. This specific script is a P2SH script, i.e., it requires a script which has a hash equaling

---

3. Again, this protocol violation cannot harm Alice.
3. Atomic Cross-chain Exchange

to the hash provided in the script above. Then, if hashes match, this script is executed.

\[
\text{OP\_IF}
\text{OP\_ELSE}
\text{OP\_ENDIF}
\]

To spend the coins, it is required by this script to execute exactly one of its branches. The first branch is controlled by both parties, because it requires valid signatures from both of them. When executed by a transaction with active time lock feature, it is guaranteed that such a transaction is invalid before the specified time. Therefore, it is safe to the other party to sign transaction \( tx2 \) (or \( tx4 \)), because it can be used only in the “rollback phase” of the protocol. At the same time, these transactions guarantee to the original owners of the coins that if transactions \( tx1 \) and \( tx3 \) remain unclaimed, i.e., the trade is interrupted, they will eventually receive their coins back. The protocol requires each party to have its transaction \( tx2 \) (or \( tx4 \)) signed before broadcasting the transaction \( tx1 \) (or \( tx3 \)).

The second branch performs two checks: equality verification of a given hash with a hashed value of \( x \) that was provided, and verification of a signature of the party that is supposed to receive these coins from the trade. If the first check fails, the script stops immediately. This branch guarantees that only the intended party can spend the coins and, at the same time, knowledge of \( x \) is required to do so.

In the beginning of the protocol, only Alice knows \( x \). But she cannot use it, because no trading transaction, i.e., \( tx1/tx3 \), has been published, and Bob will broadcast his transaction \( tx3 \) only after he sees transaction \( tx1 \) in the blockchain and verifies its correctness. This strict ordering of actions ensures that when Alice claims coins from Bob, Bob can use revealed value of \( x \) to also claim coins from Alice.
4 P2P Overlay

While removing trusted third party from execution of cryptocurrency trades is a great achievement, building the actual trading platform on a traditional client-server model would still leave in place a lot of space for potential disruption of the service. Moreover, a decentralized platform for performing exchanges of different cryptocurrencies better fits into the decentralized nature of the very cryptocurrency ecosystem. Therefore, my approach focuses on decentralizing it as much as possible, while also trying to keep the operation reasonably efficient.

In this chapter I focus only on the infrastructure of the aforementioned platform. Communication protocol itself is covered in the next chapter.

4.1 Requirements and Design Goals

To best serve the purpose and nature of a decentralized cryptocurrency service, the following requirements on the P2P overlay were narrowed down as the most crucial:

- Hiding IP addresses of peers
- Peer-to-peer communication
- Efficient routing mechanism
- NAT traversal ability
- Redundancy

4.1.1 Anonymity of Peers

While providing full anonymity to users is far beyond the scope of this work and might be eventually achieved by connecting to the overlay via Tor network, not advertising users’ IP addresses might still be a useful feature. Not knowing where the other trading party is, makes it much harder to attack it.
As there is money in question, there is an immediate incentive to attack users to cause them harm or even to try to steal their money. Therefore, traditional way of talking to other peers via direct connections, as utilized by other P2P overlays, should not be used here but rather replaced with an indirect communication carried over the P2P network.

### 4.1.2 Communication Between Peers

Every information exchanged in a cryptocurrency network is a part of a public knowledge. Therefore, it is possible to use a broadcast or a gossip protocol to spread a message through the network.

In contrast to this, in Coincer there are two basic types of information being delivered over the network. The first is the same as the one described above: market information—orders, asks, and bids. This data can be spread using similar or even the same methods as in cryptocurrency networks. Beside this, there is also information always exchanged exclusively between two specific peers—the trading parties. Messages related to trades are only relevant to the peers trading with each other, and irrelevant to everybody else.

The overlay must therefore enable any two peers in the network to communicate with each other.

### 4.1.3 Efficient Routing

Just the previous requirement alone could be easily solved by simple broadcast of every message through the whole network, but that would be far from efficient and would limit possible scalability of the overlay. Therefore, if possible, more ingenious routing mechanism should be employed for the purpose of direct communication between peers.

Moreover, not only direct communication should try to be carried out effectively. Even market messages targeted at all peers should not spread across network more than once.
4.1.4 NAT Traversal

The more are ISPs running out of IPv4 addresses and deploying address translating mechanisms (NATs) for more of their customers, the more important it is to design such means that would enable all potential users behind a NAT to join the overlay and use all of its functions.

Users from a network with a restrictive firewall, that rejects unassociated incoming traffic, are in a very similar position and should be therefore treated the same way as users behind NAT to be also able to participate in the overlay without any major restrictions. Just as in any cryptocurrency network nowadays.

4.1.5 Redundancy

One of the natural phenomena in P2P networks is churn. Peers join and leave the network, often even without notice; sometimes because of user’s action, other time due to the underlying network failure, or because of a different reason. These events might break existing routes, causing local disruptions of overlay operation. On the other hand, P2P environment offers inherent means that can be leveraged to make the overlay more resilient to such failures.

The overlay should not contain any single point of failure and should make use of redundancy for quick recovery of its operation in case of any failure.

4.2 Possible Solutions

Requirement on hiding IP addresses of peers suggests that the final P2P network will be probably unstructured, because traditional structured P2P networks, such as Chord, Pastry, etc., make extensive use of the knowledge of peers’ IP addresses [11]. Nonetheless, there still might be proposed a solution that creates a structured P2P network while obeying to this requirement.

It is observable that unstructured P2P networks share many similarities with mobile ad-hoc networks. There are several existing routing protocols for ad-hoc networks, therefore, overview of their capabilities and suitability for deployment within Coincer is provided
later in this section. However, not all of them are included. First, there are three basic types of ad-hoc routing protocols: proactive, reactive, and hybrid. Second, some of their properties render respective sets clearly unsuitable.

4.2.1 Types of Ad-Hoc Routing Protocols

Proactive Protocols

Proactive protocols continuously maintain functional network that is capable of routing messages between any nodes inside of it. Any change in the topology is propagated throughout the network so that every node has up-to-date routing information.

Reactive Protocols

Reactive protocols do not maintain state of the network. Instead, they save bandwidth and power resources. Routes are established when a node wants to communicate with some other node in the network. This, hence, leads to a higher initial latency.

The problem with nodes not knowing who is in the network is in defunct protection against replay attacks (which is described in section 4.3.4). Otherwise, these protocols could be good candidates for usage in Coincer.

Hybrid Protocols

Hybrid protocols try to combine advantages of both proactive and reactive protocols. Not all routes are established beforehand, instead, network is partitioned into zones or hierarchies. Reactive protocols are then used to build routes across various scopes, while proactive protocols are used inside of them to reduce maintenance costs.

Unfortunately, even this approach suffers from the same problem as do reactive protocols—the non-functioning prevention of replay attacks.

Therefore, this section further includes only protocols from the category of proactive protocols.
4.2.2 Optimized Link State Routing Protocol (OLSR)

OLSR protocol introduces MultiPoint Relays (MPRs) for more efficient broadcasts. MPRs regularly exchange topology information, advertise which nodes use them for communication with the rest of the network. Packets are routed via MPRs. Routing itself is hop-by-hop-based. Basic metric for routing is hop count. Nodes are allowed to have multiple addresses (one per interface), yet each has only one main address that identifies it [12].

This protocol is optimized for wireless networks, treats nodes as non-equivalent, and propagate information about neighbours of each MPR. Especially the last point makes it unsuitable for use in Coincer as it does not offer any sort of anonymity to nodes.

4.2.3 Destination-Sequence Distance Vector Protocol (DSDV)

DSDV protocol is inspired by RIP in its basic algorithm and approach for simplicity. It leverages sequence numbers in tracking freshness of routing table updates to achieve loop-free routing. Similarly to the OLSR protocol, DSDV also uses hop count as a metric for routing [13].

Mainly the frequent updates required on every lost or newly established connection to maintain properties of DSDV, and lack of redundancy in available routes are the reasons for not opting for this protocol in Coincer.

4.2.4 Babel

Babel supports asynchronous communication—instead of sending messages right away, it delays and aggregates them to avoid collisions in the physical medium. Nodes send maintenance hello messages every few seconds to signal available routes [14]. Extension of Babel for overlay networks suggests delay routing metric, although, for technical reasons, it implements round-trip time metric instead [15].

This protocol is overly complex for my relatively simple use case, and its usage would pose unnecessarily high overhead on other implementers. As there is only reference implementation of Babel in
C, for other programming languages it would need to be reimplemented, which could be non-trivial in many programming languages given its binary format.

4.2.5 B.A.T.M.A.N.

B.A.T.M.A.N is a protocol highly optimized for wireless mesh networks. It uses a similar basic routing mechanism as the DSDV, but with a time-based metric instead of a hop count-based one. Each node broadcasts a hello message with its address and a sequence number to advertise its existence in the network [16].

This protocol and its algorithms are not well-documented which would make it hard to reimplement. Juliusz Chroboczek, author of the Babel protocol, says [17] that B.A.T.M.A.N. also contains several issues, including no loop avoidance mechanism. Therefore, this protocol as a whole will not be used in Coincer.

4.3 Proposal

Proposed routing protocol shares various similarities with the protocols described above. It is based on distance vector routing approach so that nodes do not need to build image of the whole network, and it also enables not advertising of node addresses and node neighbours, because routes are decided on hop-by-hop basis.

Node that joins the network broadcasts an I-am-here message with maximum TTL, so that every node in the network knows about its presence. Nodes track arrival times of the message from their neighbours and mark as preferred the first neighbour that forwarded the message to them. This approach prefers routes that are possibly the most efficient. Hop count-based approach could make use of routes over high number of physical hops, although they consist of only few hops on the overlay level. Besides, this approach eliminates an attack on TTL, as described in the section 4.3.4.

To impose redundancy into the overlay, every node maintains at least 4 connections to other nodes in the network. This significantly lowers the probability that the network would create unconnected components (although it does not completely eliminate the risk). It
also offers redundancy in the routing itself as there are always more routes to every node.

4.3.1 Loops

Under normal circumstances the protocol does not form loops. However, it is possible that the optimal route breaks (e.g., some node on the path leaves the network) and the rest of routing information in the network could cause messages on this route loop.

Solution used by other protocols, e.g., DSDV, would be to broadcast information about broken route on every node departure, i.e., not utilizing available redundancy. The biggest disadvantage of this solution is generation of a very big traffic on high churn.

Instead, proposed solution is to remember hashes of last $N$ messages that a node forwarded. If a node detects a message it had already forwarded, it deletes its best route to the destination and forwards the message via the second best route. If there is not any available route, the node broadcasts a Where-are-you? message—if the target node is reachable, it will receive this message and will respond with an I-am-here message (also via broadcast). This refreshes routing information among all nodes.

4.3.2 Routing Table Updates

There is no passive global mechanism that could detect unannounced departures of nodes. While nodes can easily find out that their neighbour disconnected, because their connection is closed or timeouts, it is not possible to propagate such information throughout the network, because:

(a) Such information is not trustworthy—any node could send it, regardless of its trueness.

(b) One broken connection does not imply that a node left the network.

Instead, entries in routing tables eventually expire. Nodes are therefore required to periodically rebroadcast their I-am-here message. This must be done ahead of the expiration time, otherwise the node would become unreachable from within the network.
Expiration ensures that nodes do not store stale records and utilize their resources in an efficient manner.

4.3.3 Graceful Departure

While sudden disconnections without any reconnection within consecutive moments are well handled by timeouts, it inevitably causes inefficiencies in the meantime as the network might try to deliver messages to the node. To eliminate this, the node that is going to depart from the network should broadcast a \textit{bye} message. Other nodes then remove it from their routing tables, and neighbours close their connections to this node.

4.3.4 Attacks

Man-in-the-Middle Attack

This is the most obvious attack vector as messages are sent via other nodes and certain message types are even broadcast to everyone. Any node could monitor content of trading messages or even tamper with them. To prevent deliberate modifications or random corruption of messages, communication protocol must include digital signatures. To provide confidentiality, end-to-end encryption of contents should be used. In order to bypass complicated key exchange procedures, nodes could use their public keys as their identifiers. Therefore, any node could address any other node and start sending it encrypted messages right away, and everyone is also able to verify signature of every message.

To save bandwidth and processing power, corrupted messages are not forwarded, but instead immediately silently discarded.

In a case of an active attacker who duplicates market orders of other users, there are two possible outcomes from him being in between of two trading users. First, the attacker could simply forward messages of those two users who think they trade with the attacker (but they do not know he is an attacker). That means that the confidentiality is broken and this one particular attacker knows all the details of the trade. Second, the attacker could tamper with their messages, inject his own keys, etc., but that never leads to a successful
4. P2P Overlay

trade, because the exchange protocol is resilient to the other party’s misbehaviour.

Replay Attack

While modifying messages is prevented by digital signatures, attackers could still keep messages and send them later to the network again. This might create false state of affairs in the network—invalid market orders, or trading messages coming again in later stages of respective trades.

To eliminate potential replay attacks, every message sent within the network needs to carry a nonce. The nonce must be a positive ever-increasing number. This nonce is unique within messages sent by one specific node. Nodes should store the value of their last used nonce so that it persists between client restarts, or it is strongly recommended to generate new identifier. Otherwise, it might occur that if a node uses some nonces again then (a) its messages might be discarded by some nodes (possibly even by all of them), or (b) an attacker might carry out a replay attack against this node.

Trading messages  It is obvious from the diagram 5.2 of the trading protocol that peers do not send each other more than one message at a time, and they wait for response before sending next message. Therefore, it is possible to remember, for instance, within the routing table the last nonce used for direct communication and to discard any message with a lower/used nonce. However, two peers might conduct more than one trade at a time—in such situation it is possible that messages arrive in different order than they were sent. In this case, solution proposed for market messages below is more appropriate.

I-am-here message  This is a very special message, because it predetermines routability within the network and reachability of nodes. Therefore, its nonce is stored separately so that it is always available for examination.

Repetition of this message is ignored from a neighbour that has already sent it. Undetectable replaying from other neighbours (i.e.,
delaying of the I-am-here message) does not influence anything as these neighbours would form the worst available routes.

**Market messages** These messages are broadcast to every node and may arrive out-of-order, because they are independent and do not anticipate any response. It is not beneficial to remember only the last nonce, because that could easily lead to discarding also legitimate messages. This problem does not have just a single solution, so the one proposed here is a set of trade-offs between network delay tolerance and efficient resource utilization.

It is recommended to store nonces of market messages received in the last minute. If a newly arrived message has a nonce lower than the oldest nonce in the list, it is discarded. If it is higher than the oldest nonce, but still lower than the last one, the message is accepted, but its nonce is not stored at the end of the list as normally, but—in order to maintain the ever-increasing property of the nonce list—in the position it should be stored so that the list remains sorted, and not with the real time it arrived, but with the same timestamp as the preceding nonce. Nonces older than a minute are removed from the list, except for the last one—at least one nonce must always be remembered (with the exception of newly connected nodes).

**Sybil Attack**

It is possible to carry out a successful Sybil attack on Coincer’s network. Nevertheless, it is not obvious what could the attacker gain from such an attack, unless it is just a special case of a DoS attack.

In order to steal coins from a victim, the attacker would also need to carry out successful Sybil attack on the cryptocurrency the victim wants to buy. Then the attacker could trick the victim into believing that the attacker sent his transaction \(tx1\) or \(tx3\) (depending on who started the protocol) so that the victim would then send also his transaction \(tx3\) or \(tx5\).

However, described attack is independent of Coincer, i.e., it could be carried out even without attacking the Coincer network itself. Moreover, attacking Coincer does not increase chances for success in the attack described above.
Denial of Service Attack

Denial of Service (DoS) attack can have many forms, but it always has common goal: prevent user(s) from performing trades via Coincerc.

**Flooding**  When a node creates and broadcasts an excessive amount of messages in its name, its neighbours disconnect from it. Therefore, it might use a (potentially pregenerated) set of different IDs. Its neighbours could ask it not to forward them messages from those nodes (IDs) that generate excessive amount of messages. If the node does not fulfill their requests, they disconnect from it. Nodes also add that node to a blacklist so that it cannot reconnect to the network to continue with the attack.

**TTL Spoofing**  A malicious node could modify every message it forwards in such a way that it sets TTL of the message to the maximum value. However, this cannot do any harm. If the message loops, it will be caught after one iteration and silently discarded. Otherwise, the message very soon leaves the network in a normal way regardless of its TTL.
5 Communication Protocol

Difficulty of implementing a communication protocol is one of the factors that predetermine whether the protocol will be implemented in a number of applications or silently ignored. The goal is therefore to design a simple, straightforward, yet easily extensible protocol, possibly with a little overhead.

5.1 Protocol Form

Binary protocols are very efficient in terms of bandwidth usage or parsing overhead. However, the overhead grows as extensibility is being built into the protocol. As signatures should be also included in the messages, every message will have two parts: signed and unsigned. The latter includes, for example, the signature itself, or a TTL. A considerable disadvantage of binary protocols is a difficulty to implement the protocol in various scripting languages that do not contain a rich set of instruments for dealing with binary data. Even in languages like C, implementations are often error-prone.

Text protocols, on the other hand, require more data for their formatting overhead and text representation of binary data, also their parsing is more complex. Still, they could be very easily extensible. E.g., protocols based on XML or JSON formats take advantage of the fact that these formats are well-defined and there are many stable open-source libraries available for many programming languages—this makes implementation of the protocols considerably easier.

Despite advantages of binary protocols, I will propose a text protocol for Coincer, because of ease of implementation, debugging, and extensibility.

5.2 Basic Protocol Format

In the context of cryptocurrencies, a JSON format is often used for encoding in protocols, e.g., in Stratum protocol or Bitcoin Core RPC API. This means that most applications already feature some JSON encoding/decoding library. Choosing the same format for Coincer
means that there would be no dependency overhead for other applications to implement Coincer’s protocol.

JSON is a structured format. However, different order of fields still result in the same data structure. As a result, signing subset of a message needs to happen given strict ordering of fields and formatting of JSON (e.g., spaces), otherwise deserializing a message, updating a TTL and serializing it again on forwarding could render the signature invalid if the message fields were serialized in an order different from the original one. Another possible approach is to nest a JSON message into a JSON message, i.e., forming a data carrying part and its envelope. Its advantage lies in its immutability—during routing process the inner part never gets deserialized, therefore, the signed data is transmitted to recipients without any unintentional changes in its content.

Content of the envelope is as follows (written in a pseudo JSON):

```
{
   "version": integer,
   "from": string(64),
   "ttl": integer(128),
   "data": string,
   "sig": string(128)
}
```

In this notation, keys are real keys, while “values” indicate types of real values. Numbers in parentheses then denote maximum values for integers and exact length for strings. Those types that do not include detailed information about possible size of their content are not restricted by any explicit limit.

The keys have the following objectives:

**version** Denotes version of the protocol. Versioning helps to maintain backward compatibility.

**from** Identifier of the sender. As suggested in previous chapter, public key of the sender (serialized as a hexadecimal string) is used as his identifier.

**ttl** How many more hops can the message pass before it is discarded.
**5. Communication Protocol**

**data** Inner message that is signed by the sender. It is stored as a serialized JSON with escaped quotes.

**sig** Signature of the content of the field *data*. It is serialized as a hexadecimal string.

### 5.3 Inner Message Format and Data

Inside of the envelope *data* field is data that no one can tamper with along the message delivery. Its basic structure, common to all message types, is shown below in the pseudo JSON:

```json
{
    "to": string(64),
    "type": string,
    "nonce": integer(18446744073709551615)
}
```

These fields have the following purposes:

- **to** Identifier of the recipient. Again, it is his public key serialized as a hexadecimal string.

- **type** Type is used as a discriminator of the data carried inside of the message. This section presents all possible message types.

- **nonce** Number used for encryption purposes (more information on this topic is later in the section 5.5.2) and for elimination of replay attacks (details in the section 4.3.4). The maximum value is derived from the limitation of used encryption algorithm that uses 64-bit nonce.

These three fields are included in every message. Moreover, fields *type* and *nonce* always carry a value, while *to* might be empty, i.e., NULL.

### 5.3.1 Types of Messages

There are several major types that cover groups of message subtypes designated to accomplish certain goals. Such grouping simplifies rules for preprocessing of messages as a single rule can be
applied to a whole set of subtypes at once, even to those that do not exist yet.

Syntactically is the type formed by concatenating subtype(s) with dots, e.g., p2p.peers.sol.

There are four main message types: p2p, market, encrypted, and trade. Type encrypted is a special one. It encapsulates direct communication of two particular peers, and it does not have any subtypes, it just serves a function of a service type to provide message encryption. All messages of the type trade are sent solely inside of encrypted message type, i.e., encrypted.

5.3.2 Messages of Type p2p

Messages belonging under the type p2p are service messages needed for the P2P network organization and maintenance.

p2p.hello

**Description** Hello message is sent by every two neighbours to each other when they establish a connection in order to let the other neighbour know about their ID, version of client, and a network port they listen on.

**Format**

```json
{
    "client": string,
    "port": integer(65535)
}
```

**client** Textual representation of the name of the client software and its version.

**port** Network port number on which the client listens. Allowed values: 0–65535.

**Rules** Message of this subtype is first sent by a neighbour that initiates a connection. This neighbour does not know ID of the other neighbour so he sets recipient’s ID to NULL. TTL of the message must
be 1, so that it is under no circumstances forwarded any further to the network.

p2p.bye

**Description**  Announcement of a peer about leaving the network so that all other peers can delete it immediately from their routing tables.

**Format**

```
{}
```

I.e., no specific data is carried inside of this message.

**Rules**  Just before terminating, the peer should send this message. Unlike the hello message, this one is broadcast to everyone (i.e., to is set to NULL) and the TTL is set to the maximum—128.

p2p.iamhere

**Description**  Presence announcing message that spreads through the network and builds information about the best and alternative routes to the originating peer.

**Format**

```
{}
```

I.e., no specific data is carried inside of this message.

**Rules**  After a peer which establishes a connection with other peer in the network receives a hello message, it sends this message over the new connection. It is sent in broadcast-like manner, i.e., the recipient is set to NULL. Besides that this message type is always sent with a maximum TTL, which is 128.
5. Communication Protocol

p2p.whereareyou

**Description**  Serves as a way to proactively discover new routes to peers that are known to be with high probability still present somewhere in the network.

**Format**

```json
{
    "you": string(64)
}
```

**you**  Identifier of the searched peer. It is his public key serialized as a hexadecimal string.

**Rules**  When a node during a routing process runs out of possible routes to the target node, it broadcasts this message. It is also a part of the process of eliminating loops, as described in section 4.3.1. TTL is set to its maximum allowed value. In a case when a node is supposed to forward a message, but does not know any route to the destination, it does not broadcast this message—as a protection against provoked flooding.

p2p.peers.sol

**Description**  To eliminate any centralized source of addresses of nodes in the network, peers can ask each other for addresses of other nodes they know.

**Format**

```json
{}
```

I.e., no specific data is carried inside of this message.

**Rules**  A node sends this message to its neighbours, with TTL set to 1. There is no limitation of when it could be sent. It is also usually used by new nodes to extend their database of node addresses retrieved via DNS bootstrap mechanism.
p2p.peers.adv

**Description**  This message type advertises known node addresses, but does not include any information about connectedness of the node with any of these addresses.

**Format**

```json
{
    "addresses": [
        [string, integer(65535)], ...
    ]
}
```

**addresses**  Unsorted array of tuples of an IP address and a port number. IP address can be either IPv4 or IPv6, both in a common human-readable format.

**Rules**  Node sends this message when it receives a p2p.peers.sol message requesting a list of peers known to the node being asked. TTL is always set to 1.

### 5.3.3 Messages of Type market

Messages belonging under the type market maintain a state of the decentralized market, enable users to create or cancel their orders, and the like.

**market.order**

**Description**  Basic market building message. It says which and how many coins are offered and which and how many demanded.
5. Communication Protocol

Format
{
    "bid": [
        string(64),
        string
    ],
    "ask": [
        string(64),
        string
    ]
}

**bid** Information about offered coins. It is a tuple of a coin identifier and an amount. As a coin identifier Coincer uses hash of the coin’s genesis block, i.e., the block that is the very first in the blockchain. It is serialized as a hexadecimal string. Amount is a string containing a floating point number in standard notation with point as a decimal separator, e.g., “31.41529”. The decimal part of the number is optional.

**ask** Information about desired coins. It is of the same format as the **bid** field.

**Rules** Market orders are broadcast with standard TTL. Validity of orders is 1 hour since their reception. If a peer broadcasts the same order again, it prolongs validity of the order to one hour since reception of this update.

**market.cancel**

**Description** User should be able to cancel his order even before its expiration, e.g., when he starts a trade with an other user.

Format
{
    "id": string(64)
}
id This is a 256-bit SHA3 hash (in hexadecimal notation) of concatenated string data (all possibly binary data come into the input string in hexadecimal notation): ID of the seller, ID of the coin being sold, amount of coins being sold, ID of the coin demanded, and its amount. Amounts share the same rules as apply for amounts in market.order messages. In addition to that, amounts always have at least one decimal place which might be zero, otherwise no trailing zeros are allowed. Number of decimal places is limited by minimum maximum divisibility of both coins.

Rules Just as market orders, order cancellations are broadcast with standard TTL. Peers should cancel their orders before they depart from the network.

market.sol

Description To globally maintain the state of the market, peers ask their neighbours for the market orders on startup.

Format
{
    "pair": [
        string(64),
        string(64)
    ]
}

pair This is a tuple of the identifier of the coin that is being sold on the market and the identifier of the coin that is being demanded in exchange for that coin. Both identifiers are serialized as hexadecimal strings.

Rules These messages are broadcast only to immediate neighbours, therefore carry TTL equaling to 1.
market.adv

**Description**  This type of messages advertises available orders on the market of solicited pair of coins.

**Format**
{
   "pair": [
      string(64),
      string(64)
   ],
   "orders": [
      [string,
       string,
       string(64)
      ]
   ]
}

**pair**  This is the same tuple of the identifier of the coin that is being sold on the market, and the identifier of the coin that is being demanded in exchange for that coin, as is in the original request `market.sol`. Both identifiers are serialized as hexadecimal strings.

**orders**  An array of orders for the given pair of coins. Each order is a triplet of amount of the coin being sold, the coin demanded in exchange for the first coin, and ID of the seller (serialized as a hexadecimal string).

**Rules**  Message of this type is only sent as a response to the message `market.sol` and carries the same pair data. TTL is, again, set to 1.

### 5.3.4 Messages of Type trade

Messages belonging under the type `trade` secure the core functionality of the whole system—exchange of cryptocurrency coins.
trade.proposal

**Description**  The first message that begins a trade. It is sent as an answer to an order.

**Format**

```json
{
    "oid": string(64),
    "hpx": string(64)
}
```

**oid**  Order ID the trade is referring to.

**hpx**  SHA3-hashed 32-byte number that will be later used to determine who starts the actual exchange protocol. It is hashed so that the order cannot be easily manipulated.

Both values are serialized as hexadecimal strings.

**Rules**  This message is sent directly (in the means of a P2P network) to the order originator. Number px, which is sent hashed inside of this message as hpx, is generated randomly in a cryptographically secure way.

trade.accept

**Description**  If the received trade proposal is the first proposal for the given order, it is accepted by sending this message and the trade preparations continue.

**Format**

```json
{
    "oid": string(64),
    "hpx": string(64)
}
```

**oid**  Order ID the trade is referring to.
**5. Communication Protocol**

**hpx** SHA3-hashed 32-byte number that will be later used to determine who starts the actual exchange protocol.

Both values are serialized as hexadecimal strings.

**Rules**  
**trade.accept** message is a positive response to a message of type **trade.proposal** and is sent only in the event of reception of that message. It cannot be sent if the peer already accepted another trade proposal for the same order. Similarly to **trade.proposal**, messages of this type also include hashed number **px**.

**trade.reject**

**Description**  
Unless the received trade proposal is the first proposal for the given order, i.e., if the node has already accepted a proposal for this order, this rejecting message is sent to the trade proposing peer. It is also sent in a case of any error.

**Format**

```json
{
    "oid": string(64),
    "code": integer
}
```

**oid**  
Order ID the trade is referring to. It is serialized as a hexadecimal string.

**code**  
Error code explaining reasons of rejection. Their overview is given in the table 5.1.

**Rules**  
This message signals abortion of the respective trade. Trades should not be terminated by letting them to expire on timeout. Instead, message of this type needs to be sent.

**trade.px**

**Description**  
Peers exchange the numbers they previously generated and sent to each other hashed as **hpx** in order to be able to decide who starts the exchange protocol itself.
5. Communication Protocol

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Incorrect or unknown order ID.</td>
</tr>
<tr>
<td>2</td>
<td>Already trading this order with another peer.</td>
</tr>
<tr>
<td>3</td>
<td>Incorrect format of hpx.</td>
</tr>
<tr>
<td>4</td>
<td>Incorrect format of a key.</td>
</tr>
<tr>
<td>5</td>
<td>Incorrect peer (this one is not trading in the given trade).</td>
</tr>
<tr>
<td>6</td>
<td>Peer does not follow the rules of who should start and generate $x$.</td>
</tr>
<tr>
<td>7</td>
<td>Incorrect transaction, or different than expected content of a transaction.</td>
</tr>
<tr>
<td>8</td>
<td>Invalid signature.</td>
</tr>
</tbody>
</table>

Table 5.1: Overview of error codes

Format

```
{  
  "oid": string(64),  
  "px": string(64)  
}
```

**oid** Order ID the trade is referring to.

**px** Special number used to determine who starts the exchange protocol.

Both values are serialized as hexadecimal strings.

**Rules** As soon as a peer receives `trade.accept` or `trade.px` message, it reveals `px` value in his `trade.px` message.

**trade.key**

**Description** Before the start of the exchange part of the protocol, parties need to exchange their public keys they generate specifically for this trade. In addition, the peer whose `px` hashed together with `oid` is smaller sends to the other peer hash of a number $x$ which is used within the exchange protocol. The value of `px` is not used alone so as no one can cheat by choosing zero or other very small number (or vice versa $2^{256}$ and so on).
5. Communication Protocol

Format
{
   "oid": string(64),
   "key": string(64),
   "hx": string(40)
}

oid  Order ID the trade is referring to.

key  Public key to be used within the exchange protocol for locking coins.

hx   RIPEMD160-hashed SHA256 hash of a number generated in a cryptographically secure way. Important for the security of the trade.

All values are serialized as hexadecimal strings.

Rules  After sending and receiving \texttt{trade.px} message (in this order) or after receiving a message of type \texttt{trade.key}, peer responds with this message. The key is a one-shot key generated solely for the purpose of this trade. It is not even included in any of user’s wallets. The field \texttt{hx} is included only if the peer had smaller 256-bit SHA3 hash of a concatenation of the order ID and his \texttt{px} number (both in hexadecimal form) than the other peer. In this case the peer also starts the exchange protocol right after sending this message.

\texttt{trade.sigreq}

Description  Request on the other party to provide a signature for a transaction. It is used to obtain a co-signature for multisignature-based rollback transaction (see section 3.1.2 for details of the atomic protocol).

Format
{
   "oid": string(64),
   "tx": string
}
5. COMMUNICATION PROTOCOL

oid  Order ID the trade is referring to.

tax  Transaction the peer should provide a signature for.

Both values are serialized as hexadecimal strings.

Rules  Originating peer includes a complete transaction which is missing only signatures. Receiving peer checks that all its parameters are set correctly, as expected.

trade.sig

Description  Response to the signature request—it carries the requested signature. The transaction it belongs to is not included as the other peer knows it.

Format

{  
  "oid": string(64),
  "sig": string
}

oid  Order ID the trade is referring to.

sig  Signature for the transaction included in the preceding signature request.

Both values are serialized as hexadecimal strings.

Rules  Peer who receives this message should complete the transaction and verify that the received signature is valid. If it is invalid, he terminates the trade.

5.4 Trading Protocol

Objective of this section is to show that the flow of the communication protocol within a trading process is correct and always leads to a state in which each party has the original coins or newly exchanged
5. COMMUNICATION PROTOCOL

coins. To model the protocol I used MSC\(^1\) formalism. For the purpose of verification of the model I employed scstudio and its plugin for Microsoft Visio 2007 [18].

Scstudio is able to automatically detect a deadlock and a livelock, cycles, race conditions, and to verify universal boundedness. Successfully verified model of the protocol is therefore guaranteed (not) to possess these properties. Moreover, scstudio has the ability to match the model to real communication so that an implementation can be verified whether it conforms to the protocol.

All presented models were not only modeled in scstudio, but also successfully verified by it.

The communication model as a whole is depicted on a diagram 5.1. It shows that the communication breaks down into several patterns. One of them is a full, successful run, while all the others are incomplete, including situations in which one of the trading parties stopped communicating. Their listing is not complete, but instead, all trivial cases before anything is published via blockchains are “included” in the first error diagram—when such a situation happens, no action is needed, the trade is simply terminated.

---

1. Message Sequence Chart—a formal descriptive technique for description of the communication behaviour of system components and their environment by means of message interchange [19].
During modeling of communication I found a bug\textsuperscript{2} in scstudio, because of which verification of my models failed on putative race condition. Scstudio turned out to be unable to cope with action boxes. Therefore, I replaced them with comment boxes which technically accomplished the same function as I did not use the boxes for any formal records, but rather for a free-text description of what each party is doing in certain points.

5.4.1 Model of a Successful Protocol Run

This is the case where both trading parties cooperate and want to finish the trade. No message timeouts, no message is lost, everything is delivered in a timely fashion.

Besides simple message passing and comments, the model also utilizes several more MSC features. One of them is time constraint. It indicates how long in seconds a party is willing to wait for the message before she thinks the other party stopped communicating and aborted the trade. Second one is a coregion box visible on Alice’s line. It says that it is not important in which order the messages arrive, i.e., if Alice receives the latter one first, it is still a valid instance of the protocol.

5.4.2 Model of Erroneous Situation #1: Interruption Before Going into Blockchain

In this case, as shown on a diagram 5.3, the transaction $tx1$ is not stored in its blockchain within the specified time limit. Therefore, it is expected to have never been even broadcast (if it is not this case, the following situation applies). Alice does not need to do anything, she just treats the trade as terminated.

The same “action” is performed also when any other time constraint in the diagram 5.3 is not met. The protocol then does not continue.

\footnote{\url{http://sourceforge.net/p/scstudio/bugs/100/}}
5. Communication Protocol

Figure 5.2: Model of a successful protocol run.
Figure 5.3: Error case #1: Trade is interrupted before sending the first transaction into blockchain.
5.4.3 Model of Erroneous Situation #2: Missing Second Commitment

A diagram 5.4 shows a situation when the transaction \( tx1 \) is successfully stored in the blockchain, but Alice does not broadcast her committing transaction \( tx3 \) within the given time constraint. If this situation occurs, Bob waits for the time lock of his rollback transaction \( tx2 \) to expire and consequently broadcasts it.

5.4.4 Model of Erroneous Situation #3: No Coins Are Claimed

Last possible case, visualized on a diagram 5.5, describes a situation in which Bob refuses to finish the trade by not claiming coins from Alice. Both Bob and Alice have to wait till time locks of their rollback transactions \( tx4 \) and \( tx2 \) expire so that they can claim their coins back by broadcasting these transactions.

5.5 Cryptography

Usage of cryptography in such a decentralized system like Coincer is a very important measure against a wide range of attacks and other malicious activities. In a decentralized environment there is no easy way to tell honest participant with certainty from an attacker, there is no trust, anyone can misbehave.

Coincer includes in its general communication protocol two basic cryptographic elements: digital signatures and end-to-end encryption.

5.5.1 Signing

Every message sent over the network is signed by its sender. For signing is used private key associated with the public key that is used as a user identifier. This eliminates the need for key exchange or distribution between peers, because any peer can verify signature of any message right away. Therefore, each peer is also required to perform signature verification of each received message. If the verification fails, message is silently discarded. Only messages with valid signatures are allowed to be forwarded through the network.
Figure 5.4: Error case #2: tx3 does not reach the blockchain.
Figure 5.5: Error case #3: Bob does not claim his coins.
As an algorithm is used Ed25519. It was selected based on its well-founded choice of elliptic curve and constants, availability of high-grade implementations, small size of both keys and signatures, while offering reasonable security [20].

5.5.2 Encryption

Implicit availability of public keys can be leveraged into encryption of messages without any additional key exchange. In current proposal of the protocol, all trading messages have to be encrypted to keep their content confidential. If there are in the future any other sensitive messages, they should be encrypted too.

Stream cipher ChaCha20 [21] is used for the encryption. Its implementation is often distributed in libraries together with Ed25519, which limits additional dependencies on external libraries. Shared secret key is established using the ECDH algorithm—it is derived from each party’s private key and the other party’s public key. This way both parties compute the same key [22].

Nonces of this cipher are not generated randomly. Therefore, it is important to ensure that the parties never repeat a nonce for the same key. Thus, the peer with bigger ID must always generate odd nonces for the trade, while the other peer even nonces. This simple provision eliminates a need for synchronization or an initial setup.

5.6 Protocol Extensibility

In order to ensure the network can develop, add new features, fix bugs or turn deprecated functionality down Coincer takes advantage of the flexible JSON format and of the version field inside of the protocol envelope. JSON enables clients to be always able to decode the messages and obtain needed information. Any additional fields are overlooked and ignored, which widens backwards compatibility of the protocol.

Version of the protocol is important for deciding which rules to apply on communication with a given peer, which fields to expect, which to ignore, and so forth. Newer client might fallback to older version of the protocol to be compatible with the peer. Otherwise,
5. COMMUNICATION PROTOCOL

older client might decide that it misses some information and disconnect.

Adding of new message types or subtypes is never a conflicting change. Therefore, a possibility for consideration every time a change is needed to be made is not to alter rules for existing (sub)types of messages, but to add their new versions as new (sub)types instead.
6 Analysis

For the purpose of an analysis of a real-world behaviour of described algorithms and protocols, I have implemented Coincer and published it as an open-source\(^1\). With specially crafted patch that logged many different kinds of data and events going on during operation of each Coincer client, I was able to gather information from volunteers who joined a special testing session.

Unfortunately, only 5 users participated in the testing session, despite promotion of this event inside bitcoin community. Therefore, amount of gathered data is far from significant. To compensate this, I decided to set up an artificial testing environment. Although I could not effectively test trading functionality, timing and delays, or miscellaneous other situations in the network, I was still able to perform several tests and measurements.

6.1 Testing Environment

In my settings there were 50 nodes, each connecting to the network with 4 connections (except for the first 4 nodes which connected only to all previous nodes). Resulting distribution of degrees, as shown in chart 6.1, indicates that it might follow power-law distribution, although this is too small testing group to make such claim. On the other hand, in an environment where most peers are connecting from behind a NAT, i.e., inaccessible from the outside, it is highly probably that a “rich-club” formed by servers with high uptime will emerge, while the rest of the network will have only 4 connections in total. Although there might be accessible peers, if they only stay for a time necessary to perform a trade, their degree will still remain low.

I set two more questions to answer in the testing network. First, randomly choosing two nodes, is it possible to send messages between them? What routes will be used for each direction—same or different? Second, will the routing be affected if some connections between nodes are closed?

\(^1\) Implementation of Coincer is available on its homepage \texttt{http://www.coincer.org/} under a GNU GPLv3 license.
Figure 6.1: Distribution of degrees in a synthetic network of 50 nodes.
To find out answers to the given questions, I implemented two more message subtypes into the protocol: `p2p.ping` and `p2p.pong` (request and response, analogous to ICMP Echo messages). In addition to this adaptation, nodes were also able to receive a command to close a certain connection.

My experiment included three testing scenarios:

(a) Randomly selecting two nodes in the network that are not direct neighbour and consequently sending `ping` from the node that joined the network earlier (i.e., also knows route to the other node as it received the `I-am-here` message).

(b) Randomly selecting two nodes in the network that are not direct neighbour, but this time first publishing an order by the older node and then sending a `ping` message by the other node.

(c) Picking a path that is used to transport messages from a node A to a node B and randomly closing one of its connections.

When nodes gradually connect to the network, they do not know anything about nodes that already are in the network. For the node to be able to send a message to a peer who contacted him (or published an order), it is necessary to remember which node forwarded the message to him. Therefore, it could be expected that both messages (`ping` and `pong`) will travel along the same path as it is always the fastest one. However, the experiment showed that this is not guaranteed. While in the scenario (a) this assumption holds, in scenario (b) it might not be true, depending on precious structure of the network between the two nodes.

Scenario (c) showed moderate resilience to random failures. Every time a connection was closed, respective node used a different one to forward received message. However, when the node was left only with neighbours that had joined the network later than the target node, i.e., all its relevant connections were closed, it was unable to proceed and dropped the message.

Described imperfection revealed by the experiment is albeit small, but still important. It might cause temporary inoperativeness of some aspects of the network. Before deployment of the system into real environment, this issue needs to be fixed or at least its impact should be minimized.
7 Conclusion

This thesis focuses on cryptocurrencies and their possibilities for a safe exchange without any intermediary. It also discusses current state of routing protocols for ad-hoc networks.

Existing protocol for cross-chain exchange is studied and it is shown that it is indeed atomic. For this protocol to be actually usable by users, a P2P overlay is designed with inspiration from existing protocols. Properties of the overlay are designed with security in mind. Hence, different kinds of attacks are examined, and their possible impact on the overlay is evaluated. This overlay uses a custom JSON-based protocol whose trading part has been formally modelled and verified.

Final analysis of this P2P platform showed that there are still imperfections in the overlay that need to be fixed. More experiments and wider participation of the community in testing and sharing feedback is desirable. Latest development of cryptocurrencies introduced an operation \texttt{OP\_CHECKLOCKTIMEVERIFY} that could simplify the atomic exchange protocol by taking out the need for preparation of rollback transactions. For future development, this new feature should be certainly taken into account.
Bibliography


Attachments

The following electronic attachment is an integral part of this thesis:

coincer.vsd A set of MSC models of Coincer’s communication protocol.