Decentral Market

Self-regulating electronic market

M.J.G. Olsthoorn
J. Winter

Technische Universiteit Delft - Bachelor thesis

TU Delft

Decentral Market

Self-regulating electronic market

by

M.J.G. Olsthoorn
J. Winter

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Friday June 24, 2016 at 10:00 AM.

Students:	M.J.G. Olsthoorn	
	J. Winter	
Coach:	Ir. E. Bouman	TU Delft
Client:	Dr. Ir. J.A. Pouwelse	TU Delft
Bachelor Coordinator:	Prof. Dr. M.A. Larson	TU Delft

Preface

This thesis is the result of researching and designing a solution to the problem of the self-regulating decentral electronic market, by M.J.G. Olsthoorn and J. Winter. The project took place at the Delft University of Technology for 14 weeks, from early March until the end of June, 2016. First, we would like to thank Dr. Ir. J.A. Pouwelse, our client, for his support and for giving us the opportunity to carry out this project. We would also like to thank Ir. E. Bouman, our coach, for giving us valuable feedback on our project, answering our questions, and supporting us. We further extend our thanks to the Autonomous self-replicating code bachelor project group for the pleasant cooperation and project discussions, which made it possible to elevate the quality of both our projects. We also would like to thank the Tribler Team for their feedback on our problem approach, helping solve our project problems, answering all of our questions, and giving us valuable guidance. Further thanks to the Software Improvement Group for the code quality and maintainability review. Lastly, we would like to thank Dr. Ir. F.F.J. Hermans, Dr. Ir. O.W. Visser and Prof. Dr. M.A. Larson for supervising the TI3806 bachelor end-project course.

M.J.G. Olsthoorn
J. Winter
Delft, June 2016

Abstract

A decentral electronic marketplace to trade digital currencies was created for this project. Users may trade MultiChain balance for Bitcoin. MultiChain can be described as up- and download currency in a peer-to-peer network. When a peer uploads, the balance of that peer increases and the peer can download more effectively. The client for this project is the Tribler Team. Most of the Tribler Team's work revolves around Tribler: an open-source peer-to-peer program which enables its users to find, enjoy, and share content. An earlier bachelor end-project implemented a partial, functional prototype of this idea. We improved the existing concepts and re-implemented a decentralised market from scratch with scalability and as much security as possible, given the limited available development time. This market is the first electronic marketplace to be fully decentralized. There were previous attempts to create such a marketplace, but those implementations fell short in scalability and security. During the research phase, we discovered that the previous decentral market project was not scalable and not production ready, so we made the decision to, together with our client, instead build our application upon a more production-level network platform known as Dispersy. As nothing on the Internet can be trusted which is a problem on its own, we could not make the product as secure as we would have wanted to. At a technical level our contribution consists of 7000+ lines of code, 50+ test suites with a code coverage of +95% and documentation for Dispersy.

Contents

1 Problem description

This chapter defines the problem, analyses it, and lists the project requirements.

1.1. Problem definition

During our bachelor project we were tasked with creating an operational electronic marketplace, immune to shutdown by governments, lawyer-based attacks, or other real-world threats. The internet-deployed marketplace will have no central server bottleneck, no central point of trust, full self-organisation, and unbounded scalability. To keep this project realistic, the focus will be on re-using existing code, to support only Bitcoin-Multichain trading, only use a single market maker, and offer no anonymity or DDoS protection. Bitcoin, Mt Gox, and Silk Road [18] showed how vulnerable marketplaces are. Bitcoin [11] was the first-ever successful digital currency after decades of failed attempts by scientists, anarchists, fraudsters, and entrepreneurs. However, this first-generation technology suffers from a highly unsTable exchange rate, low usage level, and frequent large-scale thefts by hacking exchange platforms. TU Delft has a deposit of Bitcoins which will be used to buy Multichain coins during the final presentation on our platform. Multichain coins are developed at TU Delft. They provide a reward for relaying Tor-like onion [7] routing traffic and BitTorrent seeding. We will use Python, Twisted, Nosetest, Jenkins, and Libtribler during our work.

When we began the project, the problem description differed. The intention originally was to improve and work on the previous bachelor project called “Tsukiji”. During the research phase, we discovered that Tsukiji was not scalable nor at a production level, and that it may not be possible to achieve those aspects. A decision was made, together with the client, to make the project more challenging by creating a real-world, production-level electronic market build on top of the message distribution and peer discovery library called “Dispersy”, created by the Tribler Team. As we increased the workload of the project, we decided to begin a month earlier than the regular planning, giving us a total of 14 weeks to completion. The original project description, as provided by the client, can be found in appendix D.

1.2. Client description

The client for this project is the Tribler Team, which is a part of the Distributed Systems Group at the faculty of Electrical Engineering, Mathematics and Computer Science at the Delft University of Technology under supervision of Dr. Ir. J.A. Pouwelse. The Tribler team has been creating disruptive cooperative software since 1999. Most of the work of the Tribler Team revolves around Tribler: an open-source peer-to-peer program which enables its users to find, enjoy and share content [13].

The main building block used to develop Tribler is Dispersy. This is a distributed permission system, which is a platform to simplify the design of distributed communities [24]. Communities are overlay applications which can be built upon Dispersy. Dispersy will take care of most of the networking logic. A market community could be developed upon Dispersy by using it as a framework.

The MultiChain functionality has also been developed by the Tribler Team and is included in the code base of the Tribler repository. MultiChain can be described as up- and and download currency in a reputation-based peer-to-peer network. When a peer uploads more than it downloads, the reputation of that peer increases, and the peer can download more effectively. The MultiChain community inside Tribler enables the up- and download balance to be stored and transferred to other peers when downloading from them [12]. The MultiChain community has also been developed upon Dispersy.

1.3. Group cooperation

Another bachelor end-project issued by the Tribler Team, is the Autonomous self-replicating code project [5]. The Autonomous self-replicating code group has been tasked with creating an internet-deployed system which can earn money by selling MultiChain balance, and replicate itself by buying new virtual private servers using the earned Bitcoin, without human control. The Autonomous self-replicating code project will have this market community as a dependency and will sell MultiChain quantity on our market community in exchange for Bitcoin. We must agree how the individual systems are going to communicate with each other.

1.4. Project requirements

To determine the requirements of the clients and the priorities of these requirements during the development/implementation phase of the project, we use the MoSCoW method [21]. The requirements are subject to change during the development phase.

Must haves

The most important requirements for the project are described in the “must haves” subsection of the MoSCoW method. The “must have” requirements are the minimal, usable subset of requirement which are guaranteed to be delivered for the project. The following described requirements have the highest priority, and must be fulfilled in the final implementation of the solution:

- Users must be be able to place a bid and ask, which must be distributed across the connected part of the network.
- The market community solution must be scalable with a lot of users.
- The market community solution must be able to match a bid and an ask between peers that are connected.
- The market community solution must be implemented in Tribler [13].
- The market community solution must be a decentralised system.
- The market community solution must be constructed on Dispersy [24].
- The market community solution must be able to make Bitcoin transactions and MultiChain transactions.

Should haves

If all “must have” requirements have been fulfilled, the “should have” requirements have the next-highest priority. The “should have” requirements are important for the project but are not vital to its completion; the solution would still be viable without fulfilling these requirements. All of the following requirements should be fulfilled in the final implementation of the solution:

- The market community solution should be able to let users use incremental payments.
- The market community solution should be tested to a certain extent.

Could haves

The “could have” requirements are the requirements with the lowest priority. If all “should have” requirements have been fulfilled, the “could have” requirements have the next-highest priority. The “could have” requirement are desirable requirements for the project, but they will have less impact when left out compared to “should haves”. The following requirements could be fulfilled in the final implementation:

- The market community solution could be able to blacklist peers on an individual blacklist.
- The market community solution could be able to maintain an efficient connectivity between peers so that a maximum of possible trades are proposed without creating too much overhead on the network.
- The market community solution could fully function.

Won’t haves

The “won’t have” requirements are requirements that won’t be implemented in the final implementation. We have agreed that these requirements will not be delivered. The following requirements will not be fulfilled in the final implementation:

- The market community solution in Tribler won’t have a nice graphical user interface.
- The market community solution won’t have to protect the privacy of its users.
- The market community solution won’t have to be secured against spoofing.

1.5. Development process

During the entire project we worked in sprints of two weeks. We held meetings with our coach and client every week to discuss the preliminary progress and incorporate any feedback that they provided. This made sure that we were able to adjust our goals more towards the client needs. We also had daily meetings within our groups to stay organized and work efficiently. During development we used test-driven development methodology to have a well-tested product to suit or production-level code. At one stage, we ran into the unexpected challenge of requiring to integrate with another bachelor end-project group who were building an application on top of ours, simultaneously. We approached this through close cooperation and by documenting all decisions made.

We developed our product using GitHub for revision control. The project code is available at `https://github.com/mitchellolsthoorn/tribler`. In the development of our product, we used Jenkins for continuous integration. Parallel to the development of our solution, we used test-driven development to extend our test suite and to preserve the quality and maintainability of our implementation [6].

At one stage of the project, we had to submit our solution for a code quality and maintainability review by the Software Improvement Group (SIG). We focused on fixing the issues indicated, to improve the code quality and maintainability of our implementation. At the end of the project we had to submit our product code again to determine the difference in code quality, and whether we incorporated the feedback in our implementation.

2 Global design and existing technology

This initial sprint explores the global design and aimed to understand the existing Dispersy and Tribler technology. An initial high-level market protocol is introduced.

2.1. Sprint description

To make a good overview of the technology and libraries involved in our complex, decentralised electronic market, we started with conducting research in how decentral systems work, and documenting existing libraries that will be used in this project, to understand how we can incorporate them. We will continue doing research into possible schemes for our initial network protocol.

- Research decentral networking theory and applications
- Research Dispersy
- Research Tribler
- Documenting Dispersy
- Research possible market community protocols

2.2. Research information

During the first sprint of the project, we focused on developing a general understanding of the modules that we needed to use during our project. One of the main building blocks of our project is Dispersy [2, 15, 24]. We were tasked with documenting the concepts, installation, and the system overview. The documentation can be found at http://dispersy.readthedocs.io/ [2]. We also researched other peer-to-peer systems to understand how those systems work and interact [8, 10, 14, 23].

The market community that we are going to develop is going to be used in Tribler so we also need to develop the integration between the two libraries. For this reason, we orientated ourselves within Tribler [4, 13] and researched the previous decentral market project [17], existing marketplaces, and possible market community protocols. We found that other marketplaces are either centralized or decentralized, but dependant on Bitcoin, which makes the market unscalable. We also researched decentral networking applications [22, 25], since the market community that we are going to develop is going to be a decentralised system.

2.3. Related work

There already have been attempts to solve subsets of the problems which this project consists of, but most of them use methods which are not scalable or are not viable for production use. Another aspect of the project is MultiChain. Being a recent development of the Tribler team, there has never been an attempt to make a marketplace for MultiChain trading.

2.3.1. Tsukiji

Tsukiji is a completely decentralised marketplace to trade commodities for real money. Tsukiji was made as a bachelor end-project assigned by the Tribler Team. Tsukiji is a lightweight, terminal-based, Python application. Users can trade on Tsukiji using PayPal to transfer real money. Tsukiji is not built on top of Dispersy and Tribler. Tsukiji is a proof of concept for a decentral market, which proves that a decentral market is at least feasible.

2.3.2. BitMarkets

BitMarkets is an open-source protocol and free client for a decentralized marketplace [1] which makes use of Bitcoin as its currency and Bitmessage as its communication network [20]. BitMarkets implemented a third-party escrow agent system, which adds a certain level of security and reliability to the application. BitMarkets proves that a scalable, decentral market can be implemented, which allows users to trade with a digital currency such as Bitcoin.

2.3.3. Beaver

Beaver is a concept for a decentralized anonymous marketplace with a secure reputation system [16]. The concept of Beaver is resistant against malicious attacks, while preserving the anonymity of its customers. It allows its participants to enjoy free open enrollment, and provides every user with the same global view of the reputation of other users. Beaver is just a theoretical concept and has not been implemented.

2.4. Dispersy

Dispersy, the Distributed Permission System, is a platform to simplify the design of distributed communities [2, 15, 24]. A community is the way Dispersy makes an overlay (An overlay is an additional network built on top of an existing one, most commonly the internet). At the heart of Dispersy lies a simple identity and message handling system where each community and each user is uniquely identified. Dispersy is fully decentralized. It does not require any server infrastructure and can run on systems consisting of a large number of nodes. Each node runs the same algorithm and performs the same tasks. All nodes are equally important, resulting in increased robustness.

Dispersy implements NAT puncturing which makes it possible for peers to connect to each other without being exposed directly to the internet. This is accomplished by using a three-way puncturing scheme that uses a familiar third node as the introducing party.

Each node in the overlay holds a list where it keeps track of all the node that it is directly connected with (without going through another node first). These nodes are called neighbours in Dispersy. Initially this list is empty and is filled with the peer discovery algorithm. Which connects to a bootstrap server to find other nodes on the network.

2.5. Market Community Protocol

The market community protocol that we are going to develop during the project will be built on top of Dispersy, and the peer will be introduced to other peers in the network initially, if the peer is not connected. The Dispersy peer procedure protocol is responsible for this. The market community protocol consists of three parts:

1. Ticks which are used to distribute asks and bids across the peer-to-peer network. A bid is placed by users who are interested in buying MultiChain coin with Bitcoin. An ask is the exact opposite, users who are interested in selling MultiChain coin for Bitcoin.

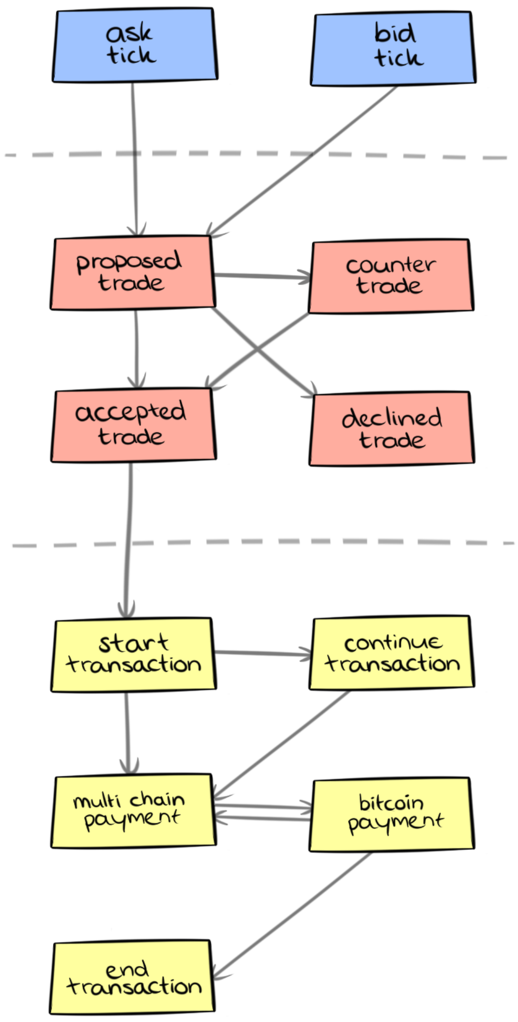
2. Trades which are used to find a peer to trade with and come to an agreement. After a matching tick has been found by a peer, it will send a trade proposal to the origin of the tick. If the tick is

still available, it can either reply with a trade accept or a counter offer. If the tick isn't available anymore, a trade decline will be replied.

3. Transactions which are used to regulate the incremental payment and the Bitcoin and MultiChain payments.

Each part of our market community protocol is executed in a sequential order as can be seen in Figure 2.1. Every node indicates a message type and the arrows indicate the possible replies for each message. Ticks are indicated by blue nodes, trades are indicated by red nodes and transactions are indicated by yellow nodes. Only trade messages can be a reply to tick messages and only transaction messages can be a reply to trade messages.

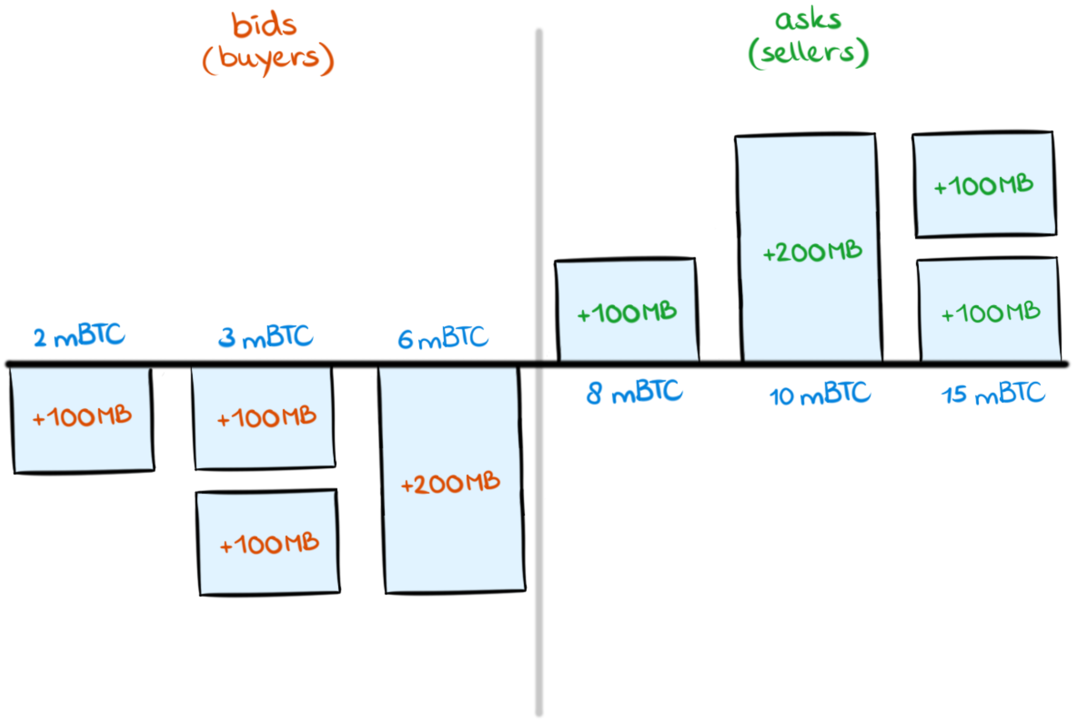
Figure 2.1: State diagram of messages and possible replies between peers.



2.6. Order book

Order books are used to store data about supply and demand, a list of offers divided into two sides, a buyer side and a seller side [9, 19]. An order book is used to record the interest of MultiChain buyers and MultiChain sellers in our project. An order book is displayed in Figure 2.2. The order book in the Figure and in general consists of two sides: the left side contains the bids, and the right side contains the asks. Each side consists of price levels which contains ticks with a certain quantity. If an ask and a bid are in the same price level, they can be removed from the order book and the matching engine can initiate a trade between the ask and the bid. The quantity in the Figure is indicated in megabytes (MB). The quantity used by the MultiChain community is indicated in bytes. The price is indicated in milliBitcoins (mBTC). The market community measures price in Bitcoin (BTC). The order book in our implementation will consist of a side with asks and a side with bids including the quantity and price of the tick.

Figure 2.2: Visualisation of an order book.



3 Order distribution

After our two-week exploratory sprint, we created a market community prototype built on top of Dispersy which was able to distribute ticks across the peer-to-peer network.

3.1. Sprint description

Now that we have a general understanding of the modules that we need to use for our market community, we can further construct the market protocol. Using this market community protocol, we started developing the first iteration of the protocol. In the process of creating the protocol, we received feedback from our coach and client, so we could check and adjust our market community protocol to the client's needs. For example, we discussed the trade-off between distributing a message across the whole peer-to-peer network, and the capacity of the network. We came to the conclusion that we should not flood the network and send messages only to close peers using a "time to live".

- Construct the market community protocol
- Documenting the market community protocol
- Developing an initial market community
- Extend and maintain the market community test suite
- Use a time to live with a range of 2 or 3 for the market community

3.2. Message identification & orders

In a decentral system there is no central party handing out identifiers so all object can be uniquely identified. This needs to be performed by the peers themselves. This is accomplished by making a composite key from the public identifier of the node and a local counter that keep track of the number of objects created. The composite key makes sure that all the object that are used throughout the application can be uniquely identified across the whole overlay network. There are separate identifiers for the orders and the messages (objects that are send over the network). This is done so that on a node it can be individually checked if a message has arrived twice or not at all.

When the user creates a bid or an ask, the system creates an order to represent that internally. In this order all the information regarding that bid or ask is collected and stored. After the order is created a tick is made from this order and distributed across the network to nearby neighbours. This tick represents the original order on another node. On arrival of the tick on a node, it is stored in the order book, so it can be matched against by that node. When eventually a trade is proposed and accepted, the information regarding the progress and the contents are added to the order of both parties involving in the trade.

3.3. Market community tick protocol

Ticks are messages which are distributed across the peer-to-peer network to indicate supply and demand of MultiChain quantity. As seen in Figure 2.1, there are two types of tick messages:

- Bids are placed by peers who are interested in buying MultiChain coin with Bitcoin.
- Asks are placed by peers who are interested in selling MultiChain coin for Bitcoin.

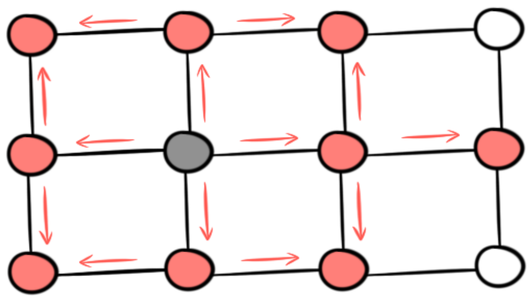
In Dispersy there are two ways to spread messages:

- Messages can be synchronized across clients, which takes a long time. Synchronising means exchanging databases, so storing the items permanently.
- Messages can be sent directly to neighbours, which is more efficient, but can quickly flood the network

Because ticks are created and deleted at a very high rate it does not make sense to synchronize them because it takes a long time and they will probably already be deleted before they are synchronized anyway. It is faster to directly spread the tick message among neighbours. A disadvantage of directly spreading the tick message among neighbours is the higher network traffic and when the order is fulfilled or cancelled, the messages are deleted. Messages can be directly sent to all neighbours or to a single neighbour. Ticks are sent to all neighbours.

In order to not flood the peer-to-peer network, we incorporated a limited broadcast which adds a "time to live" for our tick messages and thus only sends the messages to close peers, with the risk of missing a potential trade. When a peer initiates a bid or an ask, it sends the message to its peers and they also send it to their peers. However, if none of those peers contain a matching ask or bid and there exists a peer in the network who does have a matching ask or bid, the potential trade cannot be made. For example, in Figure 3.1, the peer in grey could have an ask or bid which can only be matched by an ask or bid from one of the peers in white, but because they cannot reach each other, the potential trade cannot be made. Circles indicate peers and lines between those peers indicate connections. The grey peer indicates the origin of the message, and the red lines indicate where the message is being sent from and to. Red circles indicate peers who received the message.

Figure 3.1: Indication how messages are spread across the peer-to-peer network.



There are multiple ways to distribute the tick messages across the network. A possibility would be to only spread one kind of tick: an ask-only tick distribution or a bid-only tick distribution. The other possibility is both bid and ask tick distribution, which enables a price discovery mechanism. This can contain useful information for peers in the network. The price discovery mechanism makes it possible for a peer to construct an order book containing both the asks and the bids in his or her environment. The order book in our implementation contains a side of asks and a side of bids, including the buyer or seller, and the quantity of the tick. This enables the market to be transparent [9, 19], which is important because it enables the user to place bids and asks based on contextual information. In the case of a decentral market with a limited broadcast, it is impossible to achieve complete transparency, as not all ticks are received by each peer.

4 Making trades

This advanced prototype implements a full trading protocol. Our prior prototype only supported a subset of the required functionality, focusing mostly on market order message distribution.

4.1. Sprint description

In this sprint, we continue with the development of the market community and the construction of the market community protocol. Most of the development consists of refactoring the current code base, fixing bugs, and extending our test suite. The client wanted us to test our market community on sample data, so we prepared a test file containing ask and bid messages to be used on our market community, in a simulation to see how the market community reacts to large data sets of messages. In the future, we may extend this simulation so we can analyse the efficiency of the market community, and collect valuable information which can help improve the market community protocol.

- Continue development on the market community and the market community protocol
- Document market community implementation
- Extend and maintain the market community test suite
- Make preparations to test the market community with sample data
- Fix bugs in the market community implementation

4.2. Matching engine

When asks and bids are distributed across the network, each peer can start with matching these ticks and propose trades with their peers. To match ticks, the system needs a matching engine and at least a level 2 order book with the ticks from this peer. When a peer receives a tick from another peer, the matching engine is called with the current order book and the received tick as input of this peer. If the received tick is an ask, the ask is matched with the bid side of the order book. If the received tick is a bid, the bid is matched with the ask side of the order book.

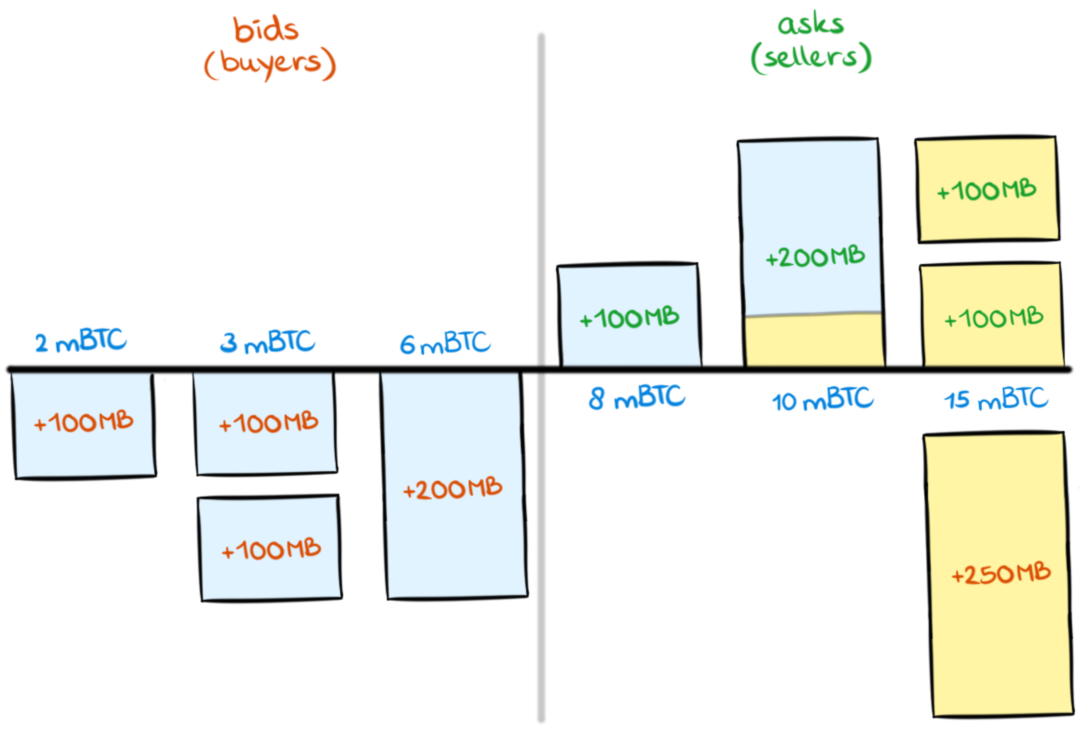
When ticks are matched, their quantity is totally or partially reserved based on the matched tick. When quantity is reserved, it cannot be used by the peer in proposed and counter trades. The quantity in ticks can be reserved because the possibility exists that a trade decline or trade counter message will be replied. When a proposed trade is declined or countered, the reserved quantity of the tick must totally or partially released, based on the matched tick. When quantity is released, it can be reused again by the peer in proposed and counter trades. The reserving of quantity in a tick makes it possible to dynamically match multiple quantity parts of a tick with multiple other ticks.

If the received tick is a bid, the highest price level under or equal to the price level of the received bid of the ask side is selected. If there is an ask with an equal or higher unreserved MultiChain quantity, the selected quantity of the ask is reserved, and a trade proposal is sent to the origin of the received tick. If there is an ask with a lower, unreserved MultiChain quantity, the total quantity of the ask is reserved, and a trade proposal is sent to the origin of the received tick. The procedure is repeated with the unmatched MultiChain quantity from the received bid, until the whole MultiChain quantity from the received bid is matched, or when no more matches can be made. The opposite rules hold for the matching engine if the received tick is an ask. An example matching, with the described matching strategy, can be seen in Figure 4.1. The 250 MB bid in the bottom right corner is matched to the ticks from this peer. The two 100 MB asks with the same price level can be matched, and 50 MB of the bid is still unreserved, which can be matched with a subset of the quantity in the 200 MB ask with a lower price level.

The matching engine can be described at a high-level in the following systematical way:

1. Retrieve the price level with the highest bid price
2. Go through all the bids in the price level
   (a) Reserve as much quantity of the ask on the current bid
   (b) If the quantity still left to be traded is zero then return, otherwise continue
3. If the quantity still left to be traded is zero then return, otherwise continue with 2. with a lower price level

Figure 4.1: Example matching by the matching engine visualised on an order book.

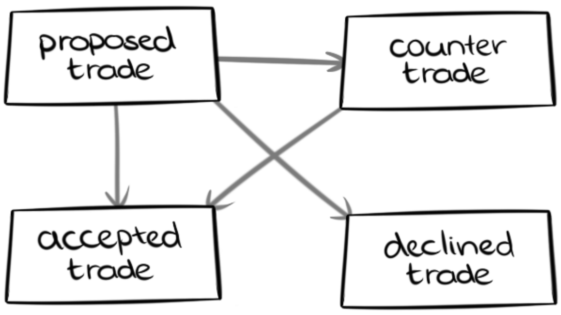


4.3. Market community trade protocol

When the tick messages have been distributed across the network, peers can start trading with each other. As can be seen in Figure 4.2, there are four different types of trade messages. The arrows indicate the possible replies for each message:

- Proposed trades are sent as a reply to a tick. It consists of a trade proposal including the MultiChain quantity to trade, and the Bitcoin price. When a trade proposal is sent, the selected quantity of the respective tick will be reserved until the trade proposal times out, if there is no reply received.
- Counter trades are similar to proposed trades. A counter trade is sent as a reply to a proposed trade when a subset of the MultiChain quantity in the respective tick has already been reserved. A counter trade can always be accepted, because the quantity of the counter trade is always smaller than the reserved quantity for the proposed trade.
- Declined trades are sent as a reply to a proposed trade when the respective tick has already been reserved for its total MultiChain quantity.
- Accepted trades are sent as a reply to a proposed trade or a counter trade when the proposed quantity is still available in the respective tick. When a trade has been accepted, the transaction can be initiated.

Figure 4.2: State diagram of trade messages and possible replies between peers.



After a tick of a certain peer, peer 1 in this example, gets distributed across the peer-to-peer network, peers with matching ticks will send trade proposals to this peer 1. If the tick is still available, they will reply with a trade accept as can be seen in Figure 4.3. Peer 1 introduces an ask message on the network, which is forwarded by peer 2. Peer 2 has no matching bid, but peer 3 does, and sends a trade proposal to peer 1. Peer 1 checks if the ask is still available and sends a trade accept to the third peer. If the tick is not available anymore, they will reply with a trade decline. If the tick has been partially reserved, they will reply with a counter-offer.

Because messages can reach nodes that are not directly connected to the originating node, and for security reasons it is not wise to relay trade and transaction messages through a third node, it was decided that when a trade is proposed with such a node a direct connection is established. Because of the way the Dispersy peer discovery algorithm works it is not possible to accomplish this without requiring that all nodes have no NAT between the node and the Internet. This requirement is not optimal but safeguard the system better against spoofing and MitM (man-in-the-middle) attacks.

As can be seen in Figure 4.4, and in practice, is that a matching ask and bid do not always contain equal quantities. Peer 1 and peer 2 are connected, and peer 2 and peer 3 are connected. Peer 2 introduces an ask message to his neighbours. Peer 1 and peer 3 peer both have a matching bid and can send a trade proposal to peer 2. If peer 1 sends a trade proposal to peer 2, peer 2 will reserve the proposed quantity, and when it receives the trade, will reserve the accepted quantity. If peer 3 sends a trade proposal which arrives after the trade accept for peer 1, peer 2 has already reserved a part of its quantity, and will send a counter-offer to peer 3 with a reduced quantity. Since peer 3 already reserved a higher quantity for the previous trade proposal, we can assure that peer 3 will send a trade accept and will release the quantity difference.

Figure 4.3: Trade sequence diagram between two peers, with an intermediary peer which distributes ticks for the other peers. All ticks indicated in the image have equal price levels.

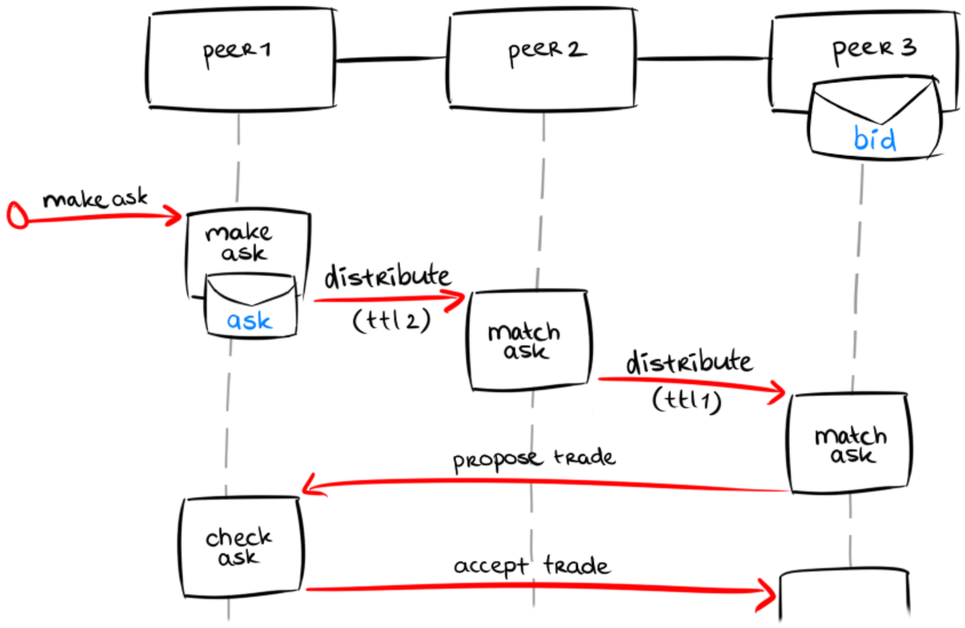
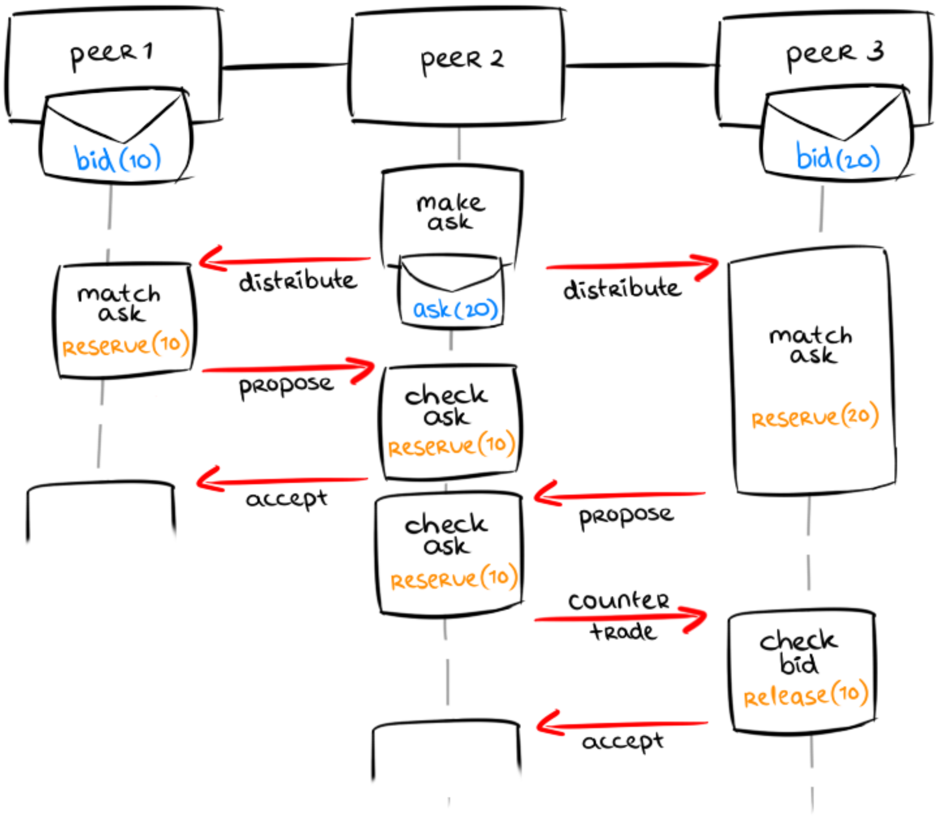


Figure 4.4: Counter trade sequence diagram between three peers. All ticks indicated in the image have equal price levels.



5 Transactions and payments

The market protocol will be fully implemented in this development cycle by adding the final transaction part. For the transaction part to function we integrated Bitcoin and MultiChain interaction to transfer funds.

5.1. Sprint description

Making Bitcoin to MultiChain coin transactions has some disadvantages. Due to the way that MultiChain coin is implemented, the disadvantage exists that it is possible to have a negative amount of MultiChain coin. Because this community will be included in the main Tribler code it would be unwise to include a Bitcoin wallet directly. This would had a great deal of code and complexity, which will only be used if the user opts for selling or buying MultiChain quantity and would also not allow the user to choose a wallet of his or her own choosing. The application instead uses a Bitcoin wallet application programming interface (API), that can connect to any wallet that implements the created API. Given how Bitcoin is implemented, it is not possible to instantly check whether a Bitcoin transaction has been processed: it takes ten minutes to securely confirm a transaction. Since the client wants to have incremental payments implemented into the market community, research needs to be conducted on what is the best solution to approach this, taking the processing time of Bitcoin transactions into account.

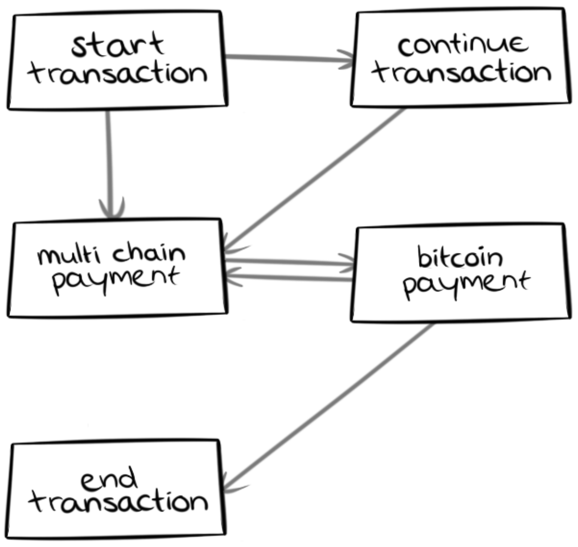
- Continue development on the market community
- Extend and maintain the market community test suite
- Research incremental payments and its efficiency and risk
- Research Bitcoin transactions
- Research Bitcoin transaction confirmation
- Research possible Bitcoin wallet APIs and integrate a Bitcoin wallet API
- Integrate Bitcoin transaction functionality
- Research MultiChain transactions
- Research MultiChain transaction confirmation
- Integrate MultiChain transaction functionality

5.2. Market community transaction protocol

Transaction messages can be categorized in four different types of messages. As can be seen in Figure 5.1, the arrows indicate the possible replies for each message:

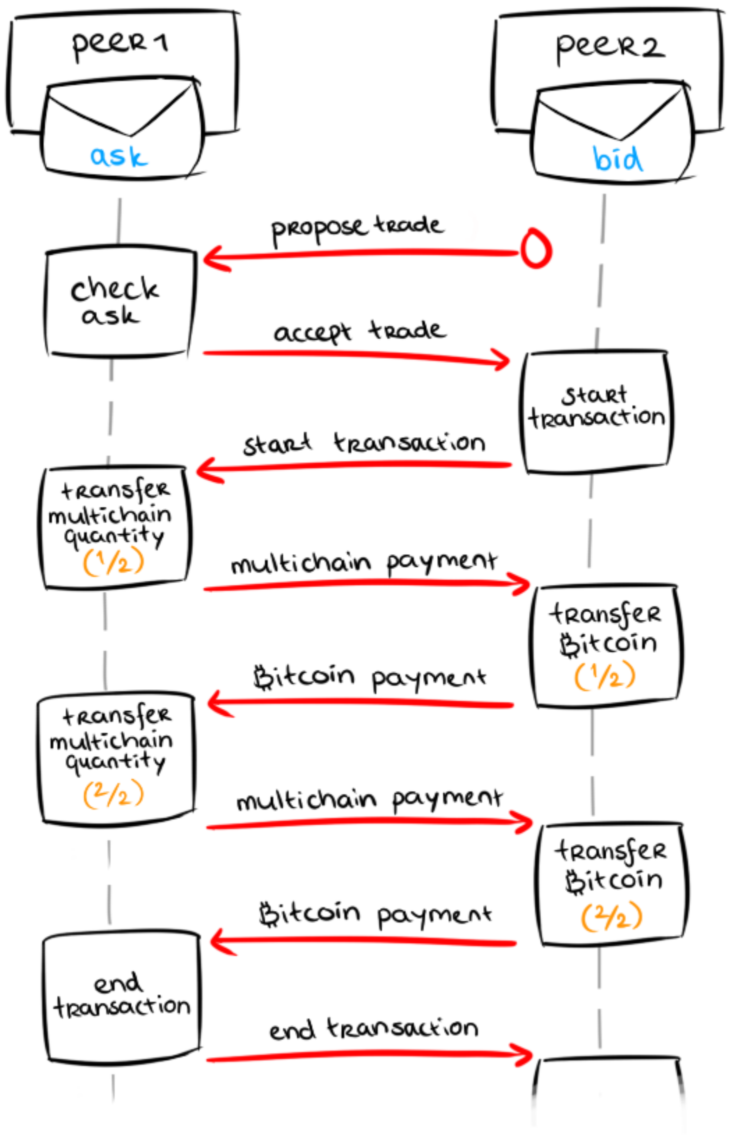
- Start transaction messages are sent as a reply to an accepted trade message. If the peer with the bid sends a trade accept, the peer will reply with a MultiChain payment message. If the peer with the ask sends a trade accept the peer will reply with a continue transaction message.
- Continue transaction messages are sent as a reply to a start transaction message. Continue transaction messages can only be received by the peer with the ask to ensure that the peer with the ask will first transfer the MultiChain quantity. We chose for this scheme because MultiChain is a virtual currency that has less real world value than Bitcoin. If a node that has bad intentions tries to trade with another node and decides not to pay, the currency lost is less significant.
- MultiChain payment messages indicate that the peer has transferred the respective MultiChain quantity to the other peer. MultiChain payment messages can be a reply to a start transaction message, continue transaction message or a Bitcoin payment message. MultiChain payment messages can only be sent by the peer with the ask.
- Bitcoin payment messages indicate that the peer has transferred the respective Bitcoin price to the other peer. Bitcoin payment messages can only be a reply to a MultiChain payment message. Bitcoin transaction payment can only be sent by the peer with the bid.
- End transaction messages are used to indicate that the transaction has been successful when the last batch of Bitcoin price has been received and confirmed. The end transaction message is always a reply to a Bitcoin payment message.

Figure 5.1: State diagram of transaction messages and possible replies between peers.



After a trade has been accepted, a transaction as can be seen in Figure 5.2 will take place. Peer 1 already has a distributed ask, and peer 2 has a matching bid and proposed a trade to peer 1. If peer 1, which has the ask, has also sent the trade accept to peer 2, peer 2 will send a start transaction message to peer 1. If peer 1 has received the trade accept from peer 2, the start transaction will be skipped and peer 1 will start transferring MultiChain coin. This way, we can always ensure that the MultiChain coin from the ask is transferred first, and the Bitcoin from the bid is transferred last. After peer 1 receives the message, they will transfer a part of the MultiChain coin amount to peer 2. After the transaction has been confirmed, peer 1 will send a MultiChain transaction message, which also contains the address to which the Bitcoin transaction has to be made. Peer 2 will now transfer an equal part of the total amount in Bitcoin to the provided address, and after the transaction has been confirmed, they will send a Bitcoin transaction message, and this process will continue until the full amount of Bitcoin and MultiChain coin has been transferred. When the full amount has been transferred, peer 1 will send an end transaction message to indicate that the transaction has been completed.

Figure 5.2: Transaction sequence diagram between two peers.



5.3. Incremental payments

This section discusses incremental payments, and their efficiency and risk. The incremental payment scheme that this program uses has to maximize efficiency, because Bitcoin transactions take time to process. It also has to minimize the risk, which is equal to the highest incremental transaction while also taking into account that the peers that are traded with must build trust before making larger transactions. Those two criteria contradict each other: if you want to decrease the risk and thus the highest incremental transaction, you have to divide the amount using extra transactions, which decreases the efficiency of the transaction.

The system is designed to be trust less. This is done because the Internet itself has not trust built into it. If the system would transfer the entire amount in one transaction, people with bad intentions could easily make abuse of this system. With a incremental payment system, the individual payment amount can be regulated to be small enough based on the risk that is posed. By first transferring a small amount, the risk of people making abuse of the system in the first payment becomes insignificant. After the first payment, the amount is increased and stays at a constant level for the remaining payment. This increased amount is high enough to not stretch the transaction out into the days, but it is small enough to not pose a too big risk for abusing the system. With this system the security increases with the amount. A higher amount would take more payments to complete the transaction. If a peer breaks the trust at any point in the transaction, it will get blocked on the current node for any future transactions. So the maximum amount a peer can acquire through abusing the system is the constant amount that is used for all the transactions after the first one. For this system the client concluded that 0.5 mBTC for the initial payment and 5 mBTC for the incremental payment were suiTable parameters. An example of an incremental payment transaction can be seen in Figure 5.3.

5.4. MultiChain interface implementation

To enable the market community to transfer MultiChain coins for the user, and to query the MultiChain balance of the user, the MultiChain community was introduced as a dependency for this community. We implemented a MultiChain payment provider, which is an interface that contains the logic to transfer MultiChain and request the balance of the user. Due to the implementation of the MultiChain community, we are not able to check for transaction confirmation using a transaction identifier. However, we can implement an approximated check for MultiChain transaction confirmation by storing the MultiChain balance of the user after every transaction, and checking the difference when another transaction confirmation occurs.

This implementation has two disadvantages. The first disadvantage occurs if MultiChain funds are transferred or received outside of the market community. Since the stored MultiChain balance isn't up to date, a false positive or a false negative can occur when checking a transaction for confirmation. The second disadvantage occurs when a user of the market community is trading with two other peers in parallel. If the user receives MultiChain funds from one of the peers, it cannot detect the origins from the MultiChain fund and may confirm the wrong transaction, which results in both a false positive and a false negative. A MultiChain balance callback was implemented in the API to be used by the self-regulating code project, as requested by the client, so there is a uniform method of determining the balance.

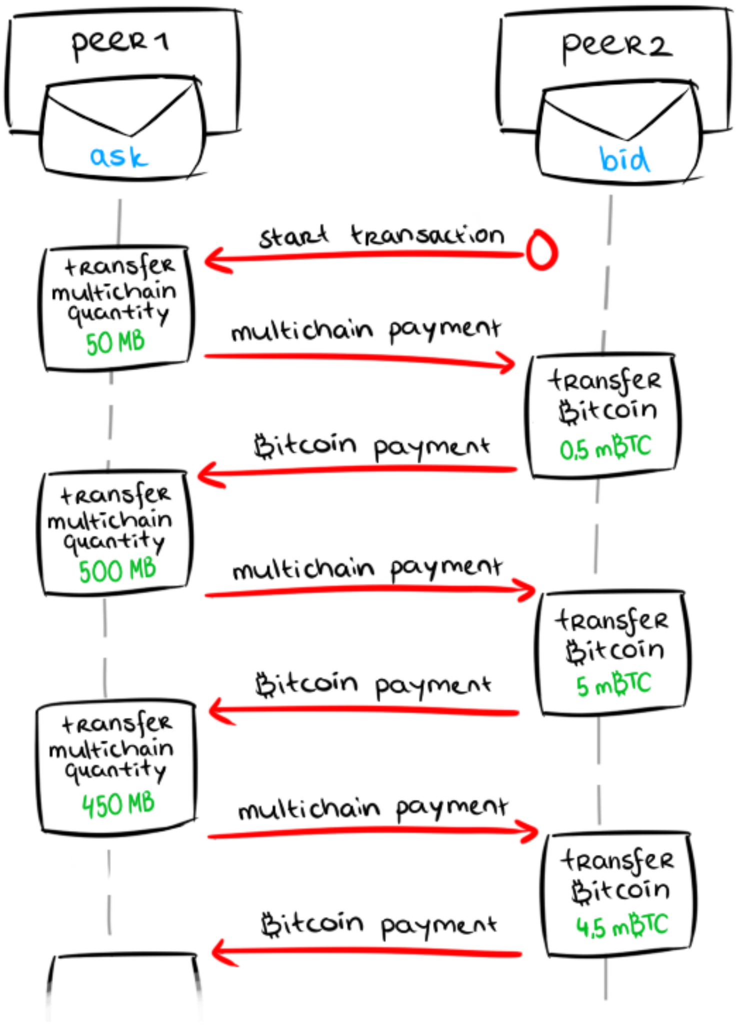
The API that was created for interacting with the MultiChain can be seen in Listing 1. It was formed with the support of the MultiChain team. Schedule block in the MultiChain community can be used to transfer MultiChain quantity to a neighbour. Get total in the MultiChain database can be used to query the MultiChain up- and download quantity of a peer.

5.5. Bitcoin interface implementation

To enable the market community to transfer Bitcoin for the user and to query the Bitcoin balance of the user, a Bitcoin wallet or a Bitcoin wallet API was needed. Together with our coach and client, we decided that we should use the Electrum Bitcoin wallet in our market community. The Electrum Bitcoin wallet was chosen because it is one of the most popular wallets and does not require the whole blockchain to be downloaded. The self-regulating code project also uses the Electrum Bitcoin wallet, which makes communication between the two systems possible.

Since our code is developed within the Tribler repository, it would not be feasible to include the

Figure 5.3: Incremental payment sequence diagram between two peers.



Electrum Bitcoin wallet as a dependency because of the size and complexity. So instead this project chose to communicate with the Electrum Bitcoin wallet through the command line. The users need to install Electrum on their system to use the market community, and if the user does not have Electrum installed, the market community is unavailable to receive or transfer Bitcoin from, and to, other peers. The Bitcoin interface implementation uses two commands to interact with Electrum as described in Listing 2. We tested the implementation by assuming the correctness of the Electrum command line and by mocking the command line. In the test we specified the input and the output as described in the Electrum documentation [3].

```
class MultiChainCommunity(Community):

    def schedule_block(self, candidate, bytes_up, bytes_down):
        % candidate - (Candidate) peer with whom you have interacted
        % bytes_down - (int) bytes you have downloaded from the peer
        % returns - (Order) order that includes all the information

    @property
    def persistence(self)
        % returns - (MultiChainDB) persistence layer for the
        % MultiChain community

class MultiChainDB(Database):

    def get_total(self, public_key):
        % public_key - public_key of the node
        % returns - (total_up (int), total_down (int)) or
        % (-1, -1) if no block is known.
```

Listing 1: Function call signatures in MultiChain community API

```
> electrum getbalance
{”confirmed”: <confirmed_amount>, ”unconfirmed”: <unconfirmed_amount>}

> electrum payto -f 0 <bitcoin_address> <bitcoin_amount>
Which transfers an amount in BTC to the specified address
```

Listing 2: Electrum commands used by the market community.

6 Integration, testing and deployment

After fully implementing the market community protocol we can now start deploying the market community for testing purposes.

6.1. Sprint description

For the final sprint and development cycle we will focus on fixing bugs in the market community and making the market community operational. Now that we have fully implemented the market community protocol that we envisioned during the first sprint we can start with fully integration testing the market community by developing an experimental deployment. We also received feedback from the code quality and maintainability review by SIG. We will focus on fixing the issues indicated to improve the code quality and maintainability of our implementation.

- Continue development on the market community
- Extend and maintain market community test suite
- Develop experimental deployment test
- Find incorrect implementations and bugs in the market community and fix those
- Make improvements based on SIG feedback

6.2. Testing and development process

Parallel to the development of our solution we used test-driven development to preserve the quality and maintainability of our implementation. Before developing a certain feature for our market community we would first decide on the design and the software architecture to use for the development of that specific feature. This allows us to develop a test suite first so we can implement the specific feature using test-driven development. We used continuous integration by using Jenkins in our development work flow. We tested our implementation by developing unit tests for each class inside our market community implementation. A subset of unit tests in the matching engine can be seen in Listing 3. We also developed basic integration tests to check the basic functionality of the market community. We implemented a total of 50+ separate test suites. The final market community test suite report as can be seen in Table 6.1 indicates a total line coverage of +95% calculated over a total of 7000+ lines.

Table 6.1: Line coverage report of the market community implementation files and directories.

Module	statements	missing	excluded	coverage
market	-	-	-	- %
market\community	314	92	0	71%
market\conversion	93	22	0	76%
market\payload	153	0	0	100%
market\socket_address	14	0	0	100%
market\ttl	17	0	0	100%
market\core	-	-	-	- %
market\core\bitcoin_address	8	0	0	100%
market\core\incremental_manager	23	0	0	100%
market\core\matching_engine	124	6	0	95%
market\core\message	75	0	0	100%
market\core\message_repository	16	0	0	100%
market\core\order	107	0	0	100%
market\core\order_manager	37	0	0	100%
market\core\order_repository	51	0	0	100%
market\core\orderbook	132	0	0	100%
market\core\payment	75	0	0	100%
market\core\payment_provider	44	0	0	100%
market\core\price	58	0	0	100%
market\core\pricelevel	65	0	0	100%
market\core\pricelevel_list	42	0	0	100%
market\core\quantity	56	0	0	100%
market\core\side	73	0	0	100%
market\core\tick	66	0	0	100%
market\core\tickentry	31	0	0	100%
market\core\timeout	21	0	0	100%
market\core\timestamp	50	0	0	100%
market\core\trade	103	0	0	100%
market\core\transaction	109	0	0	100%
market\core\transaction_manager	34	0	0	100%
market\core\transaction_repository	47	0	0	100%

6.3. Code quality

In this section we will evaluate the developed decentral market community by its code quality and how we processed the SIG feedback into our final implementation. The SIG feedback in Dutch can be seen in appendix A. We received a code quality score of 4 out of 5. The only code quality issue mentioned in our code is the high issue in the market conversion class and the matching engine class. We fixed this issue for the market conversion class by removing all duplicate code in the market conversion class and turning the removed duplicate code into a separate method. We fixed this issue for the matching engine by moving the matching engine logic in multiple functions. This also made it easier to test the matching engine implementation. We were not able to cover most lines of the matching class with our matching engine test suite which was possible after the refactoring.

```python
class PriceTimeStrategyTestSuite(unittest.TestCase):
    """Price time strategy test cases."""

    def setUp(self):
        # Object creation
        ...

    def test_match_order_divided(self):
        # Test for match order divided over two ticks
        self.order_book.insert_ask(self.ask)
        self.order_book.insert_ask(self.ask2)
        proposed_trades = self.price_time_strategy.match_order(self.bid_order2)
        self.assertEquals(2, len(proposed_trades))
        self.assertEquals(Price(100), proposed_trades[0].price)
        self.assertEquals(Quantity(30), proposed_trades[0].quantity)
        self.assertEquals(Price(100), proposed_trades[1].price)
        self.assertEquals(Quantity(30), proposed_trades[1].quantity)

    def test_match_order_different_price_level(self):
        # Test for match order given an ask order and bid in different price levels
        self.order_book.insert_bid(self.bid2)
        proposed_trades = self.price_time_strategy.match_order(self.ask_order)
        self.assertEquals(1, len(proposed_trades))
        self.assertEquals(Price(200), proposed_trades[0].price)
        self.assertEquals(Quantity(30), proposed_trades[0].quantity)

    def test_match_bid_order_insufficient(self):
        # Test for match order with insufficient tick quantity
        self.order_book.insert_ask(self.ask)
        proposed_trades = self.price_time_strategy.match_order(self.bid_order2)
        self.assertEquals(0, len(proposed_trades))
```

Listing 3: Several tests in the matching engine / price time strategy test suite.

6.4. Experiment

A smaller experiment was executed after the completion of the trades. In this experiment basic examples like shown in Figure 4.3, Figure 4.4 and Figure 5.2 were reproduced and tested for completeness and correctness. The experiment was concluded to be a success, because it fulfilled all the requirements of the application to that point in the development.

At the end of the project a real life money trial was conducted in cooperation with the Autonomous self-replicating code group to test if the whole system would work when put to the test. The success of this experiment is harder to evaluate because of the nature of decentral system. Every time a scenario is run, the composition of the nodes in the overlay changes and different trades are possible. The experiment was considered complete, because trades were made and payed for by transactions.

It was not possible to extract meaning full test data from the experiment because of time constraints. Extracting this data would require setting up a complex testing framework and infrastructure and that would have been out of the scope for this project.

6.5. Market community API

In this section the market community API as shown in Listing 4 developed for the Autonomous self-replicating code project [5] is discussed. The Autonomous self-replicating code group needs to be able to create asks and bids to trade MultiChain quantity that their system earned by providing bandwidth for uploading. To check how much MultiChain quantity is available to trade we implemented a function to provide a stable API to our MultiChain payment provider. The order book also provides statistical functions for the other team so they can use different strategies for their genetic algorithm.

```
class MarketCommunity(Community):

    def create_ask(self, price, quantity, timeout):
        % price - (float) Bitcoin price in BTC
        % quantity - (float) multichain bytes in MB 10^6
        % timeout - (float) time when the order must expire
        % returns - (Order) order that includes all the information

    def create_bid(self, price, quantity, timeout):
        % price - (float) Bitcoin price in BTC
        % quantity - (float) multichain bytes in MB 10^6
        % timeout - (float) time when the order must expire
        % returns - (Order) order that includes all the information

    def get_multichain_balance(self)
        % returns - (Quantity) amount of multichain in MB 10^6

    @property
    def order_book(self)
        % returns - (OrderBook) order book containing all the offers

class OrderBook(object):

    def bid_price(self):
        % returns - (Price)
        % the price an ask needs to have to make a trade

    def ask_price(self):
        % returns - (Price)
        % the price a bid needs to have to make a trade

    def bid_side_depth_profile(self):
        % returns - (List[(Price, Quantity)])
        % the list of bids in the provided format

    def ask_side_depth_profile(self):
        % returns - (List[(Price, Quantity)])
        % the list of asks in the format provided
```

Listing 4: Function call signatures in market community API

7 Discussion

In this chapter we will be giving a number of recommendations to the client that can be used to further improve the implementation. In this chapter we will evaluate the requirements that have been fulfilled. At the end we will be giving a number of recommendations to the client that can be used to further improve the implementation.

7.1. Fulfillment of project requirements

To determine the requirements of the clients and the priorities of these requirements during the development / implementation phase of the project we used the MoSCoW method [21]. In this section we will look back to the project requirements and will discuss which requirements have been fulfilled and how these have been fulfilled.

Must haves

The most important requirements for the project are described in the "must haves" subsection of the MoSCoW method. The "must have" requirements are the minimal, usable subset of requirement which are guaranteed to be delivered for the project. The following described requirements have the highest priority, and must be fulfilled in the final implementation of the solution:

- ✓ *Users must be be able to place a bid and a ask which must be be distributed across the connected part of the network.*
- ✓ *The market community solution must be scalable with a lot of users.* Because we implemented the market community upon Dispersy it is scalable. Tribler is proof for the scalability of Dispersy.
- ✓ *The market community solution must be able to match a bid and an ask between peers that are connected.*
- ✓ *The market community solution must be implemented into Tribler.*
- ✓ *The market community solution must be a decentralised system.*
- ✓ *The market community solution must be constructed on Dispersy.*
- ✓ *The market community solution must be able to make Bitcoin transactions and MultiChain transactions.* The market community can make transactions if users have installed Electrum so it can be called from the command line. The market community can also make MultiChain transactions although there isn't any possibility for MultiChain transaction confirmation.

Should haves

If all "must have" requirements have been fulfilled, the "should have" requirements have the next-highest priority. The "should have" requirements are important for the project but are not vital to its completion; the solution would still be viable without fulfilling these requirements. All of the following requirements should be fulfilled in the final implementation of the solution:

- ✓ *The market community solution should be able to let users use incremental payments.*
- ✓ *The market community solution should be tested to a certain extent.* We used test-driven development to preserve the quality and maintainability of our implementation. The testing report of our implementation can be seen in table 6.1.

Could haves

The "could have" requirements are the requirements with the lowest priority. If all "should have" requirements have been fulfilled, the "could have" requirements have the next-highest priority. The "could have" requirement are desirable requirements for the project, but they will have less impact when left out compared to "should haves". The following requirements could be fulfilled in the final implementation:

- ✗ *The market community solution could be able to blacklist peers on an individual blacklist.*
- ✓ *The market community solution could be able to maintain an efficient connectivity between peers so that a maximum of possible trades are proposed without creating too much overhead on the network.*
- ✓ *The market community solution could fully function.*

Won't haves

The "won't have" requirements are requirements that won't be implemented in the final implementation. We have agreed that these requirements will not be delivered. The following requirements will not be fulfilled in the final implementation:

- ✗ *The market community solution in Tribler won't have a nice graphical user interface.*
- ✗ *The market community solution won't have to protect the privacy of its users.*
- ✓ *The market community solution won't have to be secured against spoofing.*

7.2. Conclusion

The goal of this bachelor project was to create an operational decentral electronic marketplace which will have no central server bottleneck, no central point of trust, full self-organization and unbounded scalability. Users should be able to trade digital currencies with other users through a decentral electronic marketplace.

The market community solution we developed for this project is decentral, scalable and well-tested. While the previous bachelor end project group delivered a proof of concept for a decentral market we delivered a scalable decentral market by building our market community on top of Dispersy. Users of the market community are able to trade Bitcoin for MultiChain balance and MultiChain balance for Bitcoin with other users. Due to MultiChain still being in development there are still some parts that need to finalised before this product can be fully operational in production.

There are some known issues concerning our market community implementation:

- We were not able to implement the functionality which can confirm whether a MultiChain transaction has been made. There is no functionality in the MultiChain community which makes it possible to confirm whether a MultiChain transaction has been made.
- Due to MultiChain being insecure, the Market community also is insecure. There is no security measure for having a negative MultiChain balance. By running two MultiChain communities a user could be able to gain an infinite amount of MultiChain this way.

To further improve the implementation, we have giving a number of recommendations to the client:

- A trust system should be implemented to increase the security of the system. If a peer has proven to have bad intentions on the market community his reputation should be distributed to other peers to prevent theft.
- The requirement for not having a NAT between the peers could be removed by implementing a new peer discovery algorithm.

References

[1] Bitmarkets. URL https://voluntary.net/bitmarkets/whitepaper/.

[2] Dispersy Documentation. URL http://dispersy.readthedocs.io/.

[3] Electrum Documentation. URL http://docs.electrum.org/en/latest/.

[4] Tribler Architecture. URL https://www.tribler.org/TriblerArchitecture/.

[5] N.C. Bakker, R.S. van de Berg, and S.A Boodt. Autonomous self-replicating code. *TU Delft, Parallel and Distributed Systems*, 2016.

[6] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[7] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[8] V. Dumitrescu. Rewarding Good Behavior in Peer-to-Peer Networks, 2013. URL http://repository.tudelft.nl/view/ir/uuid{%}3Ad2a72d3f-d28b-4d9d-998a-b030c2f28aed/.

[9] M.D. Gould, M.A. Porter, and S. Williams. Limit order books. *Quantitative Finance*, 2013. URL http://www.tandfonline.com/doi/abs/10.1080/14697688.2013.803148.

[10] G. Logiotatidis. Splash: Data synchronization in unmanaged, untrusted peer-to-peer networks, 2010. URL http://repository.tudelft.nl/view/ir/uuid{%}3A52b586ea-6144-4b4e-a5a1-b05255ce493a/.

[11] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[12] S.D. Norberhuis. MultiChain: A cybercurrency for cooperation, 2015.

[13] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M.R. Van Steen, H.J. Sips, et al. Tribler: A social-based peer-to-peer system. *Concurrency and computation: Practice and experience*, 20(2):127, 2008.

[14] R. Rahman. Peer-to-Peer System Design: A Socioeconomic approach, 2011. URL http://repository.tudelft.nl/view/ir/uuid{%}3Aa14fc07c-7c6c-45cf-8d8b-c03533e6f603/.

[15] B. Schoon. Building a neighbourhood with Dispersy. pages 1–17, 2013. URL https://github.com/Tribler/dispersy/blob/d056b388836cec227f6752683045534325b5b5e4/doc/component_walker.pdf.

[16] K. Soska, A. Kwon, N. Christin, and S. Devadas. Beaver: A decentralized anonymous marketplace with secure reputation. 2016.

[17] M. The and H. Reinbergen. Tsukiji: A decentral market. *TU Delft, Parallel and Distributed Systems*, 2015.

[18] L.J. Trautman. Virtual currencies; bitcoin & what now after liberty reserve, silk road, and mt. gox? *Richmond Journal of Law and Technology*, 20(4), 2014.

[19] S. Viswanathan and J.D. Wang. Market architecture: limit-order books versus dealership markets. *Journal of Financial Markets*, 5(2):127 – 167, 2002. doi: http://dx.doi.org/10.1016/S1386-4181(01)00025-8. URL http://www.sciencedirect.com/science/article/pii/S1386418101000258.

[20] J. Warren. Bitmessage: A peer-to-peer message authentication and delivery system. 2012. URL https://bitmessage.org/bitmessage.pdf.

[21] K. Waters. Prioritization using moscow. *Agile Planning*, 12, 2009.

[22] N. Zeilemaker and J.A. Pouwelse. ReClaim: a Privacy-Preserving Decentralized Social Network. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, 2014. URL https://www.usenix.org/conference/foci14/workshop-program/presentation/zeilemaker.

[23] N. Zeilemaker, M. Capota, and J.A. Pouwelse. Open2Edit: A peer-to-peer platform for collaboration. 2013. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6663524.

[24] N. Zeilemaker, B. Schoon, and J.A. Pouwelse. Dispersy bundle synchronization. *TU Delft, Parallel and Distributed Systems*, 2013.

[25] N. Zeilemaker, B. Schoon, and J.A. Pouwelse. Large-scale message synchronization in challenged networks. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 481–488. ACM Press, 2014. ISBN 9781450324694. URL http://dl.acm.org/citation.cfm?id=2554850.2554908.

A
Initial SIG Feedback (Dutch)

De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size. Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. In jullie project is MarketConversion._decode_offer een voorbeeld van zo'n lange methode. Deze lengte is ook onnodig, als je naar de code kijkt herhalen jullie steeds hetzelfde stuk code. Zeker in Python zijn er diverse oplossingen om de hoeveelheid code naar beneden te krijgen, zodat je in ieder geval niet steeds hetzelfde try/catch blok hoeft te herhalen. Een ander voorbeeld van een lange methode is PriceTimeStrategy.match_order. De oorzaak van de lengte is hier de hoeveelheid functionaliteit die in de methode wordt geïmplementeerd: je ziet nu dat jullie de verschillende delen van elkaar scheiden met commentaar. Het is beter om dat met code-structuur te doen, dus er aparte methodes van te maken. Op die manier wordt je code beter leesbaar, en is het ook makkelijk om unit tests voor de aparte gedeeltes te schrijven. Over unit tests gesproken: jullie hebben een goede hoeveelheid testcode geschreven. Hopelijk lukt het om tests te blijven toevoegen in het vervolg van het project.

B Infosheet

Self-regulating electronic market
Tribler Team / Distributed Systems group
Date of final presentation 24-06-2016

Project Description:
A decentral electronic marketplace to trade digital currencies was created for this project. Users may trade MultiChain balance for Bitcoin. MultiChain can be described as up- and download currency in a peer-to-peer network. When a peer uploads, the balance of that peer increases and the peer can download more effectively. The client for this project is the Tribler Team. Most of the Tribler Team's work revolves around Tribler: an open-source peer-to-peer program which enables its users to find, enjoy, and share content. An earlier bachelor end-project implemented a partial, functional prototype of this idea. We improved the existing concepts and re-implemented a decentralised market from scratch with scalability and as much security as possible, given the limited available development time. This market is the first electronic marketplace to be fully decentralized. There were previous attempts to create such a marketplace, but those implementations fell short in scalability and security. During the research phase, we discovered that the previous decentral market project was not scalable and not production ready, so we made the decision to, together with our client, instead build our application upon a more production-level network platform known as Dispersy. As nothing on the Internet can be trusted which is a problem on its own, we could not make the product as secure as we would have wanted to.

We held meetings with our coach and client every week to discuss the preliminary progress and incorporate any feedback that they provided. This made sure that we were able to adjust our goals more towards the client needs. We also had daily meetings within our groups to keep organized and be efficient. During development we used test driven development methodology to have a solid tested product to suit or production level code. During the development we ran into the unexpected challenge of needing to integrate with another bachelor end project group who was building an application on top of ours at the same time. We dealt with this situation by tightly cooperating and documenting decisions. We made the recommendation to our client to improve upon security and trust before putting the product in production use.

Members of the project team:
M.J.G. Olsthoorn
Interest: Back-end development, security, software architecture
Role & Contribution: Developer, Dispersy integration, matching engine

J. Winter
Interest: Back-end development, testing, graphical design
Role & Contribution: Developer, test development, payment providers

All team members contributed to preparing the core implementation, report and the final project presentation.

Client & Coach
Dr. Ir. J.A. Pouwelse, DS Group, TU Delft (Client)
Ir. E. Bouman, MMC Group, TU Delft (Coach)

Contacts
Dr.Ir. J.A. Pouwelse, DS Group, TU Delft, j.a.pouwelse@tudelft.nl
J. Winter, info@joswinter.nl

The final report for this project can be found at: http://repository.tudelft.nl.
The project code is available at https://github.com/mitchellolsthoorn/tribler.

C Original Project Decription

C.1. Project description

You will create an operational electronic market place that is immune to shutdown by governments, immune to lawyer-based attacks or other real-world threats. The Internet-deployed marketplace will have no central server bottleneck, no central point of trust, full self-organization and unbounded scalability.

To keep this project realistic, the focus will be on re-using existing code, support only Bitcoin-Multichain trading, only use a single marketmaker, and offer no anonymity or DDoS protection. Bitcoin, Mt Gox, and Silk Road showed how vulnerable marketplaces are. Bitcoin was the first ever successful cybercurrency after decades of failures by scientists, anarchists, fraudsters, and entrepreneurs. However, this first-generation technology suffers from an highly unstable exchange rate, low usage level and frequent large-scale thefts by hacking exchange platforms.

TUDelft has a stash of Bitcoins which will be used to buy MultichainCoins during your final presentation on your platform. Multichain coins are developed at TUDelft. They provide a reward for relaying Tor-like onion routing traffic and Bittorrent seeding. You will use Python, Twisted, nosetest, jenkins, and Libtribler during your work.

C.2. Company description

Tribler Team. Creating disruptive cooperative software since 1999.

Read about the team at these URLs:

http://http://www.foxnews.com/tech/2012/02/10/forget-megaupload-researchers-call-new-fi
html, http://http://www.ee.princeton.edu/events/anonymous-hd-video-streaming-and-reputatio
http://news.harvard.edu/gazette/story/2007/08/creating-a-computer-currency,
http://tweakers.net/nieuws/98175/torrentclient-tribler-gebruikt-onderdelen-tor-voor-anoni
html, http://http://www.elsevier.nl/Tech/blogs/2014/12/Johan-Pouwelse-Tribler-Mijn-ideaal-

Multichain coins documentation: http://repository.tudelft.nl/view/ir/uuid%3A59723e98-ae48-4fac-

C.3. Auxiliary information

Your starting point is the operational code from previous BEP project: https://github.com/Tribler/decentral-market

Multichain coins documentation: http://repository.tudelft.nl/view/ir/uuid%3A59723e98-ae48-4fac-

D
Market protocol

D.1. Ticks

- **Ask tick**
  - trader_id
  - message_number
  - order_number
  - price
  - quantity
  - timeout
  - ttl
  - ip
  - port
  - timestamp
- **Bid tick**
  - trader_id
  - message_number
  - order_number
  - price
  - quantity
  - timeout
  - ttl
  - ip
  - port
  - timestamp

D.2. Trade

- **Proposed trade**
  - trader_id
  - message_number
  - order_number
  - recipient_trader_id
  - recipient_order_number
  - price
  - quantity
  - timestamp
- **Counter trade**
  - trader_id
  - message_number
  - order_number
  - recipient_trader_id
  - recipient_order_number
  - price
  - quantity
  - timestamp
- **Declined trade**
  - trader_id
  - message_number
  - order_number
  - recipient_trader_id
  - recipient_order_number
  - timestamp
- **Accepted trade**
  - trader_id
  - message_number
  - order_number
  - recipient_trader_id
  - recipient_order_number
  - price
  - quantity
  - ttl
  - timestamp

D.3. Transaction

- **Start transaction**
  - trader_id
  - message_number
  - transaction_trader_id
  - transaction_number
  - order_trader_id
  - order_number
  - trade_message_number
  - timestamp
- **Continue transaction**
  - trader_id
  - message_number
  - transaction_trader_id
  - transaction_number
  - timestamp
- **Multi chain payment**
  - trader_id
  - message_number
  - transaction_trader_id
  - transaction_number
  - bitcoin_address
  - transferor_quantity
  - transferee_price
  - timestamp
- **Bitcoin payment**
  - trader_id
  - message_number
  - transaction_trader_id
  - transaction_number
  - price
  - timestamp
- **End transaction**
  - trader_id
  - message_number
  - transaction_trader_id
  - transaction_number
  - timestamp