Improve dissemination in the Ethereum Network

Luís Carlos Silva Aguiar

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos
Prof. João Pedro Faria Mendonça Barreto

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Alysson Bessani

November 2019

Dedicated to everyone that supported me during these years...

Acknowledgments

I want to thank everyone that helped me in this long journey. Mainly, I want to express my sincere gratitude to my friends, teachers, and specially to my family. Without them, I would not be able to accomplish this work.

First, I would like to thank my advisors, Professor João Barreto, and Professor Miguel Matos, for all the help, guidance, patience, and knowledge that helped me to finish this work. Furthermore, I would like to thank Paulo Silva that enabled me to understand better the Geth code.

Second, to the Instituto Superior Técnico for all the knowledge that I learned during these years. I am sure that experience will be a valuable asset in my professional work.

Third to my friends, that supported me during these five long years of learning. Without them, this would never be possible.

Last but not least, to my family, who was always there for me in the bad and good moments.

This work is partially funded by Fundo Europeu de Desenvolvimento Regional (FEDER) via Programa Operacional Regional de Lisboa and by Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia though project Lisboa-01-0145-FEDER-031456 (Angainor) and UID/CEC/50021/2019. Without them, the paper that I wrote would never have gone to the Inforum conference.

Resumo

As blockchains e as criptomoedas estão a ganhar cada vez mais mercado no mundo tecnológico. Devido ao seu potencial, novas criptomoedas foram criadas como o caso do Ethereum introduzindo novos casos de uso no mundo tecnológico. O Ethereum permite o desenvolvimento de aplicações descentralizadas na sua rede. Mas apesar de esta tecnologia ter um enorme potencial, existe algumas lacunas que precisam de ser combatidas. No caso particular da disseminação, o Ethereum apresenta um grande número de duplicados na disseminação de blocos e de transações que afetam o processamento de cada nó e congestionam a rede com informação redundante. Para além disso, é possível tornar o algoritmo de disseminação mais rápido permitindo diminuir o trabalho desperdiçado na rede. Neste artigo apresentamos várias soluções para mitigar estes problemas para os blocos. Onde dependendo do objetivo podem ser aplicadas diferentes abordagens. Também desenvolvemos uma abordagem para as transações, onde o número de transações duplicadas é bastante reduzido, mas em contrapartida o tempo que cada transação demora a chegar a todos os nós é um bocado maior.

Palavras-chave: Blockchains, Ethereum, Disseminação, Duplicados, Latência

Abstract

Blockchains and cryptocurrencies are increasingly gaining market share in the technological world. Due to their potential, new cryptocurrencies were created like the Ethereum, introducing new use cases in the technological world. Ethereum enables the development of decentralized applications in the network. But while this technology has huge potential, there are some gaps that need to be addressed. In the dissemination case, Ethereum has a large number of duplicates in block and transaction dissemination that affect the processing of each node and congest the network with redundant information. In addition, it is possible to make the dissemination algorithm faster where the wasted work in the network is reduced. In this article, we present several solutions to mitigate these problems for blocks. Depending on the goal, different approaches can be applied. We have also developed an approach to transactions where the number of duplicate transactions is greatly reduced, but consequently the time that each transaction takes to reach all nodes is slightly longer.

Keywords: Blockchains, Ethereum, Dissemination, Duplicates, Latency

Contents

List of Figures

Chapter 1

Introduction

Blockchains are hugely important nowadays. They are used to store data. Cryptocurrencies use them as a public ledger to store transactional information. Also, banks, enterprises, startups are building their product using blockchain solutions [WGP$^+$17]. Leading tech giants realized the importance of blockchains, and they provide services where they create, run, and manage a blockchain, and in return, the enterprises have to pay a rent; the enterprises only have to focus on its core business activities. This is called Blockchain as a Service (BaaS). [1] [2]

1.1 Motivation

The blockchain is a growing list of blocks that contain data where each block has the hash of the previous one. What makes it appealing is its difficulty in modifying the information: since each block has the hash of the previous one, if a block is modified, the following blocks will also have to be changed to have a consistent record of the information; as it takes some work for a block to be valid, then its very expensive modifying the data. As the information is stored across multiple nodes, it does not have a centralized point of vulnerability, making it more difficult for the attackers to exploit the system. Likewise, it does not exist any central point of failure [CV17].

But blockchain also has some limitations like:

- `Absence of Privacy` because the information must be public if any device can join the system and participate in the protocol. Bitcoin does not offer a strong privacy guarantee because when analyzing the blockchain, it is easy to link different accounts to the same person. However, in the literature, there are some methods to make payments with strong privacy guarantees that Bitcoin and other blockchains can implement [SCG$^+$14].

[1] https://aws.amazon.com/pt/partners/spotlights/blockchain-partner-spotlight/
[2] https://azure.microsoft.com/en-us/solutions/blockchain/

- `Poor performance` because all nodes must reach consensus. In Bitcoin, "(Nodes) vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them." [Nak08]. They must resolve a cryptographic puzzle using their CPU for a block to be valid, and this process takes a long time to make. Also must of the work done is wasted because different nodes are trying to create concurrent blocks that eventually will be discarded by the blockchain.

- `Not environment friendly` because as said before, for a block to be valid, the nodes must solve a cryptographic puzzle using their CPU. And this solving process wastes a lot of energy. Since the majority of electricity comes from fossil energies, if the technology becomes worldwide used, then the effects on the environment could be catastrophic at a time when we are doing everything we can to reduce carbon emissions [OM14].

- `Inefficient dissemination` because the information must reach all nodes faster, to have a faster consensus in the network. Also, a lousy dissemination algorithm can cause many duplicates in the system that affects the bandwidth, and the CPU used by the nodes to process these duplicates.

In this thesis we will focus on the dissemination in the blockchains. If a naive dissemination algorithm is applied, then the number of duplicates affecting the system will be large, thus affecting the processing power of each node, slowing down the network. Also, if the dissemination algorithm is slow, the amount of wasted work will be larger because the time that each node wastes in concurrent blocks will be longer. We will explain better this case in the next section.

1.2 Topic Overview

Satoshi Nakamoto invented the blockchain to solve the double-spending problem in a peer to peer network. The blockchain acts as a public ledger where all transactional history is stored on a sequence of blocks. For a block to be validated some work must be done, so to change its content it is necessary to redo all the work. This ledger is shared among all the peers that validate all the information inside it [Nak08]. The ledger contains transactions. These transactions are collected on a block, and this block is inserted into the blockchain [DW13]. It is necessary that all peers agree on these sequence of blocks to have data consistency. This implies that all nodes agree on which transactions are done and in which order. This task of agreement between nodes is called consensus, and the consensus protocol used on Bitcoin is called Nakamoto Consensus [NG16].

The biggest problem with this type of systems is the poor performance where only a few transactions are done per second. Visa can handle 24,000 transactions per second (tps), while Bitcoin can handle only 7 tps and Ethereum can handle 15 tps [3] [SZ15, BMZ18, Nak08]. Recently, CryptoKitties, a game on top of Ethereum blockchain disrupted the Ethereum Network. This happened because the Ethereum network could not handle all the transactions made by this game, calling into question the ability of Ethereum to handle an unlimited amount of decentralized applications [Kha].

One of the reasons for the poor performance is because it is challenging to reach consensus and most of the work done is wasted [NG16]. For instance, in a certain period, it is possible to have two chains as 'truth', ending up with different nodes working in different chains. After a while, when a chain has more work than the other, the chain with less work will be discarded, and the work on it will be lost, making the write availability low [WGP+17].

Dissemination has a fundamental role in this disagreement. If the propagation time is very high and it is easy to generate a block, it is more likely that a node *n1* would mine a block *B'* when another node *n2* has already mined a concurrent block *B*, but *n1* has not yet received *B*. So if *n1* had received *B* before it had completed the mining of *B'*, it would have given up mining *B'* and thus avoided this disagreement. But even if this disagreement does not occur there is always wasted work because different nodes try to mine different blocks. The longer the block takes to reach all nodes, the greater the wasted work will be.

Although it is necessary to send blocks and transactions as fast as possible, it is required to take into account the amount of duplicate information, which will waste the node's computation unnecessarily and contribute to higher network congestion, making it slower [Mat13].

1.3 Challenges

So it is fundamental to improve the efficiency of the dissemination of blocks to reduce the amount of wasted work and also reduce the number of duplicates that affects the bandwidth and processor usage of each node. However, it is not trivial to come up with a solution:

- If a node fails or disconnects, the payload must continue to reach all the peers and not only a fraction. All nodes in the network must have the most recent blockchain to avoid attacks in the system. If an attacker wants to buy merchandise to a merchant, it issues a transaction, and then a block with this transaction is created. But since some disconnections occurred, this block is only received by a fraction of the network, including the merchant . So the merchant will send the merchandise to the attacker, and the attacker creates a concurrent

[3]https://ethereum.github.io/yellowpaper/paper.pdf

transaction sending the same money to himself. The other part of the network will receive this transaction and create a block with this concurrent transaction. In the future, if this blockchain has more work than the other, it will become the truth. In the end, the attacker has the merchandise and also the money. This attack is called double-spending, and it is easier to perform if the information is split across multiple nodes.

- It is necessary to know if the other neighbors already received the payload to avoid sending duplicates that it is a challenging task in a Byzantine model where the nodes can lie (e.g, a byzantine node may not send the block to its neighbors for its own benefit). To reduce the number of duplicates in the network, a naive approach can be implemented where all nodes ask their neighbors if they already received that payload. The number of duplicates will be reduced to 0, but the latency will be increased because there are more messages in the network.

- It is a challenging task to reduce the latency because usually when the latency is decreased, the number of duplicates increases. A naive approach can be implemented where all nodes send the block to their neighbors without asking if they have the block or not. Although the latency is reduced, the number of duplicates in the network will be significantly higher.

1.4 Objectives

The goal of this thesis is to improve the dissemination layer of state-of-the-art blockchains. To achieve that, we will focus on the Ethereum protocol as a case study. We want to reduce the number of duplicate blocks and transactions in the blockchain that effects the CPU and the bandwidth and also try to make the dissemination faster, in order to have less wasted work. In the block dissemination, depending on the selected approach , we improved the dissemination speed, the number of duplicates and consequently the CPU used and the network bandwidth consumption. In the transaction dissemination, we improved the number of duplicates that affects the CPU and the network bandwidth consumption, but the time that each transaction takes to reach all nodes is higher.

1.5 Thesis Outline

The rest of the document is organized as follow: In Chapter 2 we introduce some of the state-of-art concepts where we talk about the key concepts of the overlay network, dissemination in general, the difference between the Bitcoin and Ethereum, blockchains, the consensus in both coins, how Bitcoin and Ethereum disseminate the blocks and transactions, some dissemination systems where we tried to inspire to build our approaches, and also some security problems. In Chapter 3 we explain our approach for the block and transaction dissemination, in Chapter 5 we present the results for our approach comparing to Ethereum Vanilla and in Chapter 6 we conclude our thesis and also introduce some future work and open challenges.

Chapter 2

Related Work

In this section, we will introduce some key aspects to understand the dissertation better. From here on, we consider a *node* someone that participates in the protocol; a *full node* someone that participates in the protocol and also mines blocks; and a *user* someone that has an Ethereum account that can issue transactions communicating with a node or a full node but does not participate in the protocol.

2.1 Bitcoin and Ethereum

Bitcoin and Ethereum are designed for different purposes. While Bitcoin was created as an alternative to the fiat currencies (i.e., money created by governments such as Euro or Dollar) which does not rely on a central authority, Ethereum is a platform that allows the creation of "smart-contracts" which are executed by each node that enables the creation of distributed applications (DApps) on top of it[1] [Nak08]. Although Bitcoin also has a scripting language to write contracts these contracts are limited[2]. In Ethereum, every node runs the Ethereum Virtual Machine (EVM) which can execute code of arbitrary algorithmic complexity enabling the creation of more complex contracts[1]. As each node must run the smart-contract, a fee is charged for each computation. The amount of computation power required to perform a specific task is called Gas and each user specifies how much they are willing to pay for a unit of Gas (i.e., the amount of ether)[3].

[1]https://github.com/ethereum/wiki/wiki/White-Paper
[2]https://en.bitcoin.it/wiki/Contract
[3]https://github.com/ethereum/wiki/wiki/White-Paper#code-execution

2.2 Overlay Network

Each node has a list of nodes with whom it communicates. This list is called the *view* of the node and each node on the *view* is called a *neighbor*. The set of all the views from all the nodes makes an *overlay network*. An *overlay network* is a computer network that is built on top another network, and when creating one, it should have the following properties [Mat13, LPR07]:

1. `Connectivity`. All nodes must be connected (i.e., any node must reach every other node with a finite number of hops)

2. `Average path length`. It measures the average amount of hops of all shortest paths between all pairs of nodes in the overlay. As this value is related to the time a payload reaches all nodes, the overlay is more efficient if this value is low.

3. `Clustering coefficient`. It measures the number of links between the neighbors of a node divided by all possible links across those nodes. The clustering coefficient of an overlay is the average coefficient across all nodes. This value influences the number of duplicates in the network directly. If this number is high, then the number of redundant information is also high. Also, high values of clustering in some areas influences the isolation of these areas from the rest of the network (leading to partitions) under failures and churns.

4. `Degree Distribution`. It is the node's view size and measures the nodes reachability and their contribution to the network.

5. `Accuracy`. It measures the number of neighbors of a node that have not failed divided by its total number of neighbors. The accuracy of an overlay is the average of the accuracy of all correct nodes. If the overlay accuracy values are low, the number of failed nodes selected will be higher.

The overlay can be structured or unstructured; In structured overlays, the neighbors are selected according to some criteria such as latency, while in the unstructured the neighbors are randomly selected with some redundancy. While the structured approach is more efficient because the neighbors are selected with some criteria, it is not robust. In the presence of faults and churns it is necessary to recalculate the overall overlay network. The opposite occurs with the unstructured; it is more robust but less efficient. There are multiple paths between two nodes, so failures and churns do not prevent the message from being delivered as it will be routed to another available path. Figure 2.1 represents an example of an overlay network.

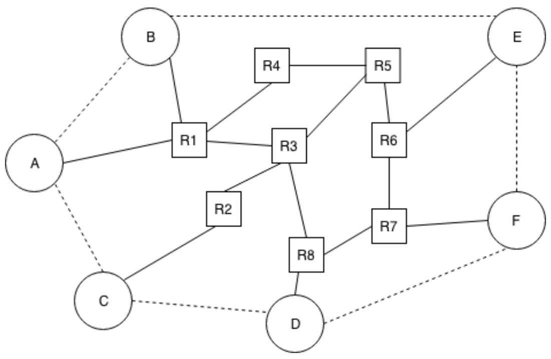


Figure 2.1: Represents an overlay network which is built on top of another network. The overlay links are represented with a dashed line and the links of the underlying network are represented with a line. The payload is sent over the underlying network where it will reach the destination overlay node.

2.2.1 Dissemination

There are three main approaches to data dissemination: *flooding*, *tree*, and *epidemic*, also know as *gossip* [EGKM04, Mat13].

Flooding is the simplest dissemination algorithm, where all the messages are relayed to all neighbors. So there will be many duplicates, which is why it is very demanding in bandwidth.

In the *tree* approach, a dissemination tree without any loop is created. The root of the tree sends the message to its children and the children of the root to their children and so on. As the tree does not have any loop, then no duplicate messages are sent. This approach has the same problem of the structured overlay networks because if a node disconnects or fails it is necessary to recalculate the tree; it is not resilient to failures and churns.

In the *epidemic* approach, the message is sent to a random number of nodes selected from its view. The size of the subset of nodes random selected is usually called *fanout*. This approach is more robust because this random selection allows the creation of multiple paths to the nodes and so the message continues to be delivered with high probability to all nodes under the presence of a large number of churns and disconnects [EGKM04]. There are two approaches to how the message is relayed [Mat13]. The sender can make the initiative to relay the message as soon as it is received (*push* approach) or the node can ask periodically for a new message to a certain neighbor, which will then relay the message to the node (*pull* approach). These approaches can be combined with a *when* decision: *eager* and *lazy* [Mat13]. In the *eager*, the payload of the message is sent to all of its neighbors, while in the *lazy* an advertisement is sent first, and only after that the payload if the neighbor does not have the message and asks for it.

Eager and *lazy* represents a tradeoff between latency and bandwidth. If the payload is always forwarded to the neighbors without advertising it, the latency will be minimal (the node only sends one message that is the payload), but the bandwidth will be higher because the number of duplicate information will be more significant (always forward the payload even if the receiver already has it). On the other hand, if a node sends an advertisement first then more messages are sent if the receiver does not have the payload (advertisement and the payload) but the number of duplicates decreases (a node only forwards a payload if the receiver does not have it and asks for it).

Bimodal multicast [BHO+99] one of the most known gossip algorithms, combines the techniques described above where it guarantees scalability and reliability with some probability. First the nodes will do a best-effort multicast (IP multicast or multicast spanning trees) to reply the message (*eager* approach) and second, each member randomly choose another member and send a digest. The receiver can solicit any message from the sender if it does not receive the message earlier (*lazy* approach). These two phases are done concurrently. But they assume a global membership where each node must know all the nodes in the system. So some work was done to provide a uniform view of the system consuming less memory and requiring no dedicated messages for membership management [EGH+03, LPR07], that are usually called peer sampling service (PSS) [JGKVS04].

The PSS can be reactive or proactive. Proactive is when the nodes periodically exchange their view with their neighbors and so their view can be updated even if the global membership is stable [VGVS05] and reactive is when the view is kept unchanged until some external event occurs on the overlay (a node fails or joins the overlay) [LPR07].

2.2.2 Blockchain

A blockchain is a list of blocks where a block points to the previous one, creating a chain. Each block has the cryptographic hash of the previous block, a timestamp, and data. Satoshi Nakamoto introduced the blockchain in Bitcoin. It is used as a public ledger where blocks record a set of transactions. Each transaction contains the sender address, the receiver address and it is signed by the sender's private key. When a node creates or receives a transaction, it will check if the transaction is valid and put it on its *transaction pool* (an unordered collection of valid transactions that are not in blocks in the main chain[4]). Then the full node will pick a set of transactions from its *transaction pool* and generates a block which will be put on the blockchain, shared by all the nodes [Nak08]. As each block has the hash of the previous one, if a middle block is changed, then it is necessary to recalculate the hash of the next ones, making the blockchain resistant to data modification [CV17]. There are two kinds of blockchain:

[4]https://en.bitcoin.it/wiki/Protocol_rules

permissionless where anyone can run a node, write on the blockchain, participate on Consensus (Section 2.2.3) and be a user. Bitcoin and Ethereum are examples of this kind of blockchains. In *permissioned* blockchains only a specific group of nodes can control, update the blockchain and issue transactions. HyperLedger is an example of a *permissioned* blockchain [CV17].

2.2.3 Consensus

It is important that all nodes agree on which order the blocks are appended, this process of agreement is called Consensus. In Consensus, all correct processes propose a value and they reach a unanimous and irrevocable decision on some value that was previously proposed [CT96]. It is usually expressed with the following properties: *agreement* indicating that if two correct processes decide they decide on the same value, *uniform validity* where a process must decide a value that was previously proposed, *termination* indicating that eventually, every correct process decides some value, and *uniform integrity* where a process must decide at most once [NG16, CT96].

Nakamoto Consensus is a quite different consensus algorithm, that it is implemented on Bitcoin, where some of these properties are relaxed to allow to reach consensus in a scalable and efficient manner on a trustless environment. For instance Nakamoto Consensus does not guarantee agreement deterministically (when a *fork* occurs), instead the agreement is met with probability close to 1 [NG16].

Nakamoto Consensus and Ethereum Consensus

Nakamoto consensus consists of three key aspects: Proof-of-Work (PoW), incentives and a chain selection rule. In Proof-of-Work, it is necessary to resolve a cryptographically puzzle where nodes vote using their CPU power. For a block to be valid, a node has to find a nonce so that the hash of the block header is smaller than the difficulty target[5] of the block, this process of finding a nonce is called mining and the node that performs it is called a miner. Bitcoin uses the Hashcash Proof-of-Work system [B$^+$02] and Ethereum uses Ethash[6]. Ethereum claims that it will change to Proof-of-Stake, but no official date is known[7]. So when a miner finds a nonce that makes the hash of the block header below to the block's difficulty target it will propagate the block over the network. All the other nodes will verify the work and accept it as a valid block, and the miner gets its reward [B$^+$02, Nak08]. It may occur that when a block is being propagated, another

[5]The difficulty is a measure of how difficult it is to find a hash below a given target. It is adjusted every 2016 blocks in order to generate a block every 10 minutes. If on average the block generation time takes more than 10 minutes then the difficulty is reduced, and if it takes less, then the difficulty is increased.

[6]https://github.com/ethereum/wiki/wiki/Ethash

[7]https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs

miner, who has not seen this block yet, mines a concurrent block and starts disseminating it. The network will see two chains, resulting in a *fork*. *Forks* are not desirable, because different nodes will believe in different truths, compromising the agreement property. As disagreements can occur in the network, the blockchain is susceptible to attacks like double-spending and selfish mining (Section 2.2.8).

Bitcoin resolves these *forks* by choosing the longest chain. When a chain has more work than the others, that chain must be considered as truth by the network. The nodes adopt this strategy because of the incentives. They only receive rewards for mining blocks that are in the chain with the greatest mining difficulty. So they are incentivized to follow this strategy [BMC$^+$15].

Ethereum, according to the White Paper, uses a modified GHOST (Greedy Heaviest Observed Subtree) implementation[8]. But the GHOST protocol replaces the "longest chain" rule by the "heaviest subtree" rule, where the uncles (i.e., blocks that are in the smallest chain, that will be discarded) contribute to the total difficulty of a chain. Contrary to what the white paper says, Ethereum does not implement this rule; it uses the "longest chain" rule [GKW$^+$16]. See also Appendix A, which demonstrates that Geth, the official Go client and the most used, does not use the uncles to calculate the total difficulty of the block. Although the uncles do not contribute to the total difficulty, the miners that generate them, receive a reward, to encourage small miners to continue mining, avoiding centralization. Ethereum only accepts seven generations of uncles to avoid many calculations of which uncles are valid to a given block and to force the miners to work on the main chain. Besides each block can only contain 2 uncles.

2.2.4 Bitcoin

Dissemination

Bitcoin dissemination system uses an advertisement mechanism to propagate blocks and transactions (i.e., it uses a *lazy* approach). Figure 2.2 shows the protocol flow between two nodes. When a node receives a block or a transaction it will verify it (step 1) and if the block or transaction is valid it will send an *inv* message to its neighbors, that contains the transaction or block hash (step 2). If the neighbors have never seen this block or transaction before, they will request it, sending a *getdata* (step 3). The node will reply with the information that its neighbors have requested (step 4). The receiver will then verify the information (step 5) and if it is valid, it will send an *inv* message to its neighbors (step 6) and so on and so forth [DW13].

As the block size increases the latency, some improvements were made to reduce the block

[8]https://github.com/ethereum/wiki/wiki/White-Paper#modified-ghost-implementation

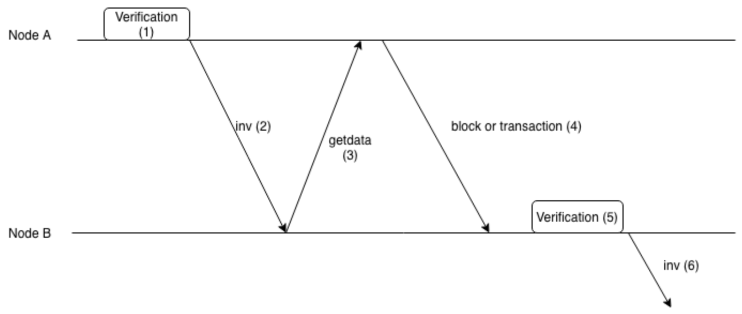


Figure 2.2: Overview of the advertisement dissemination implemented by Bitcoin [DW13] where node A sends some data to its neighbor node B.

size. A block instead of having all the transactions, it has a compact hash of each transaction. These blocks are called *CompactBlocks*.[9] As nodes have already received the transaction, they can use these transactions to figure out which of them corresponds to the compact hash, knowing then what transactions are included in the block. Note that the "normal" block has duplicated information because it is probable that the nodes already received the full set of transactions before. Nonetheless, some nodes might not have received the transaction before. In such scenarios, the node will request it. Thus *CompactBlocks* improve the efficiency of the network by reducing the amount of duplicate information where the same corresponding block is transmitted faster.

Relay Networks on Bitcoin

Currently, Bitcoin has a client, BitcoinCore, that supports relay networks[10]. A relay network is a network where a peer sends or asks a block/transactions to an intermediate node which then sends to the other nodes on the network with much less latency. This approach has some advantages: first, the *forks* will be fewer, because the latency is reduced and second, it makes smaller miners more competitive because bigger miners (mining pools) have a private network where the dissemination is faster than the "normal" peer-to-peer network. As relay networks reduce significantly the latency then the gain of bigger miners to use a private network is smaller.

But a relay network has some disadvantages as well. Each node has to communicate to a specific server, the closest to it, where it sends and receives transactions and blocks from other nodes that also use this relay network. These servers are centrally controlled which violates the premise of decentralization of the Bitcoin network. FIBRE (Fast Internet Bitcoin Relay Engine)[10] is the relay network supported by BitcoinCore which has 6 nodes distributed across the world. It uses UDP instead of TCP, allowing it to use the FEC (Forward Error Correction).

[9]https://bitcoincore.org/en/2016/06/07/compact-blocks-faq/
[10]http://bitcoinfibre.org/

FEC lets nodes to reconstruct the data even if some of it gets lost. FIBRE uses CompactBlocks which was described in the previous section.

2.2.5 Ethereum

Ethereum network communication is comprised of three protocols: Node Discovery Protocol v4 for node discovery[11] (Section 2.2.5) , DEVp2p for application session establishment[12] (Section 2.2.5) and Ethereum subprotocol to retrieve and store information on the blockchain[13] (Section 2.2.5).

Node Discovery Protocol v4

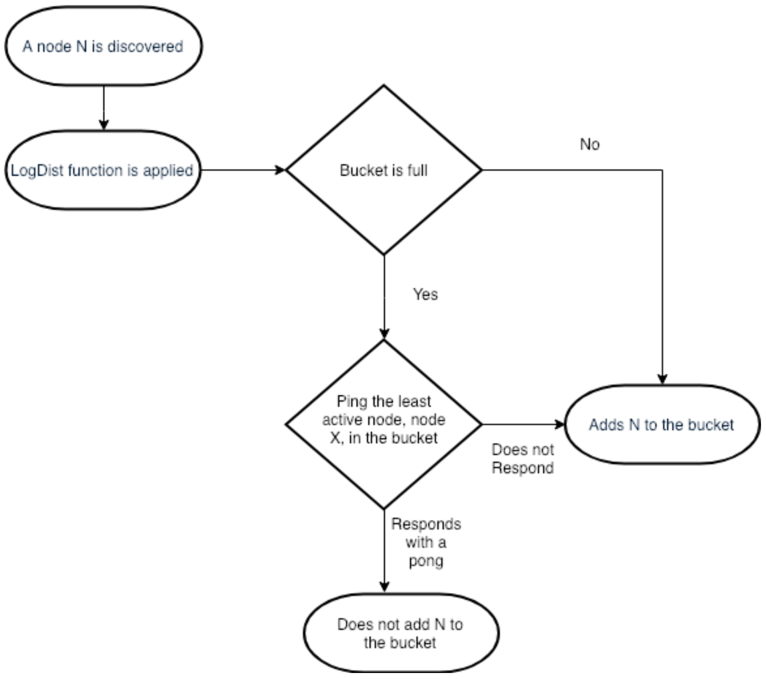

A node N is discovered
LogDist function is applied
Bucket is full
No
Yes
Ping the least active node, node X, in the bucket
Adds N to the bucket
Does not Respond
Responds with a pong
Does not add N to the bucket

Figure 2.3: Flow of adding a node N to the *table* of node Y.

Ethereum uses a modified version of Kademlia[11] [KMM+18, MHG18] to discover new peers. The purpose of Kademlia is to store and find in an efficient manner content in a peer-to-peer network, but Ethereum does not use it to find content (the only content that exists is the blockchain, and all peers store it), only to find new peers or when it is necessary to resolve an IP, that the Ethereum client does not know, for a node ID. Ethereum has *buckets* like Kademlia (i.e., a datastructure where bucket i stores network information about k peers at distance i [MHG18]), in the case of Geth, the Go official client and the most used, 17 buckets are used containing a maximum of 16 peers each.

[11]https://github.com/ethereum/devp2p/blob/master/discv4.md
[12]https://github.com/ethereum/devp2p/blob/master/devp2p.md
[13]https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol

Ethereum uses UDP connections to exchange information about the peer-to-peer network:

- *ping* message to know if the neighbor is alive, where the neighbor responds with a *pong* message;

- *findnode* to ask a peer the 16 nodes closest to a particular target t (it can be a random value or its Node ID) that have been seen by the receiver of the request, the peer will respond with a *neighbor* message with these 16 nodes[11] [MHG18].

All of these messages are timestamped to avoid replay attacks. A peer adds to the message sent an expiration field (i.e., the node's local time plus 20 seconds). The receiver will check if the expiration field is after its local time. If it is then the message is valid. Note that this is not the best solution to avoid replay attacks because an attacker can replay messages within this time range and the clocks of all the nodes must be synchronized [MCBG16, MVGV+17]. Also, an attacker can change the clock of the victim making it only accept connections with its and therefore perform an eclipse attack.

Ethereum has two data structures to store the nodes:

db Persistent and contains all the nodes that the client has seen so far (a node has been *seen* if it responds to a *ping* message sent by the client with a valid *pong* response). Every hour the client runs an eviction process and removes nodes that the last *pong* received is older than 24 hours.

table Not persistent, so when a client reboots the *table* will become empty. The *table* has 17 buckets with 16 entries each. Nodes will be put on the buckets according to the *logdist* function, that corresponds to the similarity of the most significant bits between the client's hash node ID and the node ID hash of the node that will be put on the *table.* If the bucket is full, then the client will only put in the bucket if the least recently active node in that bucket does not respond to a ping packet. Figure 2.3 shows the flowchart of this process. Recently an optimization was made[14] where every 5 seconds on average (a random value is selected between 0 and 10), the last node in a random bucket is checked and if it fails to respond it will be replaced by other node and if it responds then it will be moved to the front of the bucket. This prevents dead nodes from being stuck in the *table.*

The client to discover new nodes uses the *lookup(t)* where it will populate its *table* with the nodes that it receives. The goal of this process is to populate the *table* with distinct nodes in

[14]https://github.com/ethereum/go-ethereum/commit/9123eceb0f78f69e88d909a56ad7fadb75570198#diff-40b2cd29997bbd3f578f9d537789b9bb

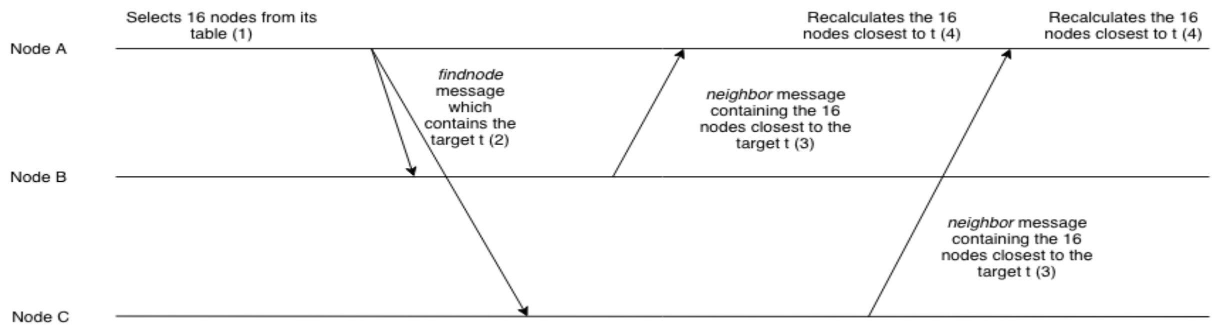

Selects 16 nodes from its table (1)
Recalculates the 16 nodes closest to t (4)
Recalculates the 16 nodes closest to t (4)
Node A
findnode message which contains the target t (2)
neighbor message containing the 16 nodes closest to the target t (3)
Node B
neighbor message containing the 16 nodes closest to the target t (3)
Node C

Figure 2.4: Node A performs a lookup(t), where it sends a *findnode* message to node B and node C.

order to have a uniform sample of the network. Figure 2.4 illustrate this procedure in a simplified way with three nodes. This function will look to the 16 nodes closest to the target t in the *table* (step 1). Then sends a *findnode* message to each of these nodes (step 2) and they will respond with the 16 closest from their *table* (*neighbor* message) to the target t (step 3). Only 3 *findnodes* will be sent at the same time (concurrency parameter defined by the protocol[11]). As soon as the client receives the *neighbor* message, it will recalculate the 16 nodes closest to t, counting with the new ones just received (step 4). This process will be done until this list remains unchanged.

When the node reboots, its *table* is initiated and a seed process will be done. A seed process is a mechanism where the client will select 30 nodes with no more than 5 days (i.e., where the last *pong* is less than 5 days) at random from the *db*. These nodes together with bootstrap nodes (nodes that are hardcoded in Geth) are added to the *table*. After that, *doRefresh* function is called which performs the following steps: one more seed process, one *lookup(self)* (where self is the hash of the client's node ID), and three *lookup(t)* where t is a random target. Note that when a *neighbor* message is received all nodes are added to the *table*.

Only after that, it is possible to add nodes in the *table* through a ping message sent by the other nodes. This is a security measure implemented recently by Geth to prevent the *table* from being completely filled by just sending *ping* repeatedly by many nodes belonging to the attacker as described in [MHG18].

DEVp2p

DEVp2p uses the RLPx transport protocol, a TCP-based transport protocol used for communication among Ethereum nodes. The maximum number of TCP connections are *maxpeers* (by default 25) + *static nodes* (nodes that are hardcoded by the admin), where 8 of *maxpeers* are outgoing connections, that is connections that are initiated by the client itself, and the others are incoming where the other nodes initiate the connection. Note that these connections are unidirectional so when a client makes an outgoing connection to a node X, the client will count towards the maximum number of connections of the node X.

First, it will try to make a connection with static nodes, after that if it does not have any peer and 20 seconds have elapsed it will try to connect to a random bootstrap node. Then, it will select the remainder of outgoing connections divided by two to random nodes in the *table*, the rest will be selected from the *lookupBuf*, which is a queue that has the result of a *lookup(t)*, that is the 16 nodes closest to a random target *t*.

DEVp2p negotiates an application session between two peers. Each node sends a *hello* message which specifies the sub-protocol and version that it supports. After defining the sub-protocol and its version, they can begin to transmit application data packets. During periods of inactivity, they send a DEVp2p *ping* message to ensure that their connection is still active. If a DEVp2p *pong* message is not received within a specific time, then a *disconnect* message is sent to end with the connection. This *disconnect* message specifies the error occurred (e.g., TCP sub-system error, Too many peers, Incompatible P2P protocol version, etc.)[12] [KMM+18].

Ethereum Dissemination

The Ethereum Sub-protocol runs on top of DEVp2p and is denoted as 'eth' during DEVp2p *hello* exchange. The nodes first sent a *status* message which includes the total difficulty of their blockchain and the hash of their most recent block. The peer with worst difficulty (i.e., the blockchain with less work) will ask to the other for the remainder of blocks[13].

After the synchronization, they can start participating in the Ethereum network. Figure 2.5 illustrates how a transaction is propagated in the Ethereum Network. First a user issues a transaction to a node (step 1). The node will then create it (step 2) and propagate it over the network to its neighbors (step 3). The neighbors will check the transaction (step 4), and if it is valid, then they will add it to their *transaction pool* and disseminate it to their neighbors and so on (they will never send the transaction to the node who send it to them) (step 5). As all the nodes send the transaction to its neighbors without advertising it, the number of duplicate transactions received by each neighbor is huge, wasting processing power and congesting the network.

The miner will pick a set of these transactions and try to mine a block. The dissemination of the block is represented in Figure 2.6. The miner mines a block (step 1) and propagates it to the maximum value between 4 and square root of its neighbors (*NewBlockMsg*) (step 2). Note that each node only sends to peers that it knows that do not have the block. If the number of peers that do not have the block is less than 4 then it will send the block to all of them. To the remaining nodes, if any, it will send the block hash and number (*NewBlockHashesMsg*)(step 3);

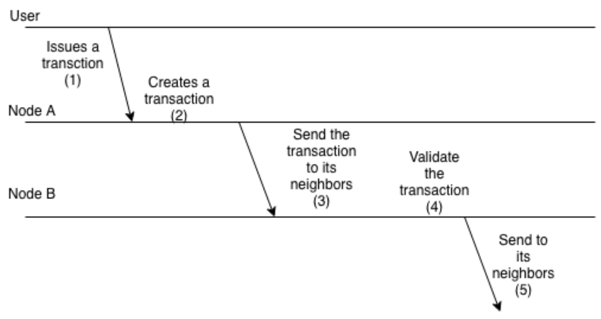

User
Issues a transction (1)
Creates a transaction (2)
Node A
Send the transaction to its neighbors (3)
Validate the transaction (4)
Node B
Send to its neighbors (5)

Figure 2.5: Creation and propagation of a transaction on Ethereum.

each of these peers will perform the following steps:

1. If this hash is unknown on its blockchain, it will request for the block header (*GetBlockHeadersMsg*), waiting at least 400ms (step 4). If the node receives that hash from more than one peer, it will request the header to a peer at random. It is not possible to request a bulk of headers to a peer, only a single one.

2. After receiving it (*BlockHeadersMsg*) (step 5), the node will perform the following verifications:

   (a) Checks if it is expecting that header. If it is not expected the header is ignored.

   (b) Checks if it already received the block header, if it is expecting for the body, or if it already has the body and it is expecting for the block to be included on its chain. If one of these cases occurs then it will reject the header received.

   (c) Checks if the header number corresponds to the number previously announced, if it does not then rejects it.

   (d) Checks if it already has the block in its chain. If it has then rejects it.

   After that, it will verify if the body is empty, i.e., if it does not exist any transaction and any uncle on the block. If this condition is met then the node will create an empty block and put it on its chain.

3. If the block is not empty then it will request for the body (*GetBlockBodiesMsg*) (step 6), waiting at most 100 ms. If meanwhile, it receives the block and puts it on its blockchain, it does not request the body. It is possible to request a bulk of bodies to a peer.

4. When the body is received (*BlockBodiesMsg*) (step 7), it checks which header corresponds to this body received and if it is already on its chain. If not, it will validate the block's proof

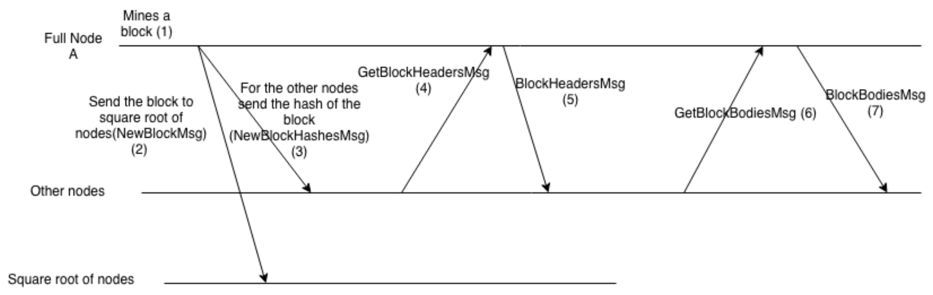


Figure 2.6: Dissemination of a block in the Ethereum Network.

of work using only the block header, and if it is valid, the node will propagate immediately to the maximum value between 4 and square root of its neighbors that it knows that do not have the block. This process can lead to duplicates because the neighbor may have already received this block.

After that, it verifies the uncles (i.e., if the block has at most 2 and if the uncles have a valid header) and checks if the transactions are valid and the transaction root of the block (i.e., if the combined hashes of the transactions sent are the same as the transaction root). Note that the hashes of the transactions are combined forming a merkle patricia trie where the transaction root depends on all of them. So if a transaction is modified then the transaction root will be different[15].

Also the node processes the state changes by processing the transactions received, and then validate the various changes that happen after a state transition (i.e., amount of gas used in the block and check if the receipt[16] and state[17] root are the same as the receipt and state root in the block header). If the block is still valid, it will send the block hash and number to the remainder of its neighbors.

Step 4 is the only step done by the peers that received the block immediately.

2.2.6 Blockchain layer two protocol

Blockchains do not scale well. They require a global consensus to prevent attacks like double-spending (Section 2.2.8). It is not possible to perform fast transactions, and the fees are very high, making it unfeasible to make micropayments. To resolve these problems a blockchain layer two protocol was created where transactions are made off-chain, avoiding the blockchain bottlenecks. In Bitcoin there is the Lightning Network[18] and in Ethereum the Raiden Network[19].

[15]https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/
[16]The receipt trie has information about each transaction after processing each one like the amount of gas used to process the transaction
[17]The state trie contains information about each account such as the balance.
[18]https://blog.bitmex.com/wp-content/uploads/2018/01/lightning-network-paper.pdf
[19]https://raiden.network/faq.html

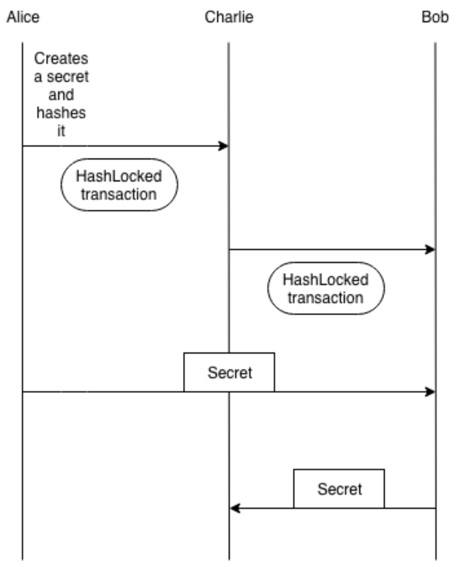


Figure 2.7: Multihop transfer in Raiden Network where Alice wants to send money to Bob but does not have a payment channel open with him.

In general, in Raiden Network [NF18], two entities create a payment channel where they deposit some tokens on a smart-contract on-chain. The two parties can now make off-chain payments that are digitally signed, avoiding the blockchain limitations. The two parties can later withdraw their money providing to the smart-contract the last transaction that each of one received. Then the contract validates it and sets the claims on the blockchain.

Whenever someone wants to transfer money to a particular party, he/she does not have to always open a payment channel. Instead he/she can use a network of payment channels that relay the transaction to the destination. So the transfer is locked by the sender using a Hashlocked transaction where the transfer cannot be claimed on-chain if the party does not know the secret.

For example Alice wants to transfer money to Bob. She does not have an open payment channel with Bob and so decides to relay the transfer to Charlie that has a payment channel open with him. She creates a Hashlocked transaction and sends to Charlie. Now Charlie does the same thing and produces the same Hashlocked transaction. Bob then will ask Alice the secret, and Alice sends it. Now Bob unlocks the transfer and sends it to Charlie that claims the Alice transfer. This example is illustrated in Figure 2.7.

2.2.7 Dissemination systems

We already discussed how the dissemination of the Ethereum works. The number of duplicate transactions is huge (the nodes only apply an *eager* approach) [KMM+18]. In the case of blocks, Ethereum loses a bit of efficiency to have fewer duplicates. The nodes apply an *eager* approach where they send to at least 4 nodes the block. To the others, the nodes apply a *lazy* approach where they advertise the block. In this section, we describe Brisa and Bar Gossip, two systems that can help to find a better solution. Brisa, that is an efficient and robust system but that only works on a crash model. And Bar Gossip, that takes into account byzantine and rational nodes like in Ethereum. Also we talk about Thicket that is an algorithm that generates multiple trees allowing load distribution where the majority of nodes are interior in one tree and in the rest are leaf.

Brisa

Brisa [MSF$^+$13] is a hybrid approach which combines the robustness and scalability of an epidemic with the efficiency of dissemination structures (trees and directed acyclic graphs (DAGs)). Brisa is built in such a way that upon failures and churns, these structures are rapidly repaired. Brisa relies on a peer sampling service (PSS) that ensures that the overlay network formed by all the views of each node is connected. i.e., every node can reach every other under a high rate of failures and churns. It uses the HyParView [LPR07] where each node has two views: a smaller *active view* which is used by the application and a larger *passive view* which is periodically exchanged and shuffled with other randomly selected neighbors. When a neighbor in the *active view* disconnects or fails, it will be replaced for another in its *passive view.*

Initially, all nodes send the message to their view (*flooding* approach), and after that, each node autonomously selects one as its parent and sends a *deactivation message* to the others. Future messages will then only be received from the parent node. The parent can be chosen using a *First-come first-picked* strategy where the node sending the first message is selected as the parent. When a parent fails or disconnects, the node will reactivate all its inbound links and proceed with a normal parent selection.

It may be useful to have many dissemination structures for example if there are many sources in the system. So the authors propose a solution where each structure is uniquely identified, *flowID* and the source tags the message with its *flowID*, and the other nodes follow this structure. The state that each node must have to maintain grows linearly with the number of trees (i.e., if a system has N nodes and all the nodes can be the source then each node must maintain the state of N trees). To mitigate this, they propose a tree reusing strategy where if the source is very close to the root of the structure, the node will use it to disseminate the message.

Bar Gossip

Bar Gossip [LCW$^+$06] is a p2p data streaming application that provides low latency and predictable throughput in a Byzantine/Altruistic/Rational (BAR) model. On this model, the node can be altruistic, if it follows the protocol, can be byzantine if it behaves maliciously in order to undermine the system and rational if it does not follow the protocol in order to have more benefit.

First, each client generates a session pair consisting of a public and private key. Clients sign up for the service providing both keys to the broadcaster. During the stream, the broadcaster divides the stream into chunks, and in each round, the broadcaster sends this chunks to the subscribers and also they are exchanged among clients. After a while, these chunks expire and are retrieved to their media players.

Because a client is unlikely to receive all updates from the broadcaster, they use two protocols to exchange the chunks across clients: *Balanced Exchange Protocol* and *Optimistic Push Protocol*. In *Balaced Exchange Protocol* the clients trade the chunks one for one. So a rational client is motivated to follow the protocol and in *Optimistic Push Protocol* the initiator forwards chunks, in the hope that its partner retributes the favor.

There are Proofs of misbehavior (POM) which ensure that if a client deviates from the protocol specifications, then this client can be evicted from the system. The auditor is responsible for policing the system by ordering to random clients for POMs against others. If a client does not have a POM, it must send a dummy message. If the client does not respond then, the auditor interprets as if it has misbehaved. Although the auditor waits sufficient time for a client to respond, an attacker can drop POMs sent by the client, and these POMs never reach the auditor. The auditor will then evict the client without it misbehaved.

Thicket

There are two types of nodes in a spanning tree: the leaf nodes and the interior nodes. The interior nodes are the nodes that are inside the tree, and they relay the message to their children. The leaf nodes are the nodes that are at the end of the tree, and because of that, they do not relay messages. The load in the interior nodes is much higher than the leaf nodes. One way to overcome this issue is to create multiple trees where each node is interior only in a few of them and leaf in the remaining. This way to construct numerous trees allow load distribution and introduce redundancy for fault-tolerance. The Thicket is an algorithm that creates multiple trees, where the majority of nodes is interior only in one tree and leaf in the other trees. The algorithm is specified in [MF10] where it only works in a crash model. For a particular tree, if an interior node decides not to send the message, the other nodes will not receive that message.

2.2.8 Security

In this section we will present some attacks that can be made in the Ethereum Network that must be taken into account when performing a solution.

Double Spending

This attack occurs at the consensus level. Double Spending is when an attacker sends some money to a merchant on transaction t1 and then creates another transaction where it transfers this money to himself, transaction t1'. While t1 is added to the blockchain, the attacker starts doing its own chain where it includes t1'. After receiving its merchandise, it will broadcast its chain. As the attacker's chain is the longest, it will be considered as the main chain, and transaction t1' will be considered valid and t1 invalid because the funds are already spent in t1'. The attacker then receives its merchandise without paying anything. To perform this attack it is necessary to control 51% of the network which is not feasible. Note that it does not always compensate to make this attack because mining blocks also has its cost [GKW$^+$16, NG16].

Selfish Mining

This attack occurs at the consensus level. Selfish mining is an attack that affects the integrity of the blockchain. It happens when a selfish miner (one miner or mining pool) does not propagate its blocks as soon as they are created. Instead it will keep them, creating its own chain, where its chain is longer than the public one. When the public chain is catching up its own chain, the selfish miner will release the blocks to the network. In the end, the selfish miner receives the block rewards to which it is entitled. It assumes that not playing fair increases its returns [GKW$^+$16].

Eclipse attack

This attack occurs at the membership level, which consists of isolating a peer from the network; the attacker establishes all incoming and outgoing connections with the victim [GKW$^+$16, MHG18, HKZG15]. Thus controlling all messages from the victim to the network and vice-versa. So the attacker can take advantage of this to attack the blockchain's consensus algorithm where it leverages the victim's CPU power to perform double spending and selfish mining.

Also, the attacker can perform attacks on the blockchain layer two protocols because the security of these protocols depend on that no off-chain transaction is made when the payment channel is closed. So the attacker can trick the victim to think that the payment channel is still open, and it accepts the attacker off-chain transaction. For example, a seller can release merchandise in exchange for an off-chain payment, and the attacker can eclipse the seller to obtain the commodity without paying.

Either, the attacker can show to the victim an inconsistent state of a smart contract that it has an interest. For instance, an Ethereum smart-contract can be used to auction a digital cat (e.g., a CryptoKitties cat) where Alice only wants to buy if the number of bids, x, is less than 3. The attacker can trick Alice showing a previous version of the smart-contract where $x < 3$ and she makes a proposal, even though $x > 3$ for the rest of the network.

2.2.9 Discussion

It is essential to have an efficient propagation to make *forks* less likely to occur. Furthermore, the number of duplicates must be taken into account because they waste processing power of each node unnecessarily and congest the network. Also, the view of each node must be a uniform sample of the system to avoid partitions in the network and to be difficult for an attacker to eclipse a particular node. But the membership part will not be addressed in this thesis. We only will focus on the dissemination and we assume that each node has a uniform sample of the system. In Ethereum the number of duplicates is very high on transactions where each node applies an *eager* approach. To disseminate blocks the nodes utilize a combination of *eager* and *lazy* where the number of duplicates is reduced in exchange for poorer efficiency, but the network also receives some duplicate blocks. In this dissertation, we will focus on improving these dissemination issues.

In the literature, there are several approaches to enhance dissemination in a p2p system. However, due to the specificities of blockchains and the strong incentives for a rational approach, a novel approach is needed. Brisa is a system robust and scalable as the epidemic and efficient because uses dissemination structures (trees, DAGs). But it only works on a crash model. In a Byzantine model, a node can decide not to forward the message to its children, can lie about its position to the source, so it is challenging to know the real path to the source. In BAR Gossip, although it already takes into account Rational and Byzantine nodes, the system is kind of centralized where there is a broadcaster that has the private and public keys of the peers. If this broadcaster is malicious, it can impersonate every node. Also, there is an auditor that evicts peers. It can also evict anyone that it wants even if they followed the protocol. So the system relies on two trusted parties: the broadcaster and the auditor. But, we want to continue to have a completely trustless environment. Also, it is necessary to take into account the attacks possible on the blockchain when developing the solution.

In the next section, we present our proposal.

Chapter 3

Block dissemination

In this chapter, we will describe our approach, which aims to reduce the number of duplicate blocks and consequently reduce the CPU usage and bandwidth consumption. Although Ethereum applies a combination of eager and lazy, the number of block duplicates remains relatively high. When a node receives a new block, it will propagate it to the maximum between 4 and the square root of its neighbors, without asking them if they already know the block or not.

In order to reduce the number of duplicates and consequently reduce the CPU usage, we build a dissemination tree similar to Brisa but adapted to a Byzantine environment.

Analyzing the Ethereum network, we notice that only a few mining pools were responsible for generating blocks: "Ethermine", "SparkPool", "Nanopool"," F2_Pool2" and "MiningPoolHub_1" [1], opening the opportunity to explore an approach based on trees. Each node only have to store a few trees occupying a few space in disk. If all nodes generate a block then each node will have to save a huge amount of trees, equal to the number of nodes in the network which will occupy a lot of space. The nodes will use the miner's address to identify each tree, which is in the block, in *coinbase address.* When a node receives a block, it will look to the *coinbase address* and use it to identify the tree that it must apply in the dissemination.

3.1 Construction of the dissemination tree

The tree is constructed in the same way as in Brisa: initially, all nodes will send the block to all of their neighbors and choose as parent the node that sent to him the block first, sending a deactivation message to all the others. A possible execution is depicted in Figure 3.1 where are represented 5 nodes: A, B, C, D and E. Node B will send the new block first (red arrow in

[1]https://etherscan.io/

step 2), to a certain *coinbase address*. The other nodes will still send the same block, because they do not know that Node A already has the block (blue arrows in step 2). As Node B sent the block first, Node A will send deactivation messages to all the other neighbors (step 3). These deactivation messages will inform the others to announce the block, instead of sending it directly. Hence, they will start to apply a lazy approach. The nodes have to continue to advertise the block, because the parent can be malicious and decide not to send the message as the environment is Byzantine. The tree is then constructed, where all *links* to where the block is sent directly constitute the tree. In Figure 3.1, in step 4, the dashed line corresponds to an indirect link and the continuous line to a direct link.

After the tree is built, when a given node generates a second block, nodes will use the *coinbase address* to identify the tree to be used in the dissemination. The block will be sent directly to all neighbors who did not send any deactivation message for this *coinbase address*. For the rest, it announces the block.

Without any Byzantine node, failure or disconnections or no network failures, the number of duplicates will be 0, since everyone is following the protocol and the tree does not need to be rebuilt.

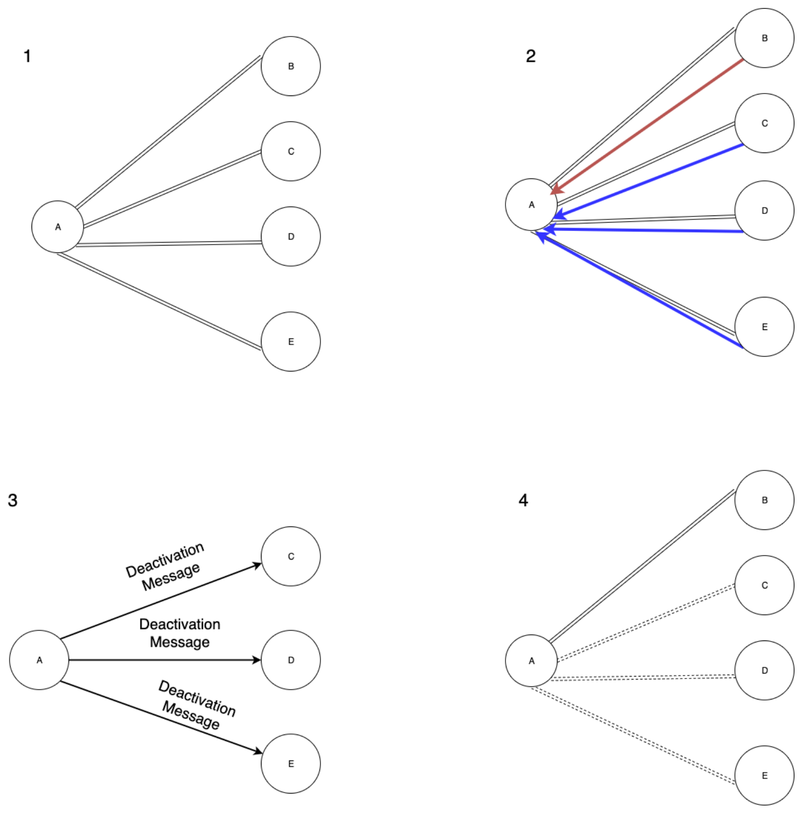


Figure 3.1: The multiple steps in the construction of the tree for a given coinbase address where the double continuous line corresponds to a direct link between two nodes and the double dashed line corresponds to an indirect link.

3.2 Repair the dissemination tree

If some problem occurs in the parent node (e.g., it is byzantine or it disconnects) and the node receives the block first by a non parent node, the tree must be repaired. Figure 3.2 shows the steps of choosing the new parent when the old one behaves incorrectly. In step 1, Node C announces and then sends the block first for a given *coinbase address* because Node B is Byzantine and it decided to not send the block to node A. Node A will then send an activation message to the new parent, Node C (i.e., notifies the node to send the next block directly) and a deactivation message to the old one, Node B. The new tree is then constructed (step 3).

If the parent fails or disconnects, the new parent can be chosen in two ways:

1. The node chooses as parent the neighbor who announces the block first more times to him to that *coinbase address*, leading to 0 duplicates. The process is similar to Figure 3.2.

2. If there are not any announced block for that *coinbase address* the node sends an activation message to all its neighbors i.e., it informs all its neighbors to send them future blocks. Then it chooses as parent the node that sends to it the block first, leading to many duplicates. Figure 3.3 shows this process where node A sends an activation message to all of its neighbors (green arrow in step 2). The neighbors will then send the next block directly to node A (step 3). Node A will then select as parent the node who sends to him the block first, following the same steps in Figure 3.1.

3.3 How a Byzantine node can attack the system

In the presence of Byzantine nodes, these can attack the system in four ways:

1. By not sending the received block to its neighbors, affecting only the time of receiving the block, if it is a parent. Its neighbors continue to receive the block by other nodes, and the one that sends the block first to him becomes the parent and the Byzantine node ceases to be (the node sends an activation message to the new parent and a deactivation message to the old one).

2. Not following the protocol and sending the blocks to all its neighbors, affecting the number of duplicates in the system. The same problem can happen in Ethereum Vanilla, so it will not be addressed in the rest of the thesis.

3. Sending a block with header and body invalid. An attack that is also not addressed in the evaluation, because the effects caused in Ethereum Vannila would be the same as in our solution (network congestion).

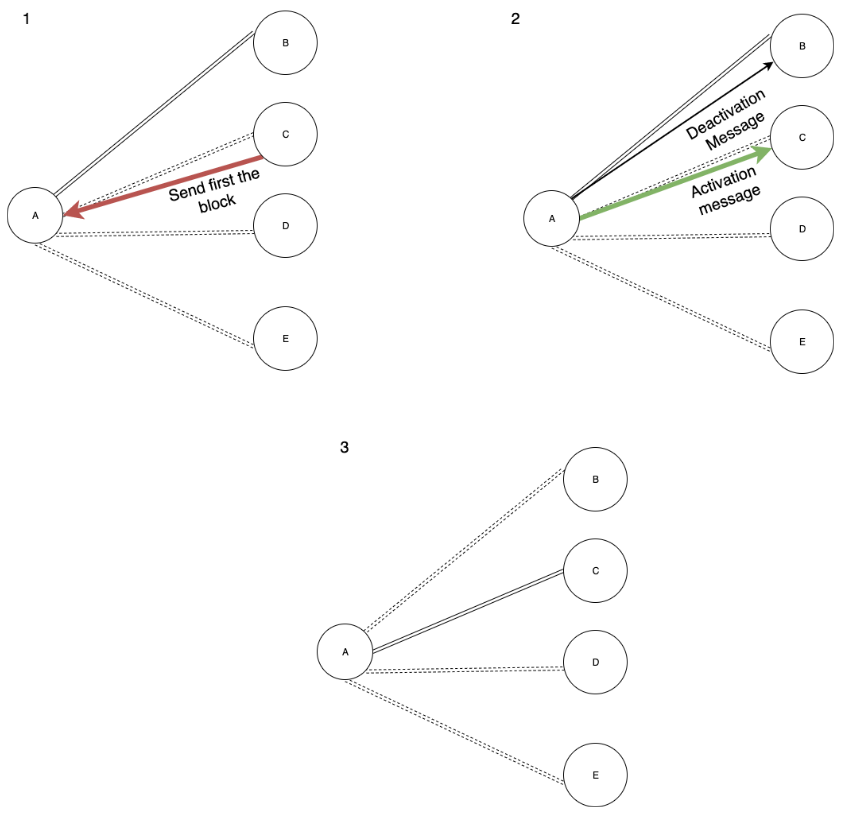


Figure 3.2: Repair a tree when the parent is byzantine. The double continuous line corresponds to a direct link between two nodes and the double dashed line corresponds to an indirect link.

4. Sending blocks with valid headers and invalid body. This situation will be addressed in the evaluation because correct nodes in Ethereum Vanilla disseminate these type of blocks, causing congestion on the network and affecting the CPU usage of each node.

3.4 Implementation

To apply our approach, we had to understand all Geth's code and adapt it to work on our local network. Additionally, we have to understand all the dissemination problems that Ethereum has and try to come up with a better solution.

So Geth has different modules where we have changed the eth module, more precisely the handler.go and fetcher.go. The handler.go is responsible for handling the messages received in the network. And the fetcher.go is responsible for checking if the header and the body are valid.

In the handler.go, we add two more messages in the network:

1. *DeactivationMessage* to inform the neighbor, for in the future, to send the hash of the block for a given *coinbase address* and to only send the full block when the neighbor asks him for it (the node must send a *BlockBodiesMsg*).

2. *ActivationMessage* to inform the node, for in the future, to send the full block for that

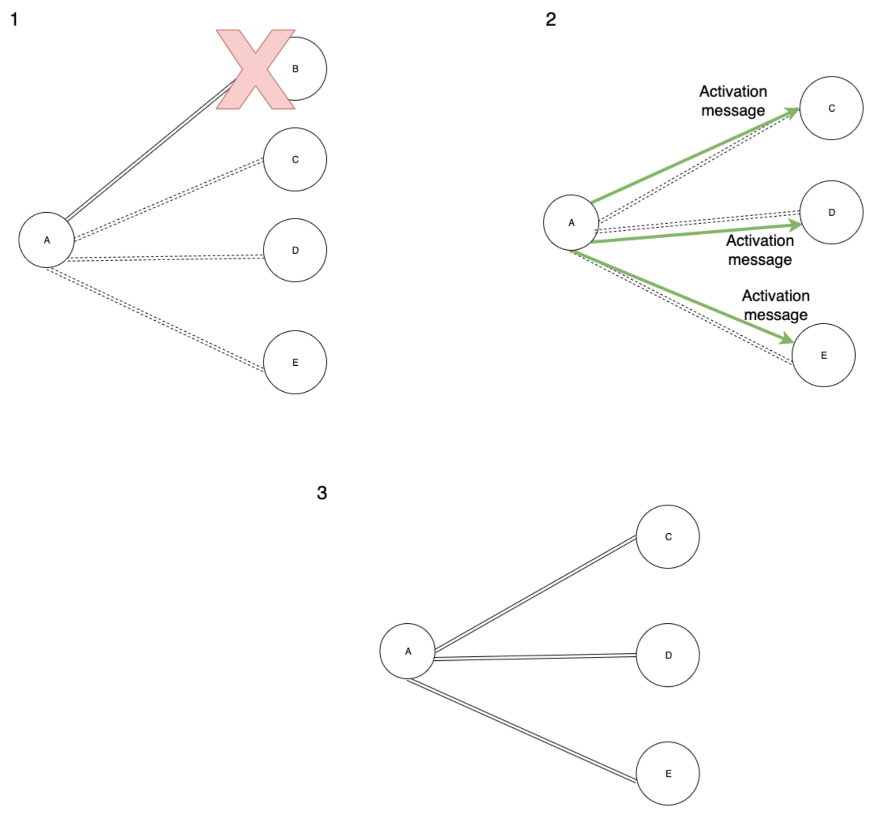


Figure 3.3: The different steps that a node does when its parent disconnects or fails and does not exist any announce for that tree. The double continuous line corresponds to a direct link between two nodes and the double dashed line corresponds to an indirect link.

given *coinbase address* to him.

We also add a hashMap in each node to store the neighbors where it must send the announce of the block for a given *coinbase address* (peer.announce[coinbase.String()]). When a node receives a *DeactivationMessage* from a neighbor, it will add it to the hashMap and if it gets an *ActivationMessage* it will remove the neighbor from the hashMap.

We also changed the BroadcastBlock function where the node will check if it must send the full block, or it must announce it for each neighbor.

If a peer fails or disconnects, the node will check if it is a parent node. If it is, the node will send an *ActivationMessage* to the node that sends the announces first more times because the link between them is faster than the others. If the node did not receive any announce for that *coinbase address* yet, it will send an *ActivationMessage* to all his neighbors.

We also implemented Byzantine nodes which can attack our system in two ways:

1. By not always send the block where they will in a probabilistic way, send the block or not. A random value is generated between 0 and 1, and if that value is below 0.5, then it will send the block. Otherwise, it will not send it.

2. By sending the block with a valid header and with an invalid body where it will change the *coinbase address* with a random one. Also, it will change the body with all transactions in his transaction pool, to make the body invalid.

In the fetcher.go, when the block is fully received and fully verified, the node will check if it was the parent who sent that block to him. If it was, nothing happens. Otherwise, if that node is not in the *blacklist HashMap* for that given *coinbase address*, it will change the parent. Also, it triggers an *ActivationMessage* to that neighbor and a *DeactivationMessage* to the other. Furthermore, it will add a count to the old parent and check if the count is equal to 3 (maximum number of times that the parent can be changed). If it is equal, then the neighbor for that given *coinbase address* is inserted into the *blacklist HashMap* and cannot be used as a future parent. If the tree is not constructed, then the node that sends the block first is selected as a parent, and it will send a *DeactivationMessage* to the other neighbors using handler.go.

Also, we have inserted our logs to check if our protocol is running correctly, where each node will write to a specific log file. And we created different Geth clients depending on the approach used regarding at what stage the block is sent (TreeBefore, TreeSquareRoot, TreeAfter), see Section 5.1.4 and also depending on if the client is byzantine or not.

3.4.1 Multiple solutions

We implemented multiple solutions in our approach:

- Without any Bizantine node in the network, the nodes will send the block as soon as they verify the block's header. As there are no invalid blocks, the block will be propagated faster in the tree.

- With Byzantine nodes we applied different approaches because the nodes can create a block with a valid header and an invalid body, resulting in a large number of invalid blocks. So in this environment, we evaluate three different solutions which are explained better later on, in Section 5.1.4.

Chapter 4

Transaction Dissemination

In Ethereum Vanilla, the nodes in the network send transactions to all of their neighbors. Consequently, as the nodes do not check if their neighbors already have the transactions, the number of duplicate transactions is significant. Therefore we need to find a solution that reduces the number of duplicates in the network without affecting too much the latency. The time that each transaction takes to reach all nodes must be shorter than the insertion block time into the blockchain because if this happens, the next block can include this transaction.

4.1 Difficulties

In contrast to block dissemination, we cannot apply a tree approach because all the nodes in the network can issue transactions and so the nodes will have to know a considerable amount of trees. Note that in the blocks' case, only a few nodes generate blocks. If the system has N nodes, each node will have to save N trees. So we have to apply a different approach to the transactions.

In Ethereum Vanilla, a node can receive the same transaction from various neighbors. Therefore when it sends the transaction, it only sends it to the other neighbors. During the sending process, if a node is simultaneously receiving the same transaction from another neighbor, the node will send the transaction anyway to that neighbor. This happens because the node only learns that neighbor has the transaction when the node has completely received the transaction from him. A possible solution but very difficult to implement is for the node to check if it is receiving something from its neighbors at the TCP level and wait until all information is received. This approach would reduce the number of duplicates, but the node would be blocked for a long time if the neighbor is sending a full block instead of that transaction, which would penalize the latency (time that each transaction takes to reach all nodes in the network) too much.

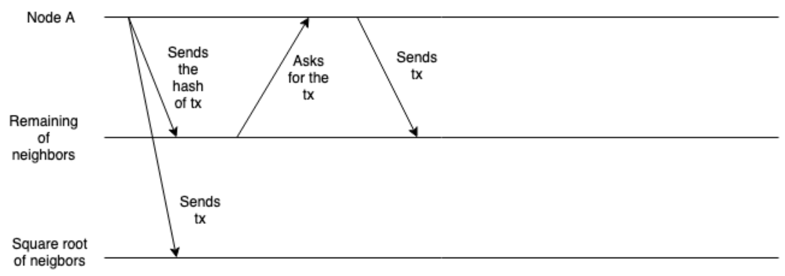


Figure 4.1: The messages exchanged when a node A is sending a transaction.

4.2 Implementation

As the previous approach is very complicated to implement and we think that can affect the latency too much, we will implement the solution that Ethereum uses in the blocks' dissemination for transactions. As said before, in Ethereum, nodes can send the transaction to the neighbors who already know it, because the nodes never ask their neighbors if they have it or not. So the best solution we found was when a node receives a transaction, it sends the transaction directly to the maximum between 4 and square root of its neighbors and for the rest announces the transaction, i.e., ask the neighbors whether or not they have the transaction. See Figure 4.1. We use these values because Ethereum Vanilla uses them in the block dissemination, so we assumed that they must be the optimal values. For future work, we want to use different values and check if they are the best. As nodes only send the full transaction to a small number of neighbors and announce it to the others, the number of duplicate transactions will be reduced. In blocks ' case, Ethereum Vanilla applies the same approach. They send the block directly to the maximum between 4 and square root of its neighbors, and if the body of the block is valid, they will announce it to the remaining.

With this approach it is expected that the number of duplicates is greatly reduced, and latency is not significantly affected.

Chapter 5

Results

5.1 Dissemination of blocks

In this section, we evaluate our approach to improve the blocks' dissemination in a realistic environment, using the following metrics:

1. Number of duplicate blocks in the network, by block and then by the number of duplicates that each node receives.

2. Latency that each block takes to reach all nodes.

3. The percentage of CPU used during our experiments in each approach.

4. The number of bytes that each node received in the network. Metric that measures the bandwidth used.

We develop a private blockchain in our local network where we use the Geth client in 1.8.17-stable version. We run 45 nodes in the same machine with the following specifications: an Intel (R) Xeon (R) Gold 6138 CPU with 2 cores of 2.00 GHz, each with 2 threads and two 32GiB DDR4 synchronous DDR4 2666 MHz RAM. Only three of these nodes will generate blocks, simulating the real environment: only a few percentage of nodes are responsible to generate blocks.

We ran one-hour tests, where we skipped the initial and final 10 minutes for the Ethereum Vanilla and for our implementation, where we had to introduce new messages into the network in order to build the dissemination tree, as explained before.

5.1.1 Ethereum Modifications

In order to generate blocks in our private blockchain, we changed the POW of Ethereum, where each block is generated approximately every 20 seconds (average block generation time in the

Ethereum network). Otherwise, it would take a lot more time for a block to be generated. In this modified version a timer is used, where the generation time of each block follows a Poisson distribution.

In the transactions' case, they are injected 4 per second, yet much less compared to those that Ethereum receives (about 12 to 15 transactions per second[1]). These transactions are propagated in the network following the Ethereum protocol. We inject only 4 per second to not overload our machine too much and thus have more nodes running in our machine. Even injecting more transactions per second, the results would be pretty much the same.

5.1.2 NEED tool

We used the NEED tool [Nev18] to introduce latency into the network and set the bandwidth to simulate the Ethereum environment as realistically as possible. We separated the nodes into several groups (4 groups for the experience without any Byzantine node and 6 groups for the experience with Byzantine nodes) and define the latency between them and the bandwidth:

1. Between *group1* and *group2* the latency is 50ms and the uploading and downloading of 100Mbps
2. Between *group1* and *group3* the latency is 100ms and the upload and download of 50MBs
3. Between *group1* and *group4* the latency is 1000ms and the upload and download of 20Mbps
4. Between *group1* and the *group5* the latency is 60 ms and the upload and download 30Mbps
5. Between *group1* and *group6* the latency is 80ms and the upload and download 10Mbps

In the experiment without Byzantine nodes, *group1*, *group2* and *group3* have 10 nodes each and *group4* 15 nodes, whereas in the experience with Byzantine nodes the *group1*, *group2*, *group3* and *group4* have 10 nodes, *group5* has 1 and *group6* has 4. These last two groups correspond to Byzantine nodes, where *group5* corresponds to nodes that send a valid header but an invalid body and *group6* corresponds to nodes that sometimes do not send the block.

5.1.3 Experiments without any Bizantine node

Figure 5.1 and 5.2 represent the number of duplicates without any byzantine node. As the reader can verify the number of duplicates is reduced to approximately 0. In Figure 5.1, is represented a Cumulative Distribution Function (CDF) of the number of duplicates in the network per block, and approximately 100 % of blocks has 0 duplicates. In Vanilla, this number

[1]https://etherscan.io/

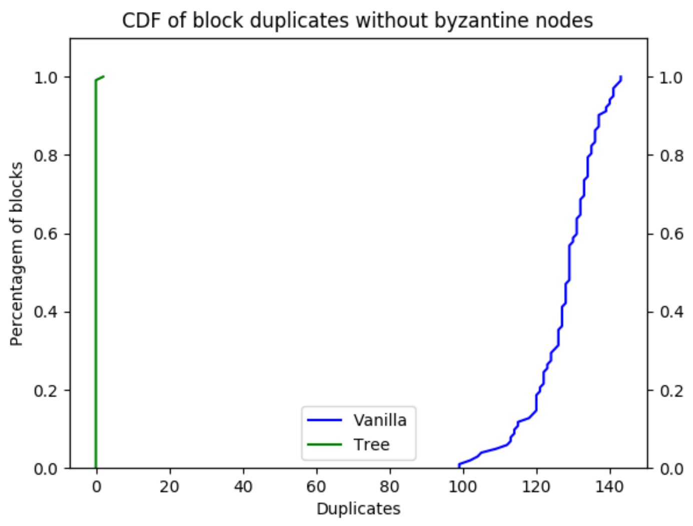


Figure 5.1: CDF of the number of duplicate blocks in the network.

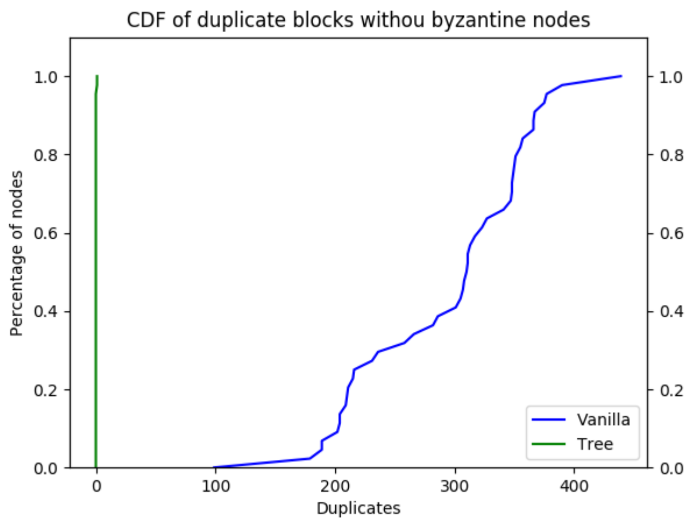


Figure 5.2: CDF of the number of duplicates that each node receives.

is significantly higher where the network received 130 duplicates in 50% of blocks. Figure 5.2 shows the number of duplicates that each node receives, and it is also reduced to approximately 0. The number of duplicates is not exactly 0, because it occurred some disconnections and the tree was reconstructed. The blocks propagated during the reconstruction process had some duplicates in the network. The nodes that do not have any parent because it disconnected, ended up to send activation messages to the other neighbors and these neighbors will send the next block to this node directly, ending up to receive duplicates. In Vanilla, some nodes received over 400 duplicates, about 4 per block because in Vanilla the nodes send the block directly to the maximum between four and square root of its neighbors.

Also, we measure the CPU used in each approach. For that, we used the *dstat* tool in our

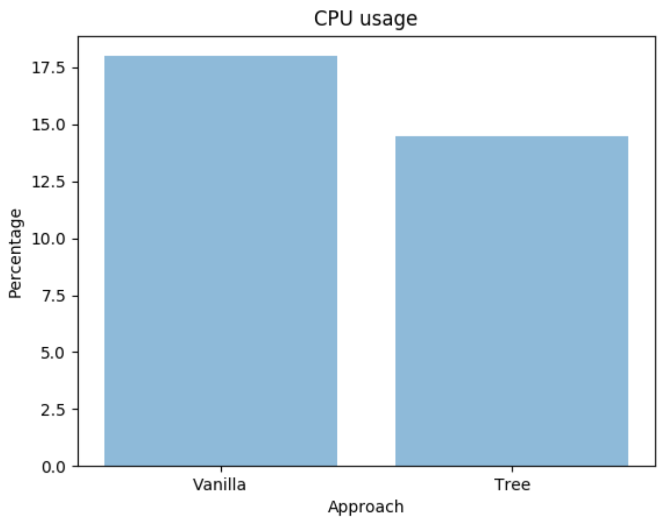


Figure 5.3: CPU used in each approach.

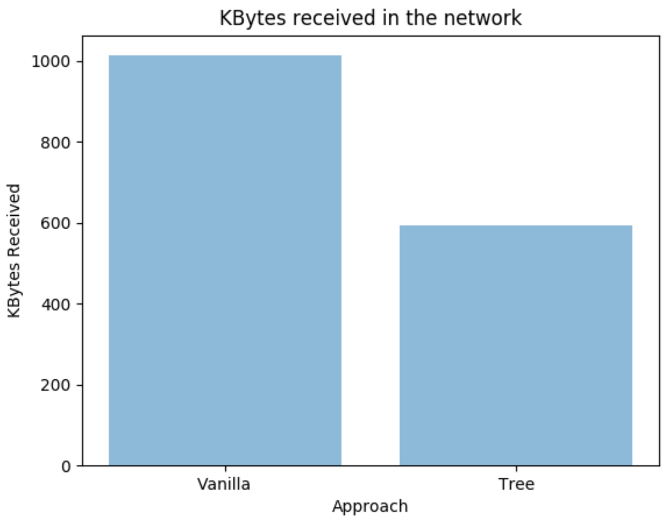


Figure 5.4: KBytes received in each approach

machine that measures the average CPU usage during the experiment. As the reader can verify, in Figure 5.3, Ethereum Vanilla uses a bit more CPU than our approach a difference of about 3%.

Also, we measure the number of bytes that the network receives. When a node receives a message, it will write the number of bytes to a file. Then we sum all the bytes that all the nodes received and divide by the number of blocks created. We divide by the number of blocks because, in different runs, it may happen that the number of blocks created is slightly different (remember that the function of creating a block is probabilistic), affecting the overall bytes in the network. Figure 5.4 represent the number of bytes received in each approach per block. This number is almost reduced by 400 Kbytes (from 1013 to 594 KBytes), because in our solution

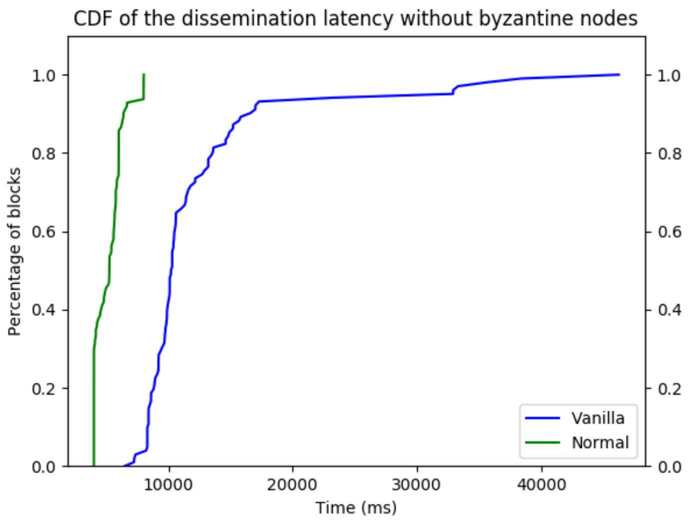


Figure 5.5: CDF of the time that each block takes to reach all nodes in the network.

the number of duplicate blocks is reduced to approximately 0, using less bandwidth.

Figure 5.5 shows the latency that each block takes to reach all nodes in the network. With our approach the latency is improved significantly. In Vanilla, some blocks took more than 40 seconds to reach all nodes while in our approach, all blocks took less than 10 seconds. Our method is much faster because the tree is built based on the speed of reception of the first block, where the parent if not Byzantine, sends the block to his child.

5.1.4 Experiments with Bizantine nodes

We also evaluated the system with byzantine nodes. In the network with 45 nodes, 5 of them are byzantine (which four of them not always send the blocks and one of them sends a block with a valid header but with an invalid body). To generate an invalid block, a byzantine node changes the *coinbase address* to a random one, when it receives a block. So when it propagates the block to their neighbors, they do not choose any tree because they did not see that *coinbase address* previously. Then, if the neighbors decide to send the block they will send it to all of their neighbors and so on. In the end, there will be many duplicates and many invalid blocks. So we have fundamentally three possible solutions to try to reduce the duplicates and invalid blocks without affecting too much the latency:

1. The node sends the block as soon as it verifies the header. The latency will be decreased, but the number of invalid blocks in the network will be huge (TreeBefore approach).

2. The node only sends the block when it verifies the block completely. This will affect the latency, but the number of invalid blocks will be significantly decreased (TreeAfter

approach).

3. The node applies an approach equal to the Ethereum Vanilla. When it verifies that the header is valid, it will send the block to the maximum between 4 and the square root of neighbors that it thinks that they do not have the block. The node privileges the neighbors that are children, i.e., the neighbors that it will send the block directly. After checking the full block, if the body continues to be valid, it will send the block to the other nodes. Otherwise, the node will not send it (TreeSquareRoot approach).

   It is expected that the latency is affected, but the number of invalid blocks is reduced, in comparison with the TreeBefore approach.

Figure 5.6 and 5.7 show the number of duplicate blocks counting with invalid blocks per block and per node, respectively. The number of duplicate blocks is practically 0 in the TreeAfter approach because the nodes only send the block if it is entirely valid. In the TreeBefore the number of duplicates is significantly high surpassing the Etherum Vanilla because correct nodes will send the invalid block to all of their neighbors. Note that the invalid block have a completely different *coinbase address* and so the block will not follow any tree. TreeSquareRoot is slightly better than Ethereum Vanilla, but 20 % of the blocks have precisely the same number of duplicates.

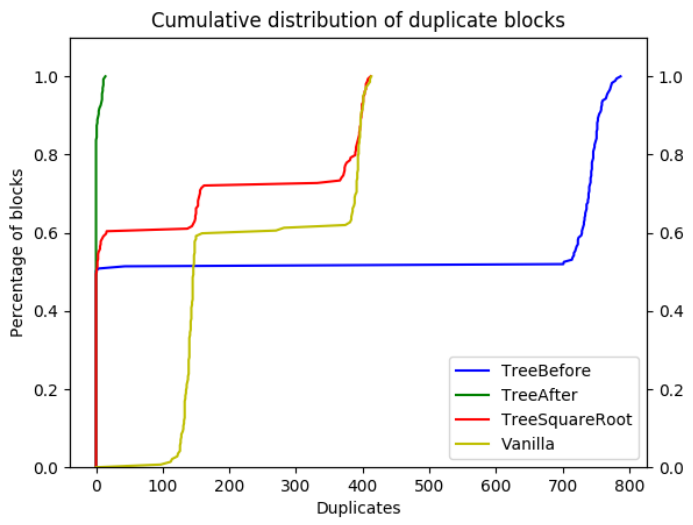


Figure 5.6: CDF of the duplicates that the network received per block.

Figure 5.8 shows a CDF of the latency that each block takes to reach all nodes in the different approaches. As the reader can see, the overall blocks take less time to reach all nodes in TreeBefore in comparison to the other methods. The TreeSquareRoot also has low latency, but about 15 % of blocks have a high latency surpassing TreeAfter. TreeAfter has a bit worse latency than Ethereum Vanilla and 30 % of blocks take less time to reach all nodes in comparison

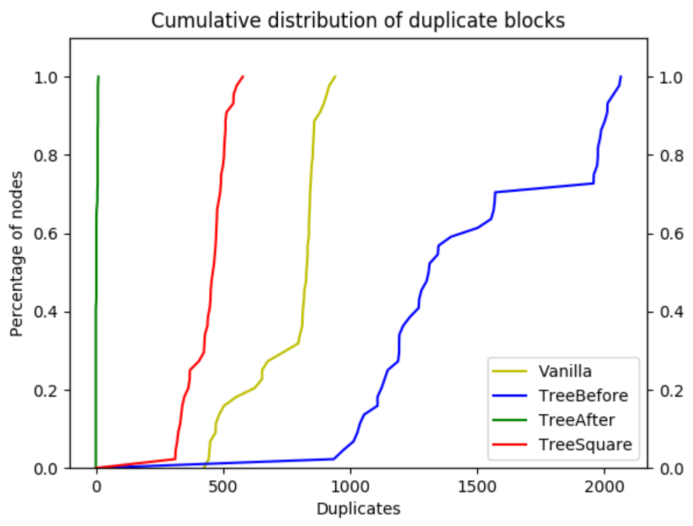


Figure 5.7: CDF of the duplicates that each node received.

with the Ethereum Vanilla.

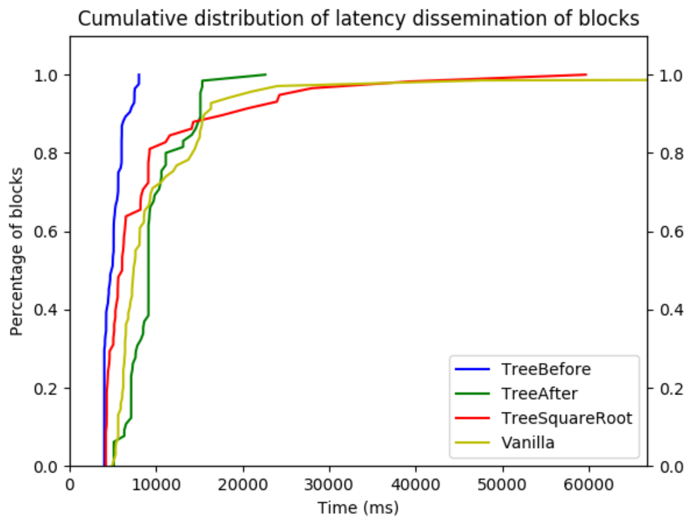


Figure 5.8: CDF of the time that each block takes to reach all nodes.

Figure 5.9 shows the number of blocks with a valid header but an invalid body that the network has received on average. As in TreeBefore, the nodes send the block to all of their neighbors as soon as they check the block's header, the number of invalid blocks in this approach will be huge. Consequently, the number of bytes that the network received is also much higher than the other approaches (see Figure 5.10). Conversely, in TreeAfter the number of invalid blocks will be minimal (the number of invalid blocks will be equal to the number of nodes that the byzantine node sent the block). This approach uses less bandwidth because the number of bytes that the network receives is fewer, as the reader can see in Figure 5.10. This happens because in this approach, correct nodes will not send invalid blocks, so there are fewer invalid blocks, and the blocks are disseminated following the tree structure. The TreeSquareRoot as it applies the same approach as Vanilla the network has received approximately the same number of invalid blocks. The number of bytes that the network received is less than Ethereum Vanilla because valid blocks in TreeSquareRoot are propagated without any duplicate if no parent has disconnected or failed (see Figure 5.10).

To measure the CPU utilized in each approach, we have done multiple runs. The CPU varied slightly in those runs, so we have done ten for each approach. Then we utilized the CPU used in each run and calculate the average CPU utilized for each approach. The results are presented in Figure 5.11, where the CPU used in TreeBefore is a bit bigger than the Ethereum Vanilla because, in TreeBefore, there are many invalid blocks. TreeAfter is the approach that used less CPU because there are less duplicates in the network.

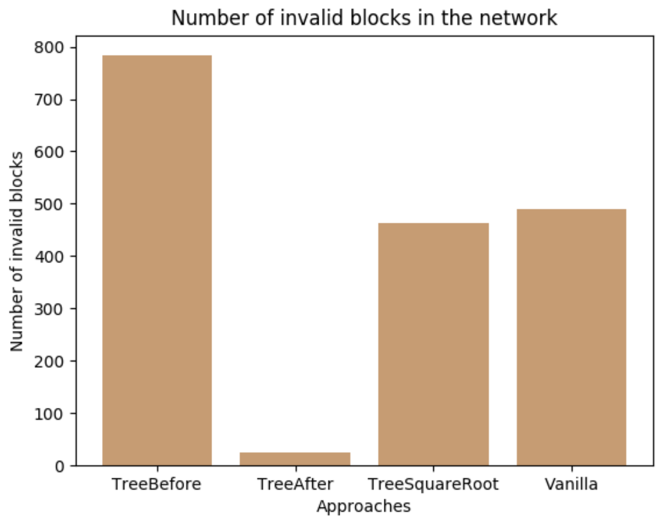


Figure 5.9: Number of invalid blocks in the network in each approach.

5.1.5 Summary

In the blocks' dissemination, if we privilege the speed of the propagation, the best solution is the TreeBefore approach. If we privilege the CPU usage, the number of duplicates in the network, and the bandwidth, the best solution is the TreeAfter approach. If the speed of the dissemination, the CPU used, the number of duplicates and the bandwidth are all important, and we want a solution better than the Ethereum Vanilla, the TreeSquareRoot approach is the better choice.

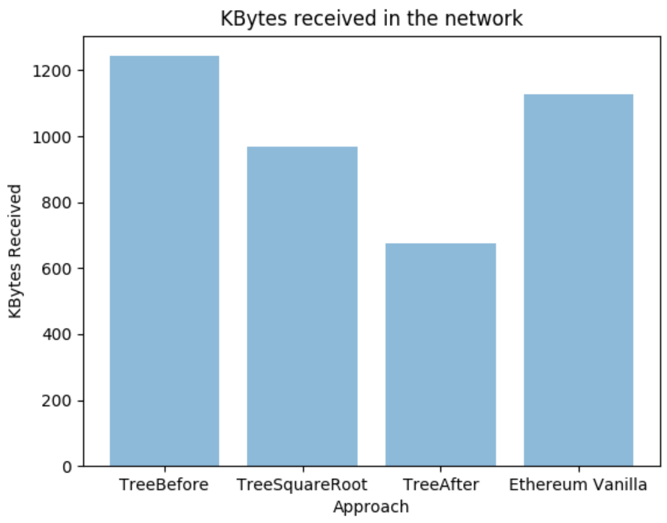


Figure 5.10: Bytes received in the network in each approach.

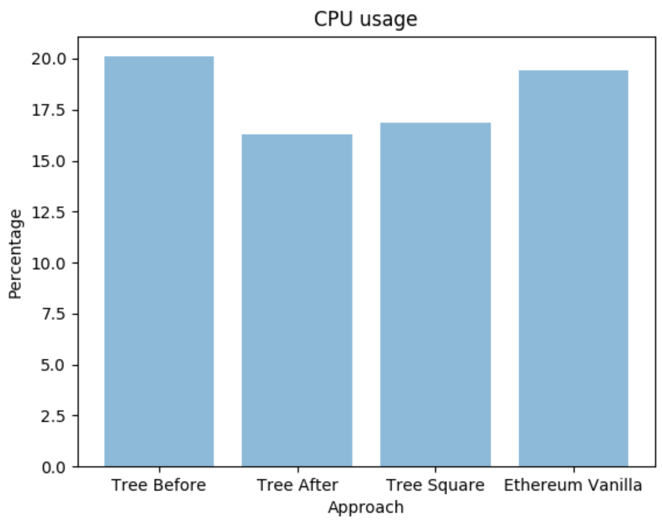


Figure 5.11: CPU used in each approach.

5.2 Dissemination of transactions

In this section, we evaluate our transaction approach, where we use the following metrics:

1. The number of duplicates in the network.

2. The latency that each transaction takes to reach all nodes.

3. The CPU utilized in each approach.

The same conditions in 5.1.1, and in 5.1.2 were applied in this approach.

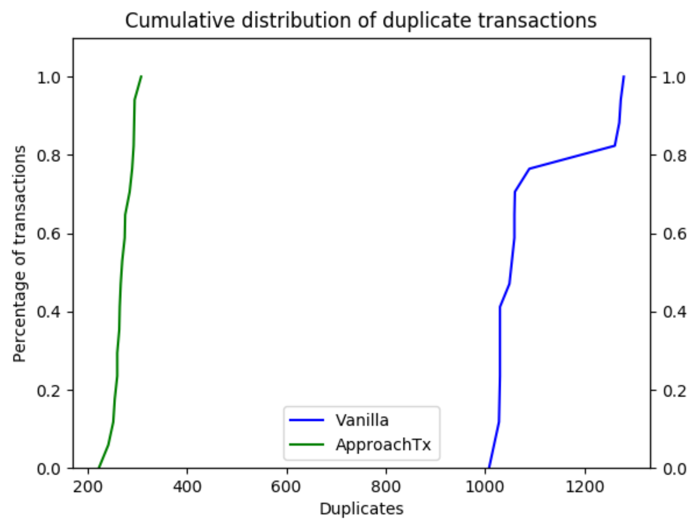


Figure 5.12: CDF of the number of duplicates that the network receives per transaction

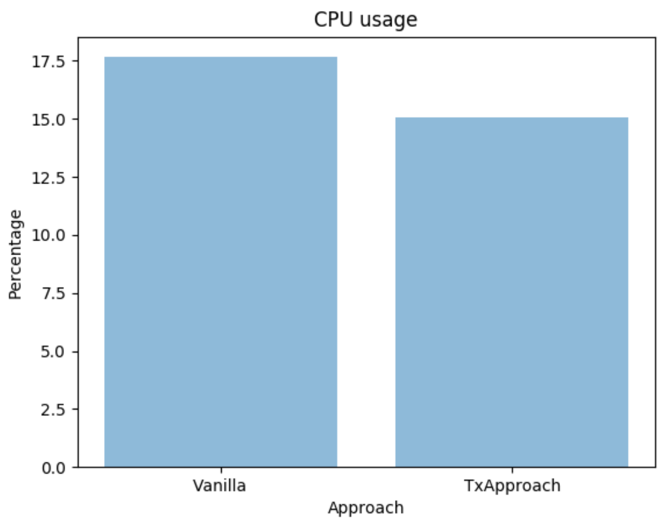


Figure 5.13: CPU used in each approach.

5.2.1 Experiments

Figure 5.12 shows a CDF of the number of duplicates that the network receives by transaction in the Ethereum Vanilla and in our approach. Analyzing the figure, the number of duplicates was reduced significantly. In our approach, the number of duplicates in the network for all transactions is under 300. In Ethereum Vanilla, there are transactions that the network received more than 1200. The CPU used in our approach is also reduced to approximately 3% (see Figure 5.13).

But although the number of duplicates was reduced the time that each transaction takes to reach all nodes is slightly bigger. Figure 5.14 shows a CDF of the time that each transaction

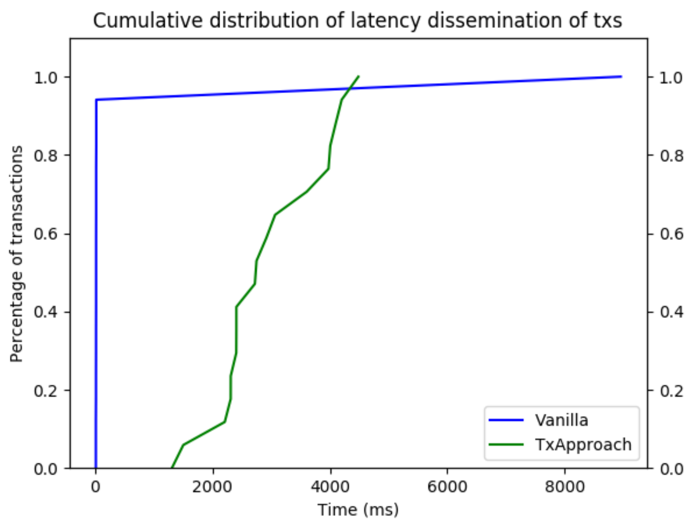


Figure 5.14: CDF of the time that each transaction takes to reach all nodes in the network

takes to reach all nodes. As the reader can see, the transactions in our approach take much time to reach all nodes where 80 % of transactions take about 4 seconds to reach all nodes. While in the Ethereum Vanilla for the same percentage of transactions, this number is less than 1 second.

5.2.2 Summary

In the transactions' dissemination, our solution improves the number of duplicates significantly, ending up to use less CPU than Ethereum Vanilla and also reduce the bandwidth consumption. Although the latency was worsened, the new transaction can always be introduced in the next block because the time that it takes to reach all nodes is at most 4 seconds, and the block insertion time into the blockchain is 10 seconds.

Chapter 6

Conclusions

Ethereum has some problems with the dissemination. Nodes receive a large number of duplicate blocks and transactions that affect the processing of each node, congest the network with redundant information because the bandwidth consumption is unnecessarily higher. Also, the blocks' dissemination can be improved. If the time that each block takes to reach all nodes in the network is shorter than the time that each node wastes in generating concurrent blocks that will not be included in the blockchain will be longer. With our approaches to the block dissemination, we improved the latency (time that each block takes to reach all nodes), and also the number of block duplicates in the network that affects the CPU used of each node and the bandwidth consumption. The results of our approach to disseminate blocks were partially presented in the Inforum conference [LAM19].

In the transaction dissemination, our approach reduced the number of duplicates, and consequently, the CPU used, and the bandwidth consumption in the network has improved. Nevertheless, the time that each transaction takes to reach all nodes was longer, but not too much. Since the block insertion time into the blockchain is about 10 seconds, and the latency in our approach is at most 4 seconds, the transaction can always be introduced in the next block.

6.1 Future Work

These tests were only made on the same machine, and we use the NEED tool to introduce latency in the network. Consequently, we have only run 45 nodes in the machine. So for future work, we desire to run these experiments in the cloud where we want to execute the Geth client in a considerable amount of nodes distributed across the globe. These will simulate the real environment more truly.

Additionally, we want to develop more solutions to the transaction dissemination and check the results. First, we want to try a different number of nodes to send the transaction directly and check the number where we obtain better results. Second, we want to implement the other possible transaction solution that we discussed previously, where each node checks if it is receiving something at the TCP level and verify if this solution has better results in comparison to our approach.

Furthermore, we want to propose these solutions to the Ethereum Community and try to them to be applied in the Geth's code.

Bibliography

[B$^+$02] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[BHO$^+$99] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[BMC$^+$15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 104–121. IEEE, 2015.

[BMZ18] LM Bach, B Mihaljevic, and M Zagar. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE, 2018.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[CV17] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[EGH$^+$03] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[EGKM04] Patrick T Eugster, Rachid Guerraoui, A-M Kermarrec, and Laurent Massoulié.

Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

[GKW$^+$16] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[JGKVS04] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 79–98. Springer, 2004.

[Kha] Olga Kharif. Cryptokitties mania overwhelms ethereum network's processing.

[KMM$^+$18] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference 2018*, pages 91–104. ACM, 2018.

[LAM19] João Barreto Luís Aguiar, Paulo Silva and Miguel Matos. Técnicas de redução de duplicados na criptomoeda ethereum. *Atas do Inforum 2019, "http://inforum.org.pt/INForum2019/docs/atas-do-inforum2019/"*, pages 205,2016, 2019.

[LCW$^+$06] Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204. USENIX Association, 2006.

[LPR07] Joao Leitao, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 419–429. IEEE, 2007.

[Mat13] Miguel Matos. Epidemic algorithms for large scale data dissemination. PhD thesis, Universidade do Minho, 2013.

[MCBG16] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. Attacking the network time protocol. In *NDSS*, 2016.

[MF10] Luis Rodrigues Mario Ferreira, Joao Leitao. Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. INESC-ID, 2010.

[MHG18] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *IACR Cryptology ePrint Archive*, 2018:236, 2018.

[MSF$^+$13] Miguel Matos, Valerio Schiavoni, Pascal Felber, Rui Oliveira, and Etienne Riviere. Lightweight, efficient, robust epidemic dissemination. *Journal of parallel and distributed computing*, 73(7):987–999, 2013.

[MVGV$^+$17] Aanchal Malhotra, Matthew Van Gundy, Mayank Varia, Haydn Kennedy, Jonathan Gardner, and Sharon Goldberg. The security of ntp's datagram protocol. In *International Conference on Financial Cryptography and Data Security*, pages 405–423. Springer, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[Nev18] João Neves. Container network topology modelling. Master Thesis, IST, 2018.

[NF18] Raiden Network-Fast. cheap, scalable token transfers for ethereum, 2018.

[NG16] Christopher Natoli and Vincent Gramoli. The blockchain anomaly. *arXiv preprint arXiv:1605.05438*, 2016.

[OM14] Karl J O'Dwyer and David Malone. Bitcoin mining and its energy footprint. 2014.

[SCG$^+$14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

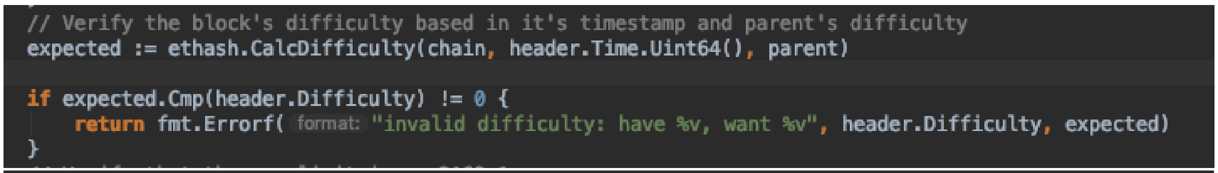
[SZ15] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[VGVS05] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217, 2005.

[WGP$^+$17] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*, pages 64–73. IEEE, 2017.

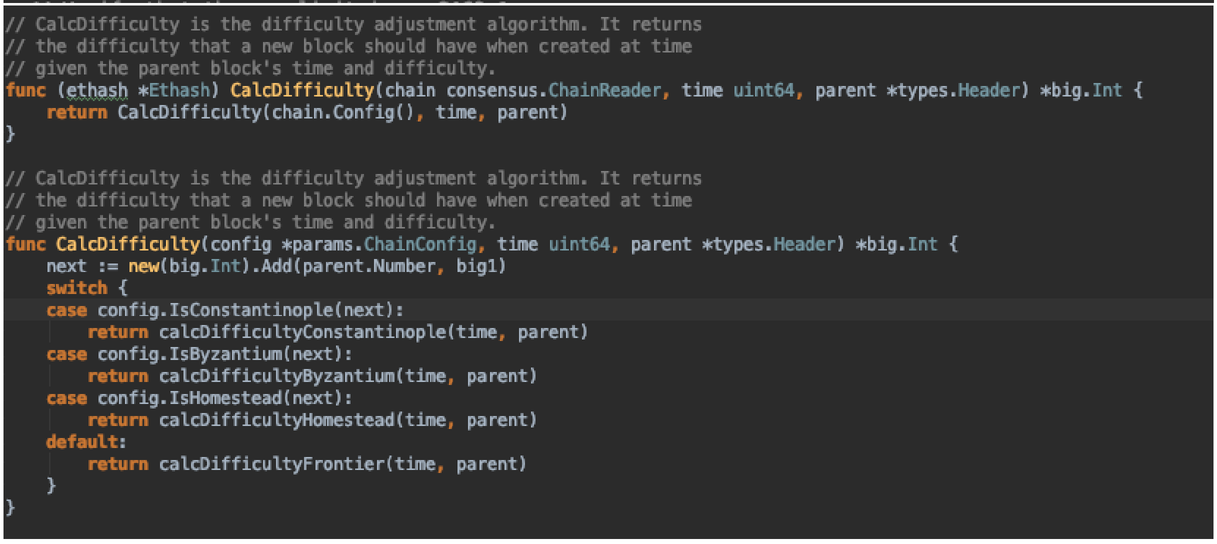
Appendix A

Ethereum code



```
// Verify the block's difficulty based in it's timestamp and parent's difficulty
expected := ethash.CalcDifficulty(chain, header.Time.Uint64(), parent)

if expected.Cmp(header.Difficulty) != 0 {
    return fmt.Errorf( format: "invalid difficulty: have %v, want %v", header.Difficulty, expected)
}
```



```
// CalcDifficulty is the difficulty adjustment algorithm. It returns
// the difficulty that a new block should have when created at time
// given the parent block's time and difficulty.
func (ethash *Ethash) CalcDifficulty(chain consensus.ChainReader, time uint64, parent *types.Header) *big.Int {
    return CalcDifficulty(chain.Config(), time, parent)
}

// CalcDifficulty is the difficulty adjustment algorithm. It returns
// the difficulty that a new block should have when created at time
// given the parent block's time and difficulty.
func CalcDifficulty(config *params.ChainConfig, time uint64, parent *types.Header) *big.Int {
    next := new(big.Int).Add(parent.Number, big1)
    switch {
    case config.IsConstantinople(next):
        return calcDifficultyConstantinople(time, parent)
    case config.IsByzantium(next):
        return calcDifficultyByzantium(time, parent)
    case config.IsHomestead(next):
        return calcDifficultyHomestead(time, parent)
    default:
        return calcDifficultyFrontier(time, parent)
    }
}
```

Figure A.1: A slice of Geth's code (version 1.8.17-stable) showing that uncles do not count to the total difficulty. When a node verifies if the difficulty of the header is correct it only uses the chain, the timestamp of the header and the parent