UNIVERSITY OF BELGRADE

SCHOOL OF ELECTRICAL ENGINEERING


# FORMAL AND SIMULATION ANALYSIS OF DATA DISSEMINATION ALGORITHMS IN A BLOCKCHAIN NETWORK

Senior thesis

Mentor:

prof. dr Jelica Protic

Candidate:

Luka Miletic 2014/0036

Belgrade, September 2018.

# TABLE OF CONTENTS

# 1. INTRODUCTION

This paper will present a formal and simulation analysis of data dissemination algorithms while using the Tendermint technology [1] as an example. Tendermint is a type of blockchain scheme. Blockchain schemes provide decentralization alongside irrefutability. Algorithm analysis is tightly connected with the Mempool component of Tendermint. This component is currently implemented in a way that it guarantees a certain outcome, yet it produces a large overhead. It relies on a P2P (*Peer to peer*) communication between servers in a distributed network. Herein, a formal and simulation analysis of several algorithms will be presented, with the goal of solving the overhead problem that exists in the current Mempool implementation in Tendermint.

Generally, there are three different approaches when it comes to dissemination algorithms. First one is based on client having the main role when disseminating data. Second one is based on performing a classification of the servers in the network into structured groups. Third one relies on connecting the servers in a completely dynamic manner, thus forming a random network graph. Algorithms explained in this thesis are based on the three aforesaid approaches.

In chapter 2., principles regarding Tendermint which are relevant to this paper will be explained. Following, definitions of concepts that are important for better understanding of the problem and proposed solutions will be provided. In this chapter a system model will be defined. Algorithms analyzed in this paper are based on that model.

In chapter 3., a formal analysis of the dissemination algorithm currently used in Tendermint will be presented, alongside three additional algorithms. For every algorithm, a performance analysis will be provided, as well as discussion of the validity of that algorithm in a network which is unreliable. In chapter 4., simulation analysis will be presented. The goal of this analysis is to point out the drawbacks of the current algorithm, as well as the advantages of one of the algorithms provided in chapter 3. It will be shown that the proposed algorithm is way more suitable than the algorithm currently used. Due to that, it may be realistically implemented within the Tendermint network. This paper concludes in chapter 5. Conclusion will provide a summary of everything that

has been done, the contributions of this thesis and will present the ideas concerning possible future work.

# 2. DATA DISSEMINATION IN A BLOCKCHAIN NETWORK

## 2.1. About Tendermint

Blockchain is a data storage model based on decentralization and indisputability. In this model, data is distributed on a large number of servers. This eliminates the SPOF (*Single Point Of Failure*) problem. Furthermore, data is completely resilient to unwanted modification by malicious third parties.

Blockchain has gained in popularity in recent years, primarily due to its usage within the *Bitcoin* transaction system [2]. *Bitcoin* is a cryptocurrency which relies on distributing information about clients' transactions into a vast number of blockchains. In *Bitcoin*, different cryptography mechanisms are used, so as to provide confidentiality. In its essence, blockchain is a linked list of blocks that contains clients' transactions [3]. Links between blocks are modeled via cryptographic hash functions. Due to the fact that this linked list is replicated on a large number of servers within a distributed network, there must exist a defined protocol which enables all the servers in the network to make a unanimous decision, reach a consensus, and decide what the next block in the blockchain will be.

It is evident that the blockchain has become increasingly popular due to Bitcoin. However, that concept can be applied in a more general context. Blockchain does not have to be used for financial transactions only. It may also be used for any transaction which holds a certain set of instructions [4].

The concept where a transaction can be pretty much anything is used in Tendermint. Detailed specification of Tendermint can be found at [16]. Tendermint is a fault-tolerant blockchain [5]. Its consensus protocol, which does not require "mining" as the consensus protocol used in Bitcoin, is used to order transactions in a blockchain in the presence of malicious third parties (Byzantine generals problem [6]).

Consensus protocol is one of the two main components of Tendermint. In this paper, consensus algorithm will mainly be considered a black box, taking into consideration the fact that all the details behind its implementation are irrelevant to the analysis of data dissemination

algorithms. Only those concepts of the consensus algorithm which are tightly connected to the data dissemination algorithm will be explained.

Second main component of Tendermint is the Mempool. Mempool is used for storing clients' transactions that are still not a part of the blockchain, as well as for disseminating those transactions throughout the network. It may be stated that the Mempool contains all those transactions for which a consensus is yet to be reached. Mempool component represents the input of the consensus algorithm. The output of the consensus algorithm is a block of transactions to be added to the blockchain.

Therefore, Tendermint is presented by a distributed network of servers where each server has its own blockchain replica. Apart from the fact that the blockchain stores information about clients' transactions, those transactions are also executed whenever they are added to the blockchain. From the client's perspective, it is important that the transaction becomes a part of the blockchain, and that it gets executed at some point in the future. Tendermint was at first a private blockchain, but is now building towards becoming a public blockchain. Tendermint's public blockchain is called Cosmos [17].

## 2.2. Definitions

In this chapter, before all else, definitions of concepts that are relevant to the data dissemination algorithms analysis in the Tendermint's Mempool, are given. Following, a system model is defined, in a way that it is easily understandable, yet complex enough to enable proper analysis of the aforesaid algorithms.

### *2.2.1. Deterministic state machine replication*

Let us assume a standard client-server architecture, as shown in Figure 2.2.1. Server provides service for a client. In order for a client to make use of that service, it has to send a request, and then wait for the response.

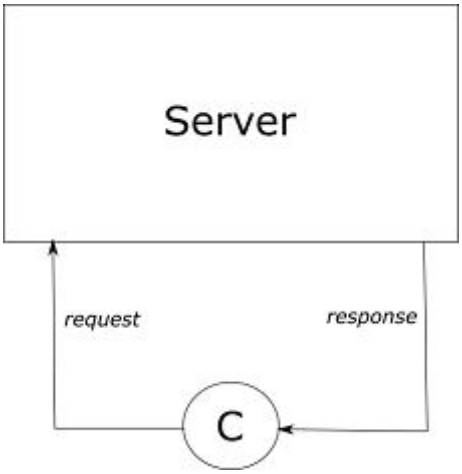


**Figure 2.2.1 – Standard client-server architecture**

- ***C*** – client

Although this is the simplest approach in implementing a client-server service, its main disadvantage is in having only one server. In case that server becomes faulty, the whole service will be unavailable to a client. That one server may be considered as a SPOF (*Single Point Of Failure*) part of the system. For that reason, we will consider a more advanced and reliable approach in implementing the client-server architecture, shown in Figure 2.2.2. This approach is based on the deterministic state machine replication approach - SMR (*State Machine Replication*).

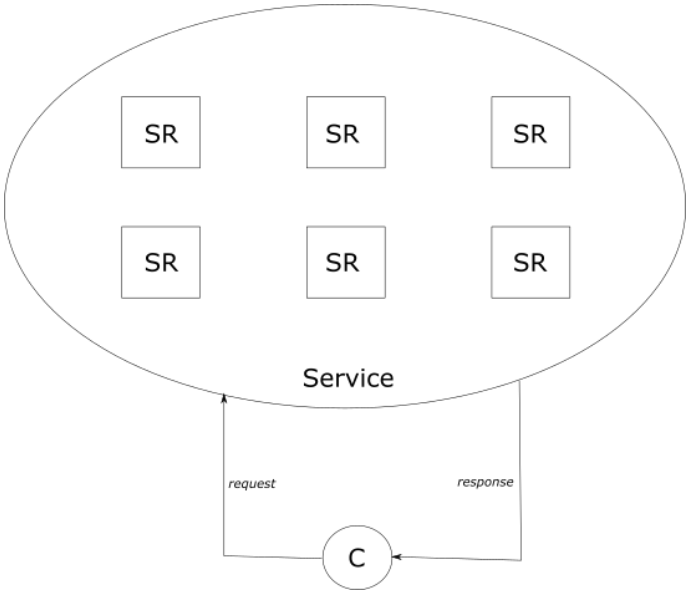


**Figure 2.2.2 – Replicating servers**

- ***SR*** – Server replica
- ***C*** – Client

In Figure 2.2.2, it can be noticed that a client is still provided with a single service. On the other hand, there are many server replicas that provide that particular service. They are completely transparent to a client. Due to that, there must exist a defined protocol that coordinates client interactions with server replicas. Discussing this particular protocol is beyond the scope of this paper.

Let us consider a following situation – Client connects to a server replica, and communicates with it in order to obtain responses and send requests. Throughout time, that server replica becomes faulty (e.g. crashes and is no longer available). The aforementioned protocol must re-connect a client to another server replica, which will then continue to provide the client with responses, with all that being completely transparent to the client. Question arises - *"How will the new server replica know what the last one was doing?"*. Additionally, all server replicas can potentially provide the service to different clients.

The solution that the SMR approach defines is - Every server replica will be implemented as a deterministic state machine, which consists of states and transitions. Whenever a client sends a request to be executed, it will be executed on all server replicas, thus leading to a transition to another state. This approach also states that transitions between states on all server replicas must be exactly the same. Consequently, when a situation mentioned before occurs, since the new server replica will be in the same state as the crashed server replica, it can easily continue where the former one left off.

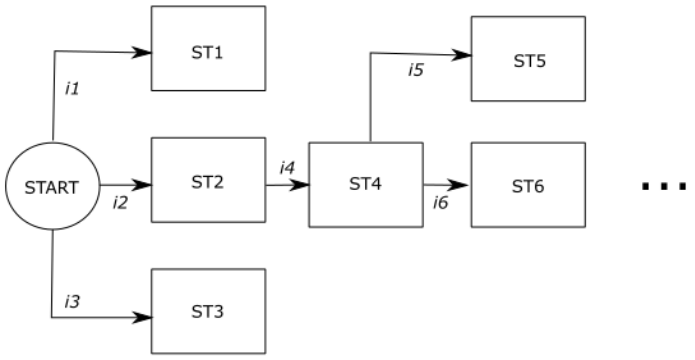


**Figure 2.2.3 – Deterministic state machine within a server replica**

- ***ST*** – State
- ***C*** – Client

An example of a deterministic state machine within a server replica is illustrated in Figure 2.2.3. The state machine has to be deterministic. If it were non-deterministic, it would have the property of being able to make a transition to different states in case of the same input. Therefore, it would be possible that there is a state machine (or state machines) whose current state would be inconsistent with the states of others.

As Schneider states [7], the key for implementing this state machine is:

- **Replica coordination** - all replicas receive and process the same sequence of requests. This can be further decomposed into:
  - **Agreement** - every non-faulty server replica receives every request.

- **Order** - every non-faulty server replica processes the requests it receives in the same relative order.

SMR approach also enables resilience to both Byzantine failures and Start-Stop failures (simply by assuming that there will never be more than a certain number of faulty/malicious server replicas). In [7], it is said that this approach guarantees proper functioning if the number of faulty replicas is less than one third of the number of all replicas in the network. It is the main idea behind blockchain, where reaching a consensus on what the next block will be is nothing more than deciding collectively what the next state of all the state machine replicas will be.

### *2.2.2. Mempool component*

Again, let us assume a standard client-server architecture using the SMR approach. Therefore, each server has its own state machine replica (blockchain)

.

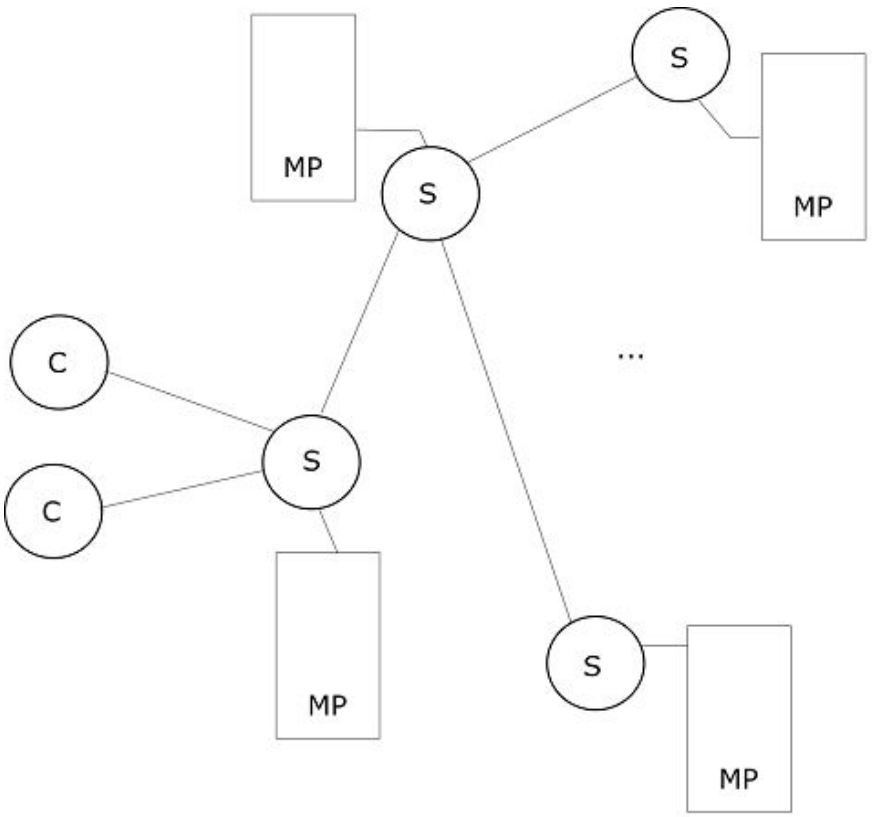


**Figure 2.2.4 – Client-server architecture with the Mempool component**

- ***S*** – Server (Server replica)

- ***C*** – Client
- ***MP*** – Mempool (*Memory pool*)

A client may send its request to a server, in order to receive a response. According to the SMR approach, this means that each server must receive this request and then process it. Yet, as shown in Figure 2.2.4, a client is connected to only one server. Furthermore, all the servers are not connected in a way that each server is linked with all the others. Now, each server is connected only to a subset of other replicas. This is one very important restriction imposed by the Tendermint network, where the servers are not a part of a single administrative domain. For that reason, it is impossible for a server to be connected to all the other servers. Tendermint is a WAN (*Wide Area Network*), not a LAN (*Local Area Network*).

For a client's request to reach all the servers in the network, a suitable dissemination algorithm must be used. This algorithm has to distribute the client's transaction all over the network, in a way that the transaction eventually reaches all non-faulty servers. Data dissemination algorithms are in literature also called gossiping protocols or epidemic protocols [8]. Disseminating a client's transaction is one of two roles of the Tendermint's Mempool component.

Data dissemination algorithms are based on periodic forwarding of certain information to the neighboring servers (peers). Often, the assumption is that a server knows about all the other servers in the network, so that it can periodically select a specific subset of them and forward a message to them. As noted earlier, this is not the case in Tendermint, due to the fact that each server knows only about a subset of other servers. Therefore, in Tendermint, data dissemination algorithm would require that the server periodically sends a message to everyone in its subset. Such periodic messaging leads to very rapid dissemination of information through the network. However, there is a redundancy when receiving messages - it is possible that the server receives a message that contains information about something it already knows. On the one hand, this represents an unnecessary burden imposed on the server. On the other hand, redundancy is useful in case some of the servers in the network become faulty/malicious, because the same message can reach the same server via different paths in the network graph.

Another role of the Mempool component is to store all the transactions that have not become a part of the blockchain yet. When a client's transaction reaches the server, it certainly gets disseminated throughout the network. However, it is obvious that it cannot be processed immediately, because a consensus by all the servers must be reached before including a particular transaction in a blockchain. Until that occurs, client's transaction is stored within a special RAM (*Random Access Memory*) memory of the server. This memory contains all the transactions that have not been included in the blockchain yet. When a consensus is reached and it is decided what is the next block of transactions to be added to the blockchain (or what is the next state of the state machine), transactions from that block are deleted from the RAM memories of all servers. In the actual Tendermint network, transactions are stored on a disc instead of in a RAM memory.

### *2.2.3. System model*

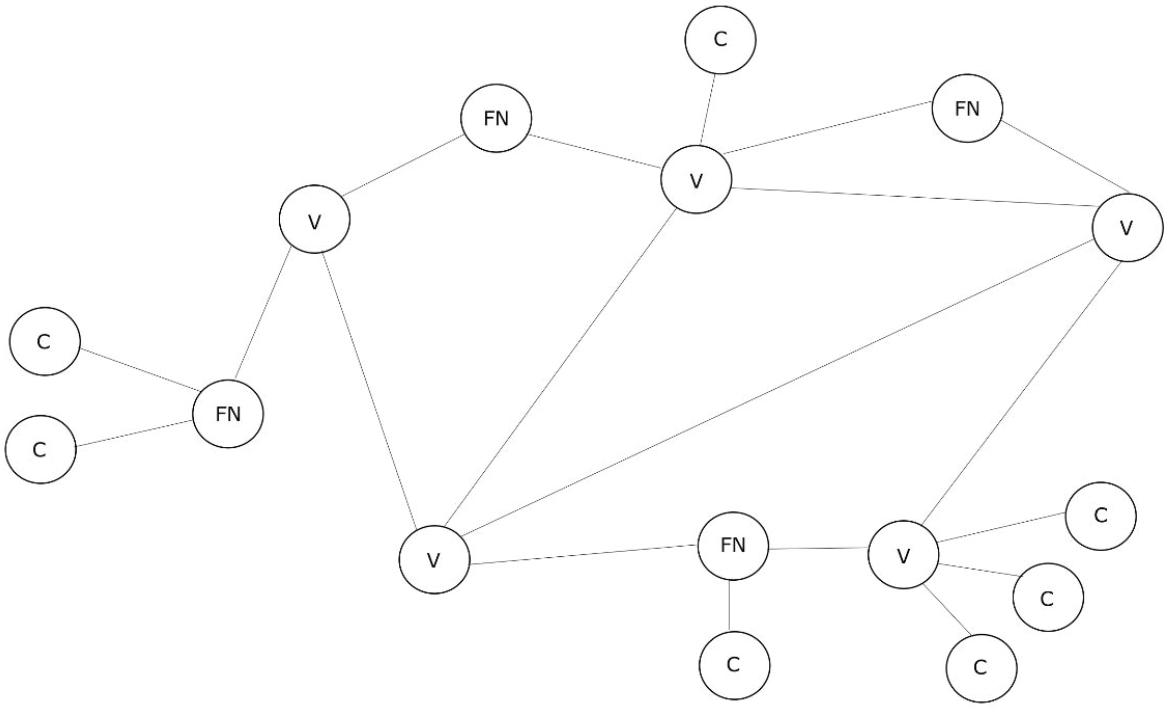


**Figure 2.2.5 – System model**

- ***C*** - Client
- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)

Let us consider a client-server architecture that consists of different types of nodes, as shown in Figure 2.2.5. Let those nodes be denoted as FN and V. In a practical situation, a node may be represented by either a single hardware device or multiple hardware devices (e.g. within an organization). Therefore, this type of model representation allows for a certain level of abstraction that omits the hardware specification, yet enables a concise discussion without loss of generality. In further text, we assume that each node is represented by a single hardware device (server). Therefore, we will distinguish between two types of nodes:

- FN server (FN node)
- V server (V node)

Every V server is also a FN server. However, vice-versa does not apply - FN server may or may not be a V server. In Figure 2.2.5, it is assumed that every server marked as FN is not a V server.

Let there be a set of ***m*** FN nodes. Nodes in this model are not part of a single administrative domain. Consequently, a full-mesh connection among them cannot be administered. Due to that reason, a non-client node inside the network may only be connected to a subset of other FN nodes. That subset is called a ***peer subset***. The links (between nodes) shown in Figure 2.2.5 are bidirectional, so that a two-way communication can be established. The peer subset is prone to change as time passes, ergo it is not a static one.

It is assumed that the minimum size of the peer subset is 1. Consequently, there cannot exist a FN node which is disconnected from every other node. If the network was to be modeled by a graph, that graph would always be connected. Each FN server has a public IP (*Internet Protocol*) address. Each FN server has a RAM memory which is a part of the Mempool component. This memory holds those transactions which have not been added to the blockchain yet. As a part of the system, there is a blockchain replicated on each FN node. This blockchain is shown in Figure 2.2.6.

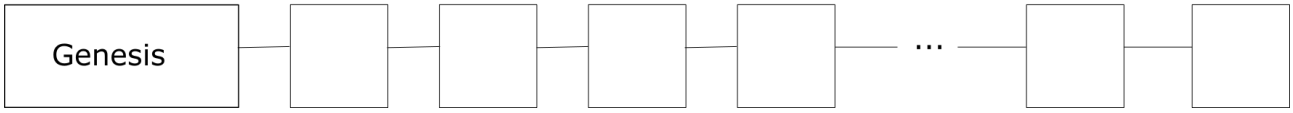


**Figure 2.2.6 – Blockchain on each server**

The blockchain consists of blocks of transactions that are linked in some way (e.g. every block contains a hash function of the previous block). The first block in the blockchain is called a ***Genesis*** block, and every other block can trace its lineage back to it.

System model described in this chapter is simplified in comparison to the actual model that exists in Tendermint. In Tendermint, there are a few more types of nodes. However, all the principles regarding the Mempool component functioning remain exactly the same in our model as they are in Tendermint. Therefore, algorithms described in this paper can easily be adjusted to correspond with the network architecture of Tendermint. As a summary, all the assumptions regarding the system model are listed:

1. FN servers have public IP addresses.
2. The connection among the nodes in the network is not a full mesh, but a peer-to-peer connection.
3. A FN may change its peer subset dynamically.
4. Minimum size of the peer subset is 1.
5. Network is non-fault-tolerant. Servers within the network are one hundred percent reliable.

### *2.2.4. Disseminating a client's transaction and reaching a consensus*

In this chapter, all the concepts of the Tendermint's consensus algorithm that are tightly connected with the data dissemination algorithm will be explained. Consensus algorithm used in Tendermint is explained in detail in [1].

As said earlier, a client wants to have its transaction processed by the Tendermint network. A transaction may represent any data that, when processed, is useful to the client (e.g. storing money transactions in a blockchain alongside executing them). Implementation-wise, a transaction is nothing but a byte array. From the client's perspective, there are two important events that need to occur:

– A transaction must be executed. <1>

– Evidence of that execution must be stored inside a blockchain. <2>

To achieve that, a client may connect to a FN node of its own choice. Upon establishing a connection with a FN node, a client may send its transaction to the FN node. A client may, at any point in time, disconnect from a connected FN node, and connect to another one.

When a FN node receives a transaction, it checks its validity. Client may also send a transaction which is invalid according to some system-related metrics. That transaction is rejected and not processed by the Tendermint network. If the transaction is valid, the node stores it inside its own RAM memory. In addition, the transaction is gossiped all over the network to all of the peers of that particular FN node, so that it could be stored inside their own RAM memories. Then the peers re-gossip the received transaction and so forth. By doing so, the transaction gets disseminated to everyone in the network. As stated in the previous chapter, there is a blockchain replicated on each server, as illustrated in Figure 2.2.7.

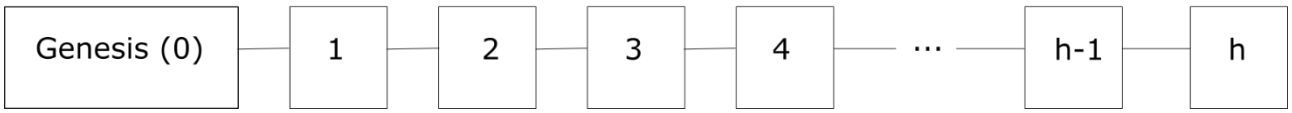


**Figure 2.2.7 – Blockchain on each server**

Each block inside the blockchain is described with a number, which is in Tendermint denoted as ***height***. Let that number be denoted as ***h***. Each block also contains a batch of transactions that are not only recorded permanently as a history of blockchain, but are also executed whenever a new block is added to the blockchain This satisfies client's conditions <1> and <2>.

In order to add a new block to the blockchain, a set of V nodes must reach a consensus for each blockchain height $h_b$. Blockchain height is nothing but a height of a block to be added to the blockchain - it represents a number of blocks inside the blockchain, excluding the Genesis block.

That set of V servers is called a ***validator set***. A ***validator set*** is a set of FN nodes that is chosen for each consensus instance, which is executed whenever a new block at height ***h*** is to be added to the blockchain. Let the size of the validator set be denoted as ***n*** where $n < m$. When a FN node is added to the validator set, it becomes a V node. The Genesis block contains information about the first ever validator set.

Blockchain height changes whenever a consensus instance is initiated. Therefore, for every new blockchain height, the validator set changes as well. The validator set is determined by an external process. This information was extracted from [18]. Algorithms for determining the validator set are beyond the scope of this paper.

However, the validator set can remain the same even throughout multiple executions of the consensus algorithm. As stated in [1], the validator set is always chosen in a way to maximize the utilization of the nodes' voting power. Each FN server has a number joined to it. This number represents the server's part in reaching a final decision when voting on what the next block of transactions will be. For that reason, there is a probability that the validator set will never change, for example in a network where there is a certain number of full nodes whose voting power dominates over others'. But in general, this must not be a presumption. Tendermint's consensus algorithm guarantees successful termination if the voting power of faulty processes is less than one third of the voting power of all processes [1].

Furthermore, each consensus instance consists of ***k*** rounds, where at each round a new ***proposer*** is potentially selected from the validator set. A ***proposer*** is a node that has the obligation to propose what the next block in the blockchain will be. It does so by selecting a certain number of transactions from its RAM memory, creating a block, and then proposing it to the others in the validator set. If the proposition becomes accepted in that consensus round, then there will be no more rounds for that particular blockchain height due to the fact that the consensus was reached. New round is initiated whenever a consensus could not be reached in the previous round. Once again, it is possible that through each round ***1...k***, the proposer remains the same, but this is not a general case.

We come to a necessary, but not a sufficient condition when it comes to the Mempool component and disseminating transactions from it - when a FN receives a client transaction inside its RAM memory, it must gossip it in a way that guarantees (or at least provides a high probability outcome) that the transaction will eventually reach a RAM memory of at least one FN node that will be a member of the validator set (hence, a V node) and a proposer in some round $k_i$ at some blockchain height $h_{bj}$ in the future. That particular course of events will then lead to a client transaction being both executed and stored within a blockchain.

Condition is not sufficient due to three reasons. First one is that the maximum number of transactions in a proposed block can be less than the number of transactions in the RAM memory of the proposer. In that case, if the transaction reaches a single proposer, it may happen that the proposer cannot propose that transaction because there is no space for it in a proposed block. This is a problem if that proposer never becomes a proposer in the future. Second reason is that the validator set is a dynamic one and prone to changes, and is therefore unknown in advance. Third reason is that the proposer in each round is a potentially different server from the validator set, thus it is also unknown in advance. That being said, we may conclude that **in order to satisfy <1> and <2>, a client's transaction must reach all the servers in the network.**

Mempool component is a bottleneck of the system. RAM memory holds transactions which are not a part of the blockchain, and the faster it is emptied, the faster is the overall execution of transactions. In case that memory becomes overloaded, a client's transaction may not be executed. Furthermore, overloading a RAM memory may lead to server crashes and failures. For that reason, it is important that the RAM memory empties as quick as possible. As we said, it is emptied whenever a transaction stored within it becomes included in the blockchain. In order to add a transaction to the blockchain, that transaction must reach all the servers in the network as fast as possible, so that a proposer from the validator set could propose it. Thus, the time between a client sending a transaction and the network executing that transaction is directly affected by the dissemination algorithm used in the network.

As a summary, in the following text there is a list of facts, concerning the consensus algorithm, which are relevant to the data dissemination algorithms analysis in the following chapters:

1. Validator set is a subset of full nodes set – $n < m$.
2. Validator set is a dynamic one – it changes potentially for each block to be added to the blockchain.

3. Proposer in each round is chosen dynamically – it potentially changes for each round.

# 3. FORMAL ANALYSIS

In this chapter, a formal analysis of the Tendermint's current algorithm used for gossiping clients' transactions will first be presented. Afterwards, a formal analysis of three different algorithms will be provided. Each of the three algorithms is based on one of the approaches listed in the introduction section of this thesis. Analysis of every algorithm will be followed by a pseudocode, performance discussion, and ways of operating of that algorithm in a network where fault-tolerance should exist.

## 3.1. Current solution

In this subsection, a discussion of the current Tendermint gossiping algorithm will be presented.

### *3.1.1. Algorithm explanation*

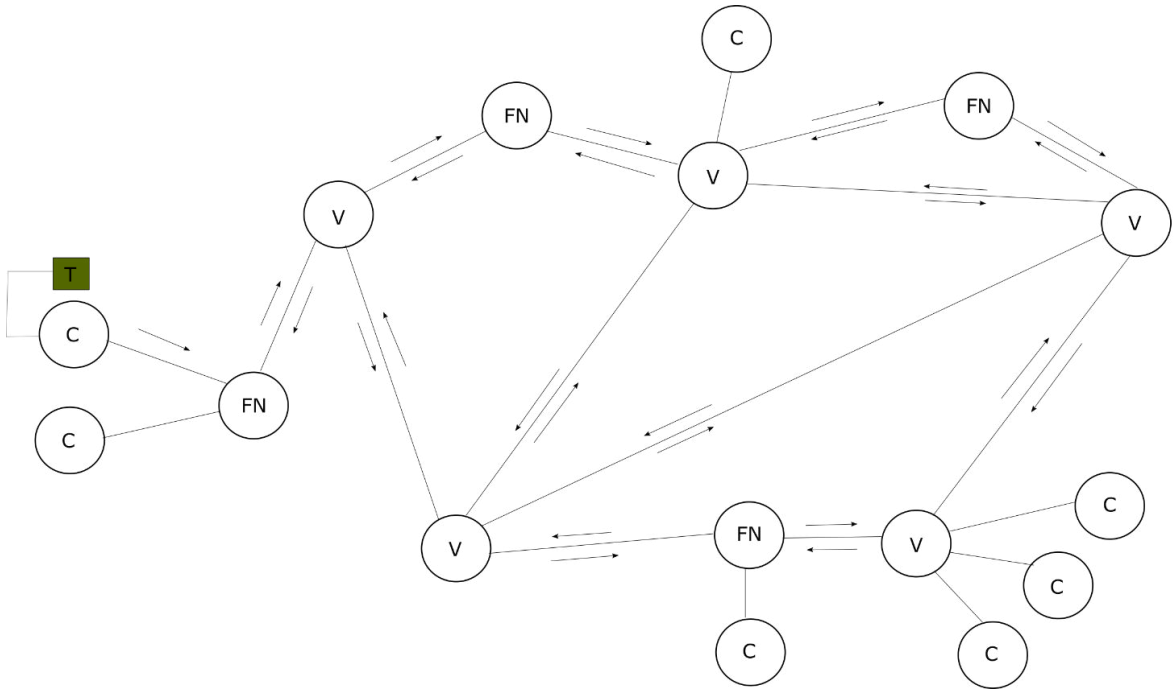


**Figure 3.1.1 – Current dissemination algorithm used in Tendermint**

- ***C*** - Client
- ***FN*** – Standard server (*Full node*)

- ***V*** – Validator server (*Validator node*)
- ***T*** – Client's transaction

Let us assume that a client wants to have its transaction processed by the Tendermint network, in a way explained earlier in the text, as shown in Figure 3.1.1. In order for a transaction to be processed, a client has to send it to a FN server with whom it had previously connected. Pseudocode of the current dissemination algorithm used in Tendermint is given in the following text. This algorithm is executed on each FN server.

**Table 3.1.1 – Current solution – pseudocode**

```
1.  upon receive(T transaction, Node sender) {
2.      // Node = {C, FN, V}
3.
4.      if (sender == C) {
5.          // In Tendermint, this condition doesn't exist
6.
7.          bool valid = checkTx(transaction);
8.          if (valid == false) {
9.             return;
10.         }
11.     }
12.
13.     bool isInMyMempool = checkMempool(transaction);
14.     if (isInMyMempool == true) {
15.         return;
16.     }
17.
18.     addMempool(transaction);
19.
20.
21.     for (Node node : getPeerSubset()) {
22.         node.send(transaction, self);
23.     }
24. }
```

Whenever a FN server receives a transaction, whether from a client or from a peer, a function ***receive(T, Node)*** is called. Within the function, validity of the transaction is checked via a call to the ***checkTx(T)***. Implementation of ***checkTx(T)*** in the *Go* programming language can be found at [19]. Considering that the assumption in our system model is that the network is completely reliable, we assume that a server always unconditionally trusts all of the peers in its peer subset. For that reason, checking a transaction's validity is performed only if the transaction was sent by a client. It should be noted that this check is always done in reality in Tendermint, due to the fact that some of the servers may be malicious or faulty. In case the transaction is invalid, nothing is done. In case the transaction is valid, it is checked if it is stored within the server's RAM memory.

If it is, a redundant message has just been received and nothing is done. If it is not inside the server's memory, the transaction is added to the RAM. After that, it is gossiped to everyone in the peer subset.

### *3.1.2. Performance analysis*

If we assume that the network is modeled via a connected graph which has ***e*** nodes and ***v*** vertices, then the number of messages exchanged for one client's transaction is equal to ***2v***, as shown in Figure 3.1.1. Optimal number of messages that are exchanged for one client's transaction, where each server would "hear" new information only once, would be equal to ***e-1*** (C → FN message is not considered since it is not a part of the gossiping protocol). Number of redundant messages that each server in the network receives is very large, as the results of simulation will show. This is the consequence of the fact that each received message is simply broadcast further through the network. This is additionally affected by the cycles in the network graph.

### *3.1.3. Concluding remarks*

Current dissemination algorithm used in Tendermint guarantees that the transaction will reach all the servers in the network, under the assumption that the minimal size of the peer subset is 1 and there are no disconnected nodes. As the message definitely reaches all the servers, conditions <1> and <2> are satisfied. By sending the message to everyone in the peer subset, this algorithm is resilient to faults and crashes. However, from everything that has been presented, it may be concluded that the current solution used in Tendermint leads to a reception of a large number of redundant messages.

Simple improvement would be that, during the gossiping phase, a transaction would not be broadcast to a FN whom it was just received from. The aforementioned ***receive(T, Node)*** function would be changed accordingly:

**Table 3.1.2 – Simple improvement of the current solution**

```
1. // ...
2. for (Node node : getPeerSubset()) {
3.     if (node != sender) {
4.         node.send(transaction, self);
5.     }
6. }
7. // ...
```

## 3.2. Client-based dissemination approach

Dissemination algorithm explained in this chapter is based on the first of three approaches listed in the introduction section. The idea is that a client is responsible for the dissemination.

### *3.2.1. Algorithm explanation*

In literature a variant of this type of algorithm can be found, which requires that a client sends its transaction to all the FN servers in the network [9]. Evidently, this would demand that the client knows the IP addresses of all the FN servers in the network. This kind of approach would definitely satisfy <1> and <2>. In this situation, client would be more burdened as the dissemination would be its responsibility. However, approach in [9] is not focused on optimizing the usage of client's resources.

Idea proposed here is even more efficient than the one in [9]. The presumption is that the client has a local module, which is in literature often denoted as *Oracle* [10]. This is illustrated on the Figure 3.2.1. In our system model, this module is capable of keeping track of all the changes in the validator set. Whenever a change occurs in the validator set, a client is notified. Client may now send its transaction only to those who are in the validator set.

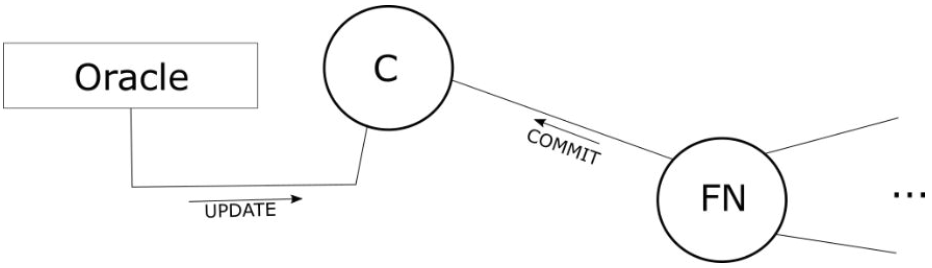


**Figure 3.2.1 – Client-based dissemination algorithm**

- ***C*** - Client

- ***FN*** – Standard server (*Full node*)

**Table 3.2.1 – Client-based dissemination algorithm – pseudocode**

```
1.  Set<Node> currentVS = EMPTYSET;
2.  Set<T> uncomitted = EMPTYSET;
3.
4.  upon receive(UPDATE, Set<Node> newVS) {
5.      for (Node node : newVS) {
6.          if (!currentVS.contains(node)) {
7.              node.send(uncomitted);
8.          }
9.      }
10.
11.     currentVS = newVS;
12. }
13.
14. upon receive(COMMIT, T transaction) {
15.     uncomitted.remove(transaction);
16. }
17.
18. upon send(T transaction) {
19.     for (Node node : currentVS) {
20.         node.send(transaction);
21.     }
22.
23.     uncomitted.add(transaction);
24. }
```

Client locally keeps track of two sets – current validator set and a set of transactions that have not been ***committed*** (added to the blockchain) yet. When a client wishes to send its transaction, a function ***send(T)*** is called. This function sends the transaction it receives as a parameter to everyone in the current validator set, and then it adds the transaction to the set of uncommitted transactions. When a consensus is reached for a certain transaction, a FN server will send a ***COMMIT*** message to the client. Upon receiving the ***COMMIT*** message, a client will remove the according transaction from the set of uncommitted transactions. Additionally, whenever a change in the validator set occurs, client's local module (*Oracle*) sends an ***UPDATE*** message to the client. Upon receiving the ***UPDATE*** message, the client sends its uncommitted transactions only to those in the new validator set who are not in the old validator set. By doing so, it is

guaranteed that the transaction will always reach everyone in the validator set. After sending all of the uncommitted transactions, the current validator set is updated.

### *3.2.2. Performance analysis*

If we denote the size of the validator set as ***n***, then in an ideal scenario, the number of messages exchanged for one client's transaction equals to ***n***. If a transaction is not added to the blockchain before the next change in the validator set occurs, it is demanded that the transaction is sent to the servers who are presented by the difference of two sets – ***newVS*** and ***currentVS*** (***newVS \ currentVS***). Therefore, the maximum number of messages that will be exchanged for one client's transaction depends on the consensus algorithm. If a consensus is reached in the first consensus instance after the transaction is sent, there will be no need for re-sending that transaction upon receiving the ***UPDATE*** message. As far as the number of redundant messages each server in the network receives is concerned, it is equal to 0. Messages are received only by those who are members of the validator set, and they are received only once.

### *3.2.3. Concluding remarks*

Client-based dissemination algorithm guarantees that the uncommitted transaction will always reach everyone in the validator set. Therefore, conditions <1> and <2> are satisfied. In case of malicious or faulty servers in the validator set, algorithm will still work properly, considering the fact that the transaction is sent to everyone in the validator set. If we assume that the number of faulty validators will never go over ***n/3***, and that the proposer is chosen uniformly at random, a client's transaction will always become proposed.

It is important to observe that this algorithm is not easy to implement. Question arises concerning the implementation of the client's local module – *Oracle*. On the other hand, considering an ideal solution is useful because it provides material for different metrics (e.g. redundancy) comparison between an ideal solution and the solutions used in real systems.

## 3.3. Tree clustering

Algorithm described in this subsection is based on the second of three approaches – classification of servers in the network into structured groups. As mentioned earlier in the paper, nodes within the system model are not part of a single administrative domain. They are part of a large scale wide area network, where full connectivity cannot be established. For that reason, a FN node can only be connected to a subset of other FN nodes, which we denoted as the *peer subset*. We assumed that the minimum size of the peer subset is 1. Therefore, if the network was to be modeled by a graph, that graph would always be connected. Question arises – Can we classify into certain groups (***clusters***) those servers that are "close" to one another according to some criteria?

### *3.3.1. Algorithm explanation*

The idea behind this algorithm combines two of the solutions proposed in [11] - ***Flat membership server based protocol*** ([11] - 2.3.1) and ***Hierarchical membership protocol*** ([11] - 2.3.2). However, those two solutions applied directly to our system model would not improve its gossiping algorithm performances in a significant manner. Due to that, additional modification to the combination of the aforesaid solutions is proposed in further text, with the goal of reducing the redundancy related to message passing.

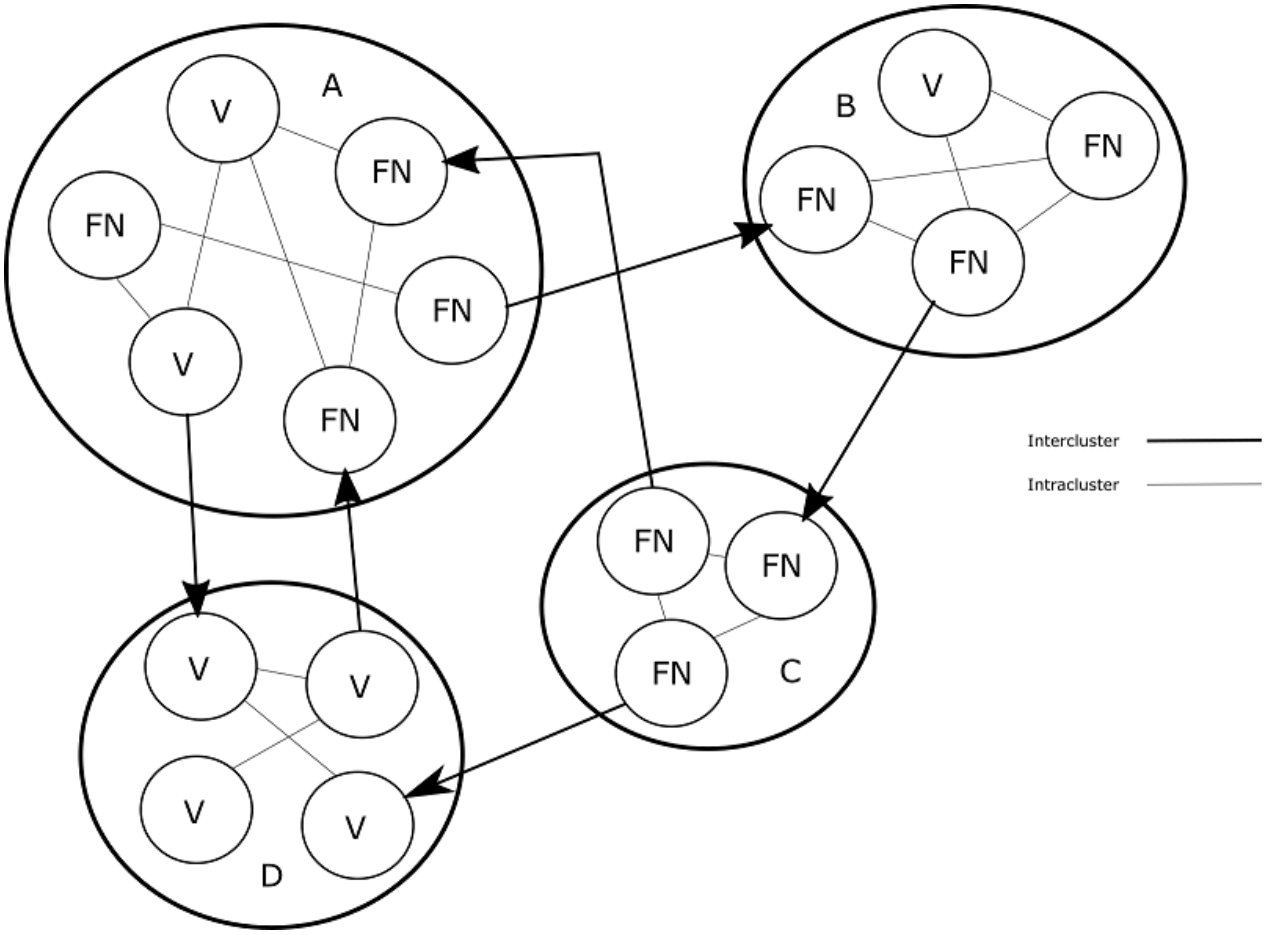


**Figure 3.3.1 – Dividing the network into clusters –** ***Hierarchical membership protocol***

- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)

First and foremost, let us assume that the network is now divided into clusters. As presented in [11], nodes are clustered according to geographical proximity or some network-related metrics that refers to network proximity (e.g. round-trip delay or number of hops between nodes). Clusters could be maintained by an external process [11]. In [11], there is a mathematical proof which states that only a small number of connections among clusters is required to keep the system connected. In Figure 3.3.1, there are four clusters, *A*, *B*, *C* and *D*. There are also two types of connections (links):

– **Intercluster** → they represent links between different clusters – they are unidirectional.

– **Intracluster** → they represent links between nodes within the same cluster – they are bidirectional.

Node may be responsible for only one intercluster connection - this is also a restriction presented in [11].

Since the number of intercluster connections doesn't have to be large, we have reduced the number of links within the network. Furthermore, by clustering the nodes according to some criterion, we achieve having nodes that are in close "proximity" to one another, thus enabling a faster communication between them. Regardless, redundancy still exists when receiving messages, both in the clusters and in the entire network. First, we will discuss how to solve the redundancy problem within a cluster. Then, we will show how to solve the redundancy problem within the network.

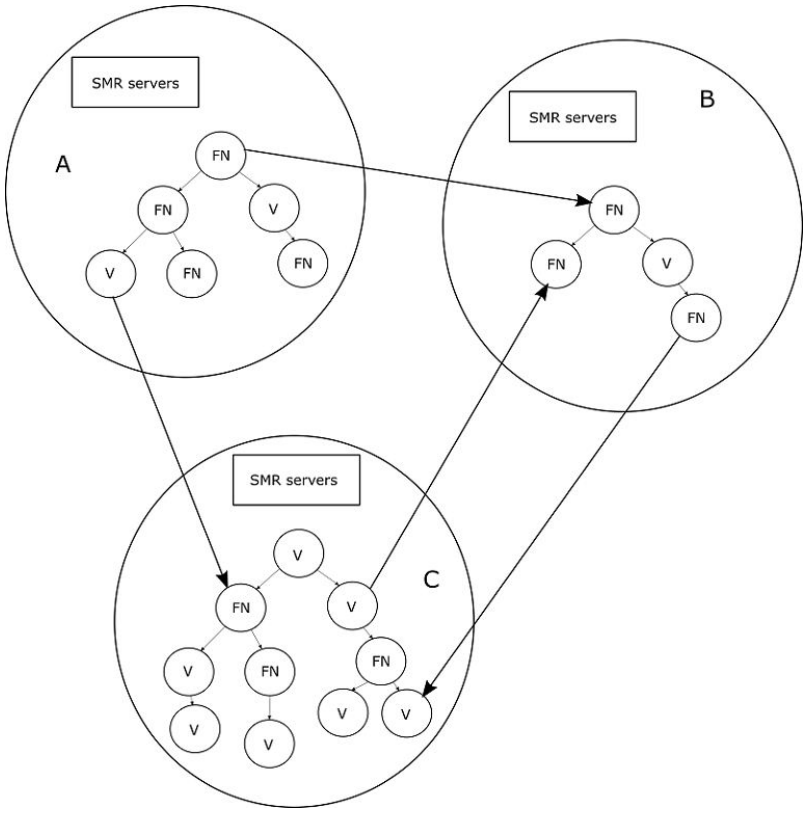


**Figure 3.3.2 – Solving the redundancy problem within the clusters - *Flat membership server based protocol***

- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)

The main cause of the overhead problem is - the graph. Graph can contain cycles. Since the network is modeled via a graph, with its links being bidirectional, a client's transaction could travel through the network using different paths, only to finally reach the same destination multiple times. To avoid that, a non-cyclical unidirectional data structure can be used - the tree. If nodes within the cluster were to be connected in a tree-like scheme, as shown in Figure 3.3.2, it would result in each node receiving the message only once. However, two questions arise:

1. Who maintains the tree within the cluster?
2. How to propagate a message upward in a tree if we use a unidirectional link?

To answer the first question, an approach explained in [11], *Flat membership server based protocol*, can be applied.

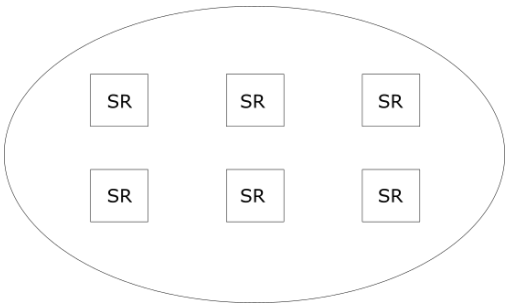


**Figure 3.3.3 – SMR servers used for maintaining trees**

- ***SR*** – Server replica

We consider having a set of servers, where each server has its own state machine replica, as shown in Figure 3.3.3. These servers should be distinguished from FN and V servers. When a new node wants to join the network, which can occur asynchronously and is therefore a dynamic process, it sends a request to a server set which is in charge of that node's most "proximate" cluster. Since the approach used here is SMR, that node is completely unaware of the existence of multiple servers. As far as it is concerned, it contacts a single server with a request to join the cluster. It is the

responsibility of all the servers in the set to reach a consensus on where in the tree will the new node be added. When that occurs, they all collectively transition to the next state.

Apart from having to handle membership requests, servers must also keep the tree as efficient as possible (balanced, or even complete). It is advisable that the tree changes dynamically from time to time, in a way that a certain node does not remain a root node, intermediate node or a leaf node forever. Leaf nodes propagate the information a lot less than interior nodes, and are therefore less utilized, which is bad in case they have a high bandwidth. Shuffling the tree periodically maximizes the utilization of nodes' bandwidth. [12]. Replicating servers eliminates the SPOF problem. However, they have to be properly synchronized, and therefore a consensus algorithm executed on them should be carefully chosen.

To answer the second question, let us consider a situation presented in Figure 3.3.4:

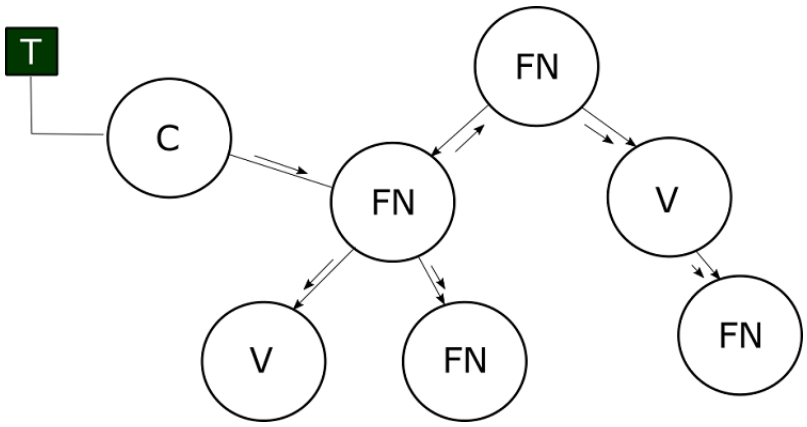


**Figure 3.3.4 – Propagating a client's transaction when using trees**

- ***C*** - Client
- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)
- ***T*** – Client's transaction

When a FN node receives client's transaction, it can easily propagate it down its subtree. However, how will the message reach the rest of the main tree? To solve this problem, we assume that each node contains information about its parent (e.g. parent's IP address). Thus, when a FN shown in Figure 3.3.4 receives client's transaction, it can propagate it towards its parent, then that parent has to propagate it to both its parent and its descendants and so on recursively. Principle can be adopted:

- Only when a node receives a message from a child or from a client, it has to propagate it to its own parent – ***upward gossiping***.
- The node always has to propagate the message down the tree – ***downward gossiping***.

Pseudocode of this solution is given in the following text.

**Table 3.3.1 – Propagating a client's transaction when using trees – pseudocode**

```
1.  // Pseudocode for a FN/V server - without clusters
2.
3.  Node parent = ...;
4.  Set<Node> children = ...;
5.
6.  upon receive(T transaction, Node sender) {
7.      if (sender == C) {
8.          bool valid = checkTx(transaction);
9.          if (valid == false) {
10.             return;
11.         }
12.     }
13.
14.     addMempool(transaction);
15.
16.     if (sender == C || children.contains(sender)) {
17.         if (parent != nil) {
18.             parent.send(transaction, self);
19.         }
20.     }
21.
22.     for (Node child : children) {
23.         if (child == sender) {
24.             continue;
25.         }
26.
27.         child.send(transaction, self);
```

```
    }
}
```

Note that there is no need for calling the ***checkMempool(T)*** function before adding a transaction to the node's RAM memory. This is due to the fact that a FN server will always receive a transaction only once. When not taking clusters into account, this solution provides ideal performances. However, if we take clusters into account, redundancy is present.

First, let us explain how that is true. If we consider having three clusters, *A*, *B* and *C*, as shown in 3.3.2 we see that the network is now again a graph, and a cyclic one. For example, the following intercluster links form a cycle:

- $B \rightarrow C$
- $C \rightarrow B$

So, if there was a client within the cluster *B*, and it sent its transaction to any FN node within that cluster, the transaction would be gossiped all over the cluster locally, via intracluster links. It would also be gossiped to cluster *C* via the intercluster B → C link. Inside the cluster C, message would be gossiped locally, but then again returned to cluster B via the C → B intercluster link. herefore, to avoid adding a duplicate of a transaction into node's RAM, this time a call to ***checkMempool(T)*** is required. In further text a pseudocode for a FN server is given. This time, clusters are taken into consideration.

**Table 3.3.2 – Solving the redundancy problem within the network – pseudocode**

```
// Pseudocode for a FN/V server - with clusters

Node parent = ...;
Set<Node> children = ...;
Node interclusterLink = ...;

upon receive(T transaction, Node sender) {
    if (sender == C) {
        bool valid = checkTx(transaction);
```

```
10.         if (valid == false) {
11.             return;
12.         }
13.     }
14.
15.     bool isInMyMempool = checkMempool(transaction);
16.     if (isInMyMempool == true) {
17.         return;
18.     }
19.
20.     addMempool(transaction);
21.
22.     if (sender == C || children.contains(sender) || isFromAnotherCluster(sender)) {
23.         if (parent != nil) {
24.             parent.send(transaction, self);
25.         }
26.     }
27.
28.     for (Node child : children) {
29.         if (child == sender) {
30.             continue;
31.         }
32.
33.         child.send(transaction, self);
34.     }
35.
36.     if (interclusterLink != nil) {
37.         interclusterLink.send(transaction, self);
38.     }
39. }
```

Observe that in this case, upward gossiping is also required when the sender is from another cluster. The check is performed by a call to ***isFromAnotherCluster(Node)*** function, which could, for example, check if the sender's IP address belongs to the receiver's cluster. Also, a new node-local-variable is added – ***interclusterLink***. Since intercluster links are unidirectional, it is assumed that this variable is set only for a node which can send a message to another cluster.

By forming a tree in a cluster, redundancy within a cluster is completely eliminated. However, within the network, redundancy still exists. A node may still receive redundant messages, due to the fact that the network graph can contain cycles. To solve this, a generalization of the idea to use trees inside a cluster can be applied to the whole network. Illustration is shown on Figure 3.3.5.

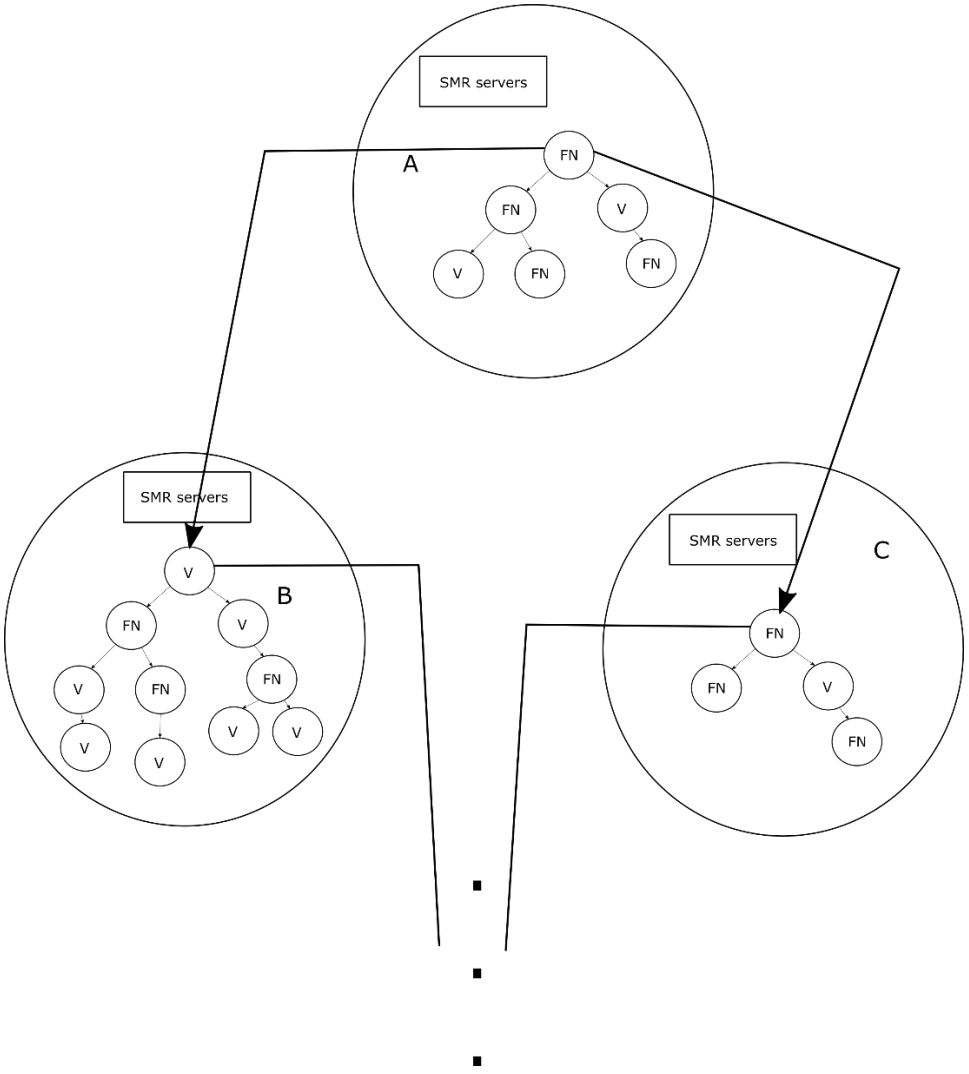


**Figure 3.3.5 – Solving the redundancy problem within the network – final solution**

- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)

Now all of the clusters that constitute the network form a tree. In this case, a node can be responsible for more than one intercluster connection - this is less restrictive than [11]. Root of every tree inside a cluster contains links (IP addresses) towards roots of trees inside its cluster's children. The information is propagated in a completely same manner as in a in-cluster tree.

Observe that the restriction in [11] did not have to be mitigated for this algorithm to work. A node could still remain responsible for only one intercluster link, but this would result in multiple nodes within a cluster being connected to different clusters. It is more efficient to tie the intercluster links only to a single node – a root node. Therefore, in further text, we assume that a only a root

node can contain intercluster connections. In the following text, a pseudocode of a final solution for the tree clustering data dissemination algorithm is given.

**Table 3.3.3 – Tree clustering – pseudocode for a root node**

```
// Pseudocode for a FN/V server - root node

Node parent = ...; // is always from another cluster
Set<Node> children = ...;
Set<Node> interclusterChildren = ...;

upon receive(T transaction, Node sender) {
    if (sender == C) {
        bool valid = checkTx(transaction);
        if (valid == false) {
            return;
        }
    }

    addMempool(transaction);

    if (sender == C || children.contains(sender)
    || interclusterChildren.contains(sender)) {
        if (parent != nil) {
            parent.receive(transaction, self);
        }
    }

    for (Node child : children) {
        if (child == sender) {
            continue;
        }

        child.send(transaction, self);
    }

    for (Node interclusterChild : interclusterChildren) {
        if (interclusterChild == sender) {
            continue;
        }

        interclusterChild.send(transaction, self);
    }
}
```

**Table 3.3.4 – Tree clustering – pseudocode for an interior node**

```
// Pseudocode for a FN/V server - interior node

Node parent = ...; // is always from the same cluster
Set<Node> children = ...;

upon receive(T transaction, Node sender) {
    if (sender == C) {
        bool valid = checkTx(transaction);
        if (valid == false) {
            return;
        }
    }
```

```
13.
14.     addMempool(transaction);
15.
16.     if (sender == C || children.contains(sender)) {
17.         // parent is never nil for an interior node
18.         parent.send(transaction, self);
19.     }
20.
21.     for (Node child : children) {
22.         if (child == sender) {
23.             continue;
24.         }
25.
26.         child.send(transaction, self);
27.     }
28. }
```

In this case, a call to ***checkMempool(T)*** is also unnecessary. Every FN server will receive a transaction only once.

### *3.3.2. Performance analysis*

First and foremost, let us consider the advantages of classifying the servers in the network into clusters. We assume that the network is modeled via a connected graph (which is actually a tree in the final solution) which has ***e*** nodes and ***v*** vertices. In the current solution, the number of messages that are exchanged when gossiping one client transaction is equal to ***2v***. Since [11] provides mathematical proof that by using the clustering approach we reduce the number of vertices, we may conclude that clustering also reduces the number of messages exchanged for a single client transaction.

Furthermore, by clustering the nodes according to some criterion, we achieve having nodes that are in close "proximity" to one another, thus enabling a faster communication between them. Clustering reduces the latency in the network. By forming a tree within a cluster, and also within the entire network, the number of messages exchanged for one client's transaction is equal to ***e-1*** and is therefore optimal. The number of redundant messages each server in the network receives equals to 0.

### *3.3.3. Concluding remarks*

This solution provides no overhead. However, in case of Byzantine or crash failures, it may not work properly. If any node in the tree becomes malicious or faulty, a client's transaction will not reach every node within the network. This means that <1> and <2> will potentially not be satisfied. To solve this, SMR servers inside each cluster could maintain multiple trees - a forest of trees such that a node which is a root in one tree is never a root in another and a node which is a leaf in one tree is always an intermediate node in another (this is done with the goal of achieving optimal usage of the servers' bandwidth). Similar idea is used in SplitStream [12]. It is evident that dealing with faults and crashes requires redundancy and produces overhead, but it is mandatory if we wish to preserve proper functioning of the network.

As far as the drawbacks of this solution are concerned, the first one regards clusters. It is not an easy task to group the nodes properly into clusters, and this is the subject of much ongoing research, as said in [11]. Secondly, implementing a state machine replication set of servers which will maintain a tree dynamically is also challenging. Determining which consensus algorithm to use within that set is vital for valid operating of the system.

On the other hand, if this approach was to be implemented, it would produce no overhead at all. Every node in the network receives information which it knows nothing about only once, and never again. When applying the tree clustering approach to our system model, we know that the transaction will certainly reach every server in the network – thus <1> and <2> are satisfied.

## 3.4. PPP HEAP PSS

Algorithm presented in this chapter is based on the third approach listed in the introduction section – dynamic server linking so as to form a random network graph. The idea of this algorithm combines a three-phase gossiping protocol described in [13] (***PPP – Push-pull-push gossip protocol***), a protocol aware of network capabilities inequality – ***HEAP*** (***HEterogeneity-Aware-Protocol***) [14], and a gossiping service for providing every node with its peer subset – the ***PSS*** (***Peer Sampling Service***) [15].

### *3.4.1. Algorithm explanation*

Therefore, the proposed algorithm consists of three different protocols:

1. *Push-pull-push*
2. *HEAP*
3. *PSS*

Each of the three protocols is additionaly optimized in order to achieve better performances, and will be explained in different subsections. Generally speaking, this solution is based on building a dynamic, unstructured overlay across the network of nodes. Each node will change its peer subset dynamically, and gossip a client's transaction only to nodes in that peer subset. The size of that peer subset will be denoted as node's ***fanout*** – ***f*** ([13], [14], [15]).

#### *i) Push-pull-push*

Let us consider a network of nodes, equivalent to that described in the chapter about the system model. Let every transaction have a unique identifier which is an integer. We will adopt an additional assumption – each node in the network must have knowledge of every other node in the network. It is evident that by adopting this hypothesis, we suppose a full-mesh connectivity, which is not how the Tendermint network operates. However, this assumption is only adopted so that explaining of the first two parts (*PPP*, *HEAP*) of the algorithm would be simplified. In the third subsection – *PSS*, we will explain how this idea is applied to a wide area network with peer-to-peer connectivity.

Hence, there exists a network of nodes where each node contains IP addresses of every other node. Basic gossiping idea is that a node, periodically, sends a message to a subset of nodes picked uniformly at random from the set of all nodes. As stated earlier, the size of that subset is denoted as ***fanout***. Theoretical [11] and experimental [13] analysis has proven that in order to keep the network graph connected with high probability (which implies that all the servers will receive a message), optimal value for ***f*** is ***ln(n)***, where ***n*** represents the number of nodes within the network.

Concept where a three-phase gossiping is used is essential when there is a high network load. First advantage is that it guarantees that a message will not be delivered (added to the RAM

memory of the server) more than once. Second advantage is that it guarantees that a server will never receive a redundant message which holds an entire transaction. In this protocol, it is possible to receive a redundant message which contains only a transaction's identifier. Seeing as the identifier is significantly smaller than the transaction itself, the amount of redundant traffic that reaches a single server is quite lower. Third advantage is that **it enables that only a transaction's identifier is gossiped**. A need for sending an entire transaction happens only if that particular transaction identifier has been explicitly requested. This significantly reduces the amount of traffic that travels through the network.

The three phases used in PPP protocol are:

1. ***PUSH*** – represented by sending a ***PROPOSE*** message
2. ***PULL*** – represented by sending a ***REQUEST*** message
3. ***PUSH*** – represented by sending a ***SERVE*** message

In further text, a pseudocode of a PPP protocol is given, followed by an explanation.

**Table 3.4.1 – PPP protocol – pseudocode**

```
1.  // This is executed on each FN server
2.
3.  // Initialization
4.  int f = ln(n);
5.  Set<int> toPropose = EMPTYSET;
6.  Set<T> delivered = EMPTYSET;
7.  Set<int> requested = EMPTYSET;
8.  start(GossipTimer(gossipPeriod));
9.
10. // Phase 1 - PUSH T ids
11. upon (receive(PUBLISH, Transaction t))) {
12. // When a C sends T to FN
13.     bool valid = checkTx(transaction);
14.     if (valid == false) {
15.         return;
16.     }
17.
18.     deliverEvent(t);
19.     requested.add(t.id);
20.     toPropose.add(t.id);
21. }
22.
23. upon (GossipTimer % gossipPeriod) == 0 {
24.     gossip(toPropose);
25.     toPropose = EMPTYSET; // Infect and die model
26. }
27.
28. // Phase 2 - PULL wanted T ids
29. upon (receive(PROPOSE, proposed)) {
30.     Set<int> wanted = EMPTYSET;
31.     for (int id : proposed) {
32.         if (!requested.contains(id)) {
```

```
            wanted.add(id);
        }
    }

    requested.add(wanted.getAll());
    reply(REQUEST, wanted);
}

// Phase 3 - PUSH requested T
upon (receive(REQUEST, wanted)) {
    Set<T> asked = EMPTYSET;
    for (int id : wanted) {
        asked.add(getEvent(id));
    }

    reply(SERVE, asked);
}

// a message will be delivered only once
upon (receive(SERVE, events)) {
    for (T t : events) {
        toPropose.add(t.id);
        deliverEvent(t);
    }
}

// Miscellaneous
function selectNodes(int f) Set<Node> {
    return f uniformly random nodes from the set of all nodes;
}

function gossip(Set<int> eventIds) {
    Set<Node> peerSubset = selectNodes(f);
    for (Node node : peerSubset) {
        node.send(PROPOSE, eventIds);
    }
}

function getEvent(int id) T {
    return T corresponding to the id;
}

function deliverEvent(T t) {
    delivered.add(t);
    addMempool(t);
}
```

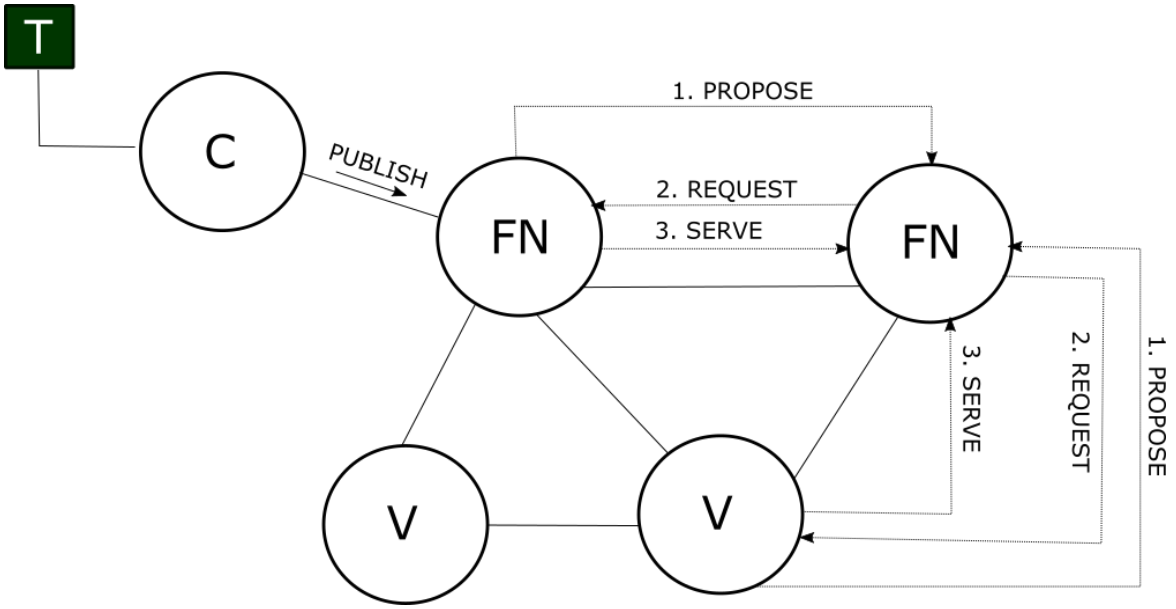


**Figure 3.4.1 – Messages exchanged during the PPP protocol**

- ***C*** – Client
- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)
- ***T*** – Client's transaction

The protocol is based on exchanging three messages between two nodes, as shown in Figure 3.4.1. Whenever a node wishes to broadcast information about uncommitted transactions inside its RAM, it sends a ***PROPOSE*** message (***PROPOSE*** **contains only the transactions' ids**). Afterwards, all the nodes who received that message can reply with a ***REQUEST*** message (***REQUEST*** **contains only the transactions' ids**), in order to get those transactions corresponding to the set of requested ids. Upon receiving a ***SERVE*** message (which is the **only message that contains actual transactions and not only the ids**), the requester may add the served transactions into its RAM memory.

Since we assumed that a node has knowledge of the entire network, its fanout ***f*** is initialized to ***ln(n)***. This represents the size of the peer subset, or more precisely, the number of nodes that are going to receive the *PROPOSE* message. Node keeps track of three sets:

- ***toPropose*** - set of integers which contains the ids of transactions that will be proposed periodically by the *PROPOSE* message
- ***delivered*** - set of transactions which contains all the transactions that were added to the RAM
- ***requested*** - set of integers which contains the ids of transactions that were requested by the *REQUEST* message.

Due to the fact that the gossiping should occur periodically, in the initialization phase, a timer is started. Whenever a certain period of time (***gossipPeriod***) passes, a node sends information about every transaction that hasn't been proposed so far. After doing so, it empties its ***toPropose*** set. This way of operating is named ***Infect and die***, because a node sends the information only once, and then "kills" that information.

Whenever a client sends its transaction to a FN server, the reception of a ***PUBLISH*** message occurs. Upon receiving this message, transaction's validity is checked. If the transaction is valid it is added to the server's RAM memory (and also to the *delivered* set). Additionally, the transaction is added to the two remaining sets – *toPropose* and *requested.* Reason why it is added to the *toPropose* set is that we want the information about that transaction to be disseminated in the following *PROPOSE* message. **Adding the transaction to the *requested* set is crucial, due to the fact that it prevents the server from receiving a redundant message which contains an entire transaction.**

Namely, upon receiving the *PROPOSE* message, a server replies with a *REQUEST* message which contains only those transactions that have been proposed to it, and that it had never requested before. It is important to observe that a server requests a single transaction only once. This is possible because the assumption is that the network is reliable. If a server did not add the transaction to the *requested* set upon receiving it, that transaction could be proposed via the *PROPOSE* message of some other server, considering that the transaction rapidly disseminates through the network (whose graph can contain cycles). Upon receiving that *PROPOSE* message, the

server that initially received a client's transaction would not have it in its *requested* set and would therefore add its identifier to the *REQUEST* message.

Upon receiving a *SERVE* message, that sever would receive, among other transactions, a transaction which it already has in its RAM. Therefore, it is necessary that upon receiving the *PUBLISH* message the server adds the received transaction to the *requested* set as well. This modification was introduced when optimizing the PPP protocol, whose original version can be found at [13].

When a FN node receives a *REQUEST* message, it replies with a *SERVE* message which contains the actual transactions requested. Transactions are fetched from the RAM memory via the call to ***getEvent(int)*** function. When a FN node receives a *SERVE* message, it adds the transactions to its RAM and marks them as future transactions to-be-proposed (when sending a *PROPOSE* message).

Push-pull-push approach reduces the payload in the network, owing to the fact that only a transaction's identifier is gossiped. Sending the actual transaction is delayed until it is explicitly required. However, this approach has one drawback. It assumes that every node in the network has a fanout $f = ln(n)$. This means that, when gossiping, every node will be equally burdened. Unfortunately, not all the nodes have the same capabilities (e.g. bandwidth). Therefore, imposing a high load on a node with lower capabilities may cause crashes, lower throughput and higher latency. HEAP, the second part of this solution, tends to solve this issue.

***ii) HEAP***

HEAP (*HEterogeneity-Aware-Protocol*) does not assume that the network is of homogeneous structure. It relies on nodes having different properties, in such a way that some nodes are faster and more productive than others. HEAP adapts a node's fanout according to its own bandwidth, average bandwidth, and the average fanout (which is, as we said, $ln(n)$).

Equation used in HEAP [14] is:

$$f = b / \_b\_ * ln(n),$$

where $f$ represents the node's fanout, $b$ represents the server's bandwidth and $\_b\_$ represents the average bandwidth in the network. The average bandwidth is updated locally on each of the servers, according to the information about the capabilities of other servers, which is also disseminated in the HEAP protocol. Updated pseudocode, which includes PPP and HEAP is given in further text, followed by an explanation.

**Table 3.4.2 – HEAP protocol – pseudocode**

```
1.  // This is executed on each FN server
2.
3.  // Initialization
4.  Set<Capability> capabilities = EMPTYSET;
5.  Bandwidth b = MY_BANDWIDTH;
6.  Bandwidth _b_ = ...; // average bandwidth
7.
8.  int f = ln(n); // average fanout
9.  Set<int> toPropose = EMPTYSET;
10. Set<T> delivered = EMPTYSET;
11. Set<int> requested = EMPTYSET;
12. start(GossipTimer(gossipPeriod));
13.
14. start(AggregationTimer(aggregationPeriod));
15.
16. // Phase 1 - PUSH T ids
17. upon (receive(PUBLISH, Transaction t))) {
18. // when a C sends T to FN
19.     bool valid = checkTx(transaction);
20.     if (valid == false) {
21.         return;
22.     }
23.
24.     deliverEvent(t);
25.     requested.add(t.id);
26.     toPropose.add(t.id);
27. }
28.
29. upon (GossipTimer % gossipPeriod) == 0 {
30.     gossip(toPropose);
31.     toPropose = EMPTYSET; // Infect and die model
32. }
33.
34. // Phase 2 - PULL wanted T ids
35. upon (receive(PROPOSE, proposed)) {
36.     Set<int> wanted = EMPTYSET;
37.     for (int id : proposed) {
38.         if (!requested.contains(id)) {
39.             wanted.add(id);
40.         }
41.     }
42.
43.     requested.add(wanted.getAll());
44.     reply(REQUEST, wanted);
45. }
46.
47. // Phase 3 - PUSH requested T
48. upon (receive(REQUEST, wanted)) {
49.     Set<T> asked = EMPTYSET;
50.     for (int id : wanted) {
51.         asked.add(getEvent(id));
52.     }
53.
54.     reply(SERVE, asked);
55. }
56.
57. // a message will be delivered only once
58. upon (receive(SERVE, events)) {
59.     for (T t : events) {
```

```
60.         toPropose.add(t.id);
61.         deliverEvent(t);
62.     }
63. }
64.
65. // Aggregation protocol
66. upon (AggregationTimer % aggregationPeriod) == 0 {
67.     Set<Node> peerSubset = selectNodes(getFanout());
68.     for (Node node : peerSubset) {
69.         Set<Capability> fresh = capabilities.get(K);
70.         node.send(AGGREGATION, fresh);
71.     }
72. }
73.
74. upon (receive(AGGREGATION, newCapabilities)) {
75.     capabilities.merge(newCapabilities);
76.     update(_b_, capabilities);
77. }
78.
79. // Fanout adaptation
80. function getFanout() int {
81.     return b / _b_ * f;
82. }
83.
84. // Miscellaneous
85. function selectNodes(int f) Set<Node> {
86.     return f uniformly random nodes from the set of all nodes;
87. }
88.
89. function gossip(Set<int> eventIds) {
90.     Set<Node> peerSubset = selectNodes(getFanout());
91.     for (Node node : peerSubset) {
92.         node.send(PROPOSE, eventIds);
93.     }
94. }
95.
96. function getEvent(int id) T {
97.     return T corresponding to the id;
98. }
99.
100.        function deliverEvent(T t) {
101.            delivered.add(t);
102.            addMempool(t);
103.        }
```

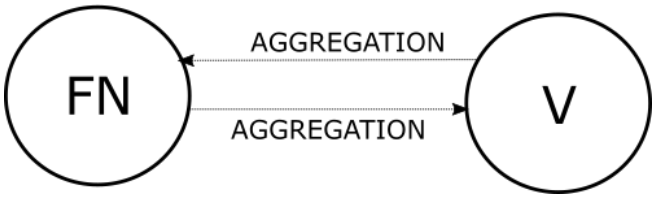


**Figure 3.4.2 – Additional messages exchanged during the HEAP protocol**

- ***FN*** – Standard server (*Full node*)
- ***V*** – Validator server (*Validator node*)

Note that a FN now has information about a set of capabilities – ***Set<Capability>***. It uses this set to update the value of average bandwidth – ***update(Bandwidth, Set<Capability>)***. Policy for updating the average bandwidth is implementation-dependent. Also, the data stored in ***Set<Capability>*** is implementation-dependent. It could be any data of interest to the network (e.g. servers' bandwidths).

In HEAP, apart from exchanging *PROPOSE*, *REQUEST* and *SERVE* messages, nodes periodically exchange ***AGGREGATION*** messages, as shown in Figure 3.4.2. Therefore, during the initialization phase, an ***Aggregation timer*** is started. On every ***aggregationPeriod***, a FN server will gossip information from its ***Set<Capability>*** set to a uniformly selected peer subset. Observe that the fanout is now calculated according to the aforementioned equation, via a call to the function ***getFanout()***. When gossiping information about capabilities, a node will choose ***K*** values from the set. Value for ***K*** is implementation-dependent. Upon receiving an *AGGREGATION* message, a FN merges the new values with the existing ones and updates its average bandwidth value accordingly.

The main role of HEAP is to optimize the utilization of nodes' capabilities (e.g. bandwidth), by adjusting their fanout in correspondence with the selected factors (***b, _b_, ln(n)***). Thus, nodes periodically gossip their information about other nodes' capabilities via *AGGREGATION* messages. This leads to nodes learning about the potential of the network and adjusting the information about that potential (*_b_*). That information is then used when calculating a node's fanout in order to disseminate some information to that node's peer subset.

***iii) PSS***

Finally, we will describe the third part of this solution, which enables it to be applied in a wide area network. So far, it was assumed that a FN node has knowledge of every other node in the network. To overcome this problem, we introduce the PSS (*Peer Sampling Service*) [15].

The fundamental purpose of PSS is to provide every node with a *peer subset*. Until now, the presumption was that a node contains IP addresses of every other node in the network, and when it gossips information, it merely selects ***f*** nodes uniformly at random, hence creating the peer subset implicitly. However, maintaining information about every node in the network has a non-negligible overhead.

PSS is based on avoiding the aforesaid presumption. Its idea is to **gossip the membership information as well**, in the same manner as data is being gossiped. Therefore, this service can be considered **gossip based**. It creates a dynamic unstructured overlay across the network. Experimental analysis in [15] provides promising results when it comes to using PSS. Note that when discussing this problem, we observe two aspects of gossip communication:

- **Peer sampling service** which provides every node with its peer subset – so far this was not gossip based, it was based on drawing a uniformly random sample from the set of all nodes.
- **Gossiping** messages of interest to that peer subset.

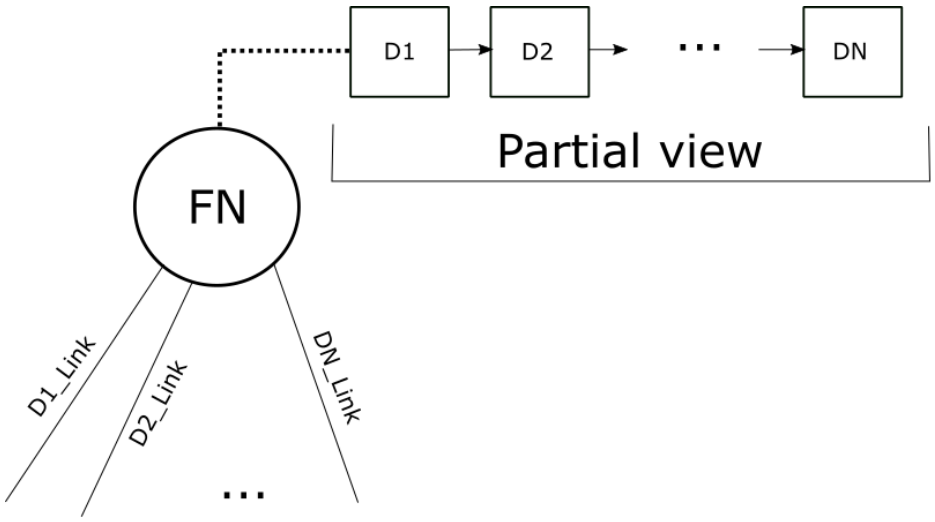


**Figure 3.4.3 – Server structure in the PSS protocol**

- ***FN*** – Standard server (*Full node*)
- ***D*** – Node descriptor

As depicted in Figure 3.4.3 and described in [15], each FN node now contains information about its peer subset, which is in [15] denoted as a node's ***partial view***. *Partial view* is a list of ***node descriptors***, where each descriptor contains a peer's ***IP address*** and a ***hop count*** to that peer. *Hop count* represents the distance between two corresponding nodes expressed in the number of network

hops between them. Partial view has at most one descriptor per peer and is ordered according to increasing hop count. Each partial view is of the same maximum size, which will be denoted as ***c***.

PSS protocol contains two threads:

- **Active thread** → which initiates communication with other nodes
- **Passive thread** → which receives incoming messages

Pseudocode of the PSS protocol is given in the following text. Pseudocode is modified in comparison to the one in [15]. After the pseudocode, an explanation is given.

**Table 3.4.3 – PSS protocol – pseudocode**

```
1.  // Peer Sampling Service - pseudocode
2.  bool push = ...; // at least one is always true
3.  bool pull = ...;
4.  PartialView myPartialView;
5.
6.  // Active thread - code being executed
7.  while (true) {
8.      wait(SOMETIME);
9.      Node p = selectPeer();
10.
11.     if (push) {
12.         // 0 is the initial hop count
13.         Descriptor myDescriptor = new Descriptor(myIPAddress, 0);
14.         PartialView buffer = merge(myPartialView, myDescriptor);
15.
16.         send(p, buffer);
17.     }
18.
19.     if (pull) {
20.         // create a pull request
21.         send(p, EMPTYSET);
22.
23.         PartialView receivedView = receiveViewFrom(p);
24.
25.         increaseHopCount(receivedView);
26.         PartialView buffer = merge(receivedView, myPartialView);
27.         myPartialView = selectView(buffer);
28.     }
29. }
30.
31. // Passive thread - code being executed
32. while (true) {
33.     {Node p, PartialView receivedView} = receiveMessage();
34.
35.     // active thread of some other node has made a pull request
36.     if (receivedView == EMPTYSET) {
37.         // 0 is the initial hop count
38.         Descriptor myDescriptor = new Descriptor(myIPAddress, 0);
39.         PartialView buffer = merge(myPartialView, myDescriptor);
40.
41.         send(p, buffer);
42.     } else {
43.         increaseHopCount(receivedView);
```

```
44.         PartialView buffer = merge(receivedView, myPartialView);
45.         myPartialView = selectView(buffer);
46.     }
47. }
```

The pseudocode is parametrized with two booleans - ***push*** and ***pull***, as well as with two functions - ***selectPeer()*** and ***selectView(PartialView)***. Different strategies for determining these parameters are provided in [13] - p.5. Function ***selectPeer()*** selects a peer to communicate with from the node's partial view. Function ***selectView()*** truncates the buffer ***buffer*** so as to make it of maximum length ***c***. Function ***merge(PartialView, PartialView)*** merges two views it receives as parameters. The resulting view is again ordered by hop count. If the resulting view contains two descriptors for the same IP address, only the one with the lower hop count is included. Function ***increaseHopCount(PartialView)*** increments the value of hop count in each descriptor of the view it received as a parameter.

Active thread periodically initiates communication with a selected peer, based on its strategy – *push*, *pull* or *pushpull* [15]. In case of pushing information, it sends its partial view and its own descriptor to the passive thread of the selected peer. In case of pulling, it creates a ***pull request*** by sending an ***EMPTY_SET*** value to the passive thread of the selected peer, after which it waits for the response.

Passive thread is used to receive information and asnwer to *pull requests* from active threads of other nodes. In case the information was pushed to the passive thread by some other active thread, it is only added to the passive thread's node's partial view in accordance with the selected strategy. In case a pull request was made by another active thread, the passive thread responds with its descriptor and partial view combined.

Therefore, we conclude that by using the PSS we avoid the need for each node knowing about every other node. Nodes periodically exchange information about their peer subsets (partial views), and update them according to chosen strategies.

### *iv) Final solution*

Final solution is based on combining all three protocols described before. It uses push-pull-push gossiping so that only a transaction's identifier is gossiped. It uses HEAP to optimally utilize the capability of a node, without overloading it (by adjusting that node's fanout). It uses PSS to provide a node with its peer subset dynamically, thus creating a dynamic unstructured overlay. Pseudocode for the final solution is given in the text that follows.

**Table 3.4.4 – PPP HEAP PSS final solution – pseudocode**

```
1.  // This is executed on each FN server
2.
3.  // Initialization
4.  Set<Capability> capabilities = EMPTYSET;
5.  Bandwidth b = MY_BANDWIDTH;
6.  Bandwidth _b_ = ...; // average bandwidth
7.
8.  PartialView myPartialView;
9.  initPSS();
10.
11. int f = ln(n); // average fanout
12. Set<int> toPropose = EMPTYSET;
13. Set<T> delivered = EMPTYSET;
14. Set<int> requested = EMPTYSET;
15. start(GossipTimer(gossipPeriod));
16.
17. start(AggregationTimer(aggregationPeriod));
18.
19. // Phase 1 - PUSH T ids
20. upon (receive(PUBLISH, Transaction t))) {
21. // when a C sends T to FN
22.     bool valid = checkTx(transaction);
23.     if (valid == false) {
24.         return;
25.     }
26.
27.     deliverEvent(t);
28.     requested.add(t.id);
29.     toPropose.add(t.id);
30. }
31.
32. upon (GossipTimer % gossipPeriod) == 0 {
33.     gossip(toPropose);
34.     toPropose = EMPTYSET; // Infect and die model
35. }
36.
37. // Phase 2 - PULL wanted T ids
38. upon (receive(PROPOSE, proposed)) {
39.     Set<int> wanted = EMPTYSET;
40.     for (int id : proposed) {
41.         if (!requested.contains(id)) {
42.             wanted.add(id);
```

```
        }
    }

    requested.add(wanted.getAll());
    reply(REQUEST, wanted);
}

// Phase 3 - PUSH requested T
upon (receive(REQUEST, wanted)) {
    Set<T> asked = EMPTYSET;
    for (int id : wanted) {
        asked.add(getEvent(id));
    }

    reply(SERVE, asked);
}

// a message will be delivered only once
upon (receive(SERVE, events)) {
    for (T t : events) {
        toPropose.add(t.id);
        deliverEvent(t);
    }
}

// Aggregation protocol
upon (AggregationTimer % aggregationPeriod) == 0 {
    Set<Node> peerSubset = selectNodes(getFanout());
    for (Node node : peerSubset) {
        Set<Capability> fresh = capabilities.get(K);
        node.send(AGGREGATION, fresh);
    }
}

upon (receive(AGGREGATION, newCapabilities)) {
    capabilities.merge(newCapabilities);
    update(_b_, capabilities);
}

// Fanout adaptation
function getFanout() int {
    return b / _b_ * f;
}

// Miscellaneous
function selectNodes(int f) Set<Node> {
    if (myPartialView.size() > f) {
        return myPartialView.getNodes(f);
    } else {
        return myPartialView.getAllNodes();
    }
}

function gossip(Set<int> eventIds) {
    Set<Node> peerSubset = selectNodes(getFanout());
    for (Node node : peerSubset) {
        node.send(PROPOSE, eventIds);
            }
        }

        function getEvent(int id) T {
            return T corresponding to the id;
        }
```

```
106.
107.        function deliverEvent(T t) {
108.            delivered.add(t);
109.            addMempool(t);
110.        }
```

FN node now contains information about its partial view within its local variable ***myPartialView***. This information is provided by the underlying peer sampling service (PSS protocol), which is also executed on every FN server. Function ***initPSS()*** initializes PSS on a given FN node. Different methods for initializing PSS are described in [15].

**Observe that only the implementation of the *selectNodes(int)* function changes.** This is the function in charge of providing a node with its peer subset. Client's transaction is disseminated to everyone in that peer subset. Peer subset is, in this case, selected according to the optimal fanout calculated by the HEAP protocol. Let us denote the optimal fanout calculated in HEAP as ***f***. Function ***selectNodes(int)*** will always chose a maximum number of ***f*** nodes from the server's partial view. The goal is to optimize the utilization of a server and to avoid overloading it. Alternative would be to simply disseminate the message to everyone in the partial view, even if the size of that partial view exceeded ***f***. Note that a node does not need to have knowledge of everyone in the network (their IP addresses). It only has to know the size of the network, ***n***, in order to calculate ***ln(n)***.

### *3.4.2. Performance analysis*

Let us consider the number of messages exchanged for each client's transaction in the PPP protocol. Let the network be modeled via a graph which has ***e*** nodes. If we consider only those messages that hold a transaction, then for a single client's transaction ***e-1*** messages will be exchanged, which is optimal. Reason for this is that each server sends a *SERVE* message which contains only those transactions that have been explicitly requested. A transaction is requested only once. Therefore, the number of redundant messages that each server receives (and that hold the actual transaction) amounts to 0, which is also ideal.

However, redundancy exists when receiving *PROPOSE* messages. Server may receive a message, from one peer, which holds a transaction identifier that has been previously requested by that server from some other peer. On the other hand, redundancy is a lot smaller in these cases in comparison to the redundancy of the current solution, as the results of simulation will prove.

### *3.4.3. Concluding remarks*

The main drawback of this solution is that it is only probabilistic. It relies on theoretical and experimental results that by using a value for $f = ln(n)$, the network graph is connected with high probability. This probability grows asymptotically toward 1. In our system model, minimum size of the peer subset is 1 and the network graph is always connected. Therefore, theoretically speaking, this algorithm will always disseminate a transaction to every other server in the network, thus satisfying <1> and <2>.

In case of Byzantine or crash failures, some kind of re-transmission would be required. Idea for this is described in [14], where a special timer, ***RetTimer***, is used. Basic idea is that node re-sends *REQUEST* messages for a certain group of transactions until it receives them in a *SERVE* message.

However, there are many advantages to this solution. It is based on a completely dynamic algorithm. It is one hundred percent gossip based. Network is not flooded with transactions, but only with transactions' identifiers. Transaction is sent only after it has been requested by a *REQUEST* message. Delivery is guaranteed to occur only once, therefore there will be no duplicates in a node's RAM memory. This algorithm is possible to implement in a real system, due to the fact that there are no concepts related to it that would require additional research, as was the case with the client-based and tree clustering algorithms.

# 4. SIMULATION ANALYSIS

In this chapter, we will present a detailed explanation of the simulation analysis of the two algorithms described in chapter 3. – current solution used in Tendermint and PPP protocol. The aim of this analysis is to compare these two algorithms based on three different metrics:

- Percentage of redundant messages (***overhead***) that each server in the network receives.
- Average delay when disseminating a transaction.
- Average maximum hop count when disseminating a transaction.

Simulation analysis is done for two particular aforementioned algorithms due to the fact that the PPP protocol could be realistically implemented within the Tendermint.

## 4.1. Simulator setup

Simulator is represented by a concurrent application whose programming code, written in JAVA, could be found at [21]. Since it is a concurrent application, the synchronization of different threads was required. For all the necessary synchronization, semaphores from the ***java.util.concurrent*** package were used. Simulator simulates a network which consists of one client and multiple servers. It is important to observe that the number of clients in the simulation is not important, because the key part is the dissemination of transactions, not the number of clients in the network.

Client sends a fixed number of transactions into the network. Those transactions are disseminated through the network using one of the two algorithms (PPP protocol or the current solution). Simulation is finished when all the servers in the network have received all the transactions and have stored them inside their respective RAM memories.

### *4.1.1. Servers in the simulation*

First and foremost, we will explain how the servers within the simulation operate. Server is represented by an abstract class ***Server***. Each server has two threads (these threads are represented by two inner classes of the ***Server*** class):

- **Active thread** – periodically gossips information to other servers in the peer subset.
- **Passive thread** – processes received requests until all the servers in the network add all the client's transactions into their respective RAM memories.

Active thread periodically (with a period that in the actual simulations had the value of 10*ms*) calls the abstract method ***gossip()***. This method will be implemented within the subclasses of the ***Server*** class – ***CurrentSolutionServer*** and ***PushPullPushServer***. Implementation of this method determines the manner in which server disseminates client's transactions. Active thread stops running when its server has received all the client's transactions and added them to its RAM – in that moment a boolean variable ***activeThreadFinished*** is set to *true*. Programming code of the active thread is given in further text.

**Table 4.1.1 – Active thread of a server**

```
1.  private class ActiveThread extends Thread {
2.
3.      @Override
4.      public void run() { // Active thread periodically disseminates information
5.          while (!activeThreadFinished) {
6.              try {
7.                  Thread.sleep(Constants.GOSSIP_PERIOD);
8.              } catch (Exception e) {
9.                  e.printStackTrace();
10.             }
11.
12.             if (activeThreadFinished) {
13.                 return;
14.             }
15.
16.             gossip();
17.         }
18.     }
```

```
19.
```

Passive thread loops inside an infinite loop. At the beginning of the loop, it waits on a ***requestsEmpty*** semaphore. This semaphore has the initial value of 0. Whenever a client or a peer wants to send a message to a specific server, method ***send(Object)*** of that server will be called. This method receives a request encapsulated within a JAVA ***Object*** class. Inside this method, the request is added to a special buffer, ***requestBuffer***, and it is signaled to the passive thread on the ***requestsEmpty*** semaphore. Upon that signaling, the passive thread can get the request it just received. Therefore, a passive thread, after it has successfully passed the semaphore, retrieves the request from the buffer and processes it via a call to the method ***processRequest(Object)***. This method will be implemented within the subclasses of the ***Server*** class – ***CurrentSolutionServer*** and ***PushPullPushServer***. Implementation of this method determines the manner in which a server processes received transactions. After processing the transaction, the passive thread checks whether that transaction was the last one that should have been received. If it is, the newly received information is gossiped for the final time (*Infect and die* model), the active thread is stopped, and it is signaled on the ***toSignal*** semaphore. This is the semaphore on which the application's main thread is waiting. Initial value of this semaphore is set in a way that the main thread can pass this semaphore only when passive threads of all the servers have signaled on it. Therefore, when the main thread passes this semaphore, it means that all the servers have received all the transactions and that the simulation is finished. After the main thread successfully acquires a permit on the ***toSignal*** semaphore, it sets a boolean variable ***passiveThreadFinished*** of all the servers to *true*, and once again signals on the ***requestsEmpty*** semaphore. This enables all the passive threads in the simulation to be stopped. Programming code of the server's passive thread, as well as the ***send(Object)*** method, are given in further text.

**Table 4.1.2 – Passive thread of a server**

```
1.  private class PassiveThread extends Thread {
2.
3.      @Override
4.      public void run() {
5.          while (true) {
6.              requestsEmpty.acquireUninterruptibly();
7.
8.              if (passiveThreadFinished) {
```

```
                return;
            }

            requestBufferSemaphore.acquireUninterruptibly();
            Object request = requestBuffer.remove(0);
            requestBufferSemaphore.release();

            processRequest(request);

            if (mempool.size() == Constants.NUM_OF_TRANSACTIONS&&!activeThreadFinished) {

                // disseminate for one last time, in order to forward new information
                // everything received from now on will be redundant
                gossip();

                activeThreadFinished = true;
                toSignal.release();
            }
        }
    }
}

// ...

protected void send(Object request) {
    requestBufferSemaphore.acquireUninterruptibly();
    requestBuffer.add(request);
    requestBufferSemaphore.release();

    requestsEmpty.release();
}
```

In further text, the implementation of two subclasses of the ***Server*** class will be discussed – ***CurrentSolutionServer*** and ***PushPullPushServer***. ***CurrentSolutionServer*** is a class which represents a server whose gossiping algorithm is the same as the one currently used in Tendermint. Implementation of the ***gossip()*** method includes sending all those transactions from the server's RAM that have not been disseminated yet, to everyone in the peer subset, by calling their ***send(Object)*** method. Since the server has two threads, an active and a passive one, and since the passive thread receives requests and the active thread disseminates them, implementation of this type of server required adding one buffer – ***toGossip***. This buffer holds transactions that are disseminated to everyone in the peer subset, in accordance with the *Infect and die* model. Passive thread adds transactions to it and the active thread gossips transactions from it. Implementation of the ***processRequest(Object)*** method is equivalent to the pseudocode of the current solution given in chapter 3.1. ***PushPullPushServer*** is a class which represents a server whose gossiping algorithm is the PPP protocol. Implementation of this class is based on the pseudocode of the PPP protocol given in chapter 3.4. Implementation of the ***gossip()*** method includes disseminating ***toPropose*** set.

Implementation of the ***processRequest(Object)*** method is equivalent to the part of the PPP protocol pseudocode which describes the reception of *PUBLISH, PROPOSE, REQUEST* and *SERVE* messages.

### *4.1.2. Client in the simulation*

In order for one simulation to be executed, the application calls the function ***simulate(int, int, SimulationStrategy)***. ***SimulationStrategy*** is an enumeration type that can have two possible values – ***CURRENT_SOLUTION*** and ***PUSH_PULL_PUSH***. Value of the method's third parameter determines whether the dissemination algorithm in the simulation will be the one currently used in Tendermint or the PPP protocol. First parameter of the method represents the number of servers in the simulation. Second parameter of the method represents the number of transactions that will be disseminated through the network.

Inside this method, at the beginning, a semaphore is created. Upon receiving all the transactions into its RAM memory, a server's passive thread will signal on that semaphore. After that, the average size of the peer subset is calculated. Simulations were executed for three different average peer subset sizes – ***ln(n), 2ln(n), 3ln(n)*** where ***n*** is the number of all the servers in the network in the according simulation. This was done with the goal of comparing differences in the metrics' values when increasing the average size of the peer subset.

After that, a method ***initServers(int, SimulationStrategy, Semaphore, int)*** is called. The aim of this method is to create all the servers in the simulation, as well as to initialize their peer subsets. A peer subset is created in a way which guarantees that the network graph is connected, which means that the minimum size of the peer subset is 1. Average size of the peer subset is determined by the fourth parameter of this method.

Client in the simulation is represented by the application's main thread. Before starting all the servers (calling the ***start()*** method of their active and passive threads), the main thread will call a function ***sendTransactions(ArrayList<Server>, int)***. This method will send a fixed number of transactions (this number is represented by the function's second parameter) to a set of randomly chosen servers.

After it has sent all the transactions into the network, the client waits for the simulation to finish on the ***toSignal*** semaphore. When that occurs, passive threads of all the servers are stopped.

Return value of the method ***simulate(int, int, SimulationStrategy)*** is a list of all the servers in the simulation that had just been executed. This is done because after a simulation was finished, all the necessary metrics values calculated throughout the simulation can be retrieved from each one of the servers. Programming code of the ***simulate(int, int, SimulationStrategy)*** is given in further text.

**Table 4.1.3 – Programming code of the client**

```
1.  // function which performs a single simulation
2.  private ArrayList<Server> simulate(int numberOfServers, int numberOfTransactions, Simulati
    onStrategy simulationStrategy) {
3.      Semaphore toSignal = new Semaphore(-numberOfServers + 1, true);
4.      int averagePeerSubsetSize = Constants.PEER_SUBSET_MULTIPLIER * ((int) Math.log(numberO
    fServers));
5.
6.      if (averagePeerSubsetSize <= 0) {
7.          averagePeerSubsetSize = 1;
8.      }
9.
10.     ArrayList<Server> servers = initServers(numberOfServers, simulationStrategy, toSignal,
     averagePeerSubsetSize);
11.
12.     sendTransactions(servers, numberOfTransactions);
13.
14.     startServers(servers);
15.
16.     toSignal.acquireUninterruptibly(); // wait until all the servers receive all the trans
    actions
17.
18.     for (Server server : servers) {
19.         server.finishPassiveThread();
20.     }
21.
22.     return servers;
23. }
```

### *4.1.3. Determining the metrics in the simulation*

Before we present the results of the simulation, we will explain how were the three metrics, mentioned at the beginning of this chapter, calculated. In the simulation, each transaction is described using an object of the ***Transaction*** class. Each transaction has a unique integer identifier. Whenever a transaction is added to the RAM memory of a server, a copy of the corresponding transaction object is created.

*i) Determining the percentage of redundancy (overhead)*

Each server has a local variable, ***numOfRedundantTransactionsReceived***, whose initial value is 0. In case of a server whose gossiping algorithm is the PPP protocol this variable is incremented for each redundant identifier received inside the *PROPOSE* message. In case of a server whose gossiping algorithm is the one currently used in Tendermint this variable is incremented for every reception of a transaction which is already inside the server's RAM memory. Sections of code that update this local variable, followed by an explanation of calculation of the overhead, are given in further text.

**Table 4.1.4 – Determining the overhead**

```
1.  // Current solution
2.  if (isInMyMempol) {
3.      numOfRedundantTransactionsReceived++;
4.
5.      return;
6.  }
7.
8.  // PPP protocol
9.  for (Integer id : proposed) {
10. if (!requested.contains(id)) {
11.         wanted.add(id);
12.         requested.add(id);
13.     } else {
14.         numOfRedundantTransactionsReceived++;
15.     }
16. }
17.
18. // Calculating the overhead
19. public double getOverhead() {
20.     if (this instanceof CurrentSolutionServer) {
21.         return (numOfRedundantTransactionsReceived * 100f) / (numOfRedundantTransactionsRe
    ceived + mempool.size());
22.     } else {
23.         int divisor = Constants.AVERAGE_TRANSACTION_SIZE_IN_BYTES / Constants.IDENTIFIER_S
    IZE_IN_BYTES;
24.         return (((double) numOfRedundantTransactionsReceived / divisor) * 100f) / (((doubl
    e) numOfRedundantTransactionsReceived / divisor) + mempool.size());
25.     }
26. }
```

Overhead is calculated based on the number of redundant **transactions** received, not based on the number of redundant **identifiers** received. Average transaction size is defined by a constant (its value is approximately 550 bytes). Information about the average size of a transaction inside the *Bitcoin* network can be found at [20]. Average size of a transaction in *Bitcoin* corresponds with the average size of a transaction in Tendermint. Size of an identifier is 32 bytes (identifier is calculated as a *SHA256* hash value of the transaction). Therefore, roughly for each 17 identifiers we have a

single transaction. Overhead is calculated in respect to that, depending on the dissemination algorithm used (current solution or the PPP protocol).

### *ii) Determining the average delay*

In order to properly depict a real network, we introduced two different types of links between the servers – LAN and WAN links. These links are represented by the ***Link*** class. Therefore, a peer subset of a server is defined as ***ArrayList<Link>***. When generating the peer subset, within the function ***initServers(int, SimulationStrategy)***, appropriate links are generated as well. In 80% of the cases a link between two servers is of WAN type. In the remaining 20% of the cases a link between two servers is of LAN type. Link between a client and a server is always of LAN type.

In relation with the different types of links that can exist in the simulation, we introduce different delays. Delay is a time expressed in *ms* which represents an interval required for a transaction to get from the starting point of a link to the ending point of a link. On WAN links, delay is generated as a random number in the range [100*ms*, 200*ms*]. On LAN links, delay is generated as a random number in the range [10*ms*, 30*ms*]. Delay is generated when gossiping a transaction and it is represented by an attribute added to the transaction – ***delay***. When it comes to PPP protocol, when generating a delay, it is assumed that the messages which hold the identifier (*PROPOSE, REQUEST*) require less time to transfer from one point to another than the messages which hold an entire transaction (*SERVE*). Manner of determining the delay is given in further text.

**Table 4.1.5 - Determining the delay and the average delay**

```
1.  // Generating the delay in the current solution
2.  int delay = peer.getLinkType().equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nex
    tInt(Constants.MIN_WAN_DELAY, Constants.MAX_WAN_DELAY) : ThreadLocalRandom.current().nextI
    nt(Constants.MIN_LAN_DELAY, Constants.MAX_LAN_DELAY);
3.  t.setDelay(t.getDelay() + delay);
4.
5.  // Generating the delay in the PPP protocol
6.  int divisor = Constants.AVERAGE_TRANSACTION_SIZE_IN_BYTES / Constants.IDENTIFIER_SIZE_IN_B
    YTES;
7.
```

```
8.  int proposeDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextIn
    t(Constants.MIN_WAN_DELAY, Constants.MAX_WAN_DELAY) : ThreadLocalRandom.current().nextInt(
    Constants.MIN_LAN_DELAY, Constants.MAX_LAN_DELAY);
9.  proposeDelay /= divisor;
10.
11. int requestDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextIn
    t(Constants.MIN_WAN_DELAY, Constants.MAX_WAN_DELAY) : ThreadLocalRandom.current().nextInt(
    Constants.MIN_LAN_DELAY, Constants.MAX_LAN_DELAY);
12. requestDelay /= divisor;
13.
14. int serveDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextInt(
    Constants.MIN_WAN_DELAY, Constants.MAX_WAN_DELAY) : ThreadLocalRandom.current().nextInt(Co
    nstants.MIN_LAN_DELAY, Constants.MAX_LAN_DELAY);
15. t.setDelay(t.getDelay() + proposeDelay + requestDelay + serveDelay);
16. // Determining the average delay
17. public double getAverageDelay() {
18.     return ((float) sumDelay) / mempool.size();
19. }
```

Variable ***sumDelay*** represents the sum of delays of all the transactions in the RAM memory of the corresponding server. It is updated whenever a new transaction is added to the server's RAM. Generating the delay in such a manner has been adopted primarily due to the reason that we wanted to compare the average delay in the current solution with the average delay in the PPP protocol.

### *iii) Determining the average maximum hop count*

Hop count represents a path traversed by a transaction on the way from a client to a destination server, expressed in the number of servers on that path. Hop count is represented with an additional attribute added to the transaction – ***hopCount.*** This variable is incremented for every transaction at the moment that transaction is received (added to the RAM memory) of some server. Average maximum hop count reflects the depth of the network graph, since it shows how many servers at most on average did a transaction have to pass in order to reach the destination server. In the following text, a code that depicts the determining of this metric is provided.

**Table 4.1.6 – Determining the maximum average hop count**

```
1.  // updating the transaction local variable
2.  transaction.setHopCount(transaction.getHopCount() + 1);
3.  mempool.add(transaction);
4.
5.  // determining the transaction with the maximum hop count
6.  public int getMaxHopCount() {
7.      int maxHopCount = 0;
8.
9.  for (Transaction t : mempool) {
10.         if (t.getHopCount() > maxHopCount) {
11.             maxHopCount = t.getHopCount();
12.         }
13.     }
14.
15.     return maxHopCount;
16. }
```

## 4.2. Simulation results

Simulations for various parameters, whose results were used to create charts that present the outcome of the simulation analysis, were executed by a call to the ***createMetrics()*** function in the ***Simulator*** class. In order to create the aforesaid charts, libraries *jfreechart* and *jcommon* were used. Simulations were executed for {100, 200, 300, 1000} servers, with a constant number of 200 transactions.

As the results of the simulation somewhat depend on the random number generator in JAVA as well as on the JVM (*Java Virtual Machine*) thread scheduling, for each combination of the parameters {100s, 200t}, {200s, 200t}, {300s, 200t}, {1000s, 200t} three simulations were executed for every algorithm (current solution and PPP). The final value used to create a point on the chart was calculated as an average value of the results obtained from those three simulations. This was done with the goal of eliminating the dependency between the simulation results and the JAVA programming language as much as possible.

As we mentioned earlier, the method ***simulate(int, int, SimulationStrategy)*** has a return value which is a list of servers that consisted the simulated network. This particular list is used when calculating metrics. Average redundancy percentage (overhead) is calculated by dividing the sum of overheads of all the servers by the number of servers. Average delay is calculated by dividing the sum of average transactions' delays on all the servers by the number of servers. Average maximum hop count is calculated by dividing the sum of maximum hop count (for a particular transaction) of all the servers by the number of servers. Based on the calculated metrics, charts are generated.

In further part of this thesis, we will present those charts, followed by a discussion of the calculated results. The *x-axis* shows the number of servers. The *y-axis* shows corresponding metrics – overhead, delay and hop count. Dashed line on a chart represents the current solution algorithm, while the bold line represents the PPP protocol.

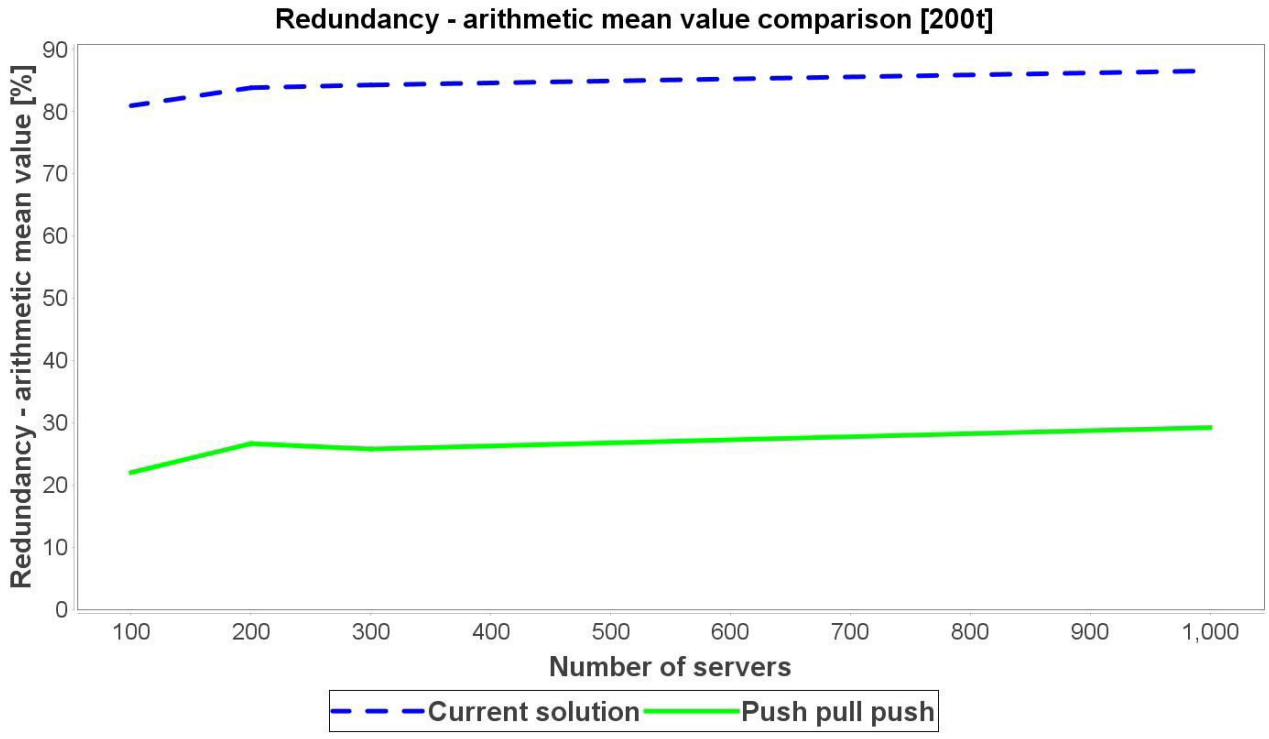


**Figure 4.2.1 – Overhead percentage – average fanout $f = ln(n)$**

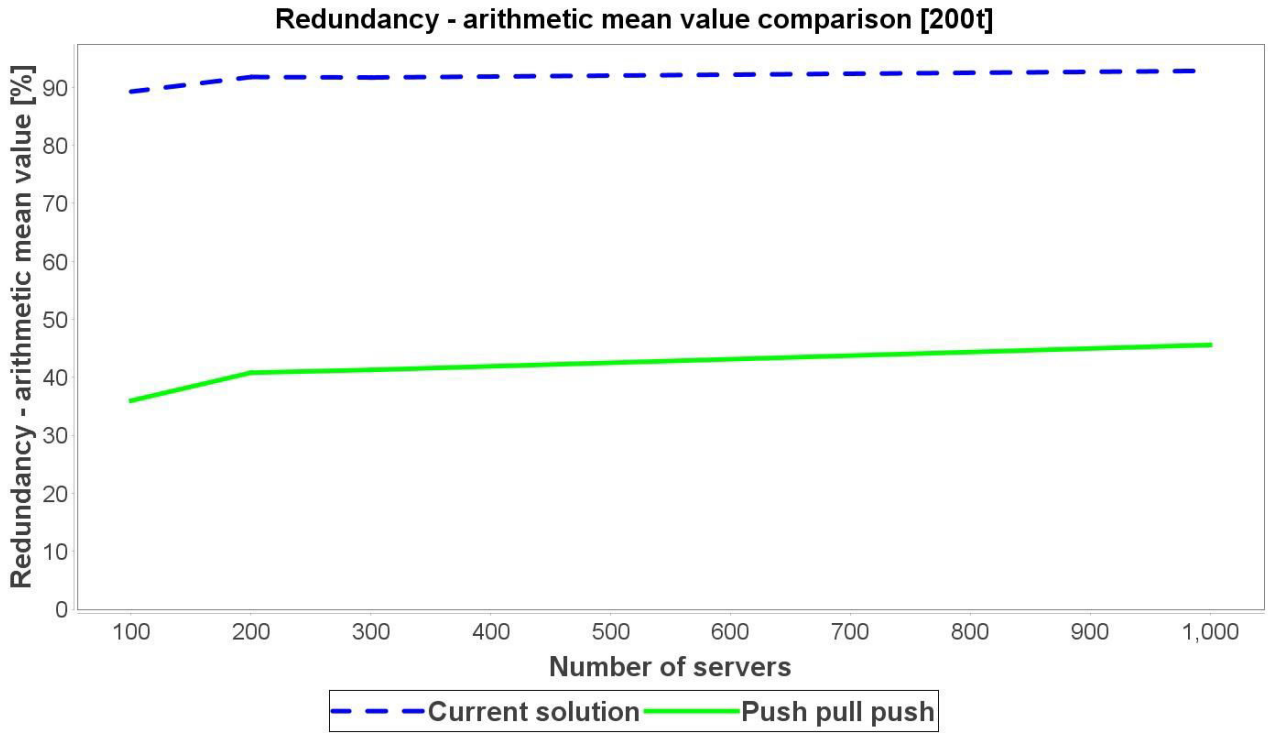


**Figure 4.2.2 – Overhead percentage – average fanout $f = 2ln(n)$**

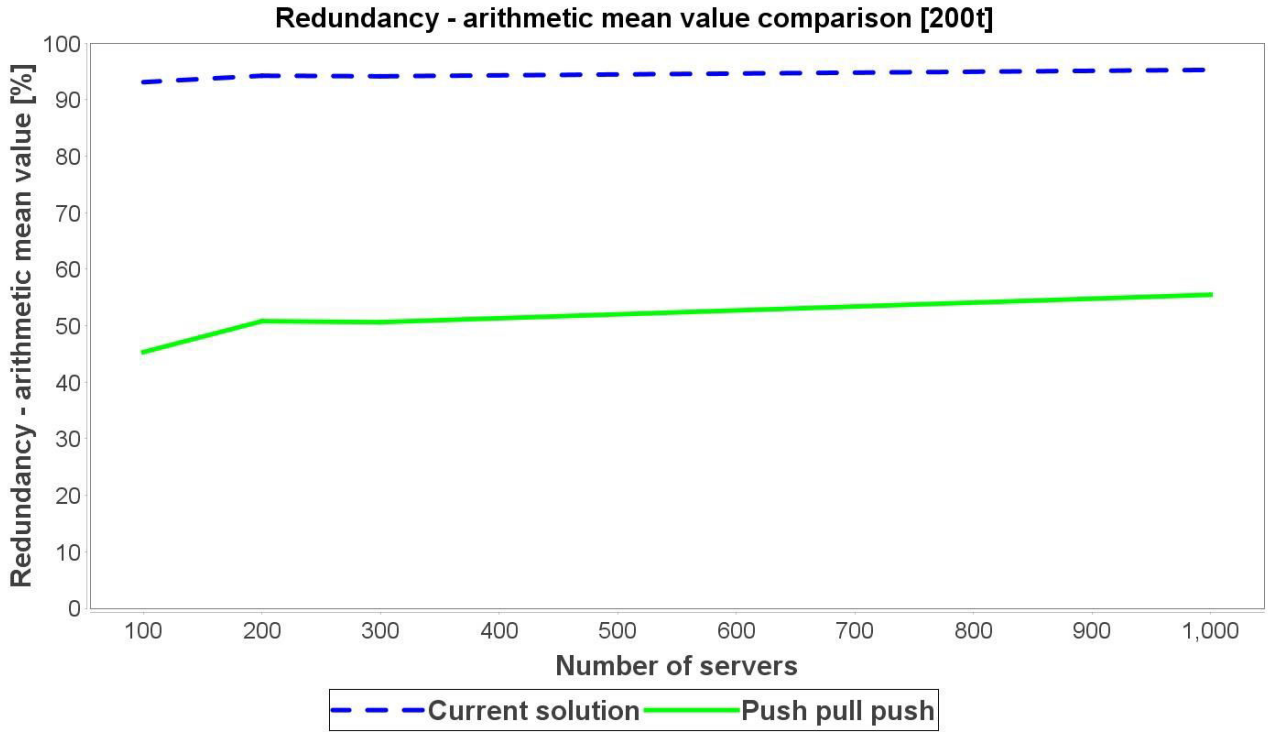


**Figure 4.2.3 – Overhead percentage – average fanout $f = 3ln(n)$**

It is evident that by increasing the average fanout (which means increasing the peer subset size) the overhead increases as well. This is due to the fact that we have more links in the network graph and a message is therefore disseminated through the network more aggressively. On the other hand, the charts show that the redundancy which exists in the current solution is far greater than the redundancy that exists in the PPP protocol, especially for the optimal size of the peer subset – ***ln(n)***.

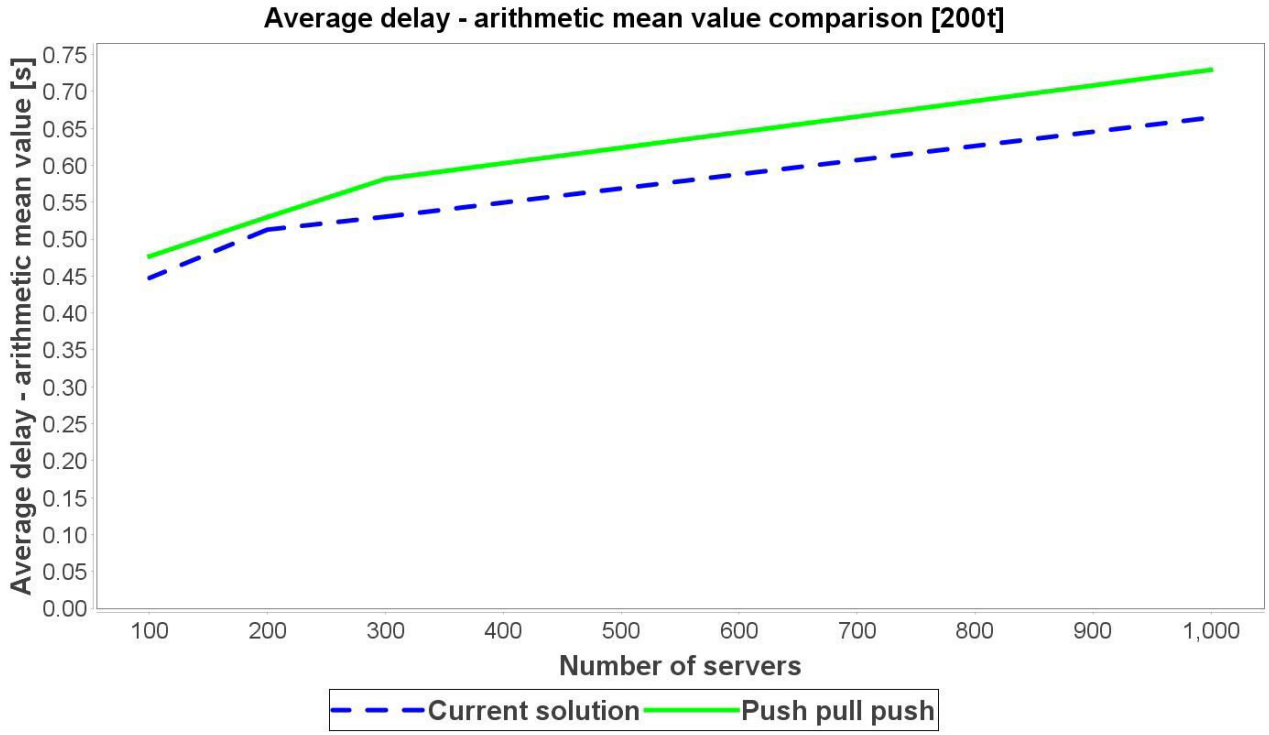


**Figure 4.2.4 – Average delay – average fanout $f = ln(n)$**

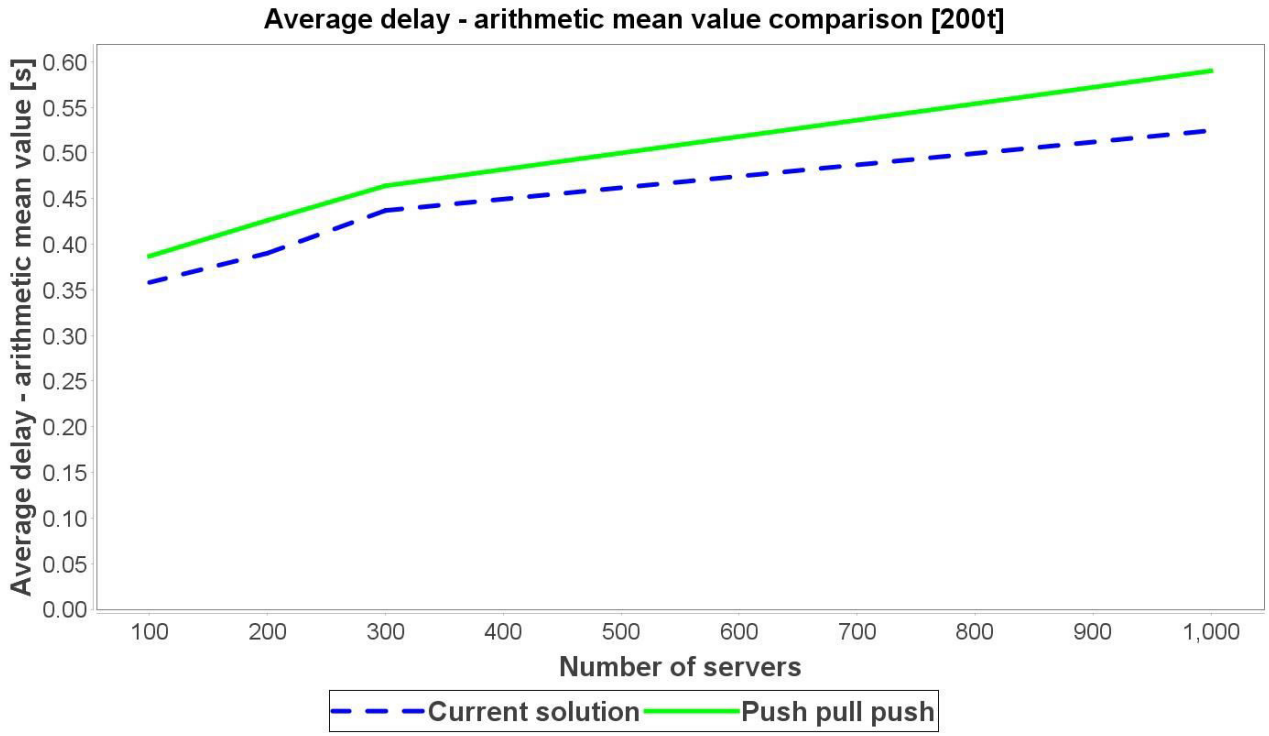


**Figure 4.2.5 – Average delay – average fanout $f = 2ln(n)$**

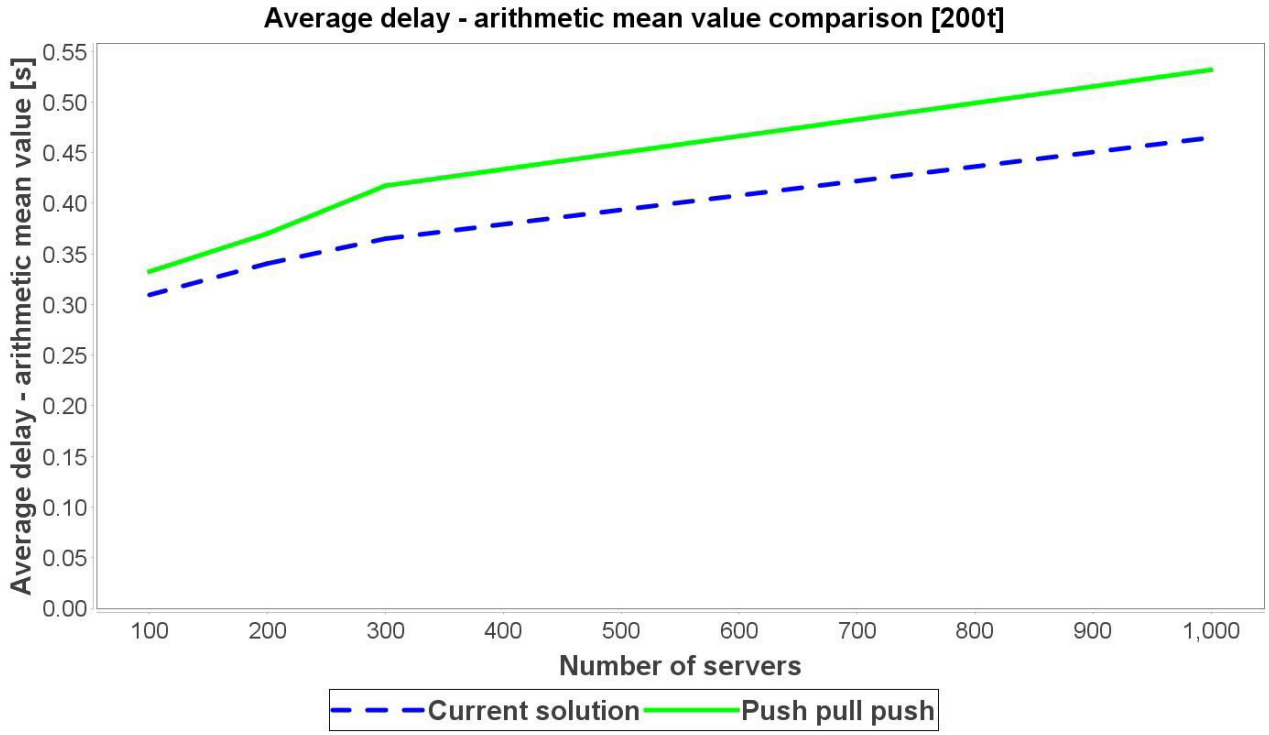


**Figure 4.2.6 – Average delay – average fanout $f = 3ln(n)$**

PPP protocol has a larger delay than the current dissemination algorithm. This is expected, considering that the PPP protocol is a three-way handshake protocol. As we increase the size of the

peer subset, the delay decreases, owing to a larger number of links in the network. When there is a greater number of links in the graph, a transaction can travel through the network faster.

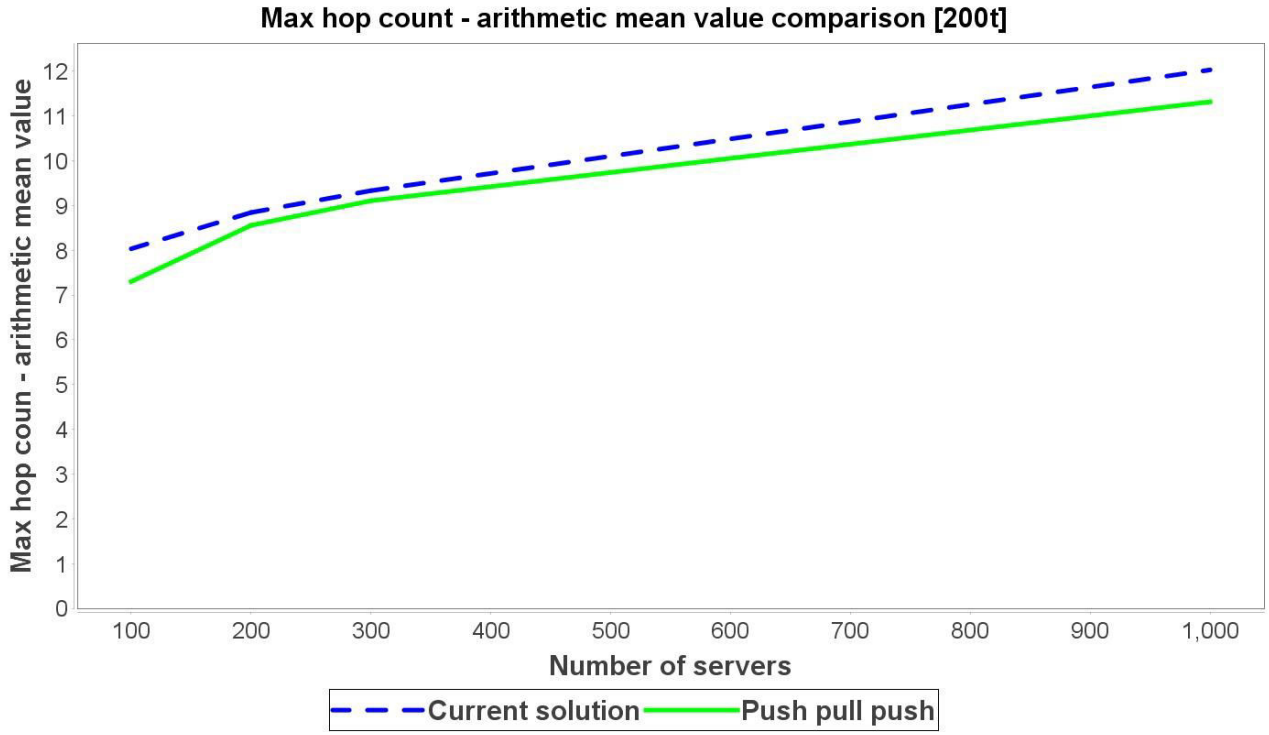


**Figure 4.2.7 – Average maximum hop count – average fanout $f = ln(n)$**

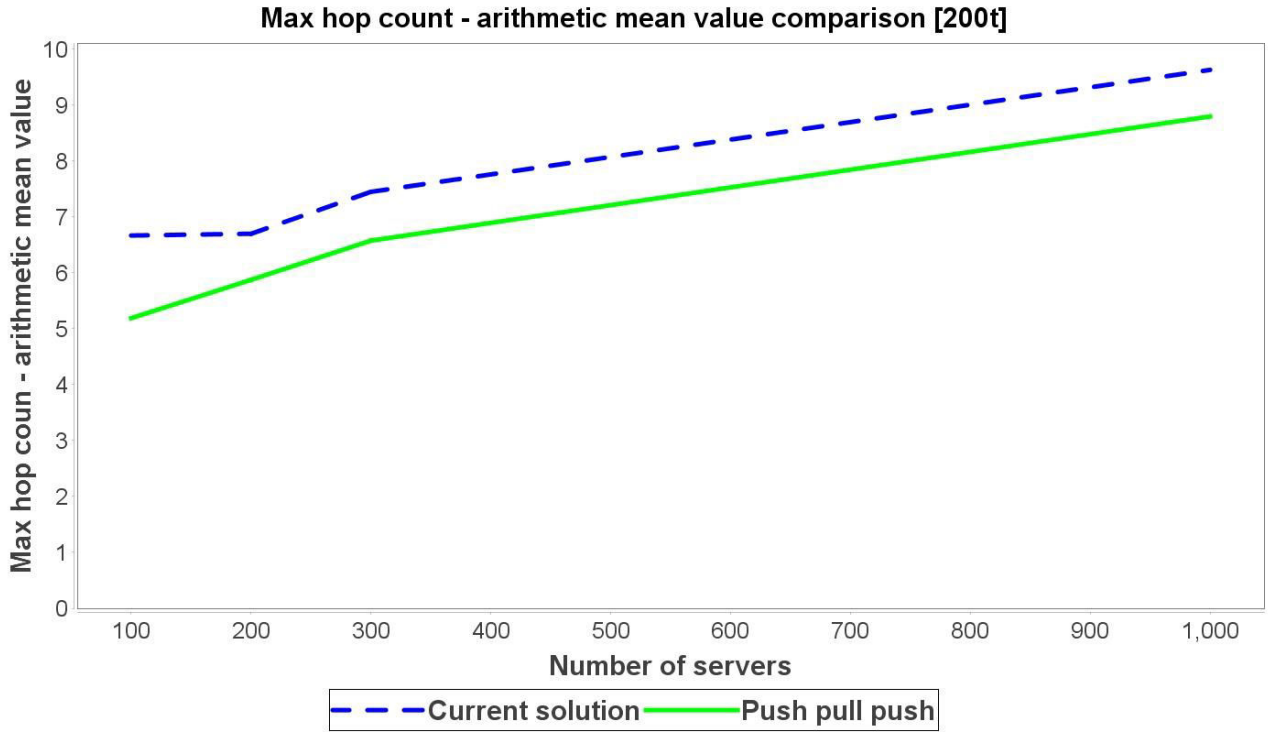

**Figure 4.2.8 – Average maximum hop count – average fanout $f = 2ln(n)$**

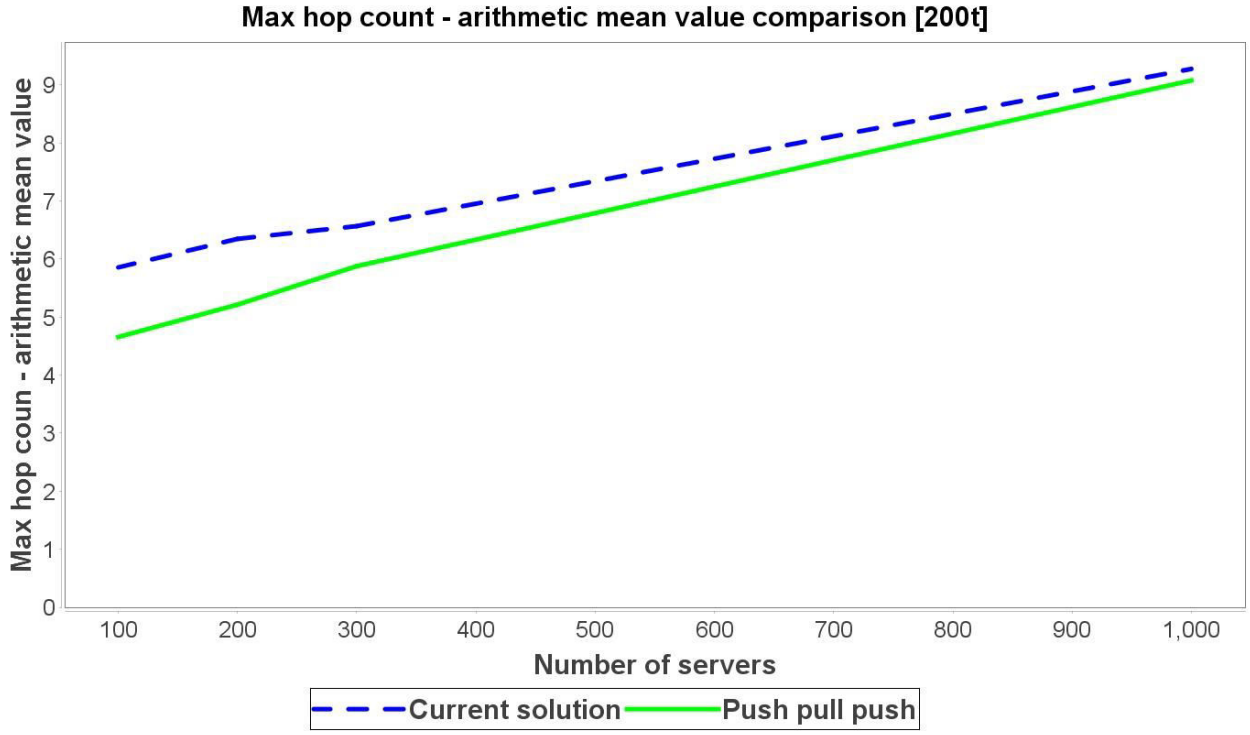


**Figure 4.2.9 – Average maximum hop count – average fanout $f = 3ln(n)$**

When increasing the peer subset size, the maximum average hop count is decreased. This is because we have more links in the network graph, thus the message can quickly reach all of the servers. Average maximum hop count is smaller when using the PPP protocol. This is due to the following reason – in the PPP protocol, a server always sends a *SERVE* message to a peer who sent a *PROPOSE* message. This means that, at the very beginning, when a client sends its transaction to a particular server, that server will send a *PROPOSE* message to everyone in its peer subset. Its peers will demand that transaction explicitly from it, thus making the hop count minimal. This property recursively reflects onto the entire network. Whenever a server receives a transaction, it sends a *PROPOSE* message to everyone in its peer subset, and those peers request the transaction from that particular server.

In the current solution, a message can reach a certain server more quickly through five LAN links than it will through one WAN link. In the PPP protocol, on each of the five LAN links all three phases of the algorithm would be initiated which would increase the total delay on each of those links. Therefore, there is a higher probability that the *PROPOSE* message will travel to a certain server via a lesser number of links.

From everything stated considering the results of the simulation, we may conclude that it is more practical to use the PPP protocol as a gossiping algorithm. First of all, average redundancy is a lot smaller with this protocol, especially for the optimal average peer subset size. As far as the size of the peer subset is concerned, its increase reduces latency and maximum hop count but it results in larger overhead. Perhaps it is possible to find such a maximum value of the peer subset size in the PPP protocol, that the overhead would never exceed a particular threshold (e.g. 50%).

# 5. CONCLUSION

The aim of this thesis was to solve the redundancy problem in a transaction dissemination algorithm. That algorithm is used in a blockchain network by the name of Tendermint.

First approach in solving this problem was a formal analysis of three different algorithms. Each of those algorithms is based on some of the basic approaches that can be found in literature. In two out of three algorithms, it was shown that there is an ideal solution in theory – a solution where there is no redundancy at all. On the other hand, some concepts of those two algorithms are rather challenging to implement. Furthermore, they are a subject of much ongoing research. Third proposed algorithm is possible to implement in a real system. It has very good properties considering it can operate in completely dynamic surroundings and that it is completely based on the principles regarding the epidemic protocols.

Second approach in solving the redundancy problem was a simulation comparative analysis of a current algorithm used in Tendermint and a dissemination protocol which consists of three phases. Simulation analysis has proved that the latter protocol has much better characteristics in comparison to the current solution, under the same simulation conditions. Redundancy is significantly lowered, as is the maximum average path that the transaction travels when being disseminated. Delay has been increased, but this is an acceptable drawback, considering all the advantages that the proposed solution brings to the table.

In further researches, it would be interesting to focus on implementing one of the two algorithms that we deemed challenging to implement. Also, it would be useful to consider an implementation of the third proposed algorithm in an unreliable network. Task would be to make the protocol resilient to network failures and crashes. Additionally, as the current simulation is a concurrent application, it would most definitely be helpful to compare its results to a simulation in a distributed environment. This could be achieved by modifying the current simulator into a distributed application.

# REFERENCES


[1] E. Buchman, J. Kwon, Z. Milosevic, "The latest gossip on BFT consensus", 2018. [Online]. Available: https://github.com/tendermint/spec/releases/download/v0.5/paper.pdf (07.09.2018.)

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system" 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf (07.09.2018.)

[3] M. Crosby, Nachiappan, P. Pattanayak, S. Verma, V. Kalyanaraman, "Blockchain technology - beyond Bitcoin" 2017. [Online]. Available: https://j2-capital.com/wp-content/uploads/2017/11/AIR-2016-Blockchain.pdf (07.09.2018.)

[4] G. Zyskind, O. Nathan, A. Pentland, "Decentralizing privacy: using Blockchain to protect personal data" 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7163223/ (07.09.2018.)

[5] J. Kwon, "Tendermint - consensus without mining" 2014. [Online]. Available: https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf (07.09.2018.)

[6] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals problem" 1982. [Online]. Available: https://dl.acm.org/citation.cfm?id=357176 (07.09.2018.)

[7] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach - a tutorial" 1990. [Online]. Available: https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf (07.09.2018.)

[8] A. Montresor, "Gossip and Epidemic Protocols" 2017. [Online]. Available: http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf (08.09.2018.)

[9] M. Castro, B. Liskov, "Practical Byzantine fault tolerance and proactive recovery" 2002. [Online]. Available: http://www.pmg.csail.mit.edu/papers/bft-tocs.pdf (09.09.2018.)

[10] M. Jelasity, "Gossip-based protocols for large-scale distributed systems" 2013. [Online]. Available: http://www.inf.u-szeged.hu/~jelasity/dr/doktori-mu.pdf (09.09.2018.)

[11] A.M. Kermarrec, L. Massoulie, A.J. Ganesh, "Probabilistic Reliable Dissemination in Large-Scale systems" 2002. [Online]. Available: https://www.researchgate.net/profile/Ayalvadi_Ganesh/publication/2832321_Probabilistic_Reliable_Dissemination_in_Large-Scale_Systems/links/00b7d535646945981f000000.pdf (09.09.2018.)

[12] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments" 2003. [Online]. Available: http://rowstron.azurewebsites.net/PAST/SplitStream-sosp.pdf (09.09.2018.)

[13] D. Frey, R. Guerraoui, A.M. Kermarrec, M. Monod, V. Quema, "Stretching Gossip with Live Streaming" 2009. [Online]. Available: https://hal.inria.fr/inria-00436130/document (09.09.2018.)

[14] D. Frey, R. Guerraoui, A.M. Kermarrec, B. Koldehofe, M. Mogensen, M. Monod, V. Quema, "Heterogeneous gossip" 2009. [Online]. Available: https://infoscience.epfl.ch/record/140640/files/middleware-monod.pdf (09.09.2018.)

[15] M. Jelasity, R. Guerraoui, A.M. Kermarrec and M.V. Steen, "The Peer Sampling Service - Experimental Evaluation of Gossip-Based Implementations" 2004. [Online]. Available: http://lpdwww.epfl.ch/upload/documents/publications/neg--1184036295all.pdf (09.09.2018.)

[16] Tendermint specification - https://github.com/tendermint/tendermint/tree/master/docs/spec (11.09.2018.)

[17] Tendermint public blockchain (Cosmos SDK) - https://cosmos.network/ (11.09.2018.)

[18] Tendermint validator documentation (Validators) - https://github.com/tendermint/tendermint/wiki/Validators (11.09.2018.)

[19] Implementation of the ***checkTx(T)*** function (Mempool.go) - https://github.com/tendermint/tendermint/blob/master/mempool/mempool.go (11.09.2018.)

[20] Average transaction size in the Bitcoin network - https://tradeblock.com/bitcoin/historical/1w-f-tsize_per_avg-01101 (11.09.2018.)

[21] Programming code for the simulator - https://github.com/lukamiletic95/simulator (11.09.2018.)

# ABBREVIATIONS LIST

HEAP – *Heterogeneity aware protocol*

IP – *Internet protocol*

JVM – *Java Virtual Machine*

LAN – *Local area network*

P2P – *Peer to peer*

PPP – *Push-pull-push*

PSS – *Peer sampling service*

RAM – *Random access memory*

SMR – *State Machine Replication*

SPOF – *Single point of failure*

WAN – *Wide area network*

# FIGURES LIST

# TABLES LIST