Bachelorarbeit

A Formal Analysis Of Hashgraph And Its Accountability Properties

Marcel Flinspach

Course of Study: Informatik
Examiner: Prof. Dr. Ralf Küsters
Supervisor: Dipl.-Math. Mike Graf
Commenced: January 11, 2022
Completed: July 11, 2022
Abstract

The Hashgraph algorithm is a distributed ledger technology (DLT) consensus algorithm that is an alternative to conventional blockchains. Generally, a distributed ledger can be seen as a database of transactions that is replicated across several locations, typically run by multiple parties. In order to reach an agreement on the validity and order of transactions, DLTs typically rely on consensus protocols as a key component.

Participants of the Hashgraph algorithm locally manage a hashgraph. This is a directed acyclic graph of events. All events include, among other (meta)data, mainly transactions that were submitted by clients. In order to reach a consensus, Hashgraph utilizes so-called virtual voting so that parties with different hashgraphs assign all events the same position in the total order of events. We call this desirable property consistency, which allows different participants to calculate and agree on the same order of transactions.

Accountability is a well-known concept in distributed systems and cryptography but new to blockchains and DLTs in general. With this concept, misbehaving parties violating predefined security goals can be identified and held accountable with undeniable cryptographic evidence to incentivize participants to behave honestly.

In this work, we put forward a rigorous proof that Hashgraph does achieve accountability w.r.t. consistency. That is, participants that misbehave by calculating a different order of transactions, by not following the Hashgraph protocol, can always be identified and rightfully blamed. To achieve this, we construct an iUC model of the hashgraph protocol with the necessary additions to hold dishonest participants accountable. In particular, we prove under relatively mild assumptions that honest participants, following the Hashgraph algorithm, will always assign events in their hashgraph the same order. That is, honest participants can reach a consensus on the total order of events and transactions. Due to the real-world applications of Hashgraph, we believe this result is of independent interest.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A hashgraph</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>A hashgraph (paths highlighted)</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>The divideRounds function of hashgraph</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>The decideFame function of hashgraph</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>The findOrder function of hashgraph</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Template for specifying protocols, see [6]. Blocks labeled with an asterisk (*) are optional. Our template differs to [6] by the addition of the block Procedures and Functions</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>The model of a Hashgraph client $\mathcal{P}<em>H</em>{\text{client}}$</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>The model of a Hashgraph node $\mathcal{P}<em>H</em>{\text{node}}$ (Part 1)</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>The model of a Hashgraph node $\mathcal{P}<em>H</em>{\text{node}}$ (Part 2)</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>The model of a Hashgraph node $\mathcal{P}<em>H</em>{\text{node}}$ (Part 3)</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>The model of a Hashgraph node $\mathcal{P}<em>H</em>{\text{node}}$ (Part 4)</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>The model of a Hashgraph node $\mathcal{P}<em>H</em>{\text{node}}$ (Part 5)</td>
<td>26</td>
</tr>
<tr>
<td>3.8</td>
<td>The ideal certificate functionality $\mathcal{F}_\text{cert}$</td>
<td>27</td>
</tr>
<tr>
<td>3.9</td>
<td>The random oracle $\mathcal{F}_\text{ro}$ (cf. [7])</td>
<td>28</td>
</tr>
<tr>
<td>3.10</td>
<td>The ideal clock functionality $\mathcal{F}_\text{clock}$</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>The ideal initialization functionality $\mathcal{F}_\text{init}$</td>
<td>29</td>
</tr>
<tr>
<td>3.12</td>
<td>The judging functionality $\mathcal{F}<em>\text{H}</em>{\text{judge}}$ for the Hashgraph model (Part 1)</td>
<td>30</td>
</tr>
<tr>
<td>3.13</td>
<td>The judging functionality $\mathcal{F}<em>\text{H}</em>{\text{judge}}$ for the Hashgraph model (Part 2)</td>
<td>31</td>
</tr>
<tr>
<td>3.14</td>
<td>The judging functionality $\mathcal{F}<em>\text{H}</em>{\text{judge}}$ for the Hashgraph model (Part 3)</td>
<td>32</td>
</tr>
<tr>
<td>3.15</td>
<td>The judging functionality $\mathcal{F}<em>\text{H}</em>{\text{judge}}$ for the Hashgraph model (Part 4)</td>
<td>33</td>
</tr>
<tr>
<td>3.16</td>
<td>The judging functionality $\mathcal{F}<em>\text{H}</em>{\text{judge}}$ for the Hashgraph model (Part 5)</td>
<td>34</td>
</tr>
</tbody>
</table>
1 Introduction

The invention of the blockchain in 2008 with Bitcoin [15], a payment system solution without central authority, instigated interest in blockchain technologies and distributed ledger technology (DLT) in general. The focus of interest and research on blockchains and distributed ledgers has developed beyond the original concepts of alternative payment systems. Specifically in the financial sector, a plethora of use cases emerged for blockchain and DLT solutions with Corda [5] and Hyperledger Fabric [1] being among the most widely adopted distributed ledgers.

DLTs. In essence, a distributed ledger can be seen as a synchronized, replicated database across multiple locations, generally run by different mutually non-trusting participants, managing a ledger of transactions. Each participant in the network usually preserves its own copy of the ledger. DLT offers new ways to validate and order transactions across distributed systems: Unlike traditional approaches which typically require central institutions and their applications to carry out transactions (e.g. money transfers), DLT is intended to reduce reliance on central parties. In order to reach an agreement on the validity and order of transactions in the distributed ledger, DLTs typically rely on so-called consensus protocols.

Blockchains. A blockchain is a distributed ledger with a growing list of records, called blocks, that are linked together to from a chain of blocks by means of cryptographic hashes. Except for the first created block, each block in the chain is linked to its preceding block by including a cryptographic hash of this block. The blocks usually contain more metadata (e.g. a timestamp of block creation) and transactions. Participants of the blockchain network usually store the chain of blocks, called blockchain, locally.

Depending on the blockchain, no special authorization or identification is required to join the blockchain network. Such blockchains (e.g. Bitcoin) are referred to as *public or permissionless*. In these networks, a consens among participants is usually reached by solving a computationally complex mathematical puzzle that requires intense computational effort. This concept is known as *proof of work*, which is widely used for many permissionless blockchains (e.g. Bitcoin and Ethereum [18]). This class of consensus algorithms often suffer from enormous energy consumption, slow transaction speeds, high transaction fees, and lack of finality. Therefore, proof of work consensus algorithms are for many application purposes not sufficient. Analogous to public blockchains, there are *private or permissioned* blockchains that allow access for authorized participants only.

Hashgraph. Since first introduced in 2016 [2], the Hashgraph protocol [2, 3, 4, 16] gained a lot of attention [9, 17]. In January 2022, the Hashgraph algorithm was made open source under the Apache License [13]. The most prominent implementation of the Hashgraph algorithm is
the Hedera Hashgraph distributed ledger that is owned and managed by the Hedera Governing Council [3]. The council’s members encompass many important companies, including Google, IBM, Boeing, Deutsche Telekom, and LG [12].

The Hashgraph consensus protocol solves many issues that exist with popular distributed ledgers; namely, high transaction throughput, high efficiency, low cost, fairness, ACID compliance, DoS resistance, and Byzantine fault tolerance [3, 4].

**Accountability.** Accountability is a well-known concept in distributed systems and cryptography; however, there are not many applications in the domain of distributed ledgers and blockchains. Graf et al. proposed in [10] a formal treatment of accountability for distributed ledgers. Moreover, the authors of this paper proved that Hyperledger Fabric, with some additional changes, does achieve accountability w.r.t. consistency.

**Related Work.** The Hashgraph consensus protocol was proved in [2] to be consistent. That is, honest participants, following the Hashgraph algorithm, do calculate the same order of transactions under the assumption of a supermajority (i.e., more than \( \frac{2}{3} \)) of honest participants. However, the proof in [2] is incomplete in several key portions; thus, a unrestrictedly convincing security analysis of Hashgraph does not exist, yet. Besides the correctness proof of Karl Crary in [8] using the Coq proof assistant, we are not aware of any other work in this field.

Our approach of proving accountability w.r.t. consistency generally follows the proof of accountability w.r.t. Fabric* in [10]. Particularly, we adopt many terms and general concepts presented in [10].

**Contribution.** In this work, we will apply the accountability framework from Küsters et al. in [14] to proof our main result that the Hashgraph protocol satisfies accountability w.r.t. consistency. To do so, we first formally define accountability w.r.t. consistency for Hashgraph. We will argue that fork-freeness is an indispensable security property to show accountability w.r.t. consistency without assuming a supermajority of honest participants.

Furthermore, we will demonstrate in a detailed proof that Hashgraph is consistent. By this, we aim to close the gaps of the proofs in [2] that are relevant to consistency.

**Structure of this work.** We first introduce the Hashgraph protocol in Chapter 2. In Chapter 3, we present our Hashgraph model after we provide a brief introduction to the iUC framework that we use for our model. The accountability framework we use is explained in Chapter 4. Finally, in Chapter 5 we present all security notions, formally define accountability w.r.t. consistency (and fork-freeness), and present all proofs, relevant to Hashgraph.
2 The Hashgraph Consensus Algorithm

We explain the Hashgraph algorithm as presented in [2], but we also adopt some notations from [8].

2.1 Overview of the Hashgraph Protocol

In Hashgraph, there are clients and nodes. Clients submit transactions, arbitrary messages, to nodes. The consensus on the order of transactions is only achieved by nodes which belong to the same Hashgraph session. All nodes in a session are predefined in the set

\[ \text{nodes} = \{ pid_1, \ldots, pid_n \} \]

with arbitrary but fixed party IDs \( pid_i \).\(^1\) We define \( n := \text{nodes} \) to be the fixed number of nodes in one session. We later prove that all honest nodes calculate a prefix of the list of ordered transactions, as long as there is a supermajority (defined to be more than \( \frac{2}{3} n \)) of nodes that behave honestly.

2.1.1 Events and Hashgraphs

Instead of ordering transactions directly, nodes group transactions into so-called events. Then, nodes can reach a consensus on the total order of events, and extract subsequently the order of transactions from ordered events. Events are created by nodes and form the vertices of the hashgraph, a directed acyclic graph, that each nodes maintains locally. Formally an event is a 7-tuple,

\[ (pid, \text{eventID}, \text{selfParent}, \text{otherParent}, \text{txs}, ts, \sigma), \]

where

- \( pid \) is the party ID of the node that created this event,
- \( \text{eventID} \) is its ID which is computed as hash over \((\text{selfParent}, \text{otherParent}, \text{txs}, ts)\),
- \( \text{selfParent} \) is the eventID of an event created by \( pid \),
- \( \text{otherParent} \) is the eventID of an event created by some node other than \( pid \),
- \( \text{txs} \) is a set of transactions,
- \( ts \in N \) is the time \( pid \) created this event,
- \( \sigma \) is the signature of the event signed by \( pid \) over \((\text{selfParent}, \text{otherParent}, \text{txs}, ts)\).
The edges of a vertex are therefore defined by the event IDs selfParent and otherParent (cf. Figure 2.1). We denote the hashgraph of a participant pid as G(pid) or simple G. Formally, a hashgraph is a set of events with the addition of the genesis event (⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥) that is contained in all hashgraphs of honest nodes.

2.1.2 Hashgraph Terminology

We first define some general concepts of the Hashgraph algorithm. Let G be the hashgraph of some node pid and E, E′ ∈ G be two events.

- Event E is an ancestor of E′ if E = E′ or there exists a down-path from E′ to E along the vertices of G.
- E is an self-ancestor of E′ if E = E′ or there exists a down-path from E′ to E such that each event included in this path was created by pid.
- The pair of events (E, E′) is a fork if E and E′ have the same creator, but neither is a self-ancestor of the other.
- E can see E′ if E′ is an ancestor of E, and the ancestors of E do not include a fork by the creator of E′.
- E can strongly see E′ if E can see E′ and there is a set S ⊆ G of events such that |S| > 2n, all events in S have pairwise different creators, and for all E_a ∈ S applies E sees E_a and E_a sees E′.

Example 1. Figure 2.2 illustrates the concept of strongly seeing; in this hashgraph with n = 5, event B5 can strongly see event B1: Let S = {A1, C3, D2, E2}, then |S| = 4 > 2n = 5, B5 can see all events in S, and all events in S can see B1.

1Throughout this work, we usually use the terms party, participant, and node synonymously if it is clear that we refer to an actual node participant in a Hashgraph session.
2.1 Overview of the Hashgraph Protocol

2.1.3 Synchronization of hashgraphs

In this section, we assume all nodes are honest and follow the described procedures. During a run of a hashgraph session, all nodes try to sync infinitely often with other random nodes. A sync is a procedure where one node, the initiator of the sync, sends a local copy of its hashgraph to another node \( \text{pid} \). In Hashgraph, this process is also called gossiping.

Upon receiving the synced hashgraph, the \( \text{pid} \) merges all events with valid signatures and hashes into its local one. Additionally, the receiving node creates a new event \( E \) with self-parent being the last event it created or the genesis event, in case it has not created any events yet. The other-parent is set to the last event, say \( E' \), the initiator node created; that is, there is no other event in the gossiped hashgraph with self-parent being the event ID of \( E' \). The receiving node includes all transactions, submitted by clients that have not yet been included in any other event, into the set \( \text{txs} \) of event \( E \).

Lastly, \( \text{pid} \) runs the following three functions in the given order: 1) \( \text{divideRounds} \), 2) \( \text{decideFame} \), and 3) \( \text{findOrder} \). The three functions above are abbreviated with \( \mathcal{H}_{\text{alg}} \). The output of \( \mathcal{H}_{\text{alg}} \) with the local hashgraph of \( \text{pid} \) as input is a total order of events. In the following, we discuss all three functions in more detail.

1) The \( \text{divideRounds} \) function (cf. Figure 2.3) assigns all events in the local hashgraph a round created (or simple round) number. Initially, the genesis event is assigned round 1. All other events have a round number \( r \) that is equal the maximum round of its two parent events. However, there is one exception: If an event \( E \) can strongly see more than \( \frac{2}{3} n \) round \( r \) witnesses, it will be assigned round number \( r + 1 \) instead. In this case, \( E \) is a witness event. A witness event is the first event created by a member in round.

We denote with \( \mathcal{W}_r^{\text{pid}} \) the witness events of round \( r \) in the local hashgraph of \( \text{pid} \). We usually omit the participant and just write \( \mathcal{W}_r \) when it is clear to which hashgraph this set belongs.

2) The \( \text{decideFame} \) function (cf. Figure 2.4) decides for all witness events \( E \in G \) whether it is famous or not. This is done by virtual voting, where a supermajority of events from different participants decide whether an event is famous or not. Each participant runs this algorithm locally, with their own local hashgraph as input, with no additional network communication.

Only witness events are eligible to participate in the election. Let \( E_r \) be a witness event in \( G \) of round \( r \). During the first voting round, all witnesses \( E_{r+1} \in \mathcal{W}_{r+1} \) vote for \( E_r \). A witness votes \( \beta = \text{true} \) for \( E_r \)'s fame if it can see it. Otherwise, it will vote \( \beta = \text{false} \) and thus against it. This concludes the first voting round.

Subsequent voting rounds \( i \geq 1 \) are different compared to the first round. Let \( E_{r+i+1} \in \mathcal{W}_{r+i+1} \) be a round \( r + i + 1 \) witness. Let \( \mathcal{W} \) be the set of all round \( r + i \) witnesses in \( G \) that \( E_{r+i+1} \) can strongly see, and let \( v \) be the majority vote of events in \( \mathcal{W} \) for \( E_{r+i+1} \). During virtual voting, there are normal voting rounds and so-called coin rounds. During a normal voting round, \( E_{r+i+1} \) always vote for \( E_r \) w.r.t. the majority decision \( v \). If there is a supermajority of events in \( \mathcal{W} \) with the same vote for \( E_r \), then \( E_{r+i+1} \) will decide the fame for \( E_r \) according to the supermajority decision \( v \). We denote this voting result with \( \text{decision}_{G}(E_r, E_{r+i+1}, v) \). At this point, the virtual voting for \( E_r \) is ended. Now, suppose we are in a coin round. In coin rounds

\[ \text{The handling of exceptions due to dishonest nodes is discussed in the subsequent chapters.} \]
events cannot decide the famousness for events. However, events still vote for \( E_r \). If there is a supermajority of round \( r + i \) events in \( \mathbb{V} \) that agree on vote \( v \), \( E_{r+i+1} \) will still vote with \( v \) for \( E_r \). Otherwise, \( E_{r+i+1} \) “flips a coin” for the vote \( E_r \) by voting according to its last bit of the signature.

For later proofs we will need the predicate

\[
\text{vote}_G(E_r, E_{r+i+1}, \beta)
\]

that is true if event \( E_{r+i+1} \), in voting round \( i \), votes for event \( E_r \)’s fame with \( \beta \).

3) The decideFame function (cf. Figure 2.5) assigns events a position in the total order of events. First, we need to define the following definition: A Unique famous witness is a famous witness that does not have the same creator as any other famous witness created in the same round. Notice that in the absence of forking, each famous witness is also a unique famous witness.

Let \( E_0 \in G \) be an arbitrary event. At the beginning, decideFame determines if there can even exist an ordering of \( E_0 \) at this point, by checking if the round received number of \( E_0 \) exists. The round received number \( r \) of an event \( E_0 \) is defined to be the first round where all unique famous witnesses of round \( r \) are descendants of \( E_0 \). If \( E_0 \) has a round received number, decideFame calculates the set \( S \subseteq G \) for \( E_0 \), which is defined as

\[
S = \{ E_1 \in G \mid \exists E_2 \in G: E_2 \in \mathbb{U}_r \\
\land E_1 \text{ is a self-ancestor of } E_2 \\
\land E_0 \text{ is an ancestor of } E_1. \\
\land E_0 \text{ is not an ancestor of } E_1 \text{'s self-parent} \}.
\]

Now, the consensus timestamp of event \( E_0 \) is determined by taking the median of the timestamps of the events in \( S \).

Lastly, the complete order of events is calculated: Events are sorted in ascending order by 1. the round received number, 2. remaining ties by consensus timestamps, and 3. any remaining ties lexicographically by their signature.

We later prove that hashgraph is consistent, i.e., honest participants calculate the same order of events if there are less than \( \frac{1}{3} n \) dishonest participants.

```
1: function divideRounds(): {Assign rounds to events and mark witnesses.}
2:   for all event x ∈ G(pid), in topological order do: {Iterate over events in the hashgraph of pid.}
3:     r ← 1.
4:     if event x has parents:
5:       r ← max round of parents of x.
6:     x.round ← r.
7:     if x strongly sees more than \( \frac{2}{3} n \) round \( r \) witnesses:
8:       x.round ← x.round + 1.
9:       x.witness ← (x has no self parent) \lor (x.round > x.selfParent.round).
```

Figure 2.3: The divideRounds function of hashgraph.
2.1 Overview of the Hashgraph Protocol

1: function decideFame() : { Decide which events are famous. }
2: for all event $x \in G(pid)$, in order from earlier rounds to later do: { For all events, in order from earlier rounds to later do: }
3: \hspace{1em} $x$.famous $\leftarrow \text{?}$; { Fame of $x$ is undecided at first. }
4: if $\exists r \in N \text{ s.t. } x \in W^r_{pid}$; { If $x$ is a witness in round $r$. }
5: for all $y \in W^r_{pid}$, do: { First round of the election; $i = 0$. }
6: \hspace{1em} $y$.vote $\leftarrow y$ can see $x$? { Fame of $x$ is undecided at first. }
7: for $i = 1, \ldots, |W^r_{pid}| \neq 0$ do: { Further rounds of the election; continues until there are no more witnesses. }
8: \hspace{1em} for all $y \in W^r_{pid+1}$ do: { This is a normal round. }
9: \hspace{2em} $W \leftarrow \{ w \in W^r_{pid+1} \mid y \text{ strongly sees } w \}$. { If supermajority is reached, decide fame of $x$. }
10: \hspace{2em} $v \leftarrow$ majority vote in $W(\text{true for tie})$. { If supermajority is reached, decide fame of $x$. }
11: \hspace{2em} $i \leftarrow$ number of events in $W$ with a vote of $v$. { Event $y$ votes pseudorandomly. }
12: \hspace{2em} if $i \mod c > 0$: { This is a normal round. }
13: \hspace{3em} $y$.vote $\leftarrow v$. { Begin coin round. }
14: \hspace{3em} if $i > \frac{2}{3} n$: { Event $y$ votes pseudorandomly. }
15: \hspace{3em} $x$.famous $\leftarrow v$. { Event $y$ votes pseudorandomly. }
16: \hspace{3em} break out of i-loop. { Event $y$ votes pseudorandomly. }
17: else: { Begin coin round. }
18: \hspace{2em} if $i > \frac{2}{3} n$: { Event $y$ votes pseudorandomly. }
19: \hspace{2em} $y$.vote $\leftarrow v$. { Event $y$ votes pseudorandomly. }
20: else: { Event $y$ votes pseudorandomly. }
21: \hspace{2em} $y$.vote $\leftarrow$ last bit of $y$.signature. { Event $y$ votes pseudorandomly. }

Figure 2.4: The decideFame function of hashgraph.

1: function findOrder() : { Establish total order of events and transactions. }
2: for all event $x \in G(pid)$ do: { Check and calculate the roundReceived number of $x$, if it exists. }
3: \hspace{1em} if there exists a round $r \in N$ s.t. { Check and calculate the roundReceived number of $x$, if it exists. }
4: \hspace{2em} 1. there is no event $y$ with $y$.round $\leq r$ that has { Check and calculate the roundReceived number of $x$, if it exists. }
5: \hspace{3em} (i) $y$.witness = true and { Check and calculate the roundReceived number of $x$, if it exists. }
6: \hspace{3em} (ii) $y$.famous = ?; and { Check and calculate the roundReceived number of $x$, if it exists. }
7: \hspace{2em} 2. $x$ is an ancestor of every unique famous witness in round $r$; and { Check and calculate the roundReceived number of $x$, if it exists. }
8: \hspace{2em} 3. $r$ is minimal, i.e., conditions 1. and 2. are false for all $r' < r$ : { Check and calculate the roundReceived number of $x$, if it exists. }
9: \hspace{3em} $x$.roundReceived $\leftarrow r$. { Check and calculate the roundReceived number of $x$, if it exists. }
10: \hspace{2em} $S \leftarrow$ set off all events $z \in G(pid)$, where { Check and calculate the roundReceived number of $x$, if it exists. }
11: \hspace{3em} 1. $z$ is a self-ancestor of a round $r$ unique famous witness, and { Check and calculate the roundReceived number of $x$, if it exists. }
12: \hspace{3em} 2. $x$ is an ancestor of $z$, and { Check and calculate the roundReceived number of $x$, if it exists. }
13: \hspace{3em} 3. $x$ is not an ancestor of $z$’s selfparent. { Check and calculate the roundReceived number of $x$, if it exists. }
14: \hspace{2em} $x$.consensusTimestamp $\leftarrow$ median of the timestamps of all events in $S$. { Check and calculate the roundReceived number of $x$, if it exists. }
15: \hspace{2em} $E \leftarrow$ list of all events in $G(pid)$ with a round received number. { Check and calculate the roundReceived number of $x$, if it exists. }
16: \hspace{2em} $E \leftarrow$ sort events in $G(pid)$ in ascending order by { Check and calculate the roundReceived number of $x$, if it exists. }
17: \hspace{3em} 1. roundReceived number, { Check and calculate the roundReceived number of $x$, if it exists. }
18: \hspace{3em} 2. consensusTimestamp, and { Check and calculate the roundReceived number of $x$, if it exists. }
19: \hspace{3em} 3. lexicographically by their signature in case of ties. { Check and calculate the roundReceived number of $x$, if it exists. }
20: return $E$. { Check and calculate the roundReceived number of $x$, if it exists. }

Figure 2.5: The findOrder function of hashgraph.
3 Security Model of Hashgraph

3.1 Introduction to the iUC Framework

The iUC framework [6] is a highly expressive universal composability model with many sensitive defaults for easier use. In particular, the framework offers a convenient template for specifying various protocols and systems. We begin reiterating the general structure of protocols as described in [6].

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$. Each machine $M_i$ implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment) via network interfaces. An instance of a machine manages one or more so-called entities. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) $pid$ running in a session with session ID (SID) $sid$ and executing some code defined by the role $role$ where this role has to be (one of) the role(s) of $M_i$ according to the specification of $M_i$. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances.

3.1.1 Structure of Protocols

Roles. Roles are piece of codes that perform a specific task in a protocol $\mathcal{P}$. Each role in $\mathcal{P}$ is implemented by a single machine $M_i$; however, one machine can implement multiple roles. Roles of a protocol can be either public or private. I/O interfaces of public roles are accessible by other entities belonging to roles in the same protocol $\mathcal{P}$ or the environment/unknown higher level protocols. Whereas I/O interfaces of private roles are only accessible by other entities belonging to the same protocol. For a protocol $\mathcal{P}$ with $p$ public and $q$ private roles, we write $(pubrole_1, \ldots, pubrole_p \mid privrole_1, \ldots, privrole_q)$. If two protocols are combined to a new protocol, all private roles will remain private, whereas previously public roles can be either public or private.

Subroutines. A machine $M_i$ implementing a specific role can implement other roles as subroutines. Then, the I/O interfaces of $M_i$ will be connected to the I/O interfaces of the machine implementing the (as subroutine) specified role.
3 Security Model of Hashgraph

Setup for the protocol \( \mathcal{Q} = \{ M_1, \ldots, M_n \} \):

<table>
<thead>
<tr>
<th>Participating roles:</th>
<th>list of all ( n ) sets of roles participating in this protocol; each set corresponds to one machine ( M_i ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corruption model:</td>
<td>corruption model of ( \mathcal{Q} ) (e.g., incorruptible or dynamic corruption).</td>
</tr>
<tr>
<td>Protocol parameters*</td>
<td>e.g., externally provided algorithms or variables parametrizing a machine.</td>
</tr>
</tbody>
</table>

Implementation of \( M_i \) for each set of roles:

<table>
<thead>
<tr>
<th>Implemented role(s):</th>
<th>the set of roles implemented by this machine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutines*</td>
<td>a list of all (other) roles that this machine uses as subroutines.</td>
</tr>
<tr>
<td>Internal state*</td>
<td>internal state variables of ( M_i ) that are used to store data across different machine invocations.</td>
</tr>
<tr>
<td>CheckID*</td>
<td>algorithm for deciding whether this machine is responsible for an entity ((psd, sid, role)).</td>
</tr>
<tr>
<td>Corruption behavior*</td>
<td>description of the DetermineCorrStatus and AllowAdvMessage algorithms.</td>
</tr>
<tr>
<td>Initialization*</td>
<td>this block is executed the first time an instance of the machine ( M_i ) accepts a message; useful to, e.g., assign initial values that are globally used for all entities managed by this instance.</td>
</tr>
<tr>
<td>EntityInitialization*</td>
<td>this block is executed only the first time that some message for a (new) entity is received; useful to, e.g., assign initial values that are specific for single entities.</td>
</tr>
<tr>
<td>MessagePreprocessing*</td>
<td>this algorithm is executed every time a new message for an uncorrupted entity is received.</td>
</tr>
<tr>
<td>Main:</td>
<td>specification of the actual behavior of an uncorrupted entity.</td>
</tr>
<tr>
<td>Procedures and Functions*</td>
<td>this block can be used to specify functions or procedures that can be used by other algorithms of this machine. Useful to, e.g., split up longer algorithms or reduce code repetitions.</td>
</tr>
</tbody>
</table>

**Figure 3.1:** Template for specifying protocols, see [6]. Blocks labeled with an asterisk (*) are optional. Our template differs to [6] by the addition of the block Procedures and Functions.

Exchanging Messages. Entities can send and receive messages using the I/O and network interfaces belonging to their respective role. The receiver of the message must always be specified, which is either the adversary in case of network interfaces of some other entity (with a role that has a connected I/O interface) in case of the I/O interface.

The iUC model specifies a convenient template for specifying protocols. This is illustrated in Figure 3.1.

### 3.1.2 Modeling Corruption

Depending on the used corruption model of a protocol, an entity might become explicitly corrupted by the adversary in a run of the system. In this case, the adversary gains full control of the corrupted entity: arriving messages at an entity are forwarded to the adversary, while the adversary can also tell an entity to send messages to other entities on behalf of the corrupted entity. The AllowAdvMessage algorithm can be used to restrict the adversary from sending messages on behalf of a corrupted entity to arbitrary entities.

However, the adversary cannot corrupt incorruptible protocols. An entity might consider itself implicitly corrupted, but in this case, the adversary does not gain control over the implicitly corrupted entity. This can be specified in the DetermineCorrStatus algorithm.

### 3.2 An iUC Model for Hashgraph

In this chapter we give an introduction to our iUC model, which is specified at the end of this chapter. We begin defining the Hashgraph protocol:
3.2 An iUC Model for Hashgraph

Definition 1 (The Hashgraph Protocol $\mathcal{P}_H$).

For protocols $\mathcal{P}^H_{\text{client}}$, $\mathcal{P}^H_{\text{node}}$, $\mathcal{F}_{\text{cert}}$, $\mathcal{F}_{\text{ro}}$, $\mathcal{F}_{\text{clock}}$, $\mathcal{F}_{\text{init}}$, $\mathcal{F}^H_{\text{judge}}$ as in Figures 3.2 to 3.16 we define the Hashgraph protocol to be

$$\mathcal{P}_H = (\text{client}, \text{init} \mid \text{node}, \text{cert}, \text{ro}, \text{clock}, \text{judge}).$$

The protocols $\mathcal{P}^H_{\text{client}}$ and $\mathcal{P}^H_{\text{node}}$ correspond to the clients and nodes as in Chapter 2. With $\mathcal{F}_{\text{init}}$ we introduce an ideal initialization functionality that starts a hashgraph session. We stress that our model is able to run multiple hashgraph session in parallel. However, we consider a participant $\text{pid} \in \text{nodes}$ to be dishonest if one of its node instances is corrupted. That is, the attacker explicitly corrupted a node instance of $\text{pid}$ or a node instance considers itself implicitly corrupted due to the corruption of the signing key of its signer entity.

3.2.1 The Ideal Signature Functionality $\mathcal{F}_{\text{cert}}$

The ideal protocol $\mathcal{F}_{\text{cert}}$ is an incorruptible signature and verifier protocol that is used for two purposes: (i) Nodes can use the signer role of this protocol to sign arbitrary messages, and (ii) other participants can use the verifier role to check the validity of signatures. By default, $\mathcal{F}_{\text{cert}}$ prevents forgeries of messages that have not been previously signed by the participant of the signing key. However, we still allow the network attacker to corrupt signing keys and give the attacker the ability to forge signatures. This will be explained in more detail later.

Entities in $\mathcal{F}_{\text{cert}}$

Observe in the definition of $\mathcal{F}_{\text{cert}}$ that one machine instance of $\mathcal{F}_{\text{cert}}$ manages all parties and roles in a single session $\text{sid}'$. This session is a triple $(\text{pid}, \text{sid}, \text{role})$ and thus represents that one machine instance of $\mathcal{F}_{\text{cert}}$ manages the ideal signing and verifying functionalities of participant $\text{pid}$ in one session $\text{sid}$ and one role $\text{role}$. For a machine instance managing all entities with SID $(\text{pid}, \text{sid}, \text{role})$, we call $\text{pid}$ also the pidowner; this becomes important when signing messages.

Verifying signatures

Other entities of machines implementing $\mathcal{F}_{\text{cert}}$ as subroutine ($M_{\text{node}}$, $M_{\text{client}}$, and $M_{\text{judge}}$ in our case) can verify if a signature $\sigma$, supposedly signed by $\text{pid}$ (in session $\text{sid}$ with role $\text{role}$), of $\text{msg}$ is indeed valid. This is done by sending the message $(\text{Verify}, \text{msg}, \sigma)$ to $e = (\text{pid}_{\text{cur}}, (\text{pid}, \text{sid}, \text{role}), \mathcal{F}_{\text{cert}} : \text{verify})$, where $\text{pid}_{\text{cur}}$ is the party ID of the calling entity and the receiver is the machine instance of $\mathcal{F}_{\text{cert}}$ that manages all entities with SID $(\text{pid}, \text{sid}, \text{role})$. We denote the reply of entity $e$ as

$$\text{verifySig}_{\text{pk}(\text{pid})}(\text{msg}, \sigma).$$

One can observe such verification requests in the protocols $\mathcal{P}^H_{\text{client}}$, $\mathcal{P}^H_{\text{node}}$, and $\mathcal{F}^H_{\text{judge}}$.

Signing messages

Similar to verifying signatures, participants (i.e., an entity $(\text{pid}, \text{sid}, \text{role})$) can sign arbitrary messages $\text{msg}$ at $\mathcal{F}_{\text{cert}}$ by sending the message $(\text{Sign}, \text{msg})$ to $e = (\text{pid}_{\text{cur}}, (\text{pid}, \text{sid}, \text{role}), \mathcal{F}_{\text{cert}} : \text{signer})$ on the I/O interface. Analogously, we denote the reply of $e$ as

$$\text{sign}_{\text{pk}(\text{pid})}(\text{msg}).$$
Notice in our iUC model for Hashgraph that only nodes implement the signer role of $F_{\text{cert}}$ as subroutine. Therefore, only nodes are able to send messages to entities of role signer. Moreover, a receiving entity $(\text{pid}, \text{sid}, \text{signer})$ of $F_{\text{cert}}$ must be the pidowner itself; this is used to prevent nodes of other parties to sign messages for $\text{pid}$. But a node with party ID $\text{pid}'$ could still send a message $(\text{Sign}, \text{msg})$ to entity $(\text{pid}, (\text{pid}, \text{sid}, \text{role}), F_{\text{cert}} : \text{signer})$, where $\text{pid} \neq \text{pid}'$ and thus sign in the name of $\text{pid}$. One can observe in Figure 3.16 that honest nodes never sign messages of other nodes. However, to ensure the same for corrupted nodes, we restrict in the implementation of AllowAdvMessage in $\mathcal{P}_{\text{node}}$ the adversary to send messages via an explicitly corrupted entity $(\text{pid}, \text{sid}, \mathcal{P}_{\text{node}} : \text{node})$ to $(\text{pid}_{\text{receiver}}, \text{sid}_{\text{receiver}}, \text{role}_{\text{receiver}})$ if $\text{pid} \neq \text{pid}_{\text{receiver}}$. We will discuss more about this later in Chapter 5.
3.2 An iUC Model for Hashgraph

<table>
<thead>
<tr>
<th>Description of $\mathcal{P}_{\text{client}}^H =$ (client):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating roles: {client}</td>
</tr>
<tr>
<td>Corrupt algorithm: Dynamic corruption without secure erasures</td>
</tr>
<tr>
<td>Protocol parameters:</td>
</tr>
<tr>
<td>$-$ $\eta \in \mathbb{N}$.</td>
</tr>
<tr>
<td>$-$ nodes $\subseteq {0, 1}^*$.</td>
</tr>
<tr>
<td>Internal state:</td>
</tr>
<tr>
<td>$-$ states $\rightarrow (\mathbb{N} \times {0, 1}^n \times {0, 1}^n \times {0, 1}^n) \cup {\emptyset}$.</td>
</tr>
<tr>
<td>CheckID(pid, sid, role):</td>
</tr>
<tr>
<td>Accept all messages with the same SID and PID.</td>
</tr>
<tr>
<td>Corrupt behavior:</td>
</tr>
<tr>
<td>DetermineCorrStatus(pid, sid, role):</td>
</tr>
<tr>
<td>return explicitCorr[entityaw]’.</td>
</tr>
<tr>
<td>Main:</td>
</tr>
<tr>
<td>recv (Submit, msg) from I/O:</td>
</tr>
<tr>
<td>send (Submit, msg) to NET:</td>
</tr>
<tr>
<td>recv Read from I/O:</td>
</tr>
<tr>
<td>msglist $\rightarrow$ 0; i $\leftarrow$ 1.</td>
</tr>
<tr>
<td>while $\exists e \in {i} \times {0, 1}^n \times {0, 1}^n \times {0, 1}^n$ s.t. hasSupernovation(e) = true do</td>
</tr>
<tr>
<td>msglist $\leftarrow$ msglist $\cup$ {e}.</td>
</tr>
<tr>
<td>i $\leftarrow$ i $+$ 1.</td>
</tr>
<tr>
<td>output $\leftarrow$ {(counter, msg)</td>
</tr>
<tr>
<td>reply (Read, output).</td>
</tr>
<tr>
<td>recv (StateUpdate, (pid_send, msglist), \sigma) from NET s.t. pid_send $\in$ nodes:</td>
</tr>
<tr>
<td>sigIsValid $\leftarrow$ verifySig(pid_send, (pid_send, msglist), \sigma).</td>
</tr>
<tr>
<td>if sigIsValid = true:</td>
</tr>
<tr>
<td>isConsecutive $\leftarrow$ true.</td>
</tr>
<tr>
<td>for i = 1, …,</td>
</tr>
<tr>
<td>if (i, …, i) $\notin$ msglist:</td>
</tr>
<tr>
<td>isConsecutive $\leftarrow$ false.</td>
</tr>
<tr>
<td>if isConsecutive = true:</td>
</tr>
<tr>
<td>if states[pid] $\subseteq$ msglist</td>
</tr>
<tr>
<td>states[pid] $\leftarrow$ msglist.</td>
</tr>
<tr>
<td>send (Evidence, (pid_send, msglist), \sigma) to (pid_send, sid_send, $\mathcal{F}_{\text{judge}}^H$ : judge);</td>
</tr>
<tr>
<td>Honest clients always report all state updates signed by pid_send to $\mathcal{F}_{\text{judge}}^H$ as evidence.</td>
</tr>
<tr>
<td>Procedures and Functions:</td>
</tr>
<tr>
<td>function hasSupernovation(entry):</td>
</tr>
<tr>
<td>voteCount $\leftarrow$ {</td>
</tr>
<tr>
<td>if voteCount $&gt;$ $\frac{3}{2}$</td>
</tr>
<tr>
<td>return true.</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>return false.</td>
</tr>
<tr>
<td>function verifySig(pid, msg, \sigma):</td>
</tr>
<tr>
<td>send (Verify, msg, \sigma) to (pid_send, (pid_send, $\mathcal{F}^H_{node}$ : node), $\mathcal{F}_{\text{cert}}$ : verifier);</td>
</tr>
<tr>
<td>wait for (VerResult, result).</td>
</tr>
<tr>
<td>return result.</td>
</tr>
</tbody>
</table>

Figure 3.2: The model of a Hashgraph client $\mathcal{P}_{\text{client}}^H$ (Part 1).
3 Security Model of Hashgraph

Participating roles: {node}

Corruption model: Dynamic corruption without secure erasures

Protocol parameters:
- $\eta \in \mathbb{N}$.
- $\text{nodes} \subseteq \{0, 1\}^*$. \{The identities of the Hashgraph nodes.
- $\text{coinRound} \in \mathbb{N}$. \{After coinRound rounds in Hashgraph, there is a “coin round” during virtual voting.

Description of $\mathcal{P}_\text{node} = \{\text{node}\}$:

Implemented role(s): {node}

Subroutines: $\mathcal{F}_{\text{cert}}: \text{signer}, \mathcal{F}_{\text{ver}}: \text{verifier}, \mathcal{F}_{\text{ra}}: \text{randomOracle}, \mathcal{F}_{\text{judge}}: \text{judge}, \mathcal{F}_{\text{clock}}: \text{clock}$

Internal state:
- $\text{events} \subseteq (\text{nodes} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*) \cup (\bot^7) = \emptyset$. \{This set represents the local hashgraph $G$ of pid$_\text{cur}$ and is a subset of events with entries (\text{pid}, \text{eventID}, \text{selfParent}, \text{otherParent}, \text{transactions}, \text{ts}, \sigma$) or, in the case of the genesis event, $(\bot, \bot, \bot, \bot, \bot, \bot, \bot)$. \}
- $\text{msglist} \subseteq \mathbb{N} \cup \{\epsilon\} \times \{0, 1\}^n \times \{0, 1\}^n \cup \{\epsilon\} \times \{0, 1\}^* = \emptyset$. \{A list of transactions: entries are of form (\text{counter}, \text{eventId}, \text{eventID}, \text{ts}). \}
- $\text{roundReceivedEvents} : \mathcal{N} \rightarrow (\text{nodes} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*) \cup (\bot^7)$. Store all events with a round received number in consecutive order, starting at index 0. Also used for sorting events; see function $\text{findOrder}$. \{ \}
- $\text{lastEvent} \in \{0, 1\}^n \cup \{\bot\} = \bot$. \{The last event created by pid$_\text{cur}$. \}
- $\text{roundCreated} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathcal{N}$. \{The round created of an event (identified via eventID); initially 0 for all entries. \}
- $\text{roundReceived} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathcal{N}$. \{The round received of an event (identified via eventID); initially 0 for all entries. \}
- $\text{tmpVotes} : \{0, 1\}^n \rightarrow \{?, \text{true}, \text{false}\}$. \{Used to store votes during virtual voting; initially ? for all entries. \}
- $\text{eventConsensusNumber} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathcal{N} \cup \{\epsilon\}$. \{Stores the total order of events from events; initially ? for all entries. \}
- $\text{round} \in \mathbb{N}$. \{The current “time”. \}
- $\text{isWitness} : \{0, 1\}^n \cup \{\bot\} \rightarrow \{?, \text{true}, \text{false}\}$. \{Stores whether an event is a witness; initially ? for all entries. \}
- $\text{isFamous} : \{0, 1\}^n \cup \{\bot\} \rightarrow \{?, \text{true}, \text{false}\}$. \{Stores whether an event is famous; initially ? for all entries. \}

CheckID$(\text{pid}, \text{std}, \text{role})$:
Accept all entities with the same PID and SID. \{Accept a single party in one session. \}

Corruption behavior:
- \text{DetermineCorrStatus}$(\text{pid}, \text{std}, \text{role})$:
  - if $\text{entity}_\text{cur} \in \text{explicitCorr}^a$:
    - return true. \{Check whether node itself is corrupted. \}
  - return corr$(\text{pid}, \text{std}, \text{role}, \mathcal{P}^\text{H}_\text{node}: \text{node})$, $\mathcal{F}_{\text{cert}}: \text{signer}^b$ \{Request corruption status at $\mathcal{F}_{\text{cert}}$ and return its value. \}
- \text{AllowAdvMessage}$(\text{pid}, \text{std}, \text{role}, \text{pid}_\text{receiver}, \text{std}_\text{receiver}, \text{role}_\text{receiver}, \text{m})$:
  - if $\text{pid} = \text{pid}_\text{receiver}$:
    - return true.
  - else:
    - return false.

EntityInitialization$^d$
- events.add($\bot, \bot, \bot, \bot, \bot, \bot, \bot$). \{Add the genesis event to the local hashgraph. \}

Description of $\mathcal{M}_\text{node}$ continues in Figure 3.4.

$^a$explicitCorr is an internal framework specific variable in iUC that stores all explicitly corrupted entities (by adversary $\mathcal{A}$) that are managed by the current machine instance, see [6].

$^b$corr is a macro in iUC to query if an entity is corrupted.

$^c$We could have also omitted this algorithm entirely, as our implementation is the default behavior, specified in [6], if AllowAdvMessage is not stated explicitly.

$^d$We could have also added this complete section to the “Initialization” block of iUC, since a machine manages exactly one entity.

Figure 3.3: The model of a Hashgraph node $\mathcal{P}^\text{H}_\text{node}$ (Part 1).
3.2 An iUC Model for Hashgraph

Description of \( M_{\text{node}} \) (cont.):

Message Preprocessing:

\[
\text{recv msg from I/O or NET: if isinitialized = true: send GetCurRound to (pid\textsubscript{cur}, sid\textsubscript{cur}, F\textsubscript{clock} : clock); wait for (GetCurRound, round). round \leftarrow round. else: abort. (Abort current machine activation if the client instance is not initialized. Upon executing the abort command, the environment gets activated by definition of the ITM model.)}
\]

Main:

\[
\text{recv (Submit, msg) from NET: \{Process a transaction submission. Any message is a valid transaction.\} txlId \leftarrow \text{hash}(msg). msglist.add(\epsilon, txId, \epsilon, msg).}
\]
\[
\text{recv (RecvGossipEvents, gossipedEvent, events) from NET s.t. \{Node receives a gossip event.\} events \subseteq (\text{nodes} \times \{0, 1\}^g \times \{0, 1\}^g \times \{0, 1\}^f \times \{0, 1\}^g \times N_x) \cup (\{\perp\}^g). valid \leftarrow \text{verifyBasicGraphCorrectness}(events). \{Ensure Basic Graph Correctness in Definition 6.\}}
\]
\[
\text{if valid \leftarrow false: break. \{If basic graph correctness is violated, break here.\}}
\]
\[
\text{if \exists (pid, \text{eventID}, \text{txs}, \ldots) \in \mathsf{s.t.} (\text{pid} \neq \text{pid}\textsubscript{cur} \lor \{\text{events}\} = 1) \land \text{eventID} = \text{gossipedEvent}: \{Ensure complete other-ancestor construction for eventID\textsubscript{new} (see below) (cf. Definition 5.1 (ii)).\}}
\]
\[
\text{if \exists (\text{pid}, \text{eventID}', \text{selfPar}, \ldots) \in \mathsf{s.t.} \text{eventID}' \neq \text{eventID} \land \text{gossipedEvent} = \text{selfPar}: \{Only \text{“}sync\text{“} with most recent event from pid, where pid is the creator of gossippedEvent.\}}
\]
\[
\text{for all (\text{txs}, \text{eventID}', \ldots) \in \mathsf{events} \setminus \{\text{events} \cap \text{events}\} \text{do:}\}
\]
\[
\text{for all msg \in \mathsf{txs} do: \{Add new messages to msglist from previous sync.\}}
\]
\[
\text{msglist.add(\epsilon, \text{hash}(msg), \text{eventID}', msg). events.add(events). \{Merge new events into pidcur's hashgraph.\}}
\]
\[
\text{txs\textsubscript{new}} \leftarrow \{\text{msg} | (\epsilon, txId, \epsilon, msg) \in \text{msglist}\}. \{Add transactions for new event.\}
\]
\[
\text{eventID\textsubscript{new}} \leftarrow \text{hash}((\text{lastEvent}, \text{gossipedEvent}, \text{txs\textsubscript{new}}, \text{round})). \{Create eventID of the new event.\}
\]
\[
\sigma\textsubscript{new} \leftarrow \text{sign}((\text{lastEvent}, \text{gossipedEvent}, \text{txs\textsubscript{new}}, \text{round})). \{Sign new event.\}
\]
\[
\text{events.add((pid\textsubscript{cur}, \text{eventID\textsubscript{new}}, \text{lastEvent}, \text{gossipedEvent}, \text{txs\textsubscript{new}}, \text{round}, \sigma\textsubscript{new})). \{Merge the new event (eventID\textsubscript{new}) into local hashgraph.\}}
\]
\[
\text{for all (\epsilon, txId, \epsilon, msg) \in \text{msglist} do: \{Assign the newly created event ID to transactions in msglist.\}}
\]
\[
\text{msglist.remove((\epsilon, txId, \epsilon, msg)). msglist.add(\epsilon, txId, \text{eventID\textsubscript{new}}, msg). lastEvent \leftarrow \text{eventID\textsubscript{new}}.}
\]
\[
\text{divideRound()}. \{Determines rounds of events and whether events are witnesses.\}
\]
\[
\text{decideFame()}. \{Determines whether events are “famous”.\}
\]
\[
\text{findOrder()}. \{Establishes total order over events and transactions.\}
\]
\[
\text{send (EvidenceNode, events) to (pid\textsubscript{cur}, sid\textsubscript{cur}, F^H : judge): \{Report current state to }^H\text{judge}.\}
\]
\[
\text{recv StateUpdate from NET: \{Process a read request from NET.\}}
\]
\[
\text{currentState} \leftarrow \emptyset. \{Extract output and drop eventID.\}
\]
\[
\text{for all (counter, txId, eventID, msg) \in \text{msglist s.t. counter} \neq \epsilon do: \text{currentState.add((counter, txId, eventID, msg)). \{Sign output.\}}}
\]
\[
\sigma \leftarrow \text{sign}((\text{pid\textsubscript{cur}}, \text{currentState})). \{\text{Sign output.\}}
\]
\[
\text{reply (StateUpdate, (pid\textsubscript{cur}, currentState), \sigma).}
\]
\[
\text{recv GossipEvents from NET: \{A triggers when nodes gossip events.\}}
\]
\[
\text{psd,} \leftarrow \text{nodes} \setminus \{\text{pid\textsubscript{cur}}\}. \{Randomly chose a node to sync current events to.\}
\]
\[
\text{reply (GossipEvents, lastEvent, events).}
\]
\[
\text{recv GetCurRound from I/O or NET: \{Current round can be requested externally.\}}
\]
\[
\text{reply (GetCurRound, round).}
\]

Description of \( M_{\text{node}} \) continues in Figure 3.5.

Figure 3.4: The model of a Hashgraph node \( H^H_{\text{node}} \) (Part 2).
3 Security Model of Hashgraph

Procedures and Functions:

**function divideRounds()**:

```plaintext
function divideRounds():
    for all \( E \in \text{events}, \text{in topological order}\) do:
        \((\_, \text{eventID}, \text{selfPar}, \text{otherPar}, 
            \ldots) \leftarrow \text{parse } E.
        \)
        \(r \leftarrow 1.
        \)
        if \( \text{eventID} \neq \_ \):
            \(r \leftarrow \max(\text{roundCreated}[\text{selfPar}], \text{roundCreated}[\text{otherPar}])).
            \)
            \(S \leftarrow \{E' \in \text{events} \mid \text{roundCreated}[E'] = r \}.
            \)
            \(S \leftarrow \{E' \in S \mid \text{StronglySees}(E, E') = \text{true}\}.
            \)
            \(\text{if } |S| > \frac{2}{3} \text{nodes;}
            \)
            \(\text{roundCreated}[\text{eventID}] \leftarrow r + 1.
            \)
        else:
            \(\text{roundCreated}[\text{eventID}] \leftarrow r.
            \)
        if (\(\text{selfPar} = \_\) \lor \(\text{roundCreated}[\text{eventID}] > \text{roundCreated}[\text{selfPar}]\)):
            \(\text{if } E \text{ is the genesis event, or it has the genesis event as self parent, or}
            \)
            \(\text{it is the first event of pid within a new round, } E \text{ is a witness.}
            \)
    \end{function}
```

**function decideFame()**:

```plaintext
function decideFame():
    \(\text{for all } E_r \in \text{events, in order from earlier rounds to later do:}
    \)
    \((\_, \text{eventID}r, \text{roundCreated}, \ldots) \leftarrow \text{parse } E_r.
    \)
    
    \(E_r \text{ is the candidate, i.e., the witness whom is voted for next.}
    \)
    \(\text{Initially, all votes for } E_r \text{ are undecided.}
    \)
    \(\text{isWitness}[\text{eventID}] \leftarrow \text{true.}
    \)
    \(\text{for all } E_{r+1} \in \text{WitnessesOfRound}(r + 1) \text{ do:}
    \)
    \((\_, \text{eventID}_{r+1}, \text{roundCreated}, \ldots) \leftarrow \text{parse } E_{r+1}.
    \)
    \(E_{r+1} \text{ is the candidate, i.e., the witness whom is voted for next.}
    \)
    \(\text{Initially, all votes for } E_r \text{ are undecided.}
    \)
    \(\text{if } \text{isWitness}[\text{eventID}] = \text{true:}
    \)
    \(\text{roundCreated}[\text{eventID}] \leftarrow r.
    \)
    \(\text{tmpVotes}[\text{eventID}_{r+1}] \leftarrow \text{true.}
    \)
    \(\text{for all } i = 1, \ldots \text{ s.t. } (W_{r+1+i} \leftarrow \text{WitnessesOfRound}(r + i + 1)) \neq \emptyset \text{ do:}
    \)
    \(W_r \leftarrow \{E_{r+1} \in \text{WitnessesOfRound}(r + i) \mid \text{StronglySees}(E_{r+1}, E_r) = \text{true}\}.
    \)
    \(W_r \text{ is a subset of votes whether } E_{r+1} \text{ is famous.}
    \)
    \(\text{All witnesses in the next round (w.r.t. } E_r \text{’s round } r \text{) vote for } E_r.
    \)
    \(\text{else:}
    \)
    \(\text{tmpVotes}[\text{eventID}_{r+1}] \leftarrow \text{false.}
    \)
    \(\text{for all } i = 1, \ldots \text{ s.t. } (W_{r+1+i} \leftarrow \text{WitnessesOfRound}(r + i + 1)) \neq \emptyset \text{ do:}
    \)
    \(W_{r+1} \leftarrow \{E_{r+1} \in W_{r+1+i} \mid \text{StronglySees}(E_{r+1}, E_r) = \text{true}\}.
    \)
    \(W_{r+1} \text{ is a subset of votes whether } E_{r+1} \text{ is famous.}
    \)
    \(\text{All witnesses in the previous round (w.r.t. } E_{r+1+i} \text{’s round}
    \)
    \(\text{r + i + 1) vote for } E_r.
    \)
    \(\text{else:}
    \)
    \(\text{vote } \leftarrow \text{true.}
    \)
    \(\text{E}_{r+1} \text{’s vote, whether } E_r \text{ is famous, is a majority voting among the witnesses it strongly sees.}
    \)
    \(\text{This is a normal round.}
    \)
    \(\text{E}_{r+1} \text{ votes in any case according to majority decision.}
    \)
    \(\text{if a supermajority is achieved during voting (i.e.,}
    \)
    \(\frac{2}{3} \text{nodes of } E_{r+1+i} \text{ votes whether } E_r \text{ is famous.}
    \)
    \(\text{Otherwise, } E_{r+1+i} \text{ “flips a coin”.}
    \)
    \(\text{Begin coin round.}
    \)
    \(\text{if } \text{mod coinRound} > 0:
    \)
    \(\text{tmpVotes}[\text{eventID}_{r+1+i}] \leftarrow \text{vote.}
    \)
    \(\text{If supermajority is reached, } E_{r+1+i} \text{ votes.}
    \)
    \(\text{Otherwise, } E_{r+1+i} \text{ “flips a coin”.}
    \)
```

Description of \(M_{\text{node}}\) (cont.):

Go over events in topological order, i.e., events are always “visited” after their parents.
Returns true or false whether the first event (first argument) “can see” the second event (second argument).
The function \(\text{StronglySees}\) returns true or false whether the first event “can strongly see” the second event.

**Figure 3.5:** The model of a Hashgraph node \(P^H_{\text{node}}\) (Part 3).
3.2 An iUC Model for Hashgraph

Description of $M_{\text{node}}$ (cont.):

```plaintext
Procedures and Functions:

```FindOrder```
  function findOrder(): {Establish total order of events and transactions (cf. Figure 2.5).
    maxRoundCreated ← 1.
    for all (..., eventID, ...) ∈ events do:
      maxRoundCreated ← max{maxRoundCreated, roundCreated[eventID]}.
    for all (E1 ← (...), eventID, ...) ∈ events do:
      for all r = 1, ..., maxRoundCreated do:
        if isRoundReceived(E1, r):
          roundReceived[eventID] ← r.
          S ← ∅.
        for all (E2 ← (...), selfPar, ... ∈ events) do:
          E2-selfPar ← (...), eventID = selfPar.
          if ∃ (E3 ← (...), eventID, ...) ∈ events s.t.
            IsSelfAncestorOf(E2, E3) = true
            ∧ roundCreated[eventID] = r
            ∧ IsFamous[eventID] = true
            ∧ IsUniqueFamousWitness(E3, r) = true
            ∧ IsAncestorOf(E1, E2) = true
            ∧ IsAncestorOf(E3, E2-selfPar) = false:
              S.add(E3).
        S.add(GetMedianWrtTimestamp(S)).
    roundReceivedEvents ← SortByRoundReceived(roundReceivedEvents).
    roundReceivedEvents ← SortTiesByEventConsensusNumber(roundReceivedEvents).
    roundReceivedEvents ← SortTiesBySignatureLexicographically(roundReceivedEvents).
    for all (entry ← (ctr, txId, eventID, msg)) ∈ msgList s.t. ctr ≠ 0 do:
      msgList.remove(entry).
    entryCounter ← 1.
    for i = 0, ..., roundReceivedEvents.length do:
      if roundReceivedEvents[i] ≠ ?
      roundReceivedEvents[i] ← SortByRoundReceived(roundReceivedEvents).
    M ← {e, txId, eventID, msg} ∈ msgList | msg ∈ txs.
    for all (e, txId, eventID, msg) ∈ M, in ascending order w.r.t. txId do:
      msgList.add((entryCounter, txId, eventID, msg)).
    entryCounter ← entryCounter + 1.
  function isRoundReceived(E1, r):
    for all (E2 ← (...), eventID, ...) ∈ events do:
      if roundCreated[eventID] ≤ r
      ∧ IsWitness[eventID] = true
      ∧ IsFamous[eventID] = true:
        return false.
      for all E3 ∈ WitnessesOfRound(r) do:
        if IsAncestorOf(E1, E3) = true
        ∧ IsUniqueFamousWitness(eventID, r):
          E1 must be an ancestor of all round r unique famous
          witnesses.
        return true.
    return true.

```

Description of $M_{\text{node}}$ continues in Figure 3.7.

*In the absence of forking, each famous witness is also a unique famous witness (cf. [2]).*
### Procedures and Functions:

**Function verifyBasicGraphCorrectness(events):**

Check basic graph correctness for new synced events, see Definition 5.

```plaintext
for all E ∈ events do:
  if ((pid, eventID, selfPar, otherPar, tzs, ts, σ) ← E) cannot be parsed s.t.
    1. pid ∈ nodes, tzs, σ ∈ {0, 1}°, ts ∈ N
    2. eventID, selfPar, otherPar ∈ {0, 1}°:
      return false.
  if E ≠ (⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥):
    return false.

for all (pid, eventID, selfPar, otherPar, tzs, ts, σ) ∈ events \ {((⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥)} do:
  if ¬(∃(pid, eventID', tzs', ts', σ) ∈ events s.t.
      selfPar = eventID'
     ):
    return false.
  if ¬(∃(pid', eventID', tzs', ts', σ) ∈ events s.t.
      otherPar = eventID' ∧ pid' ≠ pid
     ):
    return false.
  if verifySig(pid, (selfPar, otherPar, tzs, ts, σ)) = false:
    return false.
  if hash((selfPar, otherPar, tzs, ts)) ≠ eventID:
    return false.
  if (⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥) ∈ events:
    return false.
  if ∃(⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥) ∈ events s.t.
    (selfPar, otherPar, tzs, ts) ≠ (⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥):
    return false.
  if ∃(pid, eventID, tzs, ts) ∈ events s.t.
    pid, eventID' ≠ eventID:
    return false.
return true.
```

**Function sign(msg):**

Sign message at \( F_{cert} \).

```plaintext
send (Sign, msg) to (pidCur, (pidCur, sidCur, \( F_{node} \) : node), \( F_{cert} \) : signer);
wait for (Signature, σ).
return σ.
```

**Function verifySig(pid, msg, σ):**

Verify signature at \( F_{cert} \).

```plaintext
send (Verify, msg, σ) to (pidCur, (pid, sidCur, \( F_{node} \) : node), \( F_{cert} \) : verifier);
wait for (VerResult, result).
return result.
```

**Function hash(msg):**

Generate “hash” at \( F_{ro} \).

```plaintext
send (Hash, msg) to (pidCur, sidCur, \( F_{ro} \) : randomOracle);
wait for (Hash, h).
return h.
```

Figure 3.7: The model of a Hashgraph node \( P^H_{node} \) (Part 5).
3.2 An iUC Model for Hashgraph

<table>
<thead>
<tr>
<th><strong>Participating roles:</strong></th>
<th>( {\text{signer, verifier}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Corruption model:</strong></td>
<td>incorruptible</td>
</tr>
</tbody>
</table>
| **Protocol parameters:**| \begin{align*}
- \ & p \in \mathbb{Z}[x]. \\
- \ & \eta \in \mathbb{N}. \\
- \ & \text{sig}. \\
- \ & \text{ver}. \\
- \ & \text{gen}.
\end{align*}\bigskip
\begin{align*}
\text{Polynomial that bounds the runtime of the algorithms provided by the adversary.} \\
\text{The security parameter.} \\
\text{Signing algorithm, outputs a signature } \sigma \text{ on input } (\text{msg, sk}). \\
\text{Signature verifying algorithm, outputs verification result on input } (\text{msg, } \sigma, \text{pk}). \\
\text{Key generation algorithm, outputs } (\text{pk, } \sigma) \text{ on input } 1^x.
\end{align*}|

<table>
<thead>
<tr>
<th><strong>Description of M_{\text{signer, verifier}}:</strong></th>
</tr>
</thead>
</table>
| **Implemented role(s):** \( \{\text{signer, verifier}\} \)
| **Internal state:** |
| \begin{align*}
- \ & (pk, sk) \in \{0, 1\}^* \cup \{\bot\}^2 = (\bot, \bot). \\
- \ & \text{pidowner} \in \{0, 1\}^* \cup \{\bot\} = \bot. \\
- \ & \text{msglist} \subseteq \{0, 1\}^* = \emptyset. \\
- \ & \text{corrupted} \in \{\text{true, false}\} = \text{false}.
\end{align*}\bigskip
\begin{align*}
\text{Key pair.} \\
\text{Party ID of the key owner.} \\
\text{Set of recorded messages.} \\
\text{Is signature key corrupted?}
\end{align*}|

| **CheckID(pid, sid, role):** |
| Check if sid can be parsed as \( (\text{pid}', \text{sid}', \text{role}') \): |
| If this check fails, output reject. Otherwise, accept all entities with the same SID. |

| **Corruption behavior:** |
| DetermineCorrStatus(pid, sid, role): |
| return corrupted. |

| **Initialization:** |
| \begin{align*}
(pk, sk) \leftarrow \text{Gen}(1^\eta). \\
(pid, sid, role) \leftarrow \text{parse sid}\_\text{cur}. \\
pidowner \leftarrow \text{pid}.
\end{align*}\bigskip
\begin{align*}
\text{Generate public/secret key pair.} \\
\text{Assign key owner.}
\end{align*}|

| **Main:** |
| \begin{align*}
\text{recv} \ (\text{Sign, msg}) \text{ from I/O to } (\text{pidowner, , signer}): \\
\sigma \leftarrow \text{sig}(\text{msg, sk}). \\
\text{msglist}.\text{add}(\text{msg}). \\
\text{reply} \ (\text{Signature, } \sigma).
\end{align*}\bigskip
\begin{align*}
\text{Sign message; only pidowner is permitted to sign with its key.} \\
\text{Record msg for verification and return signature.}
\end{align*}|

| \begin{align*}
\text{recv} \ (\text{Verify, msg, } \sigma) \text{ from I/O to } (\text{_, _, verifier}): \\
b \leftarrow \text{ver}(\text{msg, } \sigma, \text{pk}). \\
\text{if } b = \text{true} \land \text{msg} \notin \text{msglist} \land \text{corrupted} = \text{false}: \\
\text{reply} \ (\text{VerResult, false}). \\
\text{else: } \text{reply} \ (\text{VerResult, } b).
\end{align*}\bigskip
\begin{align*}
\text{Verify signature.} \\
\text{Prevent forgery.} \\
\text{Return verification results.}
\end{align*}|

| \begin{align*}
\text{recv} \ \text{corruptSigKey} \text{ from NET:} \\
\text{corrupted} \leftarrow \text{true}. \\
\text{reply} \ (\text{corruptSigKey, ok}).
\end{align*}\bigskip
\begin{align*}
\text{Allow network attacker to corrupt signature keys.}
\end{align*}|

Figure 3.8: The ideal signature functionality \( \mathcal{F}_\text{cert} \) (cf. [11]).
3 Security Model of Hashgraph

### Description of the protocol \( \mathcal{F}_{ro} = \{\text{randomOracle}\} \):

<table>
<thead>
<tr>
<th>Participating roles:</th>
<th>{\text{randomOracle}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corruption model</td>
<td>\text{incorruptible}</td>
</tr>
<tr>
<td>Protocol parameters:</td>
<td>( \eta \in \mathbb{N} )</td>
</tr>
</tbody>
</table>

\( \mathcal{F}_{ro} \) is a random oracle that implements the protocol. The security parameter and length of the hash are denoted by \( \eta \).

### Description of \( M_{\text{randomOracle}} \):

| Implemented role(s): | \{\text{randomOracle}\} |

\( M_{\text{randomOracle}} \) is the implemented role of the random oracle. The internal state includes the history of recorded hashes.

### Description of the ideal clock \( \mathcal{F}_{clock} = \{\text{clock}\} \):

<table>
<thead>
<tr>
<th>Participating roles:</th>
<th>{\text{clock}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corruption model</td>
<td>\text{incorruptible}</td>
</tr>
</tbody>
</table>

### Description of \( M_{\text{clock}} \):

<table>
<thead>
<tr>
<th>Implemented role(s):</th>
<th>{\text{clock}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal state:</td>
<td>( \tau \in \mathbb{N} = 0 )</td>
</tr>
</tbody>
</table>

\( M_{\text{clock}} \) is the implemented role of the ideal clock. The current time is initially 0.

### Figure 3.9: The random oracle \( \mathcal{F}_{ro} \) (cf. [7]).

| recv \((\text{Hash}, x)\) from I/O or NET: | Requesting \( \mathcal{F}_{ro} \) for “hashes”.
|-----------------------------------------|----------------------------------|
| if \( \exists h \in \{0, 1\}^\eta \) s.t. \((x, h) \in \text{hashHistory}\): | Extract existing value from \( \text{hashHistory} \).
| reply \( h \) | Store generated key-value pair in \( \text{hashHistory} \).
| else: | Generate “hash value” uniformly at random.
| \( h \xleftarrow{\$} \{0, 1\}^\eta \) | Store generated key-value pair in \( \text{hashHistory} \).
| \( \text{hashHistory} \leftarrow \text{hashHistory}.\text{add}((x, h)) \) | |
| reply \((\text{Hash}, h)\) | |

| CheckID \((\text{pid}, \text{sid}, \text{role})\): | Accept all messages for the same SID. |

<table>
<thead>
<tr>
<th>Main:</th>
<th>Triggering a clock update increases the time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>recv \text{UpdateRound} from I/O or NET:</td>
<td>Handling reads from the clock.</td>
</tr>
<tr>
<td>( \tau \leftarrow \tau + 1 )</td>
<td></td>
</tr>
<tr>
<td>recv \text{GetCurRound} from I/O or NET:</td>
<td></td>
</tr>
<tr>
<td>reply ((\text{GetCurRound}, \tau))</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 3.10: The ideal clock functionality \( \mathcal{F}_{clock} \).
### 3.2 An iUC Model for Hashgraph

Description of the protocol $\mathcal{F}_\text{init}^H = \{\text{init}\}$:

<table>
<thead>
<tr>
<th>Participating roles:</th>
<th>${\text{init}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corruption model:</td>
<td>incorruptible</td>
</tr>
<tr>
<td>Protocol parameters:</td>
<td></td>
</tr>
<tr>
<td>- $\text{nodes} \subseteq {0, 1}^*$</td>
<td>The identities of the Hashgraph nodes in all sessions of hasgraph instances.</td>
</tr>
</tbody>
</table>

**Implemented role(s):** $\{\text{init}\}$

**Subroutines:** $\mathcal{P}_\text{node}^H : \text{node}$

**CheckID**($\text{pid}, \text{sid}, \text{role}$):

- Accept all entities with the same SID.

**Initialization:**

- for all $\text{pid} \in \text{nodes}$ do:
  - $\text{init}(\text{pid}, \text{sid}_{\text{cur}}, \mathcal{P}_\text{node}^H, \text{node})^H$.

**Main:**

- $\text{init}$ is a macro in iUC to initialize an entity.

*By this construction, we allow multiple Hashgraph instances to run parallel in one run of the system.*

*Initialize all nodes in the current Hashgraph instance; such instance is uniquely identified by the session ID of $\text{sid}_{\text{cur}}$, i.e., all nodes have the same SID $\text{sid}_{\text{cur}}$. Notice that $\mathcal{P}_\text{client}^H$, $\mathcal{F}_\text{cert}$, $\mathcal{F}_\text{ro}$, and $\mathcal{F}_\text{clock}$ do not have to be initialized.*

*Do nothing after initialization of the Hashgraph instance.*

**Figure 3.11:** The ideal initialization functionality $\mathcal{F}_\text{init}^H$. 

---

29
3 Security Model of Hashgraph

---

**Participating roles:** \{judge\}

**Corruption model:** incorruptible

**Protocol parameters:**
- \(\eta \in \mathbb{N}\).
- \(\text{nodes} \subseteq \{0, 1\}^*\).
- \(\text{coinRound} \in \mathbb{N}\). \{The security parameter.\}
- \(\text{nodes} \subseteq \{0, 1\}^*\). \{The identities of the Hashgraph nodes.\}
- \(\text{(tmpVotes)} \subseteq \{0, 1\}^*\). \{After coinRound rounds in Hashgraph, there is a “coin round” during virtual voting.\}

**Internal state:**
- \(W \subseteq \mathbb{N} \times \text{nodes} \times ((\text{nodes} \times \{0, 1\}^n) \times \{0, 1\}^n) \times \{0, 1\}^n \times \{0, 1\}^n \times \mathbb{N} \times \{0, 1\}^n \cup \{\bot\}) = \emptyset\). \{The collected evidences are of form \((\text{ctr}, \text{pid}, G)\), where \(G\) is a hashgraph with entries as in \(P^\text{node}_i\) (cf. events in Figure 3.3). Initially 0, since there is no evidence at the beginning.\}
- \(\text{states} : \text{nodes} \rightarrow \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n\).
- \(\text{evidenceCounter} \in \mathbb{N} = 0\). \{Counter for internally sorting evidence data; initially 0.\}
- \(\text{msglist}_{\text{max}} \subseteq \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \mathbb{N} \times \{0, 1\}^n \cup \{\bot\} = \emptyset\). \{A list of transactions: entries are of form \((\text{counter}, \text{txId}, \text{eventID}, \text{tx}, \sigma)\); solely used for calculating purposes in divideRounds, decideFame, and findOrder.\}
- \(\text{verdicts} \subseteq \{0, 1\}^n = \emptyset\). \{Set of recorded verdicts; initially 0.\}
- \(\text{forkingNodes} \subseteq \text{nodes} = \emptyset\). \{Set of all node participants that created a fork.\}
- \(\text{consistencyVerdicts} \subseteq \text{nodes} = \emptyset\). \{Set of all node participants that violated self or node-consistency (cf. \(\gamma_1, \gamma_2\) in Definition 11).\}
- \(\text{events} \subseteq \{\text{nodes} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \mathbb{N} \times \{0, 1\}^n \cup \{\bot\}\} = \emptyset\). \{Set of events of form \((\text{pid}, \text{eventID}, \text{selfParent}, \text{otherParent}, \text{transactions}, \text{tx}, \sigma)\) – only used for internal computations.\}
- \(\text{msglist} \subseteq \mathbb{N} \cup \{\epsilon\} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \mathbb{N} \times \{0, 1\}^n \cup \{\bot\} = \emptyset\). \{A list of transactions: entries are of form \((\text{counter}, \text{txId}, \text{eventID}, \text{tx}, \sigma)\) – only used for internal computations.\}
- \(\text{roundReceivedEvents} : \mathbb{N} \rightarrow \{\text{nodes} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \mathbb{N} \times \{0, 1\}^n \cup \{\bot\}\} = \emptyset\). \{Stores all events with a round received number in consecutive order, starting at index 0. Also used for sorting events; see function findOrder.\}
- \(\text{roundCreated} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathbb{N}\). \{The round created of an event (identified via eventID); initially 0 for all entries.\}
- \(\text{roundReceived} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathbb{N}\). \{The round received of an event (identified via eventID); initially 0 for all entries.\}
- \(\text{tmpVotes} : \{0, 1\}^n \rightarrow \{?, \text{true}, \text{false}\} \). \{Used to store votes during virtual voting; initially ? for all entries.\}
- \(\text{eventConsensusNumber} : \{0, 1\}^n \cup \{\bot\} \rightarrow \mathbb{N} \cup \{\epsilon\}\). \{Stores the total order of events from events; initially \(\epsilon\) for all entries.\}
- \(\text{isWitness} : \{0, 1\}^n \cup \{\bot\} \rightarrow \{?, \text{true}, \text{false}\}\). \{Stores whether an event is a witness; initially ? for all entries.\}
- \(\text{isFamous} : \{0, 1\}^n \cup \{\bot\} \rightarrow \{?, \text{true}, \text{false}\}\). \{Stores whether an event is famous; initially ? for all entries.\}

**CheckID** \((\text{pid}, \text{std}, \text{role})\):
Accept all messages with the same SID.

**Message Preprocessing:**
Description of \(M_{\text{judge}}\) continues in Figure 3.13.

---

\*Multiple verdicts of \(F_{\text{judge}}\) should be interpreted as “and connected”, e.g., for verdicts = \(\{\text{dis}(\text{pid}_1), \text{dis}(\text{pid}_2)\}\), \(F_{\text{judge}}\) states (implicitly) the verdict \(\text{dis}(\text{pid}_1) \land \text{dis}(\text{pid}_2)\).

---

**Figure 3.12:** The judging functionality \(F^H_{\text{judge}}\) for the Hashgraph model (Part 1).
Description of $M_{\text{judge}}$ (cont.):

**Message Preprocessing:**

recv $\text{msg}$ from $I/O$:
- if $\exists \text{pid} \in \text{nodes}$ s.t. $\text{dis}(\text{pid}) \notin \text{verdicts}$:
  - abort. \hspace{1em} \{If there exists no honest party, abord.\}

Main:

recv $\text{GetVerdicts}$ from $I/O$ or $\text{NET}$:
- reply $\langle \text{GetVerdicts}, \text{verdicts} \rangle$.

recv $(\text{EvidenceNode, events})$ from $I/O$ s.t. $\text{pid}_{\text{call}} \in \text{nodes}$:
- for all $E \in \text{events}$ do:
  - if $((\text{pid}, \text{eventID}, \text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}, \text{σ}) \leftarrow E)$ can not be parsed s.t.
    - 1. $\text{pid} \notin \text{nodes}$, $\text{txs}, \text{σ} \in \{0, 1\}^*$, $\text{ts} \in \mathbb{N}$
    - 2. $\text{eventID}, \text{selfPar}, \text{otherPar} \in \{0, 1\}^*$:
      - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Nodes have to report well-formed evidence (cf. Definition 6, Well-Formed Data (i)); $\alpha_2$.\}
      - nodes.\$\text{add}(\text{evidenceCounter, \text{pid}_{\text{call}}, \text{events})}$. \hspace{1em} \{Record new (witness) data.\}
      - evidenceCounter $\leftarrow$ evidenceCounter + 1.
    - if $((\text{pid}, \text{eventID}, \text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}, \text{σ}) \in \text{events}) \setminus \{(\bot, \bot, \bot, \bot, \bot)\}$ do:
      - if $\neg((\exists (\text{pid}, \text{eventID}', \ldots, \ldots)) \in \text{events}$ s.t.
        - selfPar $= \text{eventID}'$: \hspace{1em} \{All events, except for the genesis event, have to suffice a complete self-ancestor construction (cf. Definition 5, I.(i)); $\alpha_2$.\}
        - verdicts.add(dis($\text{pid}_{\text{call}}$)).
      - if $\neg((\exists \text{pid}', \text{eventID}''', \ldots, \ldots)) \in \text{events}$ s.t.
        - otherPar $= \text{eventID}'''' \land \text{pid}' \neq \text{pid}$:
          - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{All events, except for the genesis event, must satisfy a complete ancestor construction (cf. Definition 5, I.(ii)); $\alpha_2$.\}
      - if $\text{verifySig}((\text{pid}, (\text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}), \text{σ}) = \text{false}$:
        - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Nodes need to report events with valid signatures (cf. Definition 5, 2); $\alpha_2$.\}
      - if $\text{hash}((\text{selfPar}, \text{otherPar}, \text{txs}, \text{ts),} \neq \text{eventID}$:
        - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Nodes must report valid events (cf. Definition 5, 3); $\alpha_2$.\}
      - if $(\bot, \bot, \bot, \bot, \bot, \bot) \notin \text{events}$:
        - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Ensure legitimate root constructions (cf. Definition 5, 4); $\alpha_2$.\}
      - if $\exists (\bot, \bot, \bot, \bot, \bot) \in \text{events}$ s.t.
        - $\text{verifySig}((\text{pid}, (\text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}), \text{σ}) = \text{false}$:
          - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Ensure legitimate root construction; $\alpha_2$.\}
      - if $\exists (\text{pid}, \text{eventID}, \bot, \bot, \bot, \bot) \notin (\bot, \bot, \bot, \bot, \bot, \bot)$ s.t.
        - $\text{eventID} \neq \text{eventID}'$:
          - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Ensure legitimate root construction; $\alpha_2$.\}
      - if $\exists (\text{pid}, \text{eventID}, \bot, \bot, \bot, \bot, \bot) \in \text{events}$ s.t.
        - $\text{verifySig}((\text{pid}, (\text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}), \text{σ}) = \text{false}$:
          - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Ensure fork-freeness for all participants $\text{pid} \in \text{nodes}$ (cf. Definition 8); $\beta_1$.\}
    - $\text{ EW} \leftarrow \bigcup (\text{pid}, G)) \in \text{EW}.
    - for $((\text{pid}, \text{eventID}, \text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}, \text{σ}) \in \text{EW}$ do:
      - for $(\text{pid}, \text{eventID}, \text{selfPar}, \text{otherPar}, \text{txs}, \text{σ}) \in \text{EW}$ do:
        - if $\text{eventID} \neq \text{eventID}'$ $\land \text{selfPar} = \text{otherPar}'$:
          - if $\text{verifySig}((\text{pid}, (\text{selfPar}, \text{otherPar}, \text{txs}, \text{ts}), \text{σ}) = \text{true}$:
            - $\text{forkingNodes.add(pid)}$. \hspace{1em} \{Ensure fork-freeness for all participants $\text{pid} \in \text{nodes}$ (cf. Definition 8); $\beta_1$.\}
            - verdicts.add(dis($\text{pid}_{\text{call}}$)).
      - for all $\text{ctr} = 0, \ldots, \text{evidenceCounter} - 1$ do:
        - if $((\text{ctr}, \text{pid}_{\text{call}}, \text{G}')) \in \text{W}$:
          - if $\text{G}' \not\subseteq \text{events}$:
            - verdicts.add(dis($\text{pid}_{\text{call}}$)). \hspace{1em} \{Each submitted graph $\text{G}'$ by $\text{pid}_{\text{call}}$ has to be a proper superset of each previously submitted graph $\text{G}'$ by $\text{pid}_{\text{call}}$ (cf. Definition 6, valid evidence); $\alpha_3$.\}

$^a$By definition of Allow Adv Message in $M_{\text{node}}$, it always applies $\text{pid}_{\text{call}} = \text{pid}_{\text{call}}$

**Figure 3.13:** The judging functionality $J_{\text{judge}}^H$ for the Hashgraph model (Part 2).
3 Security Model of Hashgraph

Description of $M_{\text{judge}}$ (cont.):

Recall that $\text{msglist}$ is the longest known message list.

If $\text{msglist} \not\subseteq N \times \{0, 1\}^* \times \{0, 1\}^*$:

```plaintext
for all $(\text{ctr}, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$ do:
  if $\text{txId} \neq \text{hash}(\text{msg})$:
    $\text{verdicts}.\text{add}((\text{pid}_{\text{send}})\text{.dis}(\text{pid}_{\text{send}})).$
  break.
for $i = 1, \ldots, |\text{msglist}|$ do:
  if $(i, \_\ldots, \_)$ \notin $\text{msglist}$:
    $\text{verdicts}.\text{add}((\text{pid}_{\text{send}})\text{.dis}(\text{pid}_{\text{send}})).$
  break.
if $\text{msglist} \not\subseteq \text{consistencyVerdicts} \land \vert \text{forkingNodes} \vert < \frac{1}{3}$:
  Messages with “older” state information are ignored.
  Record update for $\text{pid}_{\text{send}}$.
  Record that $\text{msglist}$ is now the longest known message list.
  Recalculate longest state of nodes that have not yet been blurred for consistency violations.
if $\text{states}(\text{pid}_{\text{send}}) \not\subseteq \text{msglist}$:
  $\text{states}(\text{pid}_{\text{send}}) \leftarrow \text{msglist}$.
  $\text{msglist}_{\text{max}} \leftarrow \text{states}(\text{pid}_{\text{send}})$.
if $\text{msglist} \not\subseteq \text{states}(\text{pid}_{\text{send}}) \land \text{states}(\text{pid}_{\text{send}}) \not\subseteq \text{msglist}$:
  $\text{verdicts}.\text{add}((\text{pid}_{\text{send}})\text{.dis}(\text{pid}_{\text{send}})).$
  $\text{consistencyVerdicts}.\text{add}(\text{pid}_{\text{send}})$.
  $\text{msglist}_{\text{max}} \leftarrow \bigcup_{\text{pid} \in \text{nodes}}\text{consistencyVerdicts}.\text{states}[\text{pid}]$.
if $\text{states}(\text{pid}_{\text{send}}) \not\subseteq \text{msglist}_{\text{max}}$:
  Only true if $\text{msglist}_2$ and $\text{msglist}_{\text{max}}$ do not share a prefix.
  Some participant(s) must have misbehaved w.r.t. node-consistency.
  Let us find the culprit in $\text{generateReport}()$.
```

Procedures and Functions:

```plaintext
function $\text{generateReport}()$:
  for all $(\text{ctr}, \text{pid}, G) \in W \text{ s.t. } \text{dis}(\text{pid}) \notin \text{verdicts}$ do:
    if $\not\exists (\text{ctr}', \text{pid}, G') \text{ s.t. } \text{ctr}' > \text{ctr}$:
      events $\leftarrow G$, $\text{msglist} \leftarrow \emptyset$.
      $\text{roundCreated} \leftarrow 0$ for all entries.
      $\text{roundReceived} \leftarrow 0$ for all entries.
      $\text{isWitness} \leftarrow \emptyset$ for all entries.
      for all $(\_, \text{eventID}, \_\ldots, \_\times, \_\ldots, \_)$ \in $\text{events}$ do:
        for all $\text{msg} \in \text{txs}$ do:
          $\text{msglist}.\text{add}(\text{msg}, \text{hash}(\text{msg}), \text{eventID}, \text{msg})$.
        $\text{divideRounds}()$, $\text{decideFame}()$, $\text{findOrder}()$.
        if $\text{states}(\text{pid}) \not\subseteq \text{msglist}$:
          $\text{verdicts}.\text{add}((\text{pid})\text{.dis}(\text{pid}))$.
          $\text{consistencyVerdicts}.\text{add}(\text{pid}_{\text{send}})$.
      $\text{msglist}_{\text{max}} \leftarrow \bigcup_{\text{pid} \in \text{nodes}}\text{consistencyVerdicts}.\text{states}[\text{pid}]$.
  function $\text{verifySig}(\text{pid}, \text{msg}, \sigma)$:
    $\text{send}(\text{Verify}, \text{msg}, \sigma)$ to $\text{pid}_{\text{cur}}$, ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{F}_{\text{node}}^{\text{H}} : \text{node}, \text{F}_{\text{cert}} : \text{verifier})$.
    $\text{wait for} (\text{VerResult}, \text{result})$.
    return $\text{result}$.
  function $\text{hash}(\text{msg})$:
    $\text{send}(\text{Hash}, \text{msg})$ to $\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{F}_{\text{FO}} : \text{randomOracle}$.
    $\text{wait for} (\text{Hash}, \text{h})$.
    return $\text{h}$.
```

Description of $M_{\text{judge}}$ continues in Figure 3.15.

Figure 3.14: The judging functionality $\mathcal{F}_{\text{judge}}^H$ for the Hashgraph model (Part 3).
Procedures and Functions:

function divideRounds():
    for all \( E \in \text{events}, \) in topological order do:
        \( \langle \text{eventID}, \text{selfPar}, \text{otherPar}, \ldots \rangle \leftarrow \text{parse} \ E. \)
        \( r \leftarrow 1. \)
        if \( \text{eventID} \neq \bot \):
            \( r \leftarrow \max \{ \text{roundCreated}[	ext{selfPar}], \text{roundCreated}[	ext{otherPar}] \} \).
            \( S \leftarrow \{ \langle \text{eventID}, \text{otherPar}, \ldots \rangle \in \text{events} \mid \text{roundCreated}[	ext{eventID}] = r \}. \)
            \( S \leftarrow \{ E' \in S \mid \text{StronglySees}(E, E') = \text{true} \}. \)
            if \( |S| > \frac{2}{3} \) \text{nodes}:
                \( \text{roundCreated}[	ext{eventID}] \leftarrow r + 1. \)
            else:
                \( \text{isWitness}[	ext{eventID}] \leftarrow \text{true}. \)
                \( \text{tmpVotes}[	ext{eventID}] \leftarrow \text{false}. \)
        if \( (\text{selfPar} = \bot) \lor (\text{roundCreated}[	ext{eventID}] > \text{roundCreated}[	ext{selfPar}]) \):
            \( \text{if} \) is the geneis event, or it has the geneis event as self-parent, or \( i \) it is the first event of pid within a new round, \( E \) is a witness.

function decideFame():
    for all \( E_r \in \text{events}, \) in order from earlier rounds to later do:
        \( \langle \text{eventID}, \text{otherPar}, \ldots \rangle \leftarrow \text{parse} \ E_r. \)
        \( \text{tmpVotes} \leftarrow \bot \) for all entries.
        if \( \text{isFamous}[	ext{eventID}] = \text{true} \):
            \( \text{if} \) \( \text{isWitness}[	ext{eventID}] \):
                \( \text{for all } r \in \text{WitnessesOfRound}(r + 1) \text{ do:} \)
                    \( \langle \text{eventID}, \text{otherPar}, \ldots \rangle \leftarrow \text{parse} \ E_{r+1}. \)
                    \( \text{if} \) \( \text{Sees}^b(E_{r+1}, E_r) = \text{true} \):
                        \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{true}. \)
                        \( \text{All witnesses in the next round (w.r.t. } E_r \text{'s round) vote for } E_r. \)
                    else:
                        \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{false} \).
            \( \text{for all } i = 1, \ldots \text{s.t. } \{ W_{r+i} \leftarrow \text{WitnessesOfRound}(r + i + 1) \} \neq \emptyset \text{ do:} \)
                \( \langle \text{eventID}, \text{otherPar}, \ldots \rangle \leftarrow \text{parse} \ E_{r+i+1}. \)
                \( W \leftarrow \{ E_{r+i} \in \text{WitnessesOfRound}(r + i) \mid \text{StronglySees}(E_{r+i+1}, E_r) = \text{true} \}. \)
                \( y \leftarrow \{ \langle \text{eventID}, \text{otherPar}, \ldots \rangle \in W \mid \text{tmpVotes}[	ext{eventID} + 1] = \text{true} \}. \)
                \( n \leftarrow \{ \langle \text{eventID}, \text{otherPar}, \ldots \rangle \in W \mid \text{tmpVotes}[	ext{eventID} + 1] = \text{false} \}. \)
                \( \text{if } y \leq \frac{|W|}{2} \): \( \text{vote} \leftarrow \text{true}. \)
                \( \text{else} \):
                    \( \text{vote} \leftarrow \text{false}. \)
            \( \text{if } i \text{ mod coinRound} > 0 : \)
                \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{vote}. \)
                \( \text{else} : \)
                    \( \text{if } y \geq \frac{2}{3} \) \text{nodes} \( \lor n \geq \frac{2}{3} \) \text{nodes}:
                        \( \text{isFamous}[	ext{eventID}] \leftarrow \text{vote}. \)
                        \( \text{break out of } i \text{-loop}. \)
                    \( \text{else} : \)
                        \( \text{if } y \geq \frac{2}{3} \) \text{nodes} \( \land n \geq \frac{2}{3} \) \text{nodes}:
                            \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{vote}. \)
                            \( \text{else} : \)
                                \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{false} \).
                                \( \text{if } \) supermajority is reached, \( E_{r+i+1} \) votes whether \( E_r \) is famous.
                                \( \text{else} : \)
                                    \( \text{if } \) last bit of \( \sigma_{r+i+1} \) is \( 1 \):
                                        \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{true}. \)
                                        \( \text{else} : \)
                                            \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{false} \).
                                \( \text{else} : \)
                                    \( \text{if } \) supermajority is reached, \( E_{r+i+1} \) "flips a coin".
                                    \( \text{Otherwise, } E_{r+i+1} \) "flips a coin".
                                    \( \text{else} : \)
                                        \( \text{tmpVotes}[	ext{eventID} + 1] \leftarrow \text{false} \).

Description of \( M_{\text{judge}} \) continues in Figure 3.16.

Footnotes:
1. Go over events in topological order, i.e., events are always “visited” after their parents.
2. Returns true or false whether the first event (first argument) “can see” the second event (second argument).
3. The function \( \text{StronglySees} \) returns true or false whether the first event “can strongly see” the second event.

Figure 3.15: The judging functionality \( J^H_{\text{judge}} \) for the Hashgraph model (Part 4).
Procedures and Functions:

```plaintext
function findOrder():
maxRoundCreated ← 1.
for all (E₁ → eventID₁, s.t., ...) ∈ events do:
maxRoundCreated ← max{maxRoundCreated, roundCreated[eventID]}.
for all (E₂ → eventID₂, s.t., ...) ∈ events do:
for all r = 1, ..., maxRoundCreated do:
  if isRoundReceived(E₁, r):
    roundReceived[eventID₁] ← r.
S ← ∅.
for all (E₂ → eventID₂, s.t., ...) ∈ events do:
  E₂-selfPar ← (E₂, s.t., eventID = selfPar).
  if ∃ E₃ ← (E₃, s.t., eventID = selfPar):
    IsAncestorOf(E₂, E₃) = true
    ∧ roundCreated[eventID₃] = r
    ∧ isFamous[eventID₃] = true
    ∧ IsUniqueFamousWitness(E₃, r) = true
    ∧ IsAncestorOf(E₁, E₂) = true
    ∧ IsAncestorOf(E₁, E₂-selfPar) = false:
      S.add(E₂).
eventConsensusNumber[eventID₁] ← GetMedianWrtTimestamp(S).
counter ← 0.
roundReceivedEvents ← list of all entries.
for all (E → eventID₁, s.t., ...) ∈ events do:
  if roundReceived[eventID] > 0:
    roundReceivedEvents[counter] ← E.
    counter ← counter + 1.
for all (E → eventID₁, s.t., ...) ∈ events do:
  msglist.remove(entry).
msglist.add((e, txId, eventID, msg)).
entryCounter ← 1.
for s = 0, ..., s.t. roundReceivedEvents[s] ≠ ∅ do:
  roundReceivedEvents[s] ← list of all entries in ascending order.
  M ← {msglist | msg ∈ txs}.
for all (e, txId, eventID, msg) ∈ M, in ascending order w.r.t. txId do:
  msglist.add((entryCounter, txId, eventID, msg)).
msglist.remove((e, txId, eventID, msg)).
entryCounter ← entryCounter + 1.
```

function isRoundReceived(E₁, r):
for all (E₂ → eventID₂, s.t., ...) ∈ events do:
  if roundCreated[eventID₂] ≤ r
    ∧ isWitness[eventID₂] = true
    ∧ isFamous[eventID₂] = true:
    return false.
for all E₃ ∈ WitnessesOfRound(r) do:
  (E₃, s.t., eventID₃) ← E₃.
  if IsAncestorOf(E₁, E₃) = false
    ∧ IsUniqueFamousWitness(eventID₃, r):
    return false.
return true.
```

Figure 3.16: The judging functionality \( J_{\text{judge}}^H \) for the Hashgraph model (Part 5).
4 Accountability

Accountability is a security concept that does not prevent (directly) certain security goals from being violated. However, it does require that all involved misbehaving participants can be uniquely identified. This is achieved by means of a judging procedure or judge, which is a algorithm that outputs verdicts for misbehaving participants. By this, participants are incentivized to follow protocol rules.

For practical applications, it is usually necessary to identify at least one individual participants for a misbehavior. If for instance, only a group of participants can be held accountable, and it is known that not all participants in this group must have misbehaved, then each party will deny responsibility, rendering this “weaker” accountability notion unsuitable for practical purposes. Therefore, at least one individual participant must be held accountable. The authors in [10, 14] refer to this as individually accountability. All of our proofs in Chapter 5 of accountability w.r.t. a security goal will fulfill this notion.

4.1 Formal Definition of Accountability

In the following, we aim to formalize the concepts and objectives of accountability. For this work, we apply the formal definition of accountability in [10], which adapts and enhances the accountability framework from [14] to (i) permit the judge to render a verdict at any point during a run and to (ii) allow dynamic corruption (in addition, to only static corruption).

Let $Q$ be a system of machine instances (as in the iUC model) that includes a judge machine instance $J$, and let $\Sigma$ denote the set of all participants in $Q$. In the iUC framework, participants $p \in \Sigma$ are usually uniquely identified by party IDs (PIDs), and $p$ typically belongs to exactly one instance of a machine in a run of $Q$.

During a run of the system $Q$, we expect $J$ to render verdicts if a participant misbehaved. Formally, a verdict of $J$ is a boolean formula $\psi$ built from atomic propositions of the form $\text{dis}(p)$ for $p \in \Sigma$, which indicates (a allegedly) misbehavior of $p$, i.e., $p$ behaved dishonestly by violating prescribed protocol rules. Consider, for instance, the verdict $\text{dis}(p_1) \lor \text{dis}(p_2)$ by $J$; this states the judge’s conviction that at least one of the participants $p_1$ and $p_2$ misconducted. Analogously, if $J$ renders a verdict $\text{dis}(p_1) \lor \text{dis}(p_2)$, she claims that both $p_1$ and $p_2$ misbehaved. We denote the set of all verdicts by $\mathcal{V}$. In a run of $Q$, the judge $J$ can state multiple verdicts $\psi_1, \ldots, \psi_k \in \mathcal{V}$, which is semantically equivalent to $J$ claiming the verdict $\psi_1 \land \ldots \land \psi_k$. 
4 Accountability

Since a verdict is a boolean formula, it can be evaluated to true or false using propositional logic. Let \( \omega \) be a run of \( Q \) with \( [\omega] \) denoting some point in this run. For a participant \( p \in \Sigma \), the proposition \( \text{dis}(p) \) is set to true iff \( p \) is corrupted at point \( [\omega] \), and set to false otherwise\(^1\). Formally, we will write \( [\omega] \models \psi \) iff the verdict \( \psi \) is true at \( [\omega] \); otherwise, we write \( [\omega] \not\models \psi \).

In order to formalize and precisely capture the intended level of accountability, [14] introduces the notion of security properties and accountability constraints: A security property (or property) \( \alpha \) of a protocol \( Q \) is a subset of all runs of \( Q \). Intuitively, \( \alpha \) contains all runs of \( Q \) in which some predefined (security) goal is not satisfied as a consequence of some misbehaving protocol participant(s). By means of a property \( \alpha \) of \( Q \), we define an accountability constraint \( C \) of a protocol \( Q \) to be a tuple \( (\alpha, \psi_1, \ldots, \psi_k) \), written \( (\alpha \Rightarrow \psi_1 | \cdots | \psi_k) \), with one or more verdicts \( \psi_1, \ldots, \psi \in \mathcal{V} \). Intuitively, \( C \) specifies a set of minimal verdicts \( \psi_1, \ldots, \psi_k \), wherein at least one of these verdicts are assumed to be rendered by \( J \) if some security property is violated (i.e., the run of the protocol is in \( \alpha \)); however, \( J \) is free to state stronger verdicts that logically imply at least one of the minimal verdicts of \( C \) (in the sense of propositional logic). We say, for a run \( \omega \) of \( Q \), that \( J \) ensures \( C \) in \( \omega \) if either \( \omega \not\models \alpha \) or at some point in the run of \( \omega \) the judge renders a verdict \( \psi \) that implies one of \( \psi_1, \ldots, \psi_k \).

**Example 2.** We illustrate the notion of accountability constraints in the following examples, in which we consider \( n \) hashgraph nodes \( \text{pid}_i \) managed by different parties that are supposed to be blamed by \( J \) for any misbehavior defined by \( \alpha \):

\[
C_1 \triangleq (\alpha \Rightarrow \text{dis}(\text{pid}_1) | \cdots | \text{dis}(\text{pid}_n)) \tag{4.1}
\]

\[
C_2 \triangleq (\alpha \Rightarrow \text{dis}(\text{pid}_1) \lor \cdots \lor \text{dis}(\text{pid}_n)) \tag{4.2}
\]

In this work, we are primarily interested in assuring consistency for hashgraph; thus, the property \( \alpha \) will later contain all runs of hashgraph where consistency is violated. Constraint \( C_1^H \) requires that if \( \alpha \) (e.g., consistency) is violated, then at least one node \( \text{pid}_i \) can be held accountable by \( J \) in this run, i.e., \( J \) ensures \( C_1^H \) in a run \( \omega \in \alpha \) by stating \( \text{dis}(\text{pid}_i) \) or, more generally, \( \text{dis}(\text{pid}_i) \lor \bigwedge_{j \in \{1, \ldots, n\}, j \neq i} \text{dis}(\text{pid}_j) \) if multiple parties violated \( \alpha \) (we stress that \( J \) is not obliged to render the more general verdict if multiple parties misbehaved; we merely require that at least one party is rightfully blamed in a run). A verdict of the form \( \text{dis}(\text{pid}_i) \lor \bigwedge_{j \in \{1, \ldots, n\}, j \neq i} \text{dis}(\text{pid}_j) \) by \( J \) is not sufficient to ensure \( C_1^H \), since no individual participant can be blamed; however, \( J \) clearly ensures the weaker constraint \( C_2^H \).

As discussed in the before, in practice it is necessary to ensure individual accountability. Formally, an accountability constraint \((\alpha \Rightarrow \psi_1 | \cdots | \psi_k)\) is said to achieve individual accountability if for every \( i \in \{1, \ldots, k\} \) there exists a party \( p \in \Sigma \) such that \( \psi_i \) implies \( \text{dis}(p) \). Therefore, each minimal verdict \( \psi_i \) determines at least one misbehaving party. In the example above, \( C_1^H \) certainly provides individual accountability, but \( C_2^H \) does not.

A set \( \Phi \) of accountability constraints for the protocol \( Q \) is called an accountability property of \( Q \). Grouping different accountability constraints allows more expressiveness in defining security goals, but for our case study of hashgraph we will consider only a single constraint. Formally, we

\(^1\)This enhances the definition in [14] by allowing dynamic corruption, i.e., corruption at arbitrary points in a run. Note that once a party gets corrupted at some point in a run, we consider it as corrupted for the rest of the run.
4.1 Formal Definition of Accountability

will write $\Pr[Q(1^n) \mapsto \neg(J : \Phi)]$ to denote the probability that the judge $J$ does not ensure $C$, for some $C \in \Phi$, in a run of $Q(1^n)$, in which the probability is taken over the random coins of the run and $1^n$ is the security parameter given to the machine instances. Moreover, we denote by $\Pr[Q(1^n) \mapsto \{(J : \psi) \mid [\omega] \not\models \psi\}]$ the probability that $J$ states a verdict $\psi$ at some point $[\omega]$ in a run of $Q(1^n)$ such that $[\omega] \not\models \psi$, i.e., $J$ renders a false verdict. Intuitively, we would like to demand from $J$ to ensure all accountability constraints in $\Phi$ and never render false verdicts, except for a negligible number of cases; this is captured in the subsequent definition:

**Definition 2 (Accountability).**

Let $Q$ be a system of machine instances with participants $\Sigma$ that includes a judge $J \in \Sigma$, and let $\Phi$ be an accountability property of $Q$. We say that $J$ ensures $\Phi$-accountability for System $Q$ (or $Q$ is $\Phi$-accountable w.r.t. $J$) iff

(i) (fairness) $\Pr[Q(1^n) \mapsto \{(J : \psi) \mid [\omega] \not\models \psi\}]$ is negligible as a function in $\eta^2$, and

(ii) (completeness) $\Pr[Q(1^n) \mapsto \neg(J : \Phi)]$ is negligible as a function in $\eta$.

---

2A function $f : \mathbb{N} \to [0, 1]$ is negligible if, for every $c > 0$, there exists an $\eta_0$ such that $f(\eta) < \frac{1}{\eta^c}$, for all $\eta > \eta_0$.  

37
5 Security and Accountability of Hashgraph

In this chapter, we present our proofs for Hashgraphs’s consistency and accountability properties. We begin with some general discussions about parameters, signatures, and hash collisions. It is convenient to define some general basic correctness conditions that are enforced in a run of the system in order to prove fork-freeness and consistency; we define these conditions in Section 5.1 and prove that Hashgraph is accountable w.r.t. basic correctness. In Section 5.2 we show that Hashgraph is accountable w.r.t. fork-freeness. Then, we present our proof of Hashgraph’s consistency in Section 5.3 and show that Hashgraph is accountable w.r.t. consistency in Section 5.4. Lastly, we have a closing discussion in Section 5.5.

In what follows, we always look at runs of the system \( \{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\} \) for some environment \( \mathcal{E} \) and some adversary \( \mathcal{A} \) and require that the Hashgraph protocol \( \mathcal{P}^H \) has sound parameters. These requirements are summarized in the next definition.

**Definition 3 (Parameters for \( \mathcal{P}^H \)).**

Let \( \omega \) be a run of the system \( \{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\} \) for some environment \( \mathcal{E} \) and some adversary \( \mathcal{A} \), where \( \mathcal{P}^H \) is the Hashgraph protocol as in Definition 1 with parameters

- \( \eta \in \mathbb{N} \) the security parameter,
- a finite set of hashgraph node identities \( \text{nodes} \subseteq \{0, 1\}^* \),
- the parameter \( \text{coinRound} \in \mathbb{N}_2 \) that specifies the coin round interval during virtual voting,
- a polynomial \( p \in \mathbb{Z}[x] \) that limits the runtime of externally provided algorithms,
- an EUF-CMA secure signature scheme \( \Sigma = (\text{gen}(1^\eta), \text{sig}, \text{ver}) \), where \( \text{gen}, \text{sig}, \text{and} \text{ver} \) run for at most \( p(\eta + |x|) \) steps on input \( x \),

and where all internal subprotocols use the same parameters as \( \mathcal{P}^H \).

**Signatures.** We first take a closer look at signatures and define what we understand under a forged signature:

**Definition 4 (Forged Signature).**

Consider a run \( \omega \) as in Definition 3. For \( \text{pid} \in \text{nodes} \), the signature \( \sigma \) is a forgery of message \( \text{msg} \) for \( \text{pid} \) if, at some point \( \lfloor \omega \rfloor \),

\[
\text{verifySig}_{\text{pk}(\text{pid})}(\text{msg}, \sigma) = \text{true}
\]

but \( \text{pid} \) never signed \( \text{msg} \) at \( \mathcal{F}_{\text{cert}} \) up to point \( \lfloor \omega \rfloor \).
Recall from Section 3.2.1 that the adversary cannot simply forge valid signatures of another honest node \( \text{pid} \) (in session \( \text{sid} \)) by calling the entity \( e = (\text{pid}, (\text{pid}, \text{sid}, P^H_{\text{node : node}}, \text{signer}) \) via a corrupted node, due to the specification of \text{AllowAdvMessage}. Therefore, \( \mathcal{A} \) cannot forge valid signatures for \( \text{pid} \) if its signing entity \( e \) is uncorrupted; that is, \( \mathcal{A} \) has not sent the message \text{corruptSigKey} to \( e \) on the network interface beforehand to corrupt the signing key of \( \text{pid} \) in session \( \text{sid} \).

In the following, we often have statements that are only valid because signatures cannot be forged. We will (often) denote such statements at the end with \(^*\).

**Hashes.** The foundation of all subsequent proofs is the fact that hashes can only be forged with negligible probability:

**Lemma 1 (Hash Collisions).**
For a run \( \omega \) as in Definition 3 holds true that \( \mathcal{A} \) can only find two \( x, x' \subseteq \{0,1\}^* \) such that an instance of \( \mathcal{F}_{\omega} \) returns for both \( x \) and \( x' \) the same hash with negligible probability in \( \eta \).

**Proof.** By definition of \( \mathcal{F}_{\omega} \), the random oracle outputs hashes with length \( \eta \). Since the system runs in polynomial time, the probability of finding a collision is negligible in \( \eta \) [10].

The above lemma is pivotal to ensure basic structural conditions on the hashgraph of honest nodes (e.g. self-parent and other-parent of events should be uniquely identifiable). In particular, this is necessary to ensure that honest nodes have consistent hashgraphs (i.e., they share the same subgraph of all events in common). This will be later discussed in Sections 5.3 and 5.4.

Similar to signatures, we often have statements that are generally true only if hash collisions cannot be found. Since hash collision can be found with negligible probability, we will (often) mark such ("mostly true") statements with \(^*\).

### 5.1 Accountability of Hashgraph w.r.t. Basic Correctness

Throughout this chapter we will consider only a single hashgraph instance. Since hashes and signatures cannot be forged (except for negligible probability), one can conclude at the end of this chapter that an honest participant cannot be unrightfully blamed (except for negligible probability) due to interference of another hashgraph session.

**Definition 5 (Basic Graph Correctness).**
Let \( \text{nodes} \) be the set of Hashgraph node identities (as specified as a parameter for \( P^H \)). Let \( G_i \) be the local hashgraph of a node \( \text{pid}_i \in \text{nodes} \). We say that hashgraph \( G_i \) fulfills basic graph correctness if all the below conditions are satisfied:

1. (complete parent-construction) For each event \( E \in G_i \), different to the genesis event, exist events \( E', E'' \in G \) such that
   1. (complete self-ancestor) \( \text{pid} = \text{pid}' \) and \( \text{selfParent} = \text{eventID}' \), and
   2. (complete other-ancestor) \( \text{pid} \neq \text{pid}'' \) and \( \text{otherParent} = \text{eventID}'' \).
2. (signature validity) All events $E \in G_i$, where $E$ is not the genesis event, have a valid signature, i.e., the verification algorithm outputs

$$\text{verifySig}_{pk(pid)}((\text{selfParent}, \text{otherParent}, \text{transactions}, ts), \sigma) = \text{true}.$$ 

3. (hash validity) For all events $E \in G_i$, where $E$ is not the genesis event applies

$$\text{eventID} = \text{hash}((\text{selfParent}, \text{otherParent}, \text{transactions}, ts)).$$

4. (unambiguous root) In $G_i$, exists exactly one event with indegree\(^1 0\), and for each node $pid \in \nodes$ exists at most one event $E'$ such that $\text{selfParent}' = \bot$, i.e., the genesis event is the $\text{selfParent}$ of $E'$.

5. (role compliance) For all $E \in G_i$, where $E$ was signed by $pid$ applies $pid \in \nodes$, i.e., the creator of $E$ is a node participant.

**Definition 6 (Accountability w.r.t. Basic Correctness).**

Consider a run $\omega$ as in Definition 3. Let $J$ be an instance of $\mathcal{F}^H_{\text{judge}}$ and let nodes denote the set of all Hashgraph node identities. Let the witness set $\mathcal{W}$ denote the set of all collected evidences, of form $(\ctr, pid, G(pid))$, by $J$ for all $pid \in \nodes$ until $[\omega]$; formally,

$$\mathcal{W} \subseteq \mathbb{N} \times \nodes \times ((\nodes \times \{0, 1\}^n \times \{0, 1\}^y \times \{0, 1\}^x \times \mathbb{N} \times \{0, 1\}^*) \cup \{ot\})^7).$$

This set stores all reported hashgraphs, with well-formed data (see below), of pid with $\ctr$ depicting the amount of hashgraphs that were previously submitted by any party to $J$. We now define the following three security properties of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}_H\}$:

**$\alpha_1$ (Well-Formed Data).** The security property $\alpha_1$ includes all runs $\omega$ in which any of the succeeding condition is violated:

(i) There exists no event $E \in G(pid)$, different to the genesis event, which was reported to $J$ by $pid$ that is not of form $(pid, \text{eventID}, \text{selfParent}, \text{otherParent}, \text{transactions}, ts, \sigma)$ with $pid \in \nodes$, $\text{eventID}, \text{selfParent}, \text{otherParent} \in \{0, 1\}^n$, $\text{transactions}, \sigma \in \{0, 1\}^*$, and $ts \in \mathbb{N}$.

(ii) No client submitted a state update $((\text{pid}, \text{msglist}), \sigma)$ to $J$ such that $\text{pid} \in \nodes$ and $\text{verifySig}_{pk(pid)}((\text{msglist}, \sigma) = \text{true but msglist is not a subset of } \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^* \times \{0, 1\}^*$.

**$\alpha_2$ (Basic Graph Correctness).** Let $\omega \in \alpha_2$ if $\omega \notin \alpha_1$ and at some point during the run $\omega$ there exists an entry $(\ctr, pid, G(pid)) \in \mathcal{W}$, such that the hashgraph $G$ of $pid$ violates any of the basic graph correctness properties defined in Definition 5 (Therefore, $\alpha_1$ includes all runs in which $\omega \notin \alpha_1$ and basic graph correctness was violated in a hashgraph that was submitted to $J$).

---

\(^1\)In graph theory, \textit{indegree} is the number of edges directed into a vertex in a directed graph. In terms of hashgraph, each event has indegree 2 except for the genesis event $(\bot, \bot, \bot, \bot, \bot, \bot, \bot)$, which has indegree 0.
\(\alpha_3\) (Valid Evidence). We define \(\alpha_3\) to include all runs \(\omega\), where \(\omega \notin \alpha_1 \cup \alpha_2\) and a party \(pid \in \text{nodes}\) submitted a hashgraph that violates the following condition: For each submitted hashgraph \(G\) from \(pid\) to \(J\) applies \(G' \subseteq G\) for all entries \((\text{ctr}, pid, G') \in \mathcal{W}\) (i.e., each submitted hashgraph \(G\) by \(pid\) is a proper superset of each previously submitted hashgraph \(G'\) by \(pid\)).

\(\alpha_4\) (Valid Msglist). Let \(\text{msglist}\) be the message list that was signed by \(pid \in \text{nodes}\) and submitted to \(J\) by a client with a valid signature. The security property \(\alpha_4\) comprises all runs \(\omega\) that are not in \(\bigcup_{i=1}^{3} \alpha_i\) and where at least one of the following conditions is violated:

(i) For each entry \((i, \text{txnId}, \text{eventID}, \text{msg}) \in \text{msglist}\) applies \(\text{txnId} = \text{hash}(\text{msg})\) (i.e., nodes are obligated to compute valid hashes (i.e., transaction IDs) for all messages they include in their \(\text{msglist}\)).

(ii) There exists exactly one entry \((i, \text{txnId}, \text{eventID}, \text{msg}) \in \text{msglist}\) for all \(i = 1, \ldots, |\text{msglist}|\) (i.e., the reported \(\text{msglist}\) from \(pid\) has to be a consecutive sequence with no duplicates or gaps w.r.t. the first parameter \(i\)).

Lastly, we define with \(\alpha = \bigcup_{i=1}^{4} \alpha_i\) the accountability constraint \(C^H_1\) as follows:

\[
C^H_1 := (\alpha \Rightarrow \text{dis}(pid_1) \mid \cdots \mid \text{dis}(pid_n)).
\] (5.1)

As discussed in Example 2, this constraint ensures in a violation of \(\alpha\) that \(J\) blames at least one individual from the set nodes. Further, we set the accountability property \(\Phi_1 := \{C^H_1\}\).

We say \(\mathcal{P}^H\) with parameters as in Definition 3 is individually accountable w.r.t. basic correctness if for all environments \(\mathcal{E}\) and adversaries \(\mathcal{A}\) the system \(\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}\) is \(\Phi_1\)-accountable w.r.t. \(J\).

Lemma 2 (Hashgraph achieves accountability w.r.t. basic correctness).

Consider runs for the system \(\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}\) with parameters for \(\mathcal{P}^H\) as in Definition 3 and let \(J\) be an instance of \(\mathcal{J}^H\). Then, it holds true that \(\mathcal{P}^H\) is individually accountable w.r.t. basic correctness.

Proof. Let \(\mathcal{P}^H\) be the Hashgraph protocol with parameters from the lemma above, and let \(\mathcal{E}\) be an arbitrary environment and \(\mathcal{A}\) be an arbitrary adversary. Towards proving this lemma, we have to show that \(J\) ensures \(\Phi_1\)-accountability for the system \(\mathcal{Q} = \{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}\); we do this by proving fairness and completeness independently. Before doing this, we first discuss some general observations:

Handling of state updates in \(\mathcal{J}^H\) with invalid signatures

In Figure 3.4 nodes sign their current state, the tuple \((\text{pid}, \text{currentState})\), with their own signature key, i.e., nodes calculate the signature \(\sigma = \text{sign}_{\text{sk}(\text{pid})}((\text{pid}, \text{currentState}))\) before sending a state update, \((\text{pid}, \text{currentState})\) together with the signature \(\sigma\), to \(\text{NET}\) (which may be received as parameter \((\text{pid}_{\text{send}}, \text{msglist}), \sigma\) by a machine instance of \(\mathcal{P}^H_{\text{client}}\) (cf. Figure 3.2) or \(\mathcal{J}^H\) (cf. Figure 3.14). The signature ensures that only misbehaving nodes are blamed by \(J\) (\(\ast\)): Upon receiving a state update \((\text{pid}_{\text{send}}, \text{msglist}), \sigma\), \(J\) first checks whether \(\sigma\) is indeed a valid signature of the tuple. If this check fails, \(J\) discards the state update, since \(J\) cannot decide which party, if any, misbehaved. There are multiple scenarios which can lead to this occurrence of invalid signatures:
5.1 Accountability of Hashgraph w.r.t. Basic Correctness

1. \( \text{pid}_{\text{send}} \) is dishonest and included an invalid signature in its state update that was then reported to \( J \) by a dishonest client (By implementation of \( \mathcal{P}^H_{\text{client}} \), honest clients only submit state updates to an instance of \( \mathcal{F}^H_{\text{judge}} \) iff the signature contained in the state update is valid, i.e., \( \text{verifySig}_{\text{pk}}((\text{pid}_{\text{send}}, \text{currentState}), \sigma) \) evaluates to \( \text{true} \)).

2. \( \text{pid}_{\text{send}} \) is honest but a dishonest client reported malformed evidence (e.g., an invalid signature or message list) to \( J \) such that \( \text{verifySig}_{\text{pk}}((\text{pid}_{\text{send}}, \text{currentState}), \sigma) = \text{false} \).

This is not a complete enumeration of all possible scenarios (for instance, the attacker \( \mathcal{A} \) can also modify a state update, since all communication between clients and nodes is via network interfaces), however, both scenarios illustrate that \( \text{pid} \) can be both honest and dishonest. Thus, to achieve fairness, \( J \) does not store any verdicts for violations w.r.t. a message list from a state update containing an invalid signature.

Verdicts of \( J \) w.r.t. basic correctness

Notice that all verdicts of \( J \) w.r.t. basic correctness (see Definition 6) are of form \( \text{dis}(\text{pid}_{\text{call}}) \), \( \text{dis}(\text{pid}_{\text{send}}) \), or \( \text{dis}(\text{pid}) \), where \( \text{pid}_{\text{call}}, \text{pid}_{\text{send}}, \) and \( \text{pid} \) has to be party IDs included in the set nodes. Clearly, these verdicts achieve individual accountability for the accountability constraint \( \mathcal{C}^H \).

Fairness. To prove fairness, one has to show that \( \Pr[Q(1^n) \mapsto \{(J : \psi) \mid [\omega] \neq \psi \}] \) is negligible as a function in \( \eta \). We do this by showing that \( J \) outputs a verdict \( \psi \) in \( \omega \) which evaluates false at point \( [\omega] \) (written \( [\omega] \neq \psi \)) in at most negligible amount of runs \( \omega \in \alpha \). Subsequently, we show for each run in \( \alpha_i \), \( i \in \{1, 2, 3, 4\} \), that \( J \) never outputs a wrong verdict (except for negligible amount of runs), and thus conclude that \( J \) never outputs a false verdict w.r.t. \( \alpha \) (except for a negligible amount of runs)\(^2\); this then infers fairness of \( J \). By definition, the judge \( J \) checks all properties \( \alpha_i \), for \( i \in \{1, \ldots, 4\} \). So, to fulfill fairness, we have to show that honest nodes never report hashgraphs to \( \mathcal{F}^H_{\text{judge}} \) and send state updates to \( \text{NET} \) that violate basic correctness conditions.

\( \alpha_1 \) (Well-Formed Data). We have to show that honest nodes always provide well-formed data, their local hashgraph, to \( \mathcal{F}^H_{\text{judge}} \). Further, message lists in state updates submitted by clients (which they received from nodes via the network interface) must suffice a particular format.

(i) One can observe in Figure 3.4 honest nodes sending their internal hashgraph (the state variable events) to \( \mathcal{F}^H_{\text{judge}} \). Since the variable events already has the required format (cf. Figure 3.12), no honest node will be blamed by \( J \).

(ii) Figure 3.4 illustrates the computations of an honest node, say \( \text{pid} \), before sending a state update to \( \text{NET} \) (and therefore possibly to \( J \)): \( \text{pid} \) extracts all entries of form \( (\text{counter}, \text{txId}, \text{eventId}, \text{msg}) \in \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \) with \( \text{counter} \neq \varepsilon \) from the local state variable msglist and adds it to the local set currentState. Before nodes send their current state to \( \text{NET} \), nodes sign the tuple \( (\text{pid}, \text{currentState}) \) with their own signature key, i.e., nodes calculate the signature \( \sigma = \text{sign}_{\text{sk}(\text{pid})}((\text{pid}, \text{currentState})) \). Finally, \( \text{pid} \) sends the state update, \( (\text{pid}, \text{currentState}) \) together with the signature \( \sigma \), to \( \text{NET} \) (which may be received as parameter msglist by \( J \) in Figure 3.14). Consequently, msglist = currentState is also a

\(^2\)This holds true because negligible functions are closed under addition, i.e., for two negligible function \( f, g : \mathbb{N} \rightarrow [0, 1] \) is also the function \( x \mapsto f(x) + g(x) \) negligible.
subset of $\mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n$. In case of interference from a malicious client, the signature of the state update is invalid, i.e. $\text{verifySig}_{pk}(\text{pid}, \text{currentState}, \sigma)$ evaluates to $\text{false}$ ($\ast$). As mentioned above, $J$ discards all state updates with invalid signatures. In both cases, $J$ outputs no verdicts.

$\alpha_2$ (Basic Graph Correctness). Let $W$ denote the set of all collected evidences by participants up to point $[\omega]$, as in Definition 6. One can observe in Figure 3.13 $J$ stating the verdicts $\text{dis}([\text{pid}_\text{call}])$ for the latest submitted hashgraph of $\text{pid}_\text{call}$ for misbehavior defined in basic graph correctness, Definition 5. Therefore, it suffices to show that honest nodes only submit hashgraphs that comply with the conditions for basic graph correctness. We show this result with proof by induction:

Base case
For the base case, we observe in the implementation of $\mathcal{P}_\text{node}$ that the local hashgraph of an honest node is after initialization equal to $\{(\bot, \bot, \bot, \bot, \bot, \bot, \bot)\}$. This set clearly fulfills all conditions for basic graph correctness.

Induction step
For the induction step, we assume the local hashgraph, the set of events in $\mathcal{P}_\text{node}$, of an honest node $\text{pid}$ satisfies all conditions for basic graph correctness. Honest nodes only include new events into their local hashgraph during a hashgraph sync, i.e., the process of one node sending a gossip event together with its local hashgraph to another node. Therefore, we only need to consider the circumstance where an honest node merges another hashgraph into its own. After the merging however, a node creates a new event, containing newly submitted transactions, which also has to satisfy basic graph correctness (particularly complete parent-construction). Upon receiving a gossip event (more precisely the event ID of the gossiped event) $\text{gossipedEvent}$ together with the hashgraph events over the network interface, $\text{pid}$ first does the same basic correctness checks for events as $J$ does with the submitted hashgraphs of $\text{pid}$. In particular, $J$ checks if all events contain valid signatures of valid nodes (cf. Definition 5 role compliance). If any condition in this check is violated, $\text{pid}$ does not merge the received hashgraph into its own and thus events remains unchanged. If the previous check succeeds, $\text{pid}$ performs some additional checks to ensure $\text{gossipedEvent}$ is indeed in the gossiped hashgraph events, the creator of $\text{gossipedEvent}$ is not $\text{pid}$ itself, and $\text{gossipedEvent}$ is the last event created by the initiator of the sync. If all aforementioned checks succeed, $\text{pid}$ merges the gossiped hashgraph events into his own hashgraph (events). We denote this newly merged hashgraphs as events’, which clearly inherits the basic graph correctness properties from events and events. Moreover, $\text{pid}$ creates a new event (cf. $\text{eventID}_{\text{new}}$ in Figure 3.4) with valid signature and hash, with the self-parent being the event ID of the last event created by $\text{pid}$ and the other-parent being the gossiped event. This newly created event ensures all conditions for basic graph correctness – particularly complete parent-construction. Lastly, this newly created event is added to the local hashgraph of $\text{pid}$. We conclude, events’ still satisfies basic graph correctness ($\ast\ast$). This completes the induction step.

$\alpha_3$ (Valid Evidence). We have to show for $\alpha_3$ that an honest node, say $\text{pid}$, reports hashgraphs to $J$ that are always proper supersets of all previously submitted hashgraphs by $\text{pid}$.

---

This last property is not necessary in order to prove fairness for $J$, but it is still a desirable property we enforce for honest nodes.
5.1 Accountability of Hashgraph w.r.t. Basic Correctness

By definition of $\mathcal{P}_{\text{node}}^H$, honest nodes always create a new event (with event ID $\text{eventID}_{\text{new}}$), which is added to the local hashgraph events, before reporting their current hashgraph to $\mathcal{P}_{\text{judge}}^H$ at end of a sync. Thus, honest nodes are never blamed by $\mathcal{J}$.

**$\alpha_4$ (Valid Msglist).** It has to be shown that the message list extracted from a valid state update $((\text{pid}_{\text{send}}, \text{msglist}), \sigma)$ (i) contains valid hashes (the transaction IDs) for all messages and (ii) $\text{msglist}$ is a consecutive sequence with no duplicates or gaps. As discussed before, $\mathcal{J}$ discards all state updates from clients where the signature of $(\text{pid}_{\text{send}}, \text{msglist})$ cannot be verified since $\mathcal{J}$ cannot judge if $\text{pid}_{\text{send}}$ actually misbehaved. Therefore, it remains to prove that honest nodes obey conditions (i) and (ii) before sending their state update to $\text{NET}$.

(i) One can observe in the implementation of $\mathcal{P}_{\text{node}}^H$ two occurrences where nodes add new messages to their message list (and therefore compute the corresponding transaction IDs): First, nodes may receive a new message $\text{msg}$ on their network interface; then, nodes simply calculate the hash $\text{txId}$ for this message and add the entry $(\varepsilon, \text{txId}, \varepsilon, \text{msg})$ to their message list. Secondly, nodes can receive new events via a hashgraph sync. In this case, if nodes accept the sync, they calculate the transaction IDs for all messages contained in newly received events and add a new entry for each message to their message list. In both cases, we see that nodes compute the hashes for new messages themself and do not rely on transaction IDs from third parties (e.g., other nodes or clients). Therefore, all hashes of messages in message lists from honest nodes are always guaranteed to be valid.

(ii) We observe that honest nodes include in their message list for a state update (cf. $\text{currentState}$ in Figure 3.4) all entries $(\text{ctr}, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$, where $\text{ctr} \neq \varepsilon$ (we call such entries in what follows ordered entries). Consequently, it must be shown that at any time ordered entries in $\text{msglist}$ form a consecutive sequence with no duplicates w.r.t. $\text{ctr}$. By definition of $\mathcal{P}_{\text{node}}^H$, nodes add ordered entries only at the end of the function $\text{findOrder}$ to their message list. Before this occurs, all so far ordered entries $(\text{ctr}, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$ are replaced with (unordered) entries $(\varepsilon, \text{txId}, \text{eventID}, \text{msg})$. Thereafter, nodes iterate over all (now unordered entries $(\varepsilon, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$, where the event with ID $\text{eventID}$ has been assigned a consensus position in the total order of events, and add the ordered entry $(i, \text{txId}, \text{eventID}, \text{msg})$ to $\text{msglist}$. Thereby, $i = 1$ at the beginning, and $i$ is increased by one each time an entry is added to $\text{msglist}$. Clearly, the resulting message list is a consecutive sequence of ordered entries with no gaps and duplicates.

We conclude, $\mathcal{J}$ is fair in all but negligible amount of runs of the system $\mathcal{Q}$.

**Completeness.** To prove completeness, we have to show that $\Pr[\mathcal{Q}(1^\eta) \rightarrow \neg(J : \Phi_1)]$ is negligible as a function in $\eta$; that is, $\mathcal{J}$ must ensure the accountability constraint $C_1^H$ in all but a negligible number of runs $\omega$ of the system $\mathcal{P}^H$, $\mathcal{A}$, $\mathcal{E}$ with parameters for $\mathcal{P}^H$ as in Definition 3. Particularly, if $\omega$ is contained within the security property $\alpha$, $\mathcal{J}$ has to state a verdict $\psi$ that implies at least one of the verdicts $\text{dis}(\text{pid}_1), \ldots, \text{dis}(\text{pid}_n)$ defined by $C_1^H$ (cf. Example 2). As already argued above, all verdicts of $\mathcal{J}$ are of form $\bigwedge_{i \in I} \text{dis}(\text{pid}_i)$, for $\emptyset \neq I \subseteq \{1, \ldots, n\}$; hence, $\mathcal{J}$ achieves individual accountability because each verdict rendered by $\mathcal{J}$ implies at minimum one of the verdicts defined by the constraint $C_1^H$. In order to prove fairness, we already argued that by definition of $\mathcal{P}_{\text{judge}}^H$, the judge $\mathcal{J}$ checks all properties $\alpha_i$ for $i \in \{1, \ldots, 4\}$, i.e., $\mathcal{J}$ states a verdict if any of the conditions for basic correctness is violated during a run $\omega \in \alpha$. This can be seen in Figures 3.13 and 3.14.
We conclude this lemma that for a run $\omega \in \alpha$ the judge $J$ never renders a wrong verdict for $\omega$ except for a negligible amount of runs, and $J$ always states a verdict for a violation of $\alpha$. Therefore, we have $\Pr[Q(1^\eta) \mapsto \{ (J : \psi) | |\omega| \neq \psi \}]$ is negligible as a function in $\eta$, and $\Pr[Q(1^\eta) \mapsto \neg (J : \Phi_1)] = 0$. This concludes the proof for this lemma.

5.2 Accountability of Hashgraph w.r.t. Fork-Freeness

Definition 7 (Fork-Freeness).
Consider a run $\omega$ as in Definition 3, where $\omega \notin \alpha$. Let nodes be the set of Hashgraph node identities (as specified as a parameter for $\mathcal{P}^H$), and assume $J$ is an instance of $\mathcal{F}^H_{\text{judge}}$. Let $|\omega|$ denote one point somewhere in the run $\omega$, and let the witness set $W$ denote the set of all collected evidences by $J$ of form $(i, pid, G)$ where $G$, submitted by $pid$, is the $i$-th hashgraph that was reported to $J$ by any node prior to $|\omega|$ (see Definition 6). We define

$$E_W := \bigcup_{(i, pid, G) \in W} G$$

to be the set of all so far reported events by any node.

Fork-Freeness. We say $pid \in \text{nodes}$ satisfies fork-freeness at point $|\omega|$ if for all pairs of events $E_1, E_2 \in E_W$ signed by $pid$ it holds true that selfParent$_1 = \text{selfParent}_2$ implies eventID$_1 = \text{eventID}_2$.

The system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}$ satisfies fork-freeness in the run $\omega$ if there is no $pid \in \text{nodes}$ that violates fork-freeness at some point during the run $\omega$.

Definition 8 (Accountability w.r.t. Fork-Freeness).
Consider a run $\omega$ as in Definition 3. Let nodes be the set of all Hashgraph node identities, and assume $J$ is an instance of $\mathcal{F}^H_{\text{judge}}$. Let $W$ denote the witness set (i.e., the set of all collected evidences of nodes, see Definition 6).

$\beta_1$ (Fork-Freeness). This security property contains all runs, where $\omega \notin \alpha$ and there exists a node $pid \in \text{nodes}$ such that $pid$ does not satisfy fork-freeness at some point $|\omega|$.

We set $\beta = \beta_1 \cup \alpha$, and define the accountability constraint $C^H_2$ as follows:

$$C^H_2 := (\beta \Rightarrow \text{dis}(pid_1) | \cdots | \text{dis}(pid_n)) \tag{5.2}$$

This accountability constraint also ensures individually accountability. Finally, we set the accountability property $\Phi_2 := \{C^H_2\}$.

We call $\mathcal{P}^H$ with parameters as in Definition 3 individually accountable w.r.t. fork-freeness if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$ the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}$ is $\Phi_2$-accountable w.r.t. $J$.

Theorem 1 (Hashgraph achieves accountability w.r.t. fork-freeness).
Consider runs for the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}$ with parameters for $\mathcal{P}^H$ as in Definition 3 and let $J$ be an instance of $\mathcal{F}^H_{\text{judge}}$. Then, it holds true that $\mathcal{P}^H$ is individually accountable w.r.t. fork-freeness.
In what follows, we present our proof for Theorem 1, i.e., we demonstrate that Hashgraph does achieve individual accountability w.r.t. fork-freeness for runs as in Definition 3.

**Proof.** Consider a run \( \omega \) as in Definition 3, where \( \mathcal{E} \) is an arbitrary environment and \( \mathcal{A} \) is an arbitrary adversary. Further, let \( \Phi_2 \) be the accountability constraint from Definition 8. We show that the system \( Q = \{ P^H, \mathcal{A}, \mathcal{E} \} \) in the run \( \omega \) satisfies \( \Phi_2 \)-accountability, i.e., \( \{ P^H, \mathcal{A}, \mathcal{E} \} \) fulfills both fairness and completeness.

**Fairness.** In order to prove fairness, we have to show that \( \Pr[Q(1^n) \mapsto \{(J : \psi) \mid [\omega] \not\models \psi\}] \) is negligible as a function in \( \eta \). That is, we have to show that the judge \( J \) renders verdicts \( \psi \) that evaluate to false at point \( [\omega] \) (written \( [\omega] \not\models \psi \)) in at most negligible amount of runs of the system \( \{ P^H, \mathcal{A}, \mathcal{E} \} \) with parameters as in Definition 3. We have already shown that \( J \) is fair for verdicts w.r.t. basic correctness \( (\alpha) \), i.e., if \( \omega \notin \alpha \), \( J \) will not render verdicts for honest nodes except for negligible probability in \( \eta \). Therefore, it is left to prove that \( J \)'s verdicts w.r.t. \( \beta_1 \) are also fair:

First, one can observe in Figure 3.13 that \( J \) does the exact same checks as specified in the definition of fork-freeness before it outputs a verdict, blaming some participant: Initially, \( J \) calculates \( E_W \), the set of all so far reported events by any node, then \( J \) checks if there exists a pair of two events \( E_1, E_2 \in E_W \) created by \( \text{pid} \) for which applies \( \text{selfParent}_1 = \text{selfParent}_2 \) and \( \text{eventId}_1 \neq \text{eventId}_2 \). Thereafter, \( J \) ensures that the signatures \( \sigma_1, \sigma_2 \) for \( E_1 \) and \( E_2 \) are indeed valid, i.e., \( J \) verifies that \( \text{verifySig}_{\text{pk}(\text{pid})}(\{\text{selfParent}_1, \text{otherParent}_1, \text{txs}_1, \text{ts}_1\}, \sigma_1) \) outputs true for \( i \in \{1, 2\} \). If this check also succeeds, \( J \) states the verdict \( \text{dis}(\text{pid}) \) for node \( \text{pid} \). It remains to be shown that \( \text{pid} \) is indeed dishonest. Since \( J \) verifies the signature of two forking events \( E_1, E_2 \) created by \( \text{pid} \), node \( \text{pid} \) must have created and signed both events \((*)\). Thus, for \( J \) to be fair, honest nodes must not create forked events. By definition of \( P^H_{\text{node}} \), nodes create new events only when they receive and accept a sync from another node. Let \( E_{\text{new}} \) be such a new event created by \( \text{pid} \) (i.e., \( \text{pid} \) signed it) with self-parent being the last event that \( \text{pid} \) created before \( E_{\text{new}} \), or the genesis event in case \( E_{\text{new}} \) is the first event that \( \text{pid} \) created. Therefore, each new event created by \( \text{pid} \) has a different self-parent \((*)\); thus, honest nodes never fork \((*)\). We conclude that \( J \) is fair, since \( J \) never renders verdicts, blaming honest nodes, w.r.t. fork-freeness \((**)\); hence, \( \Pr[Q(1^n) \mapsto \{(J : \psi) \mid [\omega] \not\models \psi\}] \) is negligible in \( \eta \).

**Completeness.** To prove completeness, one has to show that \( \Pr[Q(1^n) \mapsto \neg((J : \Phi_2))] \) is negligible as a function in \( \eta \), i.e., \( J \) ensures the accountability constraint \( C^H_2 \) in all but a negligible number of runs \( \omega \) of the system \( \{ P^H, \mathcal{A}, \mathcal{E} \} \) with parameters as in Definition 3. Particularly, if the run \( \omega \) is contained within the security property \( \beta_1 \cup \alpha \), \( J \) has to state a verdict \( \psi \) that implies at least one of the verdicts \( \text{dis}(\text{pid}_1), \ldots, \text{dis}(\text{pid}_n) \) defined by \( C^H_2 \). First, observe that all verdicts of \( J \) are of form \( \bigwedge_{i \in I} \text{dis}(\text{pid}_i) \), for \( \emptyset \neq I \subset \{1, \ldots, n\} \) (notice that \( J \) can state multiple verdicts for different parties); consequently, each such verdict clearly achieves individual accountability for \( C^H_2 \) as each verdict implies at minimum one of the verdicts defined by the constraint \( C^H_2 \). Since we have already shown \( J \) fulfilling completeness for basic correctness (i.e., the run \( \omega \) is in \( \alpha \)), it remains to prove that \( J \) is complete w.r.t. fork-freeness (i.e., \( \omega \in \beta_1 \)).

In the following, let \( \omega \) be a run of \( \beta_1 \); therefore, there exists at some point \( [\omega] \) in the set \( E_W = \{ E \in G \mid (\text{ctr}, \text{pid}, G) \in W \} \) a pair of events that are both signed by some node \( \text{pid} \), such that \( \text{pid} \) does not satisfy fork-freeness. More specifically, there are two events \( E_1, E_2 \in E_W \) with \( \text{verifySig}_{\text{pk}(\text{pid})}(\{\text{selfParent}_1, \text{otherParent}_1, \text{txs}_1, \text{ts}_1\}, \sigma_1) = \text{true} \) for \( i \in \{1, 2\} \), so that
We have shown that \( \textit{eventID}_1 \neq \textit{eventID}_2 \), i.e., \( \textit{pid} \) created a fork at \( \textit{selfParent}_1 \) with \( \textit{eventID}_1 \) and \( \textit{eventID}_2 \). One can observe in Figure 3.13 that the judge, indeed, outputs a verdict \( \text{dis} (\text{pid}) \) for this violation. We conclude that \( J \) always renders a verdict in \( \beta_1 \), i.e., \( \text{Pr}[\tilde{Q}(1^y) \neg (J : \Phi_2)] = 0 \).

We have shown that \( J \) is fair and complete in all runs of \( \{P^H, A, E\} \) with parameters as in Definition 3 (**). This concludes the proof for Theorem 1.

### 5.3 Consistency of the Hashgraph Algorithm

**Definition 9 (Consistent Hashgraphs).**

Let \( G \) be a hashgraph and \( E \in G \). Let \( G[E] \subseteq G \) denote the subgraph of \( E \) in \( G \) that contains \( E \) and all ancestors of \( E \). We say two hashgraphs \( G \) and \( G' \) are consistent if for all \( E \in G \cap G' \) follows \( G[E] = G'[E] \).

**Lemma 3.**

Let \( G, G' \) be two hashgraphs, both satisfying basic graph correctness. Then, it holds true that \( G \) and \( G' \) are consistent.

**Proof.** We first prove for an arbitrary event \( E \in G \cap G' \), different to the genesis event, that both its parent events are in \( G \cap G' \), and there exist no other events in \( G \) or \( G' \) with the same event ID as one of \( E \)'s parents events except for negligible probability in \( \eta \). Because \( G, G' \) satisfy basic graph correctness, there must be self-parents \( E'_1 \in G, E'_2 \in G' \) with \( \text{pid} = \text{pid}'_1 \), \( \text{pid} = \text{pid}'_2 \) and \( \text{selfParent} = \text{eventID}'_1 \), \( \text{selfParent} = \text{eventID}'_2 \), and other-parents \( E''_1 \in G, E''_2 \in G' \) where \( \text{pid} \neq \text{pid}'_1, \text{pid} \neq \text{pid}'_2 \) and \( \text{otherParent} = \text{eventID}'_1 \), \( \text{otherParent} = \text{eventID}'_2 \). Furthermore, hash collisions cannot be found except for negligible probability in \( \eta \); therefore, there cannot be two events in \( G \cup G' \) with the same event ID (**); thus, \( E'_1 = E'_2, E''_1 = E''_2 \) and self-parent and other-parent of \( E \) are unique in \( G \) and \( G' \) (**).

Recall from Definition 9 that \( G, G' \) are consistent if for all \( E \in G \cap G' \) follows \( G[E] = G'[E] \). We know the genesis event is in \( G \) and \( G' \), and for each other event \( E \in G \cap G' \) there is exactly one self-parent event and exactly one other-parent event in \( G \cap G' \). By this, we conclude \( G[E] = G'[E] \) (**).

In the following, we will usually demand for two hashgraphs \( G, G' \) that both fulfill basic graph correctness and follow implicitly with the above lemma that \( G, G' \) are also consistent.

**Lemma 4.**

Let \( G \) and \( G' \) be two hashgraphs, both satisfying basic graph correctness. Then, it holds true that the \textit{divideRounds} algorithm assigns for all \( E \in G \cap G' \) the same round created number and witness status, when run on input \( G \) or \( G' \).

**Proof.** Let \( E \in G \cap G' \) be arbitrary. Because \( G \) and \( G' \) are also consistent, by Definition 9 applies \( G[E] = G'[E] \), i.e., both hashgraphs share the same subgraph of \( E \). We show this lemma with proof by induction over the subgraph \( G[E] \).
5.3 Consistency of the Hashgraph Algorithm

Base case
Initially, the genesis event receives round created number 1 and is assigned to be the sole witness of round 1. Additionally, each event \( E \) containing the genesis event as self-parent receives round number 2 and is assigned to be a witness of this round; basic graph correctness ensures that there is at most one such event for each creator of \( E \).

Induction step
Assume the induction hypothesis that for an event \( E_0 \in G[E] = G'[E] \) it is true that \( \text{divideRounds} \) assigned for all ancestors of \( E_0 \), in particular the self-parent \( E_1 \) and other-parent \( E_2 \), the same round created number when run on hashgraph \( G \) or \( G' \). Then, with input \( G \) and \( G' \), \( \text{divideRounds} \) will assign \( E_0 \)'s round \( r \) to be the maximum of the rounds of \( E_1 \) and \( E_2 \), and if \( E_0 \) strongly sees more than \( \frac{2}{3}n \) round \( r \) witnesses, it will instead assign \( r + 1 \) as round created number of \( E_0 \) and mark \( E_0 \) as witness of round \( r + 1 \).

Therefore, \( \text{divideRounds} \) assigns on input \( G \) and \( G' \) for each event in \( G[E] \), including \( E \), the same round and witness status.

Corollary 1 (Consistent Witnesses).
Let \( G, G' \) be two hashgraphs, both satisfying basic graph correctness, with \( G \subseteq G' \). Then it holds true that \( \mathcal{W}_r \subseteq \mathcal{W}'_r \), i.e., if \( E_r \in \mathcal{W}_r \) is a round \( r \) witness in \( G \), then \( E_r \) is also a round \( r \) witness in \( G' \).

\textbf{Proof.} By Lemma 4, for all \( E_r \in G \cap G' \) follows \( E_r \in \mathcal{W}_r, \mathcal{W}'_r \) or \( E_r \notin \mathcal{W}_r, \mathcal{W}'_r \); thus, with \( G \subseteq G' \) holds true that \( \mathcal{W}_r \subseteq \mathcal{W}'_r \).

Lemma 5.
Let \( G \) be a hashgraph fulfilling basic graph correctness, and let \( \mathcal{W}_r \) denote the set of all round \( r \) witnesses in \( G \). If there exists an event \( E_r \in \mathcal{W}_r \) whose fame is decided (by running the algorithm \( \text{decideFame} \)) by a witness \( E_{r+1+i} \in \mathcal{W}_{r+i+1} \) in round \( r + i + 1 \) \((i \geq 1)\), then it holds true for all \( k \in \{1, \ldots, i\} \) that there exist more than \( \frac{2}{3}n \) witnesses in \( \mathcal{W}_{r+k} \).

\textbf{Proof.} Let \( E_r \in \mathcal{W}_r \) be an event whose fame has been decided during \( \text{decideFame} \). Therefore, virtual voting of \( E_r \) must have at least reached the second round \( r + 2 \), otherwise \( E_r \)'s fame could not have been decided (witnesses in the first round do not decide the fame for the candidate); hence, \( i \geq 1 \) for witness \( E_{r+i+1} \) which decides the fame for \( E_r \). Suppose \( E_{r+i+1} \in \mathcal{W}_{r+i+1} \) is a witness that decides the famousness of \( E_r \). One can observe in \( \text{decideFame} \) that \( E_{r+i+1} \)'s decision depends solely on the round \( r + i \) witnesses \( E_{r+i}^1, \ldots, E_{r+i}^{m_i} \); it strongly sees, where \( m > \frac{2}{3}n \) by the definition of witnesses. If \( i = 1 \), then \(|\mathcal{W}_{r+1}| \geq m > \frac{2}{3}n \) and the above statement holds true.

Suppose \( i > 1 \). A witness \( E_{r+k+1} \in \mathcal{W}_{r+k+1} \) participates in the virtual voting for \( E_r \), if \( k = i \), or \( k \geq 1 \) and there exist \( l = i - k \) witnesses \( E_{r+k+2}, \ldots, E_{r+i+1} \in \mathcal{W}_{r+i+1} \) such that \( E_{r+p+1} \) strongly sees \( E_{r+p} \) for all \( p \in \{k + 2, \ldots, i\} \), and \( E_{r+k+2} \) strongly sees \( E_{r+k+1} \). A participating witness \( E_{r+k+1} \) strongly sees at least \( m_k > \frac{2}{3}n \) witnesses \( E_{r+k}^1, \ldots, E_{r+k}^{m_k} \in \mathcal{W}_{r+k} \) in round \( r + k \), which also participate in the election if \( k \geq 2 \). Since \( E_{r+k+i} \) participates in the election, we inductively conclude that there must exist at least \( m_k > \frac{2}{3}n \) witnesses \( E_{r+k}^1, \ldots, E_{r+k}^{m_k} \in \mathcal{W}_{r+k} \) in round \( r + k \) for all \( k \in \{1, \ldots, i\} \); thus, \(|\mathcal{W}_{r+k}| \geq m_k > \frac{2}{3}n \).
Lemma 6 (Consistent Voting).
Let $G, G'$ be two hashgraphs such that $G \subseteq G'$ and both satisfy basic graph correctness. Then, for all $E_r \in W_r$ and $E_{r+i} \in W_{r+i}$ with $vote_G(E_r, E_{r+i}, \beta)$ (where $\beta \in \{true, false\}$) holds true that $vote_G'(E_r, E_{r+i}, \beta)$, if there is no $E_{r'+i+1} \in W_{r'+i+1}$ with $decision_G'(E_r, E_{r'+i+1}, \beta')$ that votes before $E_{r+i}$ in $G'$. Moreover, if additionally applies $decision_G(E_r, E_{r+i}, \beta)$, then it holds true that $decision_G'(E_r, E_{r+i}, \beta)$ if there exists no witnesses that ends $E_r$'s election (see above) before $E_{r+i}$ can vote.

Proof. Observe that voting (and deciding the fame for some witness) is purely a function on the ancestors of voting (deciding) witnesses. Because of $G \subseteq G'$, the ancestors for events in $G$, clearly, do not change in $G'$ (note that $G \subseteq G'$ implies that $G, G'$ are consistent hashgraphs). Thus, events will vote (decide) identically for $E_r$ in $G'$ if the election for $E_r$ is not ended beforehand by some new witness $E_{r'+i+1} \in W_{r'+i+1} \setminus W_{r'+i+1}$.

The purpose of strongly seeing is to ensure that nodes with different hashgraphs achieve consistent voting results. We prove in the following lemma that this holds true as long as more than $\frac{2}{3}n$ nodes do not create forks. This result is the foundation for Hashgraph to achieve consistency, and will be used frequently for the rest of the remaining proofs.

Lemma 7 (Strongly Seeing Lemma).
Let $G, G'$ be two hashgraphs with $G \subseteq G'$ and both fulfilling basic graph correctness. Suppose $G'$ contains less than $\frac{1}{4}n$ forking nodes. Let the pair of events $(E_1, E_2)$ be a fork with self-parent $E_0$ in $G$. If $E_1$ is strongly seen by event $E$ in $G$, then $E_2$ will not be strongly seen by all events in $G'$ (cf. [2]^4).

Proof. We prove that no event in $G'$ can strongly see $E_2$. Assume $E_1$ is strongly seen by some event $E \in G$. By the definition of strongly seeing, there must exist a set of events $S_1 \subseteq G$ such that $|S_1| > \frac{2}{3}n$, all events in $S_1$ have pairwise different creators, and for all $E_s \in S_1$ applies $E$ sees $E_s$ and $E_s$ sees $E_1$. Let $\mathcal{N}_1 = \{creator_G(E_s) \mid E_s \in S_1\}$ $\subseteq$ nodes be the set of all creators of events in $S_1$, and let $\mathcal{N}_1^H \subseteq \mathcal{N}_1$ denote the set of all honest nodes (i.e., nodes without forks) in $\mathcal{N}_1$. Because of the assumption of more than $\frac{1}{4}nforking nodes, it holds true that $|\mathcal{N}_1^H| > \frac{2}{3}n$.

Let $E' \in G'$ be an arbitrary event; we prove that $E'$ cannot strongly see $E_2$ in $G'$. For the purpose of contradiction, let $S_2 \subseteq G'$ be a set of events such that $|S_2| > \frac{2}{3}n$, all events in $S_2$ have pairwise different creators, and for all events $E_s \in S_2$ holds true that $E'$ sees $E_s$ and $E_s$ sees $E_2$. We show that this set cannot exist. Again, let $\mathcal{N}_2$ be the set of all creators of events in $S_2$, where $|\mathcal{N}_2| > \frac{2}{3}n$ by the definition of strongly seeing. Then, we can conclude $|\mathcal{N}_2 \cap \mathcal{N}_1^H| \geq 1$. Therefore, there must be an honest node $pid \in \mathcal{N}_2 \cap \mathcal{N}_1^H$ with no forks in $G$ and $G'$. Moreover, there must be events $E_1^H \in S_1, E_2^H \in S_2$ created by $pid$ such that $E_1^H$ sees $E_1$ and $E_2^H$ sees $E_2$. However, this is not possible: Due to $G \subseteq G'$ and $E_1^H$ seeing $E_1$ in $G$, $E_1^H$ can also see $E_1'$ in $G'$. By the definition of seeing, the subgraph $G[E_1^H] = G'[E_1^H]$ does not contain a fork of $E_1'$; thus $E_2 \notin G'[E_1^H]$. Therefore, $E_1^H \neq E_2^H$ and $E_1^H$ is an self-ancestor of $E_2^H$. But then is also $E_1$ an ancestor of $E_2^H$. So both $E_1$ and $E_2$ are ancestors of $E_2^H$; hence, $E_2^H$ cannot see either of two the events. This is a contradiction to $E_2^H \in S_2$. Therefore, $S_2$ does not exist and $E_2$ is not strongly seen in $G'$.

The author of this paper proved a slightly stronger statement: Instead of assuming $G \subseteq G'$, Leemon Baird proved this lemma with the more strict requirement of $G, G'$ being consistent hashgraphs.
We proved the above lemma for all hashgraphs \( G, G' \) that satisfy basic graph correctness, where \( G \subseteq G' \). However, we cannot guarantee that hashgraphs of different nodes are always subsets of each other, but rather expect honest nodes to have consistent hashgraphs. We will later demonstrate in Theorem 2 that our assumption of \( G \subseteq G' \) is sufficient, and we do not have to prove Lemma 7 under the more strict condition of \( G, G' \) being consistent hashgraphs.

**Corollary 2.**
Let \( G \) be a hashgraph satisfying basic graph correctness and suppose less than \( \frac{1}{3}n \) forking nodes in \( G \). Let \( W \subseteq W_r \) be the set of round \( r \) witnesses that are strongly seen by events in \( G \). Then it holds true for all pairs of events \((E_r, E'_r)\) in \( W \) that \( \text{creator}_G(E_r) \neq \text{creator}_G(E'_r) \).

**Proof.** By definition, a witness of round \( r \) is the first event created by a node, say \( \text{pid} \), in this round. In the absence of forking, there is at most one witness created by \( \text{pid} \) in round \( r \). If \( \text{pid} \) created a fork, we know by Lemma 7 that at most one event of the forked round \( r \) witnesses of \( \text{pid} \) is strongly seen in \( G \). Hence, \( W \) contains only round \( r \) witnesses of different creators.

**Corollary 3.**
Let \( G \) be a hashgraph satisfying basic graph correctness and suppose less than \( \frac{1}{3}n \) forking nodes in \( G \). Then it holds true that all witnesses in \( W_{r+1} \) can strongly see at most \( n \) witnesses of round \( r \). In particular, there exist at most \( n \) witnesses in \( W_r \) that are strongly seen by witnesses in \( W_{r+1} \).

**Proof.** Follows immediately by Lemma 7 and Corollary 2.

**Lemma 8 (Consistent Famous Witnesses).**
Let \( G, G' \) be two hashgraphs such that \( G \subseteq G' \) and both satisfy basic graph correctness. Further, suppose \( G' \) contains less than \( \frac{1}{3}n \) forking nodes. Let \( \emptyset \neq W_r, W'_r \) be the sets of all known witnesses with round \( r \) created in \( G \) and \( G' \), respectively. Let \( F_r \subseteq W_r \) and \( F'_r \subseteq W'_r \) be the sets of all famous witnesses of round \( r \).

If the fame state for all witnesses in \( W_r \) is decided, then it holds true that \( F_r = F'_r \), i.e., it is not possible to discover new famous witnesses of round \( r \) or alter the fame state of round \( r \) witnesses once the fame of all round \( r \) witnesses is decided.

**Proof.** Let all variables be as specified above. Assume that the famousness for all witnesses in \( W_r \) is decided, i.e., each event in \( W_r \) is either famous or not. Moreover, we can assume by Corollary 1 that \( W_r \subseteq W'_r \) for all \( r \in \mathbb{N} \).

**Proof of \( F_r \subseteq F'_r \)**
Let \( E_r \in W_r \) be an arbitrary but fixed witness whose fame has been decided by some \( E_{r+i+1} \in W_r \), written decision\(_G\)(\( E_r, E_{r+i+1}, \beta \)), for \( i \geq 1 \) with decision \( \beta \in \{\text{true, false}\} \). We prove the existence of a witness \( E'_{r+i'+1} \in W'_{r+i'+1} \) with decision\(_{G'\beta}\)(\( E_r, E_{r+i'+1}, \beta' \)) such that \( \beta' = \beta \); thus, we show a stronger statement: We do not only prove that (1) \( E_r \in F_r \) implies \( E_r \in F'_r \), but also the implication (2) \( E_r \notin F_r \Rightarrow E_r \notin F'_r \). In the what follows, we present a direct proof of both implications.

By Lemma 6, we know that decision\(_{G'\beta}\)(\( E_r, E_{r+i+1}, \beta \)), if there exists no witness that ends \( E_r \)'s virtual voting (by deciding its fame) before \( E_{r+i+1} \) can vote. First, assume this is the case; thus \( E'_{r+i'+1} = E_{r+i+1} \) and (1) and (2) holds true. So, we only have to prove that (1) and (2) also
hold true if \( E'_{r+i'+1} \neq E_{r+i+1} \), which implies that \( E'_{r+i'+1} \) ends \( E_r \)'s election by deciding its fame before \( E_{r+i+1} \) has the opportunity to do so. Clearly, this is only the case if \( i' \leq i \), which we assume in the following. Also notice that \( i' \geq 1 \) (cf. proof of Lemma 5), and \( E_{r+i'+1} \in G' \setminus G \) by Lemma 6.

\( E'_{r+i'+1} \) decides the fame for \( E_r \) w.r.t. the supermajority decision of the witnesses in the previous round that \( E'_{r+i'+1} \) can strongly see; let \( W' \subseteq W'_{r+i'} \) be the set of such witnesses that \( E'_{r+i'+1} \) strongly sees. By Corollary 3 and the definition of witnesses, it must be \( \frac{2}{3} n < m' = |W'| \leq n \), and the witnesses in \( W' \) have pairwise different creators by Corollary 2. We further define \( W'_\beta \subseteq W' \) to be the set of all witnesses in round \( r+i' \) with a vote of \( \beta' \), where \( |W'_\beta| > \frac{2}{3} n \) since \( E_{r+i'+1} \) has a supermajority of votes of \( \beta' \). Next, we define the voting set of round \( r+i' \):

\[
V_{r+i'} = \{ E_{r+i'} \in W'_{r+i'} \mid \exists E_{r+i+1} \in W'_{r+i'+1} : E_{r+i+1} \text{ strongly sees } E_{r+i'} \},
\]

(5.3) to be the set of all witnesses of round \( r+i' \) in \( G' \) that are strongly seen by some witness of round \( r+i'+1 \) in \( G' \), where \( |V_{r+i'}| \leq n \) by Corollary 3. With Corollary 2 we can conclude that all pairs of witnesses in \( V_{r+i'} \) have different creators. Subsequently, we differentiate whether (i) \( i' = i \) or (ii) \( i' < i \).

**Case (i)**

Suppose \( i' = i \) first. Then there must exist a set \( W \subseteq W_{r+i} \) (cf. Lemma 5) with \( |W| > \frac{2}{3} n \) so that \( E_{r+i+1} \) can strongly see, where each pair of witnesses in \( W \) have pairwise different creators (see Corollary 2). Additionally, we conclude with Corollary 3 that \( \frac{2}{3} n < |W| \leq n \). Because of \( \text{decision}_G(E_r, E_{r+i+1}, \beta) \), we have for all \( E_{r+i} \) with \( \text{vote}_G(E_r, E_{r+i}, \beta) \) also \( \text{vote}_{G'}(E_r, E_{r+i}, \beta) \) by Lemma 6; let \( W_\beta \subseteq W_{r+i} \) be the set of all such \( E_{r+i} \). Clearly, it applies \( |W_\beta| > \frac{2}{3} n \) and \( W_\beta \subseteq V_{r+i} \) (cf. Corollary 1), but we also have \( W'_\beta \subseteq V_{r+i} \). Therefore, \( V_{r+i} \) must have more than \( \frac{2}{3} n \) witnesses with votes for \( \beta \) and \( \beta' \). With \( |V_{r+i}| \leq n \), we conclude that \( \beta = \beta' \); thus (1) and (2) are satisfied.

**Case (ii)**

Suppose \( i' < i \). This case is a little more sophisticated. We claim the existence of at least \( \frac{1}{3} n \) witnesses \( E_{r+i'} \in W_{r+i'} \) such that \( \text{vote}_{G'}(E_r, E_{r+i'}, \beta) \) (and therefore also \( \text{vote}_G(E_r, E_{r+i'}, \beta) \)) by Lemma 6); let \( W_\beta \subseteq W_{r+i'} \) denote the set of such witnesses.

We show \( |W_\beta| \geq \frac{1}{3} n \) with proof by contradiction. Suppose for the purpose of contradiction that \( |W_\beta| < \frac{1}{3} n \). Then, all round \( r+i' \) + 1 witnesses \( E_{r+i'+1} \) in \( G \) will strongly see at least \( \frac{1}{3} n \) witnesses of the previous round with vote \( \overline{\beta} \) for \( E_r \); thus, \( \text{vote}_{G'}(E_r, E_{r+i'+1}, \beta) \). Hereafter, we have to distinguish whether round \( r+i'+2 \) is a normal voting or coin round. First assume round \( r+i'+2 \) is not a coin round. Then all witnesses of this round will have a supermajority of votes \( \overline{\beta} \) of the round \( r+i'+1 \) witnesses they strongly see. Let \( E_{r+i'+2} \) be the first witness in this round that can vote, then \( \text{decide}_G(E_r, E_{r+i'+2}, \overline{\beta}) \). This is clearly a contradiction to the assumption \( \text{decision}_G(E_r, E_{r+i+1}, \beta) \). Oppositely, suppose \( r+i'+2 \) is a coin round. Notice that witnesses in coin rounds cannot decide the fame; thus \( i' - i \geq 2 \). However, if they have a supermajority of votes, as in our case in favour of \( \overline{\beta} \), they will still vote according to the supermajority voting decision; hence, \( \text{vote}_{G'}(E_r, E_{r+i'+2}, \overline{\beta}) \) for all \( E_{r+i'+2} \in W_{r+i'+2} \). Finally, the first witness of round \( r+i'+3 \) able to vote will see a supermajority of \( \overline{\beta} \) votes; thus \( \text{decision}_G(E_r, E_{r+i+3}, \overline{\beta}) \), which is again a contradiction to the assumption \( \text{decision}_G(E_r, E_{r+i+1}, \beta) \). Concluding, \( |W_\beta| \geq \frac{1}{3} n \).
So, we have $\mathcal{W}_\beta \subseteq \mathcal{W}_{r+i} \subseteq \mathcal{V}_{r+i}$, but also $\mathcal{W}_{\beta'} \subseteq \mathcal{V}_{r+i}$. We know $|\mathcal{W}_{\beta'}| > \frac{2}{3}n$, and with $|\mathcal{V}_{r+i}| \leq n$ there can be at most $\frac{2}{3}n$ witnesses in $\mathcal{V}_{r+i}$ with vote $\beta$ for $E_r$, therefore missing one vote to have a supermajority decision of $\beta$ in favor of $E_r$; thus, $\beta' = \beta$, decide$_G(E_r, E_{r+i+1}, \beta)$, and (1), (2) are fulfilled. In particular, we conclude $\mathcal{F}_r \subseteq \mathcal{F}'_r$.

**Proof of $\mathcal{F}_r \supseteq \mathcal{F}'_r$**

Proof by contrapositive. Let $E'_r \in \mathcal{W}_r$ be an arbitrary but fixed round $r$ witness that is not in $\mathcal{F}_r$. First, suppose additionally $E'_r \in \mathcal{W}_r$. Then, by implication (2) $E_r \notin \mathcal{F}_r \Rightarrow E_r \notin \mathcal{F}'_r$ from earlier, we know $E'_r \notin \mathcal{F}'_r$. So, let $E'_r \in \mathcal{W}_r \setminus \mathcal{W}_r$ in the following.

Because of $\mathcal{W}_r \neq \emptyset$, there must exist a witness $E_r \in \mathcal{W}_r$ whose fame has been decided in some round $r + i + 1$ (i $\geq 1$) by some $E_{r+i+1} \in \mathcal{W}_{r+i+1}$, written decide$_G(E_r, E_{r+i+1}, \beta)$, during decideFame. Thus, virtual voting of $E_r$ must have at least reached round $r + 2$. By Lemma 5, there must exist a set of round $r$ witnesses $\mathcal{W} \subseteq \mathcal{W}_{r+1}$, with $|\mathcal{W}| > \frac{2}{3}n$, that are strongly seen by some $E_{r+2} \in \mathcal{W}_{r+2}$ (cf. proof of Lemma 5) that voted for $E_{r}$. Because of $E'_r \notin \mathcal{W}_r$, and more generally also $E'_r \notin G$, all round $r + 1$ witnesses in $\mathcal{W} \subseteq G$ cannot be descendants of $E'_r$. Therefore, vote$_G(E'_r, E_{r+i+1}, \text{false})$ for all $E_{r+i+1} \in \mathcal{W}$. Furthermore, we now $\mathcal{W} \subseteq \mathcal{V}_{r+1}$, so there cannot be more than $\frac{2}{3}n-1$ true-votes for $E'_r$ in $\mathcal{V}_{r+1}$; thus, decide$_G(E'_r, E_{r+i+2}, \text{false})$ or decide$_G(E'_r, E'_{r+i+2}, \text{false})$ by some $E_{r+i+2} \in \mathcal{W}_{r+i+2}$ that strongly sees a supermajority of false-votes and has the ability to vote before $E_{r+i+2}$. Finally, $E'_r \notin \mathcal{F}'_r$.

Hence, $\mathcal{F}_r = \mathcal{F}'_r$ for all $r \in \mathbb{N}$. This concludes this lemma.

**Corollary 4.**

Let $G, G'$ be the two hashgraphs as in Lemma 8, and let $\mathcal{U}_r, \mathcal{U}'_r$ denote set of all unique famous witnesses in round $r$ for $G$ and $G'$, respectively. Then it holds true that $\mathcal{U}_r = \mathcal{U}'_r$ if the fame state for all witnesses in $\mathcal{W}_r$ is decided.

**Proof.** By Lemma 8 it holds $\mathcal{F}_r = \mathcal{F}'_r$ if the fame state for all witnesses in $\mathcal{W}_r$ is decided. Clearly, if $E \in \mathcal{U}_r$, it must also be $E \in \mathcal{U}'_r$ since there can be no new famous witness in $\mathcal{F}'_r$ with the same creator. Analogously, $\mathcal{U}'_r \subseteq \mathcal{U}_r$.

**Corollary 5.**

Let $G, G'$ be the two hashgraphs as in Lemma 8, and let $r \in \mathbb{N}$ be an arbitrary round. If $\mathcal{W}_r \neq \emptyset$ and the fame state for all witnesses in $\mathcal{W}_r$ is decided, then it holds true that also the fame state for all witnesses in $\mathcal{W}'_r$ is decided.

**Proof.** We have $\mathcal{W}_r \subseteq \mathcal{W}'_r$ by Corollary 1 and $\mathcal{F}_r = \mathcal{F}'_r$ according to Lemma 8. This lemma also proved that non-famous witnesses are decided to be not famous in $G'$ (cf. implication (2)). Moreover, the proof for $\mathcal{F}'_r \subseteq \mathcal{F}_r$ in Lemma 8 demonstrates for all witnesses $E'_r \in \mathcal{W}_r \setminus \mathcal{W}_r$, the existence of a round $r + 2$ witness that decides $E'_r$ to be not famous. Therefore, the famousness for all witnesses in $\mathcal{W}'_r$ is decided.

**Lemma 9 (The Hashgraph Algorithm Achieves Self-Consistency).**

Let $G, G'$ be two hashgraphs such that $G \subseteq G'$ and both satisfy basic graph correctness. Further, suppose less than $\frac{1}{3}n$ forking nodes in $G'$. We denote with $\mathcal{H}_{\text{alg}}$ the hashgraph algorithm (i.e., divideRounds, decideFame, findOrder) that some honest participant locally runs with some (local) hashgraph as input.
Then, the following statement holds true for each event $E \in G$: If $\mathcal{H}_{\text{alg}}$ with input $G$ assigned $E$ a consensus position in the total order of events, then $\mathcal{H}_{\text{alg}}$ with input $G'$ will assign $E$ the same consensus position.

Proof. First, we note that the Hashgraph algorithm is deterministic; in particular for the function decideFame, the occurrence of a coin round, $i \mod \text{coinRound}$ (cf. Figure 3.6), is predefined by the variable coinRound which is equal for all nodes. Further, voting events may “flip a coin” in a coin round by voting w.r.t. their signature; clearly, this is not nondeterministic: If $G, G'$ share an event, they also share the same signature of this event. Therefore, $\mathcal{H}_{\text{alg}}$ running with the same hashgraph as input will always output the same total order of events.

Let $G, G'$ be the two hashgraphs from above, and assume $E \in G$ is an arbitrary event that $\mathcal{H}_{\text{alg}}$, with input $G$, assigned a consensus position $x$. That is, $\text{findOrder}$ assigned $E$ a round received number $r_E \in \mathbb{N}$. Recall that, generally, an event $E \in G$ has round received number $\hat{r}_E$, if the following three conditions are fulfilled in $\hat{G}$:

1. The fame for all witnesses in $\hat{W}_r$, for all $r \leq \hat{r}_E$, is decided.
2. $\hat{E}$ is an ancestor of all unique famous witnesses in $\hat{U}_r$.
3. $\hat{r}_E$ is minimal, i.e., there is no $r < \hat{r}_E$ fulfilling the two conditions above.

Because of $\text{roundReceived}_{G}(E) = r_E$, all three above conditions are fulfilled for $E$ in $G$.

We denote with $x'$ the consensus position that $\mathcal{H}_{\text{alg}}$ with input $G'$ assigns $E$. For this theorem, we must prove $x = x'$. Next, we define the round received set of $r$ in $G$,

$$C_r = \{ E_0 \in G \mid \text{roundReceived}_{G}(E_0) = r \} \subseteq G, \quad (5.4)$$

to bet the set of all events in $G$ with round received $r$. The Hashgraph algorithm sorts events (as one can observe in the $\text{findOrder}$ algorithm, see Figure 3.6) first by the round received number, then ties by a consensus timestamp, and any remaining ties lexicographically by the signature of events. Further, let $\Delta$ denote the relative consensus number of $E$ in $C_r$, i.e., the position of $E$ in $C_r$ ordered by consensus timestamp, and then lexicographically by signatures, where $0 \leq \Delta \leq |C_r|$. Analogously, let $C'_r$ denote the round received set of $r$ in $G'$, and $\Delta'$ the relative consensus number of $E$ in $C'_r$. In order to prove this theorem, we show (i) $C_r = C'_r$ for all $r \leq r_E$ and (ii) $\Delta = \Delta'$; this implies

$$x = \bigcup_{r=1}^{r_E-1} C_r + \Delta = \bigcup_{r=1}^{r_E-1} C'_r + \Delta = \bigcup_{r=1}^{r_E-1} C'_r + \Delta' = x'. \quad (5.5)$$

Also notice that by condition (3) the sets $C_r$ (and $C'_r$) are each pairwise disjoint for all $r \leq r_E$, i.e., $C_{r_1} \cap C_{r_2} = \emptyset$ (and $C'_{r_1} \cap C'_{r_2} = \emptyset$) for all $r_1, r_2 \leq r_E, r_1 \neq r_2$.

Proof of $C_r = C'_r$.

We first show $C_r \subseteq C'_r$ and $C_r \supseteq C'_r$ subsequently.

---

Footnote: In our implementation, nodes may have different states for the internal variables $\text{isWitness}$, $\text{isFamous}$, and $\text{roundReceived}$. However, in Corollary 1 we proved that for two hashgraphs $G, G'$, with $G \subseteq G'$, follows $\forall \mathcal{W}_r \subseteq \mathcal{W}'_r$, and $\mathcal{F}_r \subseteq \mathcal{F}'_r$ (cf. proof of Lemma 8) for all rounds $r \in \mathbb{N}$. Moreover, we do not reset the variable $\text{roundReceived}$, because round received numbers do not change as we will demonstrate in this proof.
5.3 Consistency of the Hashgraph Algorithm

Let \( E_0 \in C_r \) and \( r \leq r_r \) be arbitrary; we first show \( E_0 \in C_r \Rightarrow E_0 \in C'_r \), by proving \( \text{roundReceived}_{C}(E_0) = r \), i.e., \( E_0 \) fulfills all three round received conditions for \( r \) in \( G' \). First, because of \( \text{roundReceived}_{C}(E) = r_r \), we know by condition (1) from above that the fame for all witnesses in \( \mathcal{W}_r \), for all \( r' \leq r_r \), is decided. By Corollary 5 is also the fame for all witnesses in \( \mathcal{W}'_r \), for all \( r' \leq r_r \), decided; hence, \( E_0 \) satisfies condition (1) in \( G' \).

Furthermore, by Lemma 8 applies \( F_r = F'_r \), for all \( r' \leq r_r \). This together with \( G \subseteq G' \) implies that \( E_0 \) is an ancestor of all (unique) famous witnesses in \( F'_r \subseteq G' \); hence, condition (2) is fulfilled for \( E_0 \) in \( G' \).

Condition (3), i.e. \( r \) is minimal in \( G' \), still needs to be shown: We already know that the fame for all witnesses \( \mathcal{W}'_r \), for all \( r' \leq r_r \), is decided in \( G' \); therefore, we have to prove that \( E_0 \) is not an ancestor of all unique famous witnesses in \( \mathcal{U}'_r \). Because of \( E_0 \in C_r \), \( E_0 \) cannot be an ancestor of all \( \mathcal{U}'_r \) witnesses; otherwise \( \text{roundReceived}_{C}(E_0) < r_r \), which is a contradiction to \( E_0 \in C_r \). By Corollary 4, \( \mathcal{U}'_{r-1} = \mathcal{U}'_{r-1} \); hence, \( E_0 \) is also not an ancestor of all \( \mathcal{U}'_{r-1} \). Therefore, condition (3) is satisfied by \( E_0 \) in \( G' \), and we have \( \text{roundReceived}_{C'}(E_0) = r \) and \( E_0 \in C'_r \).

Conversely, we show the contrapositive, i.e., the implication \( E_0 \notin C_r \Rightarrow E_0 \notin C'_r \). Let \( E_0 \in G' \) be arbitrary but \( E_0 \notin C_r \). First, suppose there exists a \( r' \leq r_r \), \( r' \neq r \), such that \( E_0 \in C'_r \). But then, as we just proved above, immediately follows \( E_0 \in C'_r \); thus \( E_0 \notin C'_r \). So, let us assume \( E_0 \) has either no round received in \( G \) or \( \text{roundReceived}_{C}(E_0) > r_r \), in what follows.

We want to show \( \text{roundReceived}_{C'}(E_0) \neq r \) to conclude \( E_0 \notin C'_r \). We already know that all witnesses in \( \mathcal{W}'_r \), for all \( r' \leq r_r \), are decided to be either famous or not; thus, for rounds \( r' = 1, \ldots, r_r \), all famous and, in particular, unique famous witnesses \( \mathcal{U}_r = \mathcal{U}_r' \) are known (cf. Lemma 8 and Corollary 4). Since, for all \( r' \leq r_r \), \( E_0 \notin C'_r \) and round received condition (1) is fulfilled for \( E_0 \) in \( G \), \( E_0 \) cannot be an ancestor of all \( \mathcal{U}_1, \ldots, \mathcal{U}_r \). By \( \mathcal{U}_1 = \mathcal{U}_1', \ldots, \mathcal{U}_r = \mathcal{U}_r' \), we conclude \( E_0 \) has either no round received in \( G' \) or \( \text{roundReceived}_{C'}(E_0) > r_r \); hence, \( E_0 \notin C'_r \).

**Proof of \( \Delta = \Delta' \)**

It remains to show that all events in \( C_r = C'_r \) are identically sorted by consensus timestamp, and remaining ties by their signature lexicographically. Clearly, \( H_{alg} \) will calculate the correct order for the latter if \( C_r = C'_r \). So, it is sufficient to prove that events will receive the same consensus timestamp by \text{findOrder} \ with input \( G \) and \( G' \).

The consensus timestamp of an event \( E_0 \in C_r \) is defined to be the median of the timestamps of the events in \( S \subseteq G \), which is defined as

\[
S = \{ E_1 \in G \mid \exists E_2 \in G: E_2 \in \mathcal{U}_r, E_1 \text{ is a self-ancestor of } E_2, E_0 \text{ is an ancestor of } E_1, E_0 \text{ is not an ancestor of } E_1 \text{'s self-parent} \}.
\]

Observe that each event \( E_1 \in S \) is a self-ancestor of a unique famous witness in \( \mathcal{U}_r \). Clearly, with \( \mathcal{U}_r = \mathcal{U}_r' \) and \( G \subseteq G' \) immediately follows \( E_1 \in S' \); thus, \( S \subseteq S' \). Conversely, we also have \( S' \subseteq S \); thus all events in \( C_r, C'_r \) are assigned identical consensus timestamps in \( G \) and \( G' \). We conclude \( \Delta = \Delta' \).
Further, we define the system \( \mathcal{H}_{\text{alg}} \) assigns on input \( G \) event \( E \) the same consensus position in the total order of events as \( \mathcal{H}_{\text{alg}} \) with input \( G' \).

**Theorem 2 (The Hashgraph Algorithm Achieves Consistency).**

Let \( G, G' \) be two consistent hashgraphs, both satisfying basic graph correctness, such that \( G \cup G' \) contains less than \( \frac{1}{3}n \) forking nodes. Let \( \mathcal{H}_{\text{alg}} \) denote the hashgraph algorithm (see Lemma 9).

Then, the following statement holds true for each event \( E \in G \): If \( \mathcal{H}_{\text{alg}} \) with input \( G \) assigned \( E \) consensus position \( x \) in the total order of events, then \( \mathcal{H}_{\text{alg}} \) with input \( G' \) will assign \( E \) either no consensus position or \( x \).

**Proof.** Let \( x \) be the consensus position of \( E \) by \( \mathcal{H}_{\text{alg}} \) on input \( G \). If \( \mathcal{H}_{\text{alg}} \) with input \( G' \) assigns \( E \) no consensus position, the theorem is fulfilled. So, suppose \( \mathcal{H}_{\text{alg}} \) with input \( G' \) assigns \( E \) consensus position \( x' \). Moreover, let \( y \) be the consensus position \( E \) received by \( \mathcal{H}_{\text{alg}} \) on input \( G \cup G' \), where \( G \cup G' \) inherits the basic graph correctness property from \( G \) and \( G' \). By Lemma 9, we have \( x = y \) and \( x' = y \); thus, \( x = x' \). This concludes this proof.

### 5.4 Accountability of Hashgraph w.r.t. Consistency

**Definition 10 (Consistency of Hashgraph).**

Consider a run \( \omega \) as in Definition 3, where \( \omega \notin \alpha \). Let nodes be the set of Hashgraph node identities (as specified as a parameter for \( \mathcal{P}^H \)), and assume \( J \) is an instance of \( \mathcal{F}_j^H \). Let \( ((\pi_d, \text{msglist}_i), \sigma) \) be a state update received by \( J \) such that

- \( \pi_d \in \text{nodes} \),
- \( \text{verifySig}_{\phi_i}(\pi_d, \text{msglist}_i, \sigma) = \text{true} \), and
- \( \text{msglist}_i \subseteq \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \).

Further, we define \( \mathcal{M}_i(\omega) \) to be the set of all \( \text{msglist}_i \) (from \( \pi_d \)) that have been reported up to point \( |\omega| \).

**Common Prefix.** We say two message lists \( \text{msglist}_1, \text{msglist}_2 \in \mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \) share a prefix if \( \text{msglist}_1 \subseteq \text{msglist}_2 \) or \( \text{msglist}_2 \subseteq \text{msglist}_1 \).

**Self-Consistency.** We call \( \pi_d \in \text{nodes} \) self-consistent at point \( |\omega| \) if all pairs of message lists \( \text{msglist}_1, \text{msglist}_2 \in \mathcal{M}_i(\omega) \) share a prefix.

**Node-Consistency.** We say that \( \pi_d, \pi_d' \in \text{nodes} \), \( \pi_d \neq \pi_d' \), are consistent at point \( |\omega| \) if all pairs of message lists \( \text{msglist}_1, \text{msglist}_2 \in \mathcal{M}_i(\omega) \) share a prefix.

The system \( \{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\} \) satisfies consistency in the run \( \omega \) if all nodes are self-consistent at every point of the run \( \omega \), and all pairs of nodes are consistent at every point \( |\omega| \).

**Definition 11 (Accountability w.r.t. Consistency).**

Consider a run \( \omega \) as in Definition 3. Again, we denote with nodes the set of all Hashgraph node identities and assume \( J \) to be an instance of \( \mathcal{F}_j^H \).
5.4 Accountability of Hashgraph w.r.t. Consistency

\( \gamma_1 \) (Self-Consistency). The security property \( \gamma_1 \) contains all runs, where \( \omega \notin \alpha \) and there exists some participant \( \text{pid} \in \text{nodes} \) such that \( \text{pid} \) does not satisfy self-consistency at some point \( [\omega] \).

\( \gamma_2 \) (Node-Consistency). This security property contains all runs, where \( \omega \notin \alpha \) and there exist two participants \( \text{pid}_1, \text{pid}_2 \in \text{nodes}, \text{pid}_1 \neq \text{pid}_2 \), such that \( \text{pid}_1 \) and \( \text{pid}_2 \) are not consistent at some point \( [\omega] \).

We define the accountability constraint \( C^H_3 \) with the property \( \gamma = \gamma_1 \cup \gamma_2 \cup \alpha \) as

\[
C^H_3 := (\gamma \Rightarrow \text{dis} (\text{pid}_1) \mid \cdots \mid \text{dis} (\text{pid}_n)),
\]

and set the accountability property \( \Phi_3 := \{ C^H_3 \} \).

We say \( \mathcal{P}^H \) with parameters as in Definition 3 is individually accountable w.r.t. consistency if for all environments \( \mathcal{E} \) and adversaries \( \mathcal{A} \) the system \( \{ \mathcal{E}, \mathcal{A}, \mathcal{P}^H \} \) is \( \Phi_3 \)-accountable w.r.t. \( J \).

Lemma 10.
Let \( G_1, G_2 \) be two hashgraphs, both satisfying basic graph correctness, with \( G_1 \cup G_2 \) containing less than \( \frac{1}{7} n \) forking nodes. We denote with \( \text{msglist}_1, \text{msglist}_2 \) the message list of ordered entries that \( \mathcal{H}_\text{alg} \) outputs on input \( G_1 \) and \( G_2 \), respectively. Then, it holds true that \( \text{msglist}_1 \) and \( \text{msglist}_2 \) share a prefix, i.e., \( \text{msglist}_1 \subseteq \text{msglist}_2 \) or \( \text{msglist}_2 \subseteq \text{msglist}_1 \) (**).

Proof. Let \( G_1, G_2 \) be the hashgraphs from above. We denote with \( E^1_1, E^1_2, \ldots , E^1_{k_1} \) the total order of \( k_1 \in \mathbb{N} \) events \( \mathcal{H}_\text{alg} \) outputs with input \( G_1 \), i.e., the order of events \( \text{pid}_1 \) calculates locally by running the Hashgraph algorithm \( \mathcal{H}_\text{alg} \). Analogously, let \( E^2_1, E^2_2, \ldots , E^2_{k_2} \) be the total order of \( k_2 \in \mathbb{N} \) events by \( \mathcal{H}_\text{alg} \) on input \( G_2 \). We know \( G_1 \cup G_2 \) has less than \( \frac{1}{7} n \) forking nodes and both \( G_1 \) and \( G_2 \) satisfy basic graph correctness. Moreover, \( G_1, G_2 \) are consistent hashgraphs by Lemma 3. We therefore conclude with Theorem 2, \( E^1_i = E^2_i \) for all \( i = 1, \ldots , \min \{ k_1, k_2 \} \), i.e., \( \text{pid}_1 \) and \( \text{pid}_2 \) calculate the same prefix of ordered events.

It still needs to be shown that \( \text{msglist}_1, \text{msglist}_2 \) share a prefix. Observe in the implementation of \( \text{findOrder} \) in \( \mathcal{P}^H_{\text{node}} \) or \( \mathcal{P}^H_{\text{judge}} \) that \( \text{findOrder} \) calculates with input \( G_1 \) and \( G_2 \) the same order, say \( j \), for an (yet) unordered message list entry \( (\varepsilon, \text{txId}, \text{eventID}, \text{msg}) \), where \( \text{msg} \) is from the transaction set \( \text{txs} \) of an ordered event \( E^1_k = E^2_k, k \leq \max \{ k_1, k_2 \} \). That is, \( \text{findOrder} \) adds \( (j, \text{txId}, \text{eventID}, \text{msg}) \) to the message list \( \text{msglist}_i \) with input \( G_i \), for \( i \in \{ 1, 2 \} \). This holds because of the common prefix of ordered events of \( G_1 \) and \( G_2 \) and the fact that \( \text{findOrder} \) iterates over all unordered entries, belonging to the same event, in the same order. Hence, \( \text{msglist}_1 \) and \( \text{msglist}_2 \) share a common prefix of ordered message list entries.

Notice that many of the above statements may not hold true with negligible probability. 

Theorem 3 (Hashgraph Achieves Individual Accountability w.r.t. Consistency).
Consider runs for the system \( \{ \mathcal{E}, \mathcal{A}, \mathcal{P}^H \} \) with parameters for \( \mathcal{P}^H \) as in Definition 3 and let \( J \) be an instance of \( \mathcal{P}^H_{\text{judge}} \). Then, it holds true that \( \mathcal{P}^H \) achieves individually accountability w.r.t. consistency.

Proof. Let \( \mathcal{P}^H \) be the Hashgraph protocol with parameters as described above, and let \( \mathcal{E} \) be an arbitrary environment and \( \mathcal{A} \) be an arbitrary adversary. Towards proving this theorem, we have to show that \( J \) ensures \( \Phi_3 \)-accountability for the system \( \mathcal{Q} = \{ \mathcal{E}, \mathcal{A}, \mathcal{P}^H \} \). Again, we do this by proving fairness and completeness independently. As before, we consider only a single Hashgraph instance in this proof. We first state some observations and general procedures in \( \mathcal{P}^H_{\text{judge}} \):
5 Security and Accountability of Hashgraph

Handling of state updates in $F_J^H$ with invalid signatures
We discussed earlier for the proof of Lemma 2 that all (supposed) state updates $((\text{pid}, \text{currentState}), \sigma)$ submitted to $J$ by some client are discarded if the containing signature is invalid with the tuple $(\text{pid}, \text{currentState})$ (i.e., $\text{verifySig}_{\text{pk}(\text{pid})}((\text{pid}, \text{currentState}), \sigma)$ outputs $\text{false}$) since $J$ cannot definitively decide if $\text{pid}$, the reporting client, or both misbehaved. Consequently, such invalid state updates are ignored and do not influence any processing in $F_J^H$.

Handling of valid state updates w.r.t. basic correctness at $F_J^H$
We first describe how $J$ processes valid state updates (i.e., state updates containing a valid signature) and which circumstances can lead to some node $\text{pid}$ being blamed:

- $J$ first checks whether the message list $\text{msglist}$ of the state update is a subset of $\mathbb{N} \times \{0, 1\}^n \times \{0, 1\}^*$; that is, $J$ ensures condition (ii) of $\alpha_1$.
- Next, $J$ checks if condition (i) of $\alpha_4$ is satisfied, i.e., all entries $(\text{ctr}, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$ have a valid transaction ID, namely $\text{hash}(\text{msg}) = \text{txId}$.
- Lastly, condition (ii) of $\alpha_4$ is checked by $J$, i.e., $\text{msglist}$ from $\text{pid}$ must be a consecutive sequence with no duplicates or gaps w.r.t. the parameter $\text{ctr}$; formally, there exists exactly one entry $(\text{ctr}, \text{txId}, \text{eventID}, \text{msg}) \in \text{msglist}$ for all $\text{ctr} = 1, \ldots, |\text{msglist}|$.

In Lemma 2 it was proven that $J$ is fair and complete under the above three basic correctness conditions. Particularly, all the aforementioned conditions for basic correctness hold true in a run $\omega \in \gamma$, i.e., a run that violates consistency does satisfy basic correctness, by definition of $\gamma$. Since $\mathcal{P}_J^H$ is accountable w.r.t. basic correctness, it remains solely to prove that $J$ is fair and complete w.r.t. the security properties $\gamma_1$ and $\gamma_2$.

Handling of message list states of nodes
Analogous to clients, $J$ stores in the local variable states for each node $\text{pid}$ the longest known message list, i.e., $\text{states}[\text{pid}] = \text{msglist}'$ expresses that $\text{msglist}'$ is the longest message list of $\text{pid}$, which was previously submitted by some client. After initialization of $J$ applies $\text{states}[\text{pid}] = \emptyset$, for all $\text{pid} \in \text{nodes}$, since no client submitted any state update yet.

$J$ receives a new longest message list from $\text{pid}$
If the received message list ($\text{msglist}$) contained in a valid state update passes all basic correctness conditions, and it is the longest known message list from $\text{pid}$ so far, i.e., $\text{states}[\text{pid}] \subsetneq \text{msglist}$, then $J$ updates the state for $\text{pid}$ by setting $\text{states}[\text{pid}] \leftarrow \text{msglist}$. Notice that the older state of $\text{pid}$ ($\text{states}[\text{pid}]$) before the assignment of $\text{msglist}$ must be a subset of the newly received $\text{msglist}$. If this is not the case, then there is a violation of self-consistency and $\text{pid}$ must have misbehaved (we discuss this case later).

Besides the internal variable states that stores all longest submitted message lists from each node so far, $J$ manages another internal variable for storing the longest so far received message list from all nodes that have not been blamed for a violation of self-consistency ($\gamma_1$) or node-consistency ($\gamma_2$), namely $\text{msglist}_{\text{max}}$. This initially empty message list is set to $\text{msglist}$ if not only $\text{states}[\text{pid}] \subsetneq \text{msglist}$ applies, but also $\text{msglist}_{\text{max}} \subseteq \text{msglist}$ holds true. Observe that even if some node, say $\text{pid}$, violates self-consistency, it still holds true that $\text{states}[\text{pid}] \subseteq \text{msglist}_{\text{max}}$. After $J$ outputs a verdict for $\text{pid}$, due to the violation of self-consistency, $J$ recalculates $\text{msglist}_{\text{max}}$ by including only nodes that have not violated consistency so far.
5.4 Accountability of Hashgraph w.r.t. Consistency

Nodes that $J$ has determined to have misbehaved w.r.t. consistency are captured in the set $\text{consistencyVerdicts}$. Misbehaving nodes are added to this set after $J$ outputs a verdict for such nodes. But it still remains to be shown that $J$ actually blames nodes that violate consistency. We will prove later that $J$ indeed blames nodes that violate consistency; thus, $\text{consistencyVerdicts}$ captures exactly all nodes that violated $\gamma$.

This completes the processing of a common state update in $\mathcal{F}_\text{judge}^H$ where node-consistency is not violated.

**Forking nodes**

$J$ keeps track of all nodes that violate fork-freeness by means of the set $\text{forkingNodes} \subseteq \text{nodes}$. Observe in Figure 3.13 that $J$ does not only output a verdict for a forking node but also adds the misbehaving node to the set $\text{forkingNodes}$. Since all honest nodes report their local hashgraph $G$ to $J$ before running $\mathcal{H}_\text{alg}$ with input $G$ to calculate the order of entries in the message list, $J$ will know if $E_W$, the union of all reported hashgraphs so far, contains at least $\frac{1}{3}n$ forking nodes. If this is the case, $J$ no longer outputs any verdicts regarding $\gamma_1, \gamma_2$ for the rest of the run. This is vital to ensure that $J$ does not blame any honest nodes that may calculate false states, due to the missing precondition of less than $\frac{1}{3}n$ forking nodes of Lemma 10.

Notice that two honest nodes can each have a hashgraph with less than $\frac{1}{3}n$ forking nodes, but the union of both hashgraphs, say $G$, contains $\frac{1}{3}n$ forking nodes. Besides, these two nodes may even calculate message lists that do not have a common prefix. But this implies by Lemma 10 that $G$ has more than $\frac{1}{3}n$ forking nodes.

**Verdicts of $J$ w.r.t. consistency $\gamma$**

Observe in Figure 3.14 of $\mathcal{F}_\text{judge}^H$ that there is exactly one verdict for security properties $\gamma_1$ and $\gamma_2$, namely $\text{dis}(\text{pid}_{\text{send}})$ and $\text{dis}(\text{pid})$, where $\text{pid}, \text{pid}_{\text{send}} \in \text{nodes}$. Clearly, these verdicts achieve individual accountability for the accountability constraint $C_3^H$.

**Fairness.** Analogously to the proof for fork-freeness, it has to be shown that $\Pr[Q(1^n) \mapsto \{(J : \psi) \mid |\omega| \neq \psi\}]$ is negligible as a function in $\eta$, i.e., $J$ renders a false verdict $\psi$ only in a negligible amount of runs. In the proof of Lemma 2, we already showed that $J$ is fair w.r.t. the verdicts of basic correctness ($\alpha$); therefore it is left to show that $J$ is also fair for the verdicts capturing properties $\gamma_1$ and $\gamma_2$. In the following, let $\text{msglist}$ be a message list of a valid state update of an honest node $\text{pid} \in \text{nodes}$.

**$J$ is fair w.r.t. the verdict of $\gamma_1$**

One can observe that $J$ outputs a verdict for the violation of self-consistency only if $\text{states}[\text{pid}]$ and $\text{msglist}$ do not share a common prefix, i.e., $\text{states}[\text{pid}] \not\subseteq \text{msglist}$ and $\text{msglist} \not\subseteq \text{states}[\text{pid}]$. Recall that $\text{states}[\text{pid}]$ stores the longest of all previously submitted message lists that were signed by $\text{pid}$.

\[\text{The precondition of less than } \frac{1}{3}n \text{ forking nodes can be traced back to the strongly seeing lemma.}\]
5 Security and Accountability of Hashgraph

The proof for fairness of property $\alpha_2$ illustrated (basic graph correctness) that honest nodes only submit hashgraphs to $J$ that satisfy all conditions for basic graph correctness. In particular, the aforementioned proof also demonstrates that hashgraphs of honest nodes fulfill basic graph correctness at every point during the run $\omega$, since honest nodes only merge other hashgraphs that fulfill basic graph correctness into their local one.

Let $G_1$ be the hashgraph $\text{pid}$ used to calculate states[$\text{pid}$], and $G_2$ be the hashgraph $\text{pid}$ used to calculate $\text{msglist}$. If $\text{pid}$ is honest, it will have submitted $G_1$ and $G_2$ to $J$. The judge $J$ will not render any verdicts if it detects at least $\frac{1}{2}n$ forking nodes; thus, $J$ is always fair if it detects $\frac{1}{3}n$ forking nodes. Suppose this is not the case, then it holds in particular that $G_1 \cup G_2$ contains less than $\frac{1}{3}n$ forking nodes (because $E_W$ contains less than $\frac{1}{2}n$ forking nodes). Hence, $\text{msglist}$ and states[$\text{pid}$] share a prefix by Lemma 10 (**), and thus $J$ will never blame an honest node for a violation of $\gamma_1$ (**).

$J$ is fair w.r.t. the verdict of $\gamma_2$

Observe in the implementation of $F^H_{\text{judge}}$ that $J$ blames nodes for a violation of $\gamma_2$ only if $\text{msglist}_{\text{max}}$ and $\text{msglist}$ do not share a prefix. If this is the case, $J$ recalculates $\text{pid}$’s latest state ($\text{msglist}$) on the latest submitted hashgraph, say $G$, by $\text{pid}$ and checks whether states[$\text{pid}$] is a subset of the recalculated state of $\text{pid}$. Let $G'$ be the hashgraph $\text{pid}$ used to calculate states[$\text{pid}$], i.e., $\text{pid}$ ran $H_{\text{alg}}$ with input $G'$ which outputted states[$\text{pid}$]. Honest nodes always report new hashgraphs (i.e., a merged hashgraph of a sync that $\text{pid}$ accepted) to $J$. Thus, $G' = G$ or $G' \subset G$ if $G$ is a more recent hashgraph of $\text{pid}$ (cf. basic correctness $\alpha_3$).

We know $G$ contains less than $\frac{1}{3}n$ forking nodes (otherwise $J$ would know about it and discard any new received message lists). Accordingly, we can conclude with Lemma 10 that $\text{msglist}$ and states[$\text{pid}$] share a prefix (**). Because of $G' \subset G$, states[$\text{pid}$] $\subseteq \text{msglist}$ follows. Hence, $\text{pid}$ will not be blamed by $J$ (**).

Consequently, $J$ is fair w.r.t. the verdicts for $\gamma$ except for a negligible amount of runs. That is, $\Pr[Q(1^\eta) \mapsto (J : \psi) | [\omega] \neq \psi]$ is negligible as a function in $\eta$.

Completeness. For completeness, it must be shown that $\Pr[Q(1^\eta) \mapsto \neg(J : \Phi_2)]$ is negligible as a function in $\eta$. That is, $J$ ensures the accountability constraint $C^H_3$ in all but a negligible number of runs of the system $\{E, A, F^H\}$. Specifically, for a run $\omega \in \gamma$, $J$ must state a verdict $\bigwedge_{i \in I} \text{dis}(\text{pid}_i)$ for $\emptyset \neq I \subset \{1, \ldots, n\}$ to ensure individually accountability of $\Phi_3$. As discussed above, all verdicts for runs in $\gamma_1$ or $\gamma_2$ accomplish individually accountability, and for runs in $\alpha$ we already showed that $J$ satisfies individual accountability w.r.t. basic correctness (see Lemma 2). Therefore, it is left to prove that $J$ is also complete for runs in $\gamma_1 \cup \gamma_2$. Subsequently, let $\omega$ be a run of $\gamma_1$, $\gamma_2$, or both (thus, basic correctness is always satisfied). For the rest of this proof, we differentiate whether $\omega \in \gamma_1$ or $\omega \in \gamma_2$.

1. Let $\omega \in \gamma_1$.

Then, by definition of $\gamma_1$, there exists some $\text{pid} \in$ nodes that does not satisfy self-consistency at some point $[\omega]$, i.e., some client(s) submitted message lists $\text{msglist}_1, \text{msglist}_2$ to $J$ in separate valid state updates, each containing a valid signature $\sigma_i$ of $\text{msglist}_i$ that was sigend by $\text{pid}$, for

---

1For proving completeness, the case $\omega \in \gamma_1 \cap \gamma_2$ is not relevant because it is already included in case 1. and 2..
We claim that \( i \in \{1, 2\} \). W.l.o.g. let \( |msglist_1| < |msglist_2| \). Next, we do a case distinction whether (i) \( J \) stated some verdict for \( \text{pid} \) preceding point \( [\omega] \) or \( J \) identified at least \( \frac{1}{3} n \) forking nodes (i.e., \( |\text{forkingNodes}| \geq \frac{1}{3} n \)), or (ii) \( \text{pid} \) has not been blamed by \( J \) until point \( [\omega] \).

**Case (i):** \( \text{dis}(|\text{pid}|) \in \text{verdicts} \lor |\text{forkingNodes}| \geq \frac{1}{3} n \)

First, if \(|\text{forkingNodes}| \geq \frac{1}{3} n \), then \( J \) identified and blamed at least \( \frac{1}{3} n \) nodes for violation of fork-freeness; therefore, \( J \) ensures the constraint \( C^H_3 \) and completeness is satisfied for this case.

Suppose less than \( \frac{1}{3} n \) forking nodes. Observe in Figure 3.14 of \( F^H_{\text{judge}} \) that \( J \) discards all message lists for nodes that are already blamed. Since basic correctness is satisfied, \( \text{pid} \) must have violated fork-freeness or consistency at some point prior to \( [\omega] \). Trivially, \( J \) already blamed \( \text{pid} \); thus, \( J \) also ensures the constraint \( C^H_3 \) in this case.

**Case (ii):** \( \text{dis}(|\text{pid}|) \notin \text{verdicts} \)

It must be shown that \( J \) states in this run the verdict \( \text{dis}(|\text{pid}|) \) for the violation of self-consistency by \( \text{pid} \). We prove this statement by showing the following proposition:

We claim that \( \text{pid} \) satisfies self-consistency in every point \( [\omega] \) before \( msglist_2 \) was received by \( J \), i.e., \( msglist_2 \) is the first message list received by \( J \) such that self-consistency is violated.

**Direct proof**

Let \( msglist'_1 \) be the message list from \( \text{pid} \) that was received by \( J \) such that \( \text{pid} \) violates self-consistency the first time in run \( \omega \). Accordingly, there exists another \( msglist'_1 \subset \text{states}[\text{pid}] \), so that \( msglist'_1 \) and \( msglist'_2 \) do not share a prefix. Particularly, it holds true that \( msglist'_1 \not\subset msglist'_2 \) and \( msglist'_2 \not\subset msglist'_1 \). But then we also have \( \text{states}[\text{pid}] \not\subset msglist'_2 \). Therefore, \( |\text{states}[\text{pid}]| < |msglist'_2| \), or \( \text{states}[\text{pid}] \) and \( msglist'_2 \) do not share a prefix. From both cases we conclude \( msglist'_2 \not\subset \text{states}[\text{pid}] \). Observe in Figure 3.14 that \( J \) states the verdict \( \text{dis}(|\text{pid}|) \) if, as in our case, \( \text{states}[\text{pid}] \) and \( msglist'_2 \) do not have a common prefix. However, since \( \text{dis}(|\text{pid}|) \notin \text{verdicts} \), it must be that \( msglist'_2 = msglist_2 \). We conclude that \( \text{pid} \) satisfies self-consistency before \( J \) received \( msglist_2 \); this proves the statement.

The above proof demonstrates in particular that \( J \) states a verdict for \( \text{pid} \). We conclude that \( J \) is complete w.r.t. self-consistency (**).

2. Let \( \omega \in \gamma_2 \).

By definition of \( \gamma_2 \), there exist two nodes \( \text{pid}_1, \text{pid}_2 \in \text{nodes} \), \( \text{pid}_1 \neq \text{pid}_2 \) such that \( \text{pid}_1 \) and \( \text{pid}_2 \) are not consistent at some point \( [\omega] \), i.e., there exist two message lists

\[
msglist_1 \in M_1([\omega]), \; msglist_2 \in M_2([\omega])
\]

from valid state updates that were reported to \( J \) by some client(s) such that \( msglist_1 \) and \( msglist_2 \) do not share a prefix at this point \( [\omega] \). Specifically, for the two message lists applies \( msglist_1 \nsubseteq msglist_2 \land msglist_2 \nsubseteq msglist_1 \). We have to prove for completeness that \( J \) states at least one verdict, blaming \( \text{pid}_1 \) or \( \text{pid}_2 \).
5 Security and Accountability of Hashgraph

W.l.o.g. let msglist₂ be the latter of the two message lists that J receives. We fix |ω| to be the point in ω where J just receives msglist₂ without doing any further processing. If we talk about a point in ω prior to |ω|, we assume J has not yet received msglist₂. To begin with, we assume that |ω| is the first point in ω such that node-consistency is violated. We will later show that J outputs additional verdicts for further node-consistency violations.⁸

First, assume there are at least 2\(\frac{n}{3}\) forking nodes, i.e., |forkingNodes| ≥ \(\frac{2}{3}n\). But then J must have already blamed at least \(\frac{1}{3}n\) nodes due to violation of fork-freeness before |ω|; thus, J fulfills completeness under this circumstance.

So, suppose |forkingNodes| < \(\frac{1}{3}n\). Since |ω| is the first point in ω where node-consistency is violated, J stated no verdict for pid₁ and pid₂ w.r.t. node-consistency, due to J’s fairness w.r.t. node-consistency, until |ω|. If pid₁ or pid₂ violates self-consistency at point |ω| (or before), J will (have) output(led) a verdict for pid₁ or pid₂, as demonstrated in the completeness proof of γ₁; hence, completeness for γ₂ is again satisfied. Therefore, we assume for the rest of this proof that pid₁ and pid₂ satisfy self-consistency at point |ω| and conclude pid₁, pid₂ ∉ consistencyVerdicts at |ω|.

At the beginning of this proof we discussed that msglist_max is the longest state of all nodes until there is a node-consistency violation. Particularly, at point |ω| holds true that msglist₁ ⊆ states[pid₁] and states[pid₁] ⊆ msglist_max, for i ∈ {1, 2}.

**Processing of J upon receiving msglist₂**

We describe the processing of J upon receiving msglist₂: Since msglist₂ is a proper superset of states[pid₂] (otherwise, msglist₂ ⊆ states[pid₂] implies that msglist₂ ⊆ msglist_max and thus msglist₁, msglist₂ share a prefix which is clearly a contradiction). J sets states[pid₂] ← msglist₂.

Next, we have to prove that msglist_max and msglist₂ do not share a prefix. This is important so that J can call the internal function generateReport, which tries to recalculate the state of all nodes and blames participants for which this is not possible.

**Proof: msglist_max ∉ msglist₂**

From above applies msglist₁ ⊆ states[pid₁] ⊆ msglist_max. Hence, msglist_max ∉ msglist₂ (otherwise msglist₁ ⊆ msglist_max ⊆ msglist₂, which is a contradiction to msglist₁ ∉ msglist₂).

**Proof: msglist₂ ∉ msglist_max**

Suppose msglist₂ ⊆ msglist_max for the purpose of contradiction. This and the valid statement msglist₁ ⊆ msglist_max implies that msglist₁ and msglist₂ share a prefix. Based on this contradiction, we conclude msglist₂ ∉ msglist_max.

Therefore, msglist_max and msglist₂ do not share a prefix. Consequently, J calls the internal function generateReport since msglist₂ = states[pid₂]. Due to msglist₁ ⊆ states[pid₁], msglist₂ = states[pid₂] and msglist₁, msglist₂ not sharing a prefix, it must be that states[pid₁] and states[pid₂] also cannot share a prefix. This will be important in the function generateReport:

**Proof: J outputs a verdict in generateReport**

We prove that J outputs a verdict for pid₁ or pid₂ in generateReport. Let G₁, G₂ be the latest submitted hashgraphs of pid₁ and pid₂, respectively. J recalculates the message lists for pid₁ and pid₂ by running Halg with input G₁ and G₂. Since ω ∉ α, G₁ and G₂ must satisfy basic graph correctness. Furthermore, the proof of Lemma 3 demonstrates that G₁, G₂ are consistent.

---

⁸We stress that the additional verdicts are not necessary in order to prove that J is complete w.r.t. node-consistency.
hashgraphs. Notice that because $|\text{forkingNodes}| < \frac{1}{2}n$, it holds in particular true that $G_1 \cup G_2$ has less than $\frac{1}{2}n$ forking nodes. Let $msglist_1, msglist_2$ be the message lists $J$ recalculates with input $G_1$ and $G_2$, respectively. By Lemma 10 we conclude that $msglist_1, msglist_2$ share a prefix ($\ast$), but we know states$[pid_1], states[pid_2]$ do not have a common prefix. Thus, states$[pid_1] \not\subseteq msglist_1$ or states$[pid_2] \not\subseteq msglist_2$. Hence, $J$ outputs a verdict for $pid_1$ or $pid_2$, and we conclude $J$ is complete w.r.t. node-consistency in all but a negligible amount of runs of $Q$.

**Remark**

There are many possible outcomes of $\text{generateReport}$: $pid_2$ is not necessarily misbehaving, $J$ may output a verdict for both participants, or $J$ may even output additional verdicts for participants other than $pid_1$ and $pid_2$. But we stress that at least either $pid_1$ or $pid_2$ is blamed, as the above proof demonstrates. The function $\text{generateReport}$ can blame arbitrary many participants: Instead of recalculating the states for $pid_1$ and $pid_2$ only, the functions compares all known states of participants to the recalculated state in $\text{generateReport}$. If their is any discrepancy, i.e., the known state is not a subset of the recalculated state, then $J$ blames the corresponding participant. By Lemma 10, all recalculated states in $\text{generateReport}$ share a prefix. In particular applies for all nodes $pid$ that are not blamed in $\text{generateReport}$, states$[pid] \subseteq msglist$, where $msglist$ is the recalculated state of $pid$. Thus, $\text{msglist}_{\text{max}}$ is the longest state of all nodes, not violating consistency, after $\text{generateReport}$ finishes; these are all nodes that are not in the set consistencyVerdicts. Next, we show that $J$ outputs additional verdicts for node-consistency violations.

**Additional verdicts**

Let $pid_1, pid_2$ be two nodes that violate node-consistency at some point $|\omega|$ in the run $\omega$. Again, let $|\omega|$ be the first point in $\omega$ where $J$ received both message lists, not sharing a prefix, of $pid_1$ and $pid_2$ without doing any further processing. We again assume that $\text{forkingNodes} < \frac{1}{2}n$ and $pid_1, pid_2$ have not been blamed by $J$ for any consistency violation until $|\omega|$, i.e., $pid_1, pid_2 \not\in \text{consistencyVerdicts}$. The only difference to the proof above is that we now allow possible node-consistency violations of other nodes prior to $|\omega|$. With one exception, the proof is analogous: In the former proof, we easily observed that $\text{msglist}_{\text{max}}$ must be the longest state of all nodes since there was no previous node-consistency violation (see above); but this may not be true in this case. However, because of the implementation of $\text{generateReport}$, it still applies that $\text{msglist}_{\text{max}}$ is the longest state of all nodes that have not violated consistency at $|\omega|$. Hence, states$[pid_1], states[pid_2] \subseteq \text{msglist}_{\text{max}}$ is again fulfilled upon $|\omega|$. The remaining proof is completely analogous and $J$ is guaranteed to blame $pid_1$ or $pid_2$.

**Conclusion.** We have succesfully proved that $J$ ensures $\Phi_3$-accountability for the system $\{E, A, p^H\}$ with parameters as in Definition 3, by showing that $J$ satisfies both fairness and completeness except for a negligible amount of runs, i.e., $\Pr[Q(1^n) \mapsto \{(J : \psi) \mid |\omega| \neq \psi\}]$ and $\Pr[Q(1^n) \mapsto \neg(J : \Phi_3)]$ are negligible as a function in $n$. This concludes the proof.

## 5.5 Closing Discussion

In this section we mainly discuss some strengths and weaknesses of our iUC model for hashgraph, specifically concerning consistency.
First, all our security results depend on an ideal certificate authority functionality $\mathcal{F}_{\text{cert}}$. Similar to the work by Graf et al. in [10], it can be shown that all results carry over if $\mathcal{F}_{\text{cert}}$ is replaced by its realization protocol, where signing is again done using an EUF-CMA secure signature scheme.

**What $J$ models in reality.** In our model, (honest) nodes report their hashgraph to $J$, which then uses this information to render verdicts. Additionally, $J$ collects auxiliary evidence, in from of states from nodes, to capture violations of consistency. In a real run of the hashgraph protocol, this corresponds to the situation where a client finds an inconsistency between the states of different nodes. That is, some nodes have a divergent order of messages. We capture this property in our definition for node-consistency.

Upon determining such an inconsistency, nodes would have to provide their hashgraph, which they have used to calculate their last state. These evidences are then given to other parties (e.g. other nodes or independent third parties [10]) to determine which party is responsible for the inconsistency; this is done by following the judging procedures as defined in our iUC model. Concretely, evidences (the hashgraphs and message lists) are first checked whether they fulfill basic correctness in order to prevent malicious participants to escape a verdict by providing malformed evidence. Secondly, since we do not restrict our model to less than $\frac{1}{3}$ nodes being dishonest, it must be assured that the union of all submitted hashgraphs contains less than $\frac{1}{3}$ forking nodes before making any judgements w.r.t. consistency. Only if the previous two conditions are fulfilled, the judging procedure for consistency is executed. However, we proved that in any case at least one node will always be determined to have misbehaved. Besides, if one of the nodes withholds the requested hashgraph, it is trivially guilty for disrupting the protocol [10].

**Determining a consistency violation.** In the judging procedure, recognition of consistency violations is exclusively determined by means of submitted message lists of clients (i.e., the states of nodes). No additional data, in form of hashgraphs, from nodes is required before a consistency violation is detected. This is indispensable in a real run of the protocol because it allows clients or nodes to determine a violation of consistency by following the judging procedure in $\mathcal{F}_{\text{judge}}^H$. The function call generateReport corresponds the situation where a consistency violation is detected, and nodes have to submit their hashgraphs as evidence; this situation was described in the previous paragraph.

**Relaxing requirements for real-world employments.** By the occurrence of a consistency violation, nodes have the burden of proof. That is, an honest node must put forward valid evidence in form of a hashgraph that can be used to recalculate its last ordering of messages and thus justify its state. If an honest node is unable to provide such a hashgraph, it is guilty and will accordingly be blamed in our model. In reality, this is a comparatively strict requirement: In a real implementation of the Hashgraph algorithm, nodes may not store the complete hashgraph and may only keep unordered events (i.e., events with no consensus position in the total order of events). However, nodes may agree out of band on the order of the first $m$ messages (more specifically, the ordering of message list entries). Since message list entries also contain the eventID of the event in which the message was included, nodes may also agree on the order of the first $k_m$ events. By this, nodes must agree on the total order of all messages included in these $k_m$ events. Therefore, nodes can report pruned hashgraphs without these $k_m$ events, for which consensus is already achieved. Besides some
formal inconsistency problems (such as the genesis event or the correct message list entry order), all proved results should hold true for such pruned hashgraphs. In particular, the Hashgraph algorithm should output the same order of events, for which consensus is not yet reached, for all equally pruned hashgraphs of honest nodes. Herewith, it can be again checked, following the judging procedure in generateReport, if all disputed subsets of states can be recalculated with the pruned hashgraphs. Therefore, misbehaving participants can be again undisputedly identified.
6 Conclusion

In this work, we successfully applied the accountability framework from Küsters et al. in [14] to demonstrate that the Hashgraph protocol fulfills accountability w.r.t. consistency, i.e., individual nodes of a hashgraph instance can be rightfully hold accountable for misbehavior. To achieve this, we constructed in Section 3.2 an iUC model of the Hashgraph protocol with the extension of the $F_H^{\text{judge}}$ protocol. We presented in Section 5.4 a rigorous proof that individual participants, violating consistency, can be identified and blamed. Moreover, due to the fairness property of the accountability framework, we conclude that honest nodes, following the hashgraph protocol, will never be accused of misbehavior by our judging procedure. Our proof for accountability w.r.t. consistency relies on three cornerstones:

1) the possibility to hold nodes accountable w.r.t. basic correctness which prevents participants (nodes and clients) to submit malformed evidence,
2) the ability to identify forking nodes, and
3) the consistency of the hashgraph algorithm, i.e., honest nodes with different hashgraphs will assign events (eventually) the same consensus position in the total order of events.

1) In Section 5.1, we demonstrated that the Hashgraph algorithm satisfies basic correctness w.r.t. consistency. This result enables us to enforce structural properties of submitted hashgraphs and message lists. Particularly, the basic graph correctness property of hashgraphs is an indispensable prerequisite for almost all proofs in our work.

2) Secondly, we proved in Section 5.2 that Hashgraph is accountable w.r.t. fork-freeness. Furthermore, our implementation of $F_H^{\text{judge}}$ is able to output multiple verdicts for different nodes, violating fork-freeness. This is of exceptional importance because it nullifies the precondition of having more than $\frac{2}{3}n$ honest nodes. Therefore, we were able to prove accountability w.r.t. consistency of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}^H\}$ without assuming a supermajority of honest nodes. We emphasize that our results for accountability w.r.t. consistency would not hold without the detection of forking nodes.

3) Lastly, we put forward in Section 5.3 a complete proof that illustrates the consistency of the Hashgraph algorithm. This result is captured by Theorem 2. By that, we proved that the Hashgraph algorithm will not assign two different consensus positions of an event for two hashgraphs, satisfying basic graph correctness with less than $\frac{1}{3}n$ forking nodes, as input. This result is essential to show that our judging procedure for consistency is both fair and complete. Our proof is considerably more detailed compared to the proof presented in [2] by Leemon Baird. Therefore, we believe our proof is also of interest independent to our work for accountability.

In Section 5.4 we illustrated, using the three results above, that Hashgraph does achieve accountability w.r.t. consistency. Moreover, we demonstrated in our approach that the judging procedure is able to output multiple verdicts, as long as less than $\frac{1}{3}n$ forking nodes are detected. Furthermore, we illustrated how our judging procedure could be employed for real-world applications of the
6 Conclusion

Hashgraph algorithm, and discussed potential problems for real-world usages. We also demonstrated, without proving it formally, how these problems can be mitigated if participants agree on a prefix of the list of ordered messages.
Bibliography


Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature