b
UNIVERSITÄT
BERN

Fair transaction order in Hedera Hashgraph

Bachelor Thesis

Matteo Raphael Biner
from
Zermatt VS, Switzerland

Faculty of Science, University of Bern

16. March 2022

Prof. Christian Cachin
Nathalie Steinhauer
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Modern blockchain networks are susceptible to front-running attacks. This problem can be solved by imposing a fair order on transactions. There are a number of different approaches to solving this issue. One of them is the Hedera hashgraph algorithm, on which we will focus in this thesis. We will discuss the functionality of the algorithm and look at its fairness, based on different definitions of fairness. We will also show some other algorithms that fulfill their own definitions of fairness.

Acknowledgments

I would like to thank Nathalie Steinhauer for supervising my thesis, providing me with the necessary source material and leading me in the right direction. I would also like to thank Professor Christian Cachin for providing his valuable feedback throughout the writing of this thesis and also providing me with more source material.

Contents

Chapter 1

Introduction

In modern blockchain networks transactions are broadcasted to the network, where nodes (miners) can see all of those transactions. This can lead to undesirable behavior. Let there be a transaction T which tries to buy x amount of a token X. Then a malicious node can add a personal block with a transaction T', which buys y amount of token X, to the blockchain. Since the price will increase after the transaction T, the malicious node can sell y amount of token X at a higher price after the transaction T. This is called a *front-running attack* [5], which is obviously not desired in a blockchain network .
Hedera hashgraph [7] tries to avoid this problem by creating a fair order for transactions. The hashgraph algorithm tries to achieve this by assigning a timestamp to each transaction, which is the median of the times at which the nodes received the transaction for the first time. This way malicious nodes cannot corrupt a timestamp by changing its time by much.
Here is what Hedera claims, concerning fairness [7]: Since the timestamps are fair, the transaction order is also fair. This is important in multiple use cases. Let us for example imagine a situation where two buyers Alice and Bob send transactions that try to buy the last available shares of a stock for the same price. In other blockchains the decision of which transaction to put in a block first is entirely left to the miner. Whereas in hashgraph, the only way for Alice or Bob to ensure that their transaction comes first, is to invest in better internet connection to make sure the transaction reaches all nodes first.
There are also other systems that try to solve the problem mentioned above. One example would be the *Aequitas* protocol, or to be more precise the four similar Aequitas protocols. Kelkar et al. [9] also show an example of how a malicious node could change the order in a protocol like Hedera hashgraph. Another example would be the *Wendy* protocol [10]. In both papers, where these protocols are presented, there is a clear definition of fair ordering. Such a definition is lacking in the Hedera paper, which can make it difficult to compare the different protocols based on fairness.
In Chapter 2, we will explain some of the basic theory that is required to understand the hashgraph algorithm better. In Chapter 3, we will explain the hashgraph algorithm in more detail. Then in Chapter 4, we will look at different definitions of fair ordering, and apply them to the hashgraph algorithm. Finally, in Chapter 5, we will look at other algorithms that satisfy certain definitions of fairness.

Chapter 2

Background

In this chapter, we will look at some basics that will be required to better understand the Hedera hashgraph algorithm. We will specifically look at *state machines* and *fault tolerance* as described by Schneider in [12]. We will also introduce *blockchains* as described by Yaga et al. in [13] and the flaws of *decentralized exchanges* on blockchains as described by Daian et al. in [5].

2.1 State Machines

State machines are the basic building blocks which we need for decentralized exchanges. They consist of state variables that define their state, and commands [12]. Commands can modify the state of the machine or produce outputs. A client of a state machine will send a request that specifies a state machine together with a command, which it should execute. It also contains all the information that is needed to execute the command. Clients can assume two things about the order in which their requests are processed:

- O1: Requests issued by a single client to a given state machine *s* are processed by *s* in the order they were issued.
- O2: If a request *r*, sent to the state machine *s* by a client *c*, could cause a client *c'* to send a request *r'* to *s*, then *s* processes *r* before *r'*.

It is important to note that O1 and O2 do not imply that state machines process requests in the order they were sent or received.

2.2 Fault Tolerance

First we have to define what faulty nodes are. Therefore, we take the definition by Schneider [12]. In principle a node is faulty, if its behavior is not consistent with its specifications. Here, Schneider considers two types of failures nodes can have:

- Byzantine failures, where the node can have arbitrary and malicious behavior and may collude with other faulty nodes.
- Fail-stop failures, where the node responds to a failure by changing its state, such that other nodes may detect that a failure has occurred and then stops.

The systems we discuss in this thesis are mostly designed in a way that they can tolerate a certain amount of byzantine failures. This is due to the fact that we do not necessarily have control over all nodes and that they could always be malicious.
A system is called *t fault tolerant* if it satisfies its specifications if no more than *t* nodes become faulty during a certain amount of time. There are of course more ways to measure fault tolerance. We use *t*

fault tolerance, because this measurement does not rely on the reliability of the components we use. It is only a measure of the reliability of the system architecture.

2.3 Blockchains

As described by Yaga et al. [13], blockchains are digital ledgers implemented in a distributed fashion and usually without a central authority. They enable a community of users to record transactions on a shared ledger, such that no operation on the network can be changed once published.

Ledgers: A ledger is a collection of transactions [13]. Traditionally, ledgers have been stored by a centralized trusted third party. Due to trust, security, and reliability concerns related to ledgers with centralized ownership, there is an increasing interest in ledgers with distributed ownership. Blockchain technology enables distributed ledgers by using both distributed ownership as well as distributed physical architecture.

Blocks: A block in a blockchain often consists of a block header and the block data, but this may vary depending on the implementation [13]. The header itself carries important information like the block number, the previous block headers hash value, a hash representation of the block data, a timestamp, the size of the block, and the nonce value for networks using mining. The block data usually consists of transactions. These blocks are chained together, since they contain the hash digest of the previous block's header, creating the *blockchain*. Changing a block that is in the chain does not work, because that would change its header's hash digest.

Decentralized Exchanges: Blockchains can be used to implement decentralized exchanges. In a decentralized exchange, a smart contract, which is a program executing on a blockchain, executes exchange functionality [5]. Decentralized exchanges seem to be ideally designed. Trades are visible on a blockchain, providing the appearance of transparency. Funds cannot be stolen by the exchange operator, because their custody and exchange logic is processed and guaranteed by the smart contract. But we will later see some weaknesses that decentralized exchanges have, when using a blockchain.

2.4 Flaws of Decentralized Exchanges on Blockchains

In the Introduction, we have already seen an example of a weakness of decentralized exchanges using blockchains. Front-running attacks can enable miners to gain *miner-extractable value*. The exploitation of blockchains with front-running attacks is also called *arbitrage*. This can be done due to the fact that on-chain, smart-contract-mediated trades are relatively slow. This means that traders could attempt to take orders that have already been taken or canceled but appear active due to their views of messages sent on the network. Worse still, adversaries can *frontrun* orders, observing them and placing their own orders with higher fees to ensure they are mined first [5]. This would then lead to the problem described in the Introduction.
Arbitrage can also be automated with bots, because of the atomic batch-based processing of transactions, and because transactions can themselves be initiated by smart contracts. These bots use *proxy contracts*, which can execute *batches* of orders sequentially within a single transaction. This way, they can make sure that the orders are processed in the correct order, thus generating miner-extractable value. Because they use batches of trades, they can throw an exception and revert previous trades, if any of the trades fail. For example a smart contract proxy could execute a trade buying a type-X token for 2 ETH, and another selling it for 3 ETH. If both orders are on the books, on some decentralized exchange, a smart contract executing both guarantees a revenue to the arbitrageur of 1 ETH. But if the second trade fails, the proxy contract can throw an exception, thus reverting the first trade [5].

Chapter 3

The Hashgraph Protocol

3.1 System and Threat Model

Before we can explain the functionality of the hashgraph protocol, we need to introduce some properties and talk about the possible behavior of adversaries. Adversaries are computationally bounded and cannot break the digital signature schemes and cryptographic hash functions used [1]. Furthermore, each node has a key pair for digital signatures, and all nodes know the set of nodes and the public key for each node. Out of n nodes, we assume that more than $2n/3$ are honest. Without loss of generality, we may assume that all malicious nodes may collude and are controlled by a single adversary.
Furthermore we assume that for any two honest nodes A and B, A will eventually try to sync with B, and will eventually succeed in reaching B. No other assumptions are made about network reliability, network speed, or timeout periods. This means that adversaries may delete and delay messages arbitrarily, subject to the constraint that a message between honest nodes that is sent repeatedly must eventually get through.
In the paper, Baird et al. [1] use the following definition for *Atomic Broadcast*:
Atomic broadcast protocols must satisfy the following properties in the face of adversary:

- **Agreement:** If any honest node outputs a transaction, then every honest node outputs that transaction.
- **Total Order:** For all $i > 0$, if T is the ith output of an honest node and T' is the ith output of another honest node, then $T = T'$.
- **Liveness:** If a transaction is input to an honest node, then it is eventually output by every honest node.

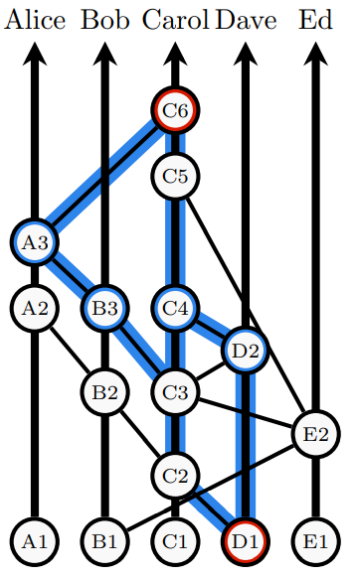
They take this definition from Cachin et al. [2], Miller et al. [11], and Duan et al. [6].

3.2 Hashgraph Algorithm

Communication: One of the goals of the hashgraph algorithm is to avoid frontrunning attacks. In the hashgraph algorithm we have a predefined group of nodes that decide which events are accepted. Events consist of the hashes of two parent events, a list of transactions, a timestamp and a signature for the rest of the tuple [1]. They are basically like the blocks in a blockchain. The timestamp is created using a clock and is only needed for practical use, since we want the order to be fair, and not have a weakness against frontrunning attacks. The algorithm itself would work without timestamps.
Nodes communicate information about these events through gossip. A node *Alice* chooses a node *Bob* at random. Alice tells Bob about all the events that she knows and vice versa. This will later be referred to as synchronization. Then Alice repeats this with a different random node, and so do all other nodes. This way, information spreads through the network. Each time *Alice* synchronizes with a random node, *Alice* creates a new event. This event's parent hashes will be from the latest events from the two nodes.

In Figure 3.1, the parent hashes of C6 would be from A3 and C5.
Nodes only accept events that have a valid signature and contain valid hashes referring to events they already have. Each event's ancestors (parents and parents of parents) can be verifiably recovered. So if two nodes have the same event, they will also have the same ancestors for that event. That way, each node builds up a hashgraph that will eventually be the same for every node. Though this might not hold for newer events at any given time.
In Figure 3.1 we can see an example of such a hashgraph with five nodes Alice, Bob, Carol, Dave, and Ed. The circles represent events and the non-vertical lines represent a parent-child relation where the parent is the event with the lower number. The blue lines represent an example of *strongly seeing*, which we will elaborate on later.

Figure 3.1. Example of a hashgraph [1]



Consistent Output for Ordering: If all nodes have the same hashgraph, we can determine the order of events by using a deterministic function. However, it is possible that nodes have different hashgraphs for newer events, because not all nodes have seen all events. To get consistent outputs, our deterministic function should do nothing, in the case that not all nodes have the same hashgraph.
For that we use the concept of ancestry. For example, a node that has all the events in Figure 3.1 can determine that D1 is an ancestor of C6. But a node like Bob has not yet seen C6, and cannot confirm that D1 is an ancestor of C6. In that case C6 can not yet be accepted. However, once every node has seen C6, every node will be able to confirm that D1 is an ancestor of C6.
Thus, ancestry is an example of a deterministic function of a hashgraph that at any given moment either gives no conclusion at all, or it makes a conclusion that will never change, and all nodes eventually come to that same conclusion [1].

Ordering: For the ordering process, the algorithm uses virtual voting. This voting system relies on *seeing* and *strongly seeing*. We say that an event *x sees* an event *y*, if *y* is an ancestor of *x* and *y*'s ancestors do not include a *fork* by *y*'s creator. A *fork* occurs, if a node creates two events with the same parent and gossips them to different nodes, which is malicious behavior. An event *x strongly sees* an event *y*, if *x* can see *y* through events that were created by more than *2n/3* different nodes, each of which can *see y* [1]. To give an example, in Figure 3.1 C6 strongly sees D1.
To determine event ordering, each node performs three operations on its own hashgraph [1]:

- *DivideRounds* divides the hashgraph into rounds for voting and an event gets into a new round when it strongly sees events in the prior round on more than *2n/3* of the nodes. To give an example

in Figure 3.1, all events are in round 0, except C6. C6 strongly sees the events B1, C1, D1, and E1. The initial events are all in round 0. The voting focuses on *witnesses*, which are the first events created by a node in a round [1]. If a node has not heard from other nodes in a while, it is possible for it to skip one or more rounds when it catches up. Thus a node might not have a witness in any particular round. A dishonest node could have multiple witnesses in a round, but only one of them can be strongly seen [4].

- *DecideFame* is the voting protocol that decides which witnesses are famous, meaning they have been received by many nodes at the start of the round. For each witness x in round r, each witness in round $r+1$ will vote for x if it can see x [1]. In successive rounds after $r+1$, each voter votes the way it observed the majority vote in the previous round. A round of an election is *nearly unanimous* if voters on more than $2n/3$ of nodes vote the same way. If a voter ever observes a nearly unanimous result in the previous round, then the vote in the current round will be unanimous. Thus we can end the election as soon as any event observes a nearly unanimous result in the previous round. There are also *coin rounds* for theoretical security reasons which we will discuss later [4].

- *FindOrder* orders the events, once consensus is reached that they are famous. FindOrder then determines a consensus timestamp and a consensus total order on older events. For an event x it first calculates the *round received* of x which is the round in which all the unique famous witnesses are descendants of x. Then it calculates the *received time*. For each unique famous witness y, the algorithm looks for the earliest parent z of y that was created by the same node and has seen x. The timestamp that any node that created a unique famous witness, assigns to such a z, is the time it claims to have seen x. This means it is a clock-based timestamp. The *received time* of x is then the median of all those timestamps. Then the consensus order is calculated. All events are first sorted by the round received and if that is the same round, they are sorted by their received time. If there are still ties, the order is calculated by any deterministic method [1].

Consensus Order: Now as described in [4], we have a clear definition of the consensus order: Suppose x and y are events. Let i and i' be the rounds in which x and y are determined to be in. Let t and t' be the consensus timestamps of x and y. Then x precedes y in the consensus order if:

- $i < i'$, or
- $i = i'$ and $t < t'$, or
- $i = i'$ and $t = t'$ and x is less than y using the arbitrary comparison used to select unique famous witnesses.

It is clear that the consensus order is a total order. Since all nodes can agree on the unique famous witnesses, they can all agree on the consensus order. Finally, the order is immutable: the unique famous witnesses do not change once determined, and their ancestries never change either.

Main Procedure: The main procedure of the hashgraph protocol is a combination of the communication and the ordering, described above. It looks as follows [1]:

3.3 Security Discussion

Forks: Forks, as described above, are only possible through malicious behavior. Let us assume Alice creates nodes y and z that both have the same parent x, and neither y nor z is a parent of the other. Alice could then gossip y to Bob and z to Carol. This would mean that Bob's and Carol's hashgraphs do not look the same, because one contains y and not z and vice versa. To detect and prevent such forks, Bob and Carol need to know what other nodes' view of the hashgraph is. Strongly seeing ensures that even if an attacker tries to cheat by forking, they cannot make different nodes strongly see different events. If y

Algorithm 1 Main procedure of the hashgraph protocol

```
1: while True do
2:     select a node at random
3:     sync all events with that node
4:     create a new event
5:     divideRounds
6:     decideFame
7:     findOrder
```

and z are on different branches of a fork, then x can strongly see y or z, but not both [1].
To explain this, we have to remind ourselves that x strongly seeing y means that x can see y through more than $2n/3$ nodes that can see y. The same holds for x strongly seeing z. Thus the set of events through which x sees y and the set of events through which x sees z overlap in more than $n/3$ nodes. Since fewer than $n/3$ nodes are malicious, given by byzantine fault tolerance, at least one node in the overlap must be honest. That node cannot see both y and z since they are on a fork [1].

Preventing Consensus: An adversary with enough control over the network could try to prevent the algorithm from reaching a consensus. To solve this problem, the algorithm is implemented with *coin rounds*. In these rounds, every voter who sees a nearly unanimous result in the previous round will continue to vote with the majority. However, the remaining voters will determine their votes using a coin flip. Eventually, the voters that voted randomly, will vote the same way as the others. The following round, every voter will see a unanimous result and the election will end [4].

Chapter 4

Fair Order Properties

4.1 Order Fairness

4.1.1 Wendy

The Wendy algorithm as defined by Kursawe [10] is a good point to start from when talking about fairness. For this algorithm, they use the following, most commonly used, definition of fairness:

- Every request eventually gets scheduled.
- Every request gets scheduled within a bounded time or number of implementation related messages.

Now for order fairness we could have a very simple definition: A byzantine fault tolerant total ordering protocol is called *order fair* if the following holds: If all honest parties receive request r_1 before request r_2, then r_1 is delivered before r_2. However this definition is not only impossible to achieve, it is inherently contradictory even if only one node is corrupt. A proof sketch for this is provided in [10]. We will show a shorter version of that proof sketch as follows:

Proof sketch: Suppose there are n parties $P_1, P_2, ..., P_n$ and n requests $r_1, r_2, ..., r_n$. For this proof sketch we will think of r_1 as r_{n+1} and r_n as r_{1-1}. Then let every party P_i receive the transaction requests in the order $r_i, r_{i+1}, ..., r_n, r_1, ..., r_{i-1}$. Now for every j, there is exactly one party P_j, such that P_j sees r_j before r_{j-1}. So if all parties are honest there can never be a dedicated message order. If one party P_j is dishonest, then r_j must be ordered after r_{j-1}. Since the parties do not know which parties are dishonest, this must be the case for all parties. So for every i, r_i must come after r_{i-1}, which is a contradiction.

A better definition would be the following, which we will discuss later in more detail: A byzantine fault tolerant total ordering protocol is called *order fair* if the following holds: If all honest parties receive request r_1 before r_2, then r_1 is delivered in the same block as r_2 or earlier. In this case, we will no longer have undecidability problems, but we can still not guarantee fairness. We can show that we cannot guarantee termination. The full proof of this is provided in [10]. The idea behind the proof is that we create a schedule that would create a block of unlimited size. We do that by taking the idea of the earlier proof and force an arbitrarily large amount of requests in the same block.

4.1.2 Order Fairness for Byzantine Consensus

Kelkar et al. [9] discuss different definitions for order-fairness. In this section, we will look at four of those.

Send-Order-Fairness: In Section 4.1.1, they immediately talk about receiving transactions. One could also define order fairness in terms of send-order. For instance, if a transaction x is sent before a transaction

y, then x should appear before y in the agreed upon log [9]. We have to note that x and y can be sent by different clients. For this definition to work, we would need a trusted way of timestamping a transaction at the client side. Assuming we had such a timestamp mechanism, network synchrony would still be required to ensure a transaction cannot be delayed arbitrarily [9]

Receive-Order-Fairness: Kelkar et al. [9] take a slightly different approach to receive-order-fairness than what we have seen in Section 4.1.1. Instead they say that fair ordering is defined by looking at when enough nodes receive a particular transaction. They use γ as a parameter for what enough nodes means. So if γn nodes receive a request x before a different request y, then x is delivered before y by any honest node. This definition runs into the same problem as we have discussed in Section 4.1.1. In the paper they describe some scenarios, in which this definition works well. We will not go further into this, since those scenarios do not fit the assumptions made about the Hedera hashgraph protocol [9].

Approximate-Order-Fairness: This is a weaker definition of order-fairness. It focuses only on unfairness in the ordering of two transactions if they were received sufficiently apart in time. It also only makes sense in synchronous and partially synchronous settings. Here they introduce a new parameter ξ, that defines the number of rounds two transactions are apart. Approximate-order-fairness is satisfied if the following holds:

- For any two transactions x and y, let n be the number of nodes that received both transactions in a fixed timeframe. If at least γn nodes receive x more than ξ rounds before y, then all honest nodes do not deliver y, unless they have previously delivered x.

This definition is still not very useful, since it suffers from the same problems as receive-order-fairness discussed in Section 4.1.1 [9].

Block-Order-Fairness: In this definition Kelkar et al. [9] choose to ignore the ordering of two transactions within a block. This allows them to circumvent the problem described in Section 4.1.1 by aggregating any transactions with non-transitive orderings into the same block. Block-order-fairness is satisfied if the following holds:

- For any two transactions x and y, let n be the number of nodes that received both transactions in a fixed timeframe and $\gamma > \frac{1}{2}$. If at least γn nodes receive x before y, then all honest nodes do not deliver x in a later block than y.

This definition is almost equivalent to the second definition in Section 4.1.1. Instead all honest nodes receiving one transaction before the other, here a majority γ receives one transaction before the other [9].

4.1.3 Quick Order Fairness

Cachin et al. [3] take a slightly different approach to defining order-fairness and give two definitions in the process.

Weak Differential Order-Fairness: In this paper they first introduce a function $b(x, y)$, that denotes the number of honest nodes, that order-fair-broadcast x before y. They further assume w.l.o.g. that an honest node will eventually order-fair-deliver x and y and that, therefore, $b(x, y) + b(y, x) = n - f$, where n is the total number of nodes and f is the number of byzantine nodes. Now the goal is to achieve that if $b(x, y) > b(y, x)$, then no honest node will order-fair-deliver y before x. Weak differential order-fairness is then defined as follows for some $\mu > 0$: For any transactions x and y, if $b(x, y) > b(y, x) + \mu$, then no honest node atomic-delivers y before x. This definition is not strong enough, because $\mu \geq 2f$ needs to hold. We leave out the proof for this here, since it is not important for this thesis [3].

κ-Differentially Order-Fair Atomic Broadcast: A protocol for κ-differentially oder-fair atomic broadcast satisfies the properties *agreement* and *total order* as defined in Section 3.1 and additionally:

- **No duplication:** No transaction is atomic-delivered more than once.
- **Strong validity:** If *more than f* honest nodes order-fair-broadcast a transaction x, then every honest node eventually order-fair-delivers x.
- **κ-differential order fairness:** If $b(x, y) > b(y, x) + 2f + \kappa$, then no honest node order-fair-delivers y before x.

Compared to the definition of weak differential order fairness, they have $\kappa = \mu - 2f$ [3].

4.2 Fairness in Hedera

4.2.1 Definition of Fairness according to Hedera

Baird et al. [1] never exactly define fairness in the Hashgraph paper. In fact fairness is never mentioned. On their website [7], they explain three different concepts of fairness. Those are *fair access, fair timestamps,* and *fair transaction order*. Fair access means that every transaction has the chance to get into the system. Fair timestamps means that each transaction is given a timestamp that cannot be changed by much. Fair transaction order means that no attacker can change the order of transactions [7].
Crary [4] shows a more formal definition of fairness, that is very specific to the hashgraph protocol. For this definition we will first need to look at some terms, they introduced earlier. They use the parameter d, and the notion of a *supermajor* set. A witnesses election begins d rounds after its own round. A *supermajor* set is a set of events whose creators constitute a supermajority, meaning more than 2/3 of the nodes.
Their definition comes in the form of a theorem: *If $d \geq 2$ then every round's set of famous witnesses is supermajor* [4].
This implies that a majority of the famous witnesses in any round must be honest, because less than 1/3 of the nodes can be dishonest. Consequently, every event's consensus timestamp, which is the median of all the suggested timestamps, is governed by honest nodes. The timestamp might come from a dishonest nodes, but even if so, it will be bracketed on both sides by timestamps from honest nodes [4].
This definition is very different from the ones we have seen in Section 4.1, since it only focuses on fair timestamps. It shows that an events timestamp must be within the limits of what honest nodes think it should be. This goes into the direction of send-order-fairness as we have seen in Section 4.1.2. We will shortly see cases where this definition does not fulfill send-order-fairness.

4.2.2 Using the Definition from the Wendy Algorithm

Kursawe [10] requires all honest nodes to receive one transaction before the other. For an adversary that controls most of the dishonest nodes, we can give an example that does not violate the theorem in Section 4.2.1, but violates the definition in the Wendy paper, in the case that two transactions are delivered on different blocks:

- **Example 1:** Let x be the event that is sent first, and y be the event that is sent second. Now let t_0 be the time at which the honest node n_0 learns about the event e_x that contains x, and $t_n > t_0$ be the time at which the honest node n_n learns about that same event. We further assume all honest nodes learn about e_x between t_0 and t_n. We now assume the same for y where t_0' and t_n' are the times at which n_0 and n_n learn about the event e_y respectively, and that there is a significantly large overlap between $[t_0{:}t_n]$ and $[t_0'{:}t_n']$ such that the median of the honest node's timestamps for e_x and e_y are close. We also assume that all honest nodes learn of e_x before e_y. Then we consider an adversary, that controls a large amount of dishonest nodes. Such an adversary can set all timestamps for e_x later than t_n, and all timestamps for e_y earlier than t_0'. This way e_x, in the end, might receive a timestamp that is later than the timestamp for e_y.

We note that this example is very unlikely to happen, but it is still theoretically possible, since only a majority of the timestamps come from honest nodes.

4.2.3 Using the Definitions in Order Fairness for Byzantine Consensus

Here we will look at the compatability of the hashgraph algorithm with the definitions described by Kelkar et al. [9].

Send-Order-Fairness: In the description in Section 4.1.2 it is mentioned that send-order-fairness requires a fair timestamping mechanism at the client side. In the hashgraph algorithm, there is no timestamping at the client side, but a timestamp for every event. Since the ordering of transactions on a single event is arbitrary, the hashgraph algorithm will never be able to achieve send-order-fairness. We can even give an example situation where send-order-fairness is not achieved for transactions on different events. For that, we use **Example 1** given in Section 4.2.2 with a few modifications. Here we do not require all honest nodes to have received e_x and e_y. We do also not need to require all honest nodes except n_0 to have received e_x before e_y. This way, send-order-fairness can be violated, even without an adversary.

Approximate-Order-Fairness: As discussed in Section 4.1.2, we will not discuss receive-order-fairness, as it only works in scenarios that do not match the Hedera hashgraph algorithm. Approximate-oder-fairness can be applied though, since the hashgraph algorithm already uses rounds. For an event e_0 in round 0 to be validated in the hashgraph algorithm, it must receive votes from other events in a later round. Let us now consider a different event e_ξ that contains a transaction that was received ξ rounds after a transaction on e_0. It is clear that e_ξ cannot be delivered before e_0, because it would need to receive votes from events, that are in an earlier round than ξ, which is not possible. Therefore approximate-order-fairness holds for the hashgraph algorithm, if we define rounds as they are defined in the hashgraph algorithm. In the original definition of approximate-order-fairness by Kelkar et al. [9] though, they say that this definition is only applicable to synchronous protocols, which indicates that rounds would need to be split by a full synchronization, which the hashgraph algorithm does not have.
As mentioned by Kelkar et al. [9], this definition is not very useful anyway, since it suffers from the same problem as receive-order-fairness, which is the same as we have discussed in Section 4.1.1.

Block-Order-Fairness: Here we can once again refer to **Example 1** described in Section 4.2.2. If two transactions are on different events, and both events are in the same round, then they can be ordered the wrong way around. **Example 1** is not the only example where the hashgraph algorithm fails the block-order-fairness definition. In the paper by Kelkar et al. [9] there is a different example concerning protocols that use timestamps for ordering. Here we adapt this example slightly such that it fits the hashgraph algorithm.

- **Example 2:** We consider 5 nodes *A, B, C, D* and *E*, where *E* is a malicious node, and two events e_x and e_y, that contain the transactions *x*, and *y* respectively. The event e_x is received by nodes *A,..., E* at times 1, 1, 4, 4, 2 while e_y is received by the nodes at times 2, 2, 5, 5, 3. Now all the nodes have received e_x before e_y and we also assume that e_x and e_y are not the same event. Then e_x should be delivered before e_y if all nodes gave honest timestamps. However, notice how *E* can invert the timestamps it gives to e_x and e_y, since it both times sets the median timestamp. Thus e_x may be delivered after e_y.

Example 2 also violates block-order-fairness, among other definitions, if *x* is received before *y* by γn nodes. Note that both examples do not violate the fairness definition discussed in Section 4.2.1.

4.2.4 Using the definitions in Quick Order Fairness

Here we will look at the compatability of the hashgraph algorithm with the definitions described by Cachin et al. [3].

Weak Differential Order-Fairness: We will first look at **Example 1** as described in Section 4.2.2. Here $b(x, y) > b(y, x) + \mu = \mu$ holds for $\mu < 2n/3$. Because of $b(x, y) + b(y, x) = n - f \geq 2n/3$, this is always the case. Hence the hashgraph algorithm is not order-fair according to this definition.
We can also look at **Example 2** as described in Section 4.2.3. Here $b(x, y) > b(y, x) + \mu$ clearly holds for $\mu < 4$. Since $b(x, y) \leq n - f$, it would make no sense to define $\mu \geq 4$. So we get to the same conclusion for **Example 2**.

κ-Differential Order-Fairness: For **Example 1**, as described in Section 4.2.2, we need to make one further assumption. We assume that f is small enough, such that $b(x, y) > b(y, x) + 2f + \kappa$ holds. We can make that assumption because for $f = (n-1)/3$ and $\kappa > 0$, the definition would not be applicable. If the number of malicious nodes in **Example 1** is smaller, we can just assume that the medians of all honest node's timestamps for x and y are closer. That way it is still possible for some nodes to deliver y before x.
For **Example 2** as mentioned in Section 4.2.3, we use the same argument as for weak differential order-fairness. Since only one malicious node is required to disturb the order, we can add arbitrarily many honest nodes to $b(x, y)$ for $b(x, y) > b(y, x) + 2f + \kappa$ to hold.

4.2.5 Likelihood of Examples 1 and 2

As we have seen, the hashgraph algorithm can in theory order transactions unfairly, according to most definitions of fair ordering. We should also look at how likely it is for situations like **Example 1** or **Example 2** to occur. In addition to that, it is also relevant whether or not these weaknesses can easily be exploited by adversaries.
For **Example 1** it is clear to see that an adversary cannot switch the order of arbitrary events, because there needs to be an overlap in the timestamps for each event. If there is an overlap, the adversary needs to generate and communicate a large amount of fake timestamps. Especially for the timestamps that are later than the honest timestamps that is a problem. Dishonest nodes with fake timestamps for an event e need to create events e_{late}, that have very late timestamps for e. Now if the final timestamp for the event e is determined by events in round r, then all events e_{late} need to become unique famous witnesses of round r. Since the adversary does not control more than 1/3 of the nodes, it is not possible to ensure that this is always possible. Even if the adversary has enough unique famous witnesses in a certain round to influence a vote, the adversary is still restricted in the choice of events that can be reordered.
For **Example 2** it is relatively hard for an adversary to exploit the weakness. An adversary can only switch the order of two transactions, if a few conditions are met. The adversary has to create timestamps t_m and t'_m for events e_x and e_x', that are in both cases the median timestamps of all timestamps for each event. Now if we consider the timestamps t_{m-1} and t_{m+1} for e_x and the timestamps t'_{m-1} and t'_{m+1} for e_x', it is easy to see that there needs to be an overlap between $[t_{m-1};t_{m+1}]$ and $[t'_{m-1};t'_{m+1}]$ such that t_m and t'_m fit inside that overlap. Only in that case can the order of the two transactions be switched around. This overlap cannot be influenced by the adversary for arbitrary events. An adversary could still profit from switching two transactions on almost random events though.

Chapter 5

Other Algorithms

5.1 Aequitas

The Aequitas protocols are described in the paper *Order-Fairness for Byzantine Consensus* by Kelkar et al. [9]. There are four different Aequitas protocols:

- $\Pi^{\text{sync, nolead}}$, a leaderless protocol that provides consistency, weak-liveness, and block-order-fairness in the completely synchronous setting.
- $\Pi^{\text{sync, lead}}$, a leader-based protocol that provides consistency, weak-liveness, and block-order-fairness in the completely synchronous setting.
- $\Pi^{\text{async, nolead}}$, a leaderless protocol that provides consistency, eventual-weak-liveness, and block-order-fairness in any setting.
- $\Pi^{\text{async, lead}}$, a leader-based protocol that provides consistency, eventual-weak-liveness, and block-order-fairness in any setting.

Here we will give an overview of the functionality that the synchronous and asynchronous protocols have in common. We will only really differentiate between leaderless, and leader-based protocols. A more precise description can be found in the paper by Kelkar et al. [9].

Functionality: The protocol is divided into three steps: Gossip / Broadcast, Agreement on local logs, and Finalization. The difference between the leaderless and leader-based protocols will only be seen in the third stage.

- **Stage I: Gossip / Broadcast.** Each node FIFO-broadcasts transactions as they are received as input from the environment. In parallel to broadcasting transactions, a node also receives and processes broadcasts from other nodes. For a node i, broadcasts sent by a node j are appended to a local log when they get delivered to i. Intuitively, this log denotes node i's view of how transactions were received by node j [9].
- **Stage II: Agreement on local logs.** To determine the ordering of a transaction x, a node i waits until it has received x from sufficiently many other nodes. The properties of the FIFO-broadcast defined by Kelkar et al. [9] guarantee that if two honest nodes i and j have a transaction x_k from a different node k on their logs, then both logs are identical until x_k occurs.
  Now all nodes need to agree on which local logs to use to determine the ordering for a transaction x. For this each node starts an instance of a different protocol, using the set of nodes that have broadcast x.
  Finally, at the end of the agreement phase, every honest node has agreed on a set of nodes whose transaction orderings should be used to determine the final ordering for the transaction x in consideration [9].

- **Stage III: Finalization.** To decide the final ordering for a transaction x, nodes first build a graph that represents any ordering dependencies between transactions. The graph consists of vertices that represent transactions and directed edges that represent ordering dependencies. Specifically, an edge from a transaction x to a transaction y denotes that x appeared before y on sufficiently many local logs. It is guaranteed that every edge that exists on a node i, will eventually exist on a node j, if both nodes are honest. It is not guaranteed though, that the graph is acyclic or complete. To break ties between transactions without an edge, they differentiate between a leaderless and leader-based finalization.
  In the leader-based finalization, a leader proposes and broadcasts a new block. Instead of just checking the syntactical validity of transactions, each node i checks that the proposal does not conflict with any required order-fairness in the graph.
  For the leaderless finalization, they suggest a finalization via local computation. For two transactions that have no edge between them in the graph, the protocol will wait until both transactions have a common descendant. The ordering is then based on which transaction has the most descendants.
  This description of the finalization stage is simplified and does not solve the Condorcet paradox. The Condorcet paradox is basically the problem described in Section 4.1.1. Furthermore, adversarial transactions could result in a node waiting for unbounded periods of time [9].

Order-Fairness: For the ordering, we only have to look at the finalization stage. As described above, when ordering two transactions x and y, each node waits until one of the transactions appears first on sufficiently many logs. If x appeared before y on very few logs, the node waits for y to be delivered before it delivers x. Therefore, the Condorcet paradox can cause non-transitive waiting, where the transactions are on a cycle in the algorithm's graph. The solutions for transactions on a cycle is to deliver them at the same time by placing them in the same block. To be exact, the algorithm places all strongly connected components of the graph in the same block. In a strongly connected component, each vertex can reach each other vertex and it is a maximal subgraph [9]. We will not go into detail how the different versions of Aequitas deal with transactions that are not connected on the graph, since that would be beyond the scope of this thesis.
The Aequitas protocols clearly are block-order-fair. Intuitively, if more than γn nodes receive a transaction x before a transaction y, then each node waits before delivering y unless x and y are both on the same cycle, in which case they will be delivered in the same block. The formal proofs for block-order-fairness can be found in the paper by Kelkar et al. [9].

5.2 Themis

The Themis protocol is another protocol presented by Kelkar et al. in a separate paper [8]. The Themis protocol is based on the leader-based Aequitas protocol. It can also be used on top of any leader-based protocol. Themis gives all nodes a way of checking the ordering of transactions in the leader's proposal, whereas in standard protocols, nodes only check the validity of transactions.

Functionality: To enable the leader to construct a fair ordering, all nodes will first submit their local transaction orderings to the leader. The leader then constructs a fair-ordered proposal from a set of local orderings. The algorithm used for this will guarantee that most transactions are proposed but at the same time, fairness is not violated even when a new leader needs to continue the ordering proposed by an earlier leader. To be exact any honest leader uses the *FairPropose* and *FairUpdate* algorithms which are described in the Themis paper by Kelkar et al. [8]. Here we will only give an overview of the *FairPropose* algorithm.
The *FairPropose* algorithm's goal is to propose as many transactions as possible while making sure that the exclusion of any transaction from the proposal does not violate fairness. The algorithm first constructs a dependency graph G by adding a vertex for each non-blank transaction. An edge between

two transactions (x, y) is added to the graph whenever x appears before y on sufficiently many local orderings. If this is also the case for y, then only the edge that corresponds to more local orderings is added. Then the condensation of the graph G is computed, which is basically the same process that avoids the Condorcet paradox in the Aequitas algorithm. The algorithm then creates the graph S by sorting the new graph topologically. Let now V be the last vertex in S that contains a solid transaction. The algorithm then removes all transactions from G, that are part of vertices after V in S. Finally, the algorithm outputs the graph G [8].

Order-Fairness: In the Themis paper by Kelkar et al. [8], they use the definition of *batch-order-fairness*, which is in essence equivalent to block-order-fairness. They also provide proof that the Themis protocol satisfies batch-order-fairness. For that proof we will first introduce a lemma from the Themis paper:

Lemma 5.2: If *FairPropose* outputs a transaction x and does not output y and if y is received before x by more than γn nodes, then x and y are in the same potential Condorcet cycle.

The full proof for this lemma can be found in the Themis paper by Kelkar et al. [8] and is not relevant for this thesis.

We will now give the most relevant part of the proof that Themis satisfies batch-order-fairness: Consider transactions x and y such that x was received before y by at least γn nodes. Now, we follow a few cases:

- (Case 1) At some time, the current correct leader proposal includes x and y is not included in this or any earlier proposal. Now, by Lemma 5.2, we now that y must be in the same or later Condorcet cycle as x.
- (Case 2) At some time, the current correct leader proposal includes y and x is not included in this or any earlier proposal. Since we are also given that x was received before y by γn nodes, by Lemma 5.2, we now that y must be in the same potential Condorcet cycle as x.
- (Case 3) At some time, the current correct leader proposal includes both x and y. Since x was received before y by γn nodes, we know that there is an edge from x to y within the dependency graph. Therefore, once again, either y will be output in the same potential cycle as x or a later one.

The above cases show that the final transaction ordering can be split into contiguous potential Condorcet cycles, and consequently, it will hold that batch-order-fairness is satisfied.

5.3 Quick Order Fairness

The Quick Order Fairness protocol was presented by Cachin et al. [3]. The protocol runs a Byzantine FIFO consistent broadcast channel (BCCH) and uses validated Byzantine consensus (VBC), which are defined in the paper. In this thesis, we will not go into detail about the definitions of BCCH and VBC. The Quick Order Fairness protocol uses some ideas that are similar to the previous protocols, but complies with its own definition of order fairness that we have seen in Section 4.1.3. In the paper, they talk about payload messages instead of transactions.

Functionality: An incoming of-broadcast event with a payload message m triggers BCCH and bcch-broadcasts m to the network. Every node keeps a local vector clock that counts the payloads that have been bcch-delivered from each sending node. Every node also maintains an array of lists *msgs* that record all payloads from each node. When a node bcch-delivers the payload message m, the vector clock and the list *msgs* are updated accordingly. As soon as sufficiently many new payloads are found in *msgs*, a

new round starts and each node signs its vector clock and sends it to all other nodes. The received vector clocks are collected in a matrix. A new VBC instance is triggered once $n - f$ valid vector clocks are recorded. Here f refers to the number of faulty or dishonest nodes. The node proposes the matrix and the signatures for consensus, and VBC decides on a common matrix with valid signatures. This matrix defines a *cut*, which is a vector of indices, with one index per node. This index indicates up to which entry in *msgs* payload messages are considered for creating the fair order in the round.
Once all nodes received the payloads up to the cut, the algorithm starts building a graph similar to the graph in the Aequitas protocol. The difference is that here the vertices are all new payload messages defined by the cut and an edge (*m*, *m'*) indicates that *m* should be at most be of-delivered before *m'*.
For the construction of the graph, the node first creates a vertex for every payload message that appears in more than f distinct lists in *msgs*. Then the algorithm builds a matrix M such that M_{ij} counts how many times m_i appears before m_j in *msgs* up to the cut. If the difference between entries M_{ij} and M_{ji} is large enough, the protocol adds a directed edge (m_i, m_j) to the graph [3].

Order Fairness: Here they use their definition of κ-differential order fairness. The precise requirement for adding an edge from m_i to m_j is that $M_{ij} > M_{ji} - f + \kappa$. Then they show that if $b(m_i, m_j) > b(m_j, m_i) + 2f + \kappa$, then $M_{ij} > M_{ji} - f + \kappa$ holds. For κ-differential order fairness to hold, it must be guaranteed that m_j is not of-delivered before m_i whenever $M_{ij} > M_{ji} - f + \kappa$ holds. The full proof can be found in the paper by Cachin et al. [3]. We show the basic idea behind it as follows:

Proof sketch: Since M cannot report on more than f incorrect processes, it follows that $b(m_i, m_j) \geq M_{ij} - f$. Furthermore, because M_{ij} can include reports about m_i or m_j in *msgs* from up to f incorrect processes, $b(m_i, m_j) \leq M_{ij} + 2f$. The conclusion then follows from a simple calculation.

There can still be cycles in the graph that was constructed this way. Similar to the other algorithms we have seen, the Quick Order Fairness algorithm collapses the graph such that all payload messages in cycles will be delivered together as a set. In the end, the algorithm selects all vertices without an incoming edge and sorts them in a deterministic way. After the corresponding payload messages are of-delivered, the processed vertices are removed from the graph and another iteration through the graph starts. As soon as there are no vertices left, the protocol proceeds to the next round.

5.4 Pompē

The Pompē algorithm is introduced in the Byzantine Ordered Consensus without Byzantine Oligarchy paper by Zhang et al. [14]. It is explicitly designed for Byzantine ordered consensus that preserves the same interface as a standard BFT protocol.

Functionality: The algorithm is divided into an *ordering phase* and a *consensus phase*. Here we will focus on the ordering phase but also give an overview of the consensus phase.

- **Ordering phase:** Pompē uses timestamps as ordering indicators. In a first step, a node n_i with a transaction x collects signed timestamps for x from $2f + 1$ nodes, where f is the maximum amount of byzantine nodes. The median of those timestamps is then assigned to x and determines the position of x in the total order. In the second step, the node broadcasts x along with its timestamp and waits for it to be accepted by $2f + 1$ nodes. Once that happens, the transaction x is then guaranteed to be included in the totally-ordered ledgers of correct nodes, and its position is guaranteed to be determined by the given timestamp. In the paper, they refer to such transactions as *sequenced*.
  A node n_j will accept a transaction x, only if the timestamp of x is higher than the node's *localAcceptThresholdTS*. The *localAcceptThresholdTS* of a node n_j is an integer that tracks what n_j believes to be the latest possible timestamp of any transaction in the ledger.
  A more precise description of all datastructures used by each node can be found in the paper [14].

- **Consensus phase:** The goal of the consensus phase is to ensure that all correct nodes agree that a certain prefix of the total order constructed in the previous phase is now forever immutable. To accomplish this, Pompē employs any leader-based BFT SMR protocol that offers a primitive to agree on a value for each slot in a sequence of consensus slots. We will not go into more detail about the consensus phase here, since it is not important for this thesis.

Order Fairness: Zhang et al. [14] show a number of theorems and lemmas concerning safety and liveness. There are also two statements that affect order fairness. Like the Hashgraph protocol, the assigned timestamp for a transaction is bounded by timestamps from correct nodes. In addition to that, they also show that if there is no overlap between the possible timestamps for two transaction, the ordering of two transactions relative to each other is fixed. The latter property is referred to as ordering linearizability, which is a different definition of order fairness.
Since the Pompē algorithm also uses timestamps, it can suffer from the problems we talked about in the examples in Sections 4.2.2 and 4.2.3. In the Themis paper by Kelkar et al. [8], they also refer to Pompē's transaction ordering as *ordering linearizability* and not necessarily *batch-order-fairness* or *block-order-fairness*. The proof that the Pompē protocol's order is linearizable can be found in the paper by Zhang et al. [14].

Chapter 6

Conclusion

In this thesis, we have given an overview of how the Hedera hashgraph protocol works. We did not look at all the details of its functionality, since we also wanted to look at its order-fairness in particular. This was challenging because there are many definitions of order-fairness. Since order-fairness was not mentioned in the original *Hashgraph Protocol* paper by Baird et al. [1], we tried to apply different definitions to the Hashgraph protocol, as well as a definition found in the *Verifying the Hashgraph Consensus Algorithm* paper by Crary [4]. This was not always very straightforward, because some definitions are somewhat specific to the protocol they were meant for.
The essential idea behind the definitions presented in most papers is some form of block-order-fairness. We have mostly seen how the Hashgraph protocol does not comply with this notion of order-fairness. We may assume though, that the cases in which the Hashgraph protocol is not block-order-fair, are relatively rare. Compared to traditional blockchains, the Hashgraph protocol is certainly not as exploitable in that regard.
To give a comparison to other algorithms, we have shown some algorithms that are block-order-fair, as well as one that also is not block-order-fair. It was interesting to see that most of those other algorithms were not as complex as the Hashgraph algorithm. Though we did not cover them in nearly as much detail as the Hashgraph algorithm.
Further work may include looking at more definitions of order-fairness that the Hashgraph algorithm complies with. It would also be interesting to see, to what extend the algorithm's ordering can be exploited. This would likely have to include a more precise description on how transactions are placed into events.

Bibliography

[1] Leemon Baird and Atul Luykx. The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*, pages 1–7. IEEE, 2020. https://doi.org/10.1109/COINS49042.2020.9191430.

[2] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[3] Christian Cachin, Jovana Micic, and Nathalie Steinhauer. Quick order fairness. *CoRR*, abs/2112.06615, 2021. https://arxiv.org/abs/2112.06615.

[4] Karl Crary. Verifying the hashgraph consensus algorithm. *CoRR*, abs/2102.01167, 2021. https://arxiv.org/abs/2102.01167.

[5] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 910–927. IEEE, 2020. https://doi.org/10.1109/SP40000.2020.00040.

[6] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: asynchronous BFT made practical. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[7] Hedera. https://docs.hedera.com/guides/core-concepts/hashgraph-consensus-algorithms#fairness, November 2021. Last accessed in November 2021, https://docs.hedera.com/guides/core-concepts/hashgraph-consensus-algorithms{#}fairness.

[8] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *IACR Cryptol. ePrint Arch.*, page 1465, 2021.

[9] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 451–480. Springer, 2020.

[10] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pages 25–36. ACM, 2020. https://doi.org/10.1145/3419614.3423263.

[11] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[12] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. https://doi.org/10.1145/98163.98167.

[13] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. Technical report, National Institute of Standards and Technology U.S. Department of Commerce, 2018.

[14] Yunhao Zhang, Srinath T. V. Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 633–649. USENIX Association, 2020.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 16.03.2022
Ort/Datum


Unterschrift