jamk.fi

Designing a payment system to the RGCE network

Timothy Sikorski

Bachelor's thesis
March 2017
Information and communications technology
Degree programme in Information Technology

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Sikorski, Timothy	Bachelor's thesis	16.03.2017
		Language of publication: English
	Number of pages 78	Permission for web publication: x

Title of publication

Designing a payment system to the RGCE network.

Degree programme

Bachelor's Degree in Information technology

Supervisor(s)

Häkkinen Antti, Kotikoski Sampo

Assigned by

Vatanen Marko
Jyväskylä Security Technology

Abstract

The thesis was implemented in the JYVSECTEC facilities inside the JAMK University of Applied Sciences. The purpose for the thesis was to design and implement a working payments system to the private RGCE network and verify that it works as intended. The original BTC protocol source code was used as a basis and was modified to a version fit for the environment.

The chosen Bitcoin version was 0.10 but was later changed to 0.9 due to a few features not available in 0.10. As for the operation system, Debian Jessie was chosen.

The network was created with four computers, from which two were acting as servers and the other two as normal participants of the network. The functionality of the network was verified in the RGCE network.

As the end result, a working network for payments was achieved with a command line operated software which is able to generate digital currency and process transactions from one wallet to another.

Keywords/tags (subjects)

Bitcoin, Linux, Debian, Blockchain, Network

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Sikorski, Timothy	Opinnäytetyö, AMK	16.03.2017
	Sivumäärä 78	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Maksuliikennejärjestelmän suunnittelu RGCE-verkkoon

Tutkinto-ohjelma

Tietotekniikan koulutusohjelma

Työn ohjaaja(t)

Häkkinen Antti, Kotikoski Sampo

Toimeksiantaja(t)

Vatanen Marko
Jyväskylä Security Technology

Tiivistelmä

Opinnäytteen toteutuspaikkana toimi Jyväskylän ammattikorkeakoululla toimivan JYVSECTEC: in toimipiste Lutakossa. Työn tavoitteena oli suunnitella lohkoketjuteknologiaa käyttäen maksuliikennejärjestelmä yksityiseen RGCE-verkkoon sekä todeta sen toimivuus. Pohjana käytettiin Bitcoin protokollan koodipohjaa josta muokattiin ympäristöön sopiva versio.

Bitcoin versioksi valittiin aluksi versio 0.10 mutta lopulta vaihdettiin versioon 0.9 muutaman ominaisuuden vuoksi, jotka oli poistettu versiosta 0.10.

Käyttöliittymäksi valittiin Debian Jessie. Verkko toteutettiin käyttäen neljää eri konetta joista kaksi toimi palvelimina ja toiset kaksi normaaleina verkkoon liittyvinä asiakaslaitteina. Maksuverkon toimivuus todennettiin RGCE-verkossa.

Työn lopputuloksena saatiin toimiva verkko maksuliikennettä varten sekä komentolinjalla pyöritettävä ohjelma, jolla voidaan generoida digitaalista valuuttaa sekä suorittaa siirtoja lompakosta toiseen.

Avainsanat (asiasanat)

Bitcoin, Linux, Debian, Blockchain, Network

Muut tiedot

Abbreviations

BTC	Bitcoin
DNS	Domain Name System
GBT	Get Block Template
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IP	Internet Protocol
ISP	Internet Service Provider
JYVSECTEC	Jyväskylä Security Technology
MD	Message Digest
NTP	Network Time Protocol
OS	Operating System
POW	Proof of Work
RGCE	Realistic Global Cyber Environment
RIPE	RACE Integrity Primitives Evaluation
RPC	Remote Procedure Call
SHA	Secure Hashing Algorithm

Contents

Figures

1 Baseline of the thesis

1.1 The client

JYVSECTEC is a project started in 2011 with the co-operation of JAMK University of Applied Sciences to address the challenges of today. The agenda of the project has been to create an environment for research and development in Central Finland to develop a national and international network of co-operation between companies and independent actors. The RGCE network has been able to provide an environment in which research and development are separate from the public network and offer a space where different cybersecurity scenarios and different pre-implemented systems can be tested safely. Piippukatu 2 is the address of the JAMK Lutakko campus, where JYVSECTEC resides as well. (JYVSECTEC 2016)

1.2 The objective

The objective is to create a working payment system to the private RGCE network by utilizing blockchain technology. BTC protocol will be used as the basis for the code but will be altered when necessary. One of the requirements of the network is to have a small number of computers part of it to make the network more secure, in this assignment four to six computers will be adequate enough.

What is also needed, is to be able to shut down the network and reset it, if necessary. Adding a new node to the network should be made relatively easy and quick. The implementation of the project will be written in the form of a guide to make it easier to replicate the process.

1.3 The production environment

The production environment will be in the RGCE network. The RGCE network is a part of JYVSECTEC and was made to simulate the real internet, to function as it would in the public network while being a private one. The BTC protocol will be implemented in this private network, which remains separate from the public network. This assures that the implementation will remain free of external threats and the focus can remain in testing the network instead of securing it. The machines on the network will be created virtually and operated on a VMware based platform.

The RGCE environment is made to be very similar to the public network with multiple ISP: s, DNS servers, NTPS: s and different services such as news sites and social media sites etc. The service providers have automatically generated traffic from customer networks to simulate the operations of the public network. (JYVSECTEC 2016)

2 Data encryption and key exchange

To understand what the BTC protocol and the blockchain is; breaking the idea down to smaller subcategories is needed to make the understanding of the concept easier. It is also required to have a basic idea of what public key cryptography, digital signatures and cryptographic hashing is. A short introduction of the three previously mentioned subjects will be made in the next few chapters to make it convenient for the reader.

2.1 Public key cryptography

In symmetric-key cryptography, both parties involved in the transaction of data agree upon a common key that is used to encrypt and decrypt data. The problem in this form of cryptography is that the key has to be shared between both parties in an insecure manner. Public key cryptography has solved this problem by creating two separate keys for encryption and decryption. Data is signed with the receiver's public key so that only the receiver's private key can decrypt it.

The disguise for the sent information is done with Encryption and decryption. The sender does the encryption, which transforms the information into a form in which it is not understandable without decrypting it. The receiver of the information has to decrypt it before it can be read. While the data is being transported to the receiver, if a third party is listening to the connection, the message cannot be understood without the appropriate tools to decrypt the message. (Lackey 2012)

This is also where the term "digital signatures" come in to play, more on the subject in chapter 2.2. Figure 1 exhibits the basic principal of the exchange of data using the public key cryptography process.

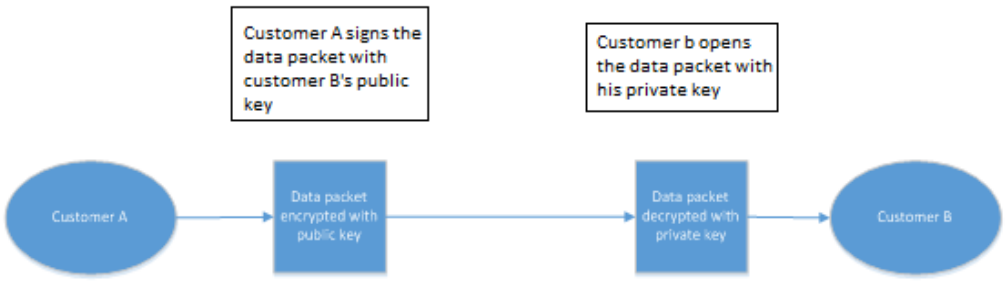


Figure 1. Public key process

2.2 Digital signatures

A digital signature ensures that any digital document signed with a signature can be verified to be from a trusted source and not altered by a third party. This is also done with websites to assure users that the site is verified by a trusted source and safe to use. (HowStuffWorks 2016)

To ensure authentication of the digital signatures, different encryption methods are used. When encrypting, the data sent is changed to a form in which it is not readable unless the receiving party decrypts it. When authenticating, the source of the information is verified. These two processes form the basis for digital signatures. (HowStuffWorks 2016)

When ensuring that the digital signature is valid, the public key of the signature provider is used to test if the site is valid. Since the signature was built with the private key, the public key, which is a pair to the private key is the only key which can verify that the signature is valid.

Hash values are the basis of the keys. The keys value is calculated using a hashing algorithm, computed from a base input number. The original number to create the hash is impossible to figure out from the hash without knowing the data for the hash creation. (HowStuffWorks 2016)

2.3 Cryptographic hashing

The hash algorithm completes mathematical calculations that include the insertion of random data as input and produces an output with a pre-determined data size. When given the same data for input, the output also remains the same.

Data given as an input is usually called a message, the output of these messages are message digests. Virtually any kind of data can be defined as a message, for example text in an e-mail, files in binary mode or network packets. An example of a hash function:

According to Silva, when a hash function accepts a message, it outputs a digest with a fixed data size of one bit. The hash function then returns one of two numbers depending on the output of the message digest. If the digest has an even number of characters, the return is zero. When the digest has an uneven number of characters, the return is one. (Silva 2003)

Hash functions have the same feature, the fact that it is impossible to try to figure out the original input of the hash function from the output of the message digest. For example in the previous example, knowing that the output was one, only tells that the digits in the input are uneven. It would be impossible to figure out the original input before the digest because the number could be numbers, a string of text or anything between. The only known thing about the data is that the length of the data was uneven in numbers. It is impossible to try to determine what the message was from the output "1". Because hash functions are designed to work this way, in one direction, it makes the deduction from the output impossible. (Silva 2003)

Cryptographic hash functions are much stronger than plain hash functions. The difference between the two is the fact that collisions are much more difficult to produce. Collisions happen when two different inputs before the digest produce the same output after digesting. The example given earlier is not a cryptographic function because the output would be the same regardless of what data were inserted, as long as the number of characters was uneven. (Silva 2003)

Cryptographic hash functions can be used to determine if a file has been altered, for example when sending data, a message digest of a binary file is usually given to the

receiver. The digest can be re-calculated from the binary files baseline digest and compared to the original message digest to see if it has changed. If changes has occurred, the data sent has been either corrupted or changed during the transfer. (Sptizner 2016)

To get a collision from an altered file is extremely rare in cryptographically digested hash functions, it is so unlikely that if the digest matches the file is most probably unchanged. With the knowledge of how the function works, it can be said that cryptographic hash functions can be used with a fair amount of certainty to validate the integrity of files. (Silva 2003)

The Bitcoin protocol uses sha-256 as its base hashing function. When a shorter hash is required, for example when a Bitcoin address is created, a hash function called RIPEMD-160 is used.

3 Bitcoin and the blockchain

3.1 Introduction to Bitcoin

Bitcoin was designed to be a version of digital cash which would allow payments to be sent over a network directly between two points without a trusted third party (Banks, Paypal etc.). It is essentially a trustless network where user's payments and wallet balances are recorded on a public ledger verified by everyone on the network who are hosting nodes.

A total of 21 million BTC: s will be generated with BTC protocol over a period of a 100 years. For every found block, 50 coins are rewarded to the node which found the block. Every four years the algorithm halves this reward, bringing the reward of 50 coins to 25. The difficulty of the network (how hard it is to find an acceptable hash of a block) readjusts itself approximately every two weeks and starts from difficulty one. It changes to a higher number the more hash power the network has.

This is one of the reasons why in the public network, small time miners find it hard to receive any rewards due to the fact that their single machine is overpowered by the superior hashing power of the mining pools and similarly powerful mining rigs. The chance to find a block with an average hash rate GPU is not likely nor cheap when the cost of electricity is taken into account.

3.2 Transactions

Digital currency is defined as a chain of digital signatures, according to Nakamoto. In every transaction, a signature from the previous owner and the public key of the next owner are added to the end of the coin. When the signatures are added, the ownership can be verified when the signatures are compared. (Nakamoto 2008)

Figure 2 exhibits how the previous examples come together in the Bitcoin protocol, owner 1 signs the data with a private key while hashing the data. Owner 2 can check the validity of the data by decrypting the data with the owner 1's public key. If the data is intact and checks out, it is included in the next block of transactions.

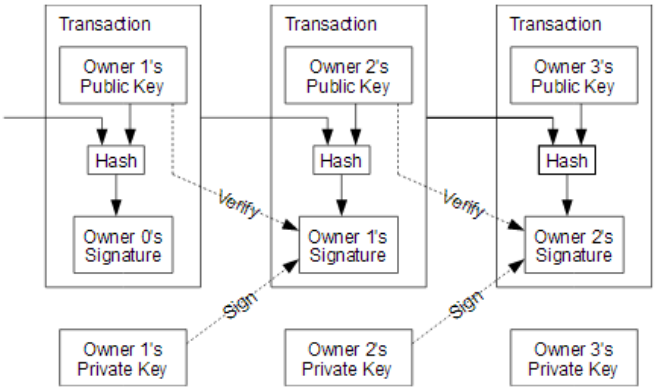


Figure 2. Transactions. (Nakamoto 2008)

This type of handling payments is efficient and reliable but needs something to prohibit users from double spending; the network has to be aware of the previous transactions to confirm account balances.

3.3 Blockchain

In a timestamp server, hash function is performed to a block of items and advertised to the participants of the network to be validated. When the hash has been timestamped, it proves that data has existed at a certain time since it had to be for it to be able to change in to a hash. When the previous timestamp of a block is included to the next, it creates a chain of timestamps and thus reinforces the position of the blocks in the chain. An example of how the timestamp includes itself to the next block can be viewed in figure 3. (Nakamoto 2008)

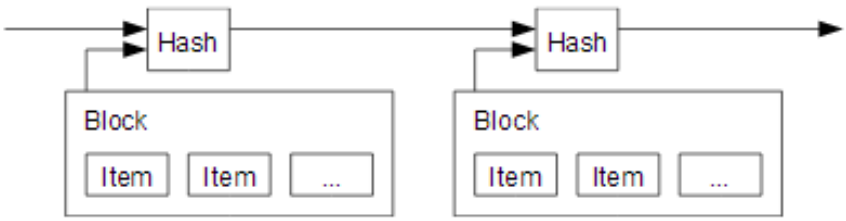


Figure 3. Timestamp server. (Nakamoto 2008)

Blockchain is basically a timestamp server, each computer on the network is informed on changes on the blockchain and collectively verify the transactions on the network. It is very difficult to try to remove a block from a blockchain ledger once it has been recorded there. When an addition has been made, the participants of the network run an algorithm to try to verify that the transaction is valid, if it is accepted it is added to the history of the blockchain and broadcasted to everyone. The removal of a block would require the whole network (or at least the majority) to agree to the change. Figure 4 shows an example of a block hash linking to the next. (Norton 2015)

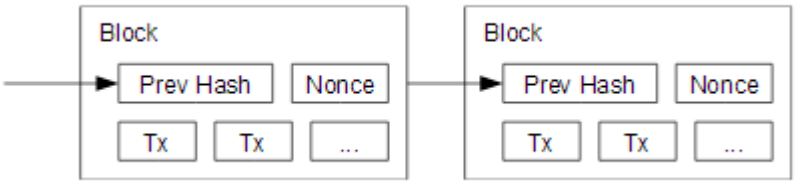


Figure 4. Block chaining. (Nakamoto 2008)

3.4 SHA-256 and RIPEMD-160

SHA-256

SHA-256 was created by NSA back in 2001, "SHA" is a cryptographic hash function designed to hash data to a form of numbers and digits, which can be compared to an original hash of the same file to validate the integrity of the file, for example before and after sending it to someone. SHA-256 is a member of the SHA-2 group, an improved version compared to the SHA-1 group. The number 256 is an identity number for the function and it is defined by the number of bits used in each word in the hash, in this case 32 is used. (Poiroux 2015)

SHA-256 is a popular version of the SHA-2 group and is used mainly to verify software packages, for example for LINUX, or to verify transactions in cryptocurrency chains. Besides BTC, many alternative coins use SHA-256 to calculate POW transactions which tells that it is still secure and powerful enough to prevent collisions. (Poiroux 2015)

The SHA-256 algorithm works by padding a message into 512-bit blocks, the same way as SHA-1. The padding consists of multiple 512-bit long data chunks which are then parsed in to 512-bit blocks. $M^1; M^2;:::; M^n$. The process of modifying the message blocks is done one at a time, a fixed hash value H^0 is inserted and computed sequentially. (IWR)

$$H^1 = H^{(i-1)} + CC_{M(i)} (H^{(i-1)})$$

In the function above, C is the function for the compression of SHA-256 and + means the addition of the word-wise mod 2^{32} addition. $H^{(N)}$ represents the hash of M. The message block size in SHA-256 consists of 512-bit blocks and has a 256-bit hash value. It can be considered as a 256-bit cipher algorithm, which uses the message block as a key to encrypt the hash value. (IWR)

RIPEMD-160

RIPEMD-160, designed by Hans Dobberto, is a cryptographic hash function used as a replacement for a 128-bit functions such as; MD4, MD5 and RIPEMD. Ron Rivest developed MD4 and MD5 for RSA Data Security, the RIPEMD was created as a framework for an Eu project called RIPE. (Bosselaers 2012)

Compared to RIPEMD, RIPEMD-160 is a stronger version with a longer 160-bit hash and is projected to be secure for the next decade or more. The algorithm is tuned for 32-bit processors since it is estimated that the processor type will still remain important in the future. Experience gained from the previous versions of hash functions are taken in to consideration when designing these new hashing algorithms. (Bosselaers 2012)

3.5 Proof of work

In cryptographic hash functions, the randomness of the hashing output is a sought quality. POW takes an advantage of this during the verification process of transactions. The randomness of the hashing output makes it very unlikely that the hash

creates collisions, it is almost impossible to create the same outcome from different data before hashing. (Bitcoin Project)

To create a block and to prove that work has been done, a hash of a block header has to not exceed a certain length. For example, if a requirement for a valid hash is $2^{256} - 1$, the calculations are accepted if it is less than 2^{255}. (Bitcoin Project)

With the previous example, an acceptable hash is found on estimate every second try. The success rate of finding a hash is possible to estimate and in the BTC protocol this requirement increases in a linear fashion. The lower the threshold is set, the more hashing attempts have to be made for an acceptable one. (Bitcoin Project)

Only when the hash meets the requirements of the consensus protocol, will the block be added to the public ledger. The difficulty for these requirements is calculated approximately every two weeks. (Bitcoin Project)

- If the time to generate 2016 blocks is less than two weeks, the difficulty increases to match the generation rate requirements of the blocks so that it remains at two weeks at the current speed.
- If more than two weeks were needed to create 2016 blocks, the difficulty is decreased proportionally in the same manner as when increasing.

To succeed in modifying the blockchain, it is needed to have the hashing power of at least 51% hashing power of the entire network. This is because of the fact that each hash of a header of a block has to be under the target threshold and because each block is linked to the previous block. To make a modified block acceptable, the majority hashing power of the network must be on the accepting side of the block to consider it a valid addition to the chain. This is why 51% is needed to make this kind of a situation possible. (Bitcoin Project)

A wait for new transactions is not required for the obtainment of new hashes, this is because the header of the block has easily modifiable fields, like the nonce field. Only the block header of 80-bytes is POW hashed, so the inclusion of transaction data with a higher volume does not slow the hashing down. The addition of more transaction

data only has a requirement of calculating the ancestor hashes of the merkle tree again. (Bitcoin Project)

3.6 Network

The Requirements for running a successful BTC network:

1) New transactions should be broadcasted to other nodes.

2) Each node includes transactions in a new block.

3) Each mining node does POW to find a block.

4) Found blocks are broadcasted to the entire network.

5) A block is accepted if the reward is not already spent.

6) The accepted hash of a previous block is included to the next found block.

Participant nodes in the network consider that the longest chain is the one where work should be done to extend it. If for example two chains are broadcasted to a node, it will continue to work on the chain which was received first. The other chain will be saved to memory for the chance that it becomes the longer chain. If this happens and the saved chain becomes the longer one, the node switches to that chain. (Nakamoto 2008)

Occasionally a transaction might not reach every node on the network at the same time, but if it reaches enough nodes, it will be included in to a block eventually. Nodes can also miss block updates but the mistake is corrected when the node receives the next block update and finds out that a block is missing from the database. (Nakamoto 2008)

To make a secure and trustworthy network, it is important to note that the more participants on the network, the better. Decentralization requires that nobody has the control of the network (holds 51% of the networks hashing power). If this re-

quirement is not met, it is possible to plan an attack where the holders of the majority hash power decides which chain is the dominate one.

In an example scenario person A sends person B a BTC over the network. Everybody on the network confirms the transaction valid and broadcasts this to others on the chain. Person A holds majority of the hash power and decides that a time in the blockchain where the transaction did not occur is what should be supported and starts pouring hashing power in to it. With person A having the majority power of the separate chain, the chain will surpass the original main chain in length, thus replacing it. Now person A is able to spend the same BTC all over again.

The bigger the network, the more this scenario costs. With hundreds of thousands participants, the network is big enough that the cost of trying to achieve being the majority hash power on the network negates the benefits. Figure 5 shows an example of the scenario:

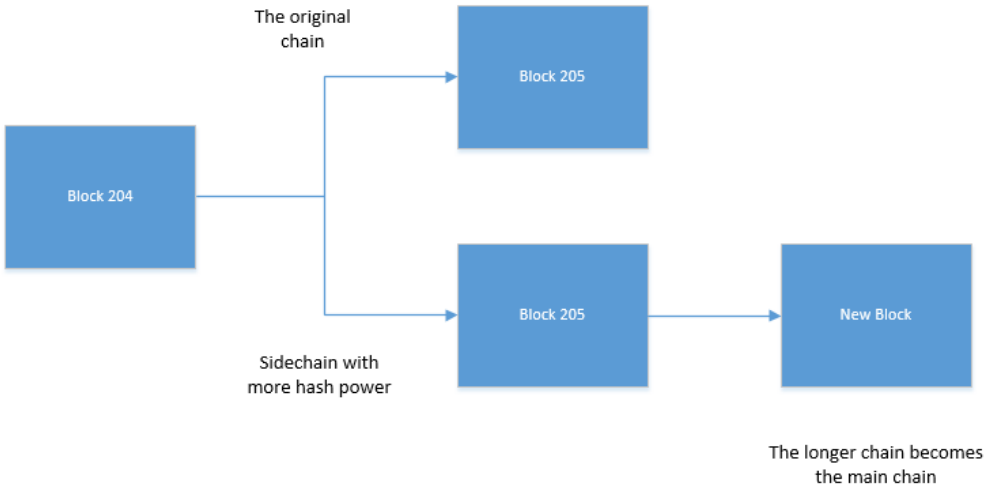


Figure 5. Longer chain wins

3.6.1 The block size issue

There has been a lot of debate on whether the block size of 1 Mb is too small for the amount of transactions happening on the network. More than one group argues that

the block size should be increased so that more transactions would fit per block instead of being transferred to the next available block.

At the moment if one wants to make a transaction, a moderately higher transaction fee is required to make the payment a priority for the miners to validate it quicker. This is arguably a big scaling problem as the BTC user base grows and more and more transactions have to be validated each second. Increasing the block size indefinitely would solve the short-term problem but would prove to be a burden on the network long-term.

It is argued, that increasing the block size only creates more stress and centralization to the network. When the block size is increased, miners might not get enough transaction fees as rewards through doing POW and choose not to continue. This leaves the big pools and the ones with more hashing power in charge of the network, decreasing the decentralization of the network. (Deschapell 2016)

A peak transaction speed of 47000 per second was achieved by the payment network Visa in the holiday season of 2013. Currently the average amount of transactions made per day is in the hundreds of millions. The issue with the block size in BTC is that the network supports currently only about 7 transactions per second with the 1-megabyte limit. It is said that the equivalent of the Visa transaction speeds would require the BTC blocks to be at the size of 8 gigabytes. This would add more than 400 terabytes of data to the blockchain per year, bloating it completely. (Poon, Dryja 2016)

This is not possible to achieve with the current technology since an average user would have to have so many hard drives that it would become quite impractical. Not many could afford it thus resulting in a centralization of the network. The security of the network would be compromised as a result and the fundamental idea of what BTC is would be ruined. (Poon, Dryja 2016)

To make a tradeoff between decentralization and the performance of the network would be a major blow to the whole concept, decentralization is a key component of what the BTC protocol represents, the ability of the people to be free of manipula-

tion of their currency and a chance to not have to put trust in a third party. The lightning network is a new solution which will be implemented on the BTC protocol soon, it solves most of the problems presented earlier, more can be read about it in the chapter 3.6.2. Figure 6 exhibits graphically the constantly rising blockchain size.

Figure 6. Blockchain size. (Kieren 2015)

3.6.2 The lightning network

The BTC network has always been a process of learning and progressing through continuous public testing by its users, both for good and bad. During the 8 years of operating, BTC has had only one major security issue which was patched fairly quickly by rolling back the network, which the participants of the network accepted without scrutiny. In august 2010, a flaw was found in a block header where somehow an additional of 92 billion BTC existed, while supposedly the protocol should have limited the amount of coins mined to 21 million during a prolonged time period. The error type was known as an "integer overflow error". In short, an unknown network participant had found a way to flood the code while creating massive amounts of BTC in the process. (National Vulnerability Database 2012)

The best way to create one of the most secure networks is to stress test it publicly. There are more great minds available through the internet than can be brought to-

gether privately, it is the ultimate test to have anonymous users try to find any exploit possible. It does not matter if it is done for one's own nefarious reasons or to make the network more secure, the end result is the same.

So now the time has come to address the problems with the block size. The lightning network was a solution explained in a whitepaper, written by two individuals called Joseph Poon and Thaddeus Dryja in early 2016.

The lightning network is a network of micro transactions built upon the BTC protocol. When doing a transaction between two parties, a micropayment channel is opened in which the transactions take place. Not every transaction is immediately made public but instead, only the final wallet balances of the payment channel participants are announced to the blockchain to be verified, reducing the majority of unnecessary data sent. (Poon, Dryja 2016)

Bitcoin could be capable of billions of transactions per day with a network of micropayment channels and just an average computer would be enough to process them. Transactions sent in a micropayment channel would enable to send large amount of funds while keeping the decentralization aspect of the network. (Poon, Dryja 2016)

The micropayment channels would enable two parties to send multiple transactions in the channel and to only inform the final balance of the wallets to the rest of the network when the transactions have been completed. The channels are not a separate network on bitcoin, the transactions consists of actual currency inside the blockchain, the update of the transaction is deterred to a later date trustlessly. (Poon, Dryja 2016)

Referring to the example in figure 7, a bidirectional payment channel has been opened between Janice and Chandler. Janice wants to send a transaction worth one BTC to a third party, in this case to Monica. A direct channel between Janice and Monica could be opened but is not necessary with the lightning network since Janice

can pay Monica through Chandler with the already opened channel between Chandler and Monica. (Wirdum 2016)

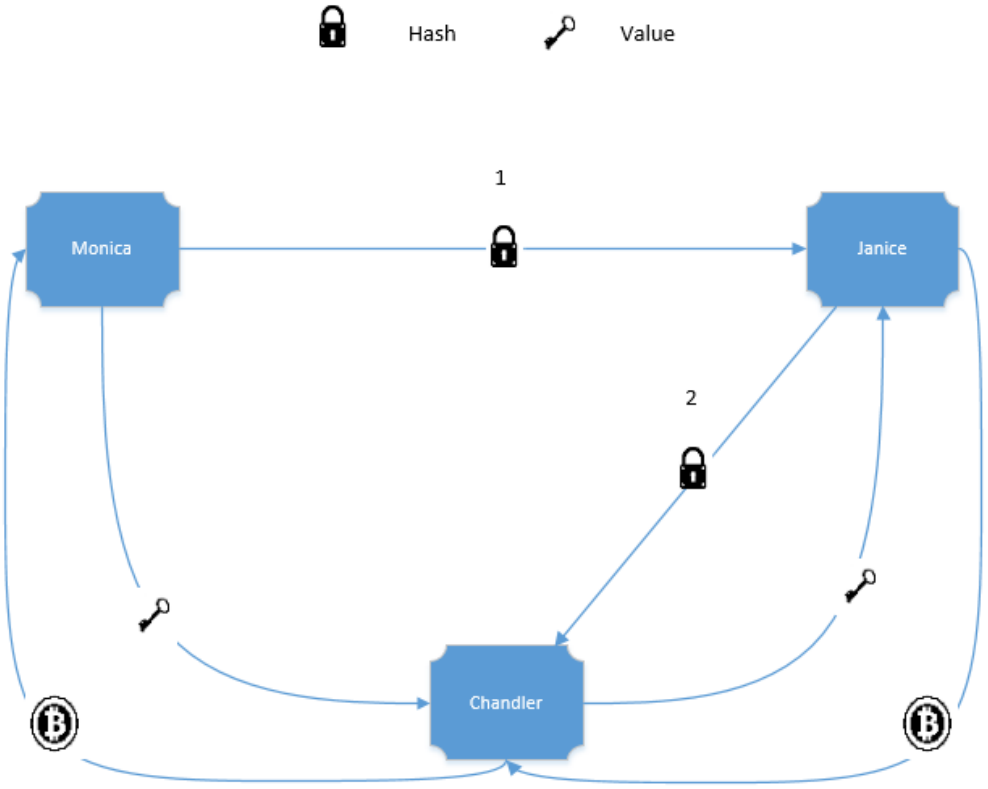


Figure 7. Lightning network

In this scenario, Janice would pay Chandler a BTC and then Chandler would do the same for Monica, this is done trustlessly, since the protocol ensures that Chandler will only receive the one BTC if one BTC is paid to Monica. This is done by Janice instructing Monica to give Janice a random hash number. Monica is also informed to exchange the original value for a BTC. (Wirdum 2016)

Janice receives the hash from Monica and instructs Chandler that he will receive a BTC if he provides the value only Monica has. Chandler pays Monica for the value and sends the value to Janice. Since Janice is informed of the exchange and that the value could only have come from Monica, it can be proven that Monica has received a BTC from Chandler. Janice can then safely pay Chandler one BTC. (Wirdum 2016)

It is not yet clear when this update will come in to effect, the development for the update has been going for more than a year. The network consensus is also a matter

to be taken seriously; some major mining pools have implied their unwillingness to contribute to the change. The update will be a welcome fix to the long-standing problem, provided that a network consensus is reached on accepting the change.

3.7 The Incentive for mining

A minimum of one transaction can be found in each created block, this is the reward for finding the block for the creator of the block. The reward for finding the block gives an incentive to the node to support the network by mining. It also brings more coins to the network since the distribution is not issued as in a traditional banking system. The operation is comparable to any real life scenario where precious metals are mined from the earth to be added to circulation. The cost for the mining is in the electricity and the hardware; machines wear out and use up energy. When no new coins can be brought in to circulation, due to protocol restrictions, the incentive will be provided through transaction fees. When there are enough found coins in circulation, the network will pay miners entirely in transaction fees making the currency noninflationary. (Nakamoto 2008)

In a case where an attacker had more than half of the networks hashing rate, a few things could be done. Transactions could be reversed or the network could be destabilized, this however would be less profitable than just using the hashing power to secure the network to acquire more coins. There would be no point in destabilizing the network if it would also hurt the perpetrators funds in the process. (Nakamoto 2008)

The incentive to mine new coins stays as long as the cost of electricity does not become more expensive than the coins mined while using said electricity. This is why most of the hashing power (60% of all new bitcoins mined) comes from a low cost country like China where mining farms are considerably cheaper to run.

Most hashing power comes from mining pools where people lend their computational power to a group pool which pays each contributor a share of the coins mined, based on the amount of hashing power each one has given to the service of the pool. In figure 8, the biggest mining pools hash contribution on the network.

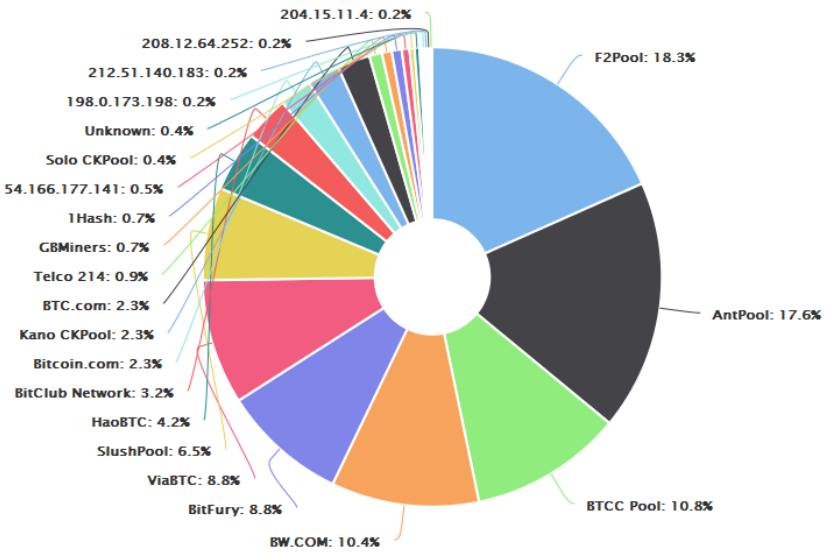


Figure 8. Mining pools. (Pollnow 2016)

3.8 Anonymity

BTC is not an anonymous currency as transactions are stored in a public ledger, which is known as the blockchain. Any Bitcoin address or transaction can be checked on the blockchain which makes the flow of the currency highly trackable. Bitcoin users' identities however, cannot be connected to a specific address unless it is used in conjunction with a payment with a third party.

For example, on a website a customer decides to pay for an ordered item with BTC. Personal details has to be filled in order for the package to be sent. In this instance, the seller has personal information about the buyer and can now connect the address from which the payment is sent to the buyer. The same scenario would happen if one wanted to sell a BTC on an exchange and send the acquired currencies to one's bank account. The third party participants in both of these exchanges will always have something to tie the BTC user to an address.

Multiple Wallets

Anonymity can be increased by using several wallets, as wallet addresses are harder to tie to an identity if multiple addresses are used. A recommended software for this

is MultiBit, it works on most operating systems and does not require to download the whole blockchains blockdata to be usable. (Cornwell, Atkinson 2013)

Mixing Services

Since the transactions in the blockchain are public, mixing services are used to blur the trail of payments. In short, a different transaction history for ones Bitcoin can be acquired when sent bitcoins is mixed with other Bitcoins and sent back in a random order. This however requires trust to the service; there are no guarantee that the service will honor the promise to return the coins sent. The service might also keep a transaction record, which would nullify the effort of trying to mix the payment trail. (Cornwell, Atkinson 2013)

eWallets or Online Wallets

Sending coins to a web hosted wallet can increase anonymity if the coins are mixed in to a big pool of coins from other users (for example in an exchange). The service must be an active one and the amount of coins must not be over 10% of the services own BTC balance. The anonymity however is completely dependent on the policies the web wallet host has set. Records of all the users could be possibly kept on a private ledger so there are no certain guarantees when using these services. (Cornwell, Atkinson 2013)

There are other cryptocurrencies which have a greater emphasis on user anonymity and privacy, such as Monero, Zcash and Dash. The difference between BTC and these alternative currencies is that different parts of the chain have been made anonymous, for example, the senders address for the receiver or the ledger itself. There are many frontrunners in the race towards complete anonymity in blockchain based transaction systems, the future will tell how each succeeds.

3.9 Node types and roles

Network nodes are the backbone of the BTC network; they validate the transactions and keep the history of all the changes made on the network. There are however,

differences between the node types and other participants throughout network (O'reilly Media, inc 2013);

- **Bitcoin core client** - The Bitcoin core client is the most basic and well-known client of them all. It has the wallet, complete blockchain database and a routing node on the BTC network.

- **Full blockchain node** - Has the full blockchain database and does network routing on the BTC network. It has no wallet.

- **Solo miner** - Can mine on the network, contains the full copy of the blockchain and is capable of routing on the network.

- **Lightweight SPV wallet** - Has a wallet and is capable of routing on the BTC network, it does not have the copy of the blockchain. Multibit is a wallet like this.

- **Pool protocol servers -** These are routers acting as a gateway to the BTC network for other nodes which use different protocols like stratum or pools.

- **Mining nodes -** Does not contain a copy of the blockchain, mines using the stratum protocol or some other pool mining protocol.

- **Lightweight stratum wallet** - Has a wallet and runs a network node with the stratum protocol. It does not contain a copy of the blockchain.

All of the participants of the network form a spider web of connections, running different types of nodes, gateway servers, edge routers, wallet clients and different protocols, which binds them together. Check appendix 2 for a detailed figure of the networks participants.

4 Implementation

Instructions will be given on how to modify the parameters in the source code, mine the genesis block, reset and start the network. Some advice on how to troubleshoot can be found in chapter 4.4.

4.1 Choosing the OS

At the beginning of the project, it was a clear choice that the network would be set up in a Linux environment. Linux is a great OS for control, security and its ability to easily transform in to a server. Debian Jessie was chosen out of the different Linux distributions, because of the familiarity of the OS and the quantity of different repositories available. Different versions of Debian such as Wheezy might offer packages Jessie does not have, but the source of the repository is relatively simple to change to retrieve packets from another distribution of Linux. The GUI coming with the distribution is not necessary but a welcome addition. It will not be featured in this guide however.

4.2 Planning the network

When forking an alternative coin out of the base BTC source code, a few minutes of thought should be given on how to name the coin and what use will the coin have. If the coin will run in a private network, the name and port change will not be necessary. In a public network, the name and port change is needed because if the software uses the same port as the original BTC software, the clients can be tricked in to connecting to a different network.

Some thought should also be given on how many transactions the network will process per day as this is a parameter which should be changed from the original in the BTC source code. If the network is designed to reside in the public internet, some designated nodes should be implemented in the source code so that the clients connecting to the network for the first time will always have a peer to connect to if no other peers have not for some reason connected to the client. The original BTC source code had multiple such seed nodes but have been removed in the process of editing the source code.

4.3 Prerequisites

Before starting to work on the BTC source code, the operating system must be updated, dependencies and the necessary Berkley database must be installed for the

wallet to be able to operate. The node will be a fully operational node, which downloads the full current height of the blockchain from connected peers and has a functional wallet.

The following instructions will be given on how to configure BTC on Debian 8 (Jessie). When using another operating system, especially if it is not a Linux based OS, some independent research might be necessary to replicate the process.

4.3.1 Updating the operating system

First of all, the operating system must be updated to have the latest packets so that the dependencies can be retrieved. To do that, a connection to internet is a requirement and the list of packet sources has to be set correctly. The sources.list file can be found as shown in figure 9.

```
root@debian:~# cd /etc/apt/
root@debian:/etc/apt# ls
apt.conf.d           listchanges.conf  sources.list~   trusted.gpg.d
db-4.8.30.NC         preferences.d     sources.list.d
db-4.8.30.NC.tar.gz  sources.list      trusted.gpg
root@debian:/etc/apt#
```

Figure 9. Updating the sources

When the sources.list has been opened with any text editor software (in this example, Nano is used), the file must be edited to include servers from which to retrieve the data from. The closest server to the current location of the computers location is recommended. The full list of available Debian servers can be found from the operating systems website (Debian.org 2016). In figure 10 is given an example of how to configure the sources.list when the computer resides in Finland.

```
# Debian RGCE repos enabled
deb http://ftp.fi.debian.org/debian/ jessie-updates main
deb-src http://ftp.fi.debian.org/debian/ jessie-updates main
deb http://ftp.fi.debian.org/debian/ jessie main contrib non-free
deb-src http://ftp.fi.debian.org/debian/ jessie main contrib non-free
```

Figure 10. Editing the list

Once the list has been edited, the update command must be given as in figure 11. This updates the location of the repositories and tells the computer where to look for new updates.

```
root@debian:/etc/apt# apt-get update
```

Figure 11. The update command

When the update of the repository is completed, the upgrade command must be given to actually start upgrading the machine. This is done as shown in figure 12:

```
root@debian:/etc/apt# apt-get upgrade
```

Figure 12. The upgrade command

4.3.2 Dependencies

Next in the installation process is the retrieval and installation of the required dependencies for the BTC software. This can be done with a single string of commands. This will be pasted as text in the appendices to make it easy for the user to copy the necessary commands (Chk. appendix 1). An example of inserting the commands presented in figure 13.

```
root@debian:/etc/apt# apt-get install build-essential autoconf libssl-dev libb
oost-dev libboost-chrono-dev libboost-filesystem-dev libboost-program-options-
dev libboost-system-dev libboost-test-dev libboost-thread-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
autoconf is already the newest version.
libboost-chrono-dev is already the newest version.
libboost-filesystem-dev is already the newest version.
libboost-program-options-dev is already the newest version.
libboost-system-dev is already the newest version.
libboost-test-dev is already the newest version.
libboost-thread-dev is already the newest version.
build-essential is already the newest version.
libssl-dev is already the newest version.
```

Figure 13. Dependencies commands

4.3.3 Berkley database

Berkley database is what the BTC software uses for database purposes, it contains the necessary functions to make the wallet of the node to work correctly. The node can be installed without the database, however wallet functions will not be supported. Since Debian Jessie does not have repositories which has the Berkley database, another repository has to be used. This can be done by once again editing the sources.list file and inserting the text shown in the appendices (Chk. appendix 3). An example of this is presented in figure 14.

```
deb http://archive.debian.org/debian-archive/debian/ squeeze main contrib non-free
deb http://archive.debian.org/debian-archive/debian/ squeeze-lts main contrib non-free
```

Figure 14. Squeeze repository

After the text is added, the process which is shown in figures 10 and 11 must be repeated for the system to be able to retrieve the packets. Next in line is the download and installment of the Berkley database. In the example, Berkley database version 4.8 is used but different database versions can be an option. After editing the sources list, updating and upgrading the computer, type the following to the console:

apt-get install libdb4.8++-dev libdb4.8-dev

This will retrieve and install the Berkley database and the pre-requisites for compiling BTC source code should be complete.

4.3.4 Installing Bitcoin

Now that all the dependencies necessary for successful installation have been set up, it is time to actually download the BTC software and get it ready for the editing process, which can be read in chapter 4.6 and 4.7. To download the source code, first make a directory (for example “src”) somewhere, root directory is recommended. Then enter the directory and input the following command:

git clone -b 0.9 https://github.com/bitcoin/bitcoin.git

This will clone files from the Github bitcoin directory to the current directory on the computer. After all the files have been cloned, enter the bitcoin folder. The contents of the folder should look like in figure 15.

```
root@debian:~# cd src/bitcoin/
root@debian:~/src/bitcoin# ls
autogen.sh    contrib  doc      Makefile.am  qa          share
configure.ac  COPYING  INSTALL  pkg.m4       README.md   src
root@debian:~/src/bitcoin# cd src/
root@debian:~/src/bitcoin/src# cd ~/.bitcoin./bitcoin.conf
```

Figure 15. Downloaded files

To make sure that all the dependencies etc. have been installed correctly, a test compilation should be achieved successfully. To do this, the necessary files have to be generated with the following command:

./autogen.sh

After the command has run its course, run the following configuration command:

./configure --without-gui --with-incompatible-bdb

This command configures the software to be used on the command line and makes sure that the Berkley database does not cause incompatibility issues. When the configuration passes without errors, the actual compilation should be done with the following command:

Make install

This will generate the necessary files, which will be used to control the BTC node. After these commands, when it is time to edit the source code and compile again, it is only needed to repeat the “Make” command since all of the configuration and generation have been completed.

4.3.5 Name and port change

If the coin will be used in a private network, this step can be skipped. For the public network, a name change and port change is recommended. The name should be a single word; attention should be given that the name matches how the string bitcoin is in the command. The three-letter combination should be in a similar form as bitcoin is.

The name change

These few next commands will change the string “Bitcoin” to the name one has chosen to replace it, in this example, the name for the coin is “yourcoin”. These commands will change the strings in every file in the bitcoin folder:

cd ~/src/yourcoin

find . -type f -print0 | xargs -0 sed -i 's/bitcoin/yourcoin /g'

find . -type f -print0 | xargs -0 sed -i 's/Bitcoin/Yourcoin /g'

find . -type f -print0 | xargs -0 sed -i 's/BitCoin/YourCoin /g'

find . -type f -print0 | xargs -0 sed -i 's/BITCOIN/YOURCOIN/g'

find . -type f -print0 | xargs -0 sed -i 's/BTC/YCN/g'

find . -type f -print0 | xargs -0 sed -i 's/btc/ycn/g'

find . -type f -print0 | xargs -0 sed -i 's/Btc/Ycn/g'

These commands will change the folder names associated with bitcoin:

find . -exec rename 's/bitcoin/yourcoin/' {} ";"

find . -exec rename 's/btc/YCN/' {} ";"

The port change

Normally, bitcoin uses ports 8333 and 18333 for the mainnet and ports 8332 and 18332 for the testnet. These have to be changed to different numbers so that the clients do not have a chance to connect to the wrong client software. Insert the following commands:

find . -type f -print0 | xargs -0 sed -i 's/8332/9443/' {} ";"

find . -type f -print0 | xargs -0 sed -i 's/8333/9444/' {} ";"

4.4 Genesis block creation

Even though BTC source code is used as the basis for the guide, the original data for the network itself will not be used, the genesis block and the following block data must be original for the network to be a separate unique chain from the original. For this to happen, a new first block hash has to be created (a genesis block). Since the original code for doing this directly in the source code is not included there anymore, an external program for this part of the guide was used. There are other software but for this; a free program called GenesisH0 was used (Hartikka 2016).

4.4.1 Genesis block generator

The generator is coded in python, so python must be installed on the computer. By default, if the OS is up to date, version 2.7 should be already installed (check version with python –V). The next step is to install pip if it is not already on the OS. Pip is a system used for the installation and management of Python written software packages. This can be done with the following command in figure 16.

```
root@debian:~# apt-get install pip
```

Figure 16. Installing pip

Before dependencies for the program can be installed, python-dev must be installed on the OS as in figure 17.

```
root@debian:~# apt-get install python-dev
```

Figure 17. Installing python-dev

Now that the base requirements for installment of the dependencies are met, the following command in figure 18 must be inserted in the console.

```
root@debian:~# sudo pip install scrypt construct==2.5.2
```

Figure 18. Scrypt construct

When the dependencies have been successfully installed, the actual program should be cloned next from Github to any chosen directory on the OS with the following command:

git clone https://github.com/lhartikk/GenesisH0.git

This will create a directory where the files for this program will be copied. Once cloning is complete, enter the folder. The program is now usable by entering genesis.py in front of the rest of the commands, for example the next command will bring out a guidebook on how to use the program.

python genesis.py –help

4.4.2 The creation of the genesis block

Next is the actual creation of the genesis block using the python program genesis.py. To make it more likely for the program to find a valid hash, some of the parameters will be close to what is used in the original BTC. It will take a long while for the program to calculate a valid hash for the genesis block. The calculation took 7.5 hours with a virtual machine. An example of the command given for the creation of the genesis block is as in figure 19.

```
root@debian:~/test/GenesisH0# python genesis.py -z "Tama on testiviesti 23/12/
2016" -n 2083236893 -t 1482502754
```

Figure 19. Genesis block parameters

Explanation for the parameters:

- **-z** is the timestamp, usually a headline of a news article, or anything desired.
- **-n** is the nonce which is a variable for finding the acceptable hash.
- -**t** is the epoch time which can be generated in a Linux environment.

For the timestamp, anything can be chosen but historically a news article dated for the day of the block creation is usually chosen. This is so that it can be verified that the block was created at least past the day the article was created.

The nonce used is the same nonce as in the original BTC source code, the changed timestamp and epoch time ensures it will not produce the same hash as in the original. The epoch time is a way of describing time since 1970, it's a series of numbers which changes upwards as time passes. For example in January 1st 1980, the epoch time was 315532800 and in January 1st 1995 the epoch time was already 788918400.

The epoch time can be found with the command: "**date +%s**" in a Linux environment. After all the parameters has been predetermined, it is time to insert the variables in to the command as in figure x. The calculation of the hash might take a long while depending on the machine one is running. The output however is as presented in the figure 20.

```
nominatim@GeoMap:~/GenesisH0$ python genesis.py -z "Tama on testiviesti 23/12/2016" -n 2083236893
 -t 1482502754
04ffff001d01041e54616d61206f6e2074657374697669657374692032332f31322f32303136
algorithm: SHA256
merkle hash: 22e338dd8dc369c10e12a353cc7a161ac8cbf1274dda744cdc95e5e8275e9603
pszTimestamp: Tama on testiviesti 23/12/2016
pubkey: 04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e
51ec112de5c384df7ba0b8d578a4c702b6bf11d5f
time: 1482502754
bits: 0x1d00ffff
Searching for genesis hash..
160907.0 hash/s, estimate: 7.4 hgenesis hash found!
nonce: 3492649622
genesis hash: 0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da4c4
```

Figure 20. Blockhash found

From the output as described in figure 20, the following variables must be written down for later use when the actual editing of the source code begins:

- Genesis hash
- Merkle hash
- Nonce
- pszTimestamp
- Pubkey
- Time

The same genesis blockhash is used in both mainnet and testnet but for regtest, a new blockhash must be created with an easier difficulty (for example nonce 2). The steps to produce a genesis blockhash for the regtest mode are the same with the exception of a different nonce.

4.5 The creation of the alertkeys

The alertkey system was designed to alert the node users of occurred urgent problems in the network, this feature was removed in BTC version 0.13.0 but since the version used in this guide is still version 0.9, the following part has to be done. The private and public keys must be created for both the mainnet and the testnet. This can be done quickly for the mainnet with the first two following commands:

openssl ecparam -genkey -name secp256k1 -out mainnetkey.pem

openssl ec -in mainnetkey.pem -text > mainnetkey.hex

And for the testnet:

openssl ecparam -genkey -name secp256k1 -out testnetkey.pem

openssl ec -in testnetkey.pem -text > testnetkey.hex

These two commands create a pair of private and public keys and binds them together in hex format. The keys can be read by utilizing the “cat” command:

cat mainnetkey.hex

cat testnetkey.hex

What is desired from these files, are the series of numbers between the lines “pub” and “ASN1 OID: secp256k1”. The copied data must be cleared of colons and linefeeds. When it has been done, save the result for later use.

4.6 Editing the source code part one

After acquiring the necessary data in previous chapters (genesis block, alertkeys etc.), the time has come to start the actual editing of the source code. In the BTC version 0.9, there are two major files which will be the focus for the editing (**chainparams.cpp** and **checkpoints.cpp**). In the following subchapters, the instructions for editing the file **chainparams.cpp** will be given in detail. These instructions are specifically for version 0.9 but the core file structure between versions remain relatively similar. Both of the files mentioned in this chapter can be found in the BTC directory: **/bitcoin/src**. Check appendix 4 for the example of a completed edit of the file **chainparams.cpp.**

4.6.1 Removing the seed nodes

Open the file Chainparams.cpp located at /bitcoin/src in any text editing software (Nano recommended). As the first thing in editing this file, remove the hard coded seed nodes starting from line 23 and ending at line 97. The seed nodes are coded in hex format, and will be used to try to connect to BTC public nodes if they are not removed from the code. After removing the seed nodes, the outcome should look like in figure 21.

```
unsigned int pnSeed[] =
{

};
```

Figure 21. Removed seeds

Next, the vSeeds must be removed, look for the line which says:

vSeeds.push_back

And replace it with the following text:

vFixedSeeds.clear();

vSeeds.clear();

This should be done for mainnet and testnet, regtest does not have any fixed seeds.

4.6.2 Alertkeys

Next, the previously created alert keys (chapter 4.5) will be inserted in the code, both for mainnet and testnet. Look in "CMainParams(){}" for the part where it says vAlertPubKey = ParseHex("…."); and insert the key for the mainnet. The same should be done for the testnet (CTestNetParams())with the equivalent key. An example of the outlook of this step in figure 22.

```
vAlertPubKey = ParseHex("04f47beaaa243f2c864baf3aab2b980ba96b6749
```

Figure 22. Mainnet alertkey

4.6.3 Timestamp

Next, the timestamp acquired in chapter 4.4.2 will be inserted in the code. Look for a string of text which says:

const char* pszTimestamp =

Insert the timestamp defined in the mentioned chapter. An example of this is presented in figure 23.

```
const char* pszTimestamp = "Tama on testiviesti 23/12/2016";
```

Figure 23. Timestamp

4.6.4 Public key

Next, search for the following text:

txNew.vout[0].scriptPubKey =

and insert the pubkey acquired in chapter 4.4.2. An example of this is presented in figure 24, the same pubkey should be used for the testnet. It is not needed for the regtest mode.

```
txNew.vout[0].scriptPubKey = CScript() << ParseHex("04678afdb(
```

Figure 24. Pubkey

4.6.5 The Epoch time and nonce

Epoch time

The epoch time is the current unix time on a Linux machine. The same time should be used when the genesis block was created. Look for text which says:

genesis.nTime =

And insert the epoch time. This should be done for mainnet, testnet and the regtest.

Nonce

The nonce used will be the same as in the output when creating a genesis block. Look for the following text in the code:

genesis.nNonce =

And insert the previously acquired nonce there. The nonce is the same for both mainnet and testnet. Regtest will use a different nonce (Chk. chapter 4.4.2).

4.6.6 The Genesis hash

Next, the genesis hash for mainnet, testnet and regtest will be inserted. The hash for mainnet and testnet is the same while regtest has its own unique one. Look for text which says:

assert(hashGenesisBlock == uint256

And insert the previously generated genesis hash in the bracket. This should be done for both mainnet and testnet. Do the same for regtest with the regtest block hash (Chk. chapter 4.4.2).

4.6.7 The Merkle root

The Merkle root is only needed for the mainnet. Look for the following line:

assert(genesis.hashMerkleRoot == uint256

and insert the merkle hash in to the bracket. After this is done, the editing of the file **chainparams.cpp** is concluded. Chapter 4.7 explains what to do with the other file.

4.7 Editing the source code part two

The first part for editing the source code was for the file **chainparams.cpp**, this chapter is for the other file **checkpoints.cpp**. This file is what contains the checkpoints for important blocks so that syncing with the network is easier. It contains blocks from the original bitcoin source code but these will be removed and replaced with the genesis block created in the guide. Check appendix 5 for an example of a completed edit of the file **checkpoints.cpp**.

The file has pre-set checkpoints from the original version of BTC, which have to be removed and replaced with the genesis blockhash created before. In figure 25 can be found an example of the part which should be edited.

```
static MapCheckpoints mapCheckpoints =
    boost::assign::map_list_of
    ( 11111, uint256("0x0000000069e244f73d78e8fd29ba2fd2ed618bd6fa2ee92559f542fdb26e7c1d"))
    ( 33333, uint256("0x000000002dd5588a74784eaa7ab0507a18ad16a236e7b1ce69f00d7ddfb5d0a6"))
    ( 74000, uint256("0x0000000000573993a3c9e41ce34471c079dcf5f52a0e824a81e7f953b8661a20"))
    (105000, uint256("0x00000000000291ce28027faea320c8d2b054b2e0fe44a773f3eefb151d6bdc97"))
    (134444, uint256("0x00000000000005b12ffd4cd315cd34ffd4a594f430ac814c91184a0d42d2b0fe"))
    (168000, uint256("0x000000000000099e61ea72015e79632f216fe6cb33d7899acb35b75c8303b763"))
    (193000, uint256("0x00000000000000059f452a5f7340de6682a977387c17010ff6e6c3bd83ca8b1317"))
    (210000, uint256("0x00000000000000048b95347e83192f69cf0366076336c639f9b7228e9ba171342e"))
    (216116, uint256("0x00000000000001b4f4b433e81ee46494af945cf96014816a4e2370f11b23df4e"))
    (225430, uint256("0x00000000000001c108384350f74090433e7fcf79a606b8e797f065b130575932"))
    (250000, uint256("0x000000000000003887df1f29024b06fc2200b55f8af8f35453d7be294df2d214"))
    (279000, uint256("0x0000000000000001ae8c72a0b0c301f67e3afca10e819efa9041e458e9bd7e40"))
    (295000, uint256("0x00000000000000004d9b4ef50f0f9d686fd69db2e03af35a100370c64632a983"))
    ;
```

Figure 25. Checkpoints

The first number **1111** is the block height of the hash. The series of characters after **uint256** form the hash which serves as the checkpoint. What is needed in this part, is to remove every checkpoint and insert the genesis block hash as a replacement. The outcome should look similar to figure 26.

```
static MapCheckpoints mapCheckpoints =
    boost::assign::map_list_of
    ( 0, uint256("0x0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da4c4"))
    ;
```

Figure 26. Mapcheckpoints

The next thing is to change some variables as described in figure 27.

```
static const CCheckpointData data = {
    &mapCheckpoints,
    1397080064, // * UNIX timestamp of last checkpoint block
    36544669,   // * total number of transactions between genesis and last checkpoint
                //   (the tx=... number in the SetBestChain debug.log lines)
    60000.0     // * estimated number of transactions per day after checkpoint
};
```

Figure 27. Checkpoint data

The first series of numbers **1397080064** should be changed to the timestamp used in the creation of the genesis block (Chk. chapter 4.4.2.). the number **36544669** should be changed to 0 because there are no transactions before the genesis block. The last

set of numbers **60000** Should be changed to whatever number the creator of the network feels comfortable with. 500 was used in this guide. These changes should be made for the mainnet, testnet and regtest with the exception that regtest has its own separate genesis hash. The checkpoint data is 0 for all three parameters. An example of the regtest part in figure 28.

```
static MapCheckpoints mapCheckpointsRegtest =
    boost::assign::map_list_of
    ( 0, uint256("1625cbffa183e8037f7c2255d5f5f2c19cc89a4881738e85b67dd48b96f23176"))
    ;
static const CCheckpointData dataRegtest = {
    &mapCheckpointsRegtest,
    0,
    0,
    0
};
```

Figure 28. Regtest checkpoint

4.8 Starting up the network

To start up the network, at least two nodes are required for the POW to successfully validate found blocks. The program used to start the network are located in the following directory path (may vary depending where bitcoin file directory has been cloned to):

/src/bitcoin/src/bitcoind

This program can be started in 3 different modes; mainnet, testnet and regtest. To start the net in the chosen mode, one of the next commands should be inserted:

./bitcoind –printtoconsole

./bitcoind –testnet –printtoconsole

./bitcoind –regtest –printtoconsole

When attempting to start the mainnet, the command line prints the debug.log data, creates files to the root directory (~/.bitcoin) and complains that the bitcoin.conf file has not been generated. This file is what contains the basic configurations for the node, whether it is used for just the wallet functions or a node which gives access to external miners. To create this file, simply go to the bitcoin root directory at

~/.bitcoin and create the file with any available text editor. An example of the file, if it is used as a server node as illustrated in figure 29.

```
gen=1
listen=1
server=1
rpcuser=foo
rpcpassword=bar
addnode=207.243.197.55
rpcallowip=127.0.0.1
rpcallowip=207.243.197.55
rpcport=8332
```

Figure 29. Bitcoin.conf

The configuration options have the following meaning:

- **Gen=** Starts the miner when the wallet software is started.
- **Listen=** Accepts external connections.
- **Server=** Makes the node accept RPC commands.
- **Rpcuser=** Username for the RPC interface.
- **Rpcpassword=** Password for the RPC interface.
- **Addnode=** Command for the software to find a peer from that address.
- **Rpcallowip=** Gives access to the RPC interface for a specific IP-address.
- **Rpcport=** Defines the port used for the RPC connections.

For a normal wallet client, only the username and password for the RPC interface, connection to a peer and the **rpcport** and **rpcallowip** to the localhost are needed. If it is necessary to allow other nodes to connect to the RPC interface, the addresses must be added separately with the **rpcallowip** option.

The network nodes are controlled by two main components on the console, the server part is called **bitcoind** and the second part is called **bitcoin-cli**. Bitcoin-cli is the one that sends commands to the server and tells it what to do. For example, to find connected peers, check wallet balance, create a new address or to send a transaction, the following commands can be inserted:

./bitcoin-cli getpeerinfo

./bitcoin-cli getbalance

./bitcoin-cli getnewaddress

./bitcoin-cli sendtoaddress 1AkCxBdjCKQGF956B5Hc6ZjJLbh19FKdQq 1000

4.9 Resetting the network

To reset the blockchain back to the genesis block in a large public network is impossible without the agreement between the majority of the network participants. If the majority of the network would refuse this change, the ones who reset the blockchain would only create an insecure side chain with less hashing power than the original.

However, if the blockchain was created in a private network with full control of all of the nodes connected, the consensus is automatically reached when the change is done with every participant node on the network. This can be done by just simply removing the directory **~/.bitcoin** and repeating the process described in chapter 4.8 for every node on the network. The network will start mining the next block after the genesis block and every transaction or wallet balance have been reset to how it was in the beginning.

4.10 Troubleshooting

In the course of writing the guide some mistakes and errors have been made, this chapter is to inform on how to avoid a few of them. It is advised to first check the **debug.log** located at **~/.bitcoin/** to further troubleshoot for problems.

Configuration error with Berkley DB

After inserting the **./configure** command, this error might appear if the Berkley database is different from version 4.8. In this case, the command

./configure –with-incompatible-bdb

installs the database in spite of the version difference. To use the client without the GUI, the command should be like the following:

./configure –without-gui –with-incompatible-bdb

Connection refused

When the debug.log tells that a certain node connection defined in the bitcoin.conf file is refused, the node the client is trying to connect to is either offline or only allows connections from predefined addresses.

Bitcoind closes after first try

It will comment on a few things, as described in figure 30, this is normal as it creates the ~/.bitcoin directory. When the RPC username and password are added with the "listen" and "server" option, external miners are able to connect to the node server.

```
/home/pavel/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=EwJeV3fZTyTVozdECF627BkBMnNDwQaVLakG3A4wXYyk
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

Figure 30. Conf error.

Genesis block creation error

When the nonce is chosen for the algorithm to find an acceptable hash, it is recommended to use a nonce close to the one used in the original BTC code since that is most likely to work. If other values are used, it might take a few tries to succeed in finding the hash. This can take a while, since it took over 7 hours to find the hash in chapter 4.4.2.

5 Verification

This section is for the verification of the networks ability to operate correctly and the validation of core functions of the wallet software. Some examples on how to operate the wallet will also be illustrated.

5.1 Network topology

The network was created in an isolated production environment, separate from the public internet. The emulation of the public BTC network was done with just a few computers since no more was necessary. The network consists of two server nodes that accept incoming connections and listens to RPC commands coming towards the RPC interface. Both of these server nodes have a client node connected to them with only the basic functions enabled (including coin generation).

In the configuration options on the server nodes, server 1 added server 2 as a peer and the same was done vice versa. The client nodes only added one of the servers as peers. The **addnode** option in the configuration file lets other nodes to learn connections through the directly defined node in the configuration file. For example when looking at figure 31, Client 1 has added Server 1 as a peer. Client 1 learns of other connections through Server 1 and adds Server 2 and client 2 as a peer, which tells where to find Client 2. Figure 32 illustrates the connection count from Client 1.

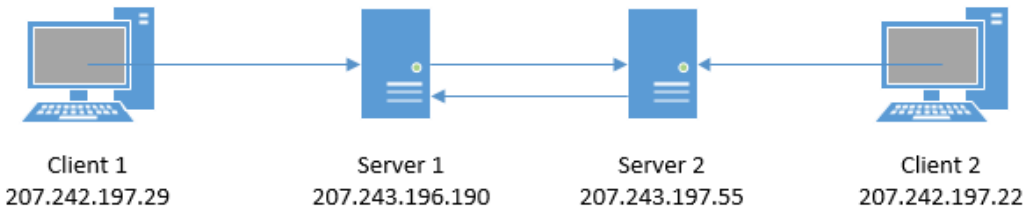


Figure 31. Network topology

```
root@debian:~/src/bitcoin/src# ./bitcoin-cli getconnectioncount
3
```

Figure 32. Connection count

5.2 Connected peers

Peers are needed for the network to have a chance of operating, at least 2 nodes must be a part of the network for it to be able to function as a mediate for the validation of blocks. Peers are easy to add through the bitcoin.conf configuration file. The original BTC source code however has hardcoded seed nodes in its source code to ensure connectivity; this is not the case in this version since only a few nodes are needed to add to the network for testing purposes. The peers connected to a node can be verified by typing the **getpeerinfo** command as in the following figure 33.


```
    "addr" : "207.243.197.55:8333",
    "addrlocal" : "207.242.196.190:8333",
    "services" : "00000001",
    "lastsend" : 1486723196,
    "lastrecv" : 1486723426,
    "bytessent" : 967,
    "bytesrecv" : 1468,
    "conntime" : 1486708996,
    "pingtime" : 0.10020600,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.5/",
    "inbound" : false,
    "startingheight" : 3533,
    "banscore" : 0,
    "syncnode" : false
},
{
    "addr" : "207.242.197.22:8333",
    "addrlocal" : "207.242.196.190:8333",
    "services" : "00000001",
    "lastsend" : 1486723196,
    "lastrecv" : 1486723426,
    "bytessent" : 3141,
    "bytesrecv" : 2403,
    "conntime" : 1486708997,
    "pingtime" : 0.10020200,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.5/",
    "inbound" : false,
    "startingheight" : 3533,
    "banscore" : 0,
    "syncnode" : false
},
{
    "addr" : "207.242.197.29:52699",
    "addrlocal" : "207.242.196.190:8333",
    "services" : "00000001",
```


Figure 33. Peer info

The connected peers can also be verified by checking the debug.log when the BTC node is started. There should appear a "received version message", which informs of a connected peer with the same protocol in use. An example of this is presented in figure 34.

```
receive version message: /Satoshi:0.9.5/: version 70002, blocks=6255, us=207.2
42.197.29:53197, them=207.242.196.190:8333, peer=207.242.196.190:8333
Added time data, samples 2, offset +2 (+0 minutes)
receive version message: /Satoshi:0.9.5/: version 70002, blocks=6255, us=207.2
42.197.29:48731, them=207.243.197.55:8333, peer=207.243.197.55:8333
Added time data, samples 3, offset +22 (+0 minutes)
keypool reserve 1
CreateNewBlock(): total size 1000
Running BitcoinMiner with 1 transactions in block (188 bytes)
receive version message: /Satoshi:0.9.5/: version 70002, blocks=6255, us=207.2
42.197.29:8333, them=207.242.197.22:8333, peer=207.242.197.22:52971
Added time data, samples 4, offset +1 (+0 minutes)
hashmeter   1533 khash/s
P2P peers available. Skipped DNS seeding.
```

Figure 34. Connected peers

5.3 Network info

The command **getinfo** gives the basic information about the network difficulty. It can be seen in figure 35 that the network difficulty has risen from the basic 1.0000 to over 1.05 during the 3 weeks the protocol has been running.

```
root@debian:~/src/bitcoin/src# ./bitcoin-cli getinfo
{
    "version" : 90500,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 26300.00000000,
    "blocks" : 6255,
    "timeoffset" : 0,
    "connections" : 3,
    "proxy" : "",
    "difficulty" : 1.05399016,
    "testnet" : false,
    "keypoololdest" : 1487949727,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "relayfee" : 0.00001000,
    "errors" : ""
}
```

Figure 35. Getinfo

It is also possible to find information on the status of the network hashrate (the amount of calculation power in the network) with the command **getmininginfo** as illustrated in figure 36.

```
root@debian:~/src/bitcoin/src# ./bitcoin-cli getmininginfo
{
    "blocks" : 6256,
    "currentblocksize" : 1000,
    "currentblocktx" : 0,
    "difficulty" : 1.05399016,
    "errors" : "",
    "genproclimit" : -1,
    "networkhashps" : 6570755,
    "pooledtx" : 0,
    "testnet" : false,
    "generate" : true,
    "hashespersec" : 1321158
}
root@debian:~/src/bitcoin/src#
```

Figure 36. Network hashrate

5.4 POW in action

For the network to function, mining has to be enabled on at least two machines so that the verification of transactions can begin. The following figure 37 is of the found block number 194 on the Bitcoin mining server. This extra information is shown when the block found is the result of that computers calculation power. Normally if the block is only verified by a peer the information provided by the feed will be much less. The figure shows the following;

- **Calculated Block hash** (CBlock(hash=”00000000d372…”)).
- **The nonce nNonce=(3867..)**, which is the variable used to calculate the block hash.
- **The previous block hash**, which shows that the current block is linked to the previously found block number 193.
- **Merkle root hash (hashMerkleRoot=(e4ee..)**, which makes sure that the hash received is not an altered or fake when received by peers on the network.
- **The generated amount of coins (generated 50.00)** and the wallet in which the coins were sent to (AddToWallet e4ee..)

Figure 37. POW

Figure 38 is from a peer connected to the mining server and illustrates the same found block hash.

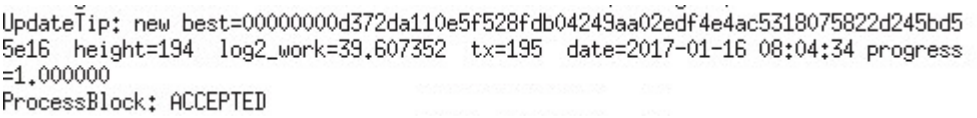


Figure 38. New block

This snippet of information only shows the new best, or highest found block (height=194), the hash of it (new best=00000000d37..), the date and that the block was accepted as valid.

5.5 GetBlockTemplate

For external miners to be able to get work from the BTC node server, the server must be able to provide the miner with the GBT which tells the miner how to search for the next block. If this is not provided, the miner software has no clue on how to proceed.

To verify that the GBT is working, the command depicted in figure 39 should be inserted.

```
root@BTCtestnet:~/src/bitcoin/src# ./bitcoin-cli getblocktemplate
{
    "version" : 3,
    "previousblockhash" : "00000000f338698e2dd1a9461e4bcdd924ae0048b92e3889660ecf26ae771258",
    "transactions" : [
    ],
    "coinbaseaux" : {
        "flags" : "062f503253482f"
    },
    "coinbasevalue" : 5000000000,
    "target" : "00000000ffff0000000000000000000000000000000000000000000000000000",
    "mintime" : 1484561650,
    "mutable" : [
        "time",
        "transactions",
        "prevblock"
    ],
    "noncerange" : "00000000ffffffff",
    "sigoplimit" : 20000,
    "sizelimit" : 1000000,
    "curtime" : 1484567911,
    "bits" : "1d00ffff",
    "height" : 213
}
```

Figure 39. GBT

5.6 Using the wallet

The next series of figures depict how the wallet is used in basic operations such as creating a new address for the wallet, sending coins to another wallets address and verifying the transaction. In figure 40 the first node creates a new address for receiving payments.

```
root@BTCtestnet:~/src/bitcoin/src# ./bitcoin-cli getnewaddress
14WXL3CfcpYAncx3ZY82JsNCaNBAGmWAPi
```

Figure 40. Address creation

As illustrated In figure 41, this address is then used on the second node when 1000 BTC is sent to the wallet of the first node, including the transaction fee used to maintain the network (0.0001 btc).

```
root@BTCtestnet:~/src/bitcoin/src# ./bitcoin-cli sendtoaddress 14WXL3CfcpYAncx3Z
Y82JsNCaNBAGmWAPi 1000.0001
336861e1cfaa57546198e01f3900b4a43c8bec94159640032e21f4e465f7d8eb
```

Figure 41. Transaction

The network shows the transactions made in the block on the second node, usually there is only a single transaction in a block, which is the 50 default coins generated from finding a block.

This time two transactions are found, one of which is the 1000.0001 BTC sent to the first node. The following figure 42 is from the second nodes feed where the transaction was first initiated.

```
CTxOut(nValue=49.99960000, scriptPubKey=OP_DUP OP_HASH160 c2419e8d2f6f)
CTxOut(nValue=1000.00010000, scriptPubKey=OP_DUP OP_HASH160 267dc1c9ab37)
```

Figure 42. Transaction value

The next figure 43 is from the first node where the transaction was received, which shows that an additional transaction has been added to the block mined.

```
Running BitcoinMiner with 2 transactions in block (2649 bytes)
hashmeter   3194 khash/s
AddToWallet 336861e1cfaa57546198e01f3900b4a43c8bec94159640032e21f4e465f7d8eb  update
UpdateTip: new best=0000000088ce8ffba43bf1425f99d13d0e43b204b6dc7b863d54d4300be3d652  height=201  log2_work=39.658233  tx=203  date=
2017-01-16 09:52:15 progress=1.000000
AddToWallet 336861e1cfaa57546198e01f3900b4a43c8bec94159640032e21f4e465f7d8eb
```

Figure 43. Transaction received

The transaction amount received by the first node can be checked as in the next figure 44.

```
root@BTCtestnet:~/src/bitcoin/src# ./bitcoin-cli getreceivedbyaddress 14WXL3Cfcp
YAncx3ZY82JsNCaNBAGmWAPi
1000.00010000
root@BTCtestnet:~/src/bitcoin/src#
```

Figure 44. Received by address

5.7 Wallet encryption

The wallet can and should be encrypted if operated in a public network. The ”**wallet.dat”** is the file which needs to be secured with encryption because anyone with access to the file can spend the coins if no passphrase is implemented. For example, if the computer was compromised and the wallet.dat was imported to another wal-

let, the coins would be spendable. This can be done by inserting the command in figure 45.

```
root@debian:~/src/bitcoin/src# ./bitcoin-cli encryptwallet root66
wallet encrypted; Bitcoin server stopping, restart to run with encrypted walle
t. The keypool has been flushed, you need to make a new backup.
root@debian:~/src/bitcoin/src#
```

Figure 45. Wallet encryption.

5.8 Wallet backup

When encryption of the wallet has been done, it is advised to create a backup of the wallet to a secure location, preferably without an internet connection. An example of how this is done is presented in figure 46.

```
root@BTCtestnet:~/src/bitcoin/src# ./bitcoin-cli backupwallet ~/src/
root@BTCtestnet:~/src/bitcoin/src# cd
root@BTCtestnet:~# cd src/
root@BTCtestnet:~/src# ls
bitcoin  wallet.dat
root@BTCtestnet:~/src#
```

Figure 46. Wallet backup

6 Conclusion

The aim of the thesis was to create a working payments system utilizing the blockchain technology of the BTC protocol. The nodes and wallets were ment to be operated with the command line. This is a technology developed very recently (released in 2009) so the information on the internet about how to actually go and alter the source code is very scarce. The end result was successful, specially considering the inexperience as a developer in blockchain technologies. The nodes are all up and running and mining to find blocks. The wallets are able to send transactions and receive them. All changes in the blockchain are verified by broadcasting the change to other nodes.

An additional aim for the thesis was to make the resetting and adding to the network easy. It was found, because of the nature of the technology, that the resetting process can be quite vexing if the network built is large. The data from the blockchain must be removed completely from each node on the network to successfully reset the transaction history because every node has a copy of the blockchain.

This is achievable only because the network was implemented in a private, protected environment with no access to the internet. On a public network, the process for resetting the network would require the approval of most of the participants on the network, which would be unlikely if the aim was to wipe all of the balances and transactions on the network clean.

A major problem, which occurred at the end of the project, was that the software had no internal miners implemented. This meant that external miners should be used to be able to find blocks on the blockchain. Since only the first genesis block was created, the software had no way of generating info for the external miners to connect to and was stuck in downloading the blockchain. In the end there was no other way except to change the BTC software version to 0.9 instead of the original 0.10.

The version 0.9 of BTC had a different file structure so it took a while to figure out the core structure. The main file, which is in version 0.10, was split into two separate files. The version 0.9 had an internal miner in the software and was able to mine the

first blocks which solved the problem. Changing back to version 0.10 was possible but not convenient for the client, since if a reset of the network was needed, the process of reverting to 0.9 would be required.

Since the software was aimed to be run on the command line, the next step would be to implement a working GUI wallet. A GUI version would make it easier for the average user to operate the node and wallet, but not necessary to be able to build a functional network. Because the BTC protocol is designed to be very public, a blockchain explorer would be a good addition for the network. A blockchain explorer would essentially be a website which can be used to search for transactions on the network or wallet address balances, similar to what is available in the public network.

To make it easy for the client to add to the network, a template of a machine with a ready to be compiled node software was created. Copying the machine using this template takes only a couple minutes. Since the thesis was written in the form of a guide, additional instructions on how to change the name and ports of the currency were added as a chapter even though it was not necessary for the client.

Sources

Bitcoin Project. N.d. Proof Of Work. Referenced 4.10.2016.

https://bitcoin.org/en/developer-guide#proof-of-work

Bosselaers, A. 2012. The hash function RIPEMD-160. Referenced 4.10.2016.

http://homes.esat.kuleuven.be/~bosselae/ripemd160.html

Cornwell, S & Atkinson, J. 2013. Bitcoin simplified. Referenced 31.10.2016.

http://bitcoinsimplified.org/learn-more/anonymity/

Debian.org. 2016. Debian worldwide mirror sites. Referenced 24.1.2017.

https://www.debian.org/mirror/list

Deschapell, A. 2016. Why a 1MB Block Size May Be Right for Bitcoin Today. Referenced 26.10.2016.

http://www.coindesk.com/1mb-block-size-today-bitcoin/

Hartikka, L. 2016. SHA256/scrypt/X11/X13/X15 genesis blocks for virtual currencies. Referenced 25.1.2017

https://github.com/lhartikk/GenesisH0

HowStuffWorks. N.d. What is a digital signature? Referenced 28.9.2016.

http://computer.howstuffworks.com/digital-signature.htm

IWS. N.d. Descriptions of SHA-256, SHA-384, and SHA-512. Referenced 4.10.2016.

http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf

JYVSECTEC. N.d. Cyber Operation Environment – RGCE. Referenced 21.9.2016.

http://jyvsectec.fi/en/cyber-environment/

JYVSECTEC. N.d. Who we are. Where we come from. Referenced 21.9.2016.

http://jyvsectec.fi/en/about-us/

Kieren, J. 2015. Blockchain scalability. Referenced 31.10.2016.

https://www.oreilly.com/ideas/blockchain-scalability

Lackey, E. 2012. Red Hat Certificate System Common Criteria Certification 8.1 Deployment, Planning, and Installation. Referenced 28.9.2016.

https://access.redhat.com/documentation/en-US/Red_Hat_Certificate_System_Common_Criteria_Certification/8.1/html/Deploy_and_Install_Guide/index.html

Nakamoto, S. 2008. Bitcoin: A Peer-to-Peer Electronic Cash system. Referenced 4.10.2016.

https://bitcoin.org/bitcoin.pdf

National Vulnerability Database. 2012. Vulnerability Summary for CVE-2010-5139. Referenced 12.10.2016

https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-5139

Norton, S. 2016. CIO Explainer: What Is Blockchain?. Referenced 4.10.2016.

http://blogs.wsj.com/cio/2016/02/02/cio-explainer-what-is-blockchain/

O'reilly Media, inc. 2013. Chapter 6. The Bitcoin Network. Referenced 24.2.2016

http://chimera.labs.oreilly.com/books/1234000001802/ch06.html#_the_extended_bitcoin_network

Poiroux, J. .2015. SHA256 vs Scrypt vs x11 Algorithms. Referenced 4.10.2016.

http://blockgen.net/sha256-vs-scrypt-vs-x11-algorithms/

Pollnow, V. 2016. Why Investors Should Care About Bitcoin Mining Pools. Referenced 31.10.2016.

https://bitcoinira.com/news/2016-09-12/investors-care-bitcoin-mining-pools-1441

Poon, J & Dryja, T. 2016. The Bitcoin Lightning Network: Scalable On-chain Instant Payments. Referenced 31.10.2016.

https://lightning.network/lightning-network-paper.pdf

Silva, J. 2003. An overview of cryptographic hash functions and their uses. Referenced 28.9.2016.

https://www.sans.org/reading-room/whitepapers/vpns/overview-cryptographic-hash-functions-879

Sptizner, L. N.d. What is MD5, and why do I care? Referenced 28.9.2016.

http://www.spitzner.net/md5.html

Wirdum, A. 2016. Understanding the Lightning Network, Part 2: Creating the Network. Referenced 31.10.2016.

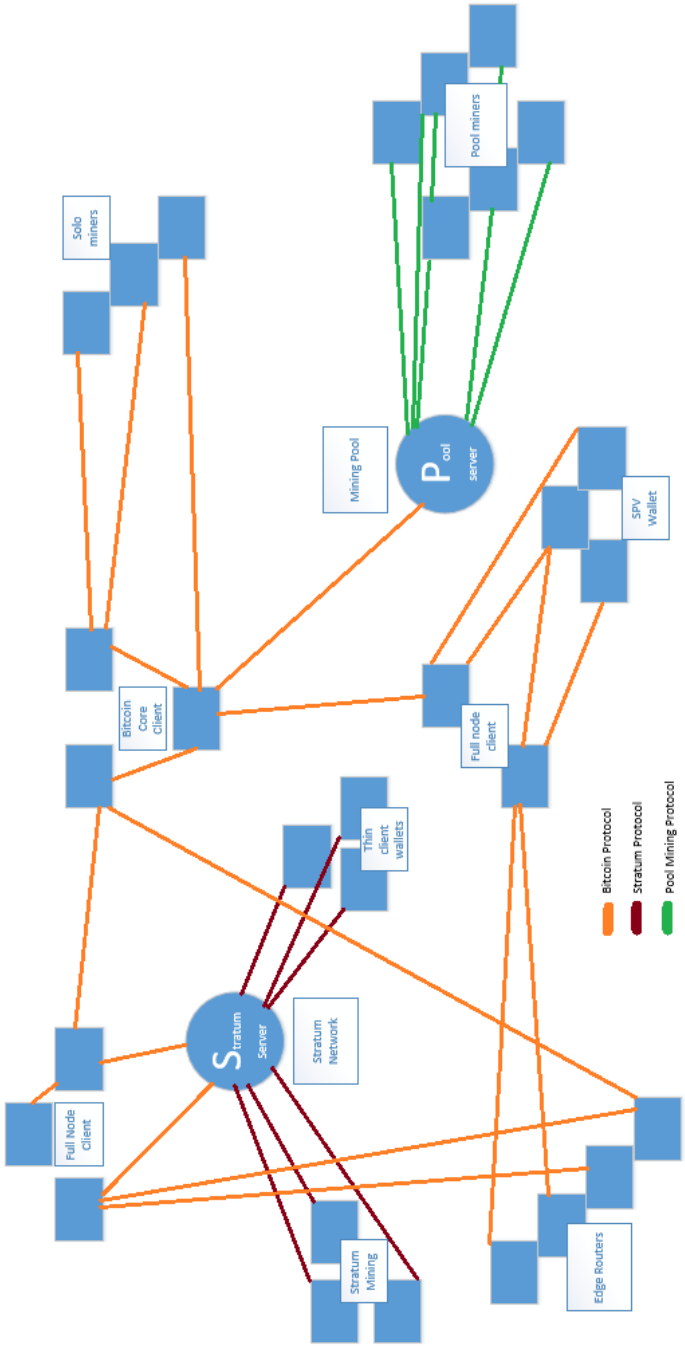
https://bitcoinmagazine.com/articles/understanding-the-lightning-network-part-creating-the-network-1465326903

Appendices

Appendix 1. BTC dependencies

apt-get install build-essential autoconf libssl-dev libboost-dev libboost-chrono-dev libboost-filesystem-dev libboost-program-options-dev libboost-system-dev libboost-test-dev libboost-thread-dev

Appendix 2. BTC network



Appendix 3. Repository address

deb http://archive.debian.org/debian-archive/debian/ squeeze main contrib non-free

deb http://archive.debian.org/debian-archive/debian/ squeeze-lts main contrib non-free

Appendix 4. Chainparams.cpp

```
// Copyright (c) 2010 Satoshi Nakamoto
// Copyright (c) 2009-2014 The Bitcoin developers
// Distributed under the MIT/X11 software license, see the accompanying
// file COPYING or http://www.opensource.org/licenses/mit-license.php.

#include "chainparams.h"

#include "assert.h"
#include "core.h"
#include "protocol.h"
#include "util.h"

#include <boost/assign/list_of.hpp>

using namespace boost::assign;

//
// Main network
//

unsigned int pnSeed[] =
{
```

```
};


class CMainParams : public CChainParams {
public:
   CMainParams() {
      // The message start string is designed to be unlikely to occur in normal data.
      // The characters are rarely used upper ASCII, not valid as UTF-8, and produce
      // a large 4-byte int at any alignment.
      pchMessageStart[0] = 0xf9;
      pchMessageStart[1] = 0xbe;
      pchMessageStart[2] = 0xb4;
      pchMessageStart[3] = 0xd9;
      vAlertPubKey =
ParseHex("04f47beaaa243f2c864baf3aab2b980ba96b67497a5cc2a888c3dd6788202
41c4df0dece8436cf3a824500b632ebe8eb41377a7c0ab6e7924adc7b1f9832199479")
;
      nDefaultPort = 8333;
      nRPCPort = 8332;
      bnProofOfWorkLimit = CBigNum(~uint256(0) >> 32);
                  nSubsidyHalvingInterval = 210000;


      // Build the genesis block. Note that the output of the genesis coinbase cannot
      // be spent as it did not originally exist in the database.
      //
```

```
// CBlock(hash=000000000019d6, ver=1, hashPrevBlock=00000000000000,
hashMerkleRoot=4a5e1e, nTime=1231006505, nBits=1d00ffff$

// CTransaction(hash=4a5e1e, ver=1, vin.size=1, vout.size=1, nLockTime=0)

// CTxIn(COutPoint(000000, -1), coinbase
04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63
656c6c6f$

// CTxOut(nValue=50.00000000, scriptPubKey=0x5F1DF16B2B704C8A578D0B)

// vMerkleTree: 4a5e1e

const char* pszTimestamp = "Tama on testiviesti 23/12/2016";

CTransaction txNew;

txNew.vin.resize(1);

txNew.vout.resize(1);

txNew.vin[0].scriptSig = CScript() << 486604799 << CScriptNum(4) << vec-
tor<unsigned char>((const unsigned char*)pszTimestamp, (const unsigned
char*)pszTimestamp + strlen(pszTimestamp));

txNew.vout[0].nValue = 50 * COIN;

        txNew.vout[0].scriptPubKey = CScript() <<
ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f6
1deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f")
<< OP_CHECKSIG;

        genesis.vtx.push_back(txNew);

genesis.hashPrevBlock = 0;

        genesis.hashMerkleRoot = genesis.BuildMerkleTree();

genesis.nVersion = 1;

genesis.nTime    = 1482502754;

genesis.nBits    = 0x1d00ffff;
```

```
genesis.nNonce   = 3492649622;


hashGenesisBlock = genesis.GetHash();

assert(hashGenesisBlock ==
uint256("0x0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da
4c4"));

assert(genesis.hashMerkleRoot ==
uint256("0x22e338dd8dc369c10e12a353cc7a161ac8cbf1274dda744cdc95e5e8275e
9603"));


vFixedSeeds.clear();

vSeeds.clear();


base58Prefixes[PUBKEY_ADDRESS] = list_of(0);

base58Prefixes[SCRIPT_ADDRESS] = list_of(5);

base58Prefixes[SECRET_KEY] =     list_of(128);

base58Prefixes[EXT_PUBLIC_KEY] = list_of(0x04)(0x88)(0xB2)(0x1E);

base58Prefixes[EXT_SECRET_KEY] = list_of(0x04)(0x88)(0xAD)(0xE4);

            // Convert the pnSeeds array into usable address objects.

for (unsigned int i = 0; i < ARRAYLEN(pnSeed); i++)

{

   // It'll only connect to one or two seed nodes because once it connects,

   // it'll get a pile of addresses with newer timestamps.

   // Seed nodes are given a random 'last seen time' of between one and two

   // weeks ago.
```

```
            const int64_t nOneWeek = 7*24*60*60;
            struct in_addr ip;
            memcpy(&ip, &pnSeed[i], sizeof(ip));
            CAddress addr(CService(ip, GetDefaultPort()));
            addr.nTime = GetTime() - GetRand(nOneWeek) - nOneWeek;
            vFixedSeeds.push_back(addr);
        }
    }

    virtual const CBlock& GenesisBlock() const { return genesis; }
    virtual Network NetworkID() const { return CChainParams::MAIN; }
virtual const vector<CAddress>& FixedSeeds() const {
        return vFixedSeeds;
    }
protected:
    CBlock genesis;
    vector<CAddress> vFixedSeeds;
};
static CMainParams mainParams;



//
// Testnet (v3)
//
```

```
class CTestNetParams : public CMainParams {
public:
   CTestNetParams() {
      // The message start string is designed to be unlikely to occur in normal data.
      // The characters are rarely used upper ASCII, not valid as UTF-8, and produce
      // a large 4-byte int at any alignment.
pchMessageStart[0] = 0x0b;
      pchMessageStart[1] = 0x11;
      pchMessageStart[2] = 0x09;
      pchMessageStart[3] = 0x07;
                  vAlertPubKey =
ParseHex("04cf98c62aafe52e15bf897df5a8cd18b12a1cbfc073fd232db696c7548a11c
4f124acd195072a335ef9163ac9930d176e69443cea0d02d47f5f4ba07b43f84daf");
                        nDefaultPort = 18333;
      nRPCPort = 18332;
      strDataDir = "testnet3";

      // Modify the testnet genesis block so the timestamp is valid for a later start.
      genesis.nTime = 1482502754;
      genesis.nNonce = 3492649622;
      hashGenesisBlock = genesis.GetHash();
      assert(hashGenesisBlock ==
uint256("0x0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da
4c4"));
```

```
        vFixedSeeds.clear();
        vSeeds.clear();
                        base58Prefixes[PUBKEY_ADDRESS] = list_of(111);
        base58Prefixes[SCRIPT_ADDRESS] = list_of(196);
        base58Prefixes[SECRET_KEY]     = list_of(239);
        base58Prefixes[EXT_PUBLIC_KEY] = list_of(0x04)(0x35)(0x87)(0xCF);
        base58Prefixes[EXT_SECRET_KEY] = list_of(0x04)(0x35)(0x83)(0x94);
    }
    virtual Network NetworkID() const { return CChainParams::TESTNET; }
};
static CTestNetParams testNetParams;



//
// Regression test
//
class CRegTestParams : public CTestNetParams {
public:
    CRegTestParams() {
        pchMessageStart[0] = 0xfa;
        pchMessageStart[1] = 0xbf;
        pchMessageStart[2] = 0xb5;
        pchMessageStart[3] = 0xda;
        nSubsidyHalvingInterval = 150;
```

```
        bnProofOfWorkLimit = CBigNum(~uint256(0) >> 1);
        genesis.nTime = 1482502754;
        genesis.nBits = 0x207fffff;
        genesis.nNonce = 4;
        hashGenesisBlock = genesis.GetHash();
        nDefaultPort = 18444;
        strDataDir = "regtest";
        assert(hashGenesisBlock ==
uint256("1625cbffa183e8037f7c2255d5f5f2c19cc89a4881738e85b67dd48b96f23176
"));

        vSeeds.clear();  // Regtest mode doesn't have any DNS seeds.
    }

    virtual bool RequireRPCPassword() const { return false; }
    virtual Network NetworkID() const { return CChainParams::REGTEST; }
};
static CRegTestParams regTestParams;

static CChainParams *pCurrentParams = &mainParams;

const CChainParams &Params() {
    return *pCurrentParams;
}
```

```
void SelectParams(CChainParams::Network network) {
    switch (network) {
        case CChainParams::MAIN:
            pCurrentParams = &mainParams;
            break;
        case CChainParams::TESTNET:
            pCurrentParams = &testNetParams;
            break;
        case CChainParams::REGTEST:
            pCurrentParams = &regTestParams;
            break;
        default:
            assert(false && "Unimplemented network");
            return;
    }
}

bool SelectParamsFromCommandLine() {
    bool fRegTest = GetBoolArg("-regtest", false);
    bool fTestNet = GetBoolArg("-testnet", false);

    if (fTestNet && fRegTest) {
        return false;
    }
```

```
    if (fRegTest) {
        SelectParams(CChainParams::REGTEST);
    } else if (fTestNet) {
        SelectParams(CChainParams::TESTNET);
    } else {
        SelectParams(CChainParams::MAIN);
    }
    return true;
}
```

Appendix 5. Checkpoints.cpp

```
// Copyright (c) 2009-2014 The Bitcoin developers
// Distributed under the MIT/X11 software license, see the accompanying
// file COPYING or http://www.opensource.org/licenses/mit-license.php.

#include "checkpoints.h"

#include "main.h"
#include "uint256.h"

#include <stdint.h>

#include <boost/assign/list_of.hpp> // for 'map_list_of()'
#include <boost/foreach.hpp>

namespace Checkpoints
{
    typedef std::map<int, uint256> MapCheckpoints;

    // How many times we expect transactions after the last checkpoint to
    // be slower. This number is a compromise, as it can't be accurate for
    // every system. When reindexing from a fast disk with a slow CPU, it
    // can be up to 20, while when downloading from a slow network with a
    // fast multicore CPU, it won't be much higher than 1.
```

```
    static const double SIGCHECK_VERIFICATION_FACTOR = 5.0;

    struct CCheckpointData {
        const MapCheckpoints *mapCheckpoints;
        int64_t nTimeLastCheckpoint;
        int64_t nTransactionsLastCheckpoint;
        double fTransactionsPerDay;
    };

    bool fEnabled = true;

    // What makes a good checkpoint block?
    // + Is surrounded by blocks with reasonable timestamps
    //   (no blocks before with a timestamp after, none after with
    //    timestamp before)
    // + Contains no strange transactions
    static MapCheckpoints mapCheckpoints =
        boost::assign::map_list_of
        ( 0,
uint256("0x0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da
4c4"))
        ;
    static const CCheckpointData data = {
        &mapCheckpoints,
        1482502754, // * UNIX timestamp of last checkpoint block
```

```
        0,   // * total number of transactions between genesis and last checkpoint
                    //   (the tx=... number in the SetBestChain debug.log lines)
        500.0     // * estimated number of transactions per day after checkpoint
    };

    static MapCheckpoints mapCheckpointsTestnet =
        boost::assign::map_list_of
        ( 0,
uint256("0x0000000085411c2a7073223a3cfc1730276f832ee1ab8445a2fe5da77b7da
4c4"))
        ;
    static const CCheckpointData dataTestnet = {
        &mapCheckpointsTestnet,
        1482502754,
        0,
        300
    };

    static MapCheckpoints mapCheckpointsRegtest =
        boost::assign::map_list_of
        ( 0,
uint256("1625cbffa183e8037f7c2255d5f5f2c19cc89a4881738e85b67dd48b96f23176
"))
        ;
    static const CCheckpointData dataRegtest = {
```

```
    &mapCheckpointsRegtest,
    0,
    0,
    0
};

const CCheckpointData &Checkpoints() {
    if (Params().NetworkID() == CChainParams::TESTNET)
        return dataTestnet;
    else if (Params().NetworkID() == CChainParams::MAIN)
        return data;
    else
        return dataRegtest;
}

bool CheckBlock(int nHeight, const uint256& hash)
{
    if (!fEnabled)
        return true;

    const MapCheckpoints& checkpoints = *Checkpoints().mapCheckpoints;

    MapCheckpoints::const_iterator i = checkpoints.find(nHeight);
    if (i == checkpoints.end()) return true;
```

```
        return hash == i->second;
    }

    // Guess how far we are in the verification process at the given block index
    double GuessVerificationProgress(CBlockIndex *pindex, bool fSigchecks) {
        if (pindex==NULL)
            return 0.0;

        int64_t nNow = time(NULL);

        double fSigcheckVerificationFactor = fSigchecks ?
SIGCHECK_VERIFICATION_FACTOR : 1.0;
        double fWorkBefore = 0.0; // Amount of work done before pindex
        double fWorkAfter = 0.0;  // Amount of work left after pindex (estimated)
        // Work is defined as: 1.0 per transaction before the last checkpoint, and
        // fSigcheckVerificationFactor per transaction after.

        const CCheckpointData &data = Checkpoints();

        if (pindex->nChainTx <= data.nTransactionsLastCheckpoint) {
            double nCheapBefore = pindex->nChainTx;
            double nCheapAfter = data.nTransactionsLastCheckpoint - pindex->nChainTx;
            double nExpensiveAfter = (nNow - da-
ta.nTimeLastCheckpoint)/86400.0*data.fTransactionsPerDay;
            fWorkBefore = nCheapBefore;
```

```
            fWorkAfter = nCheapAfter + nExpensiveAfter*fSigcheckVerificationFactor;
        } else {
            double nCheapBefore = data.nTransactionsLastCheckpoint;
            double nExpensiveBefore = pindex->nChainTx - da-
ta.nTransactionsLastCheckpoint;
            double nExpensiveAfter = (nNow - pindex-
>nTime)/86400.0*data.fTransactionsPerDay;
            fWorkBefore = nCheapBefore +
nExpensiveBefore*fSigcheckVerificationFactor;
            fWorkAfter = nExpensiveAfter*fSigcheckVerificationFactor;
        }

        return fWorkBefore / (fWorkBefore + fWorkAfter);
    }

    int GetTotalBlocksEstimate()
    {
        if (!fEnabled)
            return 0;

        const MapCheckpoints& checkpoints = *Checkpoints().mapCheckpoints;

        return checkpoints.rbegin()->first;
    }
```

```
    CBlockIndex* GetLastCheckpoint(const std::map<uint256, CBlockIndex*>&
mapBlockIndex)
    {
        if (!fEnabled)
            return NULL;

        const MapCheckpoints& checkpoints = *Checkpoints().mapCheckpoints;

        BOOST_REVERSE_FOREACH(const MapCheckpoints::value_type& i, checkpoints)
        {
            const uint256& hash = i.second;
            std::map<uint256, CBlockIndex*>::const_iterator t =
mapBlockIndex.find(hash);
            if (t != mapBlockIndex.end())
                return t->second;
        }
        return NULL;
    }
}
```