THE SWIRLDS HASHGRAPH CONSENSUS ALGORITHM: FAIR, FAST, BYZANTINE FAULT TOLERANCE

LEEMON BAIRD
BAIRD@SWIRLDS.COM
MAY 31, 2016
SWIRLDS TECH REPORT SWIRLDS-TR-2016-01

ABSTRACT. A new system, the *Swirlds hashgraph consensus algorithm*, is proposed for replicated state machines with guaranteed Byzantine fault tolerance. It achieves *fairness*, in the sense that it is difficult for an attacker to manipulate which of two transactions will be chosen to be first in the consensus order. It has complete asynchrony, no leaders, no round robin, no proof-of-work, eventual consensus with probability one, and high speed in the absence of faults. It is based on a gossip protocol, in which the participants don't just gossip about transactions. They *gossip about gossip*. They jointly build a *hashgraph* reflecting all of the gossip events. This allows Byzantine agreement to be achieved through *virtual voting*. Alice does not send Bob a vote over the Internet. Instead, Bob calculates what vote Alice would have sent, based on his knowledge of what Alice knows. This yields fair Byzantine agreement on a total order for all transactions, with very little communication overhead beyond the transactions themselves.

Keywords: Byzantine, Byzantine agreement, Byzantine fault tolerance, replicated state machine, fair, fairness, hashgraph, gossip about gossip, virtual voting, Swirlds

CONTENTS

[1]Revision date: April 26, 2017

List of Figures

1. Introduction

Distributed databases are often required to be replicated state machines with Byzantine fault tolerance. Some authors have used the term "Byzantine" in a weak sense, such as assuming that attackers will not collude, or that communication is weakly asynchronous [1]. In this paper, "Byzantine" will be used in the strong sense of its original definition [2]: up to just under 1/3 of the members can be attackers, they can collude, and they can delete or delay messages between honest members with no bounds on the message delays. The attackers can control the network to delay and delete any messages, though at any time, if an honest member repeatedly sends messages to another member, the attackers must eventually allow one through. It is assumed that secure digital signatures exist, so attackers cannot undetectably modify messages. It is assumed that secure hash functions exist, for which collisions will never be found. This paper proposes and describes the Swirlds hashgraph consensus algorithm, and proves Byzantine fault tolerance, under the strong definition.

No deterministic Byzantine system can be completely asynchronous, with unbounded message delays, and still guarantee consensus, by the FLP theorem [3]. But it is possible for a nondeterministic system to achieve consensus with probability one. The hashgraph consensus algorithm is completely asynchronous, is nondeterministic, and achieves Byzantine agreement with probability one.

Some systems, such as Paxos [4] or Raft [5] use a leader, which can make them vulnerable to large delays if an attacker launches a denial of service attack on the current leader [6]. Many systems can even be delayed by just a single bad client [7]. In fact, the latter paper suggests that systems with such vulnerabilities might better be described as "Byzantine fault survivable" rather than "Byzantine fault tolerant". Hashgraph consensus does not use a leader, and is resilient to denial of service attacks on small subsets of the members.

Other systems, such as Bitcoin, are based on proof-of-work blockchains [8]. This avoids all the above problems. However, such systems cannot be Byzantine, because a member never knows for sure when consensus has been achieved; they only have a probability of confidence that continues to rise over time. If two blocks are mined simultaneously, then the chain will fork until the community can agree on which branch to extend. If the blocks are added slowly, then the community can always add to the longer branch, and eventually the other branch will stop growing, and can be pruned and discarded because it is "stale". This leads to inefficiency, in the sense

that some blocks are mined properly, but discarded anyway. It also means that it is necessary to slow down how fast blocks are mined, so that the community can jointly prune branches faster than new branches sprout. That is the purpose of the proof-of-work. By requiring that the miners solve difficult computation problems to mine a block, it can ensure that the entire network will have sufficiently long delays between mining events, on average. The hashgraph consensus algorithm is equivalent to a block chain in which the "chain" is constantly branching, without any pruning, where no blocks are ever stale, and where each miner is allowed to mine many new blocks per second, without proof-of-work, and with 100% efficiency.

Proof-of-work blockchains also require that electricity be wasted on extra computations, and perhaps that expensive mining rigs be bought. A proof-of-expired-time system [9] can avoid the wasted electricity (though perhaps not the cost of mining rigs) by using trusted hardware chips that delay for long periods, as if they were doing proof-of-work computations. However, that requires that all participants trust the company that created the chip. Such trust in chip venders exists in some situations, but not in others, such as when FreeBSD was changed to not rely solely on the hardware RDRAND instruction for secure random numbers, because "we cannot trust them any more" [10].

Byzantine agreement systems have been developed for Byzantine agreement that avoid the above problems. These systems typically exchange many messages for the members to vote. For n members to decide a single YES/NO question, some systems can require $O(n)$ messages to be sent across the network. Other systems can require $O(n^2)$, or even $O(n^3)$ messages crossing the network per binary decision [11]. An algorithm for a single YES/NO decision can then be extended to deciding a total order on a set of transactions, which may further increase the vote traffic. Hashgraph sends no votes at all over the network, because all voting is virtual.

2. Core concepts

The hashgraph consensus algorithm is based on the following core concepts.

- *Transactions* - any member can create a signed transaction at any time. All members get a copy of it, and the community reaches Byzantine agreement on the order of those transactions.
- *Fairness* - it should be difficult for a small group of attackers to unfairly influence the order of transactions that is chosen as the consensus.
- *Gossip* - information spreads by each member repeatedly choosing another member at random, and telling them all they know
- *Hashgraph* - a data structure that records who gossiped to whom, and in what order.
- *Gossip about gossip* - the hashgraph is spread through the gossip protocol. The information being gossiped is the history of the gossip itself, so it is "gossip about gossip". This uses very little bandwidth overhead beyond simply gossiping the transactions alone.
- *Virtual voting* - every member has a copy of the hashgraph, so Alice can calculate what vote Bob *would have* sent her, if they had been running a traditional Byzantine agreement protocol that involved sending votes. So Bob doesn't need to actually her the vote. Every member can reach Byzantine agreement on any number of decisions, without a single vote ever being sent. The hashgraph alone is sufficient. So zero bandwidth is used, beyond simply gossiping the hashgraph.
- *Famous witnesses* - The community could put a list of n transactions into order by running separate Byzantine agreement protocols on $O(n \log n)$ different yes/no questions of the form "did event x come before event y?" A much faster approach is to pick just a few events (vertices in the hashgraph), to be called *witnesses*, and define a witness to be *famous* if the hashgraph shows that most members received it fairly soon after it was created. Then it's sufficient to run the Byzantine agreement protocol only for witnesses, deciding for each witness the single question "is this witness famous?" Once Byzantine agreement is reached on the exact set of famous witnesses, it is easy to derive from the hashgraph a fair total order for all events.
- *Strongly seeing* - given any two vertices x and y in the hashgraph, it can be immediately calculated whether x can strongly see y, which is defined to be true if they are connected by multiple directed paths passing through enough members. This concept allows the key lemma to be proved: that if Alice and Bob are both able to calculate Carol's virtual vote on a given question, then Alice and Bob get the same answer. That lemma forms the foundation for the rest of the mathematical proof of Byzantine agreement with probability one.

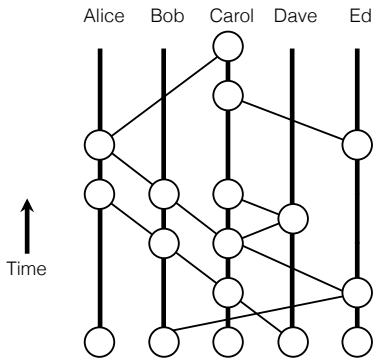

FIGURE 1. Gossip history as a directed graph. The history of any gossip protocol can be represented by a graph where each member is one column of vertices. When Alice receives gossip from Bob, telling her everything he knows, that gossip event is represented by a vertex in the Alice column, with two edges going downward to the immediately-preceding gossip events by Alice and Bob.

3. Gossip about gossip: the hashgraph

Hashgraph consensus uses a gossip protocol. This means that a member such as Alice will choose another member at random, such as Bob, and then Alice will tell Bob all of the information she knows so far. Alice then repeats with a different random member. Bob repeatedly does the same, and all other members do the same. In this way, if a single member becomes aware of new information, it will spread exponentially fast through the community until every member is aware of it.

The history of any gossip protocol can be illustrated by a directed graph like Figure 1. Each vertex in the Alice column represents a gossip event. For example, the top event in the Alice column represents Bob performing a gossip sync to Alice in which Bob sent her all of the information that he knew. That vertex has two downward edges, connecting to the immediately-preceding gossips for Alice and Bob. Time flows up the graph, so lower vertices represent earlier events in history. In a typical gossip protocol, a diagram such as this is merely used to discuss the protocol; there is no actual graph like that stored in memory anywhere.

In hashgraph consensus, that graph is an actual data structure. Figure 2 illustrates this data structure. Each *event* (vertex) is stored in memory as a sequence of bytes, signed by its creator. For example, one event by Alice (red) records the fact that Bob performed a gossip sync in which he sent her everything he knew. This event is created by Alice and signed by her, and contains the hashes of two other events: her last event and Bob's last event prior to that gossip sync. The red event can also contain a *payload* of any transactions that Alice chooses to create at that moment, and perhaps a timestamp which is the time and date that Alice claims to have created it. The other ancestors of that event (gray) are not contained within it, but are determined by the set of cryptographic hashes. Data structures with graphs of hashes have been used for other purposes, such as in Git where the vertices are versions of a file tree, and the edges represent changes. But Git stores no

record of how members communicated. The hashgraph is for a different purpose. It records the history of how the members communicated.

Gossip protocols are widely used to transfer a variety of types of information. They can involve gossiping about user identities, or gossiping about transactions, or gossiping about blockchain blocks, or gossiping about any other information that needs to be distributed. But what if the protocol were to gossip about gossip? What if the members were gossiping to transfer the hashgraph itself? When Bob gossiped to Alice, he would give her all of the events which he knew and she did not.

Gossiping a hashgraph gives the participants a great deal of information. If a new transaction is placed in the payload of an event, it will quickly spread to all members, until every member knows it. Alice will learn of the transaction. And she will know exactly when Bob learned of the transaction. And she will know exactly when Carol learned of the fact that Bob had learned of that transaction. Deep chains of such reasoning become possible when all members have a copy of the hashgraph. As the hashgraph grows upward, the different members may have slightly different subsets of the new events near the top, but they will quickly converge to having exactly the same events lower down in the hashgraph. Furthermore, if Alice and Bob happen to both have a given event, then they are guaranteed to also both have all its ancestors. And they will agree on all the edges in the subgraph of those ancestors. All of this allows powerful algorithms to run locally, including for Byzantine fault tolerance.

This power comes with very little communication overhead. If a community is simply gossiping signed transactions that they create, there is a certain amount of bandwidth required. If they instead gossip a hashgraph, and if there are enough transactions that a typical event contains at least one transaction, then the overhead is minimal. Instead of Alice signing a transaction she creates, she will sign the event she creates to contain that transaction. Either way, she is only sending one signature. And either way, she must send the transaction itself. The only extra overhead is that she must send the two hashes. But even that can be greatly compressed. In figure 2, Alice will not send Carol the red event until Carol already has all its earlier ancestors (either from Alice, or from an earlier sync with someone else). So Alice does not need to send the two hashes of the two blue parent events. It is sufficient to tell Carol that this event is the next one by Alice, and that its other-parent is the third one by Bob. With appropriate compression, this can be sent in very few bytes, adding only a few percent to the size of the message being sent.

4. Consensus algorithm

It is not enough to ensure that every member knows every event. It is also necessary to agree on a linear ordering of the events, and thus of the transactions recorded inside the events. Most Byzantine fault tolerance protocols without a leader depend on members sending each other votes. So for n members to agree on a single YES/NO question might require $O(n^2)$ voting messages to be sent over the network, as every member tells every other member their vote. Some of these protocols require receipts on votes sent to everyone, making them $O(n^3)$. And they may require multiple rounds of voting, which further increases the number of voting messages sent.

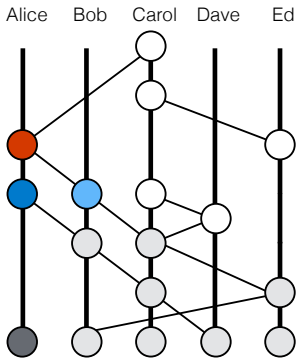

FIGURE 2. The hashgraph data structure. Alice creates an *event* (red) recording the occurrence of Bob doing a gossip sync to her and telling her everything he knows. The event contains a hash of two parent events (blue): the self-parent (dark blue) by the same creator Alice, and the other-parent (light blue) by Bob. It also contains a payload of any new transactions that Alice chooses to create at that moment, and a digital signature by Alice. The other ancestor events (gray) are not stored in the red event, but they are determined by all the hashes. The other self-ancestors (dark gray) are those reachable by sequences of self-parent links, and the others (light gray) are not.

Hashgraph consensus does not require any votes to be sent. Every member has a copy of the hashgraph. If Alice and Bob both have the same hashgraph, then they can calculate a total order on the events according to any deterministic function of that hashgraph, and they will both get the same answer. Therefore, consensus is achieved, even without sending vote messages.

Of course, Alice and Bob may not have exactly the same hashgraph at any given moment. They will typically match in the older events. But for the very recent events, each may have events that the other has not yet seen. Furthermore, there may occasionally be a new event released to the community that should be placed in a lower (earlier) location in the hashgraph. The hashgraph consensus algorithm deals with these issue using a system that is best thought of as *virtual voting*.

Suppose Alice has hashgraph A and Bob hash hashgraph B. These hashgraphs may be slightly different at any given moment, but they will always be *consistent*. Consistent means that if A and B both contain event x, then they will both contain exactly the same set of ancestors for x, and will both contain exactly the same set of edges between those ancestors. If Alice knows of x and Bob does not, and both of them are honest and actively participating, then we would expect Bob to learn of x fairly quickly, through the gossip protocol. The consensus algorithm assumes that will happen eventually, but does not make any assumptions about how fast it will happen. The protocol is completely asynchronous, and does not make assumptions about timeout periods, or the speed of gossip, or the rate at which progress is made.

Alice will calculate a total order on the events in A by calculating a series of *elections*. In each election, some of the events in A will be considered to cast a *vote*, and some of the events in A will be considered to receive that vote. Alice will

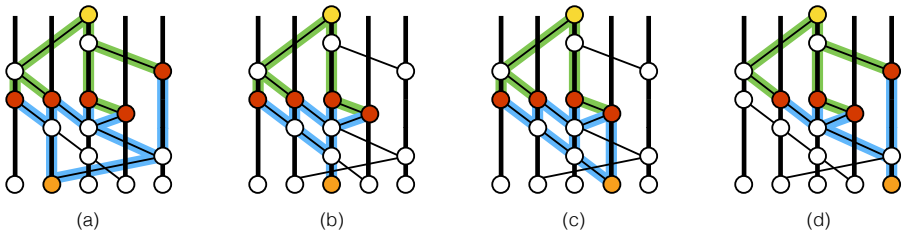

FIGURE 3. Illustration of strongly seeing. In each hashgraph, the yellow event at the top can *strongly see* one of the orange events on the bottom row. There are $n = 5$ members, so the least integer greater than $2n/3$ is 4. In (d), one event (orange) is an ancestor of each of 4 intermediate events by different creators (red), each of which is an ancestor of the yellow event. Therefore, the yellow event can strongly see the orange event. Each of the other hashgraphs is colored to show the same for a different orange event on the bottom row, which the yellow event see through at least 4 red events. If all 4 orange events and both parents of the yellow event have a created round of r, then yellow is created in round $r + 1$, because it can strongly see more than $2n/3$ witnesses created by different members in round r. Note that every event is defined to be both an *ancestor* and a *self-ancestor* of itself.

calculate multiple elections, and a given event might participate in some elections but not others, and might cast different votes in different elections. If the event was created by Bob, we will talk of Bob voting a certain way in a given election. But the actual member Bob is not involved. This is purely a calculation that Alice is performing locally, where she is calculating what vote Bob *would have* sent her, if the real Bob were actually sending votes over the internet to her.

This virtual voting has several benefits. In addition to saving bandwidth, it ensures that members always calculate their votes according to the rules. If Alice is honest, she will calculate virtual votes for the virtual Bob that are honest. Even if the real Bob is a cheater, he cannot attack Alice by making the virtual Bob vote incorrectly.

Bob can try to cheat in a different way. Suppose Bob creates an event x with a certain self-parent hash pointing to his previous event z. Then Bob creates a new event y, but gives it a self-parent hash of z, instead of giving it a self-parent hash of x as he should. This means that the events by Bob in the hashgraph will no longer be a chain, as they should be. They will now be a tree, because he has created a *fork*. If Bob gossips x to Alice and y to Carol, then for a while, Alice and Carol may not be aware of the fork. And Alice may calculate a virtual vote for x that is different from Carol's virtual vote for y.

The hashgraph consensus algorithm prevents this attack by using the concept of one state *seeing* another, and the concept of one state *strongly seeing* another. These are based on definitions of *ancestor* and *self-ancestor* such that every event is considered to be both an ancestor and self-ancestor of itself.

If Bob creates two events x and y, neither of which is a self-ancestor of the other, then Bob has cheated by forking. If some event w has x as an ancestor but doesn't

have y as an ancestor, then the event w can *see* event x. However, if both x and y are ancestors of w, then w is defined to not see either of them, nor any other event by the same creator. In other words, w can see x if x is known to it, and no forks by that creator are known to it.

If there are n members, then an event w can *strongly see* an event x, if w can see more than $2n/3$ events by different members, each of which can see x. This concept is illustrated in Figure 3. Four copies of the same hashgraph are shown, each with a different event on the bottom row colored orange. In (d), the yellow event at the top can see 4 red events by different members, each of which can see the orange event at the bottom. This is also true in (a), (b), and (c), with (a) actually having 5 red events. But only 4 are needed for strongly seeing, because this example has $n = 5$ members, and the least integer greater than $2n/3$ is 4.

This concept allows an agreement protocol to achieve Byzantine fault tolerance without any actual voting, just through local virtual voting.

In virtual voting, when event x votes on some YES/NO question (e.g., whether some other event is famous), the vote is calculated purely as a function of the ancestors of x. That vote is only considered to be sent from x to its descendant event w if w can strongly see x. It is proved in section 5 that if x and y are on different branches of an illegal fork, then w can strongly see at most one of x and y, but not both. Furthermore, if hashgraphs A and B are consistent, then it is not possible for one event to strongly see x in A and another event strongly see y in B. That lemma is the cornerstone of the Byzantine proof. It ensures that even if an attacker tries to cheat by forking, they will still be unable to cause different members to decide on different orders. Historically, some Byzantine agreement algorithms have required members to send out "receipts" to everyone for each vote they receive, to defend against Alice sending inconsistent votes to Bob and Carol. There are some similarities between that attack and a hashgraph forking attack, and between the use of receipts and the use of strongly seeing.

Given those definitions, the complete hashgraph consensus protocol can be given by the algorithms in Figures 4, 5, 6, and 7.

The main algorithm in Figure 4 shows that the communication is very simple: Alice randomly picks another member Bob, and gossips to him all the events that she knows. Bob then creates a new event to record the fact of that gossip.

That simple gossip protocol is sufficient for Byzantine Fault Tolerance and correctness. But it can be extended in various ways to improve efficiency.For example, Alice and Bob might tell each other which events they already know, then Alice sends Bob all the events that she knows that he doesn't. The protocol might require that Alice send those events in topological order, so Bob will always have an event's parents before receiving the event. The protocol might even say that after Alice syncs to Bob, then Bob will immediately sync back to Alice. Multiple syncs can happen at once, so Alice might be syncing to several members at the same time several members are syncing to her. These and other optimizations can all be used, but this simple one is sufficient.

After each sync, the member calls the three procedures to determine the consensus order for as many events as possible. These involve no communication; purely local computations are sufficient. In these procedures, each for loop visits events in *topological order*, where an event is always visited after its parents. In the first for loop of the algorithm, if x is the first event in all of history, then it won't have

parents or previous rounds, so it should be set to x.round=1 and x.witness=TRUE. The algorithm also uses a constant n, which is the number of members in the entire population, and c which is a small integer constant greater than 2, such as $c = 10$. In the following algorithm, Byzantine agreement is guaranteed with probability one.

It is useful to define a round number for each event as a function of its ancestors. In divideRounds (Figure 5), every known event is assigned an integer *round number* (definition 5.2) as a function of the round numbers of its ancestors. The hashgraphs in Figure 3 show how this is done. If all the events on the bottom row were round r, then all the rest of the events in those figures would also be round r, except for the yellow event, which would be round $r + 1$. The yellow event is advanced to the next round, $r + 1$, because it is able to strongly see more than $2n/3$ events from round r. The first event in history is defined to be round 1, so all future rounds are determined by this. Every event will eventually have both a *round created* and a *round received* number. The round created is also called the *round* or *round number*.

For any given member, the first event they create in each round is called a *witness*. It is only the witness events that send and receive the virtual votes. This occurs in the decideFame procedure shown in Figure 6. This procedure is where the Byzantine agreement occurs. For each witness, it decides whether it is *famous*. A witness is famous if many of the witnesses in the next round can see it, and it is not famous if many can't. The Byzantine agreement protocol runs an election for each witness, to determine if it is famous. For a witness x in round r, each witness in round $r+1$ will vote that x is famous if it can see it. If more than $2n/3$ agree on whether it is famous, then the community has decided, and the election is over. If the vote is more balanced, then it continues for as many rounds as necessary, with each witness in a normal round voting according to the majority of the witnesses that it can strongly see in the previous round. To defend against attackers who can control the internet, there are periodic *coin rounds* where witnesses can vote pseudorandomly. This means that even if an attacker can control all the messages going over the internet to keep the votes carefully split, there is still a chance that the community will randomly cross the $2n/3$ threshold. And so agreement is eventually reached, with probability one.

In Figure 6, the algorithm would continue to work if the line *"if d=1"* were changed to *"if d=2"*. In that revised algorithm, each election would start one round later. It would even continue to work if the two were combined in the following hybrid algorithm. In each round, first run all its elections with the *"d=1"* check. If the fame of every witness in that round is decided, and $2n/3$ or fewer members created famous witnesses in that round, then the elections for just that round are all re-run, using a $d = 2$ check. For this hybrid algorithm, all of the theorems in this paper would continue to be true, including the proof of Byzantine Fault Tolerance. For rounds that trigger the new elections, the time to consensus would increase slightly (by perhaps 20%). But that would happen very rarely in practice, and when it did, it might increase the number of famous witnesses, to ensure fairness.

Once consensus has been reached on whether each witness in a given round is famous, it is then easy to use that to determine a consensus timestamp and a consensus total order on older events. This is done by procedure findOrder, found in Figure 7.

```
run two loops in parallel:
    loop
        sync all known events to a random member
    end loop
    loop
        receive a sync
        create a new event
        call divideRounds
        call decideFame
        call findOrder
    end loop
```

FIGURE 4. The Swirlds hashgraph consensus algorithm. Each member repeatedly calls other members chosen at random, and syncs to them. In parallel with the outgoing syncs, each member receives incoming syncs. When Alice syncs to Bob, she sends all events that she knows that Bob doesn't. Bob adds these events to the hashgraph, accepting only events with valid signatures containing valid hashes of parent events he has. All known events are then divided into rounds. Then the first events by each member in each round (the "witnesses") are decided as being famous or not, through purely local Byzantine agreement with virtual voting. Then the total order is found on those events for which enough information is available. If two members independently assign a position in history to an event, they are guaranteed to assign the same position, and guaranteed to never change it, even as more information comes in. Furthermore, each event is eventually assigned such a position, with probability one.

First, the *received round* is calculated. Event x has a received round of r if that is the first round in which all the unique famous witnesses were descendants of it, and the fame of every witness is decided for rounds less than or equal to r.

Then, the *received time* is calculated. Suppose event x has a received round of r, and Alice created a unique famous witness y in round r. The algorithm finds z, the earliest self-ancestors of y that had learned of x. Let t be the timestamp that Alice put inside z when she created z. Then t can be considered the time at which Alice claims to have first learned of x. The received time for x is the median of all such timestamps, for all the creators of the unique famous witnesses in round r.

Then the consensus order is calculated. All events are sorted by their received round. If two events have the same received round, then they are sorted by their received time. If there are still ties, they are broken by simply sorting by signature, after the signature is whitened by XORing with the signatures of all the unique famous witnesses in the received round.

```
procedure divideRounds

for each event x
    r ← max round of parents of x (or 1 if none exist)
    if x can strongly see more than 2n/3 round r witnesses
        x.round ← r+1
    else
        x.round ← r
    x.witness ← (x has no self parent)
                or (x.round > x.selfParent.round)
```

FIGURE 5. The divideRounds procedure. As soon as an event x is known, it is assigned a round number x.round, and the boolean value x.witness is calculated, indicating whether it is a "witness", the first event that a member created in that round.

5. Proof of Byzantine fault tolerance

This section provides a number of useful definitions, followed by several proofs, building up from the Strongly Seeing Lemma (lemma 5.12) to the Byzantine Fault Tolerance Theorem (theorem 5.19). In the proofs it is assumed that there are n members ($n > 1$), more than $2n/3$ of which are honest, and less than $n/3$ of which are not honest. It is also assumed that the digital signatures and cryptographic hashes are secure, so signatures cannot be forged, signed messages cannot be changed without detection, and hash collisions can never be found. The syncing gossip protocol is assumed to ensure that when Alice sends Bob all the events she knows, Bob accepts only those that have a valid signature and contain valid hashes corresponding to events that he has. The system is totally asynchronous. It is assumed that for any honest members Alice and Bob, Alice will eventually try to sync with Bob, and if Alice repeatedly tries to send Bob a message, she will eventually succeed. No other assumptions are made about network reliability or network speed or timeout periods. Specifically, the attacker is allowed to completely control the network, deleting and delaying messages arbitrarily, subject to the constraint that a message between honest members that is sent repeatedly must eventually have a copy of it get through.

Definition 5.1. An event x is defined to be an *ancestor* of event y if x is y, or a parent of y, or a parent of a parent of y, and so on. It is also a *self-ancestor* of y if x is y, or a self-parent of y, or a self-parent of a self-parent of y and so on.

Definition 5.2. The *round created number* (or *round*) of an event x is defined to be $r + i$, where r is the maximum round number of the parents of x (or 1 if it has no parents), and i is defined to be 1 if x can strongly see more than $2n/3$ witnesses in round r (or 0 if it can't).

Definition 5.3. The *round received number* (or *round received*) of an event x is defined to be the first round where all unique famous witnesses are descendants of x.

```
procedure decideFame

for each event x in order from earlier rounds to later
  x.famous ← UNDECIDED
  for each event y in order from earlier rounds to later
    if x.witness and y.witness and y.round>x.round
      d ← y.round - x.round
      s ← the set of witness events in round
           y.round-1 that y can strongly see
      v ← majority vote in s (is TRUE for a tie)
      t ← number of events in s with a vote of v
      if d = 1                    // first round of the election
          y.vote ← can y see x?
      else
          if d mod c > 0          // this is a normal round
              if t > 2*n/3        // if supermajority, then decide
                  x.famous ← v
                  y.vote ← v
                 break out of the y loop
              else                // else, just vote
                  y.vote ← v
          else                    // this is a coin round
              if t > 2*n/3        // if supermajority, then vote
                  y.vote ← v
              else                // else flip a coin
                  y.vote ← middle bit of y.signature
```

FIGURE 6. The decideFame procedure. For each witness event (i.e., an event x where x.witness is true), decide whether it is famous (i.e., assign a boolean to x.famous). This decision is done by a Byzantine agreement protocol based on virtual voting. Each member runs it locally, on their own copy of the hashgraph, with no additional communication. It treats the events in the hashgraph as if they were sending votes to each other, though the calculation is purely local to a member's computer. The member assigns votes to the witnesses of each round, for several rounds, until more than 2/3 of the population agrees. To find the fame of x, re-run this repeatedly on the growing hashgraph until x.famous receives a value.

Definition 5.4. The pair of events (x, y) is a *fork* if x and y have the same creator, but neither is a self-ancestor of the other.

```
procedure findOrder

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

return all events that have roundReceived not UNDECIDED,
      sorted by roundReceived, then ties sorted by
      consensusTimestamp, then by whitened signature
```

FIGURE 7. The findOrder procedure. Once all the witnesses in round r have their fame decided, find the set of famous witnesses in that round, then remove from that set any famous witness that has the same creator as any other in that set. The remaining famous witnesses are the *unique famous witnesses*. They act as the judges to assign earlier events a round received and consensus timestamp. An event is said to be "received" in the first round where all the unique famous witnesses have received it, if all earlier rounds have the fame of all witnesses decided. Its timestamp is the median of the timestamps of those events where each of those members first received it. Once these have been calculated, the events are sorted by round received. Any ties are subsorted by consensus timestamp. Any remaining ties are subsorted by whitened signature. The whitened signature is the signature XORed with the signatures of all unique famous witnesses in the received round.

Definition 5.5. An *honest* member tries to sync infinitely often with every other member, creates a valid event after each sync (with hashes of the latest self-parent and other-parent), and never creates two events that are forks with each other.

Definition 5.6. An event x can *see* event y if y is an ancestor of x, and the ancestors of x do not include a fork by the creator of y.

Definition 5.7. An event x can *strongly see* event y if x can see y and there is a set S of events by more than $2/3$ of the members such that x can see every event in S, and every event in S can see y.

Definition 5.8. A *witness* is the first event created by a member in a round.

Definition 5.9. A *famous witness* is a witness that has been decided to be *famous* by the community, using the algorithms described here. Informally, the community tends to decide that a witness is famous if many members see it by the start of the next round. 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. In the absence of forking, each famous witness is also a unique famous witness.

Definition 5.10. Hashgraphs A and B are *consistent* iff for any event x contained in both hashgraphs, both contain the same set of ancestors for x, with the same parent and self-parent edges between those ancestors.

Lemma 5.11. *All members have consistent hashgraphs.*

Proof: If two members have hashgraphs containing event x, then they have the same two hashes contained within x. A member will not accept an event during a sync unless that member already has both parents for that event, so both hashgraphs must contain both parents for x. The cryptographic hashes are assumed to be secure, therefore the parents must be the same. By induction, all ancestors of x must be the same. Therefore the two hashgraphs are consistent. □

The purpose of the concept of *strongly seeing* is to make the following lemma true. This lemma is the foundation of the entire proof, because it allows for consistent voting, and for guarantees that different members will never calculate inconsistent results, even with purely virtual voting.

Lemma 5.12 (Strongly Seeing Lemma)**.** *If the pair of events (x, y) is a fork, and x is strongly seen by event z in hashgraph A, then y will not be strongly seen by any event in any hashgraph B that is consistent with A.*

Proof: The proof is by contradiction. Suppose event w in B can strongly see y. By the definition of strongly seeing, there must exist a set S_A of events in A that z can see, and that all can see x. There must be a set S_B of events in B that w can see, and which all see y. Then S_A must contain events created by more than $2n/3$ members, and so must S_B, therefore there must be an overlap of more than $n/3$ members who created events in both sets. It is assumed that less than $n/3$ members are not honest, so there must be at least one honest member who created events in both S_A and S_B. Let m be such a member, and their events $q_A \in S_A$ and $q_B \in S_B$. Because m is honest, q_A and q_B cannot be forks with each other, so one must be the self-ancestor of the other. Without loss of generality, let q_A be the self-ancestor of q_B. The hashgraphs A and B are consistent, and q_B is in B, so its ancestor q_A must also be in B. Then in B, x is an ancestor of q_A, which is an ancestor of q_B, so x is an ancestor of q_B. But y is also an ancestor of q_B. So both x and y are ancestors of q_B and are forks of each other, so q_B cannot see either of them. But that contradicts the assumption that q_B can see y in B. That is a contradiction, so the lemma is proved. □

At every moment, all members will have consistent hashgraphs. If two hashgraphs are consistent, and both contain an event x, then they will both contain the

same set of ancestors for x. This will cause them to agree on every property of x that is purely a function of its ancestors. That includes its round created, whether it is a witness, what events it can see, what events it can strongly see, and how it will vote in each election (if it's a witness). For most of these properties, this follows directly from the definition. The following lemma proves that it is also true for the round created.

Lemma 5.13. *If hashgraphs A and B are consistent and both contain event x, then both will assign the same round created number to x.*

Proof: If the consistent hashgraphs both contain x, then they both contain the same set of all its ancestors, including the first event in history. Then the proof is by induction: they agree on the round number of that first event, which is 1 by definition. And if they both contain an arbitrary state y, and agree on the round numbers of all its ancestors, then they will agree on the maximum round number r of the parents of y, and will agree on whether y can strongly see more than $2n/3$ witnesses created in round r by different members, and therefore will agree on the round number of y. Therefore they will agree on the round number of all events they share, including x. □

Different members may have slightly different hashgraphs, and so may have slightly different elections. However, all the votes will be consistent. If one hashgraph shows Alice sending Bob a given vote in a given round for a given election, then any consistent hashgraph must show either the same vote, or no vote at all from Alice to Bob in that round. It is impossible for two consistent hashgraphs to show two different votes for Alice in that round. This is shown in the following lemma.

Lemma 5.14. *If hashgraphs A and B are consistent, and the algorithm running on A shows that a round r event by member m_0 sends a vote v_A to member m_1 in round $r+1$, and the algorithm running on B shows that a round r event by member m_0 sends a vote v_B to an event by member m_1 in round $r+1$, then $v_A = v_B$.*

Proof: The algorithm only sends a vote from event x to event y if y can strongly see x. It is not possible for consistent hashgraphs to have two events that are forks of each other and that are both strongly seen, by the Strongly Seeing lemma (lemma 5.12). Therefore, the two votes must be coming from the same event x in both hashgraphs. An event's vote is calculated purely as a function of its ancestors, so the two hashgraphs must agree on the vote, and $v_A = v_B$. □

Byzantine agreement on a particular YES/NO question is achieved by multiple rounds of virtual voting. A given member will end their election calculations in round r if it is a normal round (not a coin round) and some round $r+1$ event strongly sees more than $2n/3$ of the members voting the same way in round r. If that happens, then every active member will end their election in round r or $r+1$ (or $r+2$ if $r+1$ is a coin round), and will decide the same way. In other words, the following lemma proves that if anyone decides on a YES/NO question, then everyone achieves Byzantine consensus almost immediately thereafter.

Lemma 5.15. *If hashgraphs A and B are consistent, and A decides a Byzantine agreement election with result v in round r and B has not decided prior to r, then B will decide v in round $r+2$ or before.*

Proof: Decisions can't happen in coin rounds, so r must be a regular round. If A decides a vote v, that means some witness in round r received votes of v from a set of members S that contains more than $2n/3$ members. Because voting is consistent (by the previous lemma), all other round r events in A and B will receive votes from more than $2n/3$ members, a majority of whom will also be in S, because two subsets of size greater than $2n/3$ drawn from a set of size n must each have a majority of their elements in common with the other. Therefore, every round r witness in both A and B will vote for v (and some may decide v). If round $r+1$ is a regular round, then every event in A and B in that round will receive unanimous votes of v and will decide v. If round $r+1$ is a coin round, then all will receive unanimous votes of v, so none will flip coins, and all will vote v, and then all will decide v in round $r+2$. □

The following theorem shows that Byzantine fault tolerance is achieved for any single YES/NO question.

Theorem 5.16. *For any single YES/NO question, consensus is achieved eventually with probability 1.*

Proof: If any member decides the question, then all members will decide the same way within 2 rounds, by the last lemma. So the only way consensus could fail is if no member ever decides, because no witness ever receives more than $2n/3$ matching votes. However, in a coin round, if such a supermajority has not yet been achieved, then all the honest members randomly choose their vote, and will have a nonzero probability of all choosing the same vote. Coin rounds occur periodically forever, so eventually the honest members will become unanimous, with probability one, and then consensus will be reached within 2 rounds. □

In the hashgraph consensus algorithm, Byzantine agreement is used to decide whether each witness in a given round is famous or not. Every round is guaranteed to have at least one witness that is famous, by the following lemma.

Lemma 5.17. *For any round number r, for any hashgraph that has at least one event in round $r+3$, there will be at least one witness in round r that will be decided to be famous by the consensus algorithm, and this decision will be made by every witness in round $r+3$, or earlier.*

Proof: Let S_{r+3} be a set containing a single witness in round $r+3$, in a hashgraph that has at least one such witness. For each $i < r+3$, let S_i be the set of all witnesses in round i that are each strongly seen by at least one witness in S_{i+1}. It must be the case that $2n/3 < |S_i| \leq n$ for all $i \leq r+2$, because the existence of an event in round $i+1$ guarantees more than $2n/3$ are strongly seen in round i, and none of the n members can create more than one witness in a given round that is strongly seen (by the Strongly Seeing lemma, lemma 5.12). Strongly seeing implies seeing, so each event in S_{r+1} sees more than two thirds of the events in S_r. Therefore, on average, each event in S_r is seen by more than two thirds of the events in S_{r+1}. They can't all be below average, so there must be at least one event in S_r (call it x) that is seen by more than two thirds of the events in S_{r+1}. So more than two thirds of S_{r+1} will vote YES in the election for x being famous. Therefore, every event in S_{r+2} will receive more YES votes than NO votes for the fame of x, and will therefore vote for x being famous (and may or may not decide that x is famous). Therefore, the event in S_{r+3} will receive unanimous votes for

x being famous, which will cause it to decide that x is famous. Therefore, every member with an event in round $r + 3$ will first decide that x is famous in either round $r + 2$ or $r + 3$. □

Lemma 5.18. *If hashgraph A does not contain event x, but does contain all the parents of x, and hashgraph B is the result of adding x to A, and x is a witness created in round r, and A has at least one witness in round r whose fame has been decided (as either famous or as not famous), then x will be decided as "not famous" in B.*

Proof: Let w be a witness in A that decided the fame for one of the witnesses in round r. None of the ancestors of w can see x, because there is no x in A. So they will also not see x in B, because they have the same ancestors in consistent hashgraphs. Therefore the ancestors of w that are witnesses in round $r + 1$ will all vote NO on the fame of x in B. So an ancestor of w in $r + 2$ will decide that x is not famous in B. □

Given the last 3 lemmas/theorems, we know that every round will eventually have all its witnesses classified as famous or not famous by universal consensus, with at least one of the witnesses being famous. After that, the set of famous witnesses for that round will never change, even if more events are added to the hashgraph. This set of famous witnesses can therefore act as a judge, to define a total order on all the events that have reached them, and a consensus timestamp on every event.

Theorem 5.19 (Byzantine Fault Tolerance Theorem)**.** *Each event x created by an honest member will eventually be assigned a consensus position in the total order of events, with probability 1.*

Proof: All honest members will eventually learn of x, by the definition of honest and the assumptions that the attackers who control the internet must eventually allow any two honest members to communicate. Therefore, there will eventually be a round where all the unique famous witnesses are descendants of x. Therefore in that round, or possibly earlier, there will be a round r where all the famous witnesses are descendants of x. Then x is assigned a received round of r, and a consensus timestamp of the median of when those members first received it, and its consensus place in history will be fixed. Furthermore, it is not possible to later discover a new event y that will come before x in the consensus order. Because, to come earlier in the consensus history, y would have to have a received round less than or equal to r. That would mean that all the famous witnesses in round r must have received y. But once the set of famous witnesses is known for a round, all of their ancestors are also known, so there is no way to discover new ancestors for them in the future as the hashgraph grows. Furthermore, it isn't possible for a round to gain new famous witnesses in the future, once the famousness of all the known witnesses in that round are known. Any new round r witness that is discovered in the future will *not* be an ancestor of the known round $r + 1$ witnesses (of which there are more than $2n/3$), and so the consensus will immediately be reached that it is not famous. Therefore, once an event is assigned a place in the total order, it will never change its position, neither by swapping with another known event, nor by new events being discovered later and being inserted before it. □

6. Fairness

Most existing systems for distributed consensus can fail to be "fair" in their consensus ordering of transactions. To see this, first consider a stock market that is run by a single server. Alice and Bob each submit a bid to that server, with Alice submitting it just before Bob. If the server is *fair*, then it will count Alice's transaction as occurring before Bob's. For some applications, the exact order does not matter, but for a stock market it can be critically important that this decision be made fairly.

Now consider a distributed peer-to-peer system, where there is no single server, but there is a community that will reach consensus on whose transaction was first. It may still be critically important that the consensus decision is fair. But what should be the definition of "fair"?

The "fair" decision on transaction order could be defined as favoring whichever transaction was created first. But that would be bad. Alice might have created her transaction one second before Bob, while she was in a cabin in the woods, disconnected from the internet. Then the community would only hear of Bob's transaction, and would assume that Bob was first. A year later, when Alice finally emerges from the woods and rejoins the internet, the community would have to revise history in order to be "fair". That would cause a host of problems. So that wouldn't be an ideal definition of fairness. There needs to be a requirement that the transaction actually be sent to the community, in order to count as being first.

The "fair" decision could be defined as reflecting the order in which the transactions reached the current *leader*. But that would also be bad. The leader might be a member chosen by the Paxos algorithm. Or it might be whichever member currently has a turn in a round-robin system. In a proof-of-work system, it would be whichever miner manages to solve a puzzle first. In any case, the leader could arbitrarily decide to ignore either Alice's or Bob's transaction for a period of time, delaying one of them, to force their transaction to come after the other. If the goal is distributed trust, then no single individual can be trusted.

The "fair" decision could be defined as reflecting when each transaction first reached a certain fraction of the entire community. This is a little better. The community is then ordering transactions by when the transactions first reached a virtual server, where "reaching the server" means reaching some fraction of the community as a whole. However, there are still issues. If the fair choice is defined as whichever transaction reached at least half of the community first, then there will be problems if Carol saw Alice first, Dave saw Bob first, and everyone else is evenly split on the question. This fails if Carol and Dave are both attackers who turn off their computers permanently before telling the community what they saw. In that case, the community could never reach a fair consensus, because they would be waiting forever on Carol and Dave to vote.

A better definition might be to say it is "fair" to consider Alice as being first if a significant fraction of the community received Alice's transaction before Bob's, and that fraction of the community then went on to communicate with most of the others quickly. Under this definition, if Alice and Bob are releasing their transactions to the gossip network at almost the same time, and both spread at about the same rate, then the consensus could go either way, and still be considered fair. However, if Alice gossips her transaction before Bob, beating him by just over the duration of a single gossip sync, then it might be expected that as both transactions spread

exponentially, doubling the number of members reached on each sync, in the end over 2/3 of the population will hear of Alice before Bob, and less than 1/3 will favor Bob. So in that case, it would be fair to favor Alice over Bob in the consensus. The hashgraph consensus algorithm is fair in this sense. The members who are online and regularly participating will generate a set of events called *famous witnesses*, and the consensus decision will be that the "first" transaction is whichever transaction reached the majority of that set first. If a small set of members are offline, or are partitioned so that they cannot communicate with the rest, then they will not have famous witnesses, and so having a transaction reach them will not count as having reached the community as a whole. But if the members in that set are communicating with the rest, then they will count as famous witnesses, and they will help decide who reached "the community" first.

There are attacks against this system that would be not be considered to be a failure of the consensus system, because they would be equally effective against a single-server solution. For example, the Byzantine proofs assume the attackers control the internet, and can delay arbitrary messages. If attackers actually had that power, they could simply disconnect Alice from the internet for as long as it takes for Bob to send a transaction and have it recorded. This could be done on the real internet by launching a denial of service attack, flooding every computer with packets from Bob in an attempt to prevent Alice from communicating. Of course, this would also be effective if Alice were communicating with a central server, so it could be considered more a failure of the internet than a failure of the consensus system.

Similarly, Bob could gain an advantage over Alice by buying more bandwidth, so that his gossips reach more people, faster. If he has 8 times the bandwidth of Alice, so that he can send his transaction initially to 8 members in the time Alice sends to 1, then he can gain an advantage of the time of about 3 gossip syncs. This is not considered a failure. If his message actually reaches the world before hers, then he should have the credit for it. This is similar to the current stock markets, where companies spend large sums of money for slightly faster connections, in order to reach the central server faster. So the consensus algorithm would not be considered "unfair" in this case, because it is behaving the same as a central server.

7. Generalizations and enhancements

7.1. **proof-of-stake.** So far, it has been assumed that every member is equal. The above algorithms refer to things depending on "more than $2n/3$ of the members" and "at least half of the famous witness events". They also use the idea of a "median" of a set of numbers. The proof shows Byzantine convergence when more than $2n/3$ of the members are honest.

It is easy to modify the algorithm to allow members to be unequal. Each member can be assumed to have some positive integer associated with them, known as their "stake". Then, the votes would be replaced with weighted voting, and the medians with weighted medians, where votes are weighted proportional to the voter's stake. In all of the above definitions, algorithms, and proofs, define "more than $2n/3$ members" to mean "a set of members whose total stake is more than $2n/3$, where n is the total stake of all members". The "median of the timestamps of events in S" would become "the weighted median of the timestamps in S, weighted by the stake of the creator of each event in S". The weighted median can be thought of as

taking each event y in S, and putting multiple copies of the timestamp of y into a bag, where the number of copies equals the stake of the member who created y. Then take the median of the timestamps in the bag.

The Byzantine proof applied as long as the attackers constituted less than 1/3 of the population. With these new definitions, it will now apply when the attackers together have a stake that is less than 1/3 of the total stake of all members.

This new proof-of-stake system is more general than the unweighted system. It can still be used to implement the unweighted system, by simply giving every member a stake of 1. But it can also be used to provide better behavior. For example, the stake might be proportional to the degree to which a member is trusted. Perhaps members who have been investigated in some way should be trusted more than others. Or it could be used to give greater weight to members who have a greater interest in the system as a whole working properly. A cryptocurrency might use each member's number of coins as their stake, on the grounds that those with more coins have a greater interest in ensuring the system runs smoothly. Or a community could be started by a group of members with mutual trust, each of which is given an equal stake. Then, each existing member could be allowed to invite arbitrarily many new members to join, subject to the constraint that the inviter must split their stake with the invitee. This would discourage a Sybil attack, where one member invites a huge number of sock puppet accounts, in order to control the voting.

The "stake record" is the list of members and the amount of stake owned by each member. So far, it has been assumed that the stake record is universally known, and is unchanging. It is easy to relax that assumption.

Assume that there is a particular form of transaction that changes the stake record. The community might set up rules at the beginning, governing which such transactions are valid. For example, each member could be allowed to invite other members, up to a total of at most 10 new members. Or perhaps anyone inviting a new member must simultaneously give the new member a portion of their own stake. The validity of such a transaction might depend on the exact order of the transactions in the consensus order. For example, if the rule is that only one new member can be invited, and Alice invites Carol at the same time Bob invites Dave, then then whichever invitation comes first in the consensus order will succeed, and the other will fail.

All of this can be accommodated. When the consensus algorithm finishes deciding the question of which round r firsts are famous, at that moment it becomes possible to find exactly which events will have a received round of r, and to calculate their exact position in the consensus order. At that time, each of the transactions in those events can be processed, and the rules can be consulted to see which are valid, and the valid transactions can be applied. This may change the stake record.

If the stake record does change, then the algorithm should be re-run for all events in round r and later. This may change the calculations of which events are strongly seen, of event round numbers, of which events are witnesses, and of which are famous witnesses.

Note that when deciding which round r witnesses are famous, the calculations are done using the old stake record. The voting for round r may continue several rounds into the future, all using the old stake record. Once round r is settled, the

future rounds will reshuffle, and the calculations for round $r + 1$ famous witnesses will be done using the new stake record.

This approach allows all members to be in agreement on exactly what stake record is being used for any given calculation. That ensures that they will always agree on the results of those calculations. And Byzantine agreement will still be guaranteed with probability one.

7.2. **signed state.** Another enhancement to the system is to have signed states. Once consensus has been reached on whether each witness in round r is famous or not, a total order can be calculated for every event in history with a received round of r or less. It is also guaranteed that all other events (including any that are still unknown) will have a received round greater than r. In other words, at this point, history is frozen and immutable for all events up to round r. A member can therefore take all the transactions from those events, and feed them into a database in the consensus order, and calculate the state that is reached after processing those transactions. Every member will calculate the same consensus order, so every member will calculate the same state. This is a *consensus state*. Each member can take the hash of this state and digitally sign it, and put the signature into a new transaction. Soon after, every member will have received by gossip many signatures for the consensus state. Once signatures are collected from at least 1/3 of the population, that consensus state, along with the set of signatures, constitutes a *signed state* that is an official consensus state for the system at the start of round r. It can be given to people outside the community, and they can check the signatures, and therefore trust the state. At this point, a member can feel free to delete all the transactions that were used to create the state, and delete all the events that contained those transactions. Only the state itself needs to be kept. It might be possible to do this every few minutes, so there will never be a huge number of transactions and events stored. Only the consensus state itself. Of course, a member is free to preserve the old events, transactions, and states, perhaps for archive or audit purposes. But the system is still immutable and secure, even if everyone discards that old information.

Given the assumption that less than 1/3 of the population is dishonest, the signed state is guaranteed to have at least one honest signature, and so can be trusted to represent the community consensus, as found by the consensus algorithm. If the set of members (or their stake) can change over time, then that stake record (and its history) will also be part of the state. The threshold of 1/3 could be replaced with something else, such as more than 2/3, and the system would still work.

7.3. **Efficient gossip.** The gossip protocol makes very efficient use of bandwidth. Suppose there are enough transactions being created that every event contains at least one transaction. In any replicated state machine, using a point-to-point network such as the internet, it will be necessary for each member to receive each signed transaction once, and to also send each signed transaction on average once. For the hashgraph gossip, the same is true, except that the signature is for the event containing the transaction, rather than for the transaction itself. The only additional overhead is the two hashes and the timestamp, plus the array of counts at the start of the sync. However, the hashes themselves don't have to be sent over the internet. It is sufficient to merely send the identity of the creator of the event, and the sequence number of its other-parent.

For example, suppose the 100th event created by Alice has an other-parent that is Ed's 50th event. If this event by Alice is sent from Bob to Carol during a sync, Bob could skip sending Carol the two hashes in the event. Instead, he could tell Carol that this is an event by Alice, and that the other-parent is Ed's 50th event. Since Bob is only sending Carol events she doesn't have according to their initial counts, Carol will know that this must be Alice's 100th event, since the last one she knows about by Alice is Alice's 99th event. So Bob doesn't have to send the hash of that self-parent, and doesn't have to send the sequence number 100. He just has to send the fact that it is by Alice. Similarly, he must send that the other-parent is by Ed, and that it is Ed's 50th event. So instead of two, large hashes, Bob is simply sending the triplet (Alice, Ed, 50). With some care, the identities and sequence numbers can be compressed to a byte or two each, so the triplet will required only 3 to 6 bytes. This is small overhead compared to the signature (which is 64 bytes for a 512-bit signature) and the transactions within the event (perhaps averaging 100 bytes or more). So if each event contains at least one transaction, then there is almost no overhead for gossiping a hashgraph, beyond simply gossiping the transactions themselves.

And because voting is virtual, there is no other bandwidth cost at all in order to achieve consensus. In this sense, the bandwidth required for hashgraph consensus is very close to the theoretical limit, which would be the bandwidth needed to simply send the signed and dated transactions themselves.

A system that merely sent the transactions could save bandwidth by not attaching timestamps to the transactions, if the application didn't need timestamps. Hashgraph consensus can do the same. In that case, the "timestamp" within an event would simply be an integer that is its self-parent's "timestamp" plus one. When Bob sends an event to Carol, that sequence number can be calculated by Carol, so there is no need for Bob to actually send it over the internet.

A system that only sent transactions could also save bandwidth by grouping together several transactions by the same creator, and attaching only a single signature to the list, rather than one per transaction. Hashgraph can do the same, by putting several transactions into a single event, and so having only a single signature for the list.

So the bandwidth requirements of hashgraph consensus are very close to the theoretical minimum in all cases.

7.4. **Fast elections.** That second part of the algorithm is a Byzantine agreement step for deciding fame. It has an interesting property. When a group of members are all online and all participating regularly, the Byzantine agreement will be applied to a set of elections where almost all the voters start with identical votes. That is because a round $r+1$ witness will strongly see many of the round r witnesses, so a round might be expected to last about two "gossip periods", where a gossip period is the time it takes for a message to propagate through the entire community. This should be the time to do $log_2(n)$ syncs, when there are n members online. For a round $r+1$ witness x to vote YES on the fame of a round r witness y, it isn't necessary for x to strongly see y. It can merely see y. It would be expected that y would propagate to all the online members in a single gossip period. So there is an overwhelmingly high probability it will propagate to them within two gossip periods. So in practice, when everyone is online and participating, the fame of

witnesses is almost always decided immediately, without the need for many rounds of voting.

Similarly, if y is a round r witness, but was created by a member who was asleep and then awoke just before the end of round r , then it is likely that almost all round $r+1$ witnesses will vote NO on y, and the election will again end immediately. There is a small window of time, on the order of the duration of a single sync, in which a member awakening and creating y can cause the round $r+1$ witnesses to start with a close to even vote split. If the online members are all choosing each other randomly and syncing frequently, then such a result will converge to a decision in about 3 rounds, with a probability of only a few percent for more than 3 rounds, and of less than a tenth of a percent for more than 6 rounds. If an attacker completely controls the internet, they can cause this to drag on for exponentially many rounds. This can be reduced to a constant expected number of rounds by using a cryptographic "shared coin" protocol, rather than the "middle bit of the signature" described in the above algorithm. The middle bit is intended to be like each member having an independent random coin flip that the attacker couldn't predict ahead of time. A shared coin protocol is the same, but ensures all members end up with the same "random" result. This addition would reduce the theoretical worst-case expected time. But such an addition seems unlikely to be worth the effort in practice. If an attacker can truly control the internet enough to keep the honest members from synching randomly with each other for a long period, then the attacker likely has the power to simply block the honest users from accessing the internet at all. So a shared coin seems to be of only theoretical interest here. But using a shared coin is always an option.

7.5. **Efficient calculations.** The first part step of the algorithm is to assign a round of either r or $r+1$ to an event, based on whether it can strongly see enough round r events. So it is necessary to calculate whether a round r witness event x can be strongly seen by an arbitrary event y. The following is one way to calculate that answer.

Give each event a sequence number that is one greater than the sequence number of its self-parent. Store an array for y and an array for x. The y array remembers the sequence number of the last event by each member that is an ancestor of y. The array for x remembers the sequence number of the earliest event by each member that is a descendant of x. Compare the two arrays, and find how many elements in the y array are greater than or equal to the corresponding element of the x array. If there are more than $2n/3$ such matches, then y strongly sees x. The comparison of the x and y arrays can be sped up by multithreading (to use more cores), packing multiple elements into one integer (to use the ALU more efficiently), using assembly language (to access the CPU vector instructions) or using the GPU (for more vector parallelism).

8. Conclusions

A new system has been presented, based on the Swirlds hashgraph data structure, and the Swirlds hashgraph consensus algorithm. It is fair, fast, Byzantine fault tolerant, and extremely bandwidth efficient due to virtual voting. The algorithm is given in pseudocode in the figures, using an imperative language, but it is also very natural to describe it in a functional form. The appendix gives the algorithm in a functional form, which is concise, and may be of interest.

References

[1] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[2] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

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

[4] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[5] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[6] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. Cryptology ePrint Archive, Report 2016/199, 2016. http://eprint.iacr.org/.

[7] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[8] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. posted to the internet November, 2008, 2008. http://bitcoin.org/bitcoin.pdf.

[9] Giulio Prisco. Intel develops 'Sawtooth Lake' distributed ledger technology for the Hyperledger project. *Bitcoin Magazine*, April 2016.

[10] Dag-Erling Smorgrav. FreeBSD quarterly status report. Posted on FreeBSD.org, 2013. http://www.freebsd.org/news/status/report-2013-09-devsummit.html#Security.

[11] Miguel Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

9. Appendix A: Consensus algorithm in functional form

An event is a tuple $e = \{d, h, t, i, s\}$ where:

d	$= data(e)$	= the "payload" data, which may include transactions.
h	$= hashes(e)$	= a list of hashes of the event's parents, self-parent first.
t	$= time(e)$	= creator's claimed date and time of the event's creation.
i	$= creator(e)$	= creator's ID number.
s	$= sig(e)$	= creator's digital signature of {d,h,t,i}.

$$
\begin{aligned}
\text{parents}(x) &= \text{set of events that are parents of event x} \\
\text{selfParent}(x) &= \text{the self-parent of event x, or } \varnothing \text{ if none} \\
n &= \text{the number of members in the population} \\
c &= \text{frequency of coin rounds (such as } c = 10) \\
E &= \text{the set of all events in the hashgraph} \\
E_0 &= \text{E} \cup \{\varnothing\} \\
\mathbb{T} &= \text{set of all possible } (time, date) \text{ pairs} \\
\mathbb{B} &= \{true, false\} \\
\mathbb{N} &= \{1, 2, 3, ...\} \\
\text{parents} &: E \to 2^E \\
\text{selfParent} &: E \to E_0 \\
\text{ancestor} &: E \times E \to \mathbb{B} \\
\text{selfAncestor} &: E \times E \to \mathbb{B} \\
\text{manyCreators} &: 2^E \to \mathbb{B} \\
\text{see} &: E \times E \to \mathbb{B} \\
\text{stronglySee} &: E \times E \to \mathbb{B} \\
\text{parentRound} &: E \to \mathbb{N} \\
\text{roundInc} &: E \to \mathbb{B} \\
\text{round} &: E \to \mathbb{N} \\
\text{witness} &: E \to \mathbb{B} \\
\text{diff} &: E \times E \to \mathbb{I} \\
\text{votes} &: E \times E \times \mathbb{B} \to \mathbb{N} \\
\text{fractTrue} &: E \times E \to \mathbb{R} \\
\text{decide} &: E \times E \to \mathbb{B} \\
\text{copyVote} &: E \times E \to \mathbb{B} \\
\text{vote} &: E \times E \to \mathbb{B} \\
\text{famous} &: E \to \mathbb{B} \\
\text{uniqueFamous} &: E \to \mathbb{B} \\
\text{roundsDecided} &: \mathbb{N} \to \mathbb{B} \\
\text{roundReceived} &: E \to \mathbb{N} \\
\text{timeReceived} &: E \to \mathbb{T}
\end{aligned}
$$

$$
\begin{aligned}
\text{ancestor}(x,y) &= x = y \vee \exists z \in \text{parents}(x),\ \text{ancestor}(z,y) \\
\text{selfAncestor}(x,y) &= x = y \vee (\text{selfParent}(x) \neq \varnothing \wedge \text{selfAncestor}(\text{selfParent}(x),y)) \\
\text{manyCreators}(S) &= |S| > 2n/3 \wedge \forall x,y \in S, (x \neq y \implies \text{creator}(x) \neq \text{creator}(y)) \\
\text{see}(x,y) &= \text{ancestor}(x,y) \wedge \neg(\exists a,b \in E,\ \text{creator}(y) = \text{creator}(a) = \text{creator}(b) \wedge \\
&\quad \text{ancestor}(x,a) \wedge \text{ancestor}(x,b) \wedge \neg\,\text{selfAncestor}(a,b) \wedge \neg\,\text{selfAncestor}(b,a)) \\
\text{stronglySee}(x,y) &= \text{see}(x,y) \wedge (\exists S \subseteq E, \text{manyCreators}(S) \\
&\quad \wedge (z \in S \implies (\text{see}(x,z) \wedge \text{see}(z,y)))) \\
\text{parentRound}(x) &= \max(\{1\} \cup \{\text{round}(y) \mid y \in \text{parents}(x)\}) \\
\text{roundInc}(x) &= \exists S \subseteq E, \text{manyCreators}(S) \\
&\quad \wedge (\forall y \in S, \text{round}(y) = \text{parentRound}(x) \wedge \text{stronglySee}(x,y)) \\
\text{round}(x) &= \text{parentRound}(x) + \begin{cases} 1 & \text{if roundInc}(x) \\ 0 & \text{otherwise} \end{cases} \\
\text{witness}(x) &= (\text{selfParent}(x) = \varnothing) \vee (\text{round}(x) > \text{round}(\text{selfParent}(x)) \\
\text{diff}(x,y) &= \text{round}(x) - \text{round}(y) \\
\text{votes}(x,y,v) &= |\{z \in \text{E} \mid \text{diff}(x,z) = 1 \wedge \text{witness}(z) \wedge \text{stronglySee}(x,z) \wedge \text{vote}(z,y) = v\}| \\
\text{fractTrue}(x,y) &= \tfrac{\text{votes}(x,y,true)}{(\text{votes}(x,y,true)+\text{votes}(x,y,false))} \\
\text{decide}(x,y) &= (\text{selfParent}(x) \neq \varnothing \wedge \text{decide}(\text{selfParent}(x),y)) \vee (\wedge\,\text{witness}(x) \wedge \text{witness}(y) \\
&\quad \wedge \text{diff}(x,y) > 1 \wedge (\text{diff}(x,y) \bmod c > 0) \wedge (\exists v \in B, \text{votes}(x,y,v) > \tfrac{2n}{3}))) \\
\text{copyVote}(x,y) &= (\neg\,\text{witness}(x)) \vee (\text{selfParent}(x) \neq \varnothing \wedge \text{decide}(\text{selfParent}(x),y) \\
\text{vote}(x,y) &= \begin{cases} \text{vote}(\text{selfParent}(x),y) & \text{if copyVote}(x) \\ 1 = \text{middleBit}(\text{signature}(x)) & \text{if } \neg\,\text{copyVote}(x) \\ & \quad \wedge\ (\text{diff}(x,y) \bmod c = 0) \\ & \quad \wedge\ (\tfrac{1}{3} \leq \text{fractTrue}(x,y) \leq \tfrac{2}{3}) \\ \text{fractTrue}(x,y) \geq \tfrac{1}{2} & \text{otherwise} \end{cases} \\
\text{famous}(x) &= \exists y \in E, \text{decide}(y,x) \wedge \text{vote}(y,x) \\
\text{uniqueFamous}(x) &= \text{famous}(x) \wedge \neg\exists y \in E, y \neq x \wedge \text{famous}(y) \\
&\quad \wedge \text{round}(x) = \text{round}(y) \wedge \text{creator}(x) = \text{creator}(y) \\
\text{roundsDecided}(r) &= \forall x \in E, ((\text{round}(x) \leq r \wedge \text{witness}(x)) \implies \exists y \in E, \text{decide}(y,x)) \\
\text{roundReceived}(x) &= \min(\{r \in \mathbb{N} \mid \text{roundsDecided}(r) \wedge (\forall y \in E, \\
&\quad (\text{round}(y) = r \wedge \text{uniqueFamous}(y)) \implies \text{ancestor}(y,x) \\
\text{timeReceived}(x) &= \text{median}(\{\text{time}(y) \mid y \in E \wedge \text{ancestor}(y,x) \wedge \\
&\quad (\exists z \in E, \text{round}(z) = \text{roundReceived}(x) \wedge \text{uniqueFamous}(z) \\
&\quad \wedge \text{selfAncestor}(z,y\}) \wedge \neg(\exists w \in E, \text{selfAncestor}(y,w) \wedge \text{ancestor}(w,x))\})
\end{aligned}
$$