Individual Diploma Thesis

IMPLEMENTATION AND EXPERIMENTAL EVALUATION OF THE HASHGRAPH BFT RANDOMIZED ALGORITHM

Georgios Papaioannou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Implementation and Experimental Evaluation of the Hashgraph BFT Randomized Algorithm

Georgios Papaioannou

Supervisor

Chryssis Georgiou

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements of obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2021

Abstract

Ledgers are distributed databases and blockchain is a specific type of distributed database which stores data in blocks that are then chained together. The above databases are built in a decentralized way so that no single person or group has control. This makes it more robust on cyber-attacks.

Byzantine fault tolerance is the property of a distributed system that is able to resist faulty behavior of nodes and continue operating. Ledgers and blockchains must have the BFT property because they are distributed and nodes in the system may have arbitrary behavior. To implement this property the system must be based on a Byzantine Fault Tolerance algorithm.

Implementing BFT algorithms has been under research for many years in an attempt to implement new efficient solutions or improve the existing suggested algorithms. Randomized algorithms has been proposed to be simpler and more efficient than deterministic algorithms. Hashgraph is one of them.

In this thesis we implemented the Hashgraph BFT algorithm in the Go programming language using the ZeroMQ framework for the communication between the nodes. We then conducted an experimental evaluation of the implementation's performance.

Contents

Chapter **1** **Introduction**.....**1**

1.1 Motivation 1

1.2 Objective 2

1.3 Methodology 2

1.4 Document Organization 3

Chapter **2** **Background**.....**4**

2.1 Byzantine Fault Tolerance 4

2.2 System Model 5

2.2.1 Byzantine replicas 5

2.2.2 Overview 5

2.3 Common Applications of BFT 6

2.4 The ZeroMQ Communication Library 6

2.5 The Go Programming Language 6

2.5.1 Overview 6

2.5.2 Concurrency 7

2.6 BFT Algorithms 7

Chapter **3** **HashGraph**.....**9**

3.1 Overview 9

3.2 The Gossip Protocol 9

3.3 See and Strongly See 10

3.4 Protocols 11

3.4.1 Divide Rounds 11

3.4.2 Decide Fame. 12

3.4.3 Find Order 12

Chapter **4** **Implementation Basics**.....**13**

4.1 Execution 13

4.2 Encryption and Hashes 14

4.3 Structs and Variables 14

4.3.1 Gossip Message 14
4.3.2 HashGraph Node 15
4.3.3 Helpful Structs and Variables 15
4.4 Channels and Messages 16
4.4.1 Nodes Channels & Messages 16
4.4.2 Client Channels & Messages 17
4.5 Sign and verification 17
4.6 Sorting Events 18
4.7 Algorithm Initialization 18
4.8 Comparing EventMessages 18
4.9 Client 18

Chapter **5** **Go Routines**……..……..……**20**
5.1 Send Gossip 20
5.2 Manage Client Requests 21
5.3 Manage Incoming Gossip 22
5.3.1 Check Gossip 23
5.3.2 Executing the Algorithms 24
5.4 Protocols 25
5.4.1 Divide Rounds 25
5.4.2 Decide Fame 26
5.4.3 Find Order 27

Chapter **6** **Implementation Details and Decisions**…...……**30**
6.1 Message Size 30
6.2 Timestamps 31
6.3 OrphanEvents 31
6.4 Comparing Transactions 31
6.5 Sorted Events list 32
6.6 Inserted Bounds 32
6.7 HashGraph Size 33
6.8 Messages Exchanged 33
6.9 Empty Transactions 33

Chapter **7** **Evaluation**…..……...……**34**

7.1 Scenarios 34

7.2 Evaluation Summary 35

7.3 Comparison with other Works 41

7.4 Conclusions 42

Chapter **8** **Conclusions**….……...……**43**

8.1 Summary 43

8.2 Notable Difficulties 43

8.3 Suggested Improvements 44

Bibliography……..**46**

Appendix A…….……..…….……....**A-1**

Appendix B…….……..….…….……...**B-1**

Appendix C…….……..….…….……...**C-1**

Chapter 1

Introduction

1.1 Motivation 1
1.2 Objective 2
1.3 Methodology 2
1.4 Document Organization 3

1.1 Motivation

Nowadays, the wide use of online systems makes malicious attacks more attractive and the consequences of a successful attack have become more serious. Therefore, implementing a service that successfully defends against such attacks is crucial. Solving such a problem requires time and resources and this subject has been under research for many years.

Online systems provide services to clients. One way to implement a service is using a centralized server but this is prone to cyber attacks as it is a single point of failure. For that reason, providing the service in a distributed way may solve the problem. However, some of the servers may have arbitrary behavior. Therefore, we use the approach of state machine replication [6], meaning that we have multiple servers providing the same service where they communicate with each other so the non-faulty eventually has consistent state.

BFT [7] is a property of a system where all the non-faulty nodes have eventually consistent state despite the existence of malicious nodes. Therefore, distributed systems must have the BFT property. A system has that property when each server executes a BFT algorithm.

Implementing a robust BFT algorithm may have negative results in terms of latency, CPU usage and throughput. Therefore, there are many attempts to design fault-tolereant algorithms with good performance. An idea that may increase the performance of BFT

algorithms is the use of randomization where is expected to create simpler algorithms with better performance than deterministic ones.

Therefore, we want to implement a randomized algorithm and experimentally evaluate its performance. We also seek to compare our experimental results with similar BFT algorithms and draw useful conclusions.

1.2 Objective

The objective of this thesis is the implementation and experimental evaluation of Hashgraph [5] in the Go programming language [13] using the ZeroMQ framework [14] for the communication between the nodes. Hashgraph is a randomized BFT algorithm, which we have implemented, executed in different scenarios and evaluated its performance.

1.3 Methodology

We separated the thesis in different stages. In the first stage, we studied research papers on distributed systems and Byzantine Fault Tolerance to get more familiar with those terms. Then we studied some several randomized BFT algorithms and chose one of them to implement. The second was divided into two parts, (a) the implementation of the Hashgraph algorithm and (b) its experimental evaluation.

Research on Distributed Systems and BFT

To become more familiar and acquire knowledge relative to distributed systems we attended the Computer Science course CS432-Distributed Algorithms. We also used online resources to deepen our understanding of notions like state machine replication [6], Byzantine faults [7] and BFT algorithms [8]. As a first step, we studied the simpler form of the problem, namely the “Two General’s Problem [7].

Research on BFT algorithms

First, we studied the PBFT algorithm [2] and an implementation of a variant of PBFT [1], which is one of the first solutions for partially asynchronous networks and is deterministic.

Then we studied in depth variants of randomized BFT algorithms. In particular we studied the protocols presented in [10,11,5,12,4].After studying the above algorithms, we concluded that the HashGraph algorithm [5] is a promising suggestion and that we would proceed with the implementation of this algorithm.

Development

We decided to implement the algorithm in the Go programming language [13] using the ZeroMQ framework [14] for the communication between the replicas and with the clients. We used the official Go website [13] to download and install Go and learn more about Go libraries. For practicing Go we used the Go official website, tutorials on YouTube and made small exercises on variables and constants, arrays and slices, maps, structs, conditional statements, loops, pointers, functions, goroutines and channels. We studied the basics of ZeroMQ using its official websites [14] and the provided Guide [15], where we implemented some of the simple examples of client-server communication.

The implementation of [1] was very helpful. Also very helpful was the implementation in [3]. Both implementations in [1] and [3] are in the Go programming language and use the ZeroMQ framework.

We developed the algorithm on a Linux operating system using the VS Code source code editor. The source code of our implementation both the server-side and the client-side, can be found on GitHub [16].

After implementing the algorithm, we performed testing locally to check its correctness. We proceeded with the experimental evaluation where we used a cluster of 9 machines. At the end we compared our results with other implementations of randomized BFT algorithms.

1.4 Document Organization

In Chapter 2, we make a background overview on Byzantine Fault Tolerance, the system model and the implementation language. In Chapter 3, we describe the Hashgraph algorithm in simple words and its core principles. In Chapter 4, we present the structs used to implement the algorithm, how we used them and some important functionalities. Then in Chapter 5, a detailed explanation of how we implemented the Hashgraph algorithm is given by describing its core protocols. Next, in Chapter 6, we explain the reasons behind some implementation decisions and analyze some algorithm characteristics. Chapter 7 describes the scenarios we developed to validate the algorithm's correctness, demonstrating the algorithms performance and compares it with other works. Chapter 8, contains our conclusions, notable difficulties and ideas that may improve algorithm's performance.

Chapter 2

Background

2.1 Byzantine Fault Tolerance 4
2.2 System Model 5
2.2.1 Byzantine replicas 5
2.2.2 Overview 5
2.3 Common Applications of BFT 6
2.4 The ZeroMQ Communication Library 6
2.5 The Go Programming Language 6
2.5.1 Overview 6
2.5.2 Concurrency 7
2.6 BFT Algorithms 7

2.1 Byzantine Fault Tolerance

BFT [6,7] is under research for many years and efficient algorithms with high throughput and low latency are in demand especially now with the popularity of the cryptocurrencies.

Online systems are expected to always operate, be reliable and have excellent performance. Systems that use BFT algorithms might be a cryptocurrency, a service on a spacecraft or a Boeing 777, therefore the implementation of an excellent solution is critical.

BFT is a systems property which states that the servers in a distributed system have eventually consistent state despite the existence of nodes with arbitrary behavior.

There are different scenarios where we use BFT algorithms and this is depending on the system model. When there is a public-blockchain we need a more robust and probably weaker algorithm in matter of performance because the number of participants is not known and anyone can join [8]. Compared to private networks we need different and possibly simpler algorithms because we know the identities of all the nodes who participate.

In this thesis we focused on a private network where the communication between the nodes is asynchronous.

BFT algorithms must be built on the Atomic broadcast which must satisfy the following properties [5]:

Agreement: If any node outputs a transaction, then every node outputs that transaction.

Total Order: For all $i > 0$, if T is the i-th output of a non-faulty node and T' is the i-th output of another non-faulty node, then $T = T'$.

Liveness: If a transaction is input to a non-faulty node, then it is eventually output by every non-faulty node.

2.2 System Model

2.2.1 Byzantine replicas

Some of the replicas in the system may have Byzantine (faulty) behavior; these faults are considered to be the most general and most difficult category of failures [7]. The Byzantine replicas may delay sending messages, omit sending messages, send messages with arbitrary content or communicate with some nodes and not others. The malicious nodes may collude and we assume that they are coordinated by a single adversary. However, we assume that they cannot produce a valid signature of a non-faulty node or find two messages with the same digest (hash value).

2.2.2 Overview

We consider a fully asynchronous distributed system where nodes are connected by a network. The system makes no timing assumptions and there is no upper bound on the communication delay. The network may fail to deliver messages, delay them, duplicate them or deliver them out of order. Algorithms are designed for a state machine replication system [6] that tolerates Byzantine faults [7] where at most $\left\lfloor \frac{n-1}{3} \right\rfloor$ out of n replicas may have arbitrary (Byzantine) behavior. Each node receives a transaction from a client or another node and by communicating with each other has to determine when to write each transaction in its local storage unit.

2.3 Common Applications of BFT

By observing common applications of BFT algorithms we can clearly see and understand why an implementation of a correct and efficient protocol is so important and has been under research for the last 20 years.

Especially nowadays, cryptocurrencies are a main topic and many people invest their money in such currencies. An example is Bitcoin [17], whose network works in parallel to generate a blockchain with proof-of-work [18] allowing the system to overcome Byzantine failures and reach a coherent global view of the system's state.

Another application is on aircraft systems and spacecrafts which are real time systems and the BFT solutions must have the least latency, close to microseconds.

We can clearly see that if the faulty nodes manage a successful attack or the system does not perform efficiently, the results might be catastrophic.

2.4 The ZeroMQ Communication Library

ZeroMQ (also spelled ØMQ, 0MQ or ZMQ) [14] is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. Its goals is zero administration, zero cost, and zero waste. It is developed by a large community of contributors and is available in more than 28 programming languages.

We used the implemented communication from the SS-BFT implementation [1] and modified some small parts for our project. Executing the algorithm each node has 2 sockets for each client and other node in the system, one is responsible to receive messages and the other one to transmit our messages.

2.5 The Go Programming Language

2.5.1 Overview

Go [13] is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing and CSP-style concurrency. It was designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases. The designers wanted to address criticism of other languages in use at Google but keep their useful characteristics such as

static typing and run-time efficiency (like C), readability and usability (like Python or JavaScript) and high-performance networking and multiprocessing.

2.5.2 Concurrency

It is said that one of the most exciting aspects of learning the language is its concurrency model. Go's concurrency primitives make creating concurrent, multi-threaded programs simple and fun. Go routines are a very simple and easy way to implement concurrency. Go routines can be thought of as lightweight threads. We can create a go routine by just adding the go keyword to the start of calling a function, then the function is executed in the background and the execution continues. Goroutines are divided onto small numbers of OS threads. They exist only in the virtual space of go runtime. Go has a segmented stack that grows when needed. That means it is controlled by Go runtime, not OS. Also, Goroutines are started faster than threads and are created with only 2 KB stack size compared to java threads which takes about 1 MB stack size. Therefore, Go seems an excellent option.

2.6 BFT Algorithms

PBFT [2] is a partially asynchronous algorithm which makes synchrony assumptions for correctness. A group of processors serves a group of clients by replicating the client's requests. At any given time, exactly one processor is assigned to be the primary processor. The client requests a service operation by sending a message to the primary which broadcasts the client's request to the rest processors. The processors executes the requests and reply to the client with the result.

SINTRA [12] is a randomized ABFT algorithm suggested in 2002. It contains broadcast primitives for reliable and consistent broadcasts, which provide agreement on individual messages sent by distinguished senders. Total order is achieved by using multiple randomized Byzantine agreement protocols for binary and multi-valued agreement and implements an atomic broadcast channel on top of agreement. Threshold cryptography is a fundamental concept in SINTRA, as it allows the group to perform a common cryptographic operation for which the secret key is shared among the servers such that no single server or small coalition of corrupted servers can obtain useful information about the key.

Honeybadger [10] is an algorithm closely related to SINTRA [12] which in its threshold encryption scheme, any one party can encrypt a message using a master public key. It requires *f+1* correct nodes to compute and reveal decryption shares for a ciphertext before the plaintext can be recovered. It is based on agreement on a random batch of incoming transaction.

BEAT [11] is consisting of five asynchronous BFT protocols that are designed to meet different goals. Three of the five protocols are applied on general state machine replication and the other two focus on BFT storage. It aims to outperform Honeybager [10] by implementing more efficient threshold encryption and optimizing broadcast protocols.

"From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures" [4] is a signature free BFT algorithm which provides atomic broadcast by implementing a vector consensus above a multi-value consensus with the help of an underline binary consensus and reliable broadcast.

We now proceed to overview the implemented algorithm, Hashgraph [5].

Chapter 3

HashGraph

3.1 Overview 9
3.2 The Gossip Protocol 9
3.3 See and Strongly See 10
3.4 Protocols 11
3.4.1 Divide Rounds 11
3.4.2 Decide Fame 12
3.4.3 Find Order 12

3.1 Overview

Hedera [19] is a public distributed ledger for building and deploying decentralized applications and microservices. Those applications and microservices execute underline the Hashgraph consensus algorithm. It is a randomized ABFT (Asynchronous Byzantine Fault Tolerance) algorithm suggested by Leemon Baird in 2016 and it is based on two techniques to reach consensus, “Gossip about Gossip” and virtual voting. It works in a private, permission-based setting means the identities of all nodes are known beforehand and the network it is not open to an arbitrary participant.

Hashgraph aims to improve throughput and latency compared to prior algorithm such as HoneyBadgerBFT [10] and BEAT [11] by executing broadcast and voting simultaneously. Compared to public blockchains (like bitcoin), those use Proof-of-Work (PoW)/Proof-of-Stake (PoS) as consensus model while Hashgraph uses the Gossip Protocol and Virtual Voting. It also expected to be faster, more efficient, cheap and fair.

3.2 The Gossip Protocol

HashGraph nodes use the gossip protocol to communicate. In this protocol each node chooses randomly another node and sends it all his events which he inserted in its graph. This mean it sends all the events it created and all the events that other nodes told him that they created. In Figure 3.1 we can see the structure of the message that is sent between the nodes, each message contains the signature of the node who created it, a

timestamp which is the time that the creator claims that it created it, the transaction and two hashes. The hashes point to the previous event in the graph and to the parent event. In Figure 3.2 we can see how the Hashgraph data structure is created by exchanging messages. Each line corresponds to each node, for example, if we are Alice in "Alice" line is the events that we created and in the other lines it's the events we learned with gossip protocol. The previous hash is the hash of the previous event in the Hashgraph line and it is the vertical line between events. Parent hash is the hash of the parent event, meaning the event from which we learned for the first time a transaction and is connected with the diagonal line.

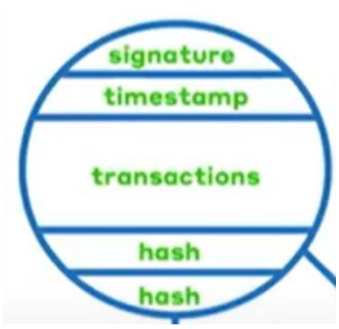


Fig3.1: Gossip Event [20]

Fig3.2: An example of the Hashgraph Data Structure [20]

3.3 See and Strongly See

The HashGraph data structure is based on the *see* and *strongly see* property to work correctly and handle malicious nodes. We will use the terms parent, self-parent, ancestor and self-ancestor.

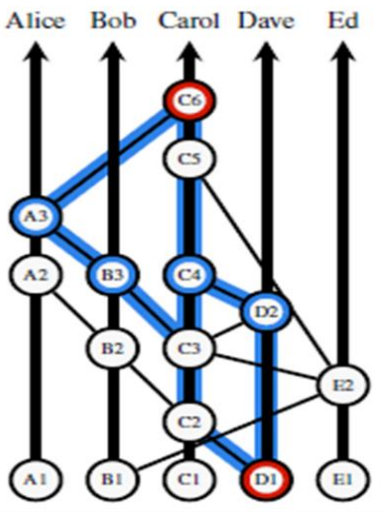


Fig3.3: The Hashgraph Data Structure [5]

In Figure 3.3 the event A2 is self-parent of A3 and B3 is parent of A3.

For example, the event A3 will contain the hash of event A2 and event B3.

An event *x* is ancestor of event *y* if *x* is *y* or a parent of an ancestor of *y*. Ancestor and see relation is the same thing and we can see the pseudocode of this property in Figure 3.4.

For example, in Figure 3.3 ancestors of A3 are the events A2, A1, B3, B2, B1, C3, C2, C1, D1, E2, E1.

```
ancestor(x,y)bool{
    if x==y
        return true

    return ancestor(x,y.parent)
        || ancestor(x,y.self-Parent)

}
```

Fig3.4: Ancestor Pseudocode

Strongly see enhances the seeing property by having many other nodes "vouch for" another vote. An event *x* can strongly see *y* if it can see (ancestor) event *y* and passes through *2n/3* events by different nodes. For example, in Figure 3.3 event C6 can strongly see D1. A more detailed implementation of see and strongly see is given in Appendix A.

3.4 Protocols

While receiving events the algorithm executes the following operations to determine the events order. The operations are: "divide rounds", "decide fame", and "find order".

3.4.1 Divide Rounds

It divides the Hashgraph into rounds by giving a round number to each event. Each event has round number *R* which is the max round number of its parents(self-parent and parent) and if it can strongly.

see more that *2n/3* round *R* events it has round *R+1*. If it's the first event that it created with round *R+1* in that Hashgraph line it is a round *r+1* witness.

Algorithm 2: The `divideRounds` procedure.

foreach *event* x **do**
  if x *has parents* **then**
    $r \leftarrow$ max round of parents of x
  else
    $r \leftarrow 1$
  if x *strongly sees* $> 2n/3$ *round* r *witnesses* **then**
    x.round $\leftarrow r + 1$
  else
    x.round $\leftarrow r$
  x.witness $\leftarrow$ (x has no self parent) or (x.round > x.selfParent.round)

Fig3.5: The Divide Rounds Pseudocode [5]

Figure 3.5 shows the divide rounds pseudocode from the Hashgraph paper [5]. The else statement when *r* is set to 1 it is only happening to the very first events on each Hashgraph line. Similarly, the "*x* has no self-parent" because all the events that inserted have self-parent except the very first which is the initialization of the graph.

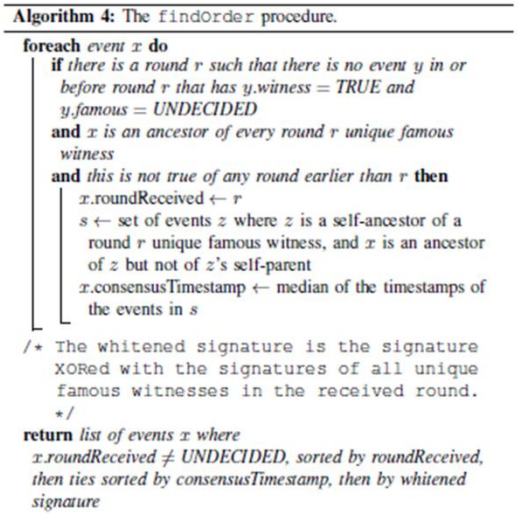
3.4.2 Decide Fame

For every 2 consecutive rounds where the round is decided we take the witnesses of those rounds. Then we have to determine which one of the witnesses in the smaller round is famous. For each witness in the smaller round we count how many witnesses from the next round can strongly see them. if it can be strongly seen by *2n/3* round *R+1* witnesses it is considered famous.

3.4.3 Find Order

For each round in which we decided who are the famous witnesses and there is not a round before where is a witness we did not decide if its famous.

If there is an event which is ancestor(see relation) from each round *R* famous witness and round *R* is the first round that this happens for this event. We set the event roundReceive as *R*, we find all the events which *x* is an ancestor, have round < *R* and take the timestamps of those events. We set the consensus timestamp of this event the median of those timestamps. Median is the timestamp in the middle position in the array, the idea is that we set consensus time, the time a transaction has reached half the world. Figure 3.6 shows the findOrder algorithm.


Algorithm 4: The `findOrder` procedure.

foreach *event* x **do**
  if *there is a round* r *such that there is no event* y *in or before round* r *that has* $y.witness = TRUE$ *and* $y.famous = UNDECIDED$
  and x *is an ancestor of every round* r *unique famous witness*
  and *this is not true of any round earlier than* r **then**
    x.roundReceived ← r
    s ← set of events z where z is a self-ancestor of a round r unique famous witness, and x is an ancestor of z but not of z's self-parent
    x.consensusTimestamp ← median of the timestamps of the events in s

`/* The whitened signature is the signature XORed with the signatures of all unique famous witnesses in the received round. */`

return *list of events* x *where* $x.roundReceived \neq UNDECIDED$, *sorted by roundReceived, then ties sorted by consensusTimestamp, then by whitened signature*

Fig3.6: The findOrder pseudocode [5]

Chapter 4

Implementation Basics

4.1 Execution 13
4.2 Encryption and Hashes 14
4.3 Structs and Variables 14
4.3.1 Gossip Message 14
4.3.2 HashGraph Node 15
4.3.3 Helpful Structs and Variables 15
4.4 Channels and Messages 16
4.4.1 Nodes Channels & Messages 16
4.4.2 Client Channels & Messages 17
4.5 Sign and verification 17
4.6 Sorting Events 18
4.7 Algorithm Initialization 18
4.8 Comparing Event Messages 18
4.9 Client 18

4.1 Execution

We can execute the algorithm with the command "HashGraphBFT $ID $N $CLIENTS $SCE $REM" where ID is the node id, N is the total number of nodes in the system, client is the total number of clients, scenario is which scenario to be executed(0-3) and rem refers to local execution or remote where we set the IP of the remote machines.

Then, it calculates the number of faulty nodes($f = \frac{N-1}{3}$), the threshold($T = N\text{-}f$) and opens a TCP connection to each client and node. Then it initializes the Hashgraph implementation and executes the go routines sendGossip, manageClientRequest and Manage incoming Gossip which are explained below.

4.2 Encryption and Hashes

Each node has a key pair for digital signatures and all nodes know the public key for each other node. The Threshenc package implements the key creation and the hashes. The nodes must get the public key of each node from a trusted dealer. To implement the trusted dealer, we created a local folder where we have the key pair for each node, then we execute the algorithm and each node reads its key pair and the public key of each other node. For the keys to be created we have to execute HashGraphBFT with first argument "generate_keys" and second the number of nodes (HashGraphBFT generate_keys $N). The keys generated is RSA keys with 2048 bits length.

The hash function we use is SHA384 which is set in the app.Hashes.go "makeHash" function where we can change it if we want.

4.3 Structs and Variables

4.3.1 Gossip Message

Figure3.7 shows the EventMessage struct, this is the message that the events gossip.

- **Signature** is an array of bytes where the node who created the event signs it and insert the signature in this field.
- **Timestamp**: the time the owner claims it created this event
- **Previous Hash**: the hash value of the previous event = the self-parent event = the previous even in the same HashGraph line
- **Parent Hash**: The Hash value of the parent event (if exists, is the diagonal line in the graph)
- **Owner**: Is the ID of the creator of this event (= we also know in which Hashgraph line it will be placed)
- Each **transaction** is unique and represented in three fields in the Message
    - **Transaction:** the string message
    - **Number:** an integer number which is increased
    - **ClientID:** The ID of the client who sent this Transaction.
- First Owner: shows who created first this transaction

```
type EventMessage struct {
    Signature     []byte
    Timestamp     int64
    Transaction   string
    PreviousHash  string
    ParentHash    string
    Owner         int
    Number        int
    ClientID      int
    FirstOwner    int
}
```

Fig4.1: The Gossip Message

EventMessage is in the package type so when is used in other packages is declared as type.EventMessage.

4.3.2 HashGraph Node

Figure 4.2 shows the EventNode which is the node we insert in the HashGraph.

Each EventNode is consists of:

```
type EventNode struct {
    EventMessage   *types.EventMessage
    PreviousEvent  *EventNode
    ParentEvent    *EventNode
    OwnHash        string
    Know           []bool
    Round          int
    Witness        bool
    Famous         bool
    RoundReceived  int
    ConsensusTime  int64
    Orderedplace   int
}
```

Fig4.2: The struct of the nodes in the HashGraph.

- **EventMessage:** The gossiped message for which this node is created.
- **PreviousEvent:** is the event that eventMessage points to and we find it from the previousHash Field
- **Parent:** is the event that eventMessage points to and we find it from the parentHash Field
- **OwnHash:** The hash value(sha384) of the Event Message
- **Know:** an array of bool of length N where we set who knows this event
- **Round:** the round we give to this event with divide rounds algorithm
- **Witness:** Boolean shows if this event is witness or not
- **Famous:** if this event is witness it can be famous witness
- **Round Received:** The received round of this even it is given from the divide order algorithm
- **Orderedplace:** W have a listed where we have sorted the events and this is the place that the event are in that list

4.3.3 Helpful Structs and Variables

Figure 4.3 shows the structs that we use which help in the implementation of the algorithm.

- **Hashgraph:** We use history to create the HashGraph. Hashgraph is a list of History struct – an array of lists.
- **OrphanParent:** event nodes that we receive whose parent event we did not have yet.
- **OrphanPrevious:** event nodes whose previous(self-parent) we did not have yet

```
//History -
type History struct {
    events []*EventNode
}

var (
    //HashGraph -
    HashGraph []History

    // EventNodes without parent which are in the hashgaph
    orphanParent []*EventNode

    // EventNodes without parent which are in the hashgaph
    orphanPrevious []*EventNode

    //all the hashgraph events in chronological order
    sortedEvents []*EventNode

    // MU - mutex to access hashgraph
    MU_HashGraph sync.RWMutex

    // Witnesses -
    Witnesses map[int][]*EventNode
)
```

Fig4.3: Helpful structures and variables

- **SortedEvents:** a list in which we have all the eventNode which are in the Hashgraph sorted by their timestamp
- **Mu_HashGraph:** a mutex for the critical section – the Hashgraph
- **Witnesses:** a map mapping integer values to array of Event nodes. Each value is the round number and the list contains the witnesses of that round.

4.4 Channels and Messages

The Figure 4.4 shows the channels we use to send and receive messages to each other node and to the clients.

4.4.1 Nodes Channels & Messages

Sending Events to other Nodes

When we want to send a gossip message(EventMessage) to another node we create an object of type Message. Inside the payload we encode our eventMessage to a stream of bytes, inside type we insert the String "event" and our ID in the field from. Then we insert the Message object in the MessageChannel and a go routine which is continuously running checks if there is any events in the messageChannel and sends them to the recipient based on the from field.

```
type Message struct {
    Payload []byte
    Type    string
    From    int
}

var ClientChannel = make(chan struct {
    clientTransaction string
    transactionNumber int
    clientID          int
}, 10000)

EventChannel = make(chan struct {
    Ev   *types.EventMessage
    From int
}, 10000)

MessageChannel = make(chan struct {
    Message types.Message
    To      int
}, 10000)

type Reply struct {
    From  int
    Value int
}
```

Fig4.4: Channels and messages for each node

Receiving Events from Other Nodes

When we receive a message its type is the Message struct. For each connection to another node we have a go routine which continuously checks whether is an event in each socket. For each event in the socket with type="event" (this always happens, type field is redundant) it decoded to an EventMessage and inserted in the EventChannel.

Sending Events to Clients

When we decide on the order of a transaction we send a message to the client. We create a Reply message where we insert our ID in the field "from" and the integer number which corresponds the number of the transaction that is ordered.

Receiving Messages from Clients

Similar to Receiving Events from other nodes, we continuously check if we have any message in the sockets connected to each client and if we have we insert the message contents in a struct and insert it in the ClientChannel.

4.4.2 Client Channels & Messages

Client Sending Events

For each node we have a client request channel where we put the messages we want to send to that node. When the client wants to send a transaction it creates a ClientMsg and inserts it in the ClientRequestChannel of the node he wants to send the transaction.

We have a go routine for each clientRequestChannel which is continuously executed checks if there are messages in this channel and transmits them to the corresponding node.

```
ClientRequestChannel = make(
    map[int]chan types.ClientMsg,
    10000)

ResponseChannel = make(chan types.Reply)

type ClientMsg struct {
    ClientTransaction string
    TransactionNumber int
}

type Reply struct {
    From  int
    Value int
}
```

Fig4.5: Client Channels and messages structs.

Client Receiving Replies

Client has a TCP connection with each node and a go routine for each connection which checks if there is a message. When there is a message to be received it is inserted in the response channel. The messages that the client receives is the Reply struct as shown in Figure 4.5.

4.5 Sign and verification

Sign: When a node creates an EventMessage, it initializes the signature field as empty, signs the message with the private key and then inserts the signature in the corresponding field

Verify: When a node receives an EventMessage based on the Owner field it knows who created that event. It creates a copy of the message with the signature field as empty and verifies the modified message with the signature and the public Owners key.
The above are implemented in the package threshenc, file name: sign_and_verify.go.

4.6 Sorting Events

When we sort the event for the sortedEvents lists we sort them based on the timestamp. When we sort the events in the Decide Order Algorithm we sort them based on the round received and then based on the consensus time.

4.7 Algorithm Initialization

- The first thing the algorithm does is to initialize all variables with the default values or creates lists with length zero.
- It inserts N event nodes in the Hashgraph, one for each node. Those events have predefined field values and are always the same for all nodes
- Always for each event we insert in the graph we insert it in the sortedEvents list.
- Then we call the Divide Rounds event for each one of the N transactions which sets the transactions field Witness to true (because they do not have parent events) and round number to one.
- It executes three functions, sendGossip, manageIncomingGossip and manageClientRequests as go routines which are always executed.

4.8 Comparing Event Messages

We compare the event messages in 2 ways using two functions.
The first function is **checkSameTransaction** which compares two messages and returns true if the messages have the same transaction meaning same transaction string, Number and ClientID.
The second function is **checkTotallySameTransactions** which compares two messages and returns true if all the fields of the two EventMessages are the same. More specifically we make the hash of the two messages and if it is the same we return true.

4.9 Client

The client is responsible to send transactions to the nodes and receives ACK from them. The Client is executed with the command
“HashGraph_Client $ClientID $Nodes $Clients $Rem” where

ClientID is the id of the client, Nodes is the total number of nodes in the system and rem refers to local execution or remote where we set the IP of the remote machines.

When a client is executed it opens TCP connections to each node. The number of clients is used to set its port number and Rem is used to set the correct IP addresses. For a remote scenario the IP addresses should be modified dependent of the machines.
Inside the app.client.go each client starts sending transactions to the nodes. A client transaction contains a string which is the transaction and a number which is an increased integer number. Given the number of nodes the client calculates the total number of faulty nodes. When a client sends a transaction, he sends it to $f+1$ nodes to make sure that an honest node received its transaction. When a node has given an order on a transaction it sends an ACK to the client who created it, the client counts how many different nodes has accept its transaction and if it is ordered by more than f ($>=f+1$) nodes it consider that the transaction is ordered and writes in a log file the time since it sent the transaction. For every ordered transaction we also write in a log file the total number to order n transaction, we keep the time we sent our transaction and for every ordered transaction we subtract it from the current time and write the time elapsed and the number of ordered transactions in a log file.
The Hashgraph algorithm needs to build the Hashgraph and then if it can, orders some transaction, due to the above empty transactions may help the last transactions to be ordered. Therefore, we set the clients to send a number of empty transactions to help ordering their transactions and to not wait for other transactions to be send. The empty transactions contains the string ”empty” as transaction and the increased integer number.

When a client is executed it waits for ID/4 seconds to start sending transactions and then sends a transaction every one second. We set that so not concurrently all clients start sending transaction despite that there will be no problem. By setting the “NumOfTransaction” and “EmptyTransactions” variables inside client.go we set how many transactions for each client to send. We will see later that the frequency on sending transactions may have an impact on the performance of the algorithm.

Chapter 5

Go Routines

5.1 Send Gossip 20
5.2 Manage Client Requests 21
5.3 Manage Incoming Gossip 22
5.3.1 Check Gossip 23
5.3.2 Executing the Algorithms 24
5.4 Protocols 25
5.4.1 Divide Rounds 25
5.4.2 Decide Fame 26
5.4.3 Find Order 27

5.1 Send Gossip

Figure 5.1 shows the send Gossip routine pseudocode.

In this routine

- each node choose randomly the ID of one other node different from his.
- Locks the HashGraph mutex because it will possibly modify data.
- Finds all the events in the Hashgraph which we know that the node we chose to sync does not know them.(we use the Boolean array Knows)
- We insert the events in two lists
    - the events we created
    - the events the others created
- First we sent the events that we created and then the rest.
- We send the EventMessage which is

```
for{
    syncWith <- random(0-N) && random!= myID

    MU_HashGraph.Lock()

    var myEvents     //empty list of EventMessage
    var othersEvents //empty list of EventMessage

    for each EventNode v in sortedEvents {
        if !v.Know[syncWith]{

            if v.EventMessage.Owner == MyID {
                myEvents.append(v.EventMessage)
            } else {
                othersEvents.append(v.EventMessage)
            }

            v.Know[syncWith] = true
        }
    }

    for each EventMessage V in myEvents {
        send V to syncWith
    }
    for each EventMessage V in othersEvents {
        send V to syncWith
    }

    MU_HashGraph.Unlock()
}
```

Fig5.1: Manage Gossip routine pseudocode.

contained in a field in the EventNode object

- Unlocks the Hashgraph mutex

We use the sortedEvents list where we have all the Hashgraph events sorted based on their timestamp so the recipient will receive events in order from the older to the newer. But this may not happen because events may arrive out of order. We send first the events we created so if the other node learns from us a new transaction it will create a transaction in his Hashgraph line pointing to the transaction to our Hashgraph line.
The algorithm also works if we sent the transactions in ascending order based on the timestamp, the pseudocode of this alternative is shown in Appendix C.

5.2 Manage Client Requests

This routine manages the request we receive from a client and is executed continuously. Upon receive, the client messages are inserted in the clientChannel. An event in the client channel has a string which is the transaction, an integer number which is the number of this transaction and it is different for each transaction from the same client and the ID of the client who sent this request.

```
for each Message in ClientChannel {
    MU_HashGraph.Lock()

    if msg exists in HashGraph
        continue

    Create an EventNode with the contents of the Received Message
    Insert the EventNode in my hashGraph line

    MU_HashGraph.Unlock()
}
```

Fig5.2: Manage Client Requests routine pseudocode.

For each request in the client channel:

- we check if we have it in the Hashgraph (in any line) using the checkSameTransaction function.
    - If it exists we discard it and continue to the next message

  If it does not exist in the Hashgraph
- We create an EventMessage and insert the received message contents in the Event message, we set the timestamp to the currentTime (in UnixNano), the ParentHash is empty because we are the first who created this event, PreviousHash (hash of self-parent) is the hash of the last event in my Hashgraph line and in FirstOwner we insert our ID.
- Then we digitally sign this eventMessage and insert our signature in the corresponding field.

Then we create an EventNode and insert the above message in the corresponding field. Also, we fill the other fields with the correct data.

- PreviousEvent(self-parent) we have the pointer which points to our last event in the Hashgraph
- ParentEvent is nil because this event does not have parent.
- OwnHash is the hash value of the eventMessage we created above
- Know is a Boolean array of size N where every field is false except the know[ID] position
- Round is set to default(=0)
- Witness is set to false
- Famous is set to false
- RoundReceived is set to default(=0)
- ConsensusTime is set to default(=0)
- OrderedPlace is set to default(=0)

Then we append the above event node in the Hashgraph, more specific we append it in the list which corresponds to the events we created (HashGraph[ID].events) and insert it in the sortedEvents list in the correct index.

5.3 Manage Incoming Gossip

This routine is responsible to receive the nodes gossip, decide whether to insert it in the graph or not, create another node based on the received event and when to execute the algorithms to determine the order of the transactions.

It is continuously executed and iterates over the EventChannel which contains the eventMessages we receive from other nodes.

For each message, it executes the checkGossip function and then calls the execute algorithms.

Both functions are shown in the Figures below and are explained.

```
for each Message in EventChannel {

    MU_HashGraph.Lock()

    checkGossip(message)
    executeAlgorithms()

    MU_HashGraph.Unlock()

}
```

Fig5.3.: Manage Incoming Gossip Routine

5.3.1 Check Gossip

The checkGossip function, Figure 5.4 , takes as input the received EventMessage and determines if its valid, if we should insert it in the graph and also checks whether we should create another node and insert it in the graph in our Hashgraph line.

```
func checkGossip(msg EventMessage){
    if msg.Timestamp > Now
        discard
    if msg.signature Not Verified
        discard
    if hashGraph has Totally same transaction
        discard

    parent,previous = Find parent & self-parent
    if has timestamp after parent or self-parent timestamp
        discard

    EventNode = new EventNode(message)
    Insert EventNode in HashGraph

    if isOrphan
        insert it in the corresponding list

    if is new Transaction
        Insert EventNode in my hashgraph line

}
```

Fig5.4: Check Gossip Function pseudocode

An event is valid when

- the timestamp it contains is not after the currentTime
- The events signature is verified, we use the event.Signature field and the event.Owner which shows who created this event
- If we do not have a totally same transaction in the Hashgraph (an eventMessage with the same hash value as the received)

- Then we find the events parent and self-parent if exists.
- Create an event node with the given field(eventMessage, parent and self-parent) and initialize its fields
    - In the Knows Boolean array we set as true our ID index and the index of the sender
- If the event has a parent or a self-parent which we do not have yet we insert it in the corresponding list. We have two lists “orphanParent” where events whose parent is not known to us yet “orphanPrevious” whose gossip-parent is not known to us yet.
- We also check if the received event is parent or self-parent of any event in the two above lists and delete the event from the corresponding list.
- Last, we have to determine if we have to create a node in our Hashgraph line pointing to this event.

  We check our Hashgraph line if we have the received transaction.

  If we have that transaction we don’t do anything, otherwise similar when receiving a transaction from a client we create an eventMessage and an

eventNode which has self-parent our last event in our Hashgraph line. The only difference from receiving a transaction from a client is that this eventMessage has parentHash the hash of the previous received event and the eventNode has pointer to the previously created eventNode.

5.3.2 Executing the Algorithms

The function in Figure 5.5. shows when the algorithm decides to execute its routines to decide the order of the received transactions.

After encountering many difficulties, we decide to set a bound on when to start executing the algorithm and for how many transactions. We set it to execute the algorithms for each 100 transactions. For example when we receive 200 transactions we set the algorithm to iterate till the 100th transaction and then we increase the bound by 100 so there is a gap of 100 transactions. (example: with bound/next = 300 it will iterate till the 200th transaction).

The algorithms iterate in the sortedEvents array which contains all the transactions sorted based on their timestamp and variable "end" shows in which index the algorithms stops.

```
var next int = 200
var gap int = 100
var end int

func executeAlgorithms(){
    if HashGraphSize <= next {
        return
    }

    end = HashGraphSize - gap

    DivideRounds()
    DecideFame()
    FindOrder()
    next+=100
}
```

Fig5.5 : Execute algorithms function.

In the paper states that the algorithm is executed for each transaction it receives but we decided to set that bound because the algorithm takes time to be executed and also to avoid executing the algorithm repeatedly on the same transactions.

This is an implementation decision which we made, however we believe that it is not implemented in the real Hashgraph algorithm and may not be a good solution. We insert it, so we are surer that the Hashgraph waits to receive a number of transactions and also give some time for transactions to be gossiped to other nodes. The fixed values is our choice and we believe it can be change or removed but with those the implemented algorithm is harder to fail.

5.4 Protocols

5.4.1 Divide Rounds

We know that in the SortedEvents list we have all the events (eventNodes) which exists in the graph in an ascending order based on their timestamp. Also, each event has a field orderedPlace which is the place of the event in the sortedEvents list but this field when we create an event and insert it in the Hashgraph is set to the default value which is zero and the divide rounds algorithm is responsible to set this value. The first N events in the list are the initial events in each Hashgraph line for each node and those events has round = 1 and their orderedPlace is already set. All the other events have orderedPlace zero, which mean this value is not the correct based on their index place in the list.

Appendix B includes the routines used to implement the DivideRounds algorithm. In the first Figure we iterate in the sorted Events list till the index is equal to end. “end” variable is initialized from the execute algorithms function above. For each eventNode which has ordered place field value not equal to its ordered position in the list we execute the divideRounds algorithm for this event. When the orderedPlace field is different than the index of the eventNode it means that either we have not already executed the algorithm for this event or that we received an event with smaller timestamp than this event so we will execute the algorithms again for this event. From the first event that we execute the algorithm, it will be executed for all the events which come after that, since if the ordered place changed for one event all the events that comes after it have to execute the algorithm again.

The divide Rounds event function is the second Figure in Appendix B. It takes as parameter the eventNode and the list sortedEvents. Then

- if the event has not previousEvent and timestamp equal to zero
    - has round number 1
    - it is round 1 witness
    - This only occurs for the first N events
- If the event is orphan (we do not have its parent or gossip-parent) we set its round number and order place to -1 so we just wait to get its prior events. If the above does not apply we find the max round number of the event’s parent and

gossip-parent round and store it in variable R. Gossip-parent may not exist if it's the first transaction created.

- Then we count how many round R witnesses can strongly see the given eventNode.
- If it's strongly seen by more than 2N/3 events we increase the R variable by one.
- Now we can set the famous variable. An event its consider famous if it has bigger round number that its parent's events.
- Then we init the round and witness field of the given event node
- If the event its famous we insert it in the map in the corresponding witness round events list
    - Also, we check if it is round r-1 witnesses and remove it from that list

Also, we have two variables, a Boolean variable "witnessChange" and an integer "witnessChangeRound". When we change a witness, meaning that we declare an event witness and insert it in the corresponding witnesses list, we set the witnessChange variable to true and we keep in the witnessChangeRound the minimum round we insert or change a witness.

5.4.2 Decide Fame

After the execution of the divide rounds protocol, we execute the decide fame protocol which is shown in Appendix B.

First, we find the minimum round we modify a witness using the minWitness change variable from the divide order algorithm and store it in a variable R. If there is not a witness change means that we should not execute the algorithm and we return.

Then starting from the round R

- we take the events of 2 consecutive rounds
- For each event in the prior round, we count how many witnesses can see it in the next round.
- If it's seen by *2N/3* round *R+1* witnesses it is considered famous.

  When we execute the algorithm for every round R witness, we increase the round *R* value by one and repeat.

In our implementation we decided to set the iteration to stop 2 rounds prior the maxRound number witness and set the minWitnesschange round to that value. This helps us wait till more events are inserted and the witnesses won't change.

5.4.3 Find Order

After we decide which witnesses are famous, we can now check if we can decide the order of some transactions. In Appendix B is the pseudocode for the find Order protocol. We have a list which contains the events we ordered and for which we have sent an ACK to the client, we also have a list of events for which we have decided their receive round and consensus time but not send an ack to the client and a variable where we have the max received round that we ordered an event. Similar to decide fame we execute the algorithm till the maxRound-3, because there might be witness changes, so we wait to receive more events and the max round witness to be increased.

There are transactions that may be ordered, but it's possible that we receive event messages containing the same transaction after we decide its order. Therefore, for each ordered event (in OrderedEvents list and tempOrder) we iterate in the sortedEvents which contains all the events and if we have the same transaction we set its roundReceive to the roundReceive that has the ordered event. It is possible that we rewrite the same number in the some events but also we may set the roundReceive to Transaction that arrived after we ordered an event containing the same transaction

Then we iterate in the sortedEvents lists, we use the end variable which is initialized in the executeAlgorithms function and iterate till that index.

For each EventNode in sortedEvents

We do not order the eventNode if

- it has timestamp == 0
    - only the first eventNode has timestamp = 0
- its round number is not set
- it has round number greater than the desired round
- we already set a round received

We order only the events which does not have parent event (and are not orphan) and we have a node pointing to them from 2n/3 different nodes.

We take the eventNode.round which is the round number determined in the divide rounds algorithm.

Then for each round witnesses starting from the *round*+1:

1. if the current round does not have famous witnesses or its greater than the desired round we break and continue with the next eventNode.
2. we check if all the famous witnesses can see this event and if that is true, we call the setRoundReceived function which is shown in Appendix B.
3. If the above does not satisfied, we increase the round number and repeat till we break the for loop or we set roundReceived to the eventMessage.

Before ending the function, we call the decideOrder function, the pseudocode is shown in Appendix B.

Above we order events that has round number. Events that have a round number are events for which we have executed the algorithm divide rounds, therefore those are events that are not orphan because divide rounds is executed for only not orphan events.

SetRoundReceived

We call this function when we decide on the round received of an event.

- Creates a list of timestamps (int64,UnixNano)
- For each event V in the Hashgraph (including the given eventNode)
    - If it's the same transaction as the given eventNode
        - we set the roundReceived as the given round
        - If the v.round is after the round of the event we decide its order and prior to the roundReceived we check whether this event can see the given eventNode
            - If it's true we insert the events timestamp in the list
- We find the median of the list
    - set that as the consensus time
    - insert the event in the tempOrder list

We can insert in the events any of the events with the same transaction because in the tempOrder we see the round received and the consensus Time. So, we set the consensus time in the event we have and insert that in the list. We do not have to insert the

consensus time in all other same transaction because we only need one with that and because the roundReceived is set in all of them we won't try to order them again.

DecideOrder

At the end of the find order protocol we call the decide order function, which is shown in Appendix B to finalize the order of the temporary ordered transactions which are in the tempOrder list. We did that that we send ACKs for a round and then proceed to the next round.

- We sort the event in tempOrder based on the receive round and then the consensus time.
- We find the max round receive
- We sent Ack to the clients for all events with received round < that the max
- We remove the events from the tempOrder list and insert them in the OrderedEvents list.

Chapter 6

Implementation Details and Decisions

6.1 Message Size 30
6.2 Timestamps 31
6.3 Orphan Events 31
6.4 Comparing Transactions 31
6.5 Sorted Events list 32
6.6 Inserted Bounds 32
6.7 HashGraph Size 33
6.8 Messages Exchanged 33
6.9 Empty Transactions 33

6.1 Message Size

Client Messages to nodes

The messages received from the client contains a string which is the transaction and for simplicity we consider it as a character which is 1 byte, and an integer value which is the transaction number and its 4 bytes.Therefore the client messages is 5 bytes.

Reply messages to Clients

The messages sent to the clients contains 2 integer values, the sender node ID and the number of the message that its Ordered. Therefore, the Reply message has size of 8 bytes.

Gossip messages between nodes

The gossip messages is type message and has fields payload, type and from. The type contains the string “event” (5Bytes), it is not used but it will be used if we send different type of messages. From is an integer value (4Bytes). Payload is a stream of bytes and contains an encoded EventMessages. The event message has fields

- Signature: 256 bytes
- Timestamp: 8 bytes
- Transaction: 1 byte
- PreviousHash: 96 bytes
- ParentHash: 96 bytes
- Owner: 4 bytes
- Number: 4 bytes
- ClientID: 4 bytes

Total: 469 bytes

So total of a gossiped message is (469+5+4) 478 bytes.

6.2 Timestamps

As **timestamp** we use the function Now().UnixNano which is the number of nanoseconds elapsed since January 1, 1970 UTC and it has type int64.

6.3 Orphan Events

We decided to implement the orphan lists and insert in the graph events that we have not yet receive the prior events they point to. However, for those events when we determine that they are orphan we do not execute the algorithms to decide round, fame and therefore ordering. We conclude to this because while implementing the algorithm to discard events that it has not receive their previous, each node created different graph despite that it showed that each node had sent the events it inserted in the graph to everyone else. Then we observe that many times the events arrive at the destination out of order so the event were discarded, since we can't wait till we receive events and decide to insert it in the Hashgraph we implement the orphan lists which helps us to solve that problem and we consider those events to not exists when we execute the algorithms till we get their previous events . However, implementing this allows us to let the node gossip to other nodes all the events in its Hashgraph including orphan events and this we believe improves the time for each other node to learn the existing events.

6.4 Comparing Transactions

Check Same Transaction and Check Totally Same Transaction

The client sends a transaction to *F+1* nodes, therefore it is possible that we learn a transaction and then receive that same transaction from a client. Since we have a transaction in my Hashgraph line we should not make another transaction with the same

fields but different timestamp. We can gossip the transaction we already have without the need to create the same. That's why in this occasion we only need the simpler check function which does not checks the timestamp or hashes.

Because the client sends its transaction to *f+1* nodes there might be multiple events with the same transaction and different owner. Similar many events with the same transaction exist when a faulty node creates multiple events with different timestamps. Therefore when receiving events we check if we have the exact same event and if we don't we insert it in the graph and gossip it so that everyone knows its existence.

6.5 Sorted Events list

SortedEvents is a list which contains all the transaction that are in the Hashgraph sorted by their received time. We use this list to calculate the roundNumber and fame so we start from the oldest event going to the newest. Similar we use this list in the sendGossip protocol so we send the oldest events to each other node and then the newest, so we uses the orphan lists as less as possible. If the events arrive out of order the orphan lists help us solve the problem.

6.6 Inserted Bounds

While executing the algorithm we observed that the graphs and the order may be different. Hedera says that the graphs in the nodes may be different at the last events but after some time when all the transactions are submitted the graphs are identical. This continuous for as long as there are transactions transmitted.

Therefore, we conclude to set some bounds on the algorithms. We chose to execute the algorithms to the max round witness minus 2 so that we wait some time for the max round witness to be increased and then execute the algorithm on the transactions. The value we placed is something we chose and it may not needed or there might be a different solution. However, Hedera does not go into details on how they solve this problem so we implement it in this way.

6.7 HashGraph Size

For each transaction with unique timestamp, we each node to create an event message with this transaction in its graph pointing to the first event or to any of the event which points to the first or any other which has the same attribute(point to the first or to an event who points to the first). So, for each client transaction that is received from a node, an eventNode is created and inserted in the graph, we expect *N* eventNodes to be in the graph.

Example: *K* Clients, each sends *m* Transaction to any node in a system of *N* nodes total =>HashGraph Size = $k*m*N+N$.

The *"+N"* is the first transactions created to initialize the Hashgraph.

6.8 Messages Exchanged

For each event node in the Hashgraph we have a Boolean list where we set to which event we sent this transaction so we know to who knows it. Also, when we receive a transaction if we have the exact same in our Hashgraph we set the know[ID] to true so we know that this event knows this transaction and fewer messages has to be sent. There are many times that two nodes know a transaction but they do not know that they both know it, in this case its possible that one or two redundant events are send among those nodes which are discarded.

6.9 Empty Transactions

If there is a steady stream of transactions entering the system the Hashgraph size is increased and eventually each transaction is ordered. There must be found a witness and a famous witness in a round so we can order all the events before that round. If there are not enough events to find famous witnesses the last transactions may not be ordered. Therefore we can set in the client to send a number of empty transaction so there are more events in the Hashgraph and it is easier to find famous witnesses and order its transactions

Chapter 7

Evaluation

7.1 Scenarios 34

7.1 Evaluation Summary 35

7.2 Comparison with other Works 41

7.3 Conclusions 42

7.1 Scenarios

When executing the algorithm the fourth parameter is the scenario. We created four (4) scenarios which are explained below

0 : Normal

Normal scenario where no malicious nodes exist**.**

1 : IDLE

The malicious nodes do not send messages to anyone. We have to execute all the nodes and set this parameter. The malicious nodes receive the gossip messages but they do not execute the “SendGossip” protocol so they do not communicate with anyone.

2 : Sleep

The malicious nodes stop sending event messages to the other nodes for some time and then continue sending from where they stop. Similar to the above scenario we have a function inside SendGossip where we set in how may seconds the malicious events to stop executing and after how many seconds to start again. The events receive transactions from other nodes but they don’t send anything therefore when they start executing they will send messages which much smaller timestamps than the current time.

3 : Fork

When a malicious node

- will send an event which it created (parentHash is nill), meaning the client told me this transaction and we insert event in the graph.
- it chooses to sync with another node with $ID < N/2$

It creates a different event message, with a much smaller timestamp than the currentTime (<1000), it also set that its previous event in the Hashgraph line is the very first event of the malicious node.

Then the malicious node signs this event and send it to the other node.

When the malicious node syncs with another node with *ID < N/2* it will make the same actions and possibly it will give a different timestamp to that transaction.

When it will sync with another node with *ID >= N/2* it will send the correct message.

In the above scenarios we execute the algorithm multiple times(and while executing for evaluation), compare the order of the transactions of the non-faulty nodes and we observed that the order is correct.

7.2 Evaluation Summary

In our experimental evaluation we executed the algorithms by increasing the clients and/or the transactions the clients send, the frequency they send transactions and by increasing the nodes. Most of our experiments where with 4 nodes and different number of clients. At the end of this sub chapter we explain the difficulties we observe when we increased the nodes.

The results shown here are executions of a system with 4 nodes. Latency and frequency are measured in seconds.

We executed the algorithm in a cluster of 9 machines.

Table 7.1 shows the number of CPU cores of each machine in the cluster.

In a system of 4 nodes we the machines with number 0,1,6,7 to simulate the nodes and machine 3 to simulate all the clients.

Machine	CPU cores
0	8
1	8
2	1
3	2
4	1
5	1
6	2
7	2
8	1

Table7.1: The CPU cores of each machine in the cluster

Operation Latency

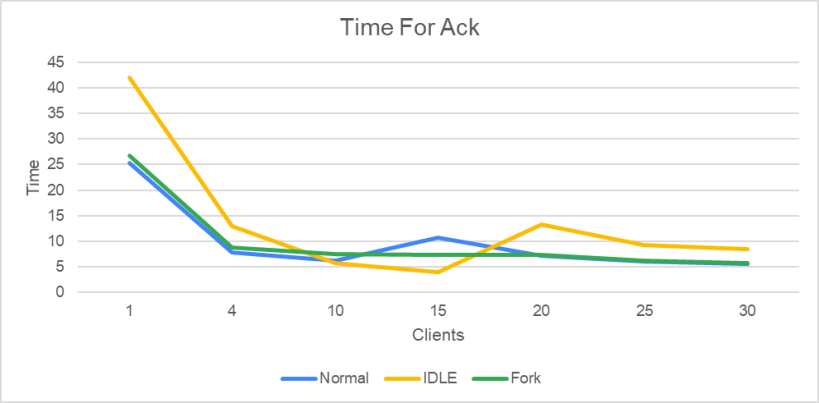


Fig7.1: Time for client to receive Ack.

Figure 7.1, shows the average time a transaction needs to be claim as ordered from a client. The clients wait to receive *f+1* acks to assume that a transaction is ordered. Here we observe that the average time vary depending on the scenario. We see that the normal and fork scenario are faster than the idle. This is because more transactions are send over each node and the graph is build faster therefore its easier and faster to find famous witnesses. The latency seems about 10 seconds on average. In an experiment the Hashgraph paper shows that the latency in 4 nodes it is about 7 seconds where in this scenario, our implementation is close in some cases but its higher that the results shown. Also, the results show 22000 transactions per second, but this is much more than the experiments we executed.

Message Complexity

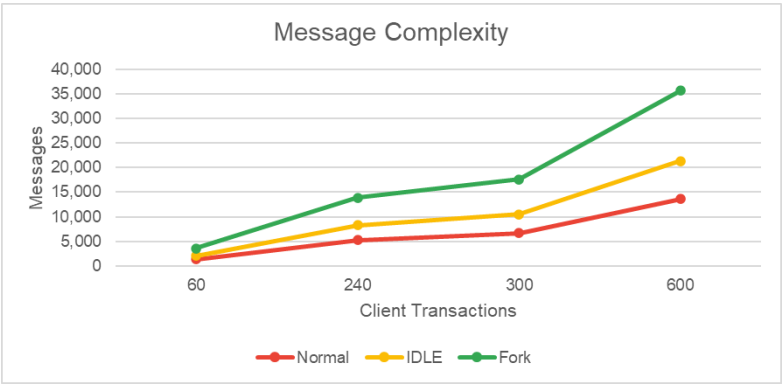


Fig7.2 : Total Messages over all transactions

	Total Messages		
Total Transactions	**Normal**	**IDLE**	**Fork**
60	1292.25	753.00	1481.33
240	5265.25	3026.67	5577.67
300	6714.25	3802.08	7068.08
600	13589.50	7805.67	14318.67

Table7.2: The values used for Figure 7.2

	Average Messages / Transactions		
	Normal	**IDLE**	**Fork**
	21.54	12.55	24.69
	21.94	12.61	23.24
	22.38	12.67	23.56
	22.65	13.01	23.86
Average	22.13	12.71	23.84

Table7.3: The average messages sent for each transaction

The above figures and tables show the message complexity in some scenarios. We made experiments by sending 60,240,300 and 600 transactions from multiple clients. When a client sends a transaction, this transaction will be sent to all nodes, so messages are affected by the *N*.

In Table 7.3 we took the values of Table 7.2 and we divided the total number of messages by the transactions so we found the average number of messages sent for each transaction. Then we calculated the average of tshose times. We concluded that the average number of messages send for each transaction is about 22 and 24 in the scenarios normal and fork. In the Idle scenario there are much less messages created because its one less node in the system and that's why the average messages send for each transaction are about 13.

As described in the Chapter 6.1 the message size is 478 bytes. So in average for each scenario there are

- Normal: 22*478B = 10.25 MB transmitted
- Fork : 24*478B = 11.2 MB transmitted
- IDLE : 12*478B = 5.6 MB transmitted

for each transaction

Frequency

We observe that the frequency the clients send transaction may affect the latency and also it may affect the number of acks send to clients. In each execution the number of acks send to clients may vary, this probably depends on the randomization and how the graph is build but how the graph is build might also affected by the frequency of transactions entering in the system.

So, we decided to execute the algorithm with different frequency of clients sending transactions. We set the system to 20 clients, 4 nodes and 50 transactions per client and we increase the frequency from 3 to 6

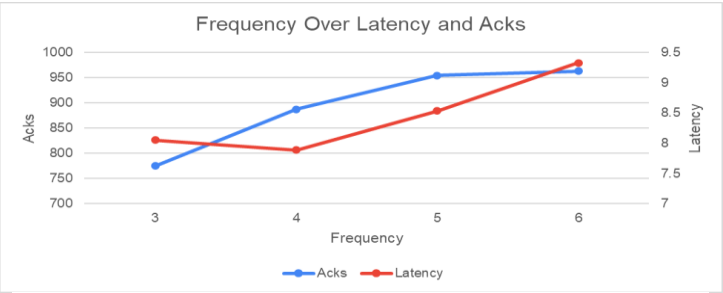


Fig7.3: How latency and Acks are affected when frequency changes

Frequency	Acks	Latency
3	775	8.05
4	887	7.88
5	954	8.53
6	963	9.33

Table7.4: The Values used for figure 7.3

In total 1000 transactions are sent, 50 clients of 20 transactions each ,and we count the average latency for each transaction and the total number of acks send to clients. We notice that there is a big difference in acks send to clients while the frequency is increased, but when is 5 and 6 seconds the number of acks is close. Also, the latency changes but we have to consider that it depends also on the Acks. We see that with frequency 4 the latency is about 7.9 and with frequency 5 is 8.5 but we must also see that there are about 60 more acks send to clients with 5 as frequency.

Throughput

We observe the Hashgraph paper [5] which shows an evaluation where it is shown that it can order 22,000 transactions per second with latency about 7s on a system of 4 nodes and we decide to check how close to this we can get. We execute the algorithms normal scenario with 4 nodes, 20 clients with frequency 5 (based on the above evaluation) and started from 50 transactions per client and increasing it by 10.

N	Clients	Frequency(s)	Transactions	Total Transactions	Acks	Latency
4	20	5	50	1000	954	8.53
			60	1200	1035	11.05
			70	1400	1112	22.47
			80	1600	1066	14.92

Table7.5: Increasing the number of transactions

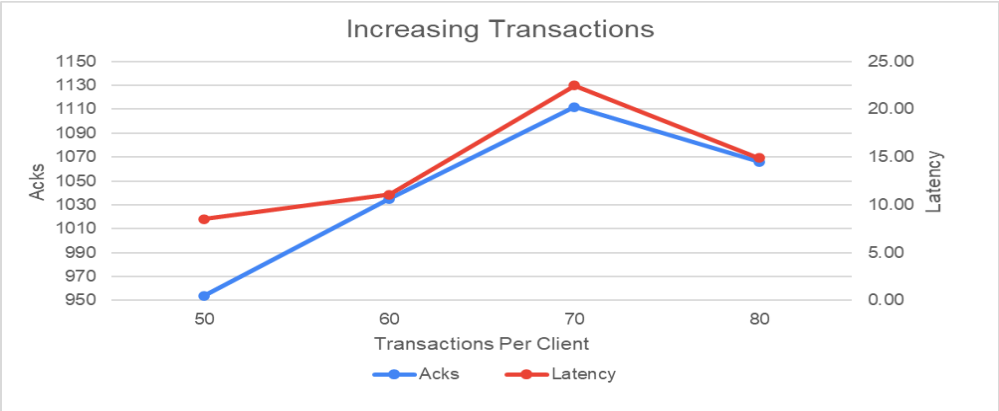


Fig7.4: How acks and latency are affected when transactions are increased

The above table shows the results of the executions. We started from 50 transactions per clients (total 50*20) till 80 transactions per client. Acks is the number of acks send to clients. It is a small scale experiment but we observe that there is a linear increase in acks but the same happens to the latency. We can see that with 70 transactions per client we have 22 seconds latency which is 3 times more than the latency in the paper.

We iterate till 80 transactions per client because the latency was increasing a lot when we try to execute the algorithm with 80 transactions. The numbers we see above is the moment we stopped the execution because it was executed for some minutes and it didn't finish or send any more acks to clients.

Performance over all transactions

Because the Hashgraph needs time to be gossiped, build we thought that it might be a good idea to observe its behavior when it send acks for multiple transactions. For example, we calculate the time needed to send Ack for *n* client transactions because many transactions are ordered in one round and each transaction has different latency. Also, we calculate the time that a number of transactions is ordered over all the clients. This metric might be similar to latency but because we calculate it for multiple transactions and not for each is less than latency.

		Per Client			System		
Clients	**Transactions**	**Client Trans**	**Total Time**	**Overall**	**Total Trans**	**Total Time**	**Overall**
20	10	8.73	44.54	**5.10**	262	50.54	**0.19**
20	12	10.56	53.96	**5.11**	264	57.16	**0.22**
20	20	18.35	93.85	**5.11**	367	97.68	**0.27**
20	50	46.20	254.57	**5.51**	924	257.91	**0.28**
20	60	51.75	299.67	**5.79**	1035	302.26	**0.29**
20	70	55.60	661.61	**11.90**	1112	680.98	**0.61**

Table7.6: Behavior over a number of transactions

The above table show the average number of transaction acks send to clients and the total number needed for those transactions. Similar shows the number of al transactions ordered and the time needed. We divide the total time with the number of transactions and we calculated the overall.

The First overall value is similar to the latency which in Table 7.1 its average value is 9 but here if we don't consider the last case it is less than 6.

The second overall value is the ratio showing how often a transaction is being ordered among all clients.

Increasing the Nodes

After the execution on 4 nodes we decided to increase the number of nodes so we observe how the algorithm scales.

*As a reminder we know that we set some bounds in the algorithms on till which event to execute the algorithms and to let some rounds as gap till it find more rounds with witnesses. Therefore, the Hashgraph has to be built and then if there are rounds with famous witnesses it will order some transactions.

We increased the number of nodes to 7 and set the number of clients to 10 where each sends 10 transactions, so total 100 transactions per client. The nodes are 7 so if each client sends 1 event there will be 700 events in the graph in the normal scenario. Because the nodes are 7 there are 2 faulty nodes, the clients sends their transactions to (f+1) 3 nodes. If the nodes insert the transaction in their graph without learning for it from other node who receive the same transaction there will be 3 same transactions in the graph with different first owner. If this happens to all transaction the maximum transasctions in the graph will be 2100 (700*3).

We observe that this behavior happens and there are about 2000 transactions in the graph and each node sends about 12600 messages total. But we observed that there are only 4 rounds created and because we set the decide fame and find order to wait for some rounds it does not order any events.

Then we decide to execute the algorithm in the Idle scenario where the faulty nodes does not send messages. In this scenario there will be less events in the graph and we wanted to see if it would order any transactions and how many. We observe that it reached only round 3 and did not order any event.

Last we decide to increase the number of transaction, so we set it to 20, with 7 nodes and 10 clients. We observed that the algorithm delays couple minutes and by calculating the time needed for the algorithm we notice that the algorithm divide rounds and then find Order takes the most time. Then we set it to execute the algorithms after having 2000 event in the Hashgraph in an attempt to reduce redundant operations, It kept taking too much time and at 3000 events we observe that it has events with round number 6, it send some Acks (14 out of 200) and it keep taking too much time so we stop the execution.

Unfortunately, we did not make it to execute the algorithm in a bigger scale and observe how it behaves because in real life scenarios it matters how easily the system is scalable. However, we notice that there is a problem in the implementation and a possible bottleneck in the real implementation of Hashgraph. We observe which protocol needs the most time and that there should be some improvements to increase the performance.

7.3 Comparison with other Works

We compare the results of our HashGraph implementation with the experimental evaluation of "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures" [4] – below is referred as "Algorithm2". Unfortunately, we were only able to compare it with the small scale experiment on a system with 4 nodes.

Similar to Hashgraph, Algorithm 2 uses cryptography only to sign and verify messages, however, a Hashgraph node has to sign all its messages and verify all the incoming gossip message where possibly takes more time.

We observe that the operation latency of Algorithm 2 with up to 75 clients sending transactions, on the normal and Idle scenario is maximum 2 seconds which is much lower than the HashGraph results, which is about 10 seconds.

Hashgraph is not affected so much with the increase of clients sending transactions where in the Algorithm 2 the operation latency is increased. However, we observe some serious delays on Hashgraph when a big amount of transactions are send in the system. Also, the forking scenario in HashGraph may have a negative and positive result. Negative because it creates even more transactions to be transmitted and the Hashgraph size is increased so the algorithms take more time. On the other side, more events mean that it is easier for the algorithm to find witnesses and famous witnesses therefore the operation latency may decrease.

Comparing to Algorithm 2 we observe that there is a significant delay in a scenario where the faulty nodes send different messages to the non-faulty nodes. It shown that with 75 clients the operation latency is increased up to 5 seconds but it is still less than Hashgraph.

Comparing the message complexity, we see that Hashgraph sends much less messages for each transaction and it is not affected by the clients but by the nodes in the graph. For example, we saw that Hashgraph sends about 20 messages for each transaction compared to Algorithm 2 that sends more than 100 messages.

7.4 Conclusions

Hashgraph aims to be efficient with improvements on the message complexity compared to other implementations. It is based on a gossip protocol and on Virtual voting which we observe that it really decreases the message complexity which is affected by the number of nodes. We concluded that the message complexity is *O(N^2)* for each client transaction, where *N* is the number of nodes.

It needs low CPU usage in matter of cryptography where it uses it only to sign and verify messages, but it's core protocols may need much resource power. We observe that the divide rounds algorithm which is based on recursive methods takes the most time and needs the most CPU usage. Improvements in this protocol may increase significantly the HashGraph performance.

The operation latency is not as low as expected, which is about 10 seconds. Scaling the system was not possible due to the big delays of the protocols divide rounds and findOrder and some serious improvements should be made.

In conclusion, Hashgraph is an efficient protocol with significantly small message complexity but high demand on CPU usage due to the algorithms executed in its core. Therefore, more research and improvements on the core algorithms may make it one of the most efficient BFT algorithm in all aspects.

Chapter 8

Conclusions

8.1 Summary 43
8.2 Notable Difficulties 43
8.3 Suggested Improvements 44

8.1 Summary

In paper [5] it was suggested that Hashgraph is an efficient algorithm with low message complexity. Despite the difficulties shown and the implementation results we still believe that with some improvements, the HashGraph protocol may have very good performance on a bigger scale. It is shown that it has small message complexity, suppresses all the previous work and that is a serious benefit. However, the CPU usage needed for its core principles is the bottleneck of the algorithm which needs more dive into details to understand and improve it. The performance may be affected from many parameters such as the frequency and number of transactions entering the system as well as the number of nodes in it.

Nowadays, especially cryptocurrencies revealed and stimulated a new demand for consensus protocols over a wide area network among large amount of node [8]. Many applications especially financial are based on BFT algorithms therefore there is a great need for robust and efficient protocols. Researchers are working to implement efficient BFT algorithms to improve some aspects, but most of the time delays in other aspects occur. Therefore, it is very challenging to implement a BFT algorithm that is efficient in all aspects compared to the existing ones.

8.2 Notable Difficulties

For implementing Hashgraph, we had to search research online, view YouTube tutorials and read the HashGraph paper [5] multiple times to understand how it works. Therefore, we implemented the algorithm based on [5]. It is a new algorithm and there are only a few examples showing how it works, especially when there is a malicious node. Hedera

has some examples on their website and on YouTube but are only few and simple and show how it works without faulty behavior of nodes. The protocol “Gossip about Gossip” was the most difficult to understand along on how the Hashgraph is built among all nodes. Also, we made some decisions on whether to run the algorithms and decide the order, and we inserted the orphan event idea which occurred when we saw that many events were arriving out of order. Those are not shown anywhere but with those, our algorithm works and it is less likely to fail. The above difficulties took us much time to understand and develop the algorithm.

In evaluating the system, we observed big delays when we put clients to send many transactions or when we tried to execute the algorithm on a larger scale. We tried to make some improvements, but we were not able to execute the algorithm on a system of 7 nodes or on a system of 4 nodes where clients send more than 2000 transactions in total due to the algorithm’s delays.

8.3 Suggested improvements

The algorithm may not be the ideal due to implementation decisions we made in order to work, such as the bounds and the orphan events. Also, we decided to check if we can realize the order of events after we receive all the incoming messages, but still there were big delays. A change in the above settings might increase the performance of the algorithm, for example, the bounds might be changed or removed, or the algorithm can be executed while receiving more events. Due to the fact that the orphan list we implemented it does not shown anywhere else makes us believe there might be a more correct way to create the Hashgraph data structure.

When we sync, we choose another event and send it all our events history; here multiple parallel syncs can be implemented so the events are gossiped as fast as possible.

Also, in our implementation, we execute three go routines, SendGossip, ManageClientRequests and Manage incoming Gossip. Each time a routine is executed:

- it locks the Hashgraph lock,
- finishes sending events(sendGossip) or checking the incoming message
- modifies data if needed, and
- unlocks the lock so another routine can be executed

The above implementation might not be ideal and if a more complex one is implemented, it is possible that it will increase the performance.

Also the results might be affected by the programming language and the communication framework. The algorithm is based on executing recursive method multiple times. We know that each event can see all the events that it's parent and self-parent sees. Therefore, if we store more data when we execute a recursive method on a parent it will reduce the operations needed for the child and the time complexity.

We did not implement all the above due to the time needed to implement it and the complexity to develop them.

Based on the above we believe that there are many improvements which can be done and combined with a deeper understanding of the algorithm, it may have a strong result on the algorithm's performance.

Bibliography

[1] Nikolas Pafitis. Implementation of Self-Stabilizing BFT Using Go And ZeroMQ, Diploma Project, Dept. of Computer Science, University of Cyprus (2017) (Accessed 02.02.2021)

[2] Castro C, Liskov B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 20(4), pp. 398-461, 2002.

[3] V.Petrou, BFTWithoutSignatures, 2021 (Accessed 25.05.2021)
V.Petrou, BFTWithoutSignatures_Client,2021(Accessed 25.05.2021)
Go Implementation of "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures"

[4] Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. Comput. J. 49(1), 82-96,2006

[5] Baird L, Luykx A.: The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers. COINS 2020, pp. 1-7

[6] Schneider F.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Comput. Surv. 22(4), pp. 299-319, 1990.

[7] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3):382-401, 1982.

[8] Zheng Z., Xie S., Dai H., Chen X., Wang H.: An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. BigData Congress 2017, pp. 557-564

[9] Gmytrasiewicz, P., Edmund D.: Decision-theoretic recursive modeling and the coordinated attack problem. Proceedings of the First International Conference on Artificial Intelligence Planning Systems, 1992, pp. 88–95.

[10] Miller A., Xia Y., Croman K., Shi E., Song D: The Honey Badger of BFT Protocols. CCS 2016: 31-42

[11] Duan S., Reiter M., Zhang H.: BEAT: Asynchronous BFT Made Practical. CCS, 2018, pp. 2028-2041

[12] Cachin C., Poritz J.: Secure Intrusion-tolerant Replication on the Internet. DSN 2002, pp. 167-176

[13] Go official website, https://golang.org/ (Accessed 25.06.2021)

[14] ZeroMQ official website, https://zeromq.org/ (Accessed 20.06.2021)

[15] ZeroMQ Guide, https://zguide.zeromq.org/ (Accessed 20.02.2021)

[16] G. Papaioannou, HashgraphBFT – Server Side (Accessed 02.06.2021)
G.Papaioannou, Hashgraph_Client – Client Side (Accessed 02.06.2021)
Go Implementation of Hashgraph BFT algorithm

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009

[18] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In International Workshop on Open Problems in Network Security (iNetSec), pages 112–125, 2015

[19] Hedera official website, Hedera (Accessed 20.05.2021)

[20] L. Baird, The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers, YouTube video, (Accessed 20.05.2021)

Appendix A

Here we show the pseudocode used to implement the see and strongly see relation.

```
ownerSee = make([]bool, variables.N, variables.N)
visited = make([]*EventNode, 0)

func See&StronglySEE(x *EventNode, y *EventNode) bool {
    ownerSee = make([]bool, N, N)
    visited = make([]*EventNode, 0)

    if seeAlgorithm{
        return see(x, y)
    }
    if stronglySeeAlgorithm{
        stronglySeeEvent(x,y)
        count := 0
        for each value in ownerSee {
            if value {
                count++
            }
        }
        return count >= 2N/3
    }
}


func see(x *EventNode, y *EventNode, choice int) bool {
    if x.OwnHash == y.OwnHash {
        return true
    }
    if x.Round < y.Round {
        return false
    }

    var a, b bool = false, false
    if x.ParentEvent != nil && !isVisited(x.ParentEvent) {
        visited = append(visited, x.ParentEvent)
        a = see(x.ParentEvent, y, choice)
    }
    if a {
        return a
    }

    if x.PreviousEvent != nil && !isVisited(x.PreviousEvent) {
        visited = append(visited, x.PreviousEvent)
        b = see(x.PreviousEvent, y, choice)
    }

    return a || b
}
```

Fig A.1 : The function who chooses to call see or Strongly see and the see function.

```
func stronglySeeEvent(x *EventNode, y *EventNode, choice int) bool {
    if x.OwnHash == y.OwnHash {
        ownerSee[x.EventMessage.Owner] = true
        return true
    }
    if x.Round < y.Round {
        return false
    }

    ownerSee[x.EventMessage.Owner] = true

    var a, b bool = false, false
    if x.ParentEvent != nil && !isVisited(x.ParentEvent) {
        visited = append(visited, x.ParentEvent)
        a = see(x.ParentEvent, y, choice)
    }
    if a {
        return a
    }

    if x.PreviousEvent != nil && !isVisited(x.PreviousEvent) {
        visited = append(visited, x.PreviousEvent)
        b = see(x.PreviousEvent, y, choice)
    }
    return a || b
}

func isVisited(x *EventNode) bool {
    for _, v := range visited {
        if x.OwnHash == v.OwnHash {
            return true
        }
    }
    return false
}
```

Fig A.2: Pseudocode of stronglySee relation

Appendix B

Here are shown the core protocols used in the algorithm.

B.1 Divide Rounds

```
hasChange = false

for index,EventNode in SortedEvents {
    if index >= end
        return

    if hasChange {
        DivideRoundOnEvent(EventNode, sortedEvents)
        v.Orderedplace = index
    } else {
        if index == EventNode.Orderedplace {
            continue
        } else {
            hasChange = true
            EventNode.Orderedplace = i
            DivideRoundEvent(v, sortedEvents)
        }
    }
}
```

FigB.1: Divide Rounds main routine

```
bool witnessChange
int witnessChangeRound

DivideRoundEvent(eventNode , sortedEvents){

    if eventNode.PreviousEvent == nil && eventNode.EventMessage.Timestamp == 0 {
        eventNode.Round = 1
        eventNode.Witness = true
        insert EventNode in round One Witnesses
    }

    if isOrphan(eventNode) {
        eventNode.Round = -1
        eventNode.Orderedplace = -1
        return
    }

    round = max(eventNode.Parent, eventNode.PreviousEvent)
    numStrongSee = 0

    for _, v := range sortedEvents {
        if numStrongSee >= 2N/3 {
            break
        }
        if v.Round == round && StronglySee(eventNode, v) {
            numStrongSee++
        }
    }

    if numStrongSee >= limit {
        round+=1
    }

    witness = eventNode.PreviousEvent == nil || round > eventNode.PreviousEvent.Round

    eventNode.Round = round
    eventNode.Witness = witness
    if witness
        insert EventNode in round $round Witnesses

}
```

FigB.2: Divide rounds function for each given EventNode

B.2 Decide Fame

```
int maxRoundWitness
int minWitnessChange = witnessChangeRound
int round = max(minWitnessChange-1, 1)

if !witnessChange {
    return
}
bool witnessChange = false

for {

    if round >= maxRoundWitness-2 {
        witnessChangeRound = maxRoundWitness - 2
        witnessChange = true
        return
    }

    currentWitnesses = Witnesses[round]
    nextWitnesses    = Witnesses[round+1]

    for _, current := range currentWitnesses {

        count := 0
        for _, next := range nextWitnesses {
            if see(next, current) {
                count++
            }
        }

        if count >= 2N/3 {
            current.Famous = true
        }
    }

    round++
}
```

FigB.2 : The Decide fame routine

B.3 Find Order

```
func findOrder() {
    int maxRoundWitness
    maxDesiredRound := maxRoundWitness - 3 //inclusive
    if maxDesiredRound < 2 {
        return
    }
    for each EventNode In OrderedEvents{
        for each EventNode2 In SortedEvents{
            if checkSameTransaction(EventNode,EventNode2)
                EventNode2.RoundReceived = EventNode.RoundReceived
        }
    }
    for each EventNode In tempOrder{
        for each EventNode2 In SortedEvents{
            if checkSameTransaction(EventNode,EventNode2)
                EventNode2.RoundReceived = EventNode.RoundReceived
        }
    }
    if !canOrder(currentEvent) {
        continue
    }

    if !seenBy(currentEvent) {
        continue
    }

    for key, currentEvent := range sortedEvents {
        if key >= end {
            break
        }
        if currentEvent.Timestamp == 0 || currentEvent.Round == -1 {
            continue
        }

        if currentEvent.Round > maxDesiredRound || currentEvent.RoundReceived != 0 {
            continue
        }

        eventRound = currentEvent.Round + 1

        for {
            Witnesses = Witnesses[eventRound]
            if there isnt any famous witness in Witnesses
                break

            countSee = Num of Witnesses.Famous who see the currentEvent
            If countSee = allFamousWitnesses
                setRoundReceived(CurrentEvent,eventRound)
                break
            eventRound++
        }
    }
    decideOrder()
}
```

FigB.3 : The find order routine

```
func setRoundReceived(EventNode,round){
    EventNode.RoundReceived = round
    times = empty list of timestamps

    for _, v := range sortedEvents {
        if checkSameTransaction(EventNode, v) {
            v.RoundReceived = round
            if v.Round >= EventNode.Round && v.Round < roundReceived {
                    if see(v,EventNode)
                        times.append(v.timestamp)
            }
        }
    }

    eventsNum = len(times)
    median = int(math.Ceil(float64(eventsNum)/2.0) - 1)
    EventNode.ConsensusTime = times[median]
    tempOrder = append(tempOrder, event)
}
```

FigB.4: Set Round Received Round

```
func decideOrder(){
    SortOrderedEvents(tempOrder)
    currentMaxReceivedRound = maxReceivedRound of an ordered Event

    for each EventNode in tempOrder {
        if v.RoundReceived < maxOrderedRoundReceive{
            OrderedEvents.append(EventNode)
            if !emptyTransaction(v){
                sent Reply to Client
            }
            remove EventNode from tempOrder
        }

    }

}
```

FigB.5: Decide Order Function

Appendix C

Here is shown a different version of some algorithms.

C.1 Send Gossip

```
for{

    syncWith <- random(0-N) && random!= myID

    MU_HashGraph.Lock()

    for each EventNode v in sortedEvents {
        if !v.Know[syncWith]{
            send V to syncWith
            v.Know[syncWith] = true
        }
    }

    MU_HashGraph.Unlock()

}
```

Fig C.1: Alternate Send Gossip routine pseudocode.