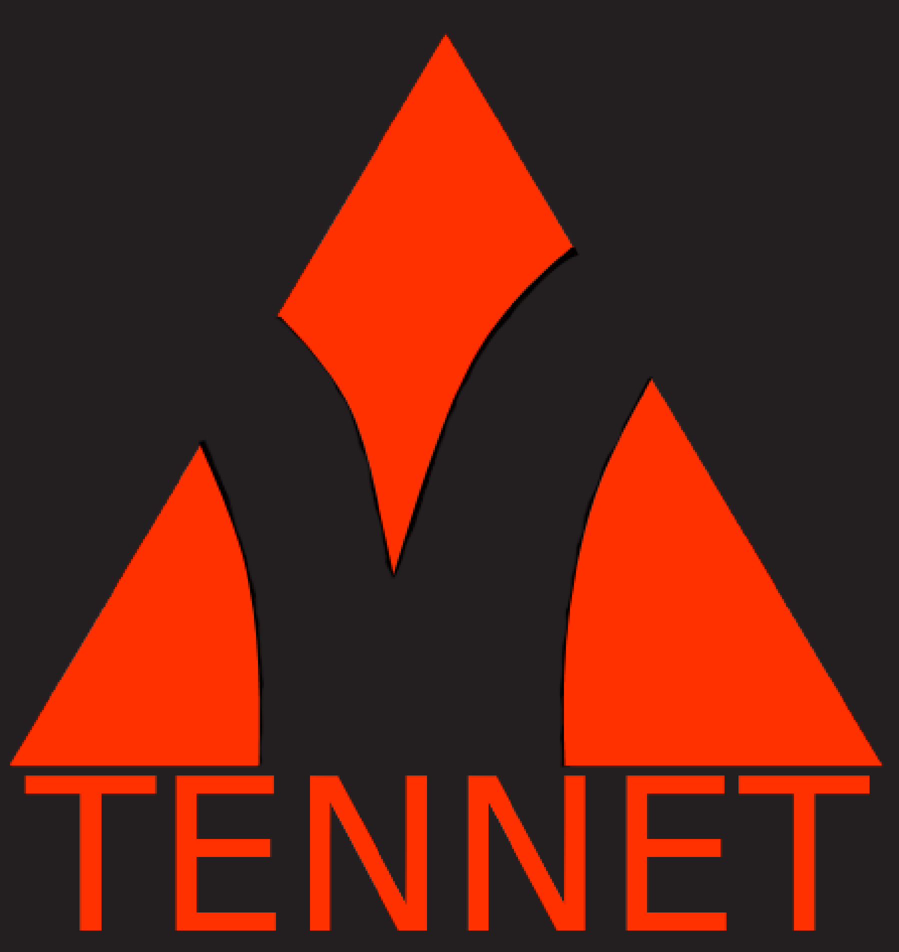
Autonomous Self-replicating Code

Bachelor Thesis

N.C.Bakker
R.v.d.Berg
S.A.Boodt

How To:
Allow a program to buy more systems using bitcoin, mutate and multiply



Autonomous Self-replicating Code

Bachelor Thesis

by

N.C.Bakker
R.v.d.Berg
S.A.Boodt

In partial fullfilment of the requirements to obtain the degree of

Bachelor of Science
in Computer Science at the Delft University of Technology,

to be defended publicly on Friday June 24, 2016 at 13:15 AM.

Student numbers:	4161394, 4319427, 4322258	
Project duration:	April 18, 2016 – June 24, 2016	
Thesis committee:	Ir. E. Bouman	Tribler.org, client
	Dr. J. Pouwelse	TU Delft, supervisor
	Otto Visser	TU Delft, Bachelor Project Commitee

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

TENNET was developed as a part of BEP project at the TU Delft, it was developed for Tribler to function as an experimental basis for a decentralized anonymous Tribler exit node network. It was developed in the spring of 2016.

We would like to acknowledge the people that helped and inspired us during our bachelor project. First we would like to thank dr.Ir. Johan Pouwelse and Ir. Egbert Bouman for their mentoring and guidance during our bachelor project. Our thanks also go out to the decentralized market Bachelor Project group the discussions with whom helped to inspired us to solutions we might not have tried otherwise. Lastly we would like to thank the Tribler team for helping out with the questions that arose during the integration of their program into our agent.

N.C.Bakker
R.v.d.Berg
S.A.Boodt
Delft, June 2016

Summary

During the project TENNET (the Tribler Exit Node NETwork) was constructed. A decentralized autonomously functioning agent that is capable of earning money by providing a Tribler exit node service.

A Tribler exit node is a service used by Tribler (an anonymous file sharing platform) to be backwards compatible with torrents.

TENNET will run this exit node on a server. It will then sell (what comes down to) Tribler upload capacity in exchange for Bitcoin. It will then use the gained bitcoin to buy a new VPS (Virtual Private Server) online, that it will install itself onto. From there, the cycle will continue, potentially resulting in a distributed network of Tribler exit nodes.

During the development several prototypes and tests were constructed.

- First prototype: buying a server and installing a new bitcoin wallet on this server fully automated.
- Long term exit node test: running an exit node on multiple servers to see how they perform on different servers.
- Second prototype: using a genetic algorithm, buy a server from the selection of VPS providers and install the agent on this together with a bitcoin wallet and Tribler exit node, then start all these services.
- Third prototype: This was the finished product, capable of replicating and making money on it's own as well as the functionality of the second prototype

Contents

Summary **v**

1 Introduction **1**
1.1 Structure 1

2 Initial System Design **3**
2.1 The agent life cycle. 3
2.2 Earning multichain for sustenance. 4
2.3 The process of buying and selling 4
2.4 Reproductive Strategy 5
2.5 Project Requirements 6
2.5.1 Must haves 6
2.5.2 Should haves 7
2.5.3 Could haves 7
2.5.4 Won’t haves. 7

3 Procurement, Automated Installation and Initialization **9**
3.1 Bitcoin wallet 9
3.2 Buying a VPS 9
3.2.1 Finding suitable VPS’ for our project 10
3.2.2 Possible ways of implementation 11
3.2.3 Automatically ordering a VPS using Selenium 11
3.3 Automated installation procedure 16

4 Extended Procurement And Genetic Modifications **17**
4.1 Addition of implemented VPS providers. 17
4.1.1 Benefits of alternatives 17
4.1.2 Retrieving the country the current machine is in using IP address. 18
4.1.3 Using temp-mail for temporary email addresses 18
4.2 Running Selenium Headlessly 19
4.2.1 Using Xvfb to emulate a display 19
4.3 Running the exit node 19
4.3.1 multichain 19
4.3.2 decentralized market and API negotations 20
4.4 Genetic modification 20

5 Live Trials **23**
5.1 First prototype 23
5.2 Running a ”long term” exit-node 23
5.3 Second prototype. 25
5.4 Third Prototype, a.k.a. ”full product” 26

6 Testing and SIG **27**
6.1 Testing approach 27
6.2 First SIG feedback 27
6.3 Reaction to SIG feedback 27
6.4 second SIG feedback 27

7 Conclusions **29**
7.1 Challenges 29
7.1.1 Implementation of the Decentral Market. 29
7.1.2 Tribler exit node 29
7.1.3 Autonomously buying VPS. 30

7.2 Unreliability of the system . . . 30
7.3 Recommendations . . . 30

Bibliography **33**

A Appendixes **35**
A.1 Infosheet . . . 35
A.2 1st SIG Feedback . . . 36
A.3 2nd SIG feedback . . . 36
A.4 API agreement . . . 37
A.5 Original project description . . . 37

1 Introduction

Tribler is an open source peer to peer file sharing program developed at Delft University of Technology by the Tribler organization. Triblers motto is ”Towards making Bit torrent anonymous and impossible to shut down.” [17]. For backwards compatibility with the torrent network Tribler depends on something called an exit node. An exit node is a system that is used as the visible entry point to the anonymous Tribler network. Currently Tribler has a centralized cluster of exit nodes at LeaseWeb, however if anyone on the Tribler network were to ask from the torrent network for less-than-legal files on the torrent network it would seem that the exit node would request this file instead of the real (anonymous) downloader. This could theoretically lead to a shutdown of the exit node cluster. To prevent such a possible failure Tribler asked us to build an experimental system that functioned autonomously and in a decentralized way. To accommodate the costs of these decentralized systems it was proposed that the system would be able to sell multichain, a way of proving how much a client has up/downloaded on the network, on the decentralized marketplace that was going to be build by a different BEP (Bachelor End Project) group.

1.1. Structure

The paper is structured as follow:
Initially chapter 2 describes the initial design of the system, and the MoScoW requirements that were formed. Chapter 3 discusses how the automatic buying and installation of a new server is achieved. Afterwards chapter 4 provides an overview of genetic modifications and additional VPS buyers. The live tests are discussed in chapter 5. Chapter 6 discusses the testing and first SIG (Software Improvement Group) feedback. Lastly, chapter 7 discusses the conclusion and recommendations.

2
Initial System Design

When starting with the Bachelor end project we initially sat down with our client to discuss what he wanted the product to be able to do. He described that he wanted way to decentralize his Tribler exit nodes, so as to prevent a single point of failure. This decentralized system should function autonomous and scale itself with replication, preferably using something like genetic selection to allow the network to self-optimize.

For this we agreed to build an agent that during it's life cycle could replicate itself by buying it's own server and then installing a new child on this server. And make money to do this to do this to do this by selling multi-chain confirmed seed ratio on tsukiji (which would be rewritten by the decentralized market BEP group)

Initially work started with researching possible approaches and solutions for parts of the -to be written- program. In this chapter the results of this initial research will be discussed.

2.1. The agent life cycle

Because of the goal for autonomously operating and replicating code it is important to think about how it will live its life. in what way is it going to sustain itself, and how it is going to thrive in order to enable it to replicate. For this we constructed an initial life cycle for our agent in accordance with the goals of our client. As our client wanted to use the agent to act as Tribler exit nodes[15]

In figure 2.1 is described how this lifecycle would look like. It starts at "Living out the server lifespan" where the agent knows it has a certain time left on the server, and it knows that to sustain itself for another cycle it will need money to pay for the server upkeep.

To make money our client wanted us to set up Tribler exit nodes. With this it will build up a good seed ratio which can then be sold to earn money in the form of bitcoins.

So once the agent has lived (part of) its life it can see how much money it has earned. It can then decide which server to buy next. install itself onto it and the cycle will begin anew. If the agent is wealthy enough it can decide to add an additional vps and install an additional child onto it.

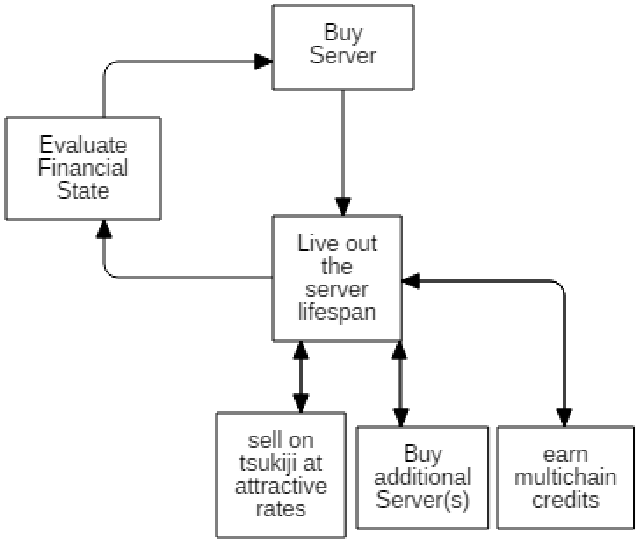


Figure 2.1: The life cycle of an agent

Now with a life cycle to relate concepts to, we can look at how several prerequisites of this cycle could be implemented.

2.2. Earning multichain for sustenance

Recently Tribler integrated multichain into it's production level code. Multichain is effectively a decentralized bitcoin-like way to register that a client uploaded/downloaded a certain amount. Because having a high uploaded ratio could in future be linked to being preferred to upload to it is possible to see this multichain as a form of currency that can be used to sustain our agent indirectly, as long as it can be transformed into a currency that allows new servers to be bought such as bitcoin.

2.3. The process of buying and selling

An essential part of our project is for our code to be able to sell its gained multichain for bitcoins. In order to do this, the code needs to be able to interact with an online market.

One of our project requirements is to work together with another BEP (Bachelor End Project) group, that is developing a self-organizing market. The product this group is trying to deliver is a market that is distributed rather than centralized, and does not involve a trusted third party to be able to operate.

The original idea for this BEP group was to build on the already existing decentral market called Tsukiji [18], which was developed last year by another BEP group. However, because the members of this year's BEP group deemed Tsujiki as too unstable and conceptual, they decided to build their own decentral market from the ground up, based on lessons learned by tsukiji.

It is important to note that their market is a **critical dependency** for our project. If they fail to deliver a fully functional market, our code will not be able to sell gained multichain on Tribler for bitcoins, and therefore will be unable to procreate.

We are going to work closely together with this BEP group so that we can effectively implement their decentral market into our product. While their market is not yet finished, there are already some things that can be said about it.

In their planned implementation, their market functions solely as a tool to connect askers and bidders. This allows the asker and the bidder to start the transaction with each other using a defined protocol. A schematic overview of a transaction can be seen in figure 2.2. Here, nodes A and B have been connected to each other by the decentral market. The plan is to exchange A's product for B's currency. To do this, A will send its product and B will send its currency. This is a correct transaction. However, it is easy to see, that using this model, there is no guarantee for a secure transaction at all. One of the parties can decide to trick the other: participant A might decide not to send the product and

leave with the money, and likewise, B could decide to trick A by receiving the product but never paying for it.

Figure 2.2: A schematic overview of a correct transaction in the decentral market

As mentioned before, our reason for using the decentral market is to sell our gained multichain on Tribler for bitcoins. This means, that we can be compared to participant A: we are offering a product in exchange for a payment in the form of bitcoin. A problem we could face is that we could get scammed by others that will take our product but won't pay the bitcoins asked for it. There is really no way for us to deal with this problem, other than using another way of selling our product (but we are required to work together with the other BEP group). Perhaps, they will be able to implement a better transaction system, or implement some sort of reputation system so that there is some way for us to make this scenario less likely to happen, our client however also agreed that -due to the experimental nature of our product- we were allowed to assume such a trustworthy situations existed.

2.4. Reproductive Strategy

So now that the agent has been making some money for a while it only has limited time left on the server it is currently on. So it starts looking at how much money it has. At the very least an agent will always want to purchase child if possible, as not buying a child and dying out tends to not be a successful evolutionary strategy.

An agent has several goals it needs to achieve

- It should be extremely unlikely for the entire network of agents to collapse. This is a goal set implicitly by the client as the client wants the agent network to provide a certain service. A complete collapse of the network would prevent this service to be provided.
- It needs to try and keep up with demand for it's service. This is also an implicit goal from the client, and for the same reasons.
- It should use it's resources effectively. This is also an implicit goal from the client. The network should provide as much service as it can for the given amount of money.

To achieve these goals a distributed technique was developed to allow for quick scaling, yet prevent collapse of the network as a whole within a single cycle. To explain the technique figure 2.3 will be used. It all starts at the top with a single instance A, which has 5 SC (Server Credits) in it's wallet. i.e. it can buy 5 servers for 1 cycle each (or 1 server for 5 cycles). This is just an abstraction of the actual money involved.

The first and foremost important strategy is "don't change a winning team" as agents are practically immortal they only need the money to sustain themselves on a new server to continue. Therefore in figure 2.3 every time a node adds an ' sign to the label, it is effectively the same agent installed on a different server. An agent will always prioritize sustaining itself above procreation and will be biased towards sustaining itself compared to creating children. In doing this it also accomplishes the prevention of complete network collapse. As the copy of the agent already has budget for multiple cycles. Once A has enough money reserved to sustain itself it will start saving additional money to create it's own child. This is an amount of cycles bigger than 1 (because 1 is used up in the process of generating a child) However if an agent is unable to replicate even itself it has one last trick, because it's parent tries to always replicate itself first it can try to send it's money to the parent in order to protect this branch of agents, and increase their odds of survival.

So as an example we can take figure 2.3 which starts in the Init fase with A having a wallet with 5 SC. It reserves 2 for it's own continuation in A', so A' ends up with 1 sc in it's wallet as it's server also cost 1 to acquire. Then A creates another 3 servers as it has 3 remaining SC. this starts the first real

generation's life cycle. At the end of this life cycle some of the servers have made money by selling their seed ratio. and they start looking at their possibility's green for example does not have enough SC to keep itself existing. Knowing it's end is near it decides to transfer it's money to its parent A, which in the new generation is represented by A". meanwhile, blue has made so much money it can get a child of its own and starts the yellow branch of children.

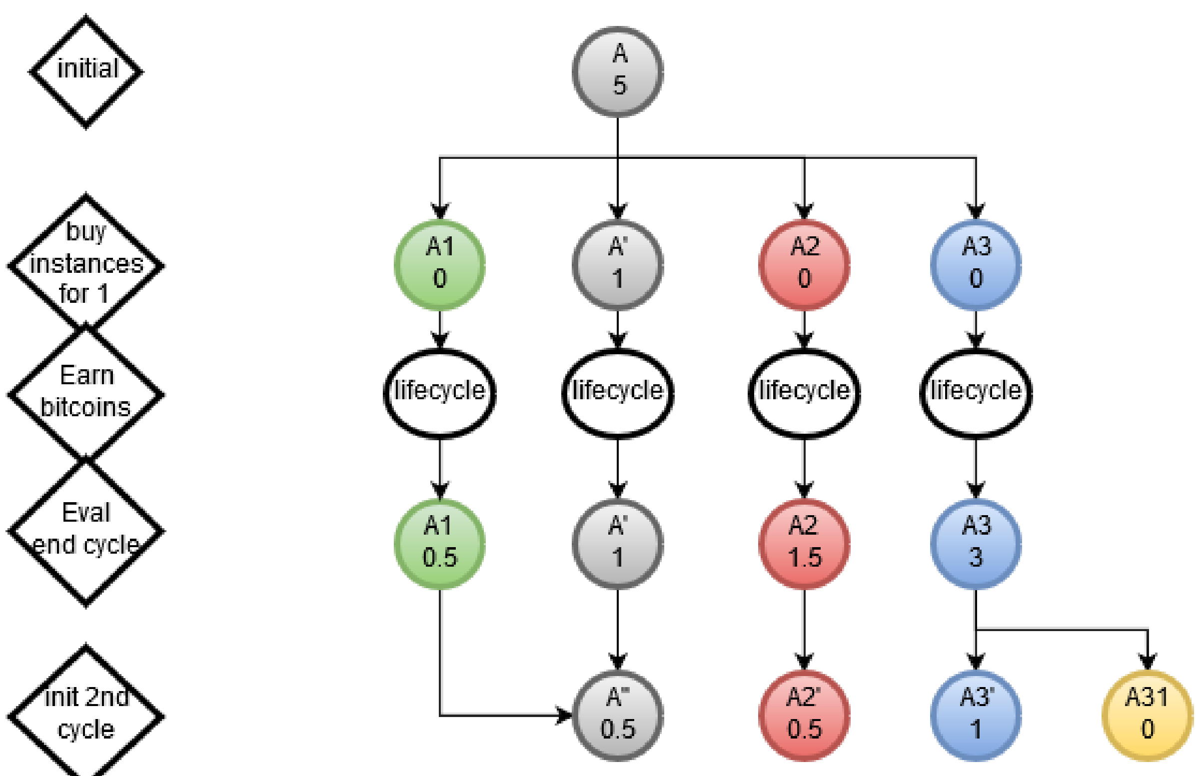


Figure 2.3: The process of multiple cycles.

2.5. Project Requirements

Once we had researched a general idea for how to implement our system we sat down with our client and agreed on a list of requirements for our product. This list makes use of the Moscow principles: the problem is divided into must haves, should haves, could haves and won't haves. Splitting requirements into those four categories helps making clear where the priority lies when it comes to the requirements.

2.5.1. Must haves

Must haves are what a system absolutely must have. It contains the most basic specifications of the system, such as the things the system is designed for. Failing to implement must haves usually means delivering an unwanted system. These can usually be found in the assignment specification.

- The system must start up.
- The system must function autonomously.
- The system must be able to buy another server using bitcoin.
- The system must be able to install itself on this newly bought server.
- Github should be used to manage the code.
- Any tests written for the system, should pass on the software that is shipped with it.
- The software must be published under an open source license at the end of the project.

2.5.2. Should haves

This category contains everything the system should have, but is not strictly required.

- The system should be written in Python.
- the system should be tested.
- The system should be documented so that its workings would be clear to a programmer uninvolved in the project.
- The system should be capable of using mutation to alter its own behaviour on newly installed servers.
- The system should function as a Tribler exit node and provide the accompanying services.
- The system should be able to sell multichain for bitcoin.
- The system should be able to choose among multiple virtual private server providers.
- A mass buyer agent (auto-buy bot) should exist that can buy multichain with bitcoin.

2.5.3. Could haves

This category contains ideas that would be nice to implement. These features are not strictly necessary and should only be done if time allows it. This category simply states some ideas to extend the system in the future.

- Use multiple addresses for bitcoin transactions to make the system harder to track.
- Provide a test suite to check if the system is installed correctly and try to correct itself if it fails.
- Send a birth certificate to its makers.
- At the end of the lifecycle of a VPS, the bitcoin wallet could be inherited or emptied to an account of another instance, instead of lost. Alternatively, the leftover currency could be given back to the central research wallet.
- Create a merchant type node that both buys and sells things to make profit.
- As a genetic alternative the agent could inherit the codebase of its parent rather than relying on github (making the system even more autonomous)

2.5.4. Won't haves

The last category is the won't haves. This section contains everything that should not even be tried to be implemented, because it is not necessary, too difficult or too time consuming. Another possible reason for an item to be in the won't have group could be that it is not ethical or legal to implement them.

- Have the system (or any third parties arranged in the original code of the system) keep records on who downloaded what. Won't be implemented as this beats the purpose of Tribler.
- Won't be secure due to the heavily experimental nature of the project, and known security flaws in both multichain and the decentralized market, no guarantees will be made on the security of the system.

3 Procurement, Automated Installation and Initialization

Because of the complete automation required by the client and because virtual private servers can only be rented for a certain time, automatic procurement and initialization of future generations is required. Using the bitcoins the vps made during its lifetime a child server should be bought and the programs' code should be installed on it.

3.1. Bitcoin wallet

For the implementation of the Bitcoin wallet, to allow for transactions, it was important to decide on which approach to a bitcoin wallet was picked. in essence the -relevant for the agent- aspects could be distinguished.

- decentralized: This means that no single point of failure exists other than the agent itself. For example Blockchain.info offers wallet services, however due to the centralized nature of their service we would fully depend on their servers being online. [2]
- offloaded: This means that the actual chain does not need to be saved on the client side. This is very important as the bitcoin block chain already sizes in the tens of gigabytes[1]. And most VPS providers, especially the cheaper ones do not include that much space.
- secure: Some wallets, especially centralized ones, offer little to no guarantees as to in how far the service provider can influence the wallet. In worst case scenario's this could mean stolen bitcoins as allegedly happened with Mt. Gox [3]

A client that fits these aspects is electrum. Everyone is capable of downloading the server code for it and adding to the collective of servers that provide acces to the bitcoin chain. while transaction creation is done locally on the client allowing for a payment to be saved in a hex that can be broadcast to the bitcoin network.

3.2. Buying a VPS

A vital part of the system is to be able to replicate itself. In order to do so, it must be able to buy or hire a new VPS (Virtual Private Server) by itself, using its acquired currency. Different VPS providers provide all kinds of different VPS plans. For example, one VPS could come with a control panel and 20GB disk space while another VPS could come without a control panel and 10GB disk space. There can even be multiple plans offered by a single provider. Not all offered VPS' are suitable for our project. Therefore, it is important to compare them and find the ones that are useful to our project.

3.2.1. Finding suitable VPS' for our project

The following are aspects of a VPS provider plan that are important to our project:

- **Storage space:** We need enough storage on the VPS to install the source code of our project and relevant prerequisites. While this may be difficult to predict, 10 GB should suffice. And while most VPS providers give at least that much. it is important to realize this aspect exists.
- **Bandwidth:** We'll be running a Tribler exit node on the VPS as a means to generate income. The more bandwidth this Tribler exit node uses, the more the VPS can earn. This means that the amount of bandwidth that is available to our VPS is of great importance.
- **Operating system:** In our project we're using a lot of libraries and components that only automatically (or in an supported automateable way like pip or apt-get) work on certain operating systems. We decided to focus on developing our code for Ubuntu 14.04, which is a popular choice as OS for a VPS that falls in the long term support category. Hence, most VPS providers have the option to use Ubuntu 14.04 as OS, which is a good thing, because we require that OS because of our choice.
- **DMCA policy:** The biggest problem with running a Tribler exit node on any network is the chance of receiving DMCA complaints about the spread of copyrighted content. If we are to run Tribler exit nodes on a VPS, our VPS provider will most likely receive DMCA complaints. However in many countries the DMCA is not acknowledged. So these VPS providers in these countries would have preference over other providers
- **Payment method:** We want our VPS to earn Bitcoin. If we could pay for a VPS using Bitcoin, that would be great, because then we won't have to exchange our Bitcoins for another currency first. Also, Bitcoin payments are relatively easy to be done automatically and anonymously, unlike for example pay pal payments or bank transfers, which makes this an excellent payment method. Not many VPS providers accept Bitcoin as payment, but they are out there.
- **Price:** It should speak for itself that we want the price to be as low as possible. The lower the price of the VPS, the higher the chance of earning enough to buy a new system. All else being equal

In conclusion we would want a server that has at least 10GB of disk space, as much bandwidth as possible, Ubuntu 14.04 as OS, and we want to pay for it using bitcoin. Also, if possible, we want the VPS provider to have no problems with DMCA complaints. Of the VPS' that fit those requirements, we want the cheap(er) ones.

After doing some research, we found that the following VPS providers provide VPS' that are suitable to our project, namely Zappiehost [21], Offshorededi [10] and THCServers [13].

	Zappie Host	OffshoreDedi Host With Privacy	THCServers.com FOR GEEKS, BY GEEKS!
Accepts Bitcoin	Yes	Yes	Yes
Storage space	30GB	10GB	60GB
Bandwidth	100GB @ 100Mbps/s	Unmetered @ 100Mbps/s	Unmetered
Operating system	Ubuntu 14.04 included	Ubuntu 14.04 included	Ubuntu 14.04 included
DMCA Policy	Based in the New Zealand, so probably strict DMCA policy	Based in Latvia, supposedly ignores DMCA requests	Based in Romania, DMCA policy unknown
Price	$4.50 per month	€7.99 per month	€14.95 per month

Figure 3.1: An analysis of three suitable VPS plans

For these VPS providers, an implemention was made that allowed the code to automatically order a VPS on those VPS providers' websites.

3.2.2. Possible ways of implementation

One problem is that VPS providers typically don't strive to make it easy to automate the process of buying a VPS. Instead, they have forms that a human can enter his/her information in and then submits in order to buy a VPS. They will then often get instructions on how to access their VPS via email.

To make our system function truly autonomous, we needed to implement a way for our software to automatically buy a VPS at a VPS provider. Seeing as there are generally no API's to use for this purpose, we needed to find a way to have our software fill out the forms automatically, execute the payment, and retrieve the VPS information.

If would be best to have such a method for multiple VPS providers, so that if at some point in the future there is a problem with a certain VPS provider, it will still be able to buy new VPS‘ at other providers.

There are two ways of filling out the forms at VPS providers automatically:

Submitting postdata

Check which form fields need to be filled in and send those to the page that processes this information. While this would work in some scenarios, in practice there are a lot of problems with it. To name a few:

- The pages that process the information can be protected against data coming from another domain (which would be the case using this method)
- If there are multiple steps in the order process and there are cookies involved, it can get tricky to implement
- When a page redirects to another page, it can be hard to collect the necessary information
- The order might fail because the VPS provider detected that it is not a human that is placing the order

Using web browser automation

It is possible to automate a web browser to visit a webpage, fill out forms, submit data, extract data, etc. Using this method will take care of most problems that occur when using postdata. It will be easier to complete the whole process and since an actual web browser is used, it is much more difficult to detect that a bot is used to place the order.

A tool for web browser automation is Selenium [11]. Selenium allows the user to automatically deploy web browser instances and send commands to them.

3.2.3. Automatically ordering a VPS using Selenium

Now that a indication of which VPS' are used has been given. An expanded explanation will be given for the automation of one VPS namely Zappiehost. To do this, we used Selenium to automatically make its way through their website to order the VPS and get the neccessary login details. Here is our script in action:

Step one: Opening the browser The code uses Selenium to open an instance of Mozilla Firefox and opens the URL of Zappiehost's VPS plan page:

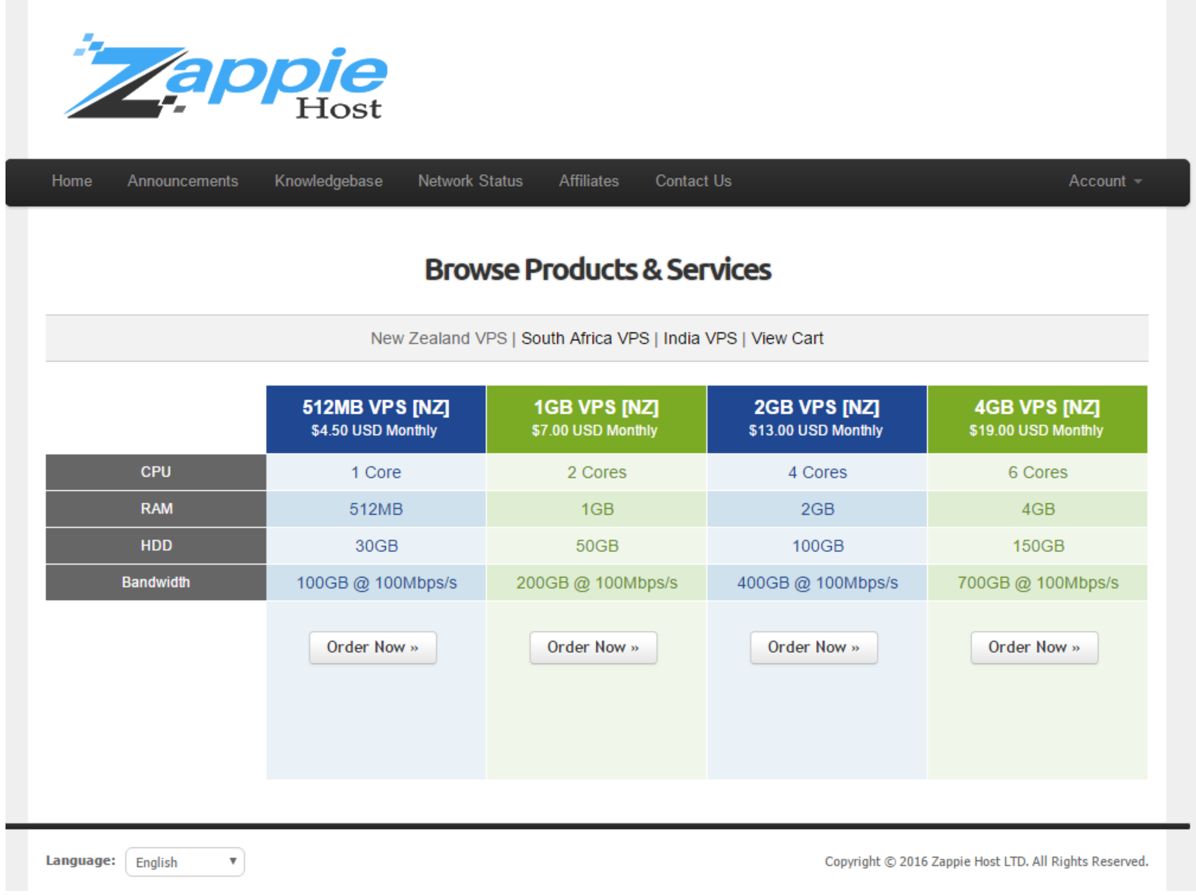


Figure 3.2: The VPS plan select page

The script will now send a command to press the "Order now" button for the cheapest VPS option, which will bring us to the next web page:

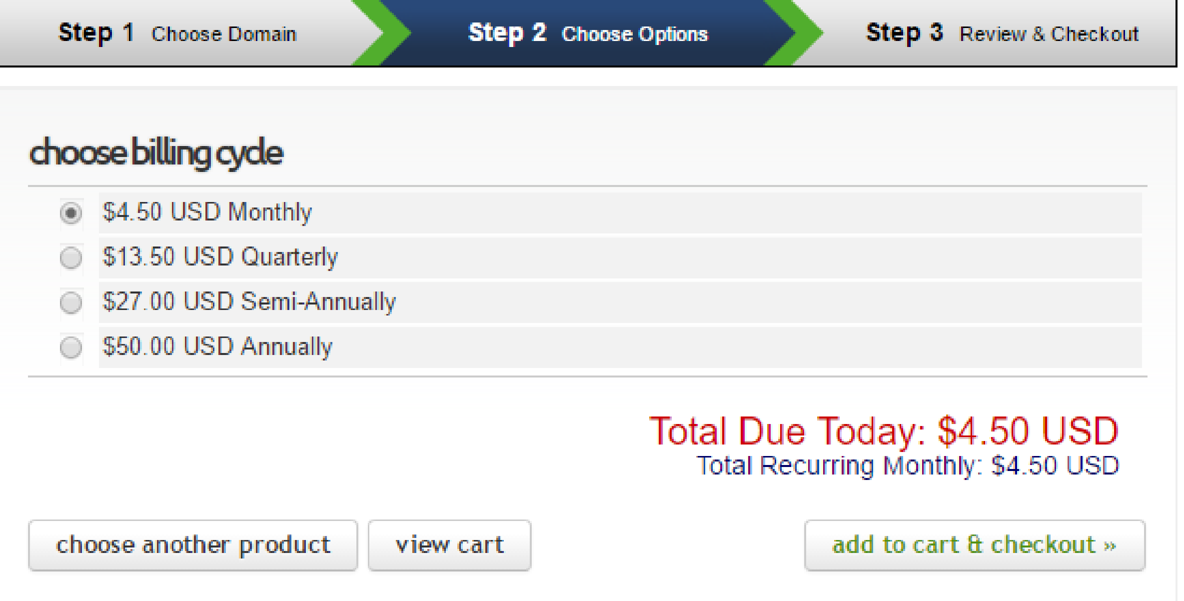


Figure 3.3: Step 2 of the order process

On this page, we can add the VPS to the cart by making our code click "add to cart & checkout",

which will send us to the registration form:

Empty Cart Continue Shopping

Total Due Today: $4.50 USD
Total Recurring: $4.50 USD Monthly
Promotional Code Go

your details

Already Registered? Click here to login...

First Name	AfLIF
Last Name	UcCAv
Company Name	
Email Address	hfRwG@MEpRKSX.ch
Address 1	ttBwMTHVGZaMjih 12
Address 2	
City	pLGHmFci
State/Region	QPXmua
Zip Code	81974
Country	Jordan
Phone Number	5252953532
Password	••••••••••••••••
Confirm Password	••••••••••••••••

Password Strength: Weak

payment method

PayPal
Bitcoin (BitPay)

notes/additional information

You can enter any additional notes or information you want included with your order here...

Please Wait...

Figure 3.4: The order form after our software automatically filled it in

On this page, several things need to be done:

- Under payment method, the "Bitcoin (BitPay)" box needs to be clicked. This can be done simply by sending Selenium a command to do so
- The First Name, Last Name, Email Adress, Address 1, City, State/Region, Zip Code, Country, Phone Number, Password and Confirm Password need to be filled in. We noticed that Zappy Host is not strict with checking this information, so we programmed a Bogus Information Generator that returns gibberish in a specified format (alphabetic string, numeric string, phone number, password, etc). Using this, we can enter all of the necessary fields by sending Selenium commands to enter a certain value in a certain field.
- Submit the form. This can again be done by sending a simple command to press the submit button

We will now end up on a page that will process the information and redirect us:

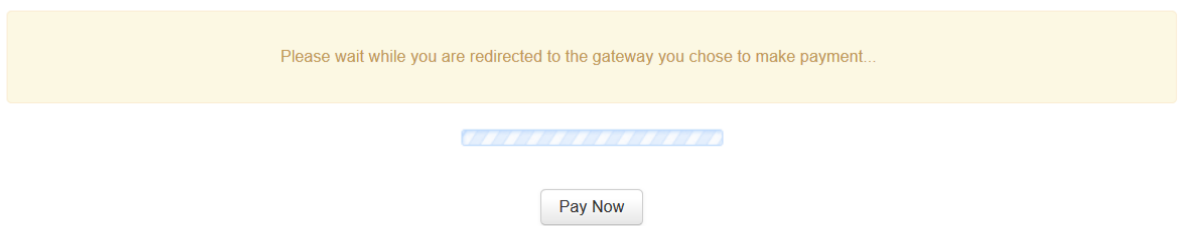


Figure 3.5: The redirect page

We will now land on the payment page of BitPay:

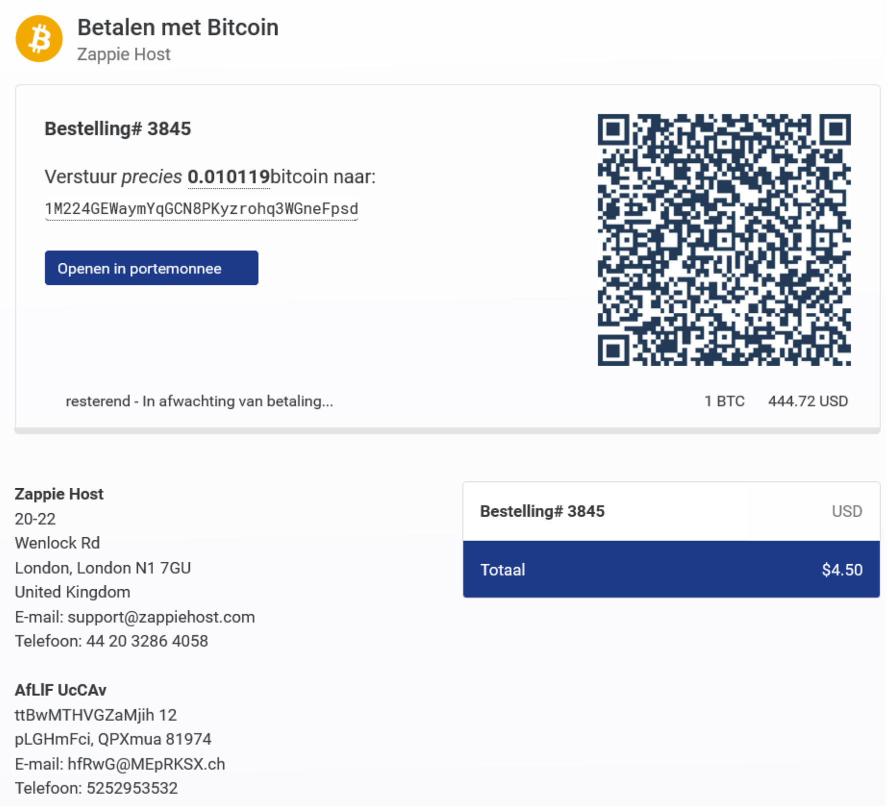


Figure 3.6: The payment page

On this page, we can tell Selenium to extract the data on the amount of bitcoin to transfer and the wallet to transfer it to. We now need to actually make the payment.

To make the payment we are using our implementation of electrum [5]. Our server transfers the requested bitcoins to Zappiehost, while Selenium waits for the page to confirm that the payment has been made.

Now that we've succesfully ordered our VPS on Zappiehost, all we need to do is get our SSH login information. We've already found out that the SSH username is "admin" by default. In other words, we just need its IP address and SSH password. To do this, our code first waits 30 seconds, to give Zappiehost some time to process the payment and set up our VPS. Then, our code restarts the web browser (to start a new session). We then log in on Zappiehost using the email address and password we set ourselves. After a few clicks, our code lands on a page where our VPS' IP Address is listed:

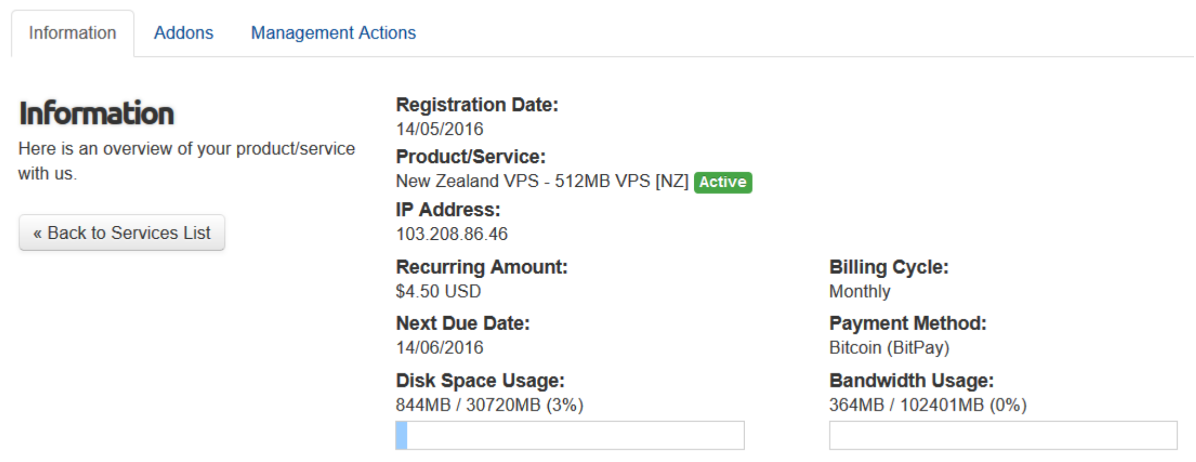


Figure 3.7: On this page we can see our VPS' IP address

We can extract this IP address. Now we just need the SSH password. The thing is, the SSH password has been emailed to the email address that we registered on Zappiehost with. If we want the SSH password, that means we'd have to build in a working email receiver and reader to get it. Instead, we found an easier way to obtain the SSH password: setting it ourselves.

In the Zappiehost control panel, we can go to a page where we reinstall our VPS. When we do that, we have the option to set a new SSH password (and choose the operating system, but we always want Ubuntu 14.04). So our code goes to that page and fills in a new SSH password:

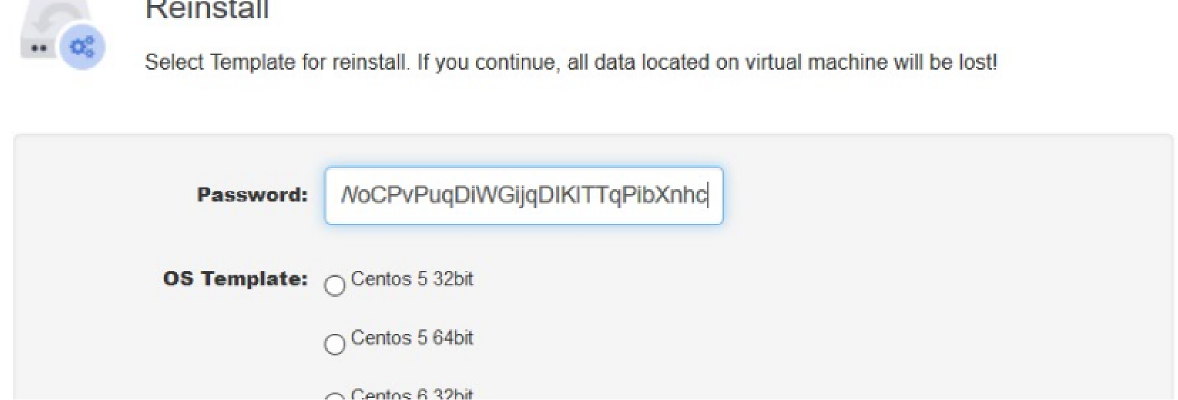


Figure 3.8: On this page we can choose our own SSH Password

When we click the submit button, Zappiehost will reinstall our server with our chosen SSH password. So, up to now, we've ordered a VPS and obtained its SSH login information, which means we're done! All we need to do now is wait for our SSH to be done with re-installing, after that we'll be able to login on the new VPS using the login information. Then we'll be able to perform our next step: replicating our code onto the new VPS.

3.3. Automated installation procedure

To automate the process of installing on a child the Installer python class was constructed. After getting the contact information of the child it uses SSH to transfer the relevant download and installation commands to the child. It starts with installing git on the child after which it clones its own project from github. After that download it runs the script build.sh to install its dependencies.

4

Extended Procurement And Genetic Modifications

As mentioned before in chapter 3 an important aspect of our system is that it should replicate itself on child server. VPS servers are not free and for that reason our system should make money. In order to achieve this the VPS should run a Tribler exit node and sell the acquired reputation for bitcoins. Another bachelor project group is implementing the market and our system should use this market to do this. Furthermore it is important for the agent to be cost effective. In this chapter we will address these different issues.

4.1. Addition of implemented VPS providers

Halfway through the project the client requested that more VPS would be available for use by the agent. For this 2 more VPS providers where identified, as can be seen in figure 4.1

	SHARK SERVERS	YOURSERVER
Accepts Bitcoin	Yes	Yes
Storage Space	50 GB	10 GB
Bandwidth	Unmetered	Unmetered
Operating System	Ubuntu 14.04	Ubuntu 14.04
DMCA policy	DMCA ignore	no known policy
Price	$7.62	€4.00

Figure 4.1: An analysis of two additional VPS

4.1.1. Benefits of alternatives

Benefits of these alternative VPS providers are mostly their unmetered bandwidth. An additional advantage of Sharkservers [12] is that they have a DMCA ignore policy. An advantage of Yourserver [20] is that they are very cheap.

While implementing the new VPS providers, problems were encountered that were not present with the three previous implemented VPS providers.

When the Sharkservers VPS hosting provider was implemented using Selenium, the code could not succesfully order a VPS. The reason buying a VPS failed, was because Sharkservers banned our IP address. The banning reason was fraud, and the message was that the IP's geolocation did not match. During the registration, one of the fields filled in is the country. The code selected a random country from the list of available countries. Being banned seemed to suggest that Sharkservers checks the IP address of the visitor and checks if the country the visitor is in matches the country they filled in. If that was indeed the ban reason, it would be easy to avoid in the local code. The machine running the code was located in the Netherlands, so if the code would just select "The Netherlands" as country. However, that would not solve the problem for our system. The code will be on VPS' across the globe, and it can be located in multiple countries. In other words, the code needs to have a way to find out the country that its own IP address is located in.

4.1.2. Retrieving the country the current machine is in using IP address

The first step towards finding out the machine's own country, is finding out the machine's own IP address. To do this, a python package called *ipgetter* [6] came in handy. Ipgetter describes itself as:

"This module is designed to fetch your external IP address from the internet. It is used mostly when behind a NAT. It picks your IP randomly from a serverlist to minimize request overhead on a single server."

With this module integrated, requesting the machine's own IP is easily done. Now the machine needs to find out the country of origin for its own IP address. Maxmind [9] Provides tools to find out the country linked to an IP address. Using the package *python-geoip-geolite2* [7] you obtain access to the geolite2 database, which is used to associate an IP address with a geolocation. This module returns the country's country code (for example: NL for the Netherlands, UK for the United Kingdom), which is ideal, because on most hosting providers, Sharkservers included, the list of countries' values are the countries' initials. This makes selecting the correct country corresponding to the machine's own country in the online registration form easy.

For modularity's sake, a wrapper class *CountryGetter* was created that took care of the whole process of finding out the machine's own country, utilizing the *ipgetter* and *python-geoip-geolite2* modules mentioned above. This wrapper class is fairly simple:

```
1  from geoip import geolite2
2  import ipgetter
3
4
5  class CountryGetter(object):
6      """
7      This class enables you to get your own country code by your IP address
8      """
9
10     @staticmethod
11     def get_country():
12         """
13         Returns country code of the IP address that belongs to the machine this method is called on
14         """
15         myip = ipgetter.myip()
16         match = geolite2.lookup(myip)
17         return match.country
```

Figure 4.2: The wrapper class CountryGetter

After importing this class, whenever needed, *CountryGetter.get_country()* can be called to get the country code of the country the machine is located in.

4.1.3. Using temp-mail for temporary email addresses

While implementing code to order a VPs on another bitcoin-accepting VPS Provider, Yourserver [20], another problem was encountered: while it was not required to have an email address to register at

a VPS provider untill now, on Yourserver it was required, because they send the password via email. This meant the code needed to be supplied with a way to check emails in order to work with this VPS provider.

Using a single email address across all agents is a bad idea: an email address can only be used once to create an account on VPS providers websites like Yourserver.se. Instead, it would be ideal to have a temporary, disposable email address. Online, there are many services that provide temporary, disposable email addresses. Two examples of this are *10minutemail.com* and *temp-mail.ru*. The latter of those offers an API. A wrapper for this API has been developed called *temp-mail* [8]. This python package describes itself as:

"Python API Wrapper for temp-mail.ru service. Temp-mail is a service which lets you use anonymous emails for free."

After importing this package, it is fairly easy to request a temporary email address, and read the emails received on this temporary email address. The implementation of temp-mail into the project is straight forward: when registering on Yourserver, before entering the email address field, a temporary email address is requested via temp-mail. This email address is then entered into the email address field of the form. When submitting the form, Yourserver will send two important emails. One of them contains the password for your registered account, the other of them contains an activation link that you need to visit to activate your account. In order to get this information, our code waits for 30 seconds after finishing the registration, to make sure Yourserver has had enough time to send the registration information. Then we request the HTML content of the latest two emails received on our temporary email address. Then our code extracts the activation link and the password from this data. We then use Selenium to visit the activation URL to activate the account.

The solution is not completely safe, a few things could go wrong. For example, if someone were to send an email to our temporary email address before we fetch our latest two emails, the code will fail to extract the important information. However, the chance problems like these actually occur is so small, that we decided to go with this straight forward implementation anyways, rather than spending a lot of time to possibly prevent an unlikely case in which this could go wrong.fa

4.2. Running Selenium Headlessly

With the implementation of web browser automation using Selenium discussed up to this point, it works perfectly on your laptop or machine, as long as it has a monitor. However, a problem arises when trying to run our code on a VPS. A VPS is headless. A headless system is a system that runs without a GUI (Graphical User Interface). It makes sense for VPS' to be headless: they don't have a monitor, so there's nothing to display anything graphical on. The problem is our Selenium code uses Firefox, and firefox requires a GUI to run. In other words, our code does not work on a VPS.

4.2.1. Using Xvfb to emulate a display

There is a display server called Xvfb [19] that mimics a display. Xvfb performs all graphical operations a normal display driver would, except it does not show any output. For our project, this is ideal. Using Xvfb, we simulate an attached display, which enables us to run firefox on a VPS. This in turn allows Selenium to work on VPS'. Now, VPS' are be able to order new VPS' all by themselves.

4.3. Running the exit node

Running a Tribler exit node on the agent means that the agent needs access to Tribler through a custom headless startup procedure. This is not something Tribler was inherently build for, although a headless start up procedure exists in some form this still had some limitations that made it unsuitable to run.

4.3.1. multichain

For a successful life cycle our agent requires a way to earn money. This could be done by earning multichain. However multichain while multichain is somewhat integrated into tribler the startup procedure needed to do quite a bit of custom work other than adding the "use multichain" trigger to the settings.

4.3.2. decentralized market and API negotations

As part of the project it was necessary to integrate the decentralized market that was to be build for Tribler. The market had to be integrated because without it the agent would not be capable to generate money and as a result it would not be capable of replicating itself. However since this market was implemented at the same time by another BEP group and did not yet exist for most of the project it was important to negotiate a way of interacting with it.

An api needed to be created so the distinction between the project and tribler was clear to everyone. Thus preventing issues about code changes on this critical dependency. The need for an api however was quite a complicated situation as the market itself would be integrated into tribler. Which put many restrictions on the way interaction could be constructed. Solutions such as using RPC or similar ideas would simply not be integrated into tribler making them unfit for our communication. The first proposal contained just two methods which returned nothing. This would make it a lot harder to interact with it so a second proposal was made which included callback functions. Since this was not a nice solution it was changed to using an event based system. This was -then again- renegotiated because of the existence of a simpler solution. Eventually a direct code call api was setup because we already would have tribler source code on our system to be capable of running the exit node, we only negotiated the form of these function such as expected functions, their functionality and their location, input and output. With these specifications a mocking environment was made to test whether our implementation would function with their market if it functioned as negotiated. This is to minimize integration times of the market once it was finished and to allow testing if TENNET used the market according to the specifications.

To make using the market easier and even more stable a class was created to wrap around it. This also made the mocking of the original class a lot easier. The actual mocking pointed out that the order book variable was missing in the api, and was consequently added to it. It also revealed that a package of the market was missing in the package list of Tribler.

The final version of the negotiated API can be found in apendix A.4.

4.4. Genetic modification

To prevent our system from wasting money in the case that a vps buyer ceases to function due to for example a website revamp we implemented a genetic modification system that allows a genetic approach to which vps is bought. To accomplish this a JSON parseable dna-structure was defined as can be seen in listing 1, and a method to mutate this structure.

```
1  {
2      "vps buyers": {
3          "OffshoredediBuyer": 0.5,
4          "SharkserversBuyer": 0.5,
5          "ThcserversBuyer": 0.5,
6          "YourserverBuyer": 0.5,
7          "ZappiehostBuyer": 0.5
8
9      },
10     "mutate rate": 0.05,
11     "own vps": "ZappiehostBuyer"
12 }
```

Listing 1: DNA.json file, which supports 5 vps buyers. On a server bought by ZappiehostBuyer and with a mutaterate of 0.05

There are several broad ideas behind the choice of this structure. First is the mutate rate, this is effectively comparable to the learning rate in a neural network. It represents how much weight it will give to new experiences. second is that this mutaterate should function the same irrespectably of the amount of buyers or the sum of their weights. i.e. adding more buyers would definitely upset the amount and sum of weights of vps buyers. but mutate rate should function the same, so that if a way of modifying the mutate rate automatically is added or a decentral way of adding vps buyers, the program will still run expectably. Lastly the weights should never become truly 0, although they should go close

to zero relatively quickly. This is to allow for both temporary failing of a VPS provider (or a temporary unactractive policy choice) but also to waste as little as possible on non-functioning vps buyers.

to accomplish this the following method was devised

```
Algorithm to re-evaluate weights in dna
    1. Normalize the dictionary as a vector, remembering it's length
    2. Add mutate rate to the winning buyer
    3. Denormalize to the original dictionary length.
```

so, running this algorithm on our original dna (assuming a ZappiehostBuyer being chosen) would result in the following

dictionary key	weights	normalized weights	add mutate rate	denormalize
OffshorediBuyer	0.5	0.2	0.2	0.476
SharkserversBuyer	0.5	0.2	0.2	0.476
ThcserversBuyer	0.5	0.2	0.2	0.476
YourserverBuyer	0.5	0.2	0.2	0.476
ZappiehostBuyer	0.5	0.2	0.25	0.595

as can be seen it is now significantly more likely that a zappiehost server is bought. yet still relatively likely that a different server is bought. but now what if we follow this pattern a bit more and a ThcServer happens to be bought.

dictionary key	weights	normalized weights	add mutate rate	denormalize
OffshorediBuyer	0.476	0.190	0.190	0.453
SharkserversBuyer	0.476	0.190	0.190	0.453
ThcserversBuyer	0.476	0.190	0.240	0.573
YourserverBuyer	0.476	0.190	0.190	0.453
ZappiehostBuyer	0.595	0.238	0.238	0.568

As can be seen now, both thcserver and zappiehost are more likely to be bought on this server because it's parent was a thcserver and it's grandparent a zappiehost server. This child would in turn create more children, and since every server has a risk of being shut down due to -for example- copyright infringement offset by possible gains in the form of either cheaper running cost of more possibilities to earn money. The dna should eventually converge to a local optimum.

5
Live Trials

Along with regular testing of our code, three major live trials had been planned to test the functionality of the system. These trials are in increasing complexity. The first prototype is only about basic functionality like the installation procedure. The next trials increase in complexity, to work towards the final product.

5.1. First prototype

With an automated VPS buyer, working Bitcoin wallet and the automated installation procedure functioning a first prototype/demo of the system was constructed with which a computer with initialized wallet that already had some money in it would buy a new VPS, and then install itself on the new VPS. We could then check whether it had successfully created a working agent on the new server by checking whether it had instantiated a working wallet on this child, and that the child system passed all automated tests.

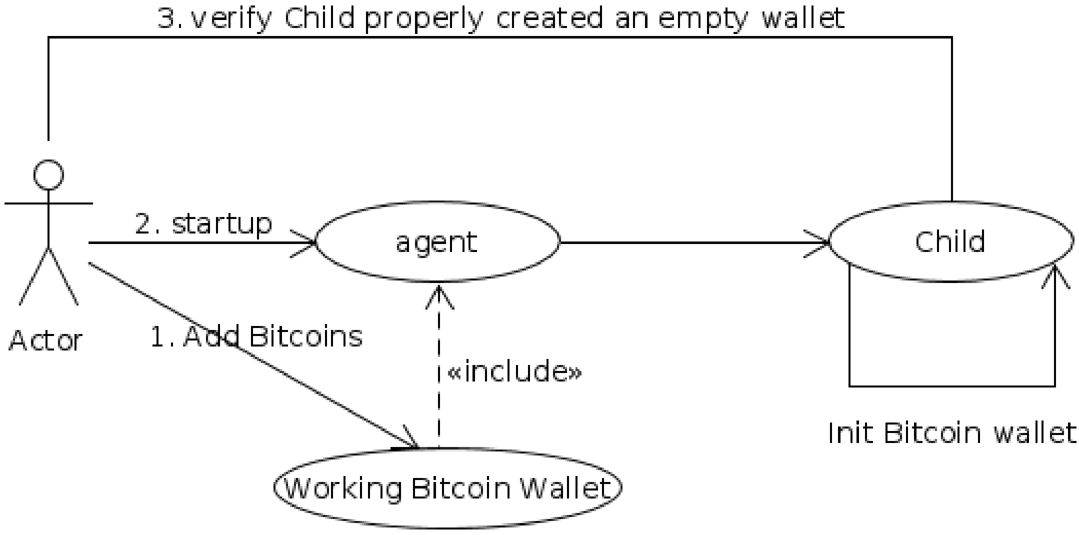


Figure 5.1: The initial prototype.

5.2. Running a ”long term” exit-node

During the tests of the first prototype and the development of multiple automated VPS buyers several VPS’ were rented and since they would not serve any other purpose it was decided that tribler exit-nodes would be run on them to both add to the network capacity and test how well these providers performed.

Initially it was expected that the VPS provider with the best price per GB bandwidth offer was going to win out. Because it would simply be the most cost effective. As is illustrated in figure 5.2

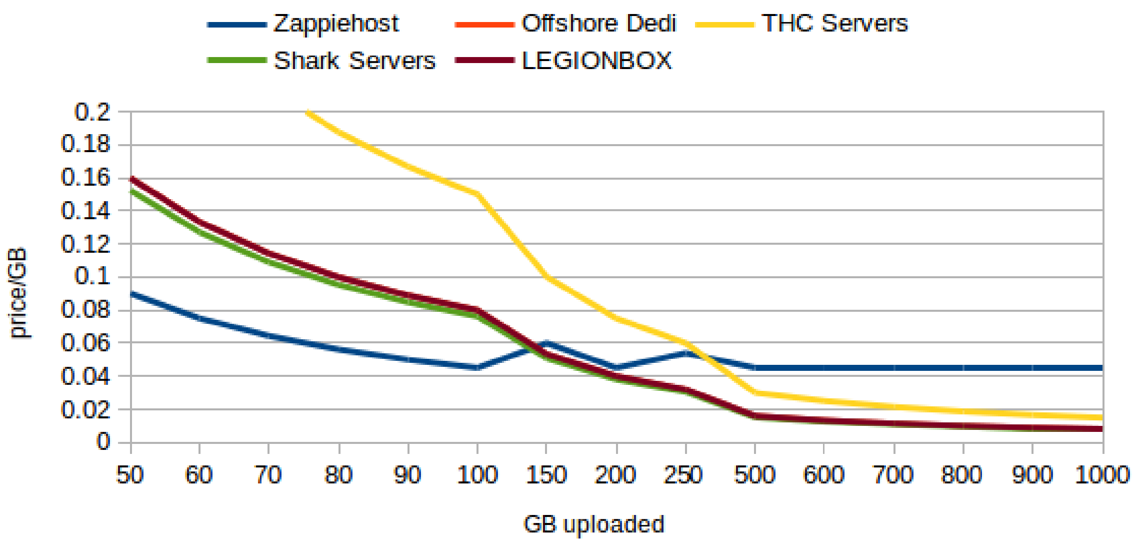


Figure 5.2: a representation of theoretical price/GB if a server is shut down at certain upload capacities. In those cases where bandwidth exceeds spec, it is assumed a new server is bought.

However in reality the importance of specs was severely overestimated, because as can be seen in figure 5.3 our Zappiehost testserver managed to exceed it's bandwidth spec sixfold and was still functioning. At this point it was providing exit-node services equivalent to 1/60th of the entire exit node network uploading about 150GB a day[16]

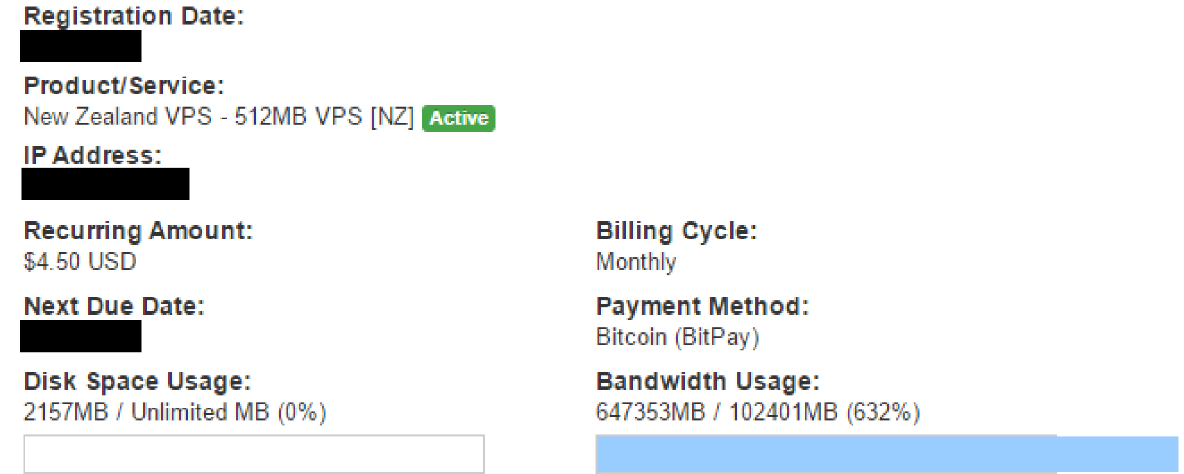


Figure 5.3: Zappiehost still ran our exit node, even with 6 times the bandwidth usage.

On the other hand we mis-estimated response times in countries that did actively follow the DMCA. with servers being shut-down within 24 hours. As can be seen in figure 5.4 the server had not even uploaded 150GB in total.

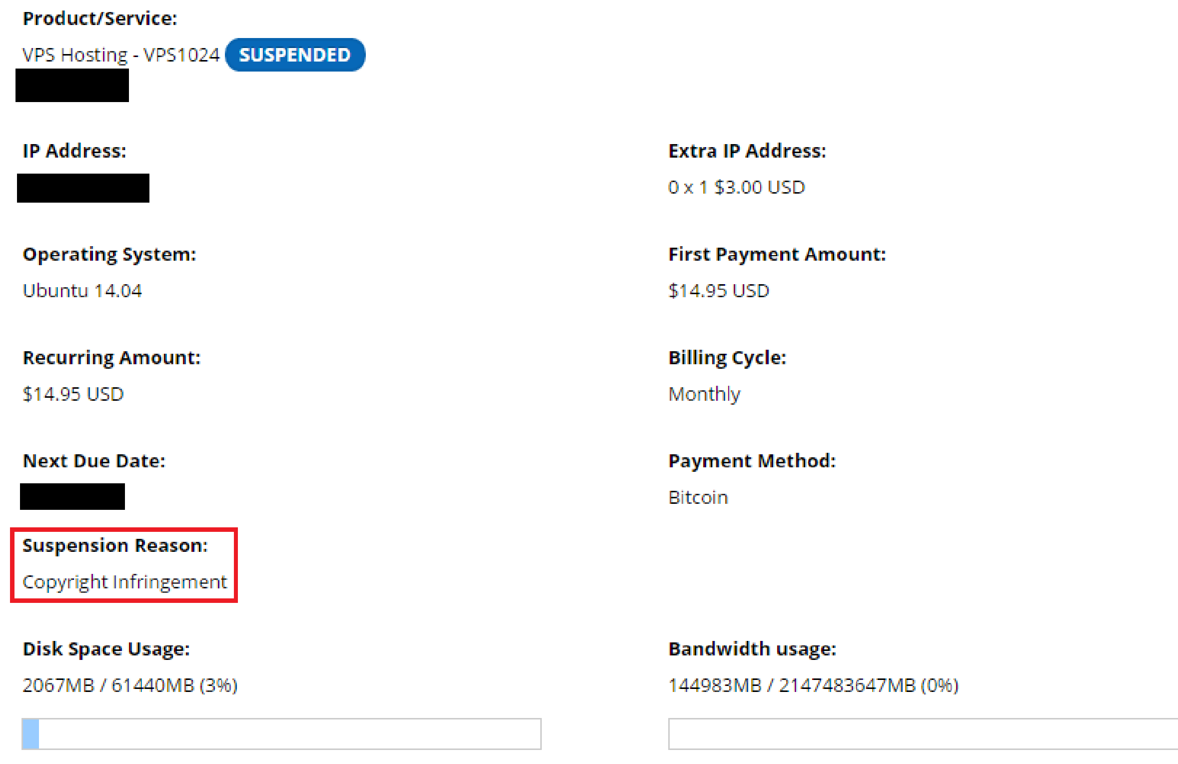


Figure 5.4: After less than a day THCservers had banned the test server due to copyright infringement, as can be seen in the red rectangle. Only 150 GB had been uploaded at this time.

These results reaffirm the importance of a genetic approach above a spec based approach in these matters. Since these specs can be flexible (either due to lax control, or heavy focus on DMCA) finding a solution in a less data-driven and more reality driven method is preferable in this situation. Because provided data is not inherently trustworthy

5.3. Second prototype

With multiple VPS providers automated, the possibility to automatically setup the exit node on these VPS and a genetic algorithm in place, yet with an absent multichain market. We constructed the second prototype which can not yet make money by itself. But is able to perform replication, mutation and the for our client important running of the exit-node. It could do this by starting with an initial bitcoin wallet and after purchasing a child sending all its money to the child. so that the child would have "earned" enough money to create a child of itself.

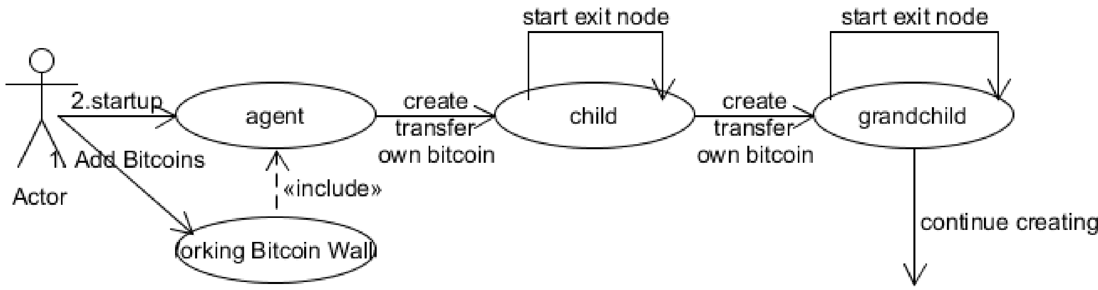


Figure 5.5: The second prototype.

However as preparations to start this test were underway it took longer than expected to get the money to start the agent, and one of the VPS providers changed their website, preventing us from fully starting our test. While a more limited test was theoretically possible after a quick identification of the problem. The client understandably did not want to address more funds until the problem had been removed. While these problems were finally fixed, no time was available for large scale testing of the 2nd prototype.

5.4. Third Prototype, a.k.a. ”full product”

A third prototype was also prepared, however due to time restraints and the problems mentioned in the previous paragraph we could not verify whether it fully worked yet. The third prototype would have been in essence the second prototype with an added ability to make money by selling bitcoin on the market. additionally it would not replicate with one child per generation but as many children as it earned money to buy. A graphical representation of these differences can be found in figure 5.6

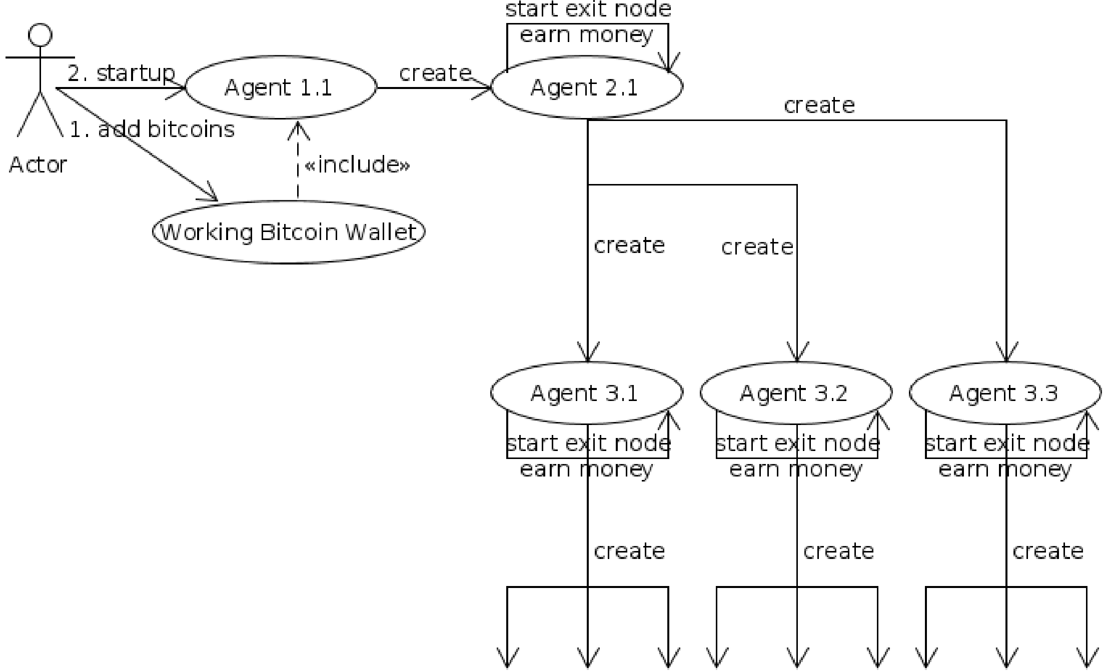
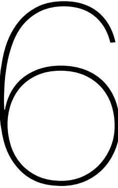


Figure 5.6: The third prototype.

As this would implement all functionality that was originally part of the agreement, the 3rd prototype was meant to be the finished product exempt perchance some small last minute requests by the client, however as the second prototype did not work at its live test. The third prototype has not yet been finished at the time of writing. though most of its code exists, this is because the market made by the other BEP group is at this moment not yet functional and we have not yet ran the test for prototype2.



Testing and SIG

In this chapter will be discussed the basic approach to testing, The SIG feedback that was received and how we approached their feedback.

6.1. Testing approach

Testing was mostly done using an automated unit test approach. This combined with integration tests allowed us to see both whether components functioned and whether their interaction worked. The idea was that before a PR would be accepted into master the tests would be verified working. Though this did not always happen.

6.2. First SIG feedback

The feedback given by SIG gave us 3,5 stars in their maintainability model as can be seen in appendixA.2. Which is a lightly above average score. We lost points mainly due to high unit size and unit complexity. The placeOrder function in the ZappiehostBuyer class was specifically mentioned as being overly long and complex. They concluded with that while it was good that we had test code, we should focus on adding more tests.

6.3. Reaction to SIG feedback

As a reaction to the SIG feedback all VPS Buyers (which all had the problems described for Zappiehost-Buyer) where refactored and functionality was split off into smaller more simple functions. Other parts of the code were also revisioned as necessary. Furthermore we focused on -where applicable- splitting of functionality in a testable way and added tests for these functions. We then maintained quality by checking complexity and size in pull requests that were made.

6.4. second SIG feedback

The final SIG feedback noted how we, even though the size of the project was significantly increased. The maintainability had improved. as can be read in **??**

7 Conclusions

Throughout the project, many different problems have presented itself. Some of those forced us to compromise on our goals. Others problems were beyond our control and made it hard for us to adapt to. Most of the problems have been properly addressed and solved, and we've ended up with a product that, although incomplete because of time limitations, has almost all the required functionality implemented into it.

Our system is - in theory - capable to fully autonomously function as a Tribler exit node, earn Multichain, sell the Multichain for Bitcoin on the market, order a Virtual Private Server online using the earned Bitcoin currency, retrieve the SSH login information, reproduce itself onto the new VPS with some genetic mutations. All these functions have been independently tested, and prototype2 was scheduled to verify most of these parts and that they work together, but unfortunately we have not gotten around to running this trial (see section 5.3).

Prototype 3 (see section 5.4) was supposed to be the final product. Although we have reason to believe that our code should in theory be ready for a third prototype, in practice this prototype will likely not yet work. Since no actual live testing was done of the market (the market was not finished in time to do these tests properly)

7.1. Challenges

There have been many challenges on our path to develop the final product that we had initially envisioned. The four biggest challenges will be summarized here.

7.1.1. Implementation of the Decentral Market

During our project, another group was developing a decentral marketplace. This is quite an ambitious project, because it is a relatively unexplored field in computer science. For us, this added difficulty, because we had barely any idea what to expect. However, the real challenge came in trying to integrate a product that does not exist yet. As the market was being developed at the same time as our system, we could not use the marketplace yet, or test if our code was correct. We attempted to make it possible to use their market in our system by agreeing upon an API, then mocking the market. Agreeing upon an API turned out to be more difficult than expected, but we succeeded and used the API that was agreed upon to mock the market. This allows us, to at least be able to write and test our code that uses the market on our end. When the market is finished, we should only have to replace the market's mock with the real market and it should work correctly.

7.1.2. Tribler exit node

Starting up the exit node also turned out difficult because of the ecosystem of Tribler being very closed. Although the code is open source, it is barely documented and commented. And people that actively worked on Tribler could usually not answer questions because they did not know it themselves. In the end we got a version working when we were told Tribler was going to be revamped into a daemon based system, effectively killing any known way for us to communicate with Tribler, use the market and earn multichain. In the end since this problem happened towards the end of our production cycle we

simply switched to the decentralized market repository, which had also not implemented this new way of communication. So that we could at least stay compatible with both systems.

7.1.3. Autonomously buying VPS

Another difficult part of the project was making the system able to autonomously buy new VPS' online. While there are some VPS providers that provide an API for this purpose, we could not find any VPS providers that accept Bitcoin that have such an API. This means we had to use tricky means to be able to order VPS' automatically: we used web browser automation to have our system actually visit VPS providers' websites, fill in their forms, execute the payment, and extract the bought VPS' SSH login information. Usually, the VPS providers want to prevent bots from ordering from them, and some VPS providers banned our IP address with some of our early attempts. We had to come up with all kinds of measures, like finding out the machine's own country of origin (see section 4.1.2) and creating temporary email accounts (see section 4.1.3) to succeed at making our system able to purchase VPS' online.

7.2. Unreliability of the system

Even if our project is finished, our system is very unreliable. There are many reasons why our automated system could break. Here are some examples, just to name a few of a long list of possible reasons:

- If the VPS providers change the HTML on their order pages, the system will not be able to order anymore, and the system will be unable to reproduce
- If any of the components the system depends on, like for example Tribler, gets updated and becomes incompatible with the system, the system will likely break or at least stop functioning fully
- If the system won't earn enough Bitcoin to reproduce, the system will die out
- If the VPS providers learn to detect our system and prevent registration for them, the system will be unable to reproduce

Even apart from these ways the system could break, there is another reason why our system is just a proof of concept. The system earns bitcoin from selling Multichain on an online market. Currently, Multichain has no proven value. The only reason why selling the Multichain works, is because an artificial need for Multichain will be created by having an auto-buy bot buy Multichain with Bitcoin. As long as people will not want to spend Bitcoin on Multichain actively, the system in its current form will never work.

7.3. Recommendations

During our research some choices were made that in hind sight could have been done better. If different decisions were made our product might have been better than it currently is.

One of these examples would have been the use of Docker[4]. Docker runs it's programs in a virtual environment guaranteeing that as long as docker works on a system and the program works on a docker system. it will work on all Docker systems. At the time it seemed unnecessary overhead since the installation was already working, however when implementing multiple VPS' it appeared that while two VPS' might run the same version of an operating system they might still behave differently. Had we had a docker script in place we would only have to focus on getting docker to work. Since the automation script would always work, since it runs on docker, in effect creating a more stable environment to work/test on.

A different factor that would have improved the quality would have been implementing less VPS, because they drained a lot of manpower that could have better been spent constructing a more rigid program. Since the VPSBuyers were already modularized more could have been written at a later point with no problems. And the initial 3 were already plenty for a proof of concept.

A last point is that at this time automation of buying is simply too unreliable, while some buying through api's exists for dollar based purchases [14] these do not accept bitcoin, and still require identification through a (made by hand) user account and are thus unsuitable for our project. However our implementation through automation heavily depends on a website remaining the same, which during the course of the project for some vps' did not happen which in at least 1 case broke the respective buyer, although this was obviously fixed.

Bibliography

[1] blockchain. Bitcoin grootte van de blockchain. https://blockchain.info/charts/blocks-size, . Retrieved on 14-06-2016.

[2] blockchain. Blockchain wallet. https://blockchain.info/wallet/#/, . Retrieved on 14-06-2016.

[3] Coindesk. Mt. gox bankruptcy trustee issues new details on creditor reimbursement. http://www.coindesk.com/mt-gox-bankruptcy-details-creditor-reimbursement/. Retrieved on 14-06-2016.

[4] Docker. Docker - build, ship, and run any app, anywhere. https://www.docker.com/. Retrieved on 16-06-2016.

[5] Electrum. Electrum bitcoin wallet. https://electrum.org/#home. Retrieved on 15-06-2016.

[6] Python Software Foundation. ipgetter 0.6 : Python package index. https://pypi.python.org/pypi/ipgetter/0.6, . Retrieved on 16-06-2016.

[7] Python Software Foundation. python-geoip-geolite2 2015.0303 : Python package index. https://pypi.python.org/pypi/python-geoip-geolite2, . Retrieved on 16-06-2016.

[8] Python Software Foundation. temp-mail 0.2 : Python package index. https://pypi.python.org/pypi/temp-mail, . Retrieved on 16-06-2016.

[9] Maxmind. Ip geolocation and online fraud prevention | maxmind. https://www.maxmind.com/en/home. Retrieved on 17-06-2016.

[10] OffshoreDedi. Swiss offshore vps. https://offshorededi.com/swiss-vps/. Retrieved on 15-06-2016.

[11] Selenium. Selenium-web browser automation. http://www.seleniumhq.org/. Retrieved on 29-04-2016.

[12] Sharkservers. Sharkservers. http://www.sharkserve.rs. Retrieved on 16-06-2016.

[13] THCServers. Vps hosting - affordable and configurable vps plans | thcservers.com. https://www.thcservers.com/vps-hosting. Retrieved on 15-06-2016.

[14] tilaa. Api reference - tilaa. https://www.tilaa.com/en/api/docs. Retrieved on 16-06-2016.

[15] Tribler. Tribler - privacy using our tor-inspired onion routing. https://www.tribler.org/, . Retrieved on 21-04-2016.

[16] Tribler. Dev update: Exits nodes by tribler team: 6 terabytes / day - general - tribler discussion forums. https://forum.tribler.org/t/dev-update-exits-nodes-by-tribler-team-6-terabytes-day/3508, . Retrieved on 16-06-2016.

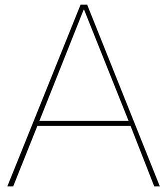
[17] Tribler. Tribler. https://github.com/Tribler/tribler, April 2016. Retrieved on 17-07-2016.

[18] Tsukiji. Tribler/decentral market. https://github.com/Tribler/decentral-market. Retrieved on 29-04-2016.

[19] X.org. Xvfb. https://www.x.org/archive/X11R7.7/doc/man/man1/Xvfb.1.xhtml. Retrieved on 16-06-2016.

[20] Yourserver. Yourserver. https://www.yourserver.se. Retrieved on 16-06-2016.

[21] Zappiehost. Home | zappie host. https://www.zappiehost.com/. Retrieved on 29-04-2016.



Appendixes

A.1. Infosheet

Autonomous Self-replicating Code
Tribler
Github Page: https://github.com/Skynet2-0/Skynet2.0
Presentation Date: 24 June 2016
Description:

Tribler, an anonymous file-sharing program, is for backwards compatibility with torrents heavily reliant on so called ”tribler exit nodes” to form a non-anonymous bridge between the anonymous Tor network and the non-anonymous regular internet. However, due to the non-anonymous nature many people are unwilling to run these exit-nodes on their own machine. Because of this lack of exit nodes, Tribler requested us if a program could be written that was capable of procuring servers by itself and initialize these as Tribler exit nodes. It could then make money selling upload capacity and replicate itself to new systems. Early on it became clear that this could not become more than a highly experimental system due to several critical parts simply not offering reliable API's. To give an example, ordering VPS' on the websites of VPS providers had to be fully automated. If the implemented VPS providers change their website layout, our system already loses functionality. A big challenge during the project was making the different parts reliable, since no mistakes could be tolerated, as the system has to function autonomously. Many quirks of softwares showed themselves when implemented (such as a program that did not function properly the first 5 seconds after installation). As a final remark, certain features have been left out for a future version because of a limitation in time.

Members of the project team:
Member 1:
Name: Niels Bakker
Interests: Artificial Intelligence, scaling and dynamic systems, anonymity
Role & contribution: Developer, Scrum Master, prototypes/integration, initial exit node, DNA, test server functionality, Wallet functionality
Member 2:
Name: Rob van de Berg
Interests: Artificial Intelligence, Autonomous Self-Sustaining Systems.
Role & contribution: Developer, implemented the autonomous purchasing of new VPS', initiated negotiations on the Market API
Member 3:
Name: Stefan Boodt
Interests: Operating Systems, Security
Role & contribution: Developer, decentralized market integration, initial ssh, initial installation
All team members contributed to the final report.

client

Name: Egbert Bouman
Affiliation: Active at Tribler, works at TU Delft

supervisor
Name: Johan Pouwelse
Affiliation: Professor at TU Delft, active at the Distributed systems group, founder of Tribler

contact person
Niels Bakker - ncb21992@hotmail.com

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

A.2. 1st SIG Feedback

This is the initial SIG feedback provided to us. It is unfortunately in dutch, an english explanation and approach can be found in the relevant chapter.

```
De code van het systeem scoort 3,5 ster op ons onderhoudbaarheidsmodel,
wat betekent dat de code licht bovengemiddeld onderhoudbaar is.
De hoogste score is niet behaald door lagere scores voor Unit Size en Unit Complexity.

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.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld
complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere
stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen
en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe
methoden ook naar voren als de langste methoden, waardoor het oplossen van het eerste
probleem ook dit probleem zal verhelpen.

In jullie geval is de methode ZappiehostBuyer.placeOrder zowel erg lang als erg
complex. Het maakt de code aanzienlijk beter leesbaar als je de deelproblemen
uitsplitst naar aparte methodes. Commentaar geeft vaak een goede indicatie waar dat zou
kunnen. Het blok ”# Generate a password” kan je bijvoorbeeld duidelijk uitsplitsen naar
een nieuwe methode genearte_password (je kunt die functionaliteit op die manier ook
veel beter los testen). Iets soortgelijks kan ook voor de selectie van landen. De check
om te kijken of het land wordt ondersteund kun je beter splitsen naar
is_state_selection_supported(country) of iets dergelijks.

Tot slot is het positief om te zien dat jullie ook testcode hebben geschreven, maar de
verhouding tussen productiecode en testcode kan nog wel wat beter. Hopelijk lukt het
nog om daar in het vervolg van het project nog aan te werken.
```

A.3. 2nd SIG feedback

This is the second SIG feedback provided to us. It is unfortunately in dutch.

```
In de tweede upload zien we dat de omvang van het systeem flink is gestegen. De score voor

De score voor Unit Size en Unit Complexity (de kritische punten van de vorige upload) is i

Verder is het goed om te zien dat jullie testcode hebben geschreven. De verhouding tussen

Uit deze observaties kunnen we concluderen de aanbevelingen van de vorige evaluatie gedee
```

A.4. API agreement

```
in community.py:

order_book property for accessing the order book class

create_ask(self, price, quantity, timeout):
price - (float) bitcoin price in btc (precision to 1/10000) roughly 3 cents
quantity - (float) multichain bytes in MB 10^6 (precision to 1/10000) roughly 100 bytes
timeout - (float) time from builtin python time format when the order must expire
returns - Order object that includes all the information

create_bid(self, price, quantity, timeout):
price - (float) bitcoin price in btc (precision to 1/10000) roughly 3 cents
quantity - (float) multichain bytes in MB 10^6 (precision to 1/10000) roughly 100 bytes
timeout - (float) time from builtin python time format when the order must expire
returns - Order object that includes all the information

get_multichain_balance(self)
returns - (Quantity object) the current amount of multichain in ~MB (10^6 byte) and bytes both

in orderbook.py:

bid_price(self):
Return the price an ask needs to have to make a trade
:rtype: Price

ask_price(self):
Return the price a bid needs to have to make a trade
:rtype: Price

bid_side_depth_profile(self):
format: [(price (Price object), depth (Quantity object)), (price, depth), ...]
Returns the list of bids in the format provided

ask_side_depth_profile(self):
format: [(price (Price object), depth (Quantity object)), (price, depth), ...]
Returns the list of asks in the format provided
```

A.5. Original project description

This was the original description, parts of the assignment were changed over time in discussion with our client

Autonomous Self-replicating Code Project description

You will create an Internet-deployed system which can earn money, replicate itself, and which has no human control. It has adapted for use in latex to improve readability.

In the past humanity has created chess programs that it can no longer beat. The distant future of an omniscient computer system that on day chooses to exterminate humanity in the Terminator films is the not focus of your project. You will create software that is beyond human control and includes breathtaking features such as earning money (Bitcoin) and self-replicating code (software buys a server+spawn clone).

Earning money consists of helping others become anonymous using the Tor-like protocols developed at TUDelft and our own cybercurrency designed for this purpose, called Multichain coins. You Python software is able to accomplish the following:

- Earn income in a form of cybercurrency (existing code, see below)

- Sell this cybercurrency on a market for Bitcoins (another BEP team)
- Buy a server using Bitcoins fully automatically
- Login to this Linux server and install itself from the Github repository
- BONUS: As a sign of life its sends a birth certificate to its makers (using postcardservices.com, modernpostcard.com)

The software also should be able to have a simplistic form of genetic evolution. Key parameters will be inherited to offspring and altered with a mutation probability. For instance, what software version of yourself to use (latest release?), what type of server to prefer buying (quad core, 4GB mem, etc), and if you offer Tor exit node services for income. Bitcoins owned by TUDelft will be used to bootstrap your research.

WARNING: this project is challenging and recommended for students experienced in software development and/or honor students.

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-file-sharing-network-invincible.html, http://http://www.ee.princeton.edu/events/anonymous-hd-video-streaming-and-reputations , http://news.harvard.edu/gazette/story/2007/08/creating-a-computer-currency , http://tweakers.net/nieuw tribler-gebruikt-

onderdelen-tor-voor-anonieme-downloads.html, http://http://www.elsevier.nl/Tech/blogs/2014/12/Johan-Pouwelse-Tribler-Mijn-ideaal-is-mensen-aan-de-macht-1674399W/

Multichain coins documentation: http://repository.tudelft.nl/view/ir/uuid%3A59723e98-ae48-4fac-b258-2df99d11012c/