Delft University of Technology

Faculty of Electrical Engineering,
Mathematics and Computer Science

Bachelor Thesis

Tsukiji, a decentral marketplace

Authors:
Michael The
Hugo Reinbergen

Supervisor:
Johan Pouwelse
TU Coach:
Raynor Vliegendhart

July 2, 2015

Preface

This report describes the development process of creating a proof of concept for a decentral market, named Tsukiji. This project was initiated as part of the TI3806 Bachelor Project course of the Computer Science program at the Delft University of Technology.

We would like to thank Johan Pouwelse for giving us the opportunity to work on this project and for providing guidance on how to attack the issue of decentralisation. We also want to thank Raynor Vliegendhart for assisting us while developing the Bachelor Project report. Finally, we want to give our thanks to Martha Larson and Felienne Hermans for supervising the project course.

Michael The
Hugo Reinbergen

Summary

This project creates a marketplace to trade commodities for real money. While many online marketplaces exist, they usually have one thing in common: a central point of authority. Our solution is Tsukiji, a completely decentralised marketplace, capable of scaling to thousands of users.

Research has shown that implementations of this idea are scarce. Since the project is assigned by the group responsible for creating Tribler [2], this project follows their example in programming languages and libraries, such as Python and Twisted [3].

Development of Tsukiji occurred in sprints of around 1 week, each sprint focusing on one or two specific features. Meetings with the product owner were planned after every development cycle, giving us feedback on the previous sprint and guiding the focus of the following sprint.

The result is a light-weight, terminal-based, python application called Tsukiji. Users can post offers on the network and trade commodities for money. They communicate by sending messages using the gossip protocol, implemented with the Twisted library. Financial transactions are made using PayPal. With this, a fully decentralised market with a trustworthy payment system is created, allowing users to make trades without the risk of their trading platform being shut down.

Contents

List of Figures

1 Introduction

The Internet is becoming the town square for the global village of tomorrow.

Bill Gates

The appearance of online marketplaces in the late 90s marked the beginning of a new era for consumers. The connection between supplier and consumer is closer than ever. Even trading between individuals has become commonplace. Sites like ebay, Amazon, and Alibaba all provide consumer-to-consumer, business-to-consumer and even business-to-business services. Compared to traditional shopping, these platforms are easier to use, cost less time, have a wider range of products available and are better at matching buyer and seller.

Another example of such a marketplace is currency exchange platforms. In recent years, Bitcoin has become of great interest to traders. With the rise of interest in Bitcoin, the rise of Bitcoin exchange markets comes naturally. The most famous of Bitcoin exchanges is the now defunct Mt.Gox. Mt.Gox was plagued by problems early on that eventually led to its bankruptcy [4]. With its downfall, it took a large amount of funds with it. The trust users have put into the institution was violated.

Similar stories exist all over the place. Hacking attempts, government shutdowns, or simply incompetent companies. These are all common among marketplaces of all kinds. In this report we present *tsukiji*, a decentralized marketplace. This proof-of-concept marketplace has no central server bottleneck, no central point of trust, full self-organisation, and unbounded scalability.

Motivation and inspiration for creating Tsukiji originally came from Bittorrent communities. The sharing of files has become more and more popular over the years. Napster, Kazaa and LimeWire no longer exist, but one protocol has survived: BitTorrent. BitTorrent is a peer-to-peer file sharing protocol. Instead of downloading from a central server, each peer downloads from and uploads to other peers. It offered a solution to the bottleneck of existing programs and protocols.

With more and more people using BitTorrent, the amount of people downloading files naturally increases. Ideally, every peer should upload at least as much as they download [5]. Sadly, this does not happen in reality. The amount of users only downloading and not uploading is significantly higher than the amount of users uploading at least as much as they download. There is no incentive for an individual to upload. With few peers who upload the download speed is significantly hampered. To solve this issue, uploaders decided to work together in separate, private groups that only allow people with a good upload/download ratio to participate.

This seems like a good solution to motivate peers to actively upload, but a couple of issues occur [6]. With peers motivated to upload, a certain oversupply of uploading rises. A peer with a low speed connection, has a hard time competing with many high-speed connections within the group. This peer now loses reputation because of competition, even though they are putting in the effort to upload.

Unpopular files form another issue. If someone downloads a large file and wants to upload it to increase his reputation, it's possible that nobody is interested in downloading that file. This gives the peer a large deficit in his reputation that he cannot solve by uploading his unpopular taste, despite their good intentions.

Torrenting may also come with legal issues. For instance, in Switzerland it is legal to download copyrighted material, whereas uploading is illegal [7][8]. Citizens of such countries would like the high speed download of private communities, but cannot participate since they are legally not allowed to share whatever they have downloaded.

To solve these issues, peers need a way te obtain reputation in a way other than uploading. We propose that users can trade reputation points for currency. Reputation is still gained by uploading. Peers can create offers on a marketplace to buy or sell reputation points. This will give peers the incentive to upload data to the swarm. At the same time, it will still be possible for peers to have access to the community by spending money and buying into a community. This will lead to a more accessible and scalable community than traditional private torrent communities.

Keeping in line with BitTorrent's philosophy, Tsukiji will be fully decentralised and peer-to-peer. With asynchronous cryptography, users will be able to keep their real identity hidden, while still being able to make trades with their network-alias. Certain users will attempt to upload as much as possible to sell large amounts of reputation, keeping the network attractive with many uploaders. With no central authority, the market is completely free to decide on a price for their reputation. Intermediaries have no added value, since peers can sell directly to each other, keeping the price as low as possible.

2 Problem definition

mar - ket - place
1. An open square or place in a town where markets or public sales are held.
2. The world of trade or economic activity.

Merriam-Webster

To combat the issues associated with centralised marketplaces, Tsukiji aims to be a fully decentralised marketplace.

A marketplace is a place where public sales are held. For our purposes, a marketplace is a place where people can place bids or asks. A bid is placed when a buyer wants to buy goods for a set price. An ask is placed when a seller wants to sell goods for a set price.

Using the example of BitTorrent communities, a transaction would occur as follows: a seller places an ”ask”, wanting to sell 500 MBit for €3. An interested buyer sees the ask and requests a trade to be made. The buyer and seller exchange information necessary for the transaction to occur (e.g. bank account numbers). A trade is made and the original ask is taken down. A similar transaction occurs when a ”bid” is placed.

Current marketplaces are centralised. Usually, this type of centralised system follows the client-server model, where one server serves a large amount of clients. In this model, information passes through one central point. This means that if this central point fails, the entire marketplace is shut down. A decentralised system operates without such a single point. Instead of a central server connected to several users, peers are connected to each other. This follows the peer-to-peer model. If a single peer fails, this might impact the marketplace, but it will not shut it down.

For a network of such peers to exist, peers first have to find each other. Peer discovery is generally done by exchanging information about other peers. When a peer first signs on, it knows of no other peers. We can bootstrap its network by having them connect to a predefined set of super peers, who are known to be well connected. From there, they can exchange information about other peers.

2.1 Related work

This section discusses earlier created software projects that have attempted to solve the described problem or offer solutions for parts of the problem.

- **pyLOB**

PyLOB [9] is an open source project that simulates a financial exchange. It does this by keeping an orderbook that contains all previously made bids and asks. Whenever two offers match, it automatically simulates a trade and removes the offers from the orderbook. While PyLOB is not decentralised, the simulation helps to understand how bids, asks, and matching trades work. It also contained a large dataset of offers from a real exchange, which can be used in different simulations.

- **BarterCast**

BarterCast [6] is a fully distributed system that manages reputation of peers among a network. It uses the epidemic protocol, also called gossip protocol, to spread information about the amount of data uploaded and downloaded by every peer. Users uploading data use this information to decide who gets priority on receiving files. Peers that have a better reputation of uploading get more slots to download from, since more peers are willing to share their data with them. We used the same epidemic communication protocol in Tsukiji as scalable messaging solution.

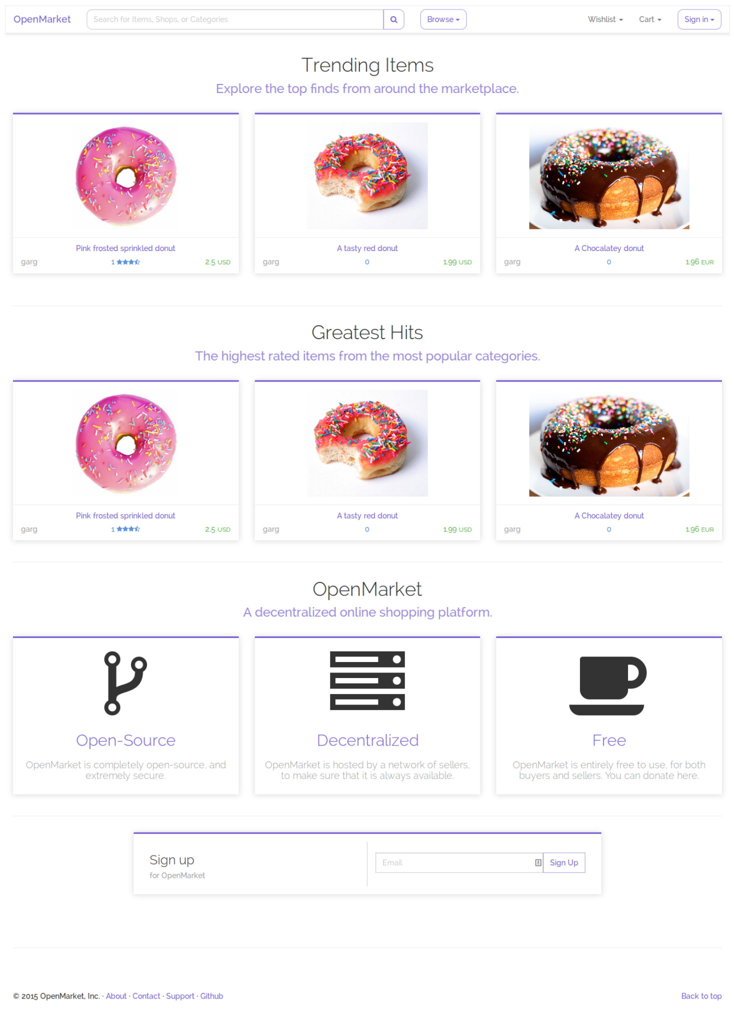
The following projects were found later in the development and their ideas did not influence Tsukiji's development process. They are nonetheless closely related projects and are worth mentioning.

- **Bitmarkets**

Bitmarkets[10] is a free OSX application for a decentralised marketplace. Users communicate using Bitmessage [11] to hide the identity of the creator of a message. These messages are sent to other users via the Tor [12] network. This way both the identity and location of the originator are hidden. The only thing visible is a public key generated by Bitmessage that can be used to reply to. Payments are done using BitCoin's multi-signature transactions. This requires both parties to sign a transaction before the BitCoins are moved from wallet to wallet.

- **OpenMarket**

OpenMarket [13] is another BitCoin-based decentralized market. Rather than one large marketplace of buyers and sellers, OpenMarket lets people set up their own webshop. This locally hosted website supports all functionality one would expect of a webshop. Items to be sold can be added to the store with images and prices. Potential buyers can go to this site and browse the catalog of this particular seller. Whenever a buyer wants to buy an item, they can add the item to a shopping cart and can checkout the whole cart. Payment is handled by the BitCoin network.



Trending Items

Explore the top finds from around the marketplace.

Greatest Hits

The highest rated items from the most popular categories.

OpenMarket

A decentralized online shopping platform.

Open-Source

OpenMarket is completely open-source, and extremely secure.

Decentralized

OpenMarket is hosted by a network of sellers, to make sure that it is always available.

Free

OpenMarket is entirely free to use, for both buyers and sellers. You can donate here.

Sign up
for OpenMarket
Email
Sign Up
© 2015 OpenMarket, Inc. · About · Contact · Support · Github
Back to top

Figure 1: Example of a webstore created by OpenMarket[1]

The difference between Tsukiji and OpenMarket is that OpenMarket is not one large marketplace. It is a lot of small markets that have no knowledge of each other. There is no network of markets, just a large amount of separate ones.

- **OpenBazaar**

OpenBazaar [14] is also a decentralised marketplace where goods are traded with BitCoin. It uses a Distributed Hash Table to connect peers to each other in a scalable way. The software opens a webpage that represents the bazaar and users can create their own shops on the bazaar. Users can visit other shops and buy items from them, or simply wait until items are bought from their own shop. Items being sold are hashed and this hash is used to create a contract. Whenever someone buys an item, they sign the contract and send it to the seller. The seller then sends it to an independent third person, or escrow, on the network. This escrow signs the contract and the buyer sends bitcoin to a multisig deposit that can only be opened by at least two of the three keys. Any conflicts between the two parties (e.g., one side did not receive their goods) are resolved by an independent arbiter [15]. While this is a great structure to reduce trust issues between parties, it is reliant on BitCoin. Not all forms of currency are as easily locked away and retrieved.

2.2 Requirements

The goal of Tsukiji is to create a decentralised market where users can trade commodities. Tsukiji is primarily a proof-of-concept, i.e. it should showcase the technologies and viability of a decentralised market. While Tsukiji is a proof-of-concept, this does not mean we have to forego a usable application altogether. The requirements are summarized using the MoSCoW method. This method assigns a priority to the requirements. The importance of a requirement influences the planning during the project.

2.2.1 Must Have(s)

- Place an offer

  The basics of a marketplace should work. This means users should be able to place an offer, i.e. an ask or bid.

- Respond to an offer, facilitating a trade

  They should also be able to see offers from other users. From an offer, a trade can be initiated.

- Decentralised

  The system should be decentralised. The lack of a clear single point of failure is the theme of this project. Tsukiji can take inspiration from peer-to-peer networks such as BitTorrent.

- Peer discovery

  Paired with a decentralised system comes some form of peer discovery.

- Scalable to thousands of users

  It should scale to thousands of users. If our goal is to create a usable application, the network should be able to handle many connections. The choice for thousands of users (as opposed to hundreds or millions) is motivated by the scale of other applications, such as Tribler [2].

- Command line input

  A good user interface is not a priority of this project. While it is important for a usable application, a lot of time would have to be devoted to this. We believe this time is better spent on other areas of the application. The user interface will feature the bare minimum, i.e. a command line interface.

2.2.2 Should Have(s)

- Trade using real money

  A user should be able to trade real commodities using real money. However, it is not a must, because virtual currencies and commodities are good enough to show that the proof of concept works.

2.2.3 Could Have(s)

- Nice user interface

  As explained above, a nice user interface is not a must. However, a better user experience is still an advantage as it allows the general public to use the application also.

2.2.4 Would Have(s)

- Privacy

  It would be nice to ensure the privacy of the users. In order to achieve this, strong encryption of messages is required. Further thought is also necessary on what information is exposed. At the very least, no personal information should be stored. Networking information, such as ip addresses, are exposed.

- Protection against hostile attacks

  It would be nice to have a network resistant against hostile takeover. Having a decentralised system plays a big role in this. However, other attack vectors could also be possible. The main weapon against this is a well thought out protocol. However, to keep this project viable within the timespan given, we will assume there to be no active attackers.

- Anti-spoofing

  Spoofing of message would of course form a big problem in a trading network. However, since this is a solved problem, the focus of the project should be on other areas. For now, it is assumed that every message is legitimate.

3 Design

This chapter describes the general design of Tsukiji. We will discuss the architecture of the code base of Tsukiji and the dependencies that the program relies on. For more detailed issues during the process of implementation, see section 4.

3.1 Architecture

Tsukiji consists of two main modules: orderbook and trader. There are several smaller utility modules, the most important of these being the cryptography module. How these modules interact is depicted in figure 2. There are also several third-party libraries Tsukiji depends on. The important ones are described later on in section 3.1.1.

The trader module consists mainly of the Trader class. Its main purpose is to communicate with other remote Traders. The Trader class is a subclass of the DatagramProtocol class provided by the Twisted library [3]. This means the Trader class implements the UDP protocol. The UDP protocol is the transport protocol used to talk to other traders. UDP is chosen over TCP, because UDP travels more easily over NATs and firewalls. This follows the design of Tribler and BitTorrent. In section A the application protocol is specified in detail.

The orderbook module has three main functionalities: 1. message creation for the application protocol and 2. managing the state of the orderbook itself and 3. matching bids and asks. Message creation is used in combination with the trader module. These are the messages that are passed around between peers. Managing the state of the orderbook is an important task. The orderbook keeps track of what offers come in, what offers are expired, or what trades have been made. Finally, the matching algorithm is a simple algorithm. For every new offer, incoming or outgoing, the algorithm checks whether there is an existing offer that matches with the new offer. It tries to match favourable trades. These functionalities are closely related, e.g. matching an ask with a bid produces a trade message, after which the orderbook is updated.

The cryptography module contains everything related to actions with public-key cryptography. At the moment, it mostly functions as a wrapper around the PyCrypto library, for easier use within Tsukiji. More on this can be read in section 4.1.4.

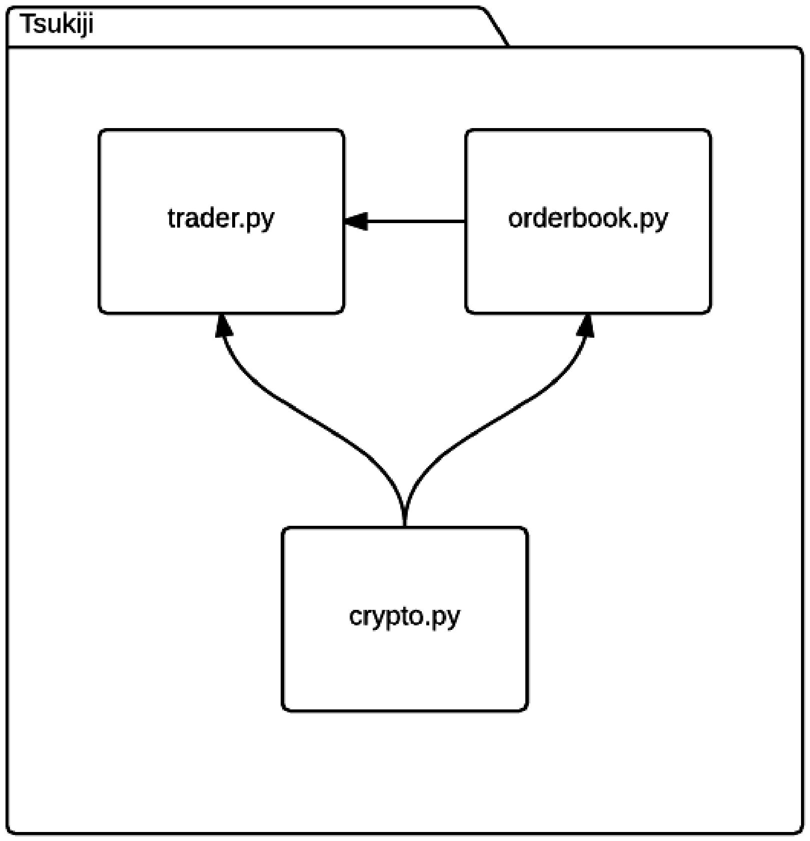


Figure 2: A diagram with all of Tsukiji's modules

3.1.1 Dependencies

Tsukiji has two main third-party libraries it relies on: Twisted [3] and PyCrypto [16]. Twisted is an open-source networking library. It enjoys wide-spread use in the python world. Tsukiji mainly makes use of its implementation of the UDP protocol. For more information, see section 4.2.1.

PyCrypto is a cryptography toolkit for python. It comes with many functions related to cryptography, but we are mostly interested in functions related to public-key cryptography. Each trader has a public/private key pair. A trader's public key acts as their identifier. For more information, see section 4.1.4.

4 Implementation

The development of Tsukiji followed aspects of the Scrum methodology. The time frame of the project was divided into sprints, each sprint lasting around a week. At the end of each sprint, the client was given a demo to discuss what should be developed in the next sprint.

4.1 Sprint 1: Sockets, broadcasting and protocol definition

In the first sprint, an early version of Tsukiji was implemented. The goal of the first sprint was to create a local simulation of the decentral market using socket connections. Messages between peers are spread via broadcasting. A basic version of the message protocol is defined. Problems encountered during this sprint are also discussed: broken pipes with disconnected sockets, identifiers, and cancel messages.

4.1.1 Broadcasting

Peers have to be able to communicate with each other by passing messages. One option considered for this was approaching a subset and relaying the information of your subset to other subsets. The issue with the subset approach was that, in a competitive market, someone that is selling an item is not easily motivated to advertise in the name of another seller. This could lead to certain peers, that are offering an item for sale, to not relay other offers that would endanger their own offers. This could lead to a disjoint of the network and therefor destroy communication between certain peers.

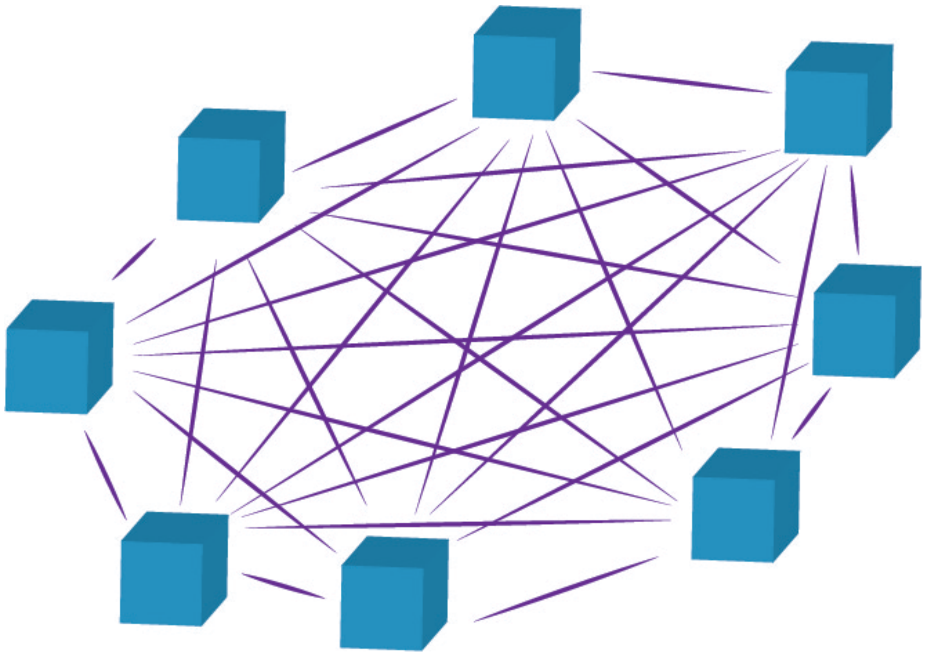


Figure 3: Broadcasting: The cubes represent peers and the lines represent connections between peers

Broadcasting does not have this issue. In broadcasting, every peer in the network is directly connected to each other, as depicted in figure 3. Everyone personally takes care of their own advertising and there is no peer between origin and destination that could disrupt the communication. The issue with broadcasting is that it scales rather poorly. The increase of data over the network increases exponentially as the amount of users rises. Because of this, it is not advised to use broadcasting in a large scale project. However, broadcasting suffices for the first iteration. Implementation is simple and it's easily replaced by a more complex system in later sprints.

4.1.2 Protocol

In this sprint, we also defined our application protocol. For a detailed specification, see A. Although we have a lot of experience in using protocols and APIs, designing a new protocol is another matter entirely. Our approach to designing the protocol was to keep it as simple as possible and make it easy to extend. Should we wish to increase functionality in the future, the protocol is easily adaptable.

4.1.3 Broken Pipes

Testing the socket connections showed a couple of issues. Whenever a client would disconnect from a peer, the server that it was connected to reported a Broken Pipe error. This did not stop the server from running, but it did add 2 empty string to the collection of received strings. However, if the server received an empty string that was sent intentionally, it ignores that message. Therefor it would not hinder the communication if the Broken Pipe errors are suppressed, since the program would not change behaviour. The empty strings that resulted from this error are now filtered out.

4.1.4 Identifiers

With the lack of a central authority confirming the identity of peers within the network, a user needs an identifier that distinguishes them from others. This id needs to be unique to the peer and needs to be constructed in such a way that it cannot be imitated by someone else. A regular username and password structure is not possible in a decentralised network since there is no entity that is trusted to store and confirm the passwords.

Simply using a username without a password is possible by broadcasting a request to receive all the usernames and allowing only nonexistent names to be used during creation. While this creates a unique identifier for the user, it does not create a safety measure against impostors. Anyone can declare that their username is Alice and can then validate transactions in the name of Alice, without the consent of the real Alice.

A solution to this problem lies in cryptography. We need a way for users to prove that they are authentic. This is possible by using asymmetric encryption. Peers can use their public key as a unique identifier, since the chances of a SHA256 collision happening are incredibly low. Users can now use the public key of the recipient of their message to force them to authenticate themselves. As long as everybody stays the sole owner of their private key, no one but themselves will be able to respond correctly to the authentication request. This enables a network where everyone can read every message, but can only answer to authentication requests when they are the original sender of the message.

4.1.5 Cancelling

One issue that arose while designing the protocol was how to cancel an offer. Our first proposal was to broadcast a special cancel message to all peers. However, there are two major problems with this approach: authentication and flooding.

At the time of writing, a public key is used as identifier. This means the same public key has to be used to send out the cancel message. A single person should be able to create and dispose of public/private key pairs at will. If, for whatever reason, the public/private key pair is lost, the offer cannot be cancelled.

Another problem is the spoofing of these cancel messages. Suppose a malicious third party sends out a cancel message. Other users in the network cannot act on this message without authenticating the sender. This would require three messages total. Suppose Alice wants to send a cancel message to Bob. The first is the cancel message from Alice to Bob. The second is Bob asking Alice for verification, e.g. by sending a random number encrypted by Alice's public key. The third message would be Alice sending Bob the answer, proving she can unencrypt the random number. This would quickly strain the network.

The solution to both of these problems is to not broadcast a cancel message at all. If someone requests a trade to a cancelled ask/bid, it is replied to with a cancel message. A timeout field is also added to every ask/bid.

Suppose Alice wants to cancel her ask/bid. Internally, she will mark her ask/bid as cancelled. If Bob requests a trade, Alice responds with a cancel message. If Bob wants to authenticate Alice, he can piggyback a verification request with the trade message. In the cancel message that Alice responds with, she will also verify her identity with her private key. This protocol only needs two messages, instead of three. This exchange only happens between interested parties, instead of the entire network. Lastly, once an offer is expired, Bob can no longer request a trade for that offer.

4.2 Sprint 2: Twisted and peer discovery

In the second sprint, a networking library called Twisted is integrated into tsukiji. Instead of hardcoding peers, a peer discovery mechanism is introduced. Finally, a gossip protocol replaces the broadcasting protocol.

4.2.1 Twisted

The networking protocol using raw sockets of sprint 1 showed how the idea of the implementation would work in practice. What we need now is a stable and well-tested implementation to further our project into scaling and to prepare it for the real deal. To save us the work required to produce such code, we chose to use a library instead.

Twisted [3] is a open source library that offers many options for networking and communication. Their site describes it as an event-based framework for internet applications. Using this framework, rather than our self-made code had a couple of benefits:

- Twisted is well-tested.
- Twisted is event-driven.
- Twisted has an implementation of the UDP protocol.
- Twisted is already implemented.

The current implementation of the broadcasting protocol of sprint 1 does not contain any tests so there is no guarantee that the code is bug-free and ready to be extended. Twisted on the other hand has numerous test cases that cover many use-cases. Because of this, a lot of time can be saved by trusting that the imported functions of Twisted will behave as promised instead of writing tests for our own implementation.

The current implementation uses a threaded approach to handling requests. Threads are not maintained particularly well and are an enormous strain on the processor. Twisted gives a structure that requires a low amount of recourses when it is not handling data for sending or receiving messages with an event-driven engine.

Torrenting networks generally use the UDP transport protocol to carry their data. The reason for this is that, compared to TCP, UDP traverses more easily over NAT and firewalls. Keeping in line with this, it would be useful to have a similar way of handling communication in extensions of these programs. Twisted provides options for both TCP and UDP.

All of the issues above could be made and added to the implementation made in the previous sprint. However, this would cost a lot of time that could be spent on solving other issues.

Given these advantages, Twisted seems like a great way to improve the current networking implementation and provide reliability and scalability.

4.2.2 Peer discovery

One issue of having a decentralised system is that there is no central authority that can provide information about all the users and all the data. Because of this, when a peer is not yet part of the network, it has no knowledge of who to contact for more information about the network and all the peers connected to it. This problem of peer discovery has not yet been solved in a truly decentralised way. The current solution is to bootstrap a user by giving them a peerlist. This peerlist contains a number of ”super peers” that should always be online. The new peer then connects with one random super peer. This super peer sends a list of other peers back to the new peer. Now the new peer knows about who is in the network. From now on, the exchanging of peer lists does not necessarily have to occur with a super peer, but can also occur with other, normal peers. With this implementation, the system is mostly decentralised, but it still requires a number of super peers to be online to support any initial connections of new users.

4.2.3 Gossip

The implementation of sprint 1 used the broadcasting protocol, where every peer is connected to every other peer. This protocol is easy to implement but does not scale well. When the network grows larger, every peer has more recipients to contact and there are more peers sending messages all around. To increase scalability, a better protocol is required.

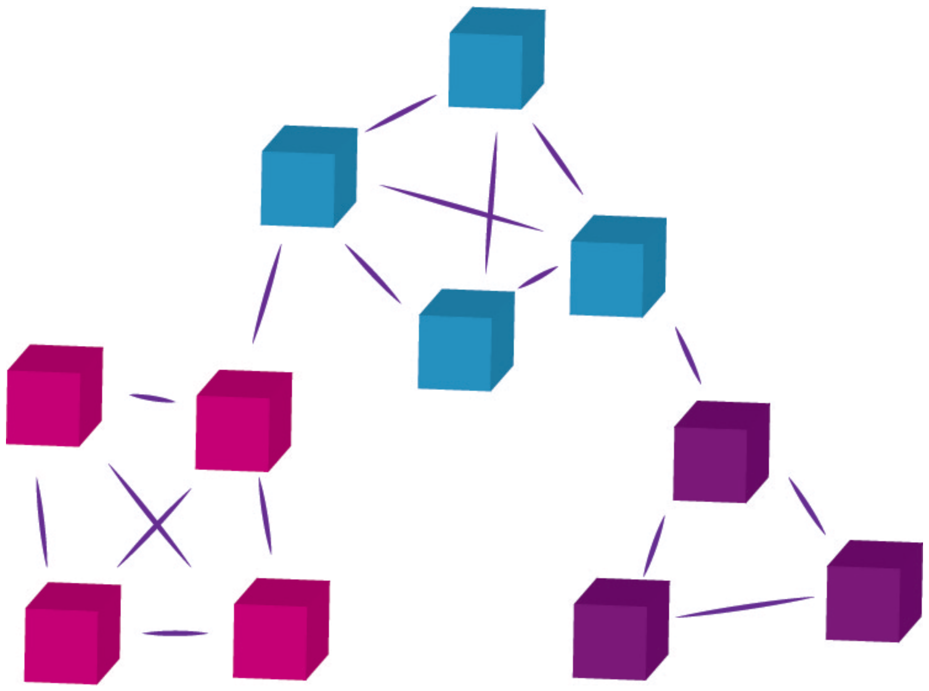


Figure 4: Gossip: The color of the cubes represent the groups of the peers and the lines represent connections between peers

The peer discovery discussed in the previous section inherently creates a situation where subgroups of peers are created that know of each other. A peer can be in multiple groups and they provide the connection between groups. The collection of groups connected to each other is called a clique. This property of the network is exactly what an epidemic or gossip protocol uses. In this type of protocol, a peer relays all the messages it receives to all peers in its group, which eventually spreads through the entire clique, as shown in figure 4. Since the peers in a group have randomly chosen who sends them their peerlist, every peer should have a slightly different group. Even in the case that every peer only encounters users within its own subgroup, the original master-peers can still act as a bridge between the groups.

Using the gossip protocol severely reduces network traffic compared to the broadcasting protocol. The addition of a new user now only increases the load of one group, rather than of the entire network. With the addition of many new peers, new groups will form to cut down the overall stress. This protocol has proven itself to be scalable [17].

4.3 Sprint 3: Testing at scale

The theme of the third sprint is testing. Testing is an important part of software development. It gives the developers certainty over the quality of their program. We felt like this sprint would be the perfect time to start testing. It is better to start testing as early as possible. We started relatively late compared to, for example, the Test Driven Development methodology, where tests are written even before the application logic is written. However, it has allowed us to quickly iterate on Tsukiji in the early sprints. At this stage, Tsukiji had reached a level of maturity where logic starts to become significantly more complex. Tests give us assurance our program works correctly for the test cases we have defined. There are two forms of testing focused on during this sprint: unit testing and scalibility testing. Another form of testing that isn't focused on during this sprint is integration testing. The purpose of integration testing is combine the components of a system and to test whether or not they produce the desired result. However, at this point tsukiji is still sufficiently small that we believe producing such tests would yield little benefit.

4.3.1 Unit testing

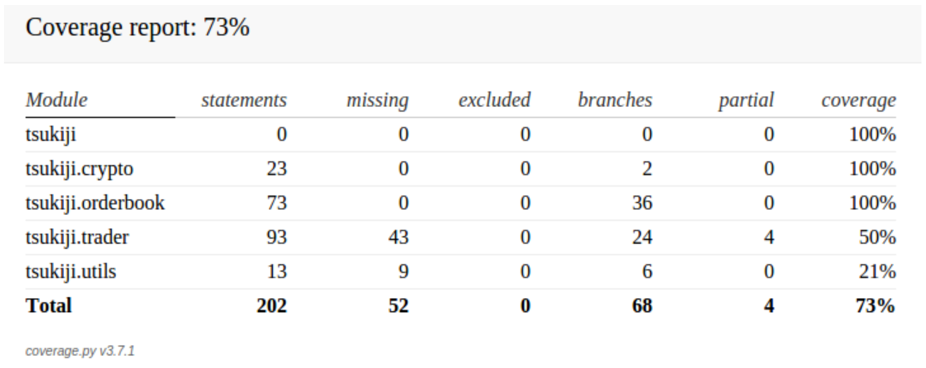
Unit testing is the testing of small units of code, ideally the smallest block of code possible. For our purposes, the smallest block of code is an individual function or class method. Each unit should be tested in isolation, i.e. each test case is independent of each other. For unit testing Tsukiji, we use the nosetests library [18]. nosetests is an extension of the default python unittest library. With nosetests, it is also possible to measure code coverage.

A large part of what makes testing easy or difficult is expressed in the testability of the code. How well is the code divided into sufficiently small enough functions, classes and modules? How large are these components, and how dependant are they upon each other? Many books, articles, and blog posts have been written on this subject. An example of such an article is Writing Testable Code [19], a guide used at Google. While this guide is written specifically for the Java programming language, its four main points still hold:

1. Constructor does Real Work
2. Digging into Collaborators
3. Brittle Global State & Singletons
4. Class Does Too Much

At this point, Tsukiji has two main modules: orderbook.py and trader.py. The orderbook module consists mainly of functions that concern offers and message creation.

From a unit testing standpoint, the orderbook module is very easy to test. The functions concerning offers are pure functions, i.e., they have no side-effects. The functions concerning message creation address a few global variables (e.g. the message counter), but there are only a few of such variables and their scope



Coverage report: 73%

Module	statements	missing	excluded	branches	partial	coverage
tsukiji	0	0	0	0	0	100%
tsukiji.crypto	23	0	0	2	0	100%
tsukiji.orderbook	73	0	0	36	0	100%
tsukiji.trader	93	43	0	24	4	50%
tsukiji.utils	13	9	0	6	0	21%
Total	**202**	**52**	**0**	**68**	**4**	**73%**

coverage.py v3.7.1

Figure 5: A table listing each module's coverage.

limited to one file. It could also be said that this module does too much and should be split up. This is definitely something to keep in mind for the future.

The trader module consists of a relatively large class called Trader. It implements the DatagramProtocol from the Twisted library. In essence, very little logic occurs here. Its main purpose is matching the right message with the right action on the orderbook. Using the right kind of mocks, this module is also relatively easy to test.

One way of measuring the effectiveness of your tests is by code coverage: how much of your code is run during your tests. nosetests provides both statement coverage and branch coverage. Statement coverage is the coverage of each line of code; 100% statement coverage means that every line of code is run at least once. Branch coverage is like statement coverage, but takes branches into account. For example, a boolean condition with one literal has two branches, True or False. A condition with two literals has four branches. See figure 5 for our coverage report.

4.3.2 Scalability testing

The other form of testing focused on during this sprint is large scale testing. The purpose of our large scale test is to test how functional the network remains with a large amount of peers. Of course, it is not very practical to run our program on hundreds of different devices. Our solution is to run a simulation locally. It takes a list of orders and has different peers post those orders in the network. The tests show that the network could handle thousands of users and that duplicate messages are correctly filtered out.

4.4 Sprint 4: Using real money

In this sprint, we will focus on our final feature. The last major challenge remaining at this point was linking the virtual marketplace to the real world. So far, users can trade their digital commodities for any amount of virtual currency. The problem was that this currency had no actual value, since the only way of obtaining it was by trading it for reputation points. So the marketplace at this point would consist of people selling reputation points and others that do not have the means to acquire their own points, trying to buy them. However, without any value attached to this virtual currency, no incentive is given to start trading. The solution to this problem is to allow users to use real money to either buy virtual currency or use that money directly in the transaction.

4.4.1 Usage of virtual currency

Virtual currency is a great way to keep the identities of users hidden when making financial transactions. The problem with this structure is that it requires a way to verify whether the digital money is actually real and not spoofed. Without a central authority that governs all transactions, it becomes very difficult to tell whether or not the digital money is legitimate. Of course, there are solutions.

For example, one could implement a block-chain structure like BitCoin has made [20]. With this, the whole user-base has to agree on a certain chain of transactions. To achieve this, it is required to have every user communicate with every other user what transactions are made. This method of regular broadcasting does not scale and is the reason Tsukiji switched to a gossip model.

The solution for Tsukiji was to abandon the idea of virtual currency altogether. If users can use existing payment options, we no longer have to worry about security en legitimacy issues that payments bring. All that is required of Tsukiji is to link one peer to another peer and facilitate an easy way to perform the transaction. The privacy issue is now brought back to the user. If they desire anonymous transactions, they will have to ensure this themselves with their preferred payment option.

4.4.2 Online banking

In order to use real money, we required a third party to handle payments for us. After some research, two options had our preference: iDeal and Paypal. Sadly, the iDeal API had the constraint that it only allowed users to give money to businesses and did not facilitate payments from person to person. This would mean that every user has to create a business account and have it approved by iDeal, just to make payments in Tsukiji. As this is not deemed feasible, we chose to use Paypal instead.

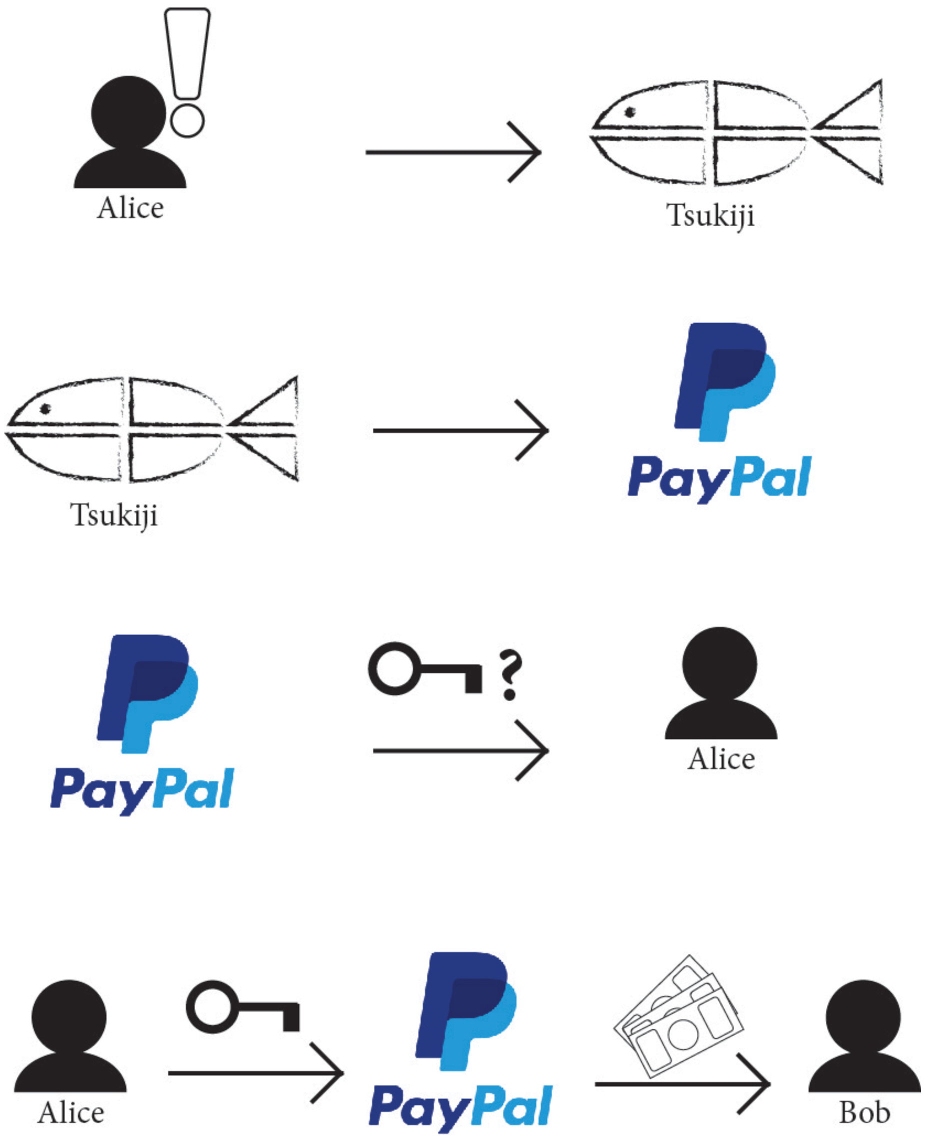


Figure 6: Alice finds an interesting item on Tsukiji. Tsukiji then aks the PayPal API to create an authorisation page. This page asks Alice for her login details and then verifies the transaction, sending the money from Alice's paypal account to Bob's.

The Paypal API [21] offers many ways to create, send and verify transactions. While poorly documented, there is a python integration suite that uses http calls with JSON objects to communicate with the Paypal servers. The current implementation in Tsukiji creates a payment page when a trade is made. The page has the amount (in euros) and the recipient pre-filled. All the user has to do now is login on the Paypal website and authorise the transaction.

The full process is depicted in Figure 6. This way the user does not need to trust Tsukiji with their Paypal credentials and does not even need to know the address of the person they are about to pay. This approach keeps Tsukiji decentralised as well, since the Paypal service is not part of the implementation of Tsukiji itself. It should still be possible to equip Tsukiji with different payment options.

Tsukiji now has a way for users to pay each other that is reliable but leaves itself unaccountable for any errors in transactions. Users only have to trust Paypal for their transactions, which complies with the zero-trust ideology of a decentralised system.

4.5 Sprint 5: Incremental Payments

With the payment structure in place, it was time to look at ways to increase the trust of Tsukiji's users. Currently when a buyer sends money to a seller, there is no guarantee that any product is shipped the buyer.

This is a problem, especially with large transactions, since more money would be lost if nothing is obtained in return. The solution BitCoin offers is a multisigned deposit, that requires multiple parties to sign for the release of the money. Sadly, no such system is available within Paypal itself.

One way to dramatically reduce any trust issues involved with large payments is by dividing them into many small payments. This way, if a seller does not deliver on their promise, the buyer has only lost a relatively small amount of money. Splitting transactions evenly would result in a drastic increase of data flow. In order to reduce this, a system of building trust can be implemented. This system is called Incremental Payments. After every successful transaction, the buyer trusts the seller more. The buyer increases the amount of money they send to the seller.

It is worth noting that such a system is hard to implement with physical goods, since not every product can be split up in parts. The original idea behind Tsukiji was to be able to buy and sell reputation points for BitTorrent groups. Digital value can easily be divided up, so it is not a problem in this scenario.

Due to time constraints, we have not been able to finish our implementation of incremental payments. Our goal was to completely automate this process. All the user has to do is approve of the trade. The design of a possible implementation of Tsukiji involved using Selenium [22] to simulate a login to PayPal and authorize a payment automatically. Selenium is a piece of software that creates a dummy browser and lets you automate the actions performed, such as filling in textboxes and pressing buttons. This way, every incremental payment can be made without the user having to authorize every individual payment, but only give its login details to Tsukiji once.

The problem lied at the receiving end of the payment. The seller required a way to know that he had received a payment. The temporary naive implementation would be to constantly poll PayPal whether any new transactions have been received. But even this would require a way to identify every payment and link it to a specific chain of transactions. This would become increasingly difficult when multiple transaction chains are made with one seller, endangering our scalability.

These issues, combined with the lack of time to create a proper solution, led to an unfinished implementation of Incremental Payments. It would be very interesting to have this system implemented at a later stage since no other project uses a similar system with non-digital currency.

5 Evaluation

This section evaluates the project's development process and end product. Tsukiji itself will be analysed by how many requirements have been fulfilled and by its code quality. An overview of known issues will be given at the end.

5.1 Development process

The planning was to spend 2 weeks researching the given problem, then spend around 6 weeks developing a solution, while documenting the decisions made during the development. This structure was held up throughout the lifespan of the project. Documentation began lacking halfway through the project, but this was quickly corrected to make sure every decision was written down as accurately as possible.

Due to the agile development methodology, it was difficult to plan every sprint ahead of time. Requirements could change and the focus of the project could diverge from the decisions made in the first meeting with the client. Regardless, the agile development methodology proved to be very useful. Exactly because of the volatile nature of the project, creating additions to the software on a week to week basis fit the project perfectly. Decisions made along the way, such as ignoring the possibility of people cheating the system, greatly simplified protocol design without losing time on pre-emptively implementing a structure for such a protocol.

A large benefit to the efficiency of the development of Tsukiji was that the team consisted out of two developers. The advantage this brings is that the team could always work together in the same room. It became very easy to discuss issues and make decisions, such as whether to use a certain library or protocol. The developers sat next to each other which eliminated any overhead or inaccuracies in communication typical in technologies such as e-mail or instant messaging. The load of work was divided such that both members were responsible for a certain feature of the software. Whenever a member finished their implementation for the current sprint, they immediately documented what they had written.

5.2 Fulfillment of requirements

Section 2.2 discusses the requirements of Tsukiji. Important in the evaluation of a project is whether it complies with the requirements given by the client. This section will follow the MoSCoW list given in section 2.2, and evaluate each item.

Must have	
Place an offer	Done
Respond to an offer	Done
Decentralised	Done
Peer discovery	Done
Scalable to thousands of users	Done
Command line input	Done
Should have	
Trade using real money	Done
Could have	
Nice user interface	Not done
Would have	
Privacy	Partially done
Protection against hostile attacks	Not Done
Anti-Spoofing	Partially done

Table 1: Overview of what requirements are completed

5.2.1 Must Have(s)

Every 'Must have'-requirement has been fulfilled by Tsukiji.

- Place an offer

  Users are able to place offers and respond to offers.

- Respond to an offer, facilitating a trade

  The response is currently handled by a trading engine that matches the asks with bids, given the same quantity and value.

- Decentralised

  Apart from a peerlist that is distributed with the software, the system is completely decentralised. Passing on messages uses the gossip protocol. Authentication is done through public and private keys. Neither of these use a central point.

- Peer discovery

  Peer discovery is achieved by having peers exchange information about each other. Initially, this is bootstrapped with a set of super peers known to be well connected and online. These super peers can share information about the network. After being bootstrapped, peer discovery is achieved by exchanging information with other peers in the network.

- Scalable to thousands of users

  The scalability is tested with a simulation on a single computer. This test showed that messages can easily be handled in the range of thousands of users and that duplicate messages are not passed along when received. More on this can be found in section 4.3.2

- Command line input

  The entire application is controlled via a linux terminal, so the requirement of control through command line input has been fulfilled as well.

5.2.2 Should Have(s)

- Trade using real money

  Users are able to spend real money (currently any currency supported by PayPal) on the goods traded in Tsukiji. The currency used by the software should not impact the program's ability to make trades

5.2.3 Could Have(s)

The current implementation does not fulfill the 'Could have' requirement.

- Nice user interface

  Implementing a User Interface takes a lot of time without increasing any functionality. Since Tsukiji is a proof of concept, we wanted to create as much functionality as possible in the time given. This lead to the decision to focus on other features than a GUI. Interaction with the program occurs through a terminal. However, we still recognize a good GUI as a huge asset to increasing usage of Tsukiji.

5.2.4 Would Have(s)

The current implementation does not fulfil any of the 'Would Have' requirements. The 'Would have'-requirements are mainly security issues. These problems can certainly be solved, but it would require a lot of time to achieve something that is outside the scope of this project. Tsukiji was created to show that it is possible to create a marketplace without a central authority, it was not created to protect users of an online marketplace from fraud. While this is certainly a requirement for a functional program to be used by many users, it is still a solved problem. This issue could be addressed in any future extensions of the project.

- Privacy

  The privacy requirement is partly fulfilled. Users do not have an identifier bound directly to their real identity. They have their SHA256 key sent across the network. The offers created can be tracked down to the hash, and to the IP that sent the message. A user could spoof their IP whenever they are creating trades to stay hidden, but this is their own responsibility. Besides that, a possible privacy breach could be a personal PayPal account. To avoid using one's real account, it is possible to create a new PayPal account on a fake email address and use that to make transactions. The bank accounts linked to PayPal are not visible in transactions, so the trades will not give away a users identity.

- Protection against hostile attacks

  It is currently unknown how resistant the network is against a hostile attack. It is certainly possible that the network may fall against a sufficiently motivated attacker.

- Anti-spoofing

  It is currently possible to spoof messages. Messages are currently not encrypted. The only authentication present in a message is the public key of a user. But since this key is public, it can easily be replicated by a malicious user, which can only be discovered when the user with ill intent is asked to identify itself with the private key. This does not stop people from creating fake transactions that will make it seem like they have more items to sell than they in reality have.

It is clear that all the crucial requirements have been met. Tsukiji sufficiently fulfils what the client had asked for.

5.3 Known issues

The current implementation of Tsukiji has a couple of issues that have not been addressed. Section 5.2 already touched on a couple of missing features such as perfect anonymity and security. There is another major factor that is problematic for the current state of the protocol.

Currently, a user can lie about his transactions. It is possible to pretend to have a certain amount of points for sale, while not actually having any at all. This is because Alice has no way to check the transactions Bob has made. Alice would only be able to see that a certain bid or ask is no longer in the list of offers. If Bob spoofed a message that says that a certain transaction has been made, Alice has no way to verify this. This creates a trust issue that is currently not resolved, so the system has to assume full trust from all parties. Section 7 provides a possible solution for this problem.

Another known issue of Tsukiji is that it is risky to make a large transaction, since incremental payments (see section 7.5) have not been implemented. Whenever a user sends someone else money to buy goods, there is no way to force the other party to actually send the goods. This creates a situation where only small amounts of goods will be traded at once and will quickly clutter the marketplace with many small offers.

We believe it to be feasible to solve these issues, but were unable to within the time frame given for this project. It would be interesting to extend Tsukiji to support solutions for these issues. These are the first steps towards transforming this proof of concept to be used a real piece of software.

5.4 Code quality

A desirable goal of Tsukiji is to keep the code of high quality. Testing is an obvious way to maintain a certain quality. To read more on our testing process, see section 4.3. Aside from our own testing, the code was also sent to the Software Improvement Group at two points during development. Appendix B contains their analysis of Tsukiji's code base. This section will cover our response and how we handled the feedback.

Overall, we are satisfied with the quality of our code. We believe our code to be easy to read. This is partly due to the fact that our code base is not very large. The rating SIG assigned to Tsukiji supports our opinion.

5.4.1 Response to first round of SIG feedback

Three points of attention are given in the first feedback: component balance, unit interfacing, and testing.

We agree on the points regarding component balance. At the time, the code was split up into three main files: orderbook.py, crypto.py, and udpreceive.py. While the first two files are fairly self-explanatory, udpreceive is fairly ambiguous. There were also some extraneous components (e.g. udpsend) that were not core to the system, but possibly confusing to newcomers. In response to this feedback, udpreceive.py was renamed to trader.py. This more accurately describes what this file stands for. If you invoke trader.py, you start up a new trader server. However, we find it difficult to split the remaining parts up into more components.

The second point given talked about unit interfacing. In this case, we believe SIG's feedback to be correct, but their solution undesirable. It is true that multiple functions contain many arguments. A lot of these shared common functionality, so we have abstracted these further into a main function. However, SIG's solution is to introduce more classes. But rather than using classes, we use a lot of pure functions that return python dictionaries, and good variable names. These can be seen as a replacement for classes. There's a great talk from pycon 2012 called "Stop writing classes" [23]. We believe the principles given in this talk to apply to our situation here.

The final piece of feedback given talked about testing. We agree completely with this feedback and have since then implemented our own test suite. The process of creating this test suite is described in section 4.3.

5.4.2 Response to second round of SIG feedback

The second round of feedback largely matches our response. We have not adjusted unit interfacing because of the reasons explained previously. We have not adjusted component balance in a major way. We have added a lot of test code, which is acknowledged. For the future course of this project, we will continue to keep a closer look at testing and component balance. We do not believe unit interfacing poses a problem currently, but it is of course possible that it might in the future. We will continue to monitor throughout the project's lifespan.

6 Conclusion

The goal of this project was to create a proof of concept for a decentralised market where user could trade commodities for real money without accessing a central server and with no governing authority. Tsukiji manages to show that this concept is definitely possible. It can connect users to a subset of the network that communicates with other groups of users via the gossip protocol. Using the PayPal API, the responsibility of handling payments and real bank transfers is taken away from Tsukiji. That API also enables anonymity within Tsukiji, since users can only see a paypal address, where any other private information is hidden. This will not stop governments from finding users within the network, but it will protect users from other malicious attackers trying to discover their identity.

Tsukiji offers a reliable, well-tested, and scalable solution to trading items online, without the weakness of a central point and without requiring any personal information.

7 Recommendations

The goal of this project is to create a proof of concept in a limited amount of time. Because of this, there are multiple possible features that have been put aside over in favour of other features deemed more essential. In this chapter we suggest some recommendations for further progress on the project. They can be added to this proof of concept to build Tsukiji into a fully fledged user program, or to further the research of possibilities of a decentral market.

- **Finishing Incremental Payment**

Section 4.5 covers the concept of incremental payments. One of the issues discussed was identifying which payment on Paypal corresponds with which trade on Tsukiji. One possible solution would be sending messages over the network of Tsukiji to the seller that a payment has been made. This message would contain a chain-id and an index of the transaction of in that chain. These id's would then have to be added a notification in the PayPal transaction and be extracted when the seller reads its transactionlist.

- **Graphical User Interface**

A big improvement to Tsukiji would be to add a graphical user interface. This would make it much easier to use the program and understand what is going on. Graphs and lists with variable ordering can vastly improve the format in which information is displayed. Additionally, with a clear interface, Tsukiji would be far more inviting to the average user than a command-line tool. Command line input brings multiple challenges, such as requiring the user to know the right commands and the syntax of those instructions. A GUI can represent those instructions with a more intuitive interface.

- **Anti-spoofing**

Section 5.3 covered the issue of people referencing false transactions. They are pretending to have more tradable goods that they in reality have. A possible solution is to implement a block chain like BitCoin has done. Their structure requires the whole userbase to agree on a set of transactions made. These trades are hashed and used to determine whether the following set of transactions are real. For a more in-depth explanation of the block-chain verification, we refer to the BitCoin paper [20].

- **Trading large quantities**

The current implementation of Tsukiji only supports trading a single unit of an item. It is very possible that users want to trade a larger quantity at a time. This will also reduce the amount of offers needed to perform a certain trade, decreasing clutter in the list of open offers.

References

[1] Openmarket store example. https://camo.githubusercontent.com/7ab83633ddfd9a08a09bd36f3c73beb0dcf7e35e/687474703a2f2f692e696d6775722e636f6d2f6477717861614c2e706e67.

[2] Tribler: Privacy using our tor-inspired onion routing. http://www.tribler.org/.

[3] Twisted. https://twistedmatrix.com.

[4] How mt. gox went down. http://money.cnn.com/2014/02/25/technology/security/bitcoin-mtgox/.

[5] D Levin, K LaCurts, N Spring, and B Bhattacharjee. Bittorrent is an auction. In *Proc. of SIGCOMM*, 2008.

[6] Michel Meulpolder, Johan Pouwelse, Dick Epema, and Henk Sips. Bartercast: Fully distributed sharing-ratio enforcement in bittorrent. *Delft University of Technology-Parallel and Distributed Systems Report Series*, 2008.

[7] Swiss federal institute of intellectual property. https://www.ige.ch/en/ip4all/legal-info/legal-areas/copyright.html.

[8] Switzerland continues work on changes to online copyright rules. http://www.ip-watch.org/2014/08/06/switzerland-continues-work-on-changes-to-online-copyright-rules/.

[9] Limit order book in python. https://github.com/ab24v07/PyLOB.

[10] Bitmarkets. https://voluntary.net/bitmarkets/.

[11] Bitmessage. https://bitmessage.org/wiki/Main_Page.

[12] Tor. https://www.torproject.org.

[13] Openmarket. https://github.com/openmarket/openmarket/.

[14] Openbazaar. https://www.https://openbazaar.org/.

[15] Dispute resolution in openbazaar. https://gist.github.com/drwasho/405d51bd1b1a32e38145.

[16] Pycrypto. https://www.dlitz.net/software/pycrypto/.

[17] Spyros Voulgaris, Márk Jelasity, and Maarten Van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Agents and Peer-to-Peer Computing*, pages 47–58. Springer, 2005.

[18] nosetests. https://nose.readthedocs.org/.

[19] Miško Hevery Jonathan Wolter, Russ Ruffer. Guide: Writing testable code. http://misko.hevery.com/code-reviewers-guide/.

[20] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

[21] Paypal api. `https://developer.paypal.com/docs/classic/`.

[22] Selenium. `http://www.seleniumhq.org/`.

[23] Jack Diederich. Stop writing classes. `http://pyvideo.org/video/880/stop-writing-classes`.

Appendix A Trade Protocol

Tsukiji uses a custom application protocol to communicate between peers. Traders communicate to each other with messages encoded as JSON objects. The choice for JSON objects was made because it has a near-equivalent data type in python called a dictionary. It is also a human-readable format. Every message has the following additional attributes:

- `id`: Public key of the trader sending the message.
- `message-id`: Incremental counter unique across all messages with the same id.
- `timestamp`: Time of creation of this message.
- `type`: The type of message.

The combination of the `id` and `message-id` attributes uniquely identifies each message.

There are 7 message types in total: `ask`, `bid`, `trade`, `confirm`, `cancel`, `greeting`, and `greeting_response`. Each type of message is used in a different scenario and holds different attributes. In the following sections, these scenario's are laid out.

A.0.3 Placing an offer

In order to place an ask or bid, a message is sent where the type is `ask` or `bid` accordingly. These types of messages have the following additional attributes:

- `price`: The price per single unit.
- `quantity`: The amount of items requested to trade.
- `timeout`: The time of expiration for this offer.

This type of message is typically sent to the entire network. For the purpose of this application, the units of `price` and `quantity` are left undefined. In the future, the protocol could be updated with a currency string, such as 'EUR' for euros, 'USD' for dollars, or 'BTC' for bitcoin. Something similar could be adopted for the `quantity`, for example, 'kg' for weight, 'm3' for volume, or 'MBit' for data.

A.0.4 Requesting a trade

When a trader sees an offer they are interested in, they might respond to such an offer with a trade message. The type of this message is `trade`. This type of message has the following additional attributes:

- `recipient`: The id of the trader who created the offer.
- `quantity`: The quantity requested to be traded.
- `trade-id`: The message-id of the original offer.

The combination of `recipient` and `trade-id` is used to uniquely identify the specific offer for which the trade is requested. The `quantity` field may be equal to or less than the quantity stated in the original offer. Perhaps a trader is interested in an offer, but only can or needs to trade a certain amount that is less than this.

A.0.5 Accepting a trade

When a trade request arrives, a trader can either accept or cancel such a request. In the event that the trader accepts the trade request, they send a confirm message back. The type of this message is `confirm`. This type of message has the following additional attributes:

- `trade-id`: the message-id of the original offer.

This time, there's no need to add a recipient attribute because the combination of the `id` and `trade-id` fields uniquely identifies the offer to be traded. The offer is removed from the internal orderbook of both parties.

A.0.6 Cancelling an offer

When a trader decides to cancel one of its offers, e.g., because they want to lower their prices, Tsukiji will respond to any incoming trade requests with a cancel message. For a more detailed discussion on why this is implemented in this manner, see section 4.1.5. The type of this message is `cancel`. This type of message has the following additional attributes:

- `trade-id`: the message-id of the original offer.

Similar to a confirm message, there's no need to add a recipient attribute because the combination of the `id` and `trade-id` fields uniquely identifies the offer to be traded. The offer is removed from the internal orderbook of both parties.

A.0.7 Peer discovery

In order to facilitate peer discovery, there needs to be some mechanism for peers to exchange knowledge of other peers. For more information, see sections 4.2.2 and 4.2.3.

For a peer to request a list of peers from somebody else, they send a greeting message. This message has type `greeting`. No additional attributes are necessary.

When a trader receives a `greeting` message, they respond with a greeting_response. This message has type `greeting_response`. This type of message has the following additional attributes:

- `peerlist`: a list of peers

When a peer receives such a list of peers, they can add it to their own list of peers. Now the peers they know about has grown with more traders to directly communicate with.

A.0.8 Example of a trade

In order to get a better understanding of the protocol, we will describe an example of a trade. See figure 7 for a graphical depiction of the exchange. Alice wants to buy 3 items at price 10. She sends out an `ask` message to the world. Bob sees this message and is interested. He sends a trade request. Alice, who has received no other trade request up until that point, agrees to the trade and returns a confirm message.

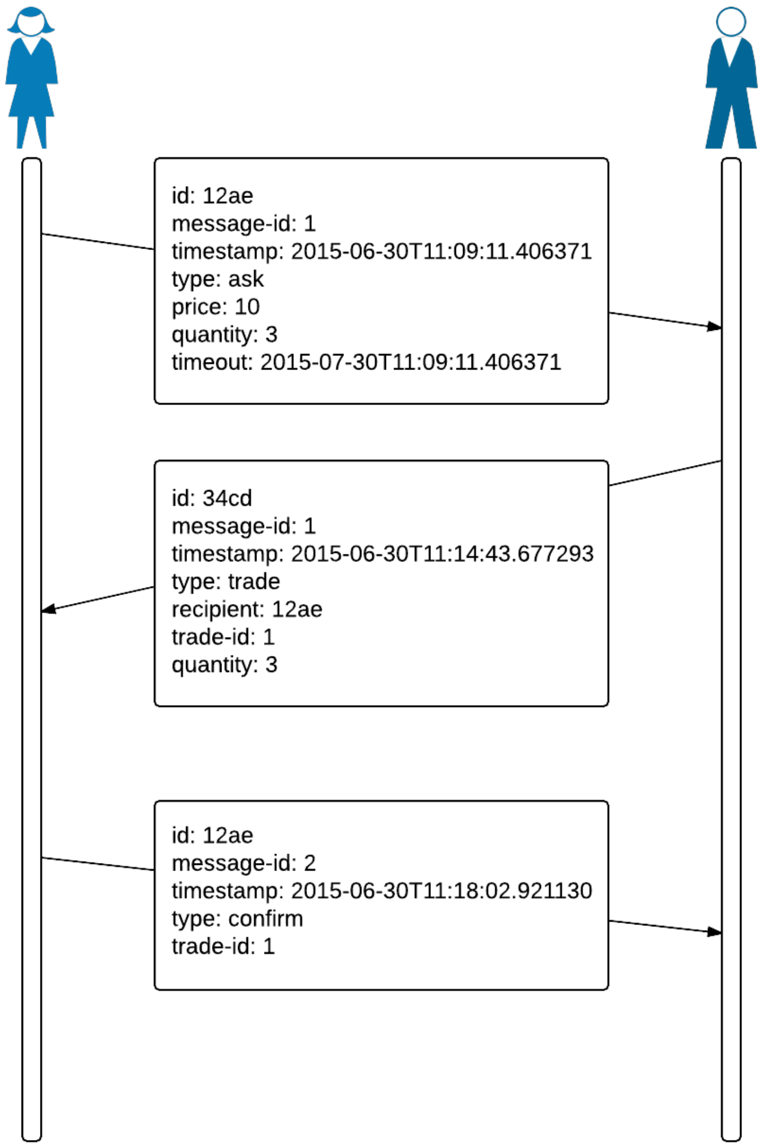


Figure 7: Example of a trade. Alice sends out an offer. Bob replies with a trade request. Alice agrees with the request and replies with a confirm message.

Appendix B SIG feedback

This appendix describes the feedback received from the Software Improvement Group (SIG). On the 26th of May 2015, project code was sent to SIG for feedback. Feedback on our code was received on the 29th of May. After processing this feedback, a second code submission was sent on June 9th. The second feedback response was received on June 17th.

B.1 First feedback (Dutch)

De code van het systeem scoort bijna vier sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Component Balance en Unit Interfacing.

Wat opvalt bij het bekijken van de code is dat er geen duidelijke componentenstructuur zichtbaar is op het file-systeem, terwijl het er wel op lijkt dat sommige files bij elkaar horen (bijvoorbeeld de twee 'udp' files). Dit maakt het voor een ontwikkelaar in eerste instantie lastiger om een algemeen beeld te krijgen van de functionaliteit die het systeem aanbied. Wij raden aan om kritisch te overwegen om de code in verschillende (functionele) componenten op te delen om zo een eerste indruk te geven van de high-level structuur van het systeem.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Binnen dit systeem wordt er op meerderee plaatsen een combinatie van 'price' en 'quantity' meegegeven, iets wat samen het concept 'bid' lijkt te vormen. Ook zien we op meerdere plekken een 'host' / 'port' combinatie, deze vormen samen het concept 'peer'. Om bij toekomstige aanpassingen duidelijker te maken wat er precies meegegeven moet worden aan deze methodes en om de code duidelijker te maken is het aan te raden een specifiek type te introduceren voor deze concepten.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

B.2 Second feedback (Dutch)

In de tweede upload zien we dat zowel het codevolume als de score voor onderhoudbaarheid ongeveer gelijk zijn gebleven.

Bij Unit Interfacing zien we dat jullie een aantal methodes hebben aangepast, maar deze verbeteringen worden grotendeels weer ongedaan gemaakt door de introductie van nieuwe methodes met 3-4 parameters. Dat lijkt niet zo veel, maar als je steeds het lijstje (price, quantity, timeout) tussen methodes moet doorgeven is dat een teken dat er iets in de abstractie niet helemaal lekker zit.

Bij Component Balance is onze eerste opmerking uit de analyse van de eerste upload nog steeds van kracht.

Jullie hebben onze aanbeveling om unit tests te gaan schrijven opgevolgd. De hoeveelheid testcode die jullie sinds de eerste upload hebben geproduceerd is ook aanzienlijk, complimenten. Op lange termijn zal dit zowel de onderhoudbaarheid als de stabiliteit van je systeem verbeteren.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie grotendeels zijn meegenomen in het ontwikkeltraject.

Appendix C Original Project Description

This project will create an electronic market place that is immune to shutdown by governments, immune to lawyer-based attacks or other real-world threats. The proof-of-concept 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-Euro trading, only use a single marketmaker, and offer no anonymity or DDoS protection. Bitcoin shows 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.

Appendix D Infosheet

Project Title
Decentral market

Client organisation
TU Delft

Date of final presentation
09-07-2015

Description
The goal was to create a proof of concept for a decentralised market where users can trade commodities for real money, with no central point of failure or governing authority. Few similar applications exists, those that do focus on BitCoin as a currency. Development occurred in sprints of one week, with the team working daily side by side to tackle the project. The end result is Tsukiji, an application where users can trade goods in a decentralised manner using real money. As a proof of concept, there are still many issues regarding trust, privacy, and user experience to be solved in future releases.

Members of the Project Team

Michael The is working on his Bachelor Computer Science at the TU Delft. In his spare time he plays too many video games. He is also active as a freelance python programmer for various companies. Michael is mostly responsible for designing the protocol and testing the application.

Hugo Reinbergen spends most of his spare time organising events for the Delftsche Studenten Bond, gaming, creating video effects and researching anonymity and system security. Hugo is responsible for the cryptography, the gossip implementation and the PayPal interaction.

Both team members contributed to writing the report and preparing the final presentation.

Client
Johan Pouwelse, TU Delft
Coach
Raynor Vliegendhart, TU Delft
Contact Person
Johan Pouwelse (J.A.Pouwelse@tudelft.nl)
Michael The (mcgthe@gmail.com)

The final report for this project can be found at: `http://repository.tudelft.nl`