Verifying the Hashgraph Consensus Algorithm

Karl Crary

Carnegie Mellon University

Abstract

The Hashgraph consensus algorithm is an algorithm for asynchronous Byzantine fault tolerance intended for distributed shared ledgers. Its main distinguishing characteristic is it achieves consensus without exchanging any extra messages; each participant's votes can be determined from public information, so votes need not be transmitted.

In this paper, we discuss our experience formalizing the Hashgraph algorithm and its correctness proof using the Coq proof assistant. The paper is self-contained; it includes a complete discussion of the algorithm and its correctness argument in English.

1 Introduction

Byzantine fault-tolerance is the problem of coordinating a distributed system while some participants may maliciously break the rules. Often other challenges are also present, such as unreliable communications. The problem is at the center of a variety of new applications such as cryptocurrencies. Such applications rely on *distributed shared ledgers,* a form of Byzantine fault-tolerance in which a set of transactions are assigned a place in a globally-agreed total order that is *immutable.* The latter means that once a transaction enters the order, no new transaction can enter at an earlier position.

A distributed shared ledger makes it possible for all participants to agree, at any point in the order, on the current owner of a digital commodity such as a unit of cryptocurrency. A transaction transferring ownership is valid if the commodity's current owner authorizes the transaction. (The authorization mechanism—presumably using a digital signature—is beyond the scope of the ledger itself.) Because the order is total, one transaction out of any pair has priority. Thus we can show that a commodity's chain of ownership is uniquely determined. Finally, because the order is immutable, the chain of ownership cannot change except by adding new transactions at the end.

Algorithms for Byzantine consensus (under various assumptions) have existed for some time, indeed longer than the problem has been named [12, 9]. Practical algorithms are more recent; in 1999, Castro and Liskov [6] gave an algorithm that when installed into the NFS file system slowed it only 3%. As Byzantine consensus algorithms have become more practical, they have been tailored to specific applications. Castro and Liskov's algorithm was designed for fault-tolerant state machine replication [13] and probably would not perform well under the workload of a distributed shared ledger.

However, in the last few years there have arisen Byzantine fault-tolerance algorithms suitable for distributed shared ledgers, notably HoneyBadgerBFT [10], BEAT [7], and—the subject of this paper—Hashgraph [2]. Moreover, the former two each claim to be the first practical *asynchronous* BFT algorithm (with different standards of practicality). Hashgraph does not claim to be first, but is also practical and asynchronous.

In parallel with that line of work has been the development of distributed shared ledgers based on *proof of work,* beginning with Bitcoin [11]. The idea behind proof of work is to maintain agreement on the ledger by maintaining a list of blocks of transactions, and to ensure that the list does not become a tree. To ensure this, the rules state that (1) the longest branch defines the list, and (2) to create a new block, one must first solve a mathematical problem that takes the list's old head as one of its inputs. The problem's solution is much easier to verify than to obtain, so when one learns of a new block, one's incentive is to restart work from the new head rather than continue work from the old head.

Bitcoin and some of its cousins are widely used, so in a certain sense they are indisputably practical. They are also truly permissionless, in a way that the BFT algorithms, including Hashgraph, cannot quite claim. Nevertheless, they offer severely limited throughput. Bitcoin is limited to seven transactions per second and has a latency of one hour, while its BFT competitors all do several orders of magnitude better. Proof-of-work systems are also criticized for being wasteful: an enormous amount of electricity is expended on block-creation efforts that nearly always fail. Finally—more to the point of this paper—the theoretical properties of proof of work are not well understood.

The Hashgraph consensus algorithm is designed to support high-performance applications of a distributed shared ledger. Like the other BFT systems, it is several orders of magnitude faster than proof of work. Actual performance depends very much on configuration choices (*e.g.,* how many peers, geographic distribution, tradeoff between latency and throughput, etc.), but in all configurations published in Miller, *et. al* [10] (for HoneyBadgerBFT) and Duan, *et al.* [7] (for BEAT), the Hashgraph algorithm equals or exceeds the published performance figures [4]. A frequently cited throughput goal is to equal the Visa credit-card network. According to Visa's published figures, Hashgraph can handle Visa's average load[1] and is in the ballpark of Visa's claimed surge capacity.[2]

The key to the Hashgraph algorithm's performance is achieving nearly zero communications overhead. Previous BFT systems exchange messages to achieve consensus, but Hashgraph does not. In Hashgraph, no peer sends any messages it did not intend to send anyway. Moreover, the overhead in each message it does send is light; each message consists mostly of transactions that are new to the recipient. In addition to the transaction payload, a message contains an array of sequence numbers (to keep track of which blocks each peer has seen), and the information needed by consensus: just two hashes, a digital signature, and a timestamp.

In the Hashgraph algorithm, peers achieve consensus by voting, but each vote is fully determined by publicly available information, so peers can determine each other's votes without communicating with them. This allows the consensus election to be carried out *virtually,* with no extra messages exchanged. There is no magic here; it still takes multiple cycles of communication to achieve consensus, but every message is just an ordinary block filled with new transactions. Thus, consensus is essentially invisible from a throughput perspective.

Another advantage of the Hashgraph algorithm is it requires no sophisticated cryptography. The only requirements are digital signatures and cryptographic hashes, both of which are now commonplace and have highly optimized implementations available.

This work The Hashgraph consensus algorithm has been realized as a large-scale, open-access, commercial system called Hedera [3], so there is considerable interest in machine-verifying that it is correct. In this paper, we discuss the first steps toward doing so. Using the Coq proof assistant [5], we formalized the batch-mode algorithm given in Baird [2] and developed a machine-checkable proof of its correctness. As usual when one formalizes a human-language proof, we found a few errors, but they were minor and easily corrected. The algorithm implemented in Hedera is an online algorithm, inspired by the batch-mode algorithm discussed here, but obviously a bit different.[3] We will discuss some of the differences in Section 5.

We begin by giving an informal overview of the algorithm, to build intuition. Then we give a human-language mathematical definition of the algorithm and prove its properties, in Section 2. We discuss the formalization in Coq starting in Section 3.

1.1 Hashgraphs in overview

A *hashgraph* is a directed graph that summarizes who has said what to whom. Each peer maintains a hashgraph reflecting the communications it is aware of. In general, each peer knows a different subset of the true graph, but because of digital signatures and cryptographic hashes, they cannot disagree about the information they have in common.

The nodes of a hashgraph are *events.* Each event is created by a particular peer. Except for each peer's initial event, each event has two parents. One parent has the same creator (we call that one the *self-parent*), and the other has a different creator (we call it the *other-parent*). Honest peers do not create *forks*, where a fork is defined as two events with the same creator in which neither is a self-ancestor of the other.[4] In other words, the events created by an honest peer will form a chain.

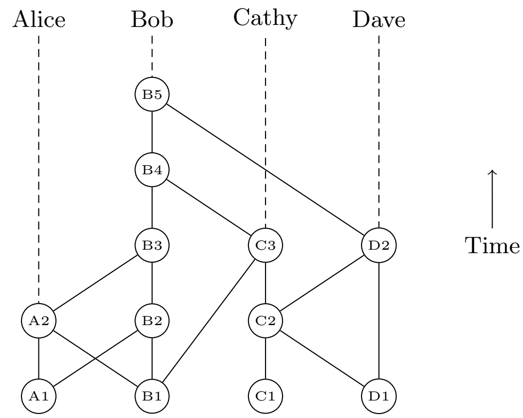

Figure 1: A Hashgraph

We can visualize a hashgraph as shown in Figure 1. In this example, all peers are behaving honestly.

In the Hashgraph network, each peer periodically chooses another at random and sends that peer its latest event. The recipient then creates a new event, with its own latest event as self-parent and the event it just received as other-parent. Every event is digitally signed, so there can be no dispute over who created it. Each event also contains the hashes of its two parents, so there is no dispute over parentage either. The recipient of the event will request from the sender any of the event's ancestors that it does not have. (For simplicity, we will ignore that part of the protocol in what follows.) The recipient also puts a timestamp into the event, which is ultimately used to determine a consensus timestamp for other events.

Finally, each event contains a payload of transactions. When a peer wishes to inject a new transaction into the network, it stores it in a buffer of outgoing transactions. The next time it creates an event (this happens multiple times per second), it uses the contents of its buffer as the new event's payload. Transactions are just along for the ride in the consensus algorithm, so we will discuss them little.

In the example, Dave sent D1 to Cathy, resulting in Cathy creating C2. Then Cathy sent C2 back to Dave, resulting in Dave creating D2. Bob sent B1 to both Alice and Cathy, resulting in A2 and C3. At about the same time, Alice sent A1 to Bob, resulting in B2. Alice sent A2 to Bob, resulting in B3. Cathy sent C3 to Bob, resulting in B4. Finally, Dave sent D2 to Bob, resulting in B5.

The algorithm partitions events into rounds, in a manner that is easy for all peers to agree on. The first event created by a peer in a round is called a *witness.* (Note that dishonest peers may have multiple witnesses in a single round.) A witness that is quickly propagated to most peers is called *famous.* Identifying the famous witnesses is the main job of the consensus algorithm.

Each round, the algorithm selects one famous witness from each peer that has one to be a *unique famous witness.*

[1] 3200 transactions per second in 2015 [14].

[2] 65,000 transactions per second in 2017 [15].

[3] As it happens, none of the minor errors we found appear to affect the implemented online algorithm.

[4] A self-ancestor is an ancestor using only self-parent edges.

An honest peer will have at most one witness per round, so if that witness turns out to be famous, it will also be unique. If a dishonest peer happens to have multiple famous witnesses, one of them is selected.

We say that an event has been *received by the network* in the first round in which it is an ancestor of all the unique famous witnesses. The round received is the primary determiner of an event's place in the order. Ties are broken using a consensus timestamp that is computed using the unique famous witnesses. Any remaining ties are broken in an arbitrary but deterministic way. Finally, the ordering of transactions is determined by the ordering of the events in which they reside, with transactions in the same event ordered by their position in the payload.

2 The Algorithm

We will begin by reviewing the Hashgraph consensus algorithm [2], and defer discussion of the formalization until Section 3. We will develop the algorithm in pieces, establishing the properties of those pieces as we go. But first, the algorithm relies on the following assumptions:

1. The network is asynchronous. However, the adversary cannot disconnect honest peers indefinitely; every honest peer will eventually communicate with every other honest peer.
2. Every peer can determine any event's creator and parents. In the real world, this assumption means that the adversary has insufficient computing power to forge a digital signature, or to find two events with the same hash.
3. A supermajority (defined to mean more than two-thirds) of the peers are honest.
4. The coin mechanism discussed below satisfies a probability assumption. (Section 2.4.1.)
5. Various uncontroversial mathematical assumptions hold. These are discussed in Section 3. For example, the parent relation is well-founded. These assumptions would ordinarily go unremarked, but formal verification requires that they be specified.

For convenience, we will also assume that there are at least two peers. (If there is only one peer, the consensus problem is trivial.)

Suppose x and y are events. We will write $x \leq y$ when x is a (non-strict) ancestor of y, and $x < y$ when x is a strict ancestor of y. We say x is a self-ancestor of y (and write $x \sqsubseteq y$) when x is an ancestor of y using only self-parent edges, and we write $x \sqsubset y$ for the strict version. In all cases, note that the older event is on the left.

We will refer to events created by an honest peer as *honest events.* We will refer to a set of events whose creators constitute a supermajority of the peers as a *supermajor set.* We will take N to be the total number of peers.

2.1 Seeing

Recall that events x and y form a *fork* if they have the same creator, and if $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$. Note that forks are defined so that no event forms a fork with itself. The main property of honest peers is they never create forks.

We say that y *sees* x (written $x \trianglelefteq y$) if (1) $x \leq y$ and (2) there does not exist any fork z, z' such that $z, z' \leq y$ and $\mathsf{creator}(x) = \mathsf{creator}(z)$. Sees is the same as ancestor except that when an event observes a fork, it blacklists the fork's creator and will see none of the fork's creator's events.

This brings us to the algorithm's main technical concept:

Definition 2.1 We say that y *strongly sees* x (written $x \ll y$) if there exists a supermajor set Z, such that for all $z \in Z$, $x \trianglelefteq z \leq y$.

Informally, for y to strongly see x means that x has made its way to most of the network, within the subgraph visible to y, with hardly anyone observing x's creator cheat.

For example, consider Figure 1. Every peer is behaving honestly, so every ancestor is seen. Then we can say that B4 strongly sees B1, using A2, B4, and C3 as intermediaries. (B4 has no intermediary on Dave, but it does not need one. Three intermediaries is enough, since $3 > {}^{2}\!/_{3} \cdot 4$.) B4 also strongly sees D1, using B4, C3, and D1 as intermediaries. B4 does not strongly see A1 or C1, as it has only two intermediaries for each. However, B5 does strongly see C1, using intermediaries B5, C2, and D2. In most of these cases, other choices could be made for the intermediaries as well.

We can state some useful properties of strongly-seeing:

Lemma 2.2

1. *If $x \ll y$ then $x < y$.*
2. *If $x \sqsubseteq y \ll z$ then $x \ll z$.*
3. *If $x \ll y \leq z$ then $x \ll z$.*

Proof

For (1), suppose $x \ll y$. We can immediately see that $x \leq y$, but we also have $x \neq y$. Since y strongly sees x, there must be intermediaries on a supermajority of peers, at least one of which must not be y's creator. (Since we assume there are at least two peers, a supermajority is at least two.) Let z be one such. Then $x \trianglelefteq z < y$, so $x < y$.

For (2), observe that $x \sqsubseteq y \trianglelefteq z$ implies $x \trianglelefteq z$, since x and y have the same creator, so any fork on x's creator is a fork on y's creator. (3) is immediate by transitivity of ancestor.

We can now state the main technical lemma, which states that at most one side of a fork can be strongly seen, even by different events:

Lemma 2.3 (Strongly Seeing) *Suppose $x \ll v$ and $y \ll w$. Then x and y do not form a fork.*

Proof

Let $Z = \{z \mid x \trianglelefteq z \leq v\}$ and $Z' = \{z' \mid y \trianglelefteq z' \leq w\}$. By the definition of strongly-sees, Z and Z' are supermajor sets. Also recall that a supermajority of peers are honest. Any three supermajorities have an element in common. Therefore there exist events $z \in Z$ and $z' \in Z'$ such z and z' share the same creator, and that creator is honest. Since their creator is honest, either $z \sqsubseteq z'$ or $z' \sqsubseteq z$. Assume the latter. (The former case is similar.) Then $x, y \leq z$. Thus x and y cannot form a fork, since $x \trianglelefteq z$.

The definition of strongly-sees limits the influence of dishonest peers, since at most one side of a fork can be strongly seen. This is helpful in various ways; one important way is it prevents dishonest peers from obtaining extra votes in an election by creating extra witnesses, since votes must be strongly seen to count.

2.2 Rounds

Every event is assigned to a *round.* (Note that the round an event belongs to is different from the round that it is received by the network.) Initial events belong to round zero. For a non-initial event x, let i be the maximum of x's parents' rounds. Then x belongs to round i, unless x strongly sees events in round i on a supermajority of peers, in which case x belongs to round $i + 1$.

For example, consider Figure 1. A1, B1, C1, and D1 are initial events, so they belong to round 0. For every other event, the maximum of the parents' rounds is 0. Thus, every other event also belongs to round 0, except B5. As noted above, B4 can strongly see B1 and D1, but not A1 or C1, so it remains in round 0. However, B5 strongly sees B1, C1, and D1, so it advances to round 1.

The first event in a round on each peer (or first events, in the case of a dishonest peer) are called *witnesses.* Witnesses are the events that can cast votes in an election.

In Figure 1, the witnesses are A1, B1, C1, and D1 (round 0), and B5 (round 1).

Note that if a peer has not heard from anyone in a while, it is possible for it to skip one or more rounds when it catches up. Thus, a peer might not have a witness in any particular round. Conversely, a dishonest peer can have multiple witnesses in a round, but (by Lemma 2.3) at most one of them can be strongly seen.

Two important properties of rounds follow more-or-less directly:

Lemma 2.4 *Suppose $i < j$ and x belongs to round j. Then there exists a supermajor set of round i witnesses W, such that for all $w \in W$, $w \ll x$.*

Lemma 2.5 *Suppose x and y are events. If there exists a supermajor set W such that for all $w \in W$, $x \leq w \ll y$, then y's round is strictly later than x's round.*

The main property we wish to establish about rounds is *progress,* which says that every round is inhabited.

Lemma 2.6 (Broadcast) *Suppose x is an honest event. Then there exists an honest event y such that, in its ancestry, x has reached every honest peer. That is, for every honest peer a, there exists z created by a, such that $x \leq z \leq y$.*

Proof Sketch

By induction on the number of honest peers, using the assumption that every honest peer eventually communicates with every other honest peer.

Lemma 2.7 (Progress) *Every round is inhabited.*

Proof

We prove the stronger property that for every round i, there exists an inhabitant of some round $j \geq i$. The inhabitant can then be wound back to i using Lemma 2.4.

The proof is by induction on i. The case $i = 0$ is immediate, so suppose $i > 0$. By induction there exists an inhabitant x of round $i - 1$. Using Lemma 2.6, there exists y such that x has reached every honest peer in y's ancestry. And again, there exists z such that y has reached every honest peer in z's ancestry.

Suppose a is an honest peer. We show that there exists v created by a that $x \leq v \ll z$. Since the honest peers constitute a supermajority, it follows by Lemma 2.5 that z's round is later than $i - 1$, and is thus at least i.

By the specification of y, there exists v created by a such that $x \leq v \leq y$. It remains to show $v \ll z$. For every honest peer b, by the specification of z there exists w created by b such that $y \leq w \leq z$. Since a is honest, $v \trianglelefteq w \leq z$. Since the honest peers constitute a supermajority, $v \ll z$.

2.3 Voting

One cannot use the raw witnesses to determine the round an event was received by the network, because one can never be sure that one has observed all the witnesses. Instead, we will use the notion of a *famous* witness—eventually everyone will know all the famous witnesses in any given round. The main function of the consensus algorithm is to determine which witnesses are famous.

Fame is determined by an election. Votes are cast by witness events, not by peers. This is important because one peer has no way of knowing what another *peer* knows. In contrast, if a peer is aware of an *event* at all, it knows that event's entire ancestry. Thus, if voting is deterministic, each witness's vote can be determined by everyone who is aware of the witness, without any additional communication. However, votes will be "collected" only from strongly-seen witnesses, so there can be at most one meaningful voter per peer per round.

We will refer to the witness whose fame is being determined as the *candidate,* and each witness that is casting votes on the candidate's fame as the *voter.* The election begins d rounds after the candidate's round, where d is a parameter that is at least 1. Thus if the candidate belongs to round i, the voters in the first round of the election belong to round $i + d$.

In the first round of an election, each voter will vote yes if the candidate is among its ancestors. (In essence, you think someone is famous if you have heard of them.) In successive rounds, each voter votes the way it observed the majority vote in the previous round.

We say that a round of an election is *nearly unanimous* if voters on a supermajority of peers vote the same way. (Note that this is a stronger condition than merely a supermajority of the voters, since some peers might not be voting.) If voter ever observes a nearly unanimous result in the previous round, then the vote in the current round will be unanimous. Clearly, once the election becomes unanimous, it will stay so. Thus we can end the election as soon as any event observes a nearly unanimous result the previous round.

Coins Under normal circumstances, this process will come to consensus quickly, but an adversary with sufficient control of the network can prevent it. Deterministically, the problem is insurmountable [8], but, as usual, it can be solved with randomization.

Every c rounds (a parameter at least $d + 3$), the election will employ a coin round: Every voter who sees a nearly unanimous result the previous round will continue to vote with the majority. However, the remaining voters will determine their votes using a coin flip. All the voters who saw a nearly unanimous result will certainly vote the same way, and eventually—by chance—the coin flippers will also vote that way. The following round, every voter will see a unanimous result and the election will end.[5]

Definition 2.8 (Voting) *Suppose x is a round i witness and y is a round j witness, with $i + d \leq j$. Then* $\mathsf{vote}(x, y)$ *(that is, y's vote on x's fame) and* $\mathsf{election}(x, y, t, f)$ *(that is, the votes on x's fame observed by y are t yeas and f nays) are defined simultaneously as follows:*

- *If $i + d = j$ then* $\mathsf{vote}(x, y)$ *is* yes *if $x \leq y$, and* no *otherwise.*
- *If $i+d < j$ and $(j-i) \bmod c \neq 0$ and* $\mathsf{election}(x, y, t, f)$, *then* $\mathsf{vote}(x, y)$ *is* yes *if $t \geq f$ and* no *otherwise.*
- *If $i+d < j$ and $(j-i) \bmod c = 0$ and* $\mathsf{election}(x, y, t, f)$, *then*

$$\mathsf{vote}(x,y) = \begin{cases} \mathsf{yes} & t > 2/3 \cdot N \\ \mathsf{no} & f > 2/3 \cdot N \\ \mathsf{coin}(y) & \text{otherwise} \end{cases}$$

- *If $i + d < j$, then* $\mathsf{election}(x, y, t, f)$ *holds if and only if:*

$$\begin{array}{rcl} W & = & \{\text{round } j-1 \text{ witnesses } w \text{ such that } w \ll y\} \\ t & = & |\{w \in W \mid \mathsf{vote}(x,w) = \mathsf{yes}\}| \\ f & = & |\{w \in W \mid \mathsf{vote}(x,w) = \mathsf{no}\}| \end{array}$$

In the above definition, $\mathsf{coin}(y)$ is a pseudo-random coin flip computed by drawing a bit from the middle of a cryptographic hash of y. It is important that the coin flip is pseudo-random, not truly random, so that other peers can reproduce it.

Definition 2.9 (Decision) *Suppose x is a round i witness and y is a round j witness, where $i + d < j$ and j is not a coin round (that is, $j - i \bmod c \neq 0$). Suppose further that* $\mathsf{election}(x, y, t, f)$. *Then y decides β on x's fame (written* $\mathsf{decide}(x, y, \beta)$*), if $t > 2/3 \cdot N$ and $\beta =$* yes, *or if $f > 2/3 \cdot N$ and $\beta =$* no.

The outcome of the election is determined as soon as any peer decides, but other peers might not realize it right away.

2.4 Consensus

For the algorithm to work, we require four properties:

1. Decisions are pervasive: once one peer decides someone's fame, that decision propagates to every other peer. (Corollary 2.11.)
2. Every round will have at least one famous witness. (Theorem 2.13.)
3. Late arrivals are not famous: if a witness is not well disseminated within $d+1$ rounds (the earliest a decision can be made), it will not be famous. (Corollary 2.15.)
4. Termination: eventually every witness will have its fame decided. (Theorem 2.17.)

Lemma 2.10 (Decision-Vote Consistency) *Suppose w, x, and y are witnesses, where x and y belong to the same round. If* $\mathsf{decide}(w, x, \beta)$ *and* $\mathsf{vote}(w, y) = \beta'$ *then $\beta = \beta'$.*

[5] This is an important theoretical property, but as a practical matter, in the unlikely event an adversary has enough control over the network to force coin rounds, it will be able to grind the algorithm to a halt for rounds exponential in the number of peers.

Proof

Let j be the round of x and y. Observe that $\mathsf{vote}(w, y)$ is given by the second case of the definition of voting. Unpacking the definitions, let $\mathsf{election}(w, x, t, f)$ and $\mathsf{election}(w, y, t', f')$. Let $V_z = \{\text{round } j - 1 \text{ witnesses } v \text{ such that } v \ll z\}$.

Suppose $\beta =$ yes. (The other case is similar.) Then $|T| > 2/3 \cdot N$ where $T = \{v \in V_x \mid \mathsf{vote}(w, v) = \mathsf{yes}\}$. Thus T is a supermajor set. By Lemma 2.4, V_y is also a supermajor set. Observe that T and V_y are both sets of round $j - 1$ witnesses. Any two supermajorities of the same set have a majority in common. Thus a majority of V_y agrees with T, but T all vote yes. Hence $t' > f'$ so $\beta' =$ yes.

Note that the proof illustrates why decisions cannot be made in coin rounds. In a coin round, y would have to see a supermajority to determine its vote, not merely a majority, and we cannot guarantee a supermajority. Thus the lemma would fail to hold.

Corollary 2.11 (Propagation) *Suppose x makes a decision on w. Then in every round after x's, except coin rounds, every witness decides the same way as x did.*

Proof Sketch

Let j be the round of x. By Lemma 2.10, every witness in round j votes β. It is easy to see that unanimity persists. Thus every witness in any round after j decides β, unless it is prohibited from doing so because it is a coin round.

An easy corollary is that all decisions agree:

Corollary 2.12 (Consistency) *Suppose* $\mathsf{decide}(w, x, \beta)$ *and* $\mathsf{decide}(w, y, \beta')$. *Then $\beta = \beta'$.*

Proof

Let z be a witness in a later round than x and y that is not a coin round. (Such a witness exists by Lemma 2.7.) By Corollary 2.11, $\mathsf{decide}(w, z, \beta)$ and $\mathsf{decide}(w, z, \beta')$. Hence $\beta = \beta'$.

Theorem 2.13 (Existence) *Every round has at least one famous witness.*

Proof

If S is a set of events, define $\mathsf{backward}(j, S) = \{w \mid w \text{ is a round } j \text{ witness and } \exists v \in S.\, w \ll v\}$.

Suppose S is an inhabited set of round k witnesses and $j < k$. By Lemma 2.4, $2/3 \cdot N < |\mathsf{backward}(j, S)| \leq N$.

Let i be arbitrary, and let z be an arbitrary witness in round $i + d + 2$, then let:

$$\begin{array}{rcl} S_3 & = & \{z\} \\ S_2 & = & \mathsf{backward}(i+d+1, S_3) \\ S_1 & = & \mathsf{backward}(i+d, S_2) \\ S_0 & = & \mathsf{backward}(i, S_1) \end{array}$$

Now consider a bipartite graph between S_0 and S_1, where there is an edge between $x \in S_0$ and $y \in S_1$ if $x \leq y$. Since strongly-seeing implies ancestor, Lemma 2.4 tells us that every event in S_1 is the terminus of more than $2/3 \cdot N$ edges. Thus there are over $2/3 \cdot N \cdot |S_1|$ edges total. Since there are at most N events in S_0, by the pigeonhole principle there is at least one event in S_0 that is the origin of over $2/3 \cdot |S_1|$ edges. Let x be such an event. We will show that x is famous.

By the specification, x is an ancestor of over two-thirds of the events in S_1. Thus, over two-thirds of S_1 will vote yes. We claim that every event in S_2 will vote yes. It follows that z will decide yes. (We know $i+d+2$ will not be a coin round since $c \geq d+3$ by assumption.)

Suppose $y \in S_2$. The events sending votes to y are exactly $\mathsf{backward}(i+d,\{y\})$ and, as above, $2/3 \cdot N < |\mathsf{backward}(i+d,\{y\})|$. Thus, the events in S_1 voting yes and the events in S_1 sending votes to y are both supermajor, so they have a majority in common. Thus y will see more yeas than nays, and will vote yes itself.

Lemma 2.14 *If* $\mathsf{vote}(x,y) = \mathsf{yes}$ *then* $x \leq y$.

Proof Sketch

By well-founded induction on y using the strict-ancestor order ($<$). In the first voting round, a voter will vote yes only if it is a descendant of x. Thereafter, a voter cannot vote yes without receiving some yes votes (half in regular rounds and a third in coin rounds), and one only receives votes from ancestors.

Corollary 2.15 (No Late Fame) *Suppose x is a round i witness and y is a round j witness, where $i+d<j$ and $(j-i)$ mod $c \neq 0$. If $x \not\leq y$ then* $\mathsf{decide}(x,y,\mathsf{no})$.

Proof

Any voter sending votes to y must be an ancestor of y, and therefore it cannot be a descendant of x. Thus, by Lemma 2.14, y will see only no votes. Since j is a round in which one is permitted to decide, y will decide no.

2.4.1 Termination

The delicate aspect of the termination proof is the probability assumption. There are two things going in the probability assumption:

The first is just a fact from probability. With probability one, a sequence of independent random variables, each with a finite range (in our case an N-tuple of booleans), will eventually hit a target value, provided that target is specified in such a way as makes reference only to earlier values in the sequence.[6]

The second is more subtle. We assume that the above holds even though coin flips are not actually random. That is, we assume that the coin flip, which is actually pseudorandom, behaves on honest peers as though it really is random. Certainly any peer can easily dictate an event's coin flip (which is drawn from a cryptographic hash of the event) by tweaking the event's payload, but we assume honest peers will not do that. Beyond that, in principle the adversary might orchestrate the network in order to control the payloads and thereby dictate the coin flips. We assume that it has insufficient computing power to do so, or least that it cannot do so forever.

We will not reason explicitly about probability. Instead, we will simply assume that the coins eventually agree with the target value. Put more precisely, we assume there is a sample space that dictates random outcomes. Rather than equip the sample space with a measure and assume—or prove—that agreement takes place in a subset of measure 1, we will instead just restrict our attention to points in the sample space in which agreement takes place. This is formalized in Section 3.6.

Lemma 2.16 (Good Coins) *Suppose x is a round i witness. Then there exists a round $j > i$, a round j witness y, and a boolean β such that:*

1. $(j-i)$ *mod* $c = 0$,
2. $\beta = \mathsf{yes}$ *if and only if y receives as many yes votes for x as no votes, and*
3. *for all honest round j witnesses w,* $\mathsf{coin}(w) = \beta$.

Proof Sketch

We construct our sequence by looking at every c-th round starting at $i+c$. So let $r_k = i+c\cdot(k+1)$. Suppose round r_k has m_k honest witnesses. Note that m_k is at most the number of honest peers. Let X_k be the m_k tuple consisting of the coin for every honest, round r_k witness.

For every k, let y_k be an arbitrary round r_k witness. Let $\beta_k = \mathsf{yes}$ if and only if y receives as many yes votes on x than no votes. Let the target T_k be the m_k tuple in which every element is β_k.

By the probability assumption, there exists k such that $X_k = T_k$. Let $j = r_k$ and $y = y_k$ and $\beta = \beta_k$. The required properties hold by construction.

Theorem 2.17 (Termination) *For every witness x, there exists z and β such that* $\mathsf{decide}(x,z,\beta)$.

Proof

Let i be the round of x. Let j, y, and β be as given by Lemma 2.16. We claim that every round j vote will be β. Then, let z be an arbitrary round $j+1$ witness. By Lemma 2.4, z receives votes from a supermajor set, and $j+1$ is not a coin round, so $\mathsf{decide}(x,z,\beta)$.

To show the claim, suppose w is a round j witness. If w uses its coin, the result is immediate. Suppose, instead, w receives a supermajority. As we have seen before, when there is a supermajority, every witness (in particular, y) will receive a majority that agrees with the supermajority, and y receives a majority of β. Thus, the supermajority that w receives (and therefore w's vote) must be β.

At this point we know that all peers can agree on the identity of the famous witnesses: For any given witness, eventually someone will decide (Theorem 2.17). Once someone decides, everyone else will make the same decision in short order (Corollary 2.11). Once a peer has settled the fame of every witness it has heard of, it can consider itself done, since any additional witnesses it hasn't heard of are guaranteed not to be famous (Corollary 2.15). Moreover, at least one famous witness will exist (Theorem 2.13).

2.5 Round Received

At this point the real work is done. Next, we identify at most one famous witness per peer as a *unique famous witness.* If a peer has only one famous witness (as will be always be the case for honest peers), that famous witness will be unique. In the unlikely event that a peer has multiple famous witnesses, one of them is chosen to be unique in an arbitrary but deterministic manner. (Textual comparison of the data that expresses the event will do fine.)

Then, using the unique famous witnesses, we define an event's *round received:*

[6] Without the proviso, one could prevent hitting the target by just setting the target to any value other than the one obtained.

Definition 2.18 *Suppose x is an event. The* round x is received by the network *is the earliest round i for which all the round i unique famous witnesses are descendants of x.*

2.6 The Consensus Timestamp

Suppose x is an event that is received in round i. We compute the consensus timestamp for x as the median of the timestamps assigned to x by the unique famous witnesses of round i. (If there is an even number of unique famous witnesses, we take the median to be the smaller of the two central elements. This provides the useful property that the consensus timestamp is the timestamp from some particular peer.)

Suppose y is a round i unique famous witness. The timestamp y assigns to x is the timestamp of the earliest z such that $x \leq z \sqsubseteq y$. Note that z certainly exists, since $x \leq y$, and that z is uniquely defined since the self-ancestors of y are totally ordered.

2.7 The Consensus Order

We can now define the consensus order:

Definition 2.19 *Suppose x and y are events. Let i and i' be the rounds x and y are received by the network. Let t and t' be the consensus timestamps of x and y. Then x* precedes y in the consensus order *if:*

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

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

3 Formalization

Our work builds on lessons learned from a previous effort to verify the Hashgraph consensus algorithm by Gregory Malecha and Ryan Wisnesky. In that earlier effort the algorithm was expressed by implementing it as a function in Gallina (the Coq specification language). This allowed the code to be directly executed within Coq, and it was thought that such code could be more easily related to the actual code in the commercial Hashgraph implementation, Hedera.

However, the specification as a Gallina implementation was challenging to work with. Although the effort did not hit a show-stopping problem, the concreteness of the implementation made it clumsy to work with. Moreover, expressing the algorithm using recursive functions, rather than inductive relations, meant that one was denied nice induction principles.

In this work we started over with new definitions, making events and other objects of interest abstract, and using inductive relations for most definitions. This allowed for streamlined proofs that usually closely resemble the human-language proofs, and nice induction principles in most cases. (An important exception, induction over votes, is discussed in Section 3.4.)

The axioms that define hashgraphs are just 258 lines. This is the trusted part of the development, apart from Coq itself.

The full proof is 14 thousand lines of Coq (version 8.9.1), including comments and whitespace. It takes 25 seconds to check using a single core of a 1.8 GHz Intel Core i5. Not all of these lines are allocated to matters that are conceptually interesting: the largest and third-largest files (2.6k lines between them) are dedicated to defining and establishing properties of the cardinalities of sets, and medians.

We will discuss changes we made from the original algorithm in Baird [2], and then touch on interesting points that arose during the formalization. The survey here will not be complete; the complete development is available at:

`cs.cmu.edu/~crary/papers/2020/hashgraph-formal.tgz`

3.1 Changes from the original algorithm

As typically happens when formalizing a human-language proof, we did uncover a few errors, but they were minor and easily corrected.[7] These corrections are already reflected in Section 2:

1. The original definition of strongly-sees was a bit different: it said that y strongly sees x if $x \trianglelefteq z \trianglelefteq y$ (for all z in some supermajor set). But that definition didn't provide the "stickiness" property of Lemma 2.2 (that is, $x \ll y \leq w$ implies $x \ll w$) since w might observe a fork that y does not.
2. Because of that change, strongly-seeing does not necessary imply seeing. The original rule for first-round voting required that for y to vote yes on x, y must see x and not merely be a descendant of x. But that meant that y strongly seeing x was not enough for y to vote yes for x, which breaks the proof of the existence of famous witnesses (Theorem 2.13).
3. Originally, when a peer had multiple famous witnesses, instead of choosing one to consider unique, none of them were. But then it was not obvious that unique famous witnesses always exist.

In addition to these changes, we added a careful proof of Progress (Lemma 2.7). Also, a trivial difference is our first round is 0, instead of 1. This is preferable so every natural number is a round number.

3.2 Peers and Events

To illustrate the abstract style we used to formalize the algorithm, these are definitions of peers and events:

```
Parameter peer : Type.

Axiom peer_eq_dec
  : forall (a b : peer), {a = b} + {a <> b}.
```

The axiom states that it is decidable whether two peers are the same. This is a good example of the sort of uncontroversial mathematical assumption we alluded to in Section 2. A typical human-language proof would not bother to make such an assumption explicit.

[7]These errors pertain to the batch-mode algorithm discussed here, not to the online algorithm that the Hedera implementation is based on. We discuss some of the differences in Section 5. For example, the online algorithm defines strongly seeing somewhat differently.

```
Parameter number_peers : nat.

Axiom number_of_peers
  : cardinality (@every peer) number_peers.

Axiom number_peers_minimum : number_peers >= 2.
```

There exists a natural number that is the number of peers, and that number is at least two. (In the preceding we called it N.) The code "`@every peer`" denotes the set of all peers.

```
Parameter honest : peer -> Prop.

Axiom supermajority_honest
  : supermajority honest every.
```

The set of honest peers is a supermajority of the set of all peers.

```
Parameter event : Type.

Axiom event_eq_dec
  : forall (e f : event), {e = f} + {e <> f}.

Parameter creator : event -> peer.
```

The equality of events is decidable, and every event has a creator.

```
Parameter parents
  : event -> event -> event -> Prop.

Axiom parents_fun :
  forall e e' f f' g,
    parents e f g
    -> parents e' f' g
    -> e = e' /\ f = f'.

Axiom parents_creator :
  forall e f g,
    parents e f g
    -> creator e = creator g.
```

We write `parents e f g` when `e` and `f` are the self-parent and other-parent of `g`. The axioms say parents are uniquely defined, and the self-parent has the same creator as the event.

```
Definition initial (e : event) : Prop :=
  ~ exists f g, parents f g e.

Axiom initial_decide :
  forall e,
    initial e \/ exists f g, parents f g e.
```

An initial event is one without parents, and it is decidable whether an event is initial.

```
Inductive parent : event -> event -> Prop :=
| parent1 {e f g} :
    parents e f g
    -> parent e g

| parent2 {e f g} :
    parents e f g
    -> parent f g.

Definition self_parent (e f : event) : Prop :=
  exists g, parents e g f.

Axiom parent_well_founded
  : well_founded parent.
```

We say `parent x y` when `x` is either parent of `y`, and `self_parent x y` when `x` is the self-parent of `y`. We assume that the parent relation is well-founded, which means we can do induction using that relation. (In classical logic, that is equivalent to saying there are no infinite descending chains.) From that assumption, we can show that the self-parent, strict ancestor, and strict self-ancestor relations are also well-founded.

3.3 Rounds

In Baird [2] the definitions of rounds and witnesses are mutually dependent. As here, a witness was the first event created by a peer in a round. Unlike here, to advance to the next round an event would need to strongly see many *witnesses* in the current round, while we require it only to strongly see many *events* in the current round. It is not hard to see that the definitions are equivalent, but our version has the virtue that rounds can be defined without reference to witnesses.

Our formalization of rounds then is:

```
Inductive round : nat -> event -> Prop :=
| round_initial {x} :
    initial x
    -> round 0 x

| round_advance {x y z m n A} :
    parents y z x
    -> round m y
    -> round n z
    -> supermajority A every
    -> (forall a,
          A a
          -> exists w,
               creator w = a
               /\ stsees w x
               /\ round (max m n) w)
    -> round (S (max m n)) x.
```

```
| round_nadvance {x y z m n A} :
    parents y z x
    -> round m y
    -> round n z
    -> superminority A every
    -> (forall a w,
          A a
          -> creator w = a
          -> stsees w x
          -> exists i,
               i < max m n /\ round i w)
    -> round (max m n) x
```

The first case says initial events belong to round 0. In each of the other two cases, x's parents' rounds are m and n. Then, in the second case, we advance to the next round (*i.e.,* give x the round $\max(m, n) + 1$) if there exists a set of peers A, comprising a supermajority of the peers, such that for each peer $a \in A$, there exists w created by a in round $\max(m, n)$ where $w \ll x$. The third case expresses the negation of that condition. There S is a "superminority" of T if S contains at least one-third of T's elements. That is, a superminority is a subset large enough to deny a supermajority to its complement.

It is necessary to express the conditions for advancing or not advancing to the next round as a pair of affirmative properties. One might imagine saying something like:

```
...
-> (b = true
    <->
    (.. x stsees enough from this round ..))
-> round (max m n + if b then 1 else 0) x
```

but with such a definition, the "strongly sees enough" portion could not be implemented, since it could not mention rounds. Any mention of rounds in that position would be a non-positive occurrence of the relation being defined, which is not permissible in an inductive definition.

The benefit to going to the trouble to use an inductive definition is one can do induction over the derivation of an event's round. When one is reasoning about rounds, this produces exactly the cases you want: one base case, one inductive case where you advance, and one where you don't.

3.4 Voting

Expressing voting as an inductive definition is a bit tricky. We gave `vote` the type:

```
sample -> event -> event -> bool -> Prop
```

Here, `vote s x y b` means y's vote on whether x is famous is b. The s is a point in the sample space, and can be safely ignored for now. (We discuss the sample space in the Section 3.6.)

The case for the first round of voting is straightforward:

```
| vote_first {s x y m n v} :
    rwitness m x
    -> rwitness n y
    -> m < n
    -> m + first_regular >= n
    -> (Is_true v <-> x @ y)
    -> vote s x y v
```

Here, `first_regular` is the formalization's name for the parameter d, `rwitness m x` means that x is a round m witness, and `@` means strict ancestor ($<$, formalized as the transitive closure of `parent`). Then y votes yes if y is a strict descendant of x, and no otherwise, provided $\mathsf{round}(x) < \mathsf{round}(y) \leq \mathsf{round}(x) + d$.[8]

The complications begin in the next case:

```
(* not a coin round *)
| vote_regular {s x y m n t f v} :
    rwitness m x
    -> rwitness n y
    -> m + first_regular < n
    -> (n - m) mod coin_freq <> 0
    -> election (vote s x) (pred n) y t f
    -> ((t >= f /\ v = true)
          \/ (f > t /\ v = false))
    -> vote s x y v
```

The first four premises say that more than d rounds have elapsed since x, and it is not currently a coin round. Then `election (vote s x) (pred n) y t f` says that when y collects votes on x from the previous round, it receives t yeas and f nays. Then y's vote is true if $t \geq f$ and false otherwise.

The trickiness lies in `election`. It has type:

```
(event -> bool -> Prop)
  -> nat -> event -> nat -> nat -> Prop
```

The first argument abstracts over the recursive call to `vote`, in order to disentangle `election` and `vote`. We fill it in with `vote s x` to give it access to everyone's votes on x. The second argument is the round to collect votes from; we fill it in with $\mathsf{round}(y) - 1$.

An auxiliary definition, `elector n w y`, specifies the round n witnesses w that can send votes to y:

```
Definition elector (n : nat) (w y : event) :=
  rwitness n w /\ stsees w y.
```

At this point, we might imagine defining `election V n y t f` as follows:

```
cardinality
  (fun w =>
     elector n w y /\ V w true) t
/\
cardinality
  (fun w =>
     elector n w y /\ V w false) f
```

Indeed, this is a perfectly good definition. However, if we define `election` this way, `vote` is not allowed to call it as above. Since `vote` calls `election` with a recursive instance of `vote`, `election` must use V only positively, and this definition does not (because cardinality does not use its first argument only positively). Thus `vote` would be ill-defined.

Instead, we code around the problem:

[8] This is a minor departure from the definition in Section 2.3. It is convenient to allow witnesses to vote before the required d rounds have elapsed, but such early votes are ignored.

```
Definition election
  (V : event -> bool -> Prop)
  (n : nat) (y : event) (t f : nat)
  :=
  exists T F,
    cardinality
      (fun w => elector n w y) (t + f)
    /\ cardinality T t
    /\ cardinality F f
    /\ (forall w, T w
          -> elector n w y /\ V w true)
    /\ (forall w, F w
          -> elector n w y /\ V w false)
```

We existentially quantify over two sets T and F, where the intention is that T is the yeas and F the nays. The final two conjuncts check that every element of T (F) is indeed an elector and votes yes (no). The previous two conjuncts check that the cardinalities of T and F are t and f.

Finally, the first conjunct ensures that we are not missing any votes: If we assume that V does not allow a single witness to vote both ways (which `vote s x` will not), it follows that T and F are disjoint. Thus, in order to check that every vote is accounted for, we need only check that there are $t + f$ electors total.

The remaining two cases of `vote` implement coin rounds. They call `election` in the same manner as above, in order to find out whether a supermajority already exists.

```
(* coin round but supermajority exists *)
| vote_coin_super {s x y m n t f} {v : bool} :
    rwitness m x
    -> rwitness n y
    -> m < n
    -> (n - m) mod coin_freq = 0
    -> election (vote s x) (pred n) y t f
    -> (if v then t else f)
         > two_thirds number_peers
    -> vote s x y v

(* coin round and no supermajority exists *)
| vote_coin {s x y m n t f} :
    rwitness m x
    -> rwitness n y
    -> m < n
    -> (n - m) mod coin_freq = 0
    -> election (vote s x) (pred n) y t f
    -> t <= two_thirds number_peers
    -> f <= two_thirds number_peers
    -> vote s x y (coin y s)
```

In the final case, `coin y s` gives y's pseudo-random coin flip.

Induction Although Coq accepts the definition of `vote`, since all its recursive occurrences are positive, the definition is still too complicated for Coq to give it a useful induction principle. It does, however, provide a useful case-analysis principle. Thus, Coq essentially promises that `vote` is a fixed point, but not a *least* fixed point.

This is inconvenient, because there are many times one would like to do induction over votes. Fortunately, we can work around the problem. The recursive instances of `vote` always deal with strict ancestors of the vote in question, so one can employ well-founded induction using the strict-ancestor relation. Within that induction, one can then do a case analysis over the vote in question. This provides the power of the induction principle that one would have liked Coq to provide automatically.

3.5 Worlds

We use *worlds* to talk about potentially incompatible evolutions of the hashgraph. (The term is motivated by the use of the term in Kripke models.) In a modest loosening of the rules from our informal presentation, we allow that an event x might have two distinct self-children y and z, even if x's creator is honest, provided that y and z exist only in distinct futures. However, y and z can never coexist in the same future (again, if the creator is honest). That is, two events *in the same world* cannot form a fork on an honest peer.

Thus, a world is a set of events that is closed under ancestor, and contains no forks:[9]

```
Definition fork (e f : event) : Prop :=
  creator e = creator f
  /\ ~ e $= f
  /\ ~ f $= e.

Record world : Type :=
mk_world
  { member : event -> Prop;

    world_closed :
      forall x y,
        x @= y -> member y -> member x;

    world_forks :
      forall a,
        honest a
        -> ~ exists e f,
               member e /\ member f /\
               fork e f /\ creator e = a }.
```

Here `@=` means ancestor ($\leq$), and `$=` means self-ancestor ($\sqsubseteq$).

Many facts rely only on the events themselves, not any worlds they inhabit, but some require a world assumption. For example, consider the strongly seeing lemma (Lemma 2.3):

```
Lemma strongly_seeing :
  forall W x y v w,
    member W v
    -> member W w
    -> fork x y
    -> stsees x v
    -> stsees y w
    -> False.
```

Recall that, in the proof, we obtained events z and z', sharing the same honest creator, where $x \trianglelefteq z \leq v$ and $y \trianglelefteq z' \leq w$. We concluded that either $z \sqsubseteq z'$ or $z' \sqsubseteq z$, since otherwise z and z' would constitute a fork on a honest peer. But in the formalization, forks are possible—even on an honest peer—if the sides of the fork belong to different futures. Requiring v and w (and therefore z and z') to belong to the same world rules that out.

[9]Note that nothing requires a world to contain all the events, even at a particular point in time. Thus, for example, we can also use worlds to express the knowledge that a particular peer has about the hashgraph.

3.6 Pseudo-probability

The first element in our treatment of pseudo-probability is a sample space. A point in the sample space determines the outcome of every random or nondeterministic action, as well as the behavior of the adversary. However, unlike a true treatment of probability, we do not establish a measure for the sample space and compute probabilities. Instead, we will simply exclude any point in the sample space in which something happens that (informally speaking) has discrete probability zero. Specifically, we exclude any point that violates the probability assumption that was given informally in Section 2.4.1 and formally by `eventual_agreement` below.

This is formalized as the type `sample`, which one can think of as the type of all points in the sample space that are not being excluded.

Since a point in the sample space determines everything that takes place, and specifically determines what events get created, a sample determines a world. We call this the *global* world determined by that sample.

```
global : sample -> world
```

For now we can think of `global` as primitive, but once we develop some more machinery, we can actually define it.

At this point we can state the termination theorem (Theorem 2.17):

```
Theorem fame_consensus :
  forall s x,
    member (global s) x
    -> witness x
    -> exists y v,
         member (global s) y
         /\ decision s x y v.
```

If s is a sample, and x is a witness that gets created in the timeline resulting from s, then there exists some y from the same timeline, such that y makes a decision on x.

Recall that termination relied on a good coins lemma (Lemma 2.16), which we can state thus:

```
Lemma good_coins :
  forall s i x,
    rwitness i x
    -> exists j y t f b,
         i < j
         /\ (j - i) mod coin_freq = 0
         /\ member (global s) y
         /\ rwitness j y
         /\ election (vote s x) (pred j) y t f
         /\ ((t >= f /\ b = true)
              \/ (f > t /\ b = false))
         /\ forall w,
              member (global s) w
              -> rwitness j w
              -> honest (creator w)
              -> coin w s = b.
```

3.6.1 Eventual agreement

Good coins depends in turn on the probability assumption. Before we can state it, we need a little more machinery. The probability assumption says that eventually all coins will agree with a target, provided the target is specified using only information available earlier. We get this notion of "earlier" from *spawn order*, defined by:

```
Axiom spawn_inj :
  forall s i j,
    spawn s i = spawn s j -> i = j.

Axiom spawn_parent :
  forall s x i,
    parent x (spawn s i)
    -> exists j, x = spawn s j /\ j < i.

Axiom spawn_forks :
  forall s i j,
    honest (creator (spawn s i))
    -> fork (spawn s i) (spawn s j)
    -> False.

Axiom honest_peers_sync :
  forall s i a b,
    honest a
    -> honest b
    -> a <> b
    -> exists j k,
         i <= j
         /\ j <= k
         /\ creator (spawn s j) = a
         /\ creator (spawn s k) = b
         /\ spawn s j @ spawn s k.

Axiom no_orphans :
  forall x, exists s i, x = spawn s i.
```

Figure 2: Spawn order axioms

```
spawn : sample -> nat -> event
```

Suppose s is a point in the sample space. Then there exists a set S of all the events that will be created in the world s determines. (In order words, S contains all the elements of `global s`.) Then sort S according to the order the events are created in real time to obtain $x_0, x_1, x_2, \ldots$. Then we define `spawn s i` $= x_i$. Obviously, the "real time" ordering cannot be used by any participant, it is used only in reasoning about the algorithm.

Several axioms (appearing in Figure 2) govern spawn order: (1) No event spawns more than once. (2) An event's parents spawn before it does. The next two axioms are general rules of hashgraphs that are convenient to formalize in terms of spawn order: (3) Honest peers do not create forks. (4) Every honest peer eventually communicates with every other honest peer. (5) Finally, it is convenient to exclude any event that is part of no future.

We can define the global world `global s` using spawn order as the set of all events x such that `exists i, spawn s i = x`. The `spawn_parent` axiom provides the `world_closed` specification, and `spawn_forks` provides `world_forks`.

To define the eventual agreement axiom, we need to formalize the notion of a target depending only on earlier information. We will say that two samples are *similar* up to the ith spawn if their first i spawned events are the same, and they give the same coin flip for all of them except possibly the last:

```
Definition similar (s s' : sample) i :=
  (forall j,
     j <= i -> spawn s j = spawn s' j) /\
  (forall j,
     j < i -> coin (spawn s j) s
              = coin (spawn s j) s').
```

This is the information the ith event's coin's target can depend on, which is why it includes the ith event, but not its coin. Then the axiom says:

```
Axiom eventual_agreement :
  forall
  (n : nat)
  (s : sample)
  (Q : nat -> event -> Prop)
  (P : nat -> sample -> Prop),
    (forall i j x, Q i x -> Q j x -> i = j)
    -> (forall i x,
          Q i x -> honest (creator x))
    -> (forall i, cardinality_lt (Q i) n)
    -> (forall i k s',
          Q i (spawn s k)
          -> similar s s' k
          -> P i s <-> P i s')
    -> exists i v,
         (forall x, Q i x -> coin x s = v)
         /\ (Is_true v <-> P i s).
```

This says that, for any n and any sample s, if:

1. Q is a sequence of disjoint sets of honest events, each of size less than n (*e.g.,* the honest witnesses of each round), and
2. P is a sequence of predicates (the targets), such that P_i depends only on events earlier than every event in Q_i,

then there exists some i such that all Q_i's events' coin flips agree with P_i.

The axiom's final antecedent expresses the second condition above. It says if i is a position in the sequence, and k is the spawn order of some element of Q_i, and s' is another sample that is similar to s up to the kth spawn, then P_i is insensitive to whether it sees s or s'.

The alert reader might wonder why `P` returns a proposition and not a boolean, when that proposition must then be converted into a boolean (`v`). This gives us the flexibility to resolve a technical issue related to Coq's type system and constructivity. In the proof of `good_coins`, one needs the target to agree with the supermajority, if a supermajority exists. To ensure this, we make the round i target be whatever majority is observed by some arbitrary event in Q_i (specifically, we use the first element of Q_i to spawn). In Coq we cannot pluck an element from a set without some form of the axiom of choice, which we prefer to avoid. However, we can *describe* an element, and then state the target based on that, but we can do that only if we are giving the target as a proposition rather than a boolean, so we have access to the needed logical machinery.

4 Fairness

We also formalized an unpublished fairness theorem by Baird [1].

Lemma 4.1 *Suppose $d \geq 2$. Suppose also that x is a round i witness. If there exists y such that $x \ll y$ and both parents of y belong to a round no later than i, then x will be famous.*

Proof

Suppose w is an arbitrary round $i+d$ witness. We claim that $x \leq w$. It follows that w votes yes. Since w is arbitrary, every round $i+d$ witness votes yes, so x will be decided to be famous in round $i+d+1$.

Since $x \ll y$, the set $U = \{u \mid x \trianglelefteq u \leq y\}$ is supermajor. By Lemma 2.4, there exists a supermajor set V of round $i+d-1$ witnesses such that forall $v \in V$, $v \ll w$. The intersection of three supermajorities is nonempty, so there exists an honest peer a and events u and v created by a such that $x \trianglelefteq u \leq y$ and $v \ll w$. Since a is honest, either $u \sqsubseteq v$ or $v \sqsubset u$.

If $u \sqsubseteq v$ then we are done, since $x \leq u \leq v \leq w$, so let us assume $v \sqsubset u$. Then $v < y$. Thus v is an ancestor of one of y's parents, both of whom belong to a round no later than i. But v belongs to round $i+d-1$, which is at least $i+1$ since $d \geq 2$. This is a contradiction, since v cannot belong to a later round than any of its descendants.

Theorem 4.2 (Fairness) *If $d \geq 2$ then every round's set of famous witnesses is supermajor.*

Proof

Let i be a round number. By Lemma 2.7, there exists a round $i+1$ witness x. By following back x's ancestry, we can obtain a round $i+1$ witness y (possibly x itself), both of whose parents belong to round at most i. Since neither of y's parents belong to round $i+1$, y must strongly see a supermajor set of round i witnesses. By Lemma 4.1, all those witnesses will be famous.

The significance of the fairness theorem is that it means a majority of the famous witnesses in any round must be honest. Consequently, every event's consensus timestamp is governed by honest peers. The timestamp might come from a dishonest peer, but even if so, it will be bracketed on both sides by timestamps from honest peers.

The formalization of the fairness theorem and its consequences regarding timestamps is just under 700 lines of Coq.

5 Further Developments and Future Work

The ultimate goal of this work is a fully verified implementation of the Hashgraph consensus algorithm. There are a number of differences between the algorithm given here and the one that is implemented. We have incorporated many of them into the formalization already, but some others are future work.

- We have completed a version of the algorithm that supports weighted peers (a.k.a., proof-of-stake) instead of giving an equal weight to every peer. In that version, a supermajority of the weight must belong to honest peers. This change is largely straightforward, but there were some complications in the re-proof of Theorem 2.13 stemming mainly from the need for a weighted version of the pigeonhole principle.
- The algorithm here is a batch algorithm, while the implemented system is online. Moving to an online version requires two main changes:

  - We cannot permanently blacklist peers who create a fork, since this would require retaining information about them indefinitely. Instead we introduce a consistent way of establishing priority between the forked events, and one can only "see" the higher priority event. This is sufficient to reestablish Lemma 2.3, since the heart of the proof was the construction of an impossible event that sees both sides of a fork, and that remains impossible. This development is complete.
  - Since the participants of the network change over time, we need a way to deal with changing weights. (When a participant leaves the network, we can view that as its weight going to zero.) This is future work, although a prerequisite (having weights at all) is already done. The main complication is ensuring that all peers always agree on all the weights, when the weights are determined by previous transactions.

- The algorithm includes some operations that are expensive to perform and that we want to avoid as much as possible. A good example is strongly seeing, which requires one to count all the different peers one can pass through between one event and another. That involves exploring many different paths. However, one can show that one can limit oneself to exploring certain canonical paths without sacrificing any key properties. A version incorporating this and other optimizations is complete.

In addition, one would like to establish the fairness theorem for $d = 1$. This is an important area for research, since $d = 1$ is preferable (faster consensus) but provably honest timestamps are also desirable.

References

[1] Leemon Baird. Personal communication.

[2] Leemon Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. Technical Report SWIRLDS-TR-2016-01, Swirlds Inc., May 2016.

[3] Leemon Baird, Mance Harmon, and Paul Madsen. Hedera: A public Hashgraph network & governing council. White paper at `https://www.hedera.com/hh-whitepaper-v2.0-17Sep19.pdf`, August 2019.

[4] Leemon Baird and Atul Luykx. The Hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In *IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 2020. To appear.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[7] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT made practical. In *ACM SIGSAC Conference on Computer and Communications Security*, January 2018.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[9] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.

[10] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, October 2016.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Disseminated to The Cryptography Mailing List, November 2008.

[12] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.

[13] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[14] Visa Corporation. Visa Inc. at a glance. Marketing document at `https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf`, 2015. Retrieved July 2020.

[15] Visa Corporation. Visa fact sheet. Marketing document at `https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf`, 2020. Retrieved July 2020.