Behavior characterization in cryptocurrency networks

David Vavřička
Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—This work explores the network protocol of Ethereum, which is one of the most prominent and innovative blockchain-based platforms, by measuring various network metrics and characteristics. The goal of our measurements is to identify the network's behavior, find out the weak points, and enable so several improvements to the existing system.**

We deploy a set of geographically distributed nodes based on an open-source implementation of the Ethereum protocol. We modify the nodes to listen to the network traffic on the mainnet of the protocol and perform one-month long measurement.

To the best of our knowledge, we are the first to measure the propagation delay of blocks and transactions between geographically dispersed Ethereum peers.

On top of that, we replicate several measurements from 2017 and show how the Ethereum network changed and evolved since then. In the end, we provide possible suggestions about how to improve the protocol.

I. Introduction

Over recent years, decentralized cryptocurrencies enjoyed a rapid growth and gathered remarkable interest no longer just from technological enthusiasts, but from the general public as well. This is due to their potential to revolutionize the current payment methods with the ambition to eliminate the need for banks or other centralized entities – acting as trusted mediators of financial operations – by guaranteeing honest and proper execution of transactions inherently by the technology itself.

In this work, we are interested in measuring and evaluating the network characteristics of the currently second most valuable cryptocurrency Ethereum.

Ethereum opearates on an underlying peer-to-peer network protocol which must support the interconnection of a large number of peers spread all over the world, prevent the creation of network partitions, and allow for a smooth transfer of all the data needed in the system.

We observe that the the Ethereum's network is poorly studied. We are missing many metrics describing the network behavior, most importantly about the information propagation between individual peers. Such a study could enable find possible, and until now hidden, pitfalls and shortcomings of the protocol, and to observe possible aspects to be optimized.

For Bitcoin, which has been out for longer, there exist many more analyses and studies. In the network behavior context, there is one highly cited paper [1] measuring, among other metrics, the propagation delay of blocks between geographically dispersed nodes in the Bitcoin network. The study concludes that the propagation delay of blocks is the primary cause of blockchain forks. As far as we are aware, no one had measured this metric in the Ethereum network before. We decided to measure this metric – among many others – in our work in order to see how things differ in Ethereum, which follows a proof-of-work based consensus algorithm similar to Bitcoin's.

Previous work [2] from 2017 studied various properties of the Ethereum platform, such as what impact the user-defined variables "gas price" and "gas limit" have on transaction inclusion, the commit time of transactions or the ratio of blockchain forks. We replicate and verify their measurements in order to observe how the properties changed over the last two years. Moreover, we take attention at details, that the authors of [2] omitted, such as the relation between fork lengths and the probability of being referenced in main blocks. On top of that, we measure various metrics regarding the forked blocks, e.g. the degree of transaction reordering, and add several other metrics, such as the degree of redundant receptions of new block announcements, with the aim to observe the network traffic overhead.

Since the measurements performed in [2] were captured all by one statical monitoring node, our novelty lies in deploying a set of geographically dispersed monitoring nodes to examine whether there is a correlation between the metrics captured and the geographical position of particular nodes.

In order to perform the measurements, we develop a tool based on an already existing open-source implementation of Ethereum client which is able to connect to the main network and collect the desired data. We deploy four these measuring instances that we place in North America, Western Asia, Western Europe and Central Europe.

A. Document Structure

The rest of this document consists of the following sections:

- Section II provides the technological background of Ethereum.
- Section III presents the measurement infrastructure.
- Section IV evaluates the results.
- Section V describes our Ethereum protocol improvement suggestion.
- Section VI discusses the final achievements and finishes this document.

II. Ethereum Network Plane

The Ethereum network communication follows a peer-to-peer (P2P) approach and takes place over TCP with the only exception of UDP-based node discovery mechanism.

A. Network Protocols

The P2P communication between Ethereum clients is governed by the underlying DEVp2p Wire Protocol and RLPx cryptographic protocol suite providing a general-purpose transport interface. The RLPx allows for node discovery, transport of messages encrypted with AES256 or packets framing to support multiplexing multiple protocols over a single connection.

This protocol provides a standardized API but the general behavior of peers on the network and their synchronizing strategies are implementation dependent.

B. Ethereum Clients

In general, we distinguish between two basic types of peers: full nodes and light ones. Full nodes possess the entire blockchain, verify the validity of all transactions and contracts and answer queries from other nodes – light nodes, or not yet synchronized full nodes – about the current state of the blockchain.

Light nodes, as the name indicates, are less memory and computationally demanding. They do not verify every block and usually do not possess a copy of the current blockchain state and therefore rely on full nodes to provide the missing information.

There is also a third, special kind of peers – bootstrap nodes. These are highly available nodes run by volunteers whose purpose is to facilitate the connection of new clients by providing them "enode identifiers" of other nodes in the network. The enode datatype is represented by a unique hexadecimal identifier concatenated with an IP address and a TCP port.

C. Information Propagation

The efficiency of the Ethereum platform depends on the time it takes for a transaction to propagate from users to miners and the time it takes for a transaction to get committed.

In the following paragraphs, we describe the approach used for the propagation of blocks and transactions. We will also discuss transaction committing.

D. Transaction Lifecycle

When an external account decides to create a new transaction, it broadcasts the transaction to all neighboring peers it knows about. On average, the Geth implementation maintains connections with 25 other nodes, whereas Parity connects to 50 peers. Upon a transaction reception, the nodes forward the transaction to their neighbors – usually just to a limited number of them, but that is implementation dependent – therefore the transaction propagates through the network until it reaches a miner. According to [3] Geth propagates transactions to all its neighbors and Parity forwards transactions just to square root of its neighbors, i.e. to 25 and 5 peers respectively. A miner includes the transaction into a new block only if: its parameters are valid, the funds are sufficient, the signatures match and the miner finds the offered gas to be profitable for them.

A transaction is considered to be committed after it appears in a block included in the main chain, with at least 12 successors. It is important to mention that transactions may never commit. Firstly, miners can for any reason refuse to include a transaction in a block. Secondly, even after getting included in a block, that block may not become part of the main blockchain. In both cases, the user may still try to broadcast the transaction again. One can be sure that a transaction will never commit when another transaction from the same account with the same nonce gets committed.

E. Block Propagation

Wire protocol – also known as the Ethereum subprotocol – defines three ways of block propagation:

- Several whole blocks can be propagated in chunks. This method is used only when the peer is synchronizing with the network and requests a specific set of blocks.
- Individual blocks can be propagated as a whole (header + body). This is the most common way of block propagation.
- A block's hash can be propagated separately without the burden of the block's body. Peers that receive this announcement, request the whole block data only if they do not possess the attached block yet. The advantage of this method lies in faster propagation due to smaller size of message.

Individual peers after receiving a new block perform various verifications such as whether the block's nonce leads into some specific mixHash. Then they verify and execute individual transactions and synchronize so with the Ethereum platform.

With the exception of the first type of message, peers upon new block reception forward the message to their neighbors.

III. Implementation and Deployment

In order to perform the measurements, we develop a tool based on an already existing open-source implementation of Ethereum client which is able to connect to the main network and collect the desired data. We deploy four these measuring instances that we place in North America, Western Asia, Western Europe and Central Europe.

All instances run the same software configuration which essentially consists of the GNU/Linux operating system, our modified Geth Ethereum client and a specially configured instance of the Rsyslog logging daemon that we present below.

A. Ethereum Client

Our Ethereum measurement peer is based on the Geth open-source Ethereum client implementation in its latest version – which is $1.8.23$[1]. We made no changes to the program's logic; the only modifications represent the addition of several syslog calls that capture all received transactions, blocks and invalid network messages as well as information about connected peers. We also prepend each of these events with a local timestamp at the exact moment of the message arrival, before

[1] https://github.com/ethereum/go-ethereum/releases/tag/v1.8.23

any message validation is done. All these changes accounted to slightly less than 1 000 of lines of code written in the Go language [2].

The client runs with the default settings except for two parameters. Firstly, we allow to connect to unlimited number of peers instead of the default 25. Secondly, we set the minimum gas price of incoming transactions to zero in order not to discard underpriced transactions. This is because we want to observe all transactions on the network for the needs of the metric measuring propagation delays of transactions. These settings are identical with the measurement client configuration used in [2], allowing us to compare our results with theirs.

B. Rsyslog

The Rsyslog was configured in order to receive all syslog calls from the Geth client and to pass them into separate log files for each message type. Next, the Rsyslog performed automatic chunking of logs to 500 MB pieces as well as an automatic zipping. That was necessary because of the great number of propagated messages. Finally, it was necessary to increase the default maximal size of syslog messages from default 8 kB to 100 kB, because we have observed messages containing blocks as big as 30 kB.

IV. Measurements and Results

We were listening to the network traffic of the Ethereum main network using the infrastructure of highly connected peers from April 1st to May 2nd in 2019, and collected information about 216 656 blocks (including forks) with the block numbers ranging from 7 479 573 to 7 680 658.

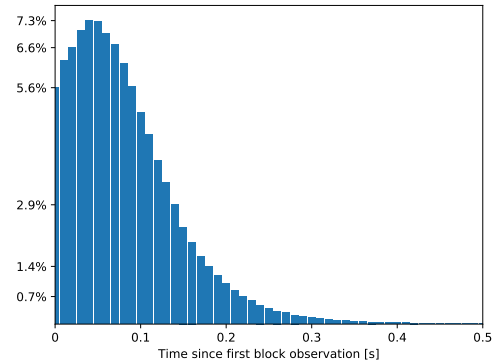
A. Information Propagation

In this metric we measure the propagation delays of blocks and transactions. To achieve that, we capture their arrival times on each measurement node from our infrastructure with synchronized clocks. The delay is then the difference between the first observation of the message and the times of arrival on the remaining peers. Finally, we accumulate all the delays and calculate their distribution.

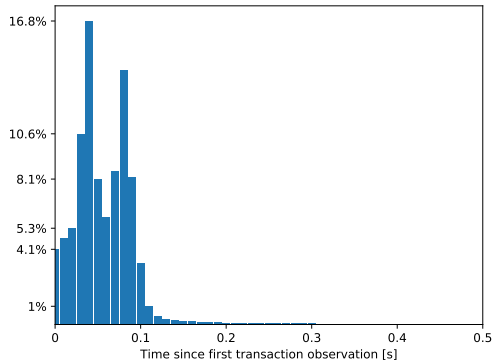
Figure 1 reveals considerably low propagation delays. The median time until a node receives a transaction is just 60 miliseconds (ms). The propagation delay of the 90 % fastest transactions is under 111 ms. Finally, it takes 0.8 seconds for 95 % and 38.5 seconds for 98 % of transactions to propagate through network. These slow transactions from the approximately 2 % tail may be problematic, e.g if by any chance there were some further transactions that depend on their execution.

Blocks, in spite of their substantially greater size, propagate only slightly slower. Their median propagation delay is 74 ms and the mean is at 109 ms. In total 95 % of blocks make it below 211 ms through the network and 99 % under 317 ms.

Back in 2013, the authors of [1] measured the propagation delays of blocks in the Bitcoin network following exactly the same technique. Their results showed that in Bitcoin, the median time until a node receives a block is 6.5 seconds and the mean is 12.5 seconds. Even after 40 seconds, still 5 % of nodes do not know about the newly propagated block. We are aware, however, that the Bitcoin's network progressed since then and so we cannot compare our results directly with [1].



(a) The histogram of times since the first block announcement.



(b) The histogram of times since the first transaction announcement.

Figure 1: Information Propagation

Our measurement reveals that information in Ethereum propagates fast, and surely is not the main cause of forks, as was the conclusion of [1] for Bitcoin. Due to the measured results it appears that miners continue mining blocks even after discovering newer blocks. They can afford that thanks to the uncle rewards, which are almost guaranteed for forks of length one, as we show in our measurement from Section IV-B.

B. Prevalence of Forks

This metric measures the total number of blockchain forks, where we distinguish between forks of lengths one, two and possibly longer ones. We also calculate the proportion of observed forked blocks and the total number of blocks.

Out of the 216 671 blocks that we captured during our one-month long measurement, 92.81 % of them became part of the main chain, 6.97 % became uncles referenced by some block from the main chain, and only 0.22 % of blocks became "so-called" unrecognized uncles. The very small number of not

[2]https://golang.org

recognized forks leads to the greater chance for even weak miners to get rewarded.

The result is presented on Table I.

Fork Length	Total	Recognized	Unrecognized
1	15 171	15 100	71
2	404	0	404
3	10	0	10

Table I: Fork lengths.

We have not observed any fork longer than 3. The interesting observation is that more than 99 % of forks of length one were later recognized. All longer forks than one were not recognized.

In 2017, the same measurement was performed in [2]. Since then, the proportion of uncle blocks increased by more than one percent and their lengths increased as well. We can only speculate what is the real cause of that. One explanation could be that the number of separate miners increased, which generally leads to the greater chance that two miners will propagate at the same time and create forks. In addition, back then the inter-block time, which is the mean time between two succeeding blocks,[3] was one second higher which also could had contributed.

C. Empty Blocks

We have observed a significant number of blocks that were empty – i.e. with no transactions. The miners of empty blocks do not waste time validating and including transactions and so can propagate them slightly faster at the cost of not getting paid for the transaction fees.

These blocks, however, are harmful for the network, because they increase commit time of transactions. This is because the transactions that could have been included in an empty block must wait for being included in next block. If an increased number of miners switched to this strategy, it would be disastrous for the platform.

We were in interested how many blocks of this type are in the network, and secondly, whether the proportion changes if we look at main and uncle blocks separately.

The results revealed 2 921 empty main blocks (1.453 %) out of 201 086 all main blocks, and 185 empty forked blocks (1.187 %) out of 15 585 all forked blocks.

We observe, that empty blocks indeed become the part of main chain more often. The second observation is, that their number (1,43 % of all blocks) is not insignificant and is already prolonging commit times of transactions.

D. Transaction Commit Time

This metric measures the distribution of the time it takes for transaction inclusion which is the difference between time the transaction was first observed at a node to the time it was included in a block. Then we analyze the commit times for various numbers of subsequent confirmation blocks.

[3]https://etherscan.io/chart/blocktime

This exact measurement had been performed in [2] where the authors had chosen to measure the commit times for 3, 12 and 36 confirmation. We follow them in order to be able to compare our results with theirs.

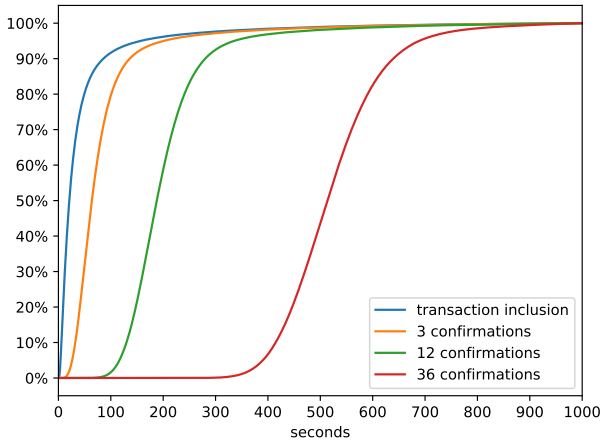


Figure 2: Time for transaction inclusion and commit (3, 12 and 36 confirmations).

Visually, we achieve the same results as in [2], that is the curves representing commit times follow similar pattern. Moreover, on Figure 2 we observe that higher number of confirmations leads to slightly less steeper curves. This indicates the increased fraction of transactions that must wait longer to be committed.

If we look in more detail at the commit times, we observe that the median waiting time for '12-block commit' is 189 seconds whilst the authors of [2] measured 200 seconds. This is, however, because of the inter-block time, which decreased[4] from 14,3 s to the current 13,3 s.

E. Transaction Reordering

In this measurement we compare commit times of "in-order" and "out-of-order" transactions. The sender stamps every transaction with a nonce, which is incremented by 1 for each new transaction. We say that two transactions assembled by the same sender were received out of order, when we first capture the transaction with higher nonce.

This is a replicated measurement from [2] that was executed in April 2017. The authors performed the measurement on a one-day sample of logs and observed that 6.18 % out of all committed transactions during that period were received with out-of-order nonces.

We observe 11.54 % out-of-order committed transactions out of all committed transactions during our one-month long measurement. We observe, however, that the proportion of out-of-order transactions fluctuates from day to day with 4 % deviation from our one-month average. As well as the authors of [2], we do not know the exact cause. We hypothesize,

[4]https://etherscan.io/chart/blocktime

that there are more independent factors, such as the varying motives of Ethereum users, or the current overall network connectivity.

Our measurement reveals the prolonged commit time of out-of-order transaction. It takes 192 seconds for 50 % and 325 seconds for 90 % of out-of-order transactions to commit. For comparison, the mean time for in-order received transactions is 189 seconds and 90 % of these transactions need 292 seconds for committing. The results indicate that out-of-order reception negatively affects commit times. It is not surprising because such transactions must wait for their delayed predecessors before committing.

F. Gas Used per Transaction Type

This measurement investigates the amount of gas consumed per transaction type. In April 2017, the same measurement was performed in [2]. For this reason, we are able to comment about how the average gas used per transaction changed since then.

We distinguish between three types of transactions:

- regular value transfers
- contract function calls
- contract creations

The differentiation is done as follows: We consider a message to serve as a contract creation if the address of the recipient is empty; as defined in the Ethereum specification [4]. There is, however, no consensus about how to distinguish between regular value transfers and contract function calls. In [2], the authors simply say that the messages that are not contract creations, and consume 21 GWei of gas (which is the base cost of transaction), are regular transfers. The remaining messages are function calls.

We consider messages that are not contract creations and which have data field empty as regular transfers, and those transaction with some data (with a message for a contract) as message calls.

1) Results: Out of 21 960 051 captured transactions, we calculated this metric from the subset consisting of 20 654 578 valid transactions included in main blocks. There were 8 917 902 regular value transfers, 11 642 182 message calls and 94 494 contract creations.

Transaction Type	# Transactions	Median	90th percentile
value transfer	8 917 902	25.2	100
function call	11 642 182	100	1 000
contract creation	94 494	1 065	4 000

Table II: Gas Used (in GWei) per Transaction Type.

As can be seen in detail on Table II or on Figure 3, the contract creations use the most gas; this is because their code consists of computationally expensive routines. The median gas used of this type of messages increased since 2017 two-fold to 1 065 Gwei, implicating that developers deploy more complex and more computationally expensive contracts. Five percent of contract creation messages consume more than 4 500 GWei, which is more than the gas limit per block back in 2017. The current block gas limit is at 8 000 GWei while in 2017 it was only at 4 000 GWei.

The median gas used per function call is 100 GWei and 25.2 Gwei in the case of regular transfer. These are relatively small numbers leading to reasonable costs of transaction fees.

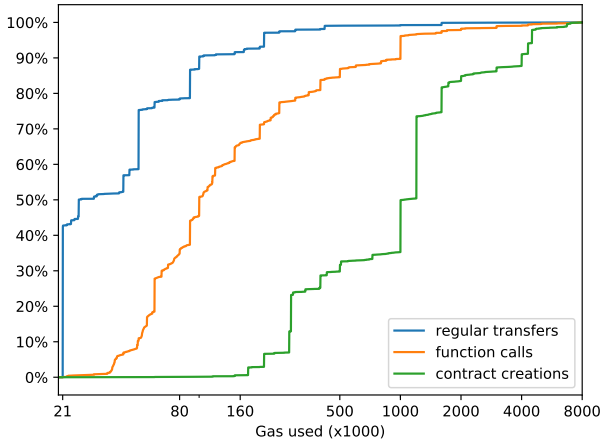


Figure 3: Distribution of gas usage for different types of transactions.

G. Impact of Gas Price

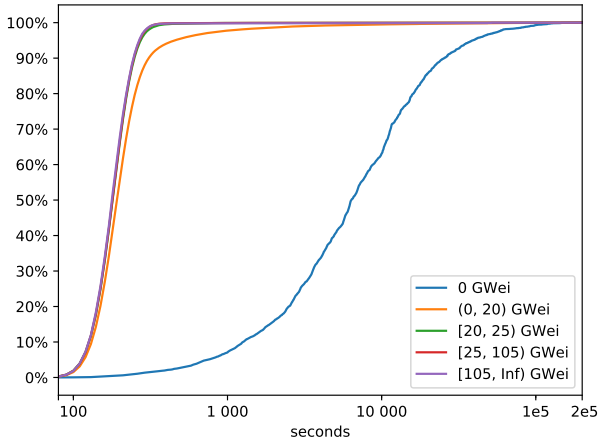


Figure 4: Commit delay (sec) for transaction based on gas price.

This is a replicated metric from [2] measured in 2017.

We divide all observed transactions into five disjoint sets according to their gas price as follows: [0, 0], (0, 20), [20, 25), [25, 105), [105, $+\infty$) Gwei. We achieve exactly the same pattern as the original authors two years earlier, but we cannot compare the exact mean commit times because their measurement lasted 2 months longer.

We observe that transactions with zero fee commit very late (in matter of hours) as they are unattractive for miners. The

transactions worth up to 20 Gwei commit significantly later than any transaction beyond this threshold. It shows, that it does not pay out setting gas price higher than 25 Gwei.

H. Never-Committing Transactions

Never committing transactions are those transactions which will certainly never commit. That is because there already exists another committed transaction with the same nonce which originated from the same account. These transactions are useless and only flood the network, so we were interested how many of them are there.

Out of 21 241 139 unique observed valid transactions we find that 20 654 578 (97.2386 %) of them are already committed, 63 446 (0.2987 %) are not committed but still may commit, and 523 115 (2.4627 %) will never commit.

The transactions labeled as "may commit" are those transactions that had not committed yet, and in addition do not satisfy the condition for being "never-committing".

I. Ethereum-Classic Network Traffic

During the one-month long measurement, we observed on each of our machines between 10 388 to 24 798 of block header messages that failed due to the DAO check – confirming so the presence of Ethereum Classic peers, that was revealed in [3].

J. Message Reception Redundancy

We were interested what is the actual number of redundant messages that the most popular client, which is Geth, with defaults settings receives. Thus, we deployed a node, that was connected to the default number of peers (25), and listened to the network between May 2nd and May 9th in 2019. For comparison, we performed the test on a highly connected Geth client which was always connected at least to 150 peers, and on average to 200 peers.

The results for Geth with 25 peers showed that the average number of redundant receptions was 9.11 with median at 9 receptions. 10 % of the most propagated blocks were received at least 12 times, and 1 % was received 15 times.

The results for Geth with 200 peers showed that the average number of redundant receptions was 73.59 with median at 75 receptions. 10 % of the most propagated blocks were received at least 109 times, and 1 % was received 139 times.

K. Discussion

How to decide, whether the measured numbers of redundant receptions are too low, too high or optimal?

The authors of the highly cited paper [5] suggest that in networks with failures, it is enough for the epidemic broadcasting strategies to disseminate information to a logarithmic number of neighbors with respect to the total number of peers.

According to the latest estimation from [3], there are around 15 000 peers, so the propagation strategies should propagate to at least ln(15 000) of peers - which equates to $\sim$ 10 peers.

Our results for Geth with default number of peers reveal the median and mean number of redundant block message receptions to be 9 in the former, and 9.11 in the latter case.

V. Ethereum Improvement Directions

As a reaction to the observation of the Ethereum-classic traffic in the non-classic network, we propose an easily implementable improvement to the Ethereum client implementations that would substantially decrease the amount of network traffic and increase performance of nodes.

Both platforms run on the DEVp2p underlying network and share the same network ID as well as the genesis hash. This is because they used to act as one network, before they forked back in July 2015. The problem of current client implementations, as is explained in [3], is that they maintain connection with Ethereum-classic peers and thus receive from them useless data.

We observe, that upon each new connection, the peers exchange a set of messages, including the information about last block headers out of which it is possible to identify Ethereum-classic peers. We suggest to exploit that and disconnect from those peers.

We propose to add the three following methods to the logic of all Ethereum client implementations:

1) Immediate disconnect after observing the "DAO fork" in the "BlockHeaders" message exchanged after a connection.
2) Maintain a local persistent table of "enode" identifiers of Ethereum-classic peers and add there the peers from Step 1.
3) Before connecting to new peers, first verify whether they do not belong to the table of Ethereum-classic peers.

The only weakness that we see is that this strategy would not disconnect from not yet synchronized Ethereum-classic peers whose latest block is before the DAO fork.

VI. Conclusion

A. Achievements

We have measured a wide range of Ethereum's network properties that may have the potential of directing the aim of future improvements of the network protocol of Ethereum as well as similar platforms.

We measured the average delay between the times of the newly mined blocks receptions on our infrastructure consisting of four geographically distant nodes to be on average at 109 ms. This indicates the very good connectivity of the platform.

By replicating measurements performed in 2017, we were able to get insights on how the network's behavior changed since then. The major differences since then are: increased number of forks, slightly increased transaction commit times and twofold increase of the average computational expensiveness of deployed smart contracts.

Our measurement nodes captured the presence of peers from the Ethereum Classic network which are not only useless, but cause significant network overhead. In Section V we propose a possible solution for this problem.

B. Future Work

Regarding possible future work, we present two suggestions:

- Our measurements were performed on the Geth client implementation which is only one out of many types. We advocate to replicate the measurements capturing network properties, such as those measuring redundant message receptions on other client types to observe if their behavior differs. In addition to this, it would be worthwhile to make a comparative study of the most prominent client implementations. The Ethereum users would appreciate knowing which implementation is the most efficient as well as the developers of the other implementations could get inspired what to improve.
- We suggest to perform our reception redundancy measurement with the aim at transactions instead of blocks. Both types of messages follow slightly different propagation strategies and their properties, such as their average size, are very different.

REFERENCES

[1] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference*, September 2013, pp. 1–10.

[2] I. Weber, V. Gramoli, A. Ponomarev, M. Staples, R. Holz, A. B. Tran, and P. Rimba, "On availability for blockchain-based systems," in *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium*. IEEE, pp. 64–73.

[3] S. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, "Measuring ethereum network peers," in *IMC '18 Proceedings of the Internet Measurement Conference 2018*.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[5] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "From epidemics to distributed computing," in *IEEE Computer*.