MultiChain:
A cybercurrency for cooperation

Steffan D. Norberhuis
steffan@norberhuis.nl

Delft University of Technology

MultiChain:
A cybercurrency for cooperation

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Steffan D. Norberhuis

7th December 2015

Author
Steffan D. Norberhuis

Title
MultiChain: a cybercurrency for cooperation

MSc presentation
December 14, 2015

Graduation Committee

dr. ir. J. A. Pouwelse	Delft University of Technology
dr. A. Iosup	Delft University of Technology
dr. ir. Z. Erkin	Delft University of Technology

Abstract

Peer-to-peer networks are often large, collaborative networks where peers can join openly. The essence of a collaborative, distributed system is that every node performs tasks for other nodes. The peers often help in singular interactions and without direct reciprocity. Malicious peers can abuse and freeride the public goods. The network without countermeasures can fall into a tragedy of the commons where no one helps another and everyone takes advantage of the generosity of peers. Only when the reputation of a peer is publicly available at scale and peers trust this reputation can the network escape the problems of freeriding and attain high utility for all participants.

This thesis focuses on designing and implementing the first step of a tamper proof reputation system within Tribler. Tribler is a peer-to-peer BitTorrent system developed at the Delft University of Technology. This first step, made by this thesis, is to create MultiChain, a proof-of-concept bookkeeping system. MultiChain tracks the upload and download amounts of peers to eliminate freeriding. MultiChain is cryptographically protected and validated. The bookkeeping system has to be scalable to be publicly available and be able to process enough transactions. The system has to work in an asynchronous network.

A new design of a distributed data structure that can be used as a ledger is introduced by this thesis. This first step with MultiChain is already more resilient to tampering than previous work, like BarterCast. BarterCast has no security measures against tampering records. The design of MultiChain is to have a chain of blocks for every peer as a ledger. Peers are participants of a peer-to-peer network. A block contains a transaction between two peers. This block is shared and added to both chains. This makes both chains of the peers intertwined and entangled at a shared block. The proposed design abandons the typical global, full ledger. The protocol of creating these blocks between peers is described. The problems faced by MultiChain in an asynchronous network are explained. The thesis proposes how the design can overcome these problems by only allowing atomic operations to be performed on the chain and to introduce unfinished blocks in the chain.

The implementation of the design is tested and experimented with within this thesis to validate it to work correctly. Furthermore, a number of weak points are discussed. These weak points have to be addressed in the future to create a tamper proof reputation system.

Preface

The huge increase in Bitcoin adoption, due to the increase in uncertainty in the security of traditional currency after the financial crisis in 2008, is only rivaled by the speculation of the enormous possibilities of the underlying technology, the block chain. The block chain is the first technology, seen with real world adoption, that allows to register transactions without a trusted third party. The block chain has several limitations in scalabillity that will limit Bitcoin in fully replacing traditional currency. It also limits block chain as a scalable base for a large scale reputation system, similar to a digital currency, with vast amounts of transactions. These developments provide motivation to research the properties and possibilities of a new distributed data structure: MultiChain.

I would like to acknowledge several people that helped me during my master's thesis. First I would to heartily thank dr.ir. Johan Pouwelse for his mentoring and helping me in setting goals and achieving these goals. Furthermore, I would like to thank dr.ir. Cor-Paul Bezemer for his guidance and help with getting to know Tribler. I also would like to thank Elric Milon and Lipu Fei for answering countless questions and helping in solving problems I encountered during the programming phase of my thesis. For the excellent feedback and help to improve my code I would like to thank dr.ir. Niels Zeilemaker. His support by adding functionality to Dispersy was also invaluable. Rob Ruigrok was doing his master's thesis at the same time and helped me to avoid problems he encountered before I experienced them as well and I am grateful for this help. I am also very thankful for the companionship I felt within my work room and I would like to thank especially Hans van den Bogert BSc, Niels Doekemeijer BSc and Ernst van der Hoeven BSc. You certainly made my work more enjoyable.

Steffan Derk Norberhuis

Delft, The Netherlands
7th December 2015

Contents

Chapter 1

Introduction

Tribler is peer-to-peer file sharing software developed at the Delft University of Technology for research purposes. Tribler was started in 2001. Tribler expands the BitTorrent protocol and has added multiple improvements on this protocol. The focus of Tribler is two fold:

- Make secure and private use of the Internet the default for every user.
- Make it impossible to shut Tribler down without shutting down the infrastructure of the Internet itself down.

A fully distributed program, without relying on any central component, is only able to achieve these goals. Tribler has been designed and build with this focus [1, 2]. A distributed network requires collaboration of it participants, called peers, to achieve success. A screenshot of Tribler can be seen in Figure 1.1. This master's thesis was conducted as part of the research mission to improve the collaboration of peers within Tribler.

1.1 Tribler

In peer-to-peer file sharing a node called a seeder uploads parts of a file to another node, the downloader. The seeders and downloaders constantly change and a node can be both at the same time for different, parallel connections. A seeder can become a downloader when he wishes to download other files and downloaders can become seeders, when they posses a different file someone else wants. The ratio between the total data downloaded and uploaded is called the seeding ratio [3]. The seeding ratio can be seen as an indication of the level of collaboration.

Uploading can be seen as an interaction of one node helping another node. Such interactions come at the cost of consuming bandwidth for both parties. However, there is only direct benefit for the downloader. The downloader receives a file he wants. There is no direct barter between the seeder and downloader. Because the seeder does not get anything in return for uploading the file. With only a very small chance, can a downloader also be a seeder for the original seeder [4].

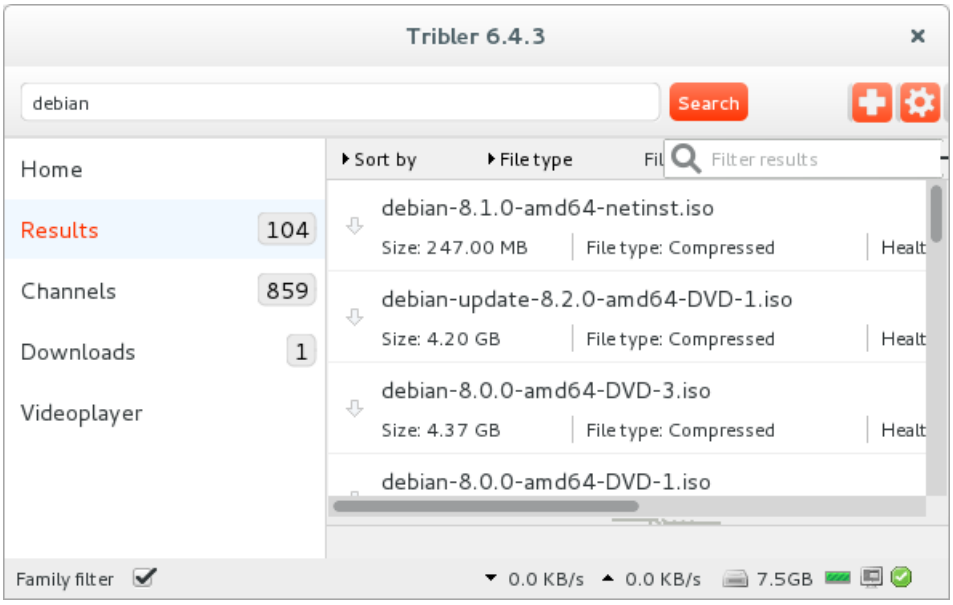


Figure 1.1: Screenshot of Tribler v6.4.3.

Both high availability and high download speeds result in a higher utility for the downloaders. If everyone contributes, files become more available and are downloaded at higher speed. This claim is supported by measurements taken in private communities [5]. In private communities, high seeding ratios are enforced by a tracker. Trackers are servers where nodes in a peer-to-peer network are introduced to each other, so that they can upload and download from each other [6]. This tracker can be seen as a central, third party.

But currently freeriding takes places in high quantities in public networks [7]. BitTorrent uses a Tit-for-Tat strategy [6] to stop freeriding, but this strategy does not effectively stop abuse [1]. The Tit-for-Tat strategy is to only provide help to peers to peers that return this help.

Tribler wants to achieve a high global seeding ratio by making it beneficial to have such a ratio. Nodes can award each other with higher cooperation if a node has a reputation of being cooperative, while malicious nodes are prevented from tampering and freeriding. Within Tribler anonymous connections have been implemented recently using onion routing [8, 9, 10]. This feature allows downloaders to become indistinguishable from other users in the network save guarding their privacy. Every data packet has to be forwarded by a number of intermediate hops between the downloader and seeder [8, 10]. The total cost of bandwidth per file is increased, because it has to be forwarded by multiple nodes. but also the number of nodes helping a single node downloading a file increases. The increase in nodes working together increases the necessity of an incentive system to reward collaboration.

Dispersy is middleware for data dissemination in a network. Dispersy is used

heavily within Tribler and is maintained by the Tribler organisation. Our work is build upon Dispersy. Dispersy is used to exchange data between two specific nodes [11]. Functionality was added to Dispersy during the thesis. The additions are described in chapter 4.

1.2 Document structure

In chapter 1 we give an overview of Tribler and the organization that develops Tribler. Chapter 2 gives a description of what the problem is with collaboration in peer-to-peer networks. Chapter 3 provides an overview of work related to solving the described problem. The design of the thesis work is described in chapter 4. The design is implemented, tested and experiments are run with the implementation in chapter 5. The design has known vulnerabillities and these are explained in chapter 6. Finally, we discuss the results of our work in chapter 7.

Chapter 2

Problem description

The essence of a collaborative, distributed system is that every node performs tasks for other nodes. With open systems, where every one can join, an unsolved problem is how to prevent nodes from only consuming and not contributing to the system. Our work is an important first step forward in solving this problem without a central component.

In this chapter, the problem will be described of how to decide to perform a task for a peer or not. The goal is to prevent that peer from freeriding. The problem will first be explained in a simplified form. Using this simplification, the problem will be transformed gradually into the real world problem faced in distributed systems today.

2.1 Freeriding public goods

One of the problems in collaborative distributed systems is nodes freeriding public goods. Nodes are participants in the system and the public goods are the willingness of these participants to help others. For decades the field of game theory has studied the problem of how participants can decide to cooperate with a peer or not [12]. Deciding not to help can prevent freeriding. The minimal form of this problem is with only two participants. This form is called the *Prisoner's Dilemma* [13, 4] and we will explain this dilemma.

Participants can help each other at a cost, a negative utility, but the recipient of the help will receive a beneficial utility from the help. The benefit received is defined to be greater than the cost and is denoted by R for reward. If one participant chooses to not help the other participant, he will still receive a beneficial utility and at no cost, T for temptation. The participant that provided the aid will now receive no benefit and only incur a cost C. If both participants choose to not help each other, they will both receive a penalty P, which is higher than the cost of helping each other. One participant will be called participant A and the other participant B. The utility received by A and B based upon their decision can be seen in Table 2.1.

This dilemma can be repeated several times with the same participants and is

A / B	cooperate	defect
cooperate	R_A/R_B	T_A/C_B
defect	T_A/C_B	P_A/P_B

Table 2.1: Prisoner's Dilemma utility matrix

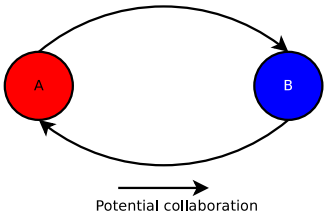


Figure 2.1: Direct Reciprocity between participants.

the *Iterated Prisoner's Dilemma*. Each time both participants will have to decide if they will help the other participant. A rational participant wants to receive maximum benefit at a minimal cost. The participant will follow a strategy that he believes will achieve this. At first it might seem that a participant will always choose to defect and freeride, because it will never incur a cost and receive maximal benefit. But the other participant will be reluctant to help a participant if the aid is never returned. Simple strategies, like tit-for-tat or win-stay, lose-shift, suffice in the Iterated Prisoner's Dilemma and perform well [14].

The participants receive *Direct Reciprocity* from the other participant [14]. Direct Reciprocity is displayed abstractly in Figure 2.1. Participant A helps participant B and in return receives help. This help will be extended in the future under the assumption it is returned.

In a large scale, distributed system this dilemma occurs with every interaction between nodes. The node can already be familiar with the peer, but more often the peer will be a peer the node has not interacted with before and will interact with in the future. A second complication is that help is almost always one way. Help cannot be exchanged and help now cannot be exchanged for help in the future [4]. A third complication is that an uncooroperative node does not incur a penalty for not helping. A node will not be incentivized to help. The performance of the tit-for-tat or win-stay, lose-shift strategies quickly deteriorate in such a situation.

For a node it is easier to abuse the generosity of others in the more anonymous situation of a large scale distributed system. Nodes can act irrational and against the best interest of the network and this is seen in the real world [7]. Nodes that help others will be penalized through the cost they incur and incentivized to adopt the malicious behavior themselves. Nodes in general will become more reluctant to help nodes [13]. In the end no node will help another and all nodes will receive a penalty. This is called the *Tragedy of the Commons* in the literature [12]. The whole network will actually receive more benefit if everyone would corporate, but nodes have no way of knowing if the peer they encounter are also willing to help. A

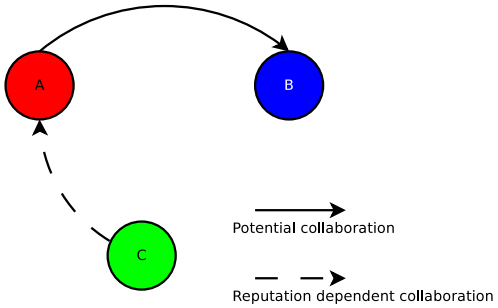


Figure 2.2: Indirect Reciprocity between nodes.

lack of altruism and selfishness will become a problem and will cause the network to be susceptible to freeriding.

2.2 Recording of community contributions

In the Iterated Prisoner's Dilemma the history of the previous transactions can be used to see if a node helped others in the past. The simple strategies previously mentioned use a history to improve performance and to prevent freeriding. If a node knows that a history is kept of his previous transactions by a peer, the node can be fearful of the consequences of his decision to freeride on public goods now. This is known as the *Shadow of the Future* [15]. The Shadow of the Future can overcome behavior to take advantage of the network.

But this private, individual kept history is not effective to achieve high overall benefit in distributed systems [4], because peers are often interacted with once. No history is known for these peers. A public history of a node is necessary to prevent abuse. The network will be able to achieve high overall benefit of the network [4]. This public history should contain every previous interaction. The nodes then receive *Indirect Reciprocity* [14]. By helping nodes they will receive help from other nodes in the future based upon their good behaviour. Indirect Reciprocity is displayed abstractly in Figure 2.2. The help of node C to node A is dependent on node A helping node B.

A history can be used to create a currency or a reputation system and can be seen as a bookkeeping system. This history has to be trustworthy. Here trust is defined as the expectation that the abillity to double spend reputation and being able to claim a false reputation are highly unlikely.

The node providing help will receive a boost in currency or a beneficial reputation. The currency or reputation can be used in the future to receive aid. In a currency system, receiving help will transfer currency to the helper. In a reputation system, only nodes with a sufficiently good reputation will be helped.

The currency or reputation has to be made publicly available to all nodes in the network. A node publicizing to hold a certain reputation is not sufficient as it is

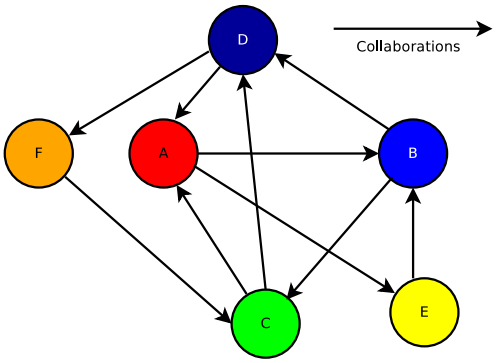


Figure 2.3: Network reciprocity between nodes.

not trustworthy. The reputation has to be verifiable and not self-constructed to be trustworthy. So an interaction history, that contains every prior interaction a node has conducted, is publicized. Nodes can calculate the amount of currency or the reputation that a node has. Based upon this calculation, the node can decide to perform a task or not.

2.3 Scalability

An interaction history has to be accessible by every peer in the network. The interaction history has to be distributed among the nodes in the network to become publicly available. The interaction history has to be distributable in an efficient manner. The overhead introduced by distributing the data of the interaction history should be minimal. Time to construct and commit a transaction should also be small. Every peer in the network should be able to gain knowledge of the reputation a peer holds and verify this reputation. The network will then be able to achieve *Network Reciprocity* [14]. Helping peers in the network will ensure that peers themselves receive help from other peers, because of their good reputation. This is displayed abstractly in Figure 2.3.

2.4 Facilitating trust

Interaction histories only prevent direct abuse of the generosity of the nodes. A malicious node can still try to tamper with the interaction history. An example of a type of these attacks are double spending attacks [16], where an interaction is altered. But a node can also try to deny an interaction. The interaction history has to be resilient to attacks that tamper with the interaction history, or no one will trust the history. A reputation or digital currency system requires a level of trust in the system to function. The challenge of this thesis is to create a tamper-proof interaction history.

A system cannot be fully secure, but still achieve a reasonable certainty such that no tampering of the interaction history can occur. Reasonable certainty can be for example that no attack can happen if more than half of the network consists of honest nodes.

2.5 Aim of this thesis

The overall aim of this thesis is to design, implement and conduct experiments with a tamper-proof interaction history in a distributed environment. The interaction history will store the history of the amount of data transferred between nodes in the network. This interaction history is cryptographically protected and validated. In the future this interaction history can be used to decide the level of cooperation with another node in the network. An incremental approach is taken with this thesis. This thesis is part of several more steps needed before a fully tamper-proof interaction history is created. Several problems with the interaction history are not solved. These problems are explained in chapter 6.

Chapter 3

Related work

In 2007, the notion of bandwidth as a currency was introduced. In 2009, the first Bitcoins were mined [17]. These events can be seen as the first steps in reputation or currency systems. Currencies have more strict rules to work correctly than a reputation system. In this chapter we will discuss related work on tamper-proof interaction histories in currency or reputation systems. The block chain used in Bitcoin is explained extensively, because there is much similairity between MultiChain. Understanding the concepts in the block chain helps understanding MultiChain. The differences between the block chain and MultiChain will be explained in section 4.1.2. Bartercast is the current implemented reputation system used by Tribler that will be replaced by MultiChain. Several other related systems are finally briefly discussed.

3.1 Block chain

Bitcoin is a digital currency that uses a global, full transaction history to keep track of transactions made between nodes. It is called a global, full transaction history, because the transaction history is shared between every peer and contains every transaction. The transaction history is a datastructure called the block chain.

The block chain imposes limits on Bitcoin in several ways. These limitations on Bitcoin can be seen as the initial motivation for our work. How Bitcoin uses the block chain technology will be introduced first. In the following sections the limitations imposed by the block chain will be explained.

We will only introduce how Bitcoins uses the block chain and why it does so. The best starting point for a full explanation of Bitcoins is the original paper by Nakamoto [16].

3.1.1 Transfer of ownership of a bitcoin

The core of the Bitcoin protocol is the block chain. The block chain contains transactions of bitcoins. We will first describe how the transactions are build up

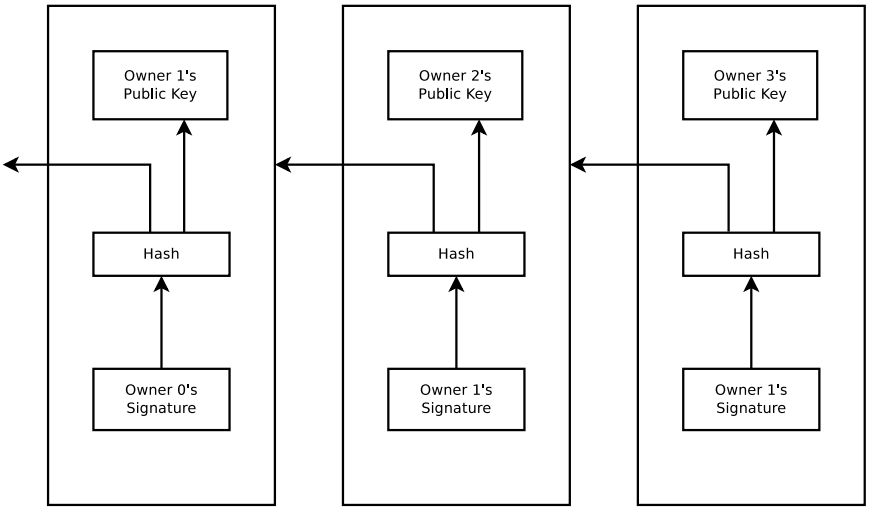


Figure 3.1: Transfer of ownership of bitcoin in a transaction chain.

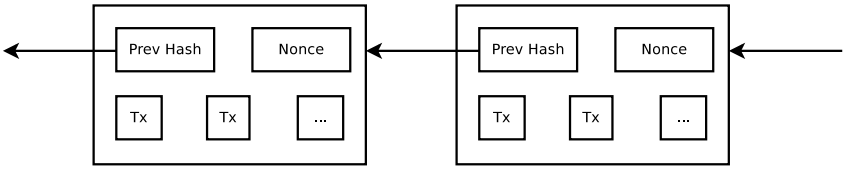


Figure 3.2: Block chain

and later how they are part of the block chain.

A transaction consists of three parts. The first part is the public key of the new owner of the bitcoin. The hash of the whole, previous transaction and the public key of the new owner is concatenated. This concatenation is hashed and this hash is the second part of the transaction. The final part is the signature by the current owner of this new hash. Inclusion of the hash of the previous transaction chains a transaction to the previous transaction.

The previous transaction is a transaction of the same bitcoin. The ownership of the bitcoin by the current owner can be verified by verifying the whole chain of ownership of the bitcoin. A transaction is usually shortend to Tx in Bitcoin related work and is used in images in this report. In Figure 3.1 a diagram can be seen of how transactions are chained.

3.1.2 Block chain

Multiple transactions are aggregrated into a single block. Every block contains the hash of the previous block. This creates the block chain. Transaction chains span across several blocks inside the block chain. A diagram can be seen in Figure 3.2 of the block chain.

These blocks in the block chain are created by nodes in the network, so called miners. A miner receives transactions from other nodes in the Bitcoin network. An attacker could malicously transmit transactions to double spend a bitcoin he owns or does not have. So every transaction is verified on arrival at a node. Any transaction that double spends a bitcoin is simply dropped by the node. No penalty is awarded to the malicous attacker.

The transactions are received in a non deterministic way induced by network characteristics. The non deterministic nature causes blocks to differ from miner to miner. The order of transactions has to be agreed upon by the network to eliminate this inconsistency.

Bitcoin uses election to pick the next block based upon a Proof-Of-Work system. A nonce is added to every block. This nonce is just a number that can be varied, but is only sound if the hash of the whole block starts with a certain number of zeros. Miners have to find the correct nonce for their block and this is a proof of work.

However, miners can still find a valid nonce at approximately the same time and notify parts of the network of their newly found block. This also leaves the network in an inconsisted state. Multiple versions of the next block attached to the previous block can be seen as branches.

To solve this inconsistency, Bitcoin nodes save both branches and continue using the longest branch. At some point one branch will become predominant in the network. More nodes will dedicate compute power to extend this branch and the growth rate will increase for this branch. The faster growth rate will ensure that the branch will be adopted by the network as a whole. The smaller branch is abandoned and the blocks are orphaned.

The amount of zeros, needed in the hash of the transaction, is adjusted to compensate for the fluctuating speed of the network to be able to find nonces. The speed of the network is called the hash rate and is the amount of hashes calculated per second. The amount of zeros balances the probability of branches occuring and the time before a new block is found, which in turn is how fast transactions are processed. The amount of zeros can be seen as the difficulty [18] of finding the a block. More zeros decreases the likelyhood a nonce will be valid. The estimated hash rate over time can be seen in Figure 3.3 [19], and the difficulty in Figure 3.4 [20].

Another possible attack to double spend a bitcoin is by sending a transaction to one part of the network, but to the other part of the network a transaction with a different receipient. It is possible that both transactions will be introduced into the block chain, but in different branches by two independend miners. Eventually one branch will win and the attack is averted.

The behaviour just described causes that a transaction can never be confirmed with full certainty. The possibility always exists that another branch over takes the current longest branch. This makes the network vulnerable if the total compute power is owned by a malicous attacker is more than the total compute power of the honest nodes, even if the attacker only has control of 51% of the compute power.

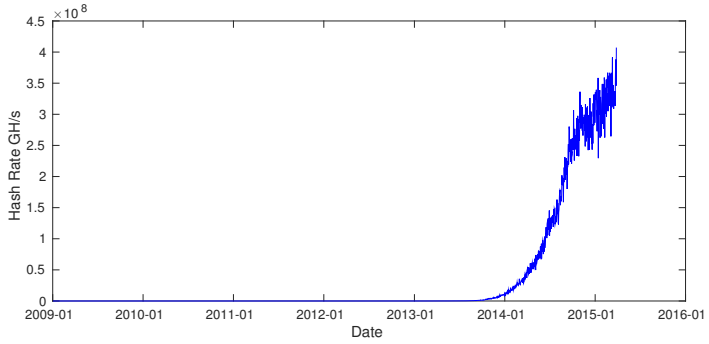


Figure 3.3: Estimated amount of hashes per second.

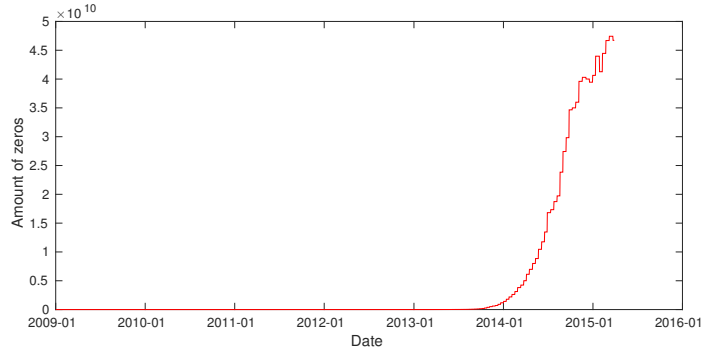


Figure 3.4: The difficulty of finding blocks.

In the end the current branch can be overtaken by a new branch started by the attacker. This new branch allows the whole transaction history to be rewritten by the attacker.

3.1.3 Limitations

In this section we will discuss the several limitations of Bitcoins that originate from the use of the block chain.

Size

To be able to prevent double spending and enable rightful spending by the owner, the node verifying a transaction needs to be aware of the full history of a bitcoin. This results in that a node needs the entire block chain.

The block chain is a data structure ever increasing in size. No block or data contained in that block is removed. The block chain has been growing since its inception in 2009. The size and growth can be seen in Figure 3.5.

The size of the block chain at time of writing already prevents less powerful devices, like smartphones, to operate on the block chain. This problem is only

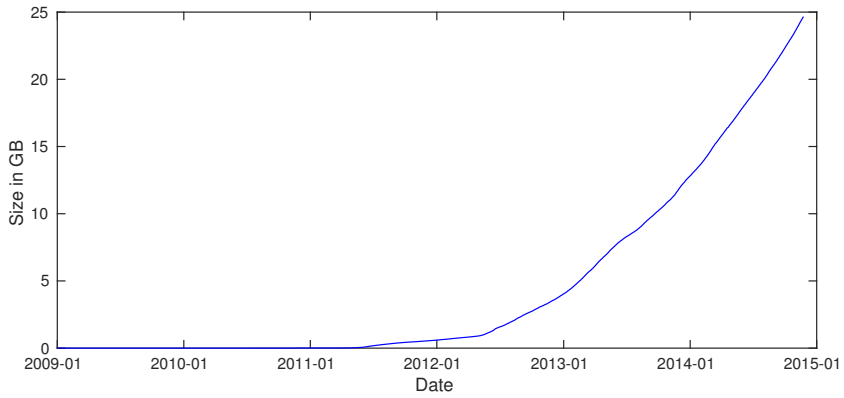


Figure 3.5: The size of the block chain [17].

going to become bigger with the continued creation of transactions and at a faster pace due to increased adoption of Bitcoin. The growth of the size of the block chain has already outpaced the growth of the power of a smartphone and will also be relatively larger compared to other types of hardware.

This problem was already identified by Nakamoto in his original paper on Bitcoin. The paper proposes Simplified Payment Verification (SPV). In SPV mode a node only downloads the block headers of the longest chain. If a transaction is to be verified, it requests from the network the specific transaction along with a Merkle tree linking it to a block in the chain. The Merkle tree can be used to verify that the transaction was included into the block chain. This allows to calculate with some confidence that the transaction was accepted.

SVP only gives reasonable confidence and is not as secure as running a full node. Trust has to be placed in the nodes that send the block headers and Merkle Trees. Secondly, a transaction that is recorded in a more recent block is less difficult to tamper with than a transaction deep down in the block chain. This is only an acceptable solution for clients willing to accept more risk due to having a less secure system. Therefor it is not a solution for the problem for every one.

Amount of transactions

The usability of a digital currency is in part determined by the time it takes to process a transaction and how many transactions can be processed in a certain time scale. The amount of transactions that can be handled by Bitcoins are determined by two factors:

- Block size
- Block creation time

The block size is currently capped at a fixed maximum size. A block can be smaller, but cannot exceed the maximum size. The current blocksize limit is 1MB.

A block has a fixed part and the rest is filled up by transactions picked by the miner. A transaction can vary in length. The number of transactions that can be fitted inside a block is limited by the maximum size.

This maximum size has no clear documentation of why it was picked as such. There are several implications of raising the size, aswell as lowering. Increasing the block size will increase the number of transactions that can be processed. But the increased block size will increase propogation time of the block in the network. A longer propogation time will in turn result in a higher orphane rate of the miner.

Large clusters of hash power owned by a single miner or mine cluster can be placed closer together reducing propogation time for this miner. This will benefit this single miner in reducing his orphaned rate and will increase the chance of his block being adopted. This will make the network as a whole more susptible to attack by a single powerful miner and will reduce the power of other miners.

The difficulty of finding a new block roughly regulates how fast new blocks are created. The difficulty is set according to the hash rate of the whole network to equal roughly a new block every 10 minutes. If the time of finding a new block is decreased, then more blocks are generated and obviously more transactions can be fitted inside these blocks. The reverse is true if the time is increased. But the time between blocks is a balance between the orphan rate of blocks and how fast transactions are committed to the block chain.

These two factors currently result in a theoretical limit of 7 transactions per second [21]. This can be calculated by dividing the maximum blocksize with the minimal size of a transaction. The performance of Bitcoin can be changed by changing the settings of these two factors. This limits the global usage of Bitcoins. In comparison, Visanet, that handles transactions under Visa, is able to process 54.000 transactions per second [22]. To be a real replacement of the current traditional currencies a higher number of transactions per second have to be achieved by a digital currency. There are at time of writing several competing proposals to increase the block size [23, 24].

3.2 BarterCast

BarterCast is the system used by Tribler to incentivise good behaviour by keeping track of reputations [25, 26, 27, 28]. It is fully decentralized, in contrast to previous reputation systems in a peer-to-peer network. Private BitTorrent communities, for example, depend on central servers to track reputation [26]. BarterCast was not designed with the aim to be fully resistant to malicious nodes that want to falsify their reputations. The initial version has been first deployed in June 2006 and subsequently has been improved. BarterCast will be briefly explained as well as its vulnerabilities.

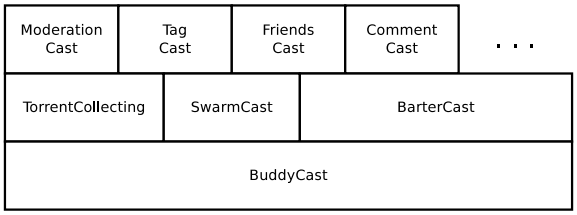


Figure 3.6: BuddyCast stack overview [25].

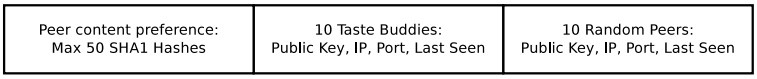


Figure 3.7: Format of initial messages in BuddyCast [25].

3.2.1 Epidemic protocol

At the base of BarterCast is BuddyCast. BuddyCast is an epidemic protocol stack and BarterCast is part as a protocol of this stack. A overview of the BuddyCast stack can be seen in Figure 3.6 and a more thorough introduction can be found in the report on BuddyCast [25].

An epidemic protocol works in a very simple way relying on a receive-and-forward primitive. Information is exchanged and forwarded between peers. This is called gossiping. BarterCasts gossips on the donation of upload bandwidth and the consumption of download bandwidth by other peers.

BuddyCasts spreads information of other peers with initial messages. This is done every 15 seconds, but a peer is only contacted every 4 hours to avoid contacting too often. These messages contain a prefered content list, a list of peers with similair prefered content and a list of random peers. For each peer a public key, IP adress, port number and last seen timestamp is provided. The peers with a similair preference are called taste buddies. A format of the these messages can be seen in Figure 3.7.

BuddyCasts uses several techniques to improve the peer selection efficiency. Peer selection efficiency is the percentage of successfully delivered outgoing messages to peers. This metric measures how well BuddyCasts selects peers on availablity and connectablity. These are important because they determine how well BuddyCasts is able to handle with peer failure and exit. Sending messages to an unavailable peer is a complete waste of resources.

The most obvious improvement that has been made is not to forward any offline peers. BuddyCast maintains a live overlay and continuously verify the online status of 10 random peers and 10 taste buddies. The random peers are updated to ensure they are in fact random peers.

Peers can detect their own connectivity issues. While they are able to connect outbound, no incoming connections can be excepted. These peers can improve the peer selection efficiency by broadcasting that they are having connectivity prob-

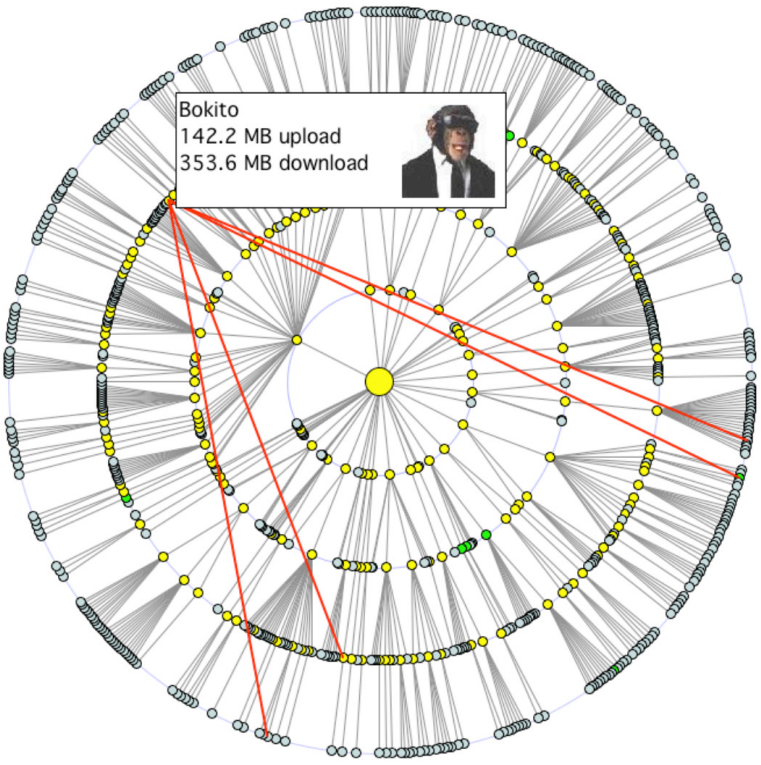


Figure 3.8: A screenshot of BarterBrowser in 2008 [25].

lems and instructing others to not gossip their identity.

Each peer collects information on how much other peers have downloaded and uploaded. This information is signed and a barter record is created. These barter records are forwarded to 10 peers. A peer has a chance to be random peer or a taste buddy. The records are forwarded by attaching them to BuddyCast messages.

The aggregrated information contains information about direct interactions with other peers, but also information about interactions between other peers gossiped. The information can be visualised using BarterBowser. A screenshot can be seen in Figure 3.8. The center contains the local peer. Every circle contains peers for a certain degree. The first circle from the centre contains peers that the local peer has direct interactions with. The second circle contains peers that peers from the first circle has direct interactions with and so forth. The figure shows how BarterCast accumulates data through other peers.

The information can be used to in conjunction with the maxflow algorithm to create a reputation metric [27]. The reputation metric measures how well or how poorly a peer has helped the network. The calculation takes into account that the difference between 0 and 100 MB is more significant then between 1000 MB and 1100 MB.

The calculation is done without relying on a third party. No authority verifies the

validity of the received information. Also there is no insurance that the information is complete.

Improvements have been done to improve the performance of BarterCast. A bloom filter algorithm is implemented to decrease the amount of duplicate barter records being sent to peers already having those records [29]. Bloom filters can be used to quickly determine with some certainty if a database contains records in limited space [30]. The bloom filter protocol works in the following way:

1. Peer A creates a specific Bloom filter for peer B.
2. Peer A sends the Bloom filter to peer B.
3. Peer B checks every record in its database using the Bloom filter.
4. Peer B sends any missing record to peer A.

The created bloom filters only contain relevant entries that have not yet been synchronized to peer B. BarterCasts knows which entries are relevant by keeping track of the last synchronization point and only using new barter records since that point.

3.2.2 Limitation

The main limitation of Bartercast is that the reputation is self-reported. This assumes the majority of nodes to be honest and to follow protocol. Barter records are not signed by any one else. So Barter records can be created by any malicious node without a limit. There is no way to verify these records and enforce truthfulness. Finally, there is no way to punish malicious nodes and therefore they are free to lie.

3.3 Other related work

The topic of reputation system is currently a subject of research for other projects as well and show the struggle to build a working reputation system. There are two projects worth to briefly mention, because of similairity to the Tribler project. These projects are not so thoroughly explained as the block chain as they were not used as a starting point in designing MultiChain.

The Tor project is working on a implementation of a reputation system in an effort to incentivize collaboration [31]. There are currently multiple proposals: PAR [32], BRAIDS [33], LIRA [34], TEARS [35], TorCoin [36], and XPAY [37]. The amount of proposals demonstrate the complexity of creating a reputation system in an anonymous system.

The InterPlanetary File System $IPFS$ is a peer-to-peer distributed file system with similairities to the Tribler project. IPFS uses an incentivized block exchange to improve collaboration [38].

Chapter 4

Design

This chapter presents the design of MultiChain and how it was implemented. The first thing that will be introduced is how MultiChain is implemented within Tribler to outline the context of the design and implementation. The contents of blocks are presented and how they form chains in MultiChain. We will cover the creation of blocks and the multiple implications of the chosen design. We will explain how MultiChain adresses the freeriding problem. Additional pheripheral systems that work with MultiChain are also explained.

Tribler uses communities to add functionalities to peers. New communities are packaged into new Tribler versions. These versions are downloaded by peers. A community provides a set of messages and endpoints for other peers. The community can communicate with the endpoints of other peers as well and send a message to these endpoints. Other peers are automatically discovered using Dispersy. Examples of communities are the TunnelCommunity that adds functionality to download anonymously [8, 10] or the AllChannelCommunity to distribute torrent files. Our system will be implemented by adding a new community to Tribler.

The MultiChain community can be run standalone, but its main use is to integrate with Tribler and track up and download amounts for torrents. It aims to replace the current reputation system Bartercast in the future.

4.1 Datastructure design

In this section we wil describe the design of the chains inside MultiChain. We will first describe the contents of a single block. Next, we will explain how blocks are chained together for a single peer. These chains are intertwined and we will clarify this entanglement. Finally, we will describe how the blocks provide security against tampering.

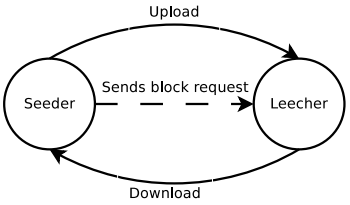


Figure 4.1: A seeder sending a protocol message to request a block.

Up	Public Key A	Total up A	Hash A	Total up B	Hash B	Signature A
Down	Public Key B	Total down A	Sequence number A	Total down B	Sequence number B	Signature B

Figure 4.2: All 14 fields in a MultiChain block.

4.1.1 Bookkeeping

A peer will upload to and download from other peers in the network. Peer A is a seeder and peer B is a downloader in the following text. A will want to increase its reputation and have a transaction transcribing his contribution. A initiates the process to create a block with peers that he has uploaded to. These peers will be referred to as B. This process can be seen in Figure 4.1 Every block only contains one transaction.

The contents of a block can be seen in Figure 4.2. The up and down represents the amount that has been transferred between A and B directly for the current transaction. Up means data uploaded by A to B and down the data downloaded by A from B. Every peer in Dispersy has a public and private key and are unique identifiers. A block contains both public keys of the peers, so it is possible to verify to which peers the transaction belongs to. The total amounts of A and B is added to the block. This is the amount summed across the whole chain of A and B respectively.

The blocks are linked to previous blocks by adding the hashes of the previous blocks of both peers. This creates a directed acyclic graph of blocks. An overview of a chain of blocks is seen in Figure 4.3. In this overview the chain of B is currently ignored. The first block references a special genesis hash identifying the block as the first block. A block has two sequence numbers, one for A and for B. These numbers allow blocks to be ordered without having to walk across the hashes linking the blocks. Both peers add their signature to the block to announce that they approve the contents of the block.

4.1.2 Scalable reciprocity

One of the main pillar of the design is to have a transaction history for every peer. The reasoning behind the idea to abandon a global, full transaction history is that

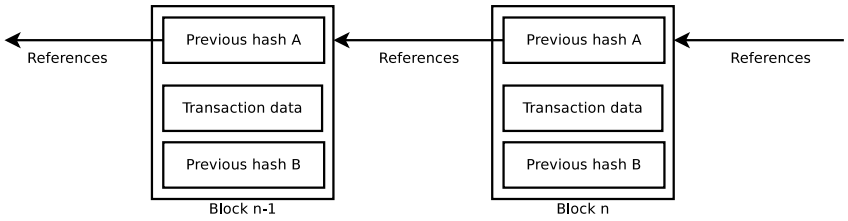


Figure 4.3: A chain of blocks with transaction data in MultiChain.

it will always become the bottleneck in the system. This will limit the amount of interactions that can be processed. Before this limit is reached, less powerful machines are already excluded in participating. For example, a global, full transaction history is the block chain of Bitcoins, discussed in section 3.1. The block chain already shows these limitations. Commonly, this way of design is used to prevent double spending. We believe that it is possible and much cheaper to detect and punish double spending than prevent double spending. This will be described in section 6.1

The reason for the limitation is that every interaction will have to be distributed to every peer in the network. Every transaction has to be processed by every node at the cost of bandwidth, computation power and storage. The cost might be very limited for a single transaction, but with greater scale these costs will add up when the amount of transactions are increased. The amount of these three resources is limited and will limit the amount of transactions that can be processed. This problem can be seen to affect Bitcoins and has been demonstrated in section 3.1.3.

Every node has its own transaction history and only needs the transaction history of the peers it interacts with. Within MultiChain peers can try to congregrate the full transaction history or only a subset This flexibility provides MultiChain with unbounded scalability for the network as a whole. Low-powered devices will be able to keep participating as long as they have enough power to process the transactions relevant to them.

4.1.3 Entanglement

Chains become entangled, because every block is shared between peers. Every block is present in two chains and references the previous block of node A and of node B. The chains quickly form a large graph with every node in the graph representing a block. Every node has two outward directed edges and two incoming directed edges except the nodes representing the genesis block and the most recent block. The outward edges point to the previous blocks and the incoming edges come from the next block in chronological ordering.

An simple example of three blocks can be seen in Figure 4.4. In this example the first block contains a transaction between peer A and B. This block references the previous block of both A and B. Peer A continues with a transaction with C

and peer B transacts with D. Now the previous hash of A and B is the same as they share a block on their chains. This hash is used in different transactions. Color has been added to denote the direction of the chain of peers.

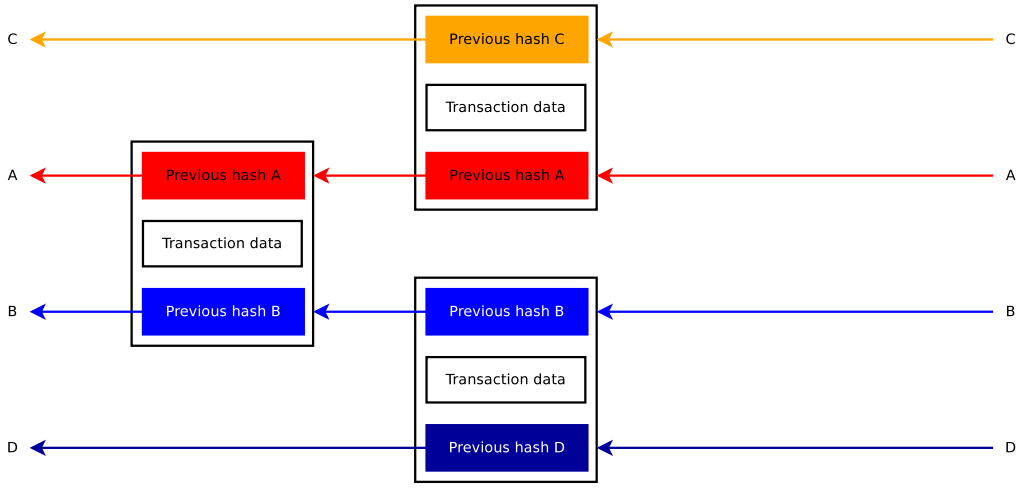


Figure 4.4: Entanglement of chains within MultiChain with four participants.

4.1.4 Protecting the blocks

The blocks have to be cryptographically protected and validated. Tampering with the contents of blocks should be detectable and useless. In part this is already done by using the hashes as pointers in subsequent blocks. If a block in a chain is changed, then the subsequent hash pointers will become invalid [39]. The chain becomes detached.

A second issue is that peers should not be able to deny conducting a transaction. This will prevent a peer to be able to deny his freeriding if this is transcibed in his chain. Both peers sign the block using their private keys to acknowledge that the transaction has happened. This signature is added to the block.

In a transaction between peer A and peer B, the whole block is not signed by every peer. Peer A can only make valid claims about, and have authority, over the interaction between peers and his own total up and download amounts. So A does not need to sign the data containing the previous hash and the total up and down of the other peer B. These amounts can be verified using the chain of peer B. The parts of what is signed by peer A and what is signed by B can be seen in Figure 4.5.

Digital signatures have the property to be non-repudiable of origin [39]. After signing a block the signer cannot later deny providing his signature. Only with the possesion of a secret key, a signature can be made; so only the signer could have made the signature. This is assuming the secret key was not comprimised. The blocks become durable records and are irrevokable and irrefutable. Because peers

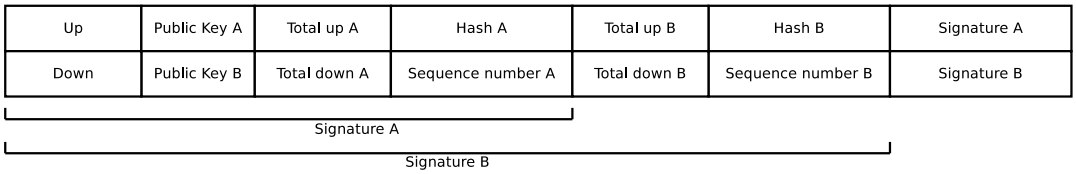


Figure 4.5: Fields protected by each of the two signatures.

cannot repudiate their own signature.

4.2 Block creation protocol

In this section we will explain how blocks can be created with a simple design that will limit and expose freeriding. Afterwards, we will explain a fundamental problem with creating blocks in an asynchronous system. This problem is present in the simple design and we will describe how MultiChain deals with this problem.

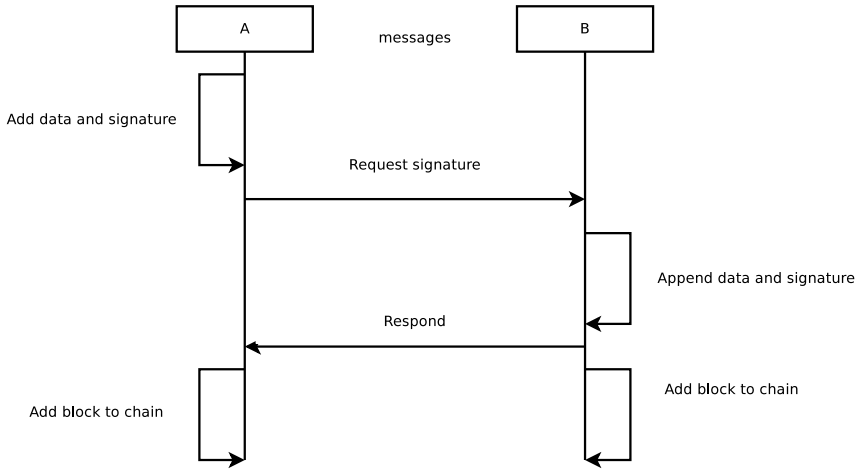
4.2.1 Exchanging signatures

Two peers in a network will create their blocks together without having to rely on a third party. Between the peers one is uploading to the other. The uploader is traditionally called the seeder in BitTorrent and the receiver of this data the downloader [3]. The seeder will initiate the block creation. co the seeder can decide how altruistic it wants to be towards the downloader regarding its collaboration. We will explain how the block creation protocol works. A sequence diagram can be seen in Figure 4.6(a).

The seeder, A, will create a packet that will be sent to the downloader, B. A will add to this packet the data uploaded and downloaded data between the peers that has not yet been added to the MultiChain. It will add these amounts to its total uploaded and downloaded data and add these total amounts aswell to the packet. Finally, it adds the public keys of both peers and its own hash pointer to the packet. This packet is signed using its private key and sent to the downloader. The data that A adds can be seen in Figure 4.6(b).

B will receive this packet and check if the amounts are correct, if the signature is correct, and if A has not used the previous hash before. If this is all correct, then B will add the amounts of uploaded and downloaded data to its own total amounts. The data contained in the previous packet, the total amounts of B and the hash of the previous block is inserted into a new packet. This packet is signed by the private key of B and sent back to A. The data that B adds can be seen in Figure 4.6(b).

Both parties now have the data of the block and can add this to their chain and continue forward. A does this upon receival of the block. B does this immediatly after sending the return packet to A. At this point a new block is created.



(a) Sequence diagram for block creation.

Up	Public Key A	Total up A	Hash A	Total up B	Hash B	Signature A
Down	Public Key B	Total down A	Sequence number A	Total down B	Sequence number B	Signature B

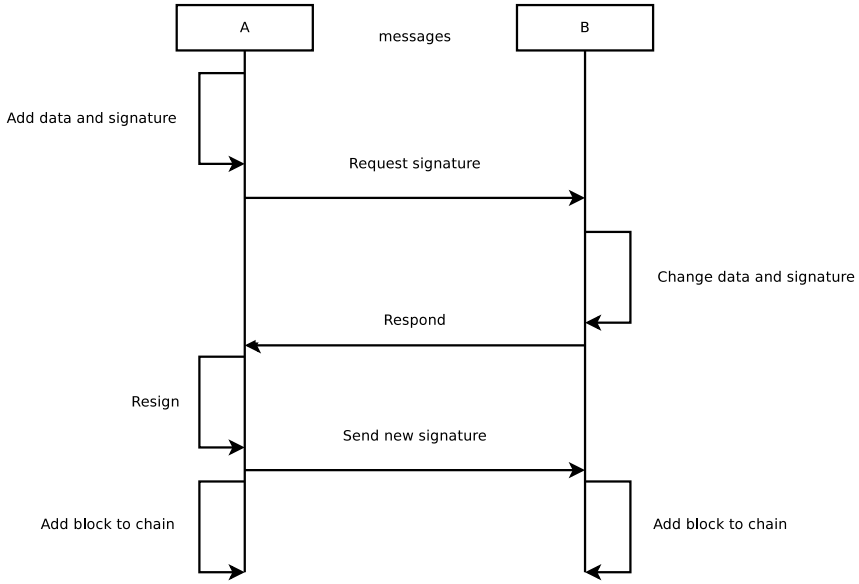
Added by A

Added by B

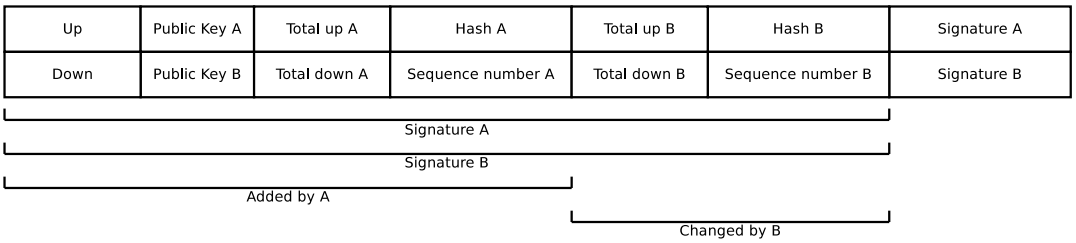
(b) Data added by peer A and B for a new block.

Figure 4.6: Exchanging data with our cleaner design in Dispersy.

Integrating with Dispersy



(a) Sequence diagram for block creation.



(b) Data added by peer A and B for a new block.

Figure 4.7: Exchanging data for block creation using the old design in Dispersy.

Within Dispersy functionality was already build to create a message, sign the message and request multiple nodes to also provide their signature on this message. This existing functionality could be used by MultiChain to exchange signatures between A and B for the creation of a new block.

A would initiate a message, insert its data into this message, sign the message, and send this message to B. The functionality would allow B to accept the message and provide its signature or modify the message and then provide its signature. Only B knows the hash of its head node, and the total up and total download metrics. So B will always modify the message and insert its own data in the message. But this would invalidate the signature of A, because the signature of A was also placed on the empty part of the message where the data of B is inserted. The contents of the message and who signs what can be seen in Figure 4.7(b).

After B returns the message, A would have to resign the message. But B also

needs this valid signature from A before it can add the block to its own chain. So A would need to send a third message to with the new, valid signature back to B. A sequence diagram can be seen in Figure 4.7(a) of how it would work in Dispersy. Functionality was added to Dispersy that allows to append data in a signature request. This allows the full signature exchange to be achieved within two messages.

4.2.2 Synchronization problem

This design does have one major problem with synchronization. It can never be decided in an asynchronous communication model if the other node will ever respond and when it will respond. The proposed block by A only needs an interaction of B to be finished and B can finish this transaction at anytime that B decides upon.

But the future block contains an immutable hash pointer to the previous block in the chain of node A. So while this transaction for a potential block is outstanding, node A cannot interact with any other node C to create a different block. If it would, then B could ultimately respond and a branch with different blocks would be created for the chain of A. This simple design introduces therefor an external response dependency problem.

The to-be-designed system has to be fault tolerant to common problems in a challenged network, such as node failure. A second design goal is to be able to process transactions quickly and on a large scale. These design goals prevent the adaption of a simple design that will halt until node B responds. This design would also result in a deadlock situation, as explained in section 4.3 We will explain three possible design solutions to this problem.

Fair exchange signature scheme

A fair exchange signature scheme (FESS) allows two players to exchange digital signatures in a fair way. Fair constitutes that no player can take advantage of the other. Either both players receive each others signature or no player receives the other's signature. It is infeasible for a node to acquire a signature without giving up their own signature [40], a FESS could be implement for the fair creation of the block.

But currently all known FESSs use a trusted third party (TTP) at some point [40]. Some schemes exist that are optimistic and will only need a TTP to resolve conflicts. But any TTP will not adhere to the Tribler philosophy of being a truly fully distributed peer-to-peer system. The TTP will introduce a central point of trust and a scalability bottleneck.

Reverse repairing of chain

Another potential solution would be to rework part of the chain. The creation of block could be deemed to have failed by node A. If at a later point the block is

still created by B, then the block could be reworked back into the chain. This would require every hash pointer to be updated afterwards. If these hash pointers are modified, then the signatures are invalid. Every signature would have to be renewed and requested at every node. This is unscalable if nodes needs to rework many blocks and can clog the system. These nodes can possibly no longer respond and would result in new invalid blocks.

Half signed blocks

The chosen solution is to keep it simple and allow for inconsistencies between the chains of A and B. It will be shown that when an inconsistency does occur, this is in favor of A.

When node A sends a signature request to B, then A will be optimistic and will wait for B for a timeout period. During this time, A will not interact with another node C. But if A does not have to interact with node C after the timeout period, then A will keepw waiting for a response of B and accept the response. There are three potential scenario's of what will happen to the message.

1. In the normal situation, node B will respond in time to the message. B will send back the message and both A and B will have added the same block in their own chains. This situation also occurs if A has timed out, but has not yet interacted with another node C.

2. Node B can be malicious or simply have failed and B will never respond. Node A will timeout, add a half signed block to the chain and can continue interacting with other nodes C. A will have to decide how to react upon B not responding. A possibility would be to stop helping B.

3. The response of B can also be late. After B has received the request by A, B will create a block and adds this block to its chain. B will send its response to A, but the message will arrive too late at A. A will have experienced a timeout and will ignore the message. The block is not added to the chain of A.

The last scenario is the most complex and requires additional clarification. A initiates the signature request as a seeder and therefore the block will be favourable for its reputation. Because this is a reputation system, this inconsistency is an allowable comprimise. In a currency system, this comprimise would not be allowable.

The design is scalable as it will not force a node to wait too long for a failed node and will try to interact as fast as possible with other nodes. A small grace period can be used to allow node B to have enough time to respond in a normal manner. No trusted third party is introduced so a distributed design is achieved.

The contents of a half signed block can be seen in Figure 4.8. The total up, total down, previous hash, sequence number and signature of B are empty in an half

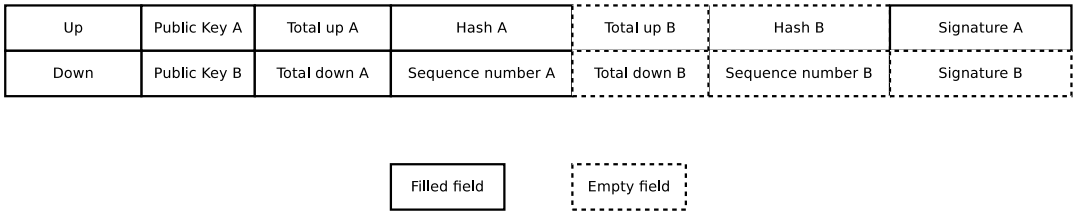


Figure 4.8: The fields of a half signed block in MultiChain. Dotted fields are empty.

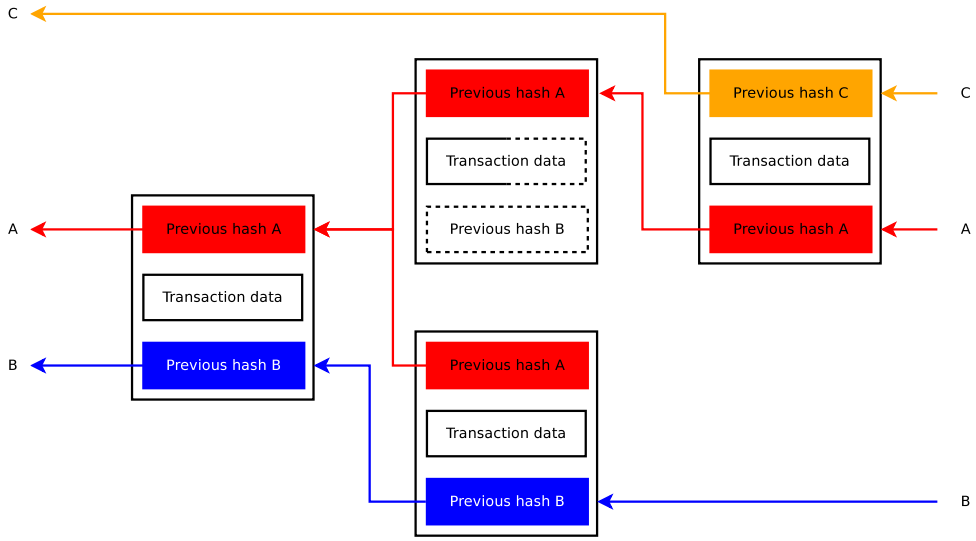


Figure 4.9: A chain of with a half signed block in MultiChain.

signed block. These are normally filled in by B and A does not have knowledge of these fields.

An example of the chain with an half signed block can be seen in Figure 4.9. In this example peer A initiates two block creations with B. The second block response does not arrive on time at A and A has timed out. The block cannot be processed as A has already interacted with C. The half signed block is never referenced again by A. B does continue using this block. This is not double spending because A does not reuse the same previous hash. The difference in block is only the addition of the data of B. This changes the hash of the block and is why B and A have a different hash.

4.3 Atomic operations on the chain

In a chain, only a single block of a peer is allowed to point to a previous block. No side branches are allowed. A peer cannot have multiple blocks belonging to him all pointing to the same previous block. As this is a potential attack as described

in section 6.1. The chain of a peer can only be moved forward by a single block at any time. Only after a new block is created, will the new hash available to be used in the next block.

To ensure this happens correctly, the MultiChain community contains mutual exclusive code that excludes any new operations on the chain if an operation is already pending. The mutual exclusion is achieved by having to acquire an atomic token to allow to perform an operation on the code. The MultiChain community will receive incoming signature requests or requests by other parts of Tribler to send out an outgoing signature request. The community will decline this request if the token is not available and return execution to other parts of Tribler. The token can be unavailable while waiting on another peer in the network to finish responding to a signature request.

When a MultiChain peer has a pending signature request, then the peer itself will not respond to incoming signature requests from other peers. The requests are dropped and we call this a drop event. These peers themselves will also not respond as they have a pending signature request. This can create a circular dependency on the availability of the token. If two peers send a request to each other at the same time, they will wait on each other. This could result in a deadlock.

MultiChain prevents this deadlock to occur by allowing a transaction to fail as explained in section 4.2.2. If MultiChain gets into this potential deadlock one of the peers will eventually time out of their own signature request and process the incoming request resolving the circular dependency. The deadlock is recovered and both peers can continue operation. This situation has occured during experimentation and it is explained in section 5.4. It is shown that MultiChain correctly recovers the potential deadlock. A potential attack vector is explained in 6.3

4.4 Block storage

The blocks in the chain have to be persisted to be usable over a prolonged time. A storage layer is added to the MultiChain community that provides all functionality to persist blocks and query blocks. This layer extends and uses functionality of the Database class in Dispersy. An overview of the layering in the software architecture can be seen in Figure 4.10.

The MultiChain Community calls functions in the storage layer that have implicit knowledge about the model. The storage layer formats SQL queries and passes these to the Dispersy layer. The Dispersy layer performs several sanitation checks and passes these queries to the SQLite Library. The SQLite Library and Dispersy layer both return the result of the SQL query. These results are transformed by the storage layer into objects of the model usable by the MultiChain Community.

The only information that is saved are blocks. The information all fits within one table. A single block is saved as a single record called a row in a relation database. Every attribute of a block is a single column in the row. All attributes are saved

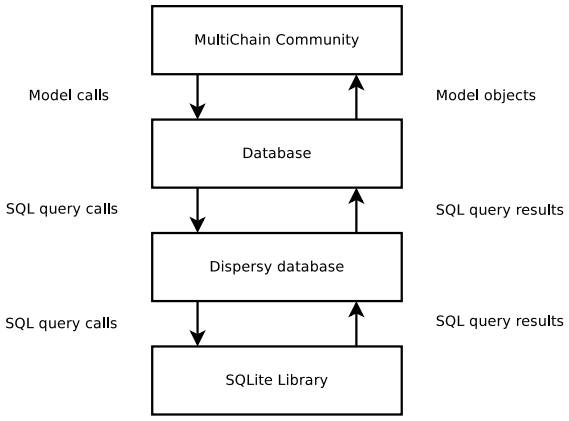


Figure 4.10: Storage layering in the software architecture

directly into the database, except for the public keys. These public keys are hashed and these hashes are used as an identifier, called mid, in Dispersy. The public keys are already saved in the Dispersy database. When a block is retrieved from the database the public key is retrieved from Dispersy using the mid.

Every attribute is queryable in the database. A public key can be converted to mid and is searched this way. Every attribute is queryable to make the system extensible and usable when the next incremental steps are implemented. It is presently unknown what information precisly will be needed, so every information is now made available for the future.

4.4.1 Dispersy database

Dispersy keeps track of information on its own. A record is kept of any message that can be retrieved using a message id. The message is saved in a converted format and will be decoded when the message is retrieved.

Instead of storing information in a separate database, the information could have been retrieved from the Dispersy database. But the Dispersy database is not queryable. Because all the information is stored in a converted format that prevents queries to search the message for its contents. For this reason, the dispersy database is not used and a separate database is used.

A future, possible improvement to Dispersy would be to save messages queryable in its database. This would eliminate the current need for separate databases that contain aggregrated information. The information is stored in two places within Tribler and this could be eliminated. It would reduce the disk footprint and the amount of read/write transactions as only one database would have to be maintained. The I/O ineractions are a problem according to Tribler maintainers.

Name	Type	Bytes	Name	Type	Bytes
Uploaded MBytes	unsigned integer	4	Downloaded MBytes	unsigned integer	4
Total Up A	unsigned integer	8	Total Up B	unsigned integer	8
Total Down A	unsigned integer	8	Total Down B	unsigned integer	8
Prior Record A	SHA1 digest	20	Prior Record B	SHA1 digest	20
Sequence Number A	signed integer	4	Sequence Number B	signed integer	4
Public Key A	EC key	64	Public Key B	EC key	64
Signature A	EC signature	40	Signature B	EC signature	40
Total		148	Total		148

Table 4.1: Block subcomponents size.

Name	Type	Bytes	Name	Type	Bytes
Uploaded MBytes	unsigned integer	1,2,4	Downloaded MBytes	unsigned integer	1,2,4
Total Up A	unsigned integer	1,2,4,8	Total Up B	unsigned integer	1,2,4,8
Total Down A	unsigned integer	1,2,4,8	Total Down B	unsigned integer	1,2,4,8
Prior Record A	SHA1 digest	20	Prior Record B	SHA1 digest	20
Sequence Number A	signed integer	4	Sequence Number B	signed integer	4
Mid A	SHA1 digest	20	Mid B	SHA1 digest	20
Signature A	EC signature	40	Signature B	EC signature	40
Total (min,max)		87,104	Total (min,max		87, 104

Table 4.2: Block subcomponents size inside the database.

4.5 Block size

The size of a block is important as it determines how fast the total size of a chain will grow and the utilization of the bandwidth of network to transfer blocks. The size is dependend on the size of the parts of the blocks and can be determined by the choice of cryptographic primitives. Hashing function turn a variable message string to a fixed size message digest [39]. The fixed size is dependend on the choice of hashing function. Similair cryptographic signing functions deterimine the size of a signature. An overview of the size of the parts of a block can be seen in Table: 4.1. The total size of a block is 296 bytes.

An overview of every attribute and the size of the attribute inside the database can be seen in Table 4.2. SQLite can grow the size of integers in its database to fit the required size and therefore it can be in the range of $1, 2, 3, 4$ bytes. Because of this dynamic allocation and because public keys are stored in the Dispersy database the total size is smaller.

Upper limit uploaded and downloaded MB

The maximum integer size of the total up and total down impose an upper limit on the total uploaded and downloaded MB that MultiChain can track. This limit is

imposed by SQLite and is 2^{62}. Bigger numbers cannot be saved in SQLite using a native datatype. The upper limit for MultiChain is therefore 4.612×10^9 petabytes. This is clearly more than sufficient for now.

4.6 Integration with Tribler

For integration a scheduler is implemented between the MultiChain community and other parts of Tribler. The scheduler tracks session upload and download amounts and schedules a block to be created, when the amount of uploaded bytes is above a certain threshold. The scheduler currently only tracks traffic of anonymous downloads. Bartercast is not yet removed and currently MultiChain runs together with Bartercast until MultiChain fully replaces Bartercast.

We recommend that the scheduler should be expanded to schedule blocks in a more sophisticated way. The functionality of the scheduler is very limited and is missing basic functionality. The most important improvement that should be introduced is the punishment of not signing blocks. Currently, nodes can deny to have their behaviour tracked. The next improvement is to actually determine the level of cooperation a node receives based upon their previous behaviour. The actual decision making based upon past behaviour is not part of the thesis. The processes of actually making the descision is a complicated process that has to take into account a lot of variables and requires extensive experimentation to validate.

4.7 Crawler

We implemented a crawler that visits other nodes and request the full chain of that node. The crawler was built to be used for the experiments and is a first step in a more sophisticated crawler that will help to solve the known vulnerabilities. These vulnerabilities will be described in chapter 6.

4.7.1 Recursively request blocks

Dispersy provides a list of other nodes that were recently found and can report when the node itself is found by another node. Both are sources of destinations nodes that the crawler will visit and request the chain from.

The crawler will first request from a node the block with sequence number -1. This denotes that he wants the latest block in his chain. The node returns this block to the crawler. The crawler will persist the block if it is not yet know.

The newly retrieved block is chained to two blocks with the previous hashes. The crawler will check if these blocks are present in the database. If any block is not present, then the crawler will request that particulair block. The peer, to whom the block belongs to, has to be known in Dispersy. If the peer is not known, the block is ignored. This is done recursively until the crawler reaches the genesis

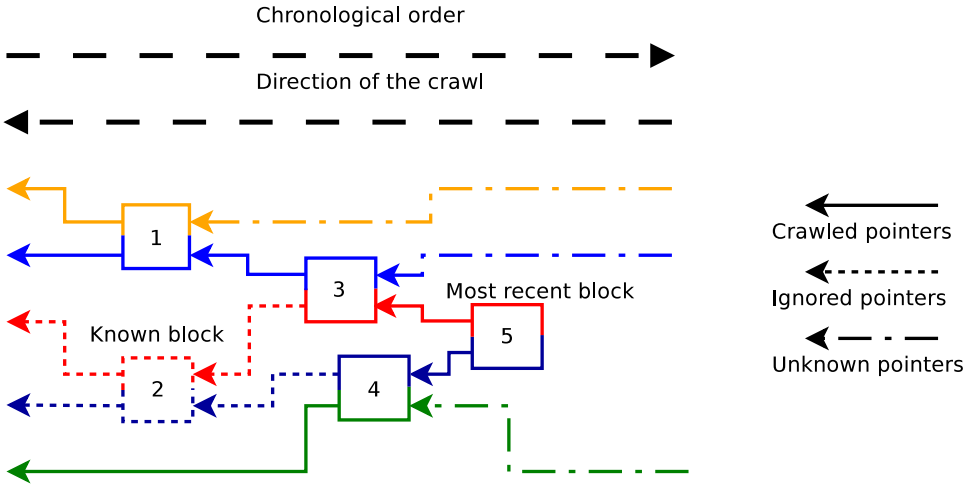


Figure 4.11: Example of the crawler looking for unknown blocks. The crawler retrieves the most recent block and crawls older blocks.

block of the chain. In this fashion a breadth first search is implemented for any unknown block that is present in the chain before the latest block.

The crawl tries to aggregrate as much blocks as possible. But it gives no certainty that the full MultiChain is collected. The crawler is able to crawl a disconnected MultiChain, if for every disconnected partition a peer is known by Dispersy.

An example can be seen in Figure 4.11. In this example the line arrows denote paths that the crawler follows, dotted lines are paths that the crawler ignores, half dotted lines are paths that the crawler will not know about. Block 2 is already known by the crawler. Block 5 is retrieved first by the crawler and the crawler sees hash links to block 3 and 4. These are retrieved and the block finds links to block 1 and 2. Because block 2 is already known, it is ignored. Only block 1 is retrieved. The crawler continues to follow the links further outside the displayed example. The half dotted lines are not know by the crawler until a block is retrieved that contains these paths. The crawler will retrieve these blocks if for example the most recent block is requested.

4.7.2 Recreation over retransmission

An effort was made to try to reuse code of the community for the crawler and to not introduce another payload type in the community. The attempt would reuse payload classes and authentication classes already used in the community for the creation of blocks. This effort failed, because Dispersy cannot handle recreation of messages well and would invalidate blocks that were recreated.

As said before, Dispersy also keeps tracks of messages received. The messages containing a block requested by the crawler could be retrieved and retransmissioned forward instead of being retrieved from the MultiChain database itself. This

would eliminate the need to construct a message and encode the message before it could be send, because the encoded format is saved and can be retrieved and send immediately. This was not used as it would be impossible to distinguish messages received as an response to a signature request or a crawler request. The two types of responses cannot be processed in the same way. A response to a signature requests has to influence the way a node responds to interactions and a crawler response should not.

In the end, the crawler uses recreation of a block from the local database and uses a new payload type to forward blocks. The main reasons this implementation was chosen is that implemenation was very simple and had none of the above mentioned problems. Maintainabillity is also much easier this way as different types of messages are not using the same functions to be received in the code. This might not be known by a new programmer working with the code.

4.7.3 Improvements

The crawler is a first, simple step towards a more sophisticated crawler. Tribler has implemented already more sophisticated crawlers for Bartercast and these techniques can be reused for the MultiChain crawler. For example bloom filters can be used in conjunction with the knowledge that every record is a part of the chain to quickly request multiple blocks [30, 29]. Blocks could also be send in a more efficient way by sending multiple blocks per message. Secondly, blocks that belong to a different node than the crawled node are requested, but the location of this different node is only known by chance. Dispersy only keeps track of 20 peers at any time, so the chance is low that the node is among these nodes. The chance can be improved by asking if the first contacted node knows the location of the different node.

4.8 Privacy

Tribler has made an effort in providing a way for users to download anonymously and privately [8, 9, 10]. There is privacy in the contents of what is downloaded and anonimity in what is downloaded. MultiChain should not leak information to break privacy and anonymity.

MultiChain only interacts with peers that are directly downloaded from or uploaded to. With these peers blocks are created and transferred. So anonymity is not broken by MultiChain. An attacker could already analyse network traffic between these peers and conclude they are downloading and uploading to each other. But Tribler does not guarantee anonymity for this. So MultiChain does not break anonymity with its interactions with peers.

A block only transcribe the amount of data that has been transferred between peers. The content of the actual data is not transcribed. The block also does not leak information of how big a single, individual transfer is between peers. The

amounts are aggregrated over all transfers. The blocks also does not break the privacy. Network analysis can already measure the total amount of data transferred. So the design and implementation of MultiChain does not break anonymity already guaranteed by Tribler.

But MultiChain does make network analysis easier as measurement points between every node do not have to be introduced. The chain of every node can be requested. The chain contains the data of every transaction of a peer and can be used to analyse the network. The network analysis can now also be performed retrospectively using data acquired from the MultiChain.

A reputation system can improve privacy. The reputation system can be used to indentify nodes with high flow of data traffic. These nodes can be used as a hop to send data to a destination. Because the nodes have more cover traffic, they will provide more privacy[41]. In the absense of high traffic and high collaboration, MultiChain should balance punishment of freeriding with the wish for using the freeriding traffic as cover traffic to increase privacy[42].

In the future we recommend to be able to transfer reputation using cryptographic functions in a new type of block: a transfer block. This transfer block identifies a portion of reputation and transfers this from one entity to another. This makes the reputation fungible and provide better safegaurds [41], like forward anonymity. Reputation can be earned and spend in an other place. Reputation mining can be performed by collaborating with other peers.

Chapter 5

Implementation and experiments

The design of MultiChain in the previous chapter has been fully implemented. In this chapter we test if MultiChain is correctly implemented according to the design and we experiment the MultiChain in various scenarios. MultiChain is experimented with to test if it can correctly create a chain tracking a download. Next, MultiChain is experimented with tracking anonymous downloads. Finally, MultiChain is experimented with in potential situations where a deadlock may occur to test if it succesfully resolve these situations. We first introduce a several aspects of our experimental setup.

Graph visualization

In this chapter multiple graphs are depicted. The graphs are the MultiChain generated by an experiment. The blocks are depicted as nodes and the previous hash pointers are edges in the graph. These graphs are generated by reading every database of every node. A graph exchange xml file is created and this is loaded into Gephi. Gephi is a graph visualization tool.

The nodes have added colouring to indicate extra meaning. This can be seen in Figure 5.1. Green nodes are a first block in a MultiChain of a peer, and as such have no inbound arrows. Blue nodes are a sequential block between the same previous peers, therefore they do not have two inbound arrows. Red nodes are half-signed blocks, and therefore only have one inbound and one outbound arrow.

Synthetic setup

In this chapter we perform several synthetic experiments. In these experiments we simulate downloads being run within Tribler using scenarios files. These files are sometimes generated using a special script we implemented and can be over 20000 lines long. Every second the download is simulated to have progressed with a preset speed. This progress is reported to every peer and their MultiChainScheduler. In the real world the progress is updated more frequently but in much smaller increments. The MultiChainScheduler waits for a certain amount of data transferred

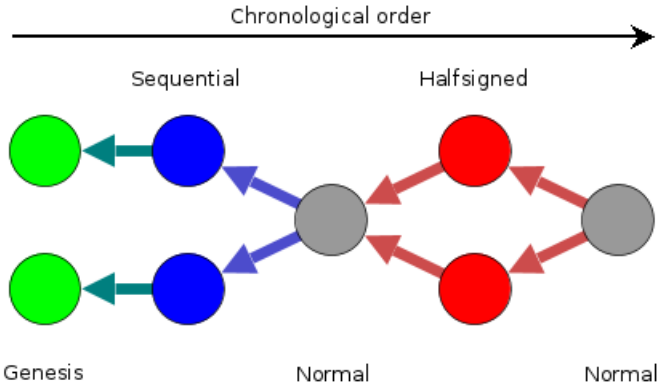


Figure 5.1: Example of coloring used in Figures with graphs.

before it schedules a block to be created. So the interval introduced by the scenario files is realistic and does not influence the outcome of the experiments. In subsection 5.1.2 we explain more about the experiment environment.

Connectability problem

Some experiments were also run multiple times because Dispersy does not always connect to every peer. At the start of the experiment the peers are forced to be introduced, but this does not always succeed. This is a problem in discoverability and has been experienced by previous work aswell [9]. The final version is chosen when Dispersy did connect all the peers. We recommend that Dispersy is changed to allow a modified behaviour when experimenting where the pool of candidates is not refreshed, as explained in section 3.2

5.1 Software engineering tests

MultiChain is tested in several ways to verify it is correctly working following standard software engineering practices. The enforcement of these types of tests is a policy recently introduced within Tribler.

5.1.1 Unit Tests & Integration tests

Tribler uses Python unit tests to validate small components of code. The tests can be run locally and are automatically run on a Jenkins build server [43, 44]. Unit tests were added to increase stability of MultiChain. Integration tests were added to test multiple components of MultiChains working integrated together.

Tribler does prove to be hard to test using unit tests. This is due to high coupling of code within Tribler. But mocking of classes helped in testing difficult to test code. The separate unit tests for the conversion, payload and database were

Filename	LOC	% covered	Conditionals	% covered
Community	187	81%	37	95%
Conversion	60	100%	6	100%
Database	100	100%	6	84%
Payload	89	100%	2	100%
Total	672		47	

Table 5.1: Unit tests coverage of MultiChain.

the first of it types inside Tribler. Also Dispersy was expanded to introduce new functionality to make it easier to create unit tests for communities in the future. An overview of the coverage can be seen in Table 5.1 In the end a high level of testing has been achieved, especially in comparison to the rest of Tribler. The total test coverage of Tribler was only 16% at time of writing [44]. A decision using code coverage tools was made that the untested code has little value to further test in comparison to the work required. The code was tested in Gumby scenarios.

5.1.2 Gumby

Next to that, Tribler uses a homemade testing environment Gumby. Gumby can start multiple instances of Tribler and follow test scenarios. Gumby can be used to perform system tests and experiments. These system tests have to be manually validated. Several scenarios have been written to validate MultiChain. These run MultiChain either in a standalone version or integrated into the TunnelCommunity.

One of these scenarios can be found in Figure 5.2 In this example basic block creation is tested. Normal situations are tested, but also situations where the signature requests are answered late and other requests arrive at the requesting peer at the same time. Additionally, signature requests are controlled to not be answered at all. During the whole test the crawler is active and scrapes the network for unknown blocks. The notation in the scenario is the time an action has to be taken place, the action that has to be taken, and by who if necessary. The "@0:" can be ignored, but is required in the Gumby format.

5.2 Tracking download and upload amounts

In this section we will experiment with MultiChain creating blocks and tracking download and upload amounts of peers. We will first start with creating a simple block in an experiment. The next experiment is to try to create a chain of blocks with simulating a download of 10MB and a larger download of 10GB. Downloads are done at different download speeds and an experiment was done to test MultiChain in this environment.

```
@0:0 set_master_member 3081a7301006072a8648ce ... 2b51
@0:0 set_community_class MultiChainNoResponseCommunity {4}
@0:0 set_community_class MultiChainDelayCommunity {5}
@0:0 set_community_class MultiChainCommunityCrawler {6}
@0:0 set_community_class {6}
@0:0 start_dispersy
@0:1 online
@0:5 reset_dispersy_statistics
@0:10 annotate start-experiment-1-peer
@0:15 introduce_candidates
@0:80 request_signature 2 {1}
@0:84 request_signature 1 {2}
@0:94 request_signature 4 {1}
@0:95 request_signature 1 {3}
@0:104 request_signature 5 {1}
@0:106 request_block 1 5 {6}
@0:110 close
@0:111 stop_dispersy
@0:112 stop
```

Figure 5.2: One of the Gumby definition files.

5.2.1 Single block creation

In this experiment we try to create a block between two nodes. This experiment validates the MultiChain to be able to correctly create a block between nodes in normal circumstances. The experiment is run using gumby with all nodes running on a single computer. Only two instances of MultiChain communities are started and between these two communities a block is created. The logging of the both nodes is captured and recorded to verify the results of the experiment.

The output of the logging can be seen in Figure 5.3. First node 1 sends a signature request to node 2. This message is received and a block is persisted. The hash of the block is displayed in the output. The block is sent back as a signature response to node 1. The block is saved by node 1 and has the same hash as shown in the output. So the block between node 1 and node 2 is the same and a block was succesfully created. The result of the experiment were also validated using the databases of both nodes.

The output also shows behaviour of MultiChain to correctly exclude any other execution from entering mutual exclusive code. The lines related to the mutual exclusion are prepended by ”Chain Exclusion”. The nodes check if it is possible to enter the mutual exclusive part and correctly acquires and releases the mutual exclusion token.

```
1: Requesting Signature for candidate: 2
1: Chain Exclusion: signature request: False
1: Chain Exclusion: acquired, sending signature request.
1: Sending signature request.
2: Received signature request.
2: Chain Exclusion: process request: False
2: Chain Exclusion: acquired to process request.
2: Persisting sr: 2F7bTMxyJU7hZkvaBimT2bYm4bY=
2: Chain Exclusion: released after processing request.
2: Sending signature response.
1: Signature response received. Modified: True
1: Valid 1 signature response(s) received.
1: Persisting sr: 2F7bTMxyJU7hZkvaBimT2bYm4bY=
1: Chain exclusion: released received signature response.
```

Figure 5.3: Output of single block creation experiment

5.2.2 Chaining blocks

In the next synthetic experiments we show that MultiChain can create a chain of blocks between two peers. The experiments are run using gumby. In the first synthetic experiment we try to create 10 subsequent blocks simulate a download of 10 MB with a speed of 1000 KB/s. The scheduler waits for 1 MB uploaded to another peer before scheduling a block. The result of the experiment can be seen in the graph in Figure 5.4. In this graph it can be clearly seen that MultiChain is succesful in creating a chain of 10 blocks.

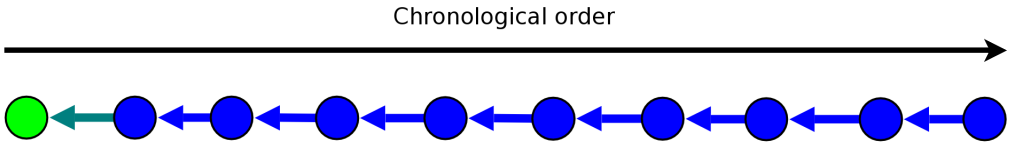


Figure 5.4: MultiChain chain graph of a single download of 10 MB.

The download amount stored in each block for the downloader is plotted in Figure 5.5. These datapoints are connected by a dotted line representing the link between these blocks. These plots show that MultiChain correctly tracks the download of 10 MB. The slope of the figure corresponds with the speed of the download. The upload amount of every seeder is identical to the amount of download of the downloader.

We also experimented with MutliChain running much longer with a bigger download. Our next synthetic experiment we simulate a download of 10 GB with a speed of 1000 KB/s. The rest of the setup is the same as in the previous example and the result is plotted in the same way in Figure 5.6. The individual points are obscured

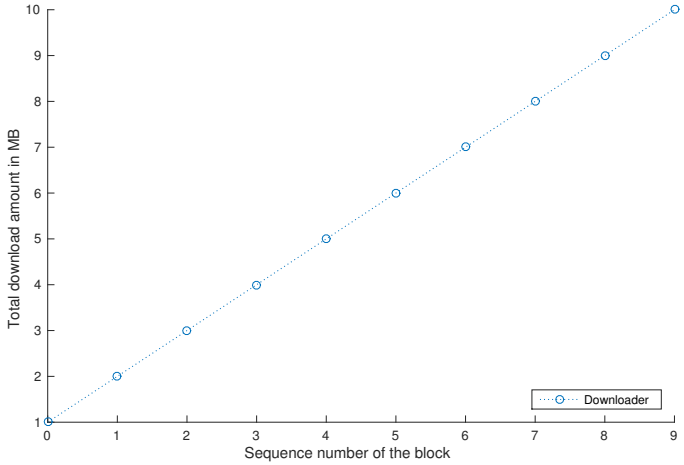


Figure 5.5: Total download amount when creating a chain of 10 blocks for a 10MB download..

by the large amount of points in the figure. The graph of the MultiChain would look identical to Figure 5.4, except it would be 10000 blocks long. It is not pictured as it is impractical to picture a chain of this length. As can be seen in the plots, MultiChain has no problem running for a longer period of time.

5.2.3 Tracking downloads with different speeds

In this synthetic experiment we measure if MultiChain can correctly track the upload and download amounts between two peers with different speeds. In this scenario a file of 100 MB is downloaded repeatedly at different speeds, respectively 500 KB/s, 750 KB/s, 1000 KB/s, 1250 KB/s, 2000 KB/s, and 3000 KB/s. The maximum speed of anonymous download was measured in experiments to be 1150 KB/s [9]. The upload and download of the file is done by different pairs of seeders and downloaders.

The total download amount of every downloader is plotted in Figure 5.7. These amounts are plotted in the same way as the previous experiment. The upload amount of every seeder is identical to the amount of download of the corresponding downloader. The plots show that MultiChain is able to correctly track the download and upload amounts without a problem. The plots are smooth and the amounts go up in fixed increments corresponding to the different speeds. Downloads at a a faster speed result in higher download amounts in the blocks. This means that MultiChain is fast enough to correctly track the amounts.

The download speeds below the threshold of 1000 MB of the scheduler are not distinguishable from the download at the threshold speed. There are three overlapping lines. We call this the scheduler effect. This is because the scheduler waits until the threshold is reached before initiating the block. The amount is tracked in

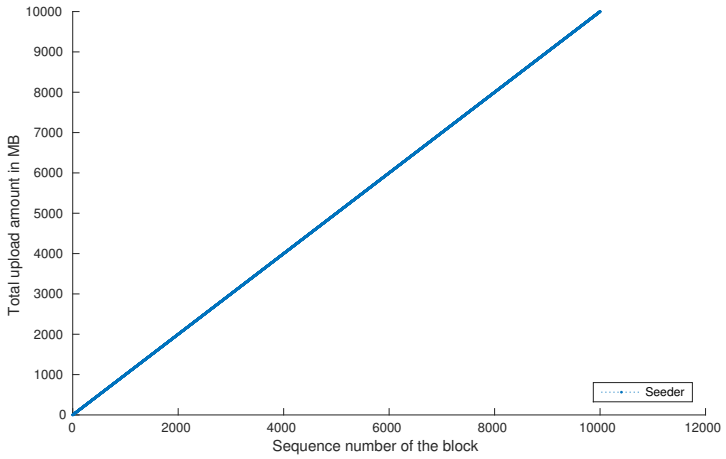


Figure 5.6: Total download amount when creating a chain of 10 000 blocks for a 10GB download.

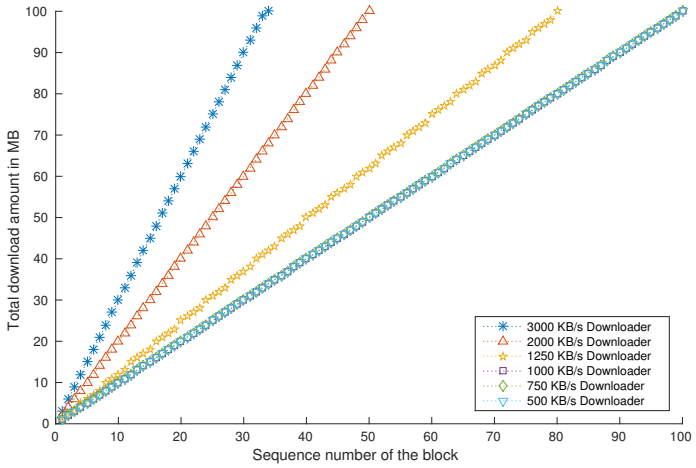


Figure 5.7: Total download amounts repeated 6 times with different speeds.

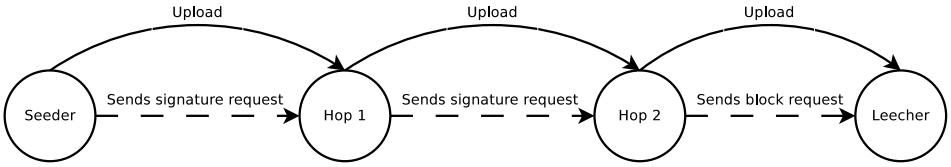


Figure 5.8: Block creation in an anonymous download.

the same amount of blocks, but the total time of the experiment is longer for these experiments. If the speed goes above the threshold, then this is reflected in the figure. The blocks tracks amounts in whole units of MB, this causes the download at 750 KB/s to not grow faster than the download at 500 KB/s. Any leftover KBs are saved for the next block.

This experiment was run several times before the final version in this report was run. Earlier versions of the experiment resulted in two bugfix and two improvements: the ability of the scheduler to create a block at the end of a download.

5.3 Tracking anonymous data transfer

In this section we will experiment with MultiChain tracking anonymous data transfer. Anonymous data transfer is a more complicated environment, where multiple peers work together to transfer data from the seeder to the downloader. We experiment in a synthetic environment and in environment using real anonymous downloads between peers.

5.3.1 Synthetic anonymous download

In an anonymous download scenario the data is downloaded through multiple hops. A seeder uploads data to the first hop. This hop relays the data to the second hop. The second hop sends the data to its destination at the downloader. This can be seen as sequence of peers and is illustrated in Figure 5.8. More hops can be added to better safegaurd the anonymity of the download.

The total download and upload amount is plotted, in the same way as the previous experiment, in Figure 5.9. The slopes of the figures are not representative for the upload and download speeds of the peers. This is because the x-axis represents the sequence-number of the block and not time. The hops create blocks that can be categorized in two types: a download validating block and an upload validating block. The download validating block only contains information about how much the hop has downloaded and is initiated by the peer in front of the peer in the sequence. An upload validating block is initiated by the peer itself with the peer next in the sequence. The seeder only has upload validating blocks and as such has half the amount of blocks. The downloader has viceversa only download validating blocks. The slope of his figure is much steeper as a result.

In the plot a discontinuity can be found at 92% of the download in the figures of the seeder and the first hop. This is the result of the first hop sending a signature request to the second hop. The second hop was not able to process this request, because it was already working on creating another block. The second hop drop this request. The first hop will still wait on the second hop to process its request until it will timeout. In turn, the seeder sent a request to the first hop that will timeout, because the first hop is not able to process this request aswell. During the timeouts of the seeder and the first hop, the second hop continues to validates its own upload amounts. No blocks are created that validate his download amount, so the slope becomes steeper during that time and the download amount remains level. The timeouted peers create no blocks. When the timeouts expires, the system returns to function as normal. In section 5.4 we further experiment with the timeouts in the system.

The seeder and downloader both only interact with one hop. These hops furthermore only interact with each other. This can be clearly seen in a part of the graph magnified in Figure 5.10(b). The middle nodes represent the interaction between the hops. The outer nodes are interactions between the seeder and the first hop and between the second hop and downloader. The blocks are created alternating resulting in the graph pictured.

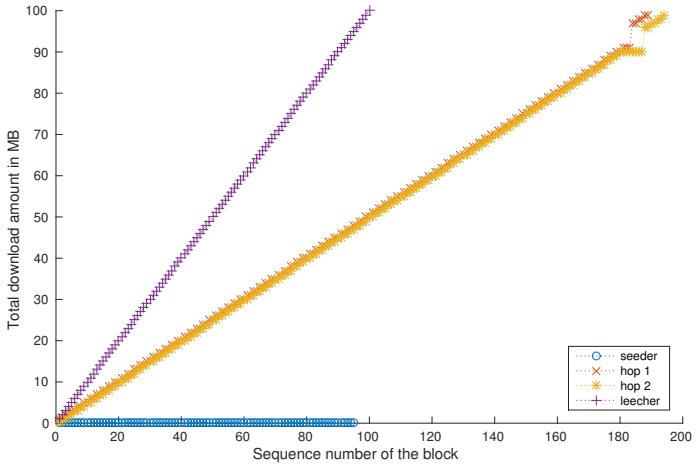
In the graph the timeout period can be seen clearly in Figure 5.11. The two half-signed block can be seen in red. The block with a reference coming from the outer block is the half-signed block belonging to the seeder. The strain of blue nodes are the blocks created between the second hop and the downloader. The red inner block is referenced by the first block created between the first hop and second hop after the timeout.

5.3.2 Integrated anonymous download

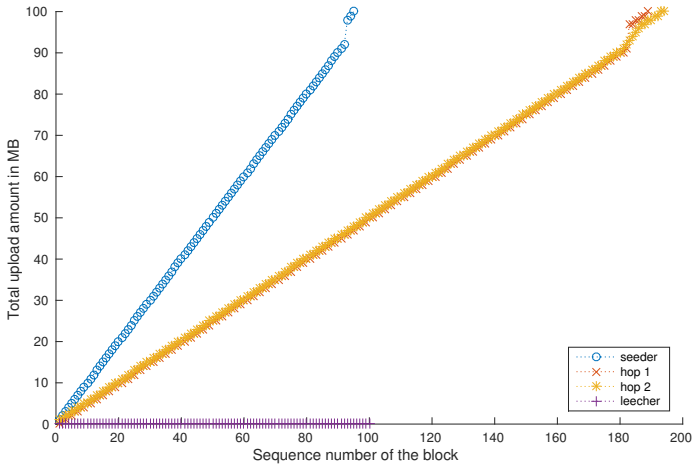
An substantial effort was made to integrated the tunnel community and hidden services community with the MultiChain community. These communities work together to provide functionality to transfer data anonymously. The tunnel community reports data transferred between peers and the MultiChain community add these amounts to the chain. They are already integrated with BarterCast. The method of integration of BarterCast was used to try to integrated MultiChain aswell. Setting up the Gumby environment to run all communities took also considerable time.

Experiments were conducted where an 100 MB file is transferred by the actual anonymity communities, while MultiChain transcribes the transfer amounts in the MultiChain. The first experiment we ran was to test the integration without any hops. In this setup the integration does not report any data transferred between the peers. As such the scheduler never sees a reason to schedule a signature request. The whole MultiChain community is never active. MultiChain should also track these downloads, but this requires MultiChain to be integrated in different places.

The second experiment was conducted with 2 hops. The result of the experiment

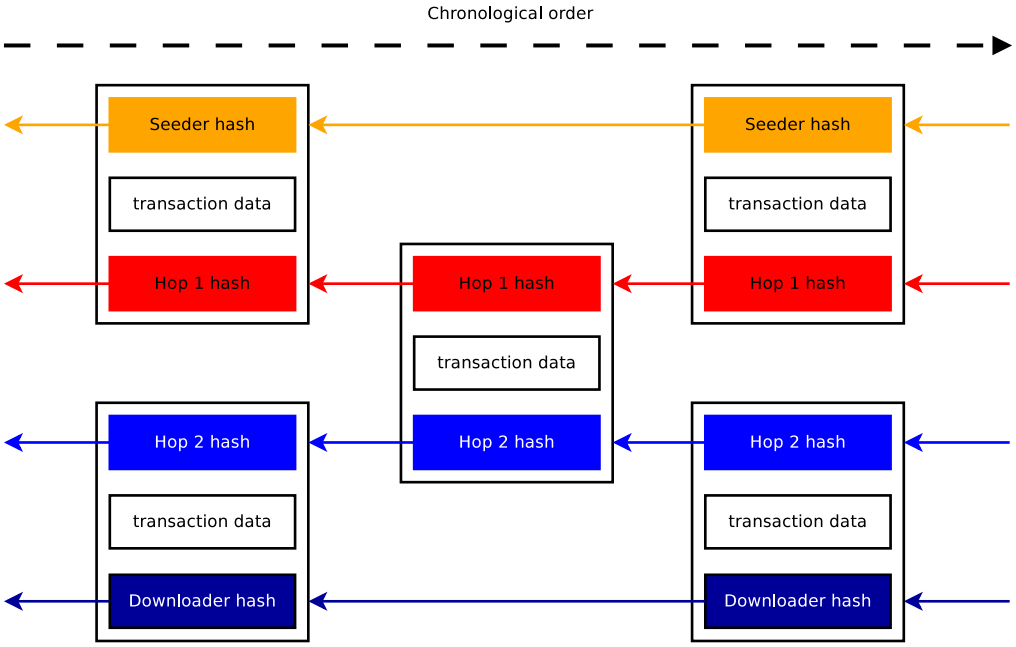


(a) Total download amount.

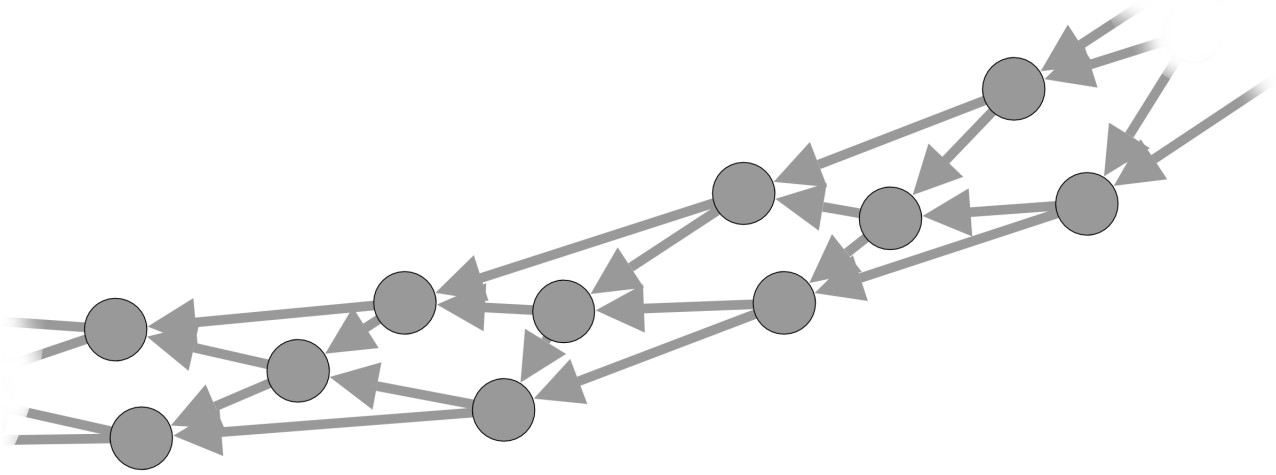


(b) Total upload amount.

Figure 5.9: Download and upload amounts during the anonymous download experiment.



(a) Partial example of expected MultiChain graph.



(b) Zoom of the actual intertwining in the MultiChain graph made with Gephi.

Figure 5.10: Intertwining of the seeder and hop 1, hop1 and hop 2, and hop 2 and the downloader.

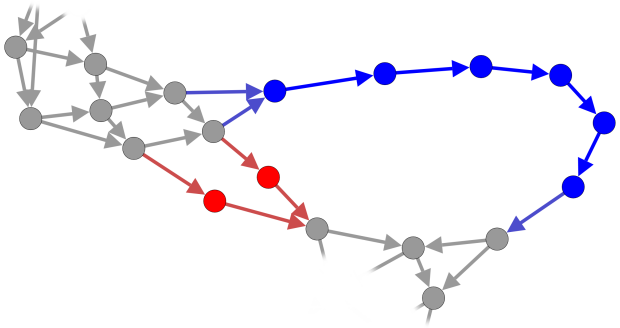


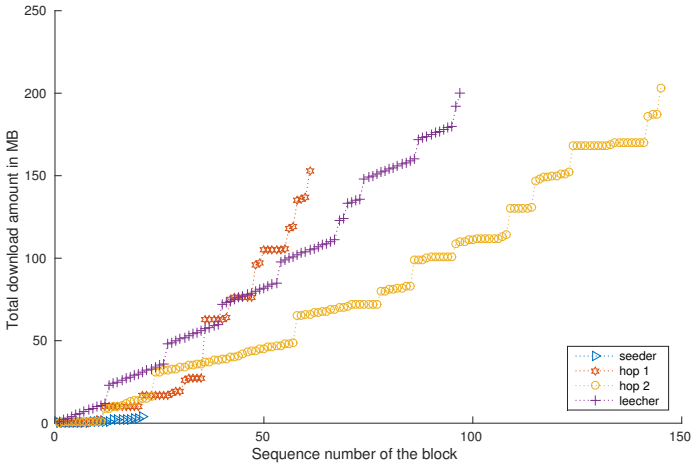
Figure 5.11: Zoom of the timeout of the seeder and hop 1, while hop 2 and the downloader continues.

can be seen in Figure 5.12. This shows that MultiChain does track the amount of upload and download for all peers in the network. The hops roughly download and upload the same amount of their data. This is what we expected as they just relay the data. The seeder and downloader respectively only uploads and downloads.

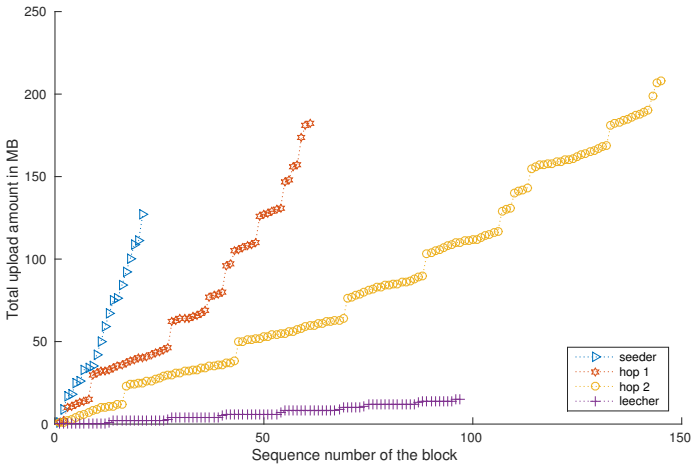
Drop events can be seen in the plots where the plot discontinues and does not grow steadily. A drop event is when a block request cannot be processed because a node is busy. The requesting node will wait until a timeout occurs. The reason these timeouts occur frequently is that thresholds of the scheduler of every node is now reached at the same time resulting in a timeout. Our recommendation is to have these thresholds to be shifted by a small amount randomly for every peer to prevent them from overlapping. We performed some very preliminary experiments that shows that our recommendation reduces the amount of timeouts.

The amount of data that is reported to be transferred is remarkable. The overhead measures twice the actual download size. We therefore the conclusion that the integration is not properly and too much data is reported. It is unclear where this data originates from.

Integrating MultiChain into the anonymous download communities is difficult, because those communities do not use the standard classes used by Tribler to send messages. These are necessary to properly integrate MultiChain and a complicated conversion was necessary. Our recommendation is to refactor the anonymous download communities to use the standard classes and to make it more clear in the code where data is sent and where data is received. Also comments should be added to clarify the code. Currently, there are no comments whatsoever. This effort should atleast clarify why so much data is reported and will probaly fix the problem.



(a) Total download amount.



(b) Total upload amount.

Figure 5.12: Download and upload amounts during the integrated anonymous download experiment.

5.4 Drop event recovery

MultiChain can run during normal operation into a situation where a multiple of peers request to create blocks from each other as explained in section 4.3. A specific experiment was conducted that created the situation manually. This situation was also encountered during experimentation and the experiment shows MultiChain correctly recovering from this situation.

5.4.1 Forced drop event

In this experiment a MultiChain node 1 tries to send a request to node 2 to create a block. Node 2 is specifically configured for the purpose of the experiment to ignore and drop all requests. Node 1 now should wait for the response of the other node for a specific time. During the time that node 1 is waiting a different request is sent from a normal node 3 to node 1. This request will be dropped by node 1, as it cannot process any incoming requests while node 1 has an outstanding request. Node 1 and node 2 should timeout and continue operations. After that node 2 will retransmit a request to node 1 and this should be completed correctly.

The experiment is locally run using gumby with all nodes running on a single computer. Only three instances of MultiChain communities are started. One of these instances never respond to a request to construct a block. The logging of the every node is captured and recorded to verify the results of the experiment.

The output of the logging can be seen in Figure 5.13. First node 1 sends a signature request to node 2. This message is ignored by node 2. During the waiting period of node 1 a block request is sent to node 1 by node 3. This request is also ignored, because node 1 cannot perform operations on the chain. After the timeout period both node 1 and node 2 save an half-signed block to their chain and can continue operations. This is validated by a block created between node 3 and node 1.

5.4.2 Naturally occuring drop events

In the experiment a 100 megabyte file was downloaded anonymously with 2 hops. Anonymously downloading is described more in the thesis report of R. Ruigrok [9]. There are 2 Triblers instances with exit functionality and 18 instances without exit functionality. All instances are run locally on one machine. The instances are run in parallel, so the fact that all instances are run on a single machine does not cause the drop event to occur. There is no packetloss in this experiment. The corresponding graph of all the MultiChains of the experiment can be seen in Figure 5.14.

The potential scenario of two peers both waiting can be seen multiple times in the graph and are encircled. The scenario generates a half-signed block at both peers. Usually the peers continue collaboration and this continuation of a sequence can be seen in subsequent blocks. In the graph a more complicated scenario can

```
1: Requesting Signature for candidate: 2
1: Chain Exclusion: signature request: False
1: Chain Exclusion: acquired, sending signature request.
1: Sending signature request.
2: Received signature request that will be ignored.

3: Requesting Signature for candidate: 1
3: Chain Exclusion: signature request: False
3: Chain Exclusion: acquired, sending signature request.
3: Sending signature request.
1: Received signature request.
1: Chain Exclusion: process request: True
1: Chain Exclusion: not acquired. Dropping request.

1: Timeout received for signature request.
1: Persisting sr: bFOXhHT2ffSrtIn9tuMfEGGarGY=
3: Timeout received for signature request.
3: Persisting sr: l1O8UquXdxWdkg+KAYZ1FYocwjo=
3: Requesting Signature for candidate: 1

3: Chain Exclusion: signature request: False
3: Chain Exclusion: acquired, sending signature request.
3: Sending signature request.
1: Received signature request.
1: Chain Exclusion: process request: False
1: Chain Exclusion: acquired to process request.
1: Persisting sr: 7Y86Ck4duwTduP6j/aIRrWHAZqw=
1: Sending signature response.
3: Signature response received. Modified: True
3: Valid 1 signature response(s) received.
3: Persisting sr: 7Y86Ck4duwTduP6j/aIRrWHAZqw=
3: Chain exclusion: released received signature response.
```

Figure 5.13: Output of the manual drop event experiment

also be seen where multiple peers timeout between each other. The graph shows that MultiChain correctly recovers from all scenario's.

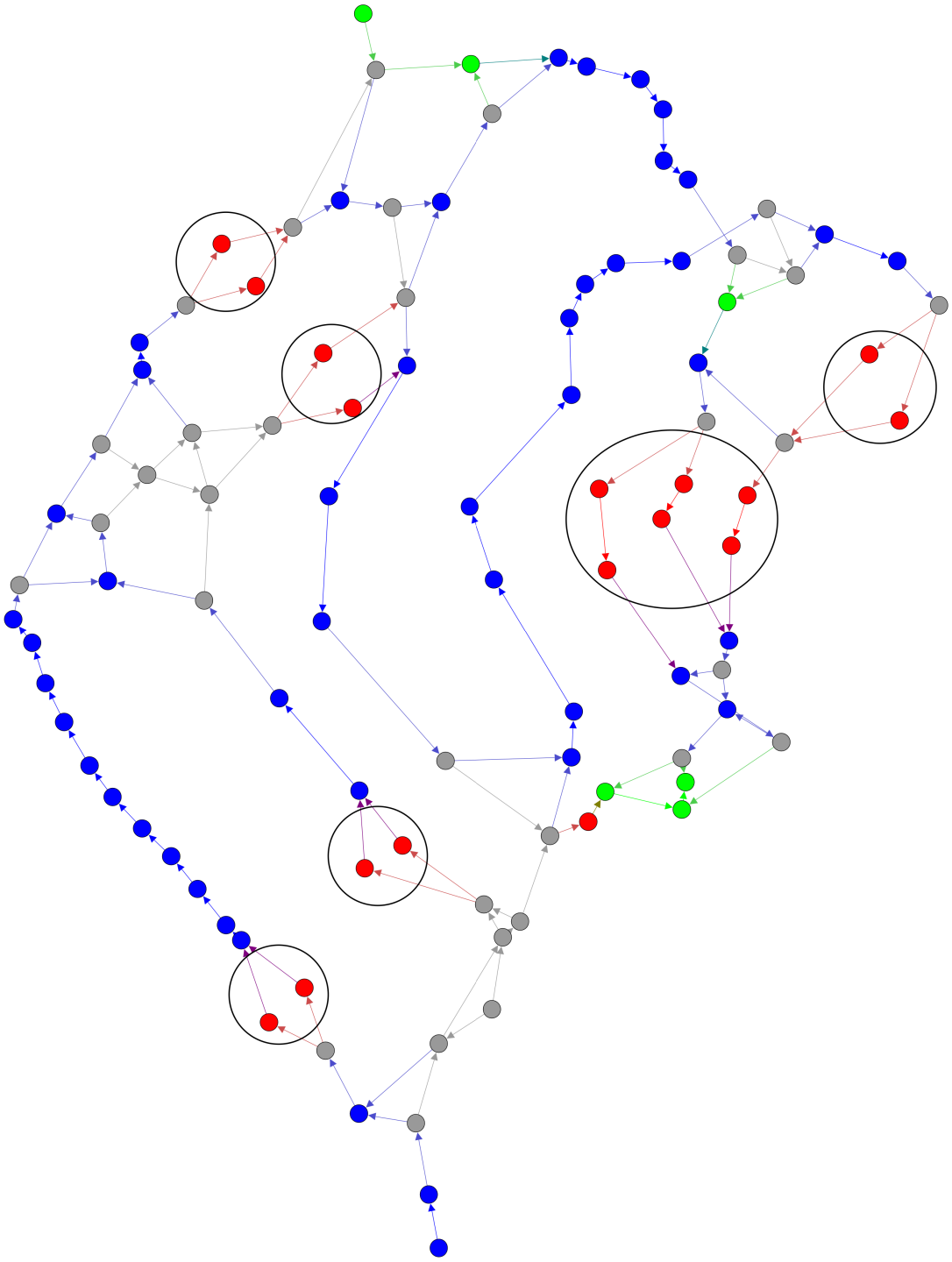


Figure 5.14: Mixing of double-signed blocks(blue/grey) and half-signed blocks(red) in MultiChain.

Chapter 6

Known vulnerabilities

There are several known security vulnerabilities with the current design. As said in section 2.5, an incremental approach was taken and a lot of work is left. As such, the design is part of a bigger system that as a whole is to be developed, before it can be deployed in the real world. The current implementation is not yet ready to fully replace BarterCast as a reputation system. It is too vulnerable for malicious nodes to attack the system. In this section we will describe the known vulnerabilities and explain the future work that is needed to limit these vulnerabilities.

6.1 Branch attack

In this section we will explain an attack that can be done by a malicious node M. The attack consists of obscuring a part of his transaction history. M creates a new branch of his transaction history that is more favourable to him. This is double spending of reputation.

6.1.1 Alternating partial transaction history

A malicious node M has his own chain of transactions and he wants to falsify his transactions after a certain point. He wants to rewrite his transaction history from that point and create an alternate transaction history. M wants to do this to whitewash his reputation. For example he could have downloaded large amounts without contributing himself with uploading data. The node can simply choose to forget and obscure blocks after that point. A new branch will be created by chaining new blocks to the desired point in history.

In Figure 6.1 an example can be seen of a branch created by M. In this example M tries to obscure block 2 and any subsequent blocks from C. When C requests the transaction history of M, M will only send the transaction history up to block 1. When M and C create a block together, M will reuse the hash of block 1 in the new block. For clarity of the diagram, the node interacting with M in block 1 is not displayed.

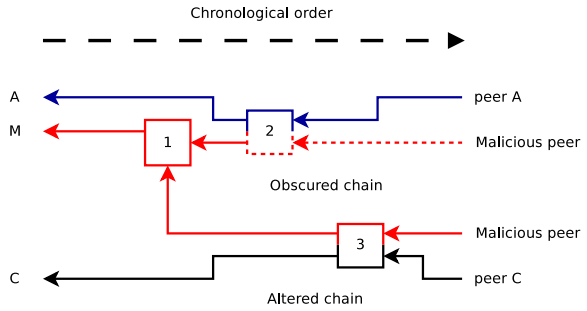


Figure 6.1: Example of a branch created by M.

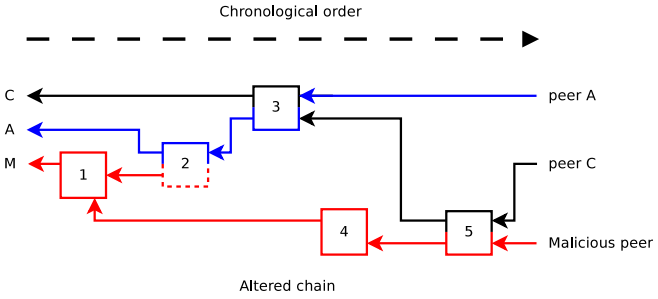


Figure 6.2: Detectable fraud by C.

Now malicious node M does have the problem that not only he knows his transaction history. When M interacted with node A a block was created that M tries to obscure. A has this block in his own chain and therefore knows about it being part of the transaction history of M. This can be seen in the example in Figure 6.1.

M will want to reduce the likelyhood of the detection of his fraud. If his fraud is detected, he might be punished and no longer to continue his abuse. The first way to minimize detection is to choose to only interact with new nodes that do not know about the alternate part of the transaction history. Nodes that have requested an alternate part of the transaction history or that have been interacted directly with are no longer interacted with. In a sufficiently healthy network this will result in node M being able to find new nodes to help him.

There is another example that will expose the fraud of M. This example can be seen in Figure 6.2. A node C might still exposes the cheating of M by chance. C can have an interaction with node A by coincidence. Before creating block 3 C will request the transaction history of node A containing an obscured block of M. Now when M wants to interact with C, M will want to create block 5. When C requests the full transaction history of M, it will detect that the transaction history of M no longer contains block 2. This exposes the fraud of M.

But C will have no sure way of exposing this type of fraud by his own doing, except for requesting every transaction history of every node in the system. This

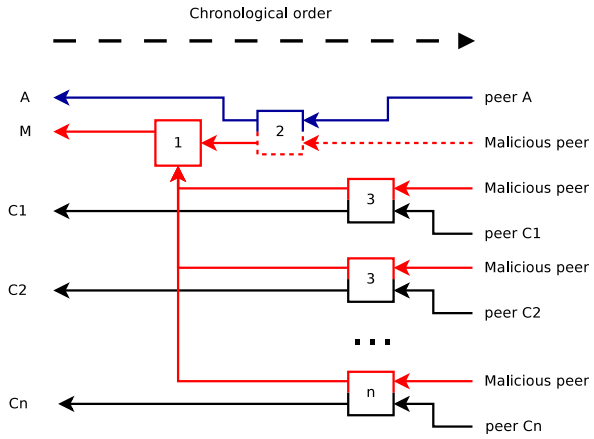


Figure 6.3: Example of multiple branches created by M.

is in a way a common, full transaction history and was chosen to be avoided by the design to become more scalable. C can limited the possibility of the attack by increasing his knowledge by collecting more transaction history of other nodes. If node A or B stop participating and exit the network, then C will have no way of detecting the fraud by M.

The second way M can limit the exposure of his cheating is in a more sophisticated way. He can present several, different transaction history to different nodes. M will continue keeping track of the unmodified transaction history. When M wants to interact with A or B, both knowing this transaction history, he will present this transaction history. So M can still interact with A and B. But when interacting with C he will present his alternate transaction history. C will only expose the fraud in the same way as previously.

This attack can always be done M and is not limited to circumstances. Also the attack is not limited and M can try to fool any number of other nodes C. As shown in Figure 6.3.

The likelyhood of exposing this attack depends on several factors and will be the only factor to limit M in performing this fraud. The likelihood depends on:

- Size of the network
- Likelihood of interactions between A or B and C

These properties will influence the chance of C coming across an obscured block.

6.1.2 Punishment of this attack

When the fraud is detected by C, then currently only C can punish M by no longer interacting with him. There is currently no way of making it globally known to

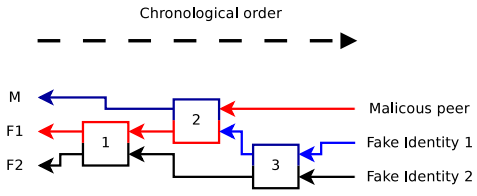


Figure 6.4: The sybil attack by M.

every node in the network that fraud was committed. So only M is punished by C and can continue his abuse of other nodes in the network.

A proposal for the future, that is already currently worked on, is to construct an additional network where the discovery of the fraud can be announced. Proof of the fraud is easily to distribute and cannot be repudiated by the malicious node. The proof of the fraud is two blocks containing the same previous hash. The proof cannot be repudiated, because it contains two signatures by the malicious node. See section 4.1.4 for more explanation. To increase the likelyhood of detection every node will walk the network to look for fraud.

6.2 The Sybil attack

In this section we will explain the Sybil attack [45] and how it can be used in the MultiChain system to create an artificial reputation. A universallly applicable solution has not yet be found [46].

6.2.1 Using fake indentities

In large distributed systems convincingly distinct identities can be presented that are infact all under the control of a single adversary M. Identities can be public keys like in the Dispersy system or other abstractions of information without direct physical knowledge. These identities can participate in the system acting like normal agents, but they can also help each other malificiently to boost each other towards a common goal.

In the MultiChain system the attack is done in the following way. M creates several other identities besides his own. This involves generating several key pairs. Now M controls several identities that can be presented convincingly. The key pairs do not have to be linked to a node responding to requests. Failure of nodes are typical in large distributed systems.

Using these key pairs M can now generate an artificial reputation. Transactions between M and the fake identities are generated by M. These are signed using the keys of the fake identities. Using a single transaction it cannot be determined if it is between two distinct identities or a single distinct identity and a fake identity.

An example of the Sybil Attack can be seen in 6.4. In this example M is the malificient node and F1 and F2 his fake identities. Block 1 and block 2 contain

a transaction that is favourable to M, but M has done nothing to deserve these. Similair blocks like block 3 can be generated between fake identities in an effort to thwart efforts to analyse the network and detect fake identities.

In this way M is able to boost his reputation without much work. The more sophisticated the generation is done by M, the harder it will be to detect that M boosted his reputation using fake identities. M can then abuse his fake reputation by solliciting cooperation from other nodes in the network. They will respond positively on the request by M based upon the false reputation M claims to have.

6.2.2 Validating Identities

A node can have three potential sources of validation of distinct identities:

- Itself
- Other entitites
- A trusted, central authority

A node itself could try to directly validate two identities to be distinct. It would challenge several identities to complete a task that only two entities could complete. The task would require more resources than a single entity possesses and will be issued simultaneously to all identities. If the identities complete the task, they have proven to be distinct. Example of required resources are communication, storage and computation. These challenges can scale to validate more entities at once.

Indirect validation can be used by a node by delegating validation to other entities. A node could accept additional identities to be distinct when an accepted identity vouches for it. The node has the delegated responsibility to challenge the entity with a challenge. This would limit the total amount of challenges needed. But an obvious pitfall of delegating this to other identities is that these can vouch for fraudelent identities. Next to this, it involves an increase in complexity as the challenges still have to be issued concurrently.

But these challenges are highly indesirable. These challenges require by definition to occupy between two nodes a limited resource to the maximum capacity of a single node. While not providing any additional functionality beyond asserting distinct identities. These challenges also prove inworkable when entities have hugely different amount of resources available. No challenge can be constructed that can be worked by less powerful devices, that could not be worked several times by more powerful devices.

Next to this, these challenges are only usable with nodes that are active and responding to challenges. If a node becomes inactive it cannot be ascertained if the node is a fake identity. But this identity still makes claim about a reputation of another entity. A choice has to be made between either not counting these claims and allow for a drop in reputation of a node or allow claims that cannot be validated to be taken into account. Both options lower the usability of the system.

A trusted, central authority would be able to vouch for distinct indentities if it has an other way outside the system of asserting that it indeed is a distinct identity. Approaches exists that implicitly rely on the authority of a trusted agency. But a central authority goes against the principles of a peer-to-peer network.

Several other defenses against the Sybil Attack have been proposed [47, 48] or variants on the defenses stated here [46]. But these either have the same limitation and are minor improvements or are non-applicable for MultiChain.

For M to conduct the Sybil Attack he will need to generate a set of key pairs. This is trivial to do and Dispersy provides functionality to do this very easily. With these keys M has to generate blocks.

Sybil Attack can have a wide range of sophistication. Ranging from a large amount of fake identities with a large amount of transaction between them to a single, fake identity boosting the reputation of M in a couple of transaction. Even the simple Sybil Attack are hard to defend against if these are not done too obviously. But Tribler will have to limit this attack in the future, but this will be a difficult challenge.

6.3 Denial of service attack

The denial of service attack is a common attack seen on the internet that disables a service by flooding it with requests of service. This is also a potential attack on MultiChain particulairly because of a bottleneck within MultiChain.

6.3.1 Mutual exclusive code

A node can only perform a single operation at a time on its chain. If blocks would be created in parallel, they would point to the same previous block. This can be seen as a double spending attack and should be punished as described in section 6.1. So safeguards have been implemented to ensure that this never occurs. Because only a single operation can be handled at once makes a node very vulnerable to a denial of service attack. A bottleneck is hereby introduced through design that cannot be scaled.

If a node is flooded by sufficient bogus requests to create a new block, then it will become overburdend with these requests to service real requests or to send send out its own requests. The node is denied service and cannot create meaningful blocks. Because the node becomes unresponsive to block creation requests other nodes will not trust him to sign future blocks and will not be granted upload bandwidth. The other node does not trust that in return for his collaboration he will receive a boost in his reputation and will stop collaboration. The node under attack will also be unable to transform his own collaboration into a boost in reputation because he cannot send his own block creation requests.

The proposed denial of service attack is more sophisticated than typical denial of service attacks. These typicaly involve simply flooding a server with requests,

but for the proposed attack the requests have to be crafted with care. They need to valid to be serviced by the attacked node and reach the mutual exclusive code segment that is the vulnerable bottleneck. A request has to be a counterpart of a real interaction or the request can be easily filtered. Any request that is not carefully constructed in this way will still impose a computational and network burden on the node, but this is always a risk and not a specific vulnerability of MultiChain.

6.3.2 Filtering requests

Detection can be implemented that will help in detecting fake requests. The detection can be run in parallel and does not have to enter the mutual exclusive part. Any request that is fake will not reach the mutual exclusive part and will not drown out the service of valid block creation. Effective analyses have to be researched and implemented in future work to harden the system against this attack. Also priority can be given to blocks initiated by the node itself. This will solve the problem of the node not being able to construct blocks he wants to create.

Chapter 7

Conclusion

A reputation system is a necessity in a collaborative network like peer-to-peer file-sharing. But the creation of a tamper-proof interaction history is a difficult undertaking, as seen as by several attempts in related work with varying success. This first step, in an incremental approach to creating a new tamper-proof interaction history, is further testimony to that statement. MultiChain is a proof-of-concept that takes a different approach with multiple chains instead of a single chain can be succesful in creating a reputation system

A scalable system has been introduced that can track interactions. The system does not rely on any central component or a central data structure. This system has been integrated with Tribler, but this requires more attention. MultiChain can be released as a first version to start measuring the system in the real world. Improvements can be implemented using these measurements.

MultiChain has been tested in software engineering tests and experiments. These tests prove the system to be correctly working as designed. The experiments show that MultiChain to behave as expected in a real world scenario. MultiChain shows promise in tracking the interactions in a scalable way.

The current implementation is not yet ready to fully replace BarterCast as a reputation system. It is too vulnerable for malicious nodes to attack the system. The possibility and impact of these attacks have to be reduced before MultiChain can be fully used as a reputation system with a reasonable amount of trust. Proposals are already made and worked on to harden the system in the future.

Bibliography

[1] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. Van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 2, pp. 127–138, 2008.

[2] A. Bakker, J.-D. Mol, J. Yang, L. dAcunto, J. A. Pouwelse, J. Wang, P. Garbacki, A. Iosup, *et al.*, "Tribler protocol specification," *URL: http://svn. tribler. org/bt2-design/protospec-unified*, 2009.

[3] B. Cohen, "The bittorrent protocol specification." `https://www.bittorrent.org/beps/bep_0003.html`, 2008.

[4] K. Lai, M. Feldman, I. Stoica, and J. Chuang, "Incentives for cooperation in peer-to-peer networks," in *Workshop on economics of peer-to-peer systems*, pp. 1243–1248, 2003.

[5] M. Meulpolder, L. D'Acunto, M. Capota, M. Wojciechowski, J. A. Pouwelse, D. H. Epema, and H. J. Sips, "Public and private bittorrent communities: a measurement study.," in *IPTPS*, p. 10, 2010.

[6] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, 2003.

[7] E. Adar and B. A. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, no. 10, 2000.

[8] R. S. Plak, *Anonymous Internet: Anonymizing peer-to-peer traffic using applied cryptography*. TU Delft, Delft University of Technology, 2014.

[9] R. J. Ruigrok, *BitTorrent file sharing using Tor-like hidden services*. TU Delft, Delft University of Technology, 2015.

[10] R. J. Tanaskoski, *Anonymous HD video streaming*. PhD thesis, TU Delft, Delft University of Technology, 2014.

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

[12] G. Hardin, "The tragedy of the commons," *science*, vol. 162, no. 3859, pp. 1243–1248, 1968.

[13] M. Nowak and K. Sigmund, "The evolution of stochastic strategies in the prisoner's dilemma," *Acta Applicandae Mathematicae*, vol. 20, no. 3, pp. 247–265, 1990.

[14] M. A. Nowak, "Five rules for the evolution of cooperation," *science*, vol. 314, no. 5805, pp. 1560–1563, 2006.

[15] P. D. Bó, "Cooperation under the shadow of the future: experimental evidence from infinitely repeated games," *American Economic Review*, pp. 1591–1604, 2005.

[16] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Consulted*, vol. 1, no. 2014, p. 28, 2008.

[17] Blockchain.info, "Block chain size." `https://blockchain.info/charts/blocks-size`, November 2014.

[18] Bitcoinwiki, "Difficulty." `https://en.bitcoin.it/wiki/Difficulty`, December 2015.

[19] Blockchain.info, "Hash rate." `https://blockchain.info/charts/hash-rate`, March 2015.

[20] Blockchain.info, "Difficulty." `https://blockchain.info/charts/Difficulty`, March 2014.

[21] Bitcoinwiki, "Scalability." `https://en.bitcoin.it/wiki/Scalability`, November 2015.

[22] Visa, "Stress test prepares visanet for the most wonderful time of the year." `http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html`, October 2013.

[23] J. Garzik, "Bitcoin improvement proposal 0102: Block size increase to 2mb," 2015.

[24] G. Andresen, "Bitcoin improvement proposal 0101: Increase maximum block size," 2015.

[25] J. A. Pouwelse, J. Yang, M. Meulpolder, D. H. Epema, and H. J. Sips, "Buddycast: an operational peer-to-peer epidemic protocol stack," in *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging* (G. Smit, D. Epema, and M. Lew, eds.), pp. 200–205, ASCI, 2008.

[26] M. Meulpolder, *Managing supply and demand of bandwidth in peer-to-peer communities*. TU Delft, Delft University of Technology, 2011.

[27] M. Meulpolder, J. A. Pouwelse, D. H. Epema, and H. J. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[28] V. Dumitrescu, *Rewarding Good Behavior in Peer-to-Peer Networks*. TU Delft, Delft University of Technology, 2013.

[29] G. Logiotatidis, *Splash: data synchronization in unmanaged, untrusted peer-to-peer networks*. TU Delft, Delft University of Technology, 2010.

[30] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[31] R. Dingledine, D. S. Wallach, *et al.*, "Building incentives into tor," in *Financial Cryptography and Data Security*, pp. 238–256, Springer, 2010.

[32] E. Androulaki, M. Raykova, S. Srivatsan, A. Stavrou, and S. M. Bellovin, "Par: Payment for anonymous routing," in *Privacy Enhancing Technologies*, pp. 219–236, Springer, 2008.

[33] R. Jansen, N. Hopper, and Y. Kim, "Recruiting new tor relays with braids," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 319–328, ACM, 2010.

[34] R. Jansen, A. Johnson, and P. Syverson, "Lira: Lightweight incentivized routing for anonymity," tech. rep., DTIC Document, 2013.

[35] R. Jansen, A. Miller, P. Syverson, and B. Ford, "From onions to shallots: Rewarding tor relays with tears," *HotPETS.(July 2014)*, 2014.

[36] M. Ghosh, M. Richardson, B. Ford, and R. Jansen, "A torpath to torcoin: Proof-of-bandwidth altcoins for compensating relays," in *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2014.

[37] Y. Chen, R. Sion, and B. Carbunar, "Xpay: Practical anonymous payments for tor routing and other networked services," in *Proceedings of the 8th ACM workshop on Privacy in the electronic society*, pp. 41–50, ACM, 2009.

[38] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[39] J. C. Van Der Lubbe, *Basic methods of cryptography*. Cambridge University Press, 1998.

[40] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," in *Advances in CryptologyEUROCRYPT'98*, pp. 591–606, Springer, 1998.

[41] A. Acquisti, R. Dingledine, and P. Syverson, "On the economics of anonymity," in *Financial Cryptography*, pp. 84–102, Springer, 2003.

[42] R. Dingledine, N. Mathewson, and P. Syverson, "Reputation in p2p anonymity systems," in *Workshop on economics of peer-to-peer systems*, vol. 92, 2003.

[43] J. Smart, *Jenkins: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[44] Tribler, "Jenkins build server." `http://jenkins.tribler.org/`, September 2015.

[45] J. R. Douceur, "The sybil attack," in *Peer-to-peer Systems*, pp. 251–260, Springer, 2002.

[46] B. N. Levine, C. Shields, and N. B. Margolin, "A survey of solutions to the sybil attack," *University of Massachusetts Amherst, Amherst, MA*, 2006.

[47] J. Newsome, E. Shi, D. Song, and A. Perrig, "The sybil attack in sensor networks: analysis & defenses," in *Proceedings of the 3rd international symposium on Information processing in sensor networks*, pp. 259–268, ACM, 2004.

[48] J. Dinger and H. Hartenstein, "Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration," in *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pp. 8–pp, IEEE, 2006.