Hedera Hashgraph

Hedera: A Public Hashgraph Network & Governing Council

The trust layer of the internet

Dr. Leemon Baird, Mance Harmon, and Paul Madsen

Vision

To build a trusted, secure, and empowered digital future for all.

Mission

We are dedicated to building a trusted and secure online world that empowers you.

Where you can work, play, buy, sell, create and engage socially.

Where you have safety and privacy in your digital communities.

Where you are confident when interacting with others.

Where this digital future is available to all.

Hello future.

Executive Summary

Distributed ledger technologies (DLT) are disrupting and transforming existing markets in multiple industries. However, in our opinion there are five fundamental obstacles to overcome before distributed ledgers can be widely accepted and adopted by enterprises. In this paper we will examine these obstacles and discuss why Hedera Hashgraph is well-suited to become the world's first mass-adopted public distributed ledger, supporting a vast array of applications.

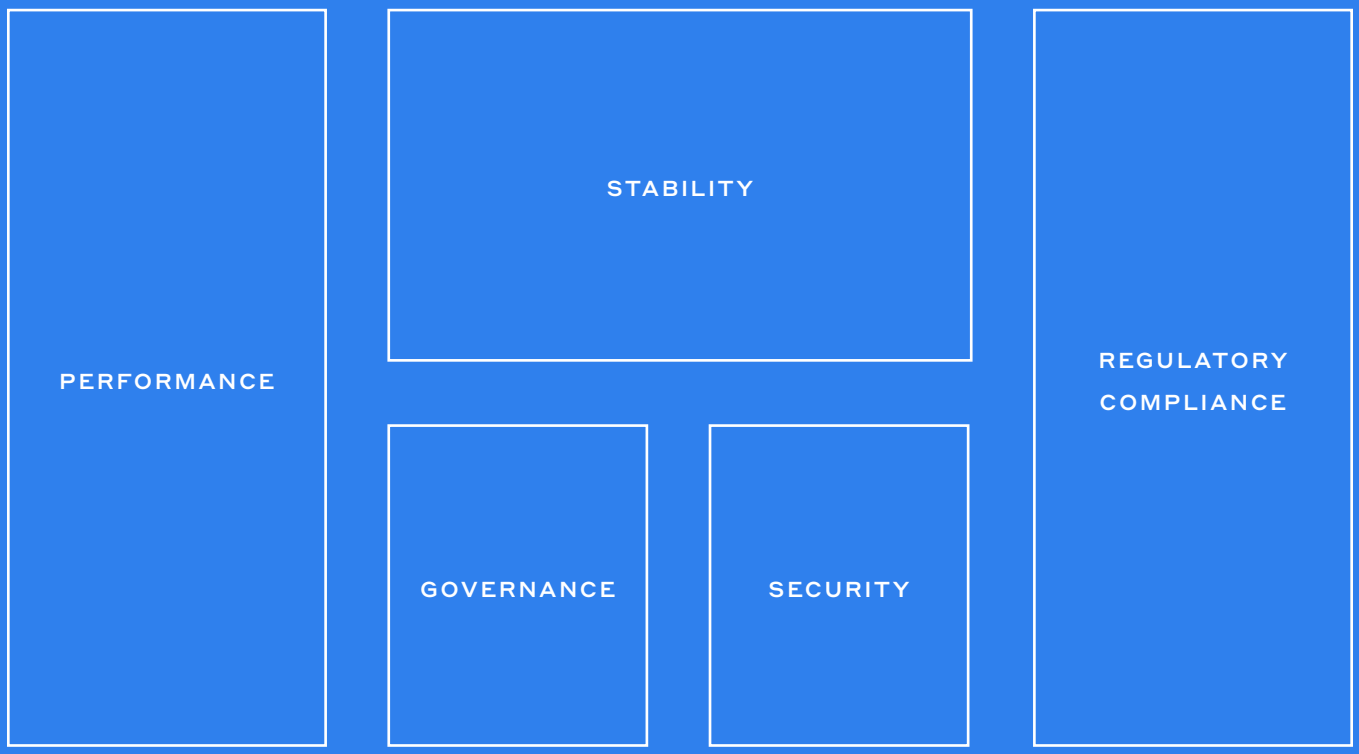
1 **PERFORMANCE** - The most compelling use cases for DLT require hundreds of thousands of transactions per second, and many require consensus latency measured in seconds. These performance metrics are orders of magnitude beyond what current public DLT platforms can achieve.

2 **SECURITY** - If public DLT platforms are to facilitate the transfer of trillions of dollars of value, they will be targeted by hackers, and so will need the strongest possible network security. Having the strongest possible security starts with the consensus algorithm itself, with its security properties formally proven mathematically. Vectors of security vulnerabilities shouldn't be mitigated; they should be eliminated entirely. To address the performance issues cited above, other public DLT platforms are making compromises with regard to decentralization, and in so doing are potentially compromising security.

3 **GOVERNANCE** - A general-purpose public ledger should be governed by representatives from a broad range of market sectors, each with world-class expertise in their respective industries, and also selected to provide global geographic representation for all markets. Those responsible for network governance need technical expertise so they can competently manage the technical roadmap. They need business expertise so they can manage business operations of the organization. They need expertise in economics and currency markets so they can manage the cryptocurrency. They need legal expertise to help navigate the evolving regulatory environment. In other words, the network should be governed be by those globally recognized as world leaders in their respective industries, and representative of every market in the world.

4 **STABILITY** - Without technical and legal mechanisms to enforce the decisions of the governing body, public platforms are at risk of devolving into chaos. Strong security and mature governance will enable a stable platform—one that engenders the necessary trust and confidence among those that would build commercial or sensitive applications on it.

5 **REGULATORY COMPLIANCE** - We expect that governments will continue to increase oversight of public ledgers and associated cryptocurrencies and tokens. We consider that a public distributed ledger must be capable of providing necessary tools for all members of its ecosystem to comply with applicable laws and regulations and enable appropriate Know Your Customer (KYC) and Anti Money Laundering (AML) checks.



What is required to move the DLT industry forward and enable it to realize its full potential?

A platform that provides a combination of high performance, strong security, stable governance, and both technical and legal controls to ensure the platform's stability. Only then do we think mainstream markets will trust a DLT platform enough to adopt it.

Introducing Hedera – a public hashgraph network and governing body designed to address the needs of mainstream markets.

The Hedera network will be governed by a council of leading global enterprises, across multiple industries and geographies. Its vision is a cyberspace that is trusted, secure, and without the need for central servers. Its licensing and governance model protects users by eliminating the risk of forking, guaranteeing the integrity of the codebase, and providing open access to review the underlying software code. All governing members of the Hedera Council will have equal voting rights, and Council membership is term-limited (with the exception of Swirlds, Inc.), ensuring that governance is decentralized.[1]

Hedera is distributed ledger platform that resolves the factors that constrain adoption of public DLT by the mainstream.

1. **PERFORMANCE** - The platform is built on the hashgraph distributed consensus algorithm, invented by Dr. Leemon Baird. The hashgraph consensus algorithm provides near-perfect efficiency in bandwidth usage and consequently can process hundreds of thousands of transactions per second in a single shard (a fully-connected, peer-to-peer mesh of nodes in a network). Consensus latency is measured in seconds, not minutes, hours, or days.

2. **SECURITY** - Hashgraph achieves the gold standard for security in the field of distributed consensus: asynchronous Byzantine Fault Tolerance (aBFT). Other platforms that use coordinators, leaders, or communication timeouts tend to be vulnerable to Distributed Denial of Service (DDoS) attacks against those vulnerable areas. Hashgraph is resilient to these types of attacks against the consensus algorithm, and achieves the theoretical limits of security defined by aBFT. Achieving this level of security at scale is a fundamental advance in the field of distributed systems as it is the gold standard for security in this category.

   Many applications require that the consensus order of transactions match the actual order in which the transactions are received by the network. It should not be possible for a single party to prevent the flow of transactions into the network, nor influence the order of transactions in the eventual network consensus. A fair consensus algorithm ensures that if a user can submit a transaction to the network at all, then the transaction will be received by the network and the order in which it was received will be a fair ordering. Hashgraph uniquely ensures that the actual order transactions are received by the network will be reflected in the consensus order. In other words, hashgraph ensures both *Fair Access* and *Fair Ordering*.

   Formal proofs of the ABFT and fairness properties for the hashgraph consensus algorithm exist and have been available for public review since June, 2016. Furthermore, the hashgraph algorithm has been validated as ABFT by a math proof checked by computer using the Coq system in October, 2018.[2]

3. **GOVERNANCE** - The Hedera Network will be governed by a council of up to 39 leading global enterprises. Council members will bring needed experience in process and business expertise that has been absent in previous public ledger platforms. Council membership is designed (i) to reflect a range of industries and geographies, (ii) to have highly respected brands and trusted market positions, and (iii) to encompass competing perspectives. The terms of governance ensure that no single Council member will have control, and no small group of members will have undue influence over the body as a whole.

4. **STABILITY** - Hedera relies on both technical and legal controls to ensure the stability of the platform.

Hedera technical controls enable two capabilities.

i) **First, the hashgraph technology ensures that software clients validate the pedigree of the Hedera hashgraph prior to use through a shared state mechanism.** It isn't possible for a network node to fork the official version of the Hedera hashgraph platform, make changes, and then have those changes accepted as valid. If the original hashgraph platform and the copy are changed independently, software clients will know which is the valid version and which is not.

ii) **Second, the hashgraph technology makes it possible for the Hedera Council to specify the software changes to be made to network nodes and to ensure precisely when those changes are adopted, and to guarantee that they are adopted.** When the Hedera Council releases a software update, all honest network providers will have their software automatically update, and all will do so at exactly the same moment in history. Anyone with invalid software will no longer be able to modify the hashgraph and have the world accept their version of the hashgraph as legitimate.

Hedera legal controls ensure the platform will not fork into a competing platform and cryptocurrency.

iii) The Hedera codebase will be governed by the council, and will be released for public review with Version 1.0. It will not be open source, but anyone will be able to read the source code, recompile it, and verify that it is correct. No license will be required to use the Hedera platform. No license will be required to write software that uses the services of the Hedera platform. No license will be required to build smart contracts on top of the Hedera platform. Applications built upon the Hedera platform can be open source or proprietary. They do not require any license or any approval from Hedera. Swirlds and Hedera will simultaneously embrace open review, while bringing stability by using hashgraph software patents defensively to prevent forks. In this way, Hedera will provide a transparent codebase that will provide the stability that markets demand for mainstream adoption of a public ledger.

The combination of technical and legal controls provide the governing body with the mechanisms needed to enable meaningful governance, and to bring the stability that we think is required for broad-based adoption.

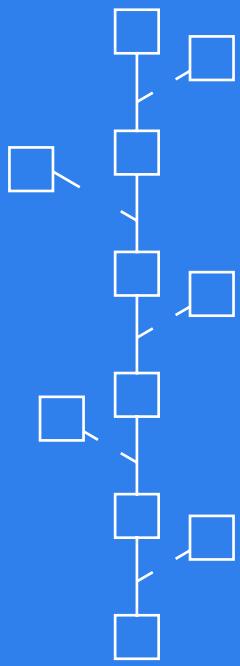
5. **REGULATORY COMPLIANCE** - The Hedera technical framework includes an Opt-In Verified Identity mechanism that will give users and developers a choice to bind verified identities to otherwise pseudonymous Hedera network accounts, which is a fundamental tool for enterprises to comply with existing KYC/AML regulations. This will be completely optional for users, and each user can decide what kinds of credentials, if any, to reveal. We intend to work with regulators and provide such tools to developers and enterprises to enable the same level of protection to public distributed ledgers as is currently present in the financial system.

[1] Swirlds, Inc. (Swirlds) is a Delaware corporation that holds the patent titles to the hashgraph consensus algorithm. Swirlds will be a permanent member of the Hedera Hashgraph Council.
[2] See Appendix 3 for a full definition of the hashgraph algorithm, including proofs of ABFT. Furthermore, see https://hedera.com/platform#security for the formalized ABFT proof.

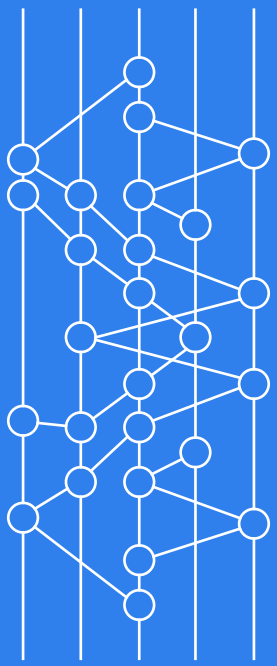
Part 1
An introduction to Hedera Hashgraph

The hashgraph data structure and consensus algorithm provides a new platform for distributed consensus. This introduction gives an overview how hashgraph works, and of some of its properties.

The goal of a distributed consensus algorithm is to allow a community of users to come to an agreement on the order in which some of them generated ***transactions***, when no single member is trusted by everyone. In this way, it is a system for generating trust, when individuals do not already trust each other. Hashgraph achieves this in a fundamentally new way.



BLOCKCHAIN



HASHGRAPH

A blockchain is like a tree that is continuously pruned as it grows - this pruning is necessary to keep the branches from growing out of control. In hashgraph, rather than pruning new growth, such growth is woven back into the body of the hashgraph.

In both blockchain and hashgraph, any member can create a transaction, which will eventually be put into a container (the "block"), and will then spread throughout the distributed network. In blockchain, those containers are intended to form a single, long chain. If two miners create two blocks at the same time, the network nodes will eventually choose one to continue, and discard the other one. It's like a growing tree that is constantly having all but one of its branches chopped off.

In hashgraph, every container of transactions is included—none are discarded—so it is more efficient than blockchains. All the branches continue to exist forever, and eventually grow back together into a single whole.

Furthermore, blockchain fails if the new containers arrive too quickly, because new branches sprout faster than they can be pruned. That is why blockchain needs proof-of-work or some other mechanism to artificially slow down the growth. In hashgraph, though, nothing is thrown away. There is no harm in the hashgraph data structure growing quickly. Every member can create transactions and containers whenever they want. So it is very simple, and tends to be very fast.

Finally, because the hashgraph doesn't require pruning, it allows more powerful mathematical guarantees, such as *Byzantine agreement* and *fairness*. Distributed databases such as Paxos are Byzantine, but do not guarantee fairness in the ordering of transactions. Blockchain is neither Byzantine nor fair. Hashgraph is both Byzantine and fair.

The hashgraph algorithm accomplishes being ***fair, fast, Byzantine, ACID compliant, efficient, inexpensive, timestamped, and DoS resistant.***

Performance

COST

The hashgraph is **inexpensive** in that it avoids energy-intensive proof-of-work. Individuals and organizations running hashgraph nodes do not need to purchase expensive custom mining rigs. Instead, they can run readily available hardware.

EFFICIENCY

The hashgraph is **100% efficient**, as that term is used in the blockchain community. In blockchain, work is sometimes wasted mining a block that later is considered stale and is discarded by the community. In hashgraph, the equivalent of a "block" never becomes stale. Hashgraph is also efficient in its use of bandwidth. Whatever is the amount of bandwidth required *merely* to inform all the nodes of a given transaction (even without achieving consensus on a timestamp for that transaction), hashgraph adds only a very small overhead beyond that absolute minimum. Additionally, hashgraph's voting algorithm does not require any additional messages be sent in order for nodes to vote (or those votes to be counted) beyond those messages by which the community learned of the transaction itself.

THROUGHPUT

The hashgraph is ***fast***. It is limited only by the bandwidth. If each member has enough bandwidth to download and upload a given number of transactions per second, the system as a whole can handle close to that many. Even a fast home internet connection could be fast enough to handle all of the transactions of the entire VISA card network, worldwide.

THE FOLLOWING CHARTS GIVE REPRESENTATIVE PERFORMANCE RESULTS FOR THE HASHGRAPH.

Figure 1

Hashgraph Latency vs Throughput

1 region, m4.4xlarge

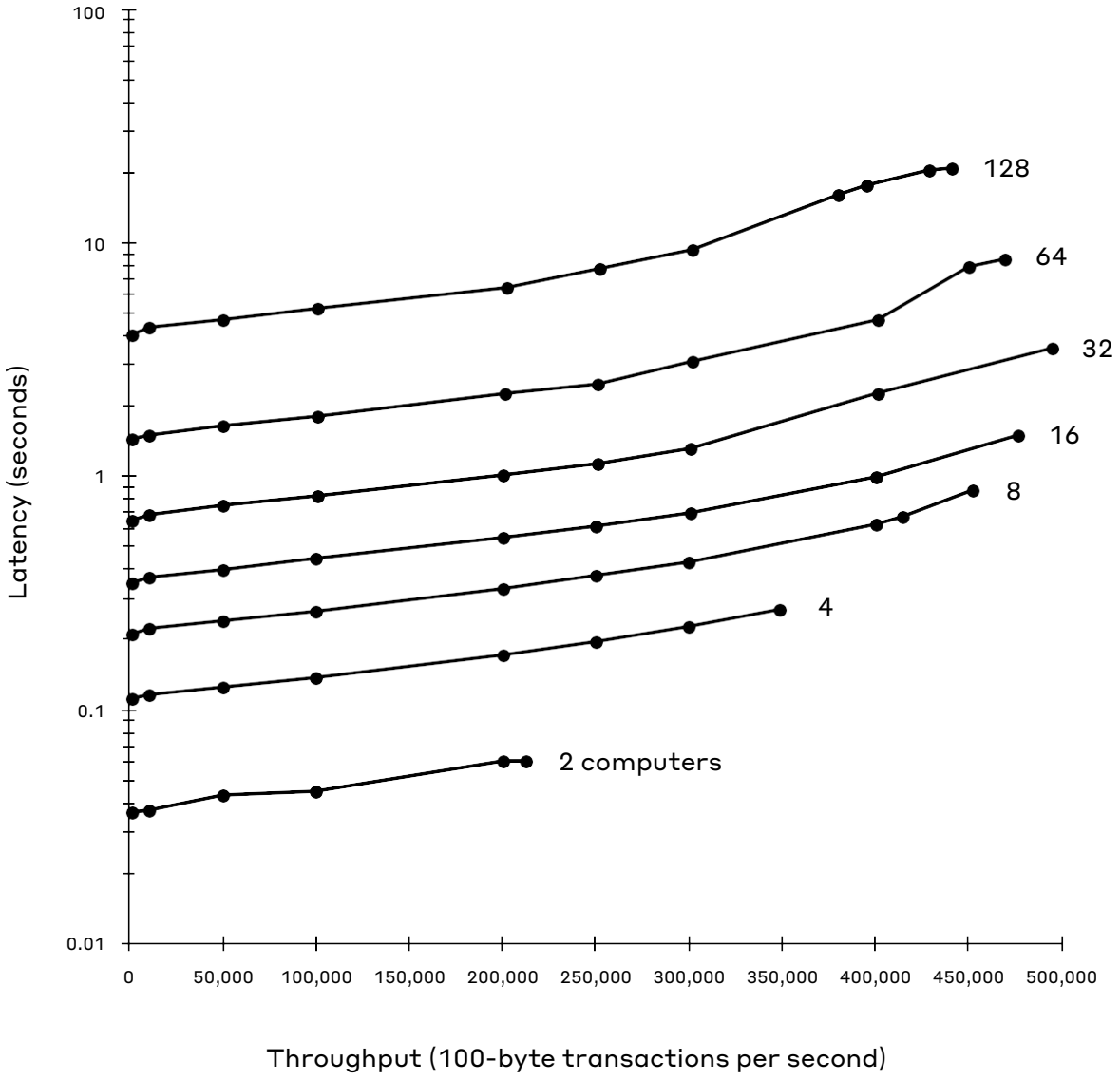


Figure 2

Hashgraph Latency vs Throughput

2 regions, m4.4xlarge

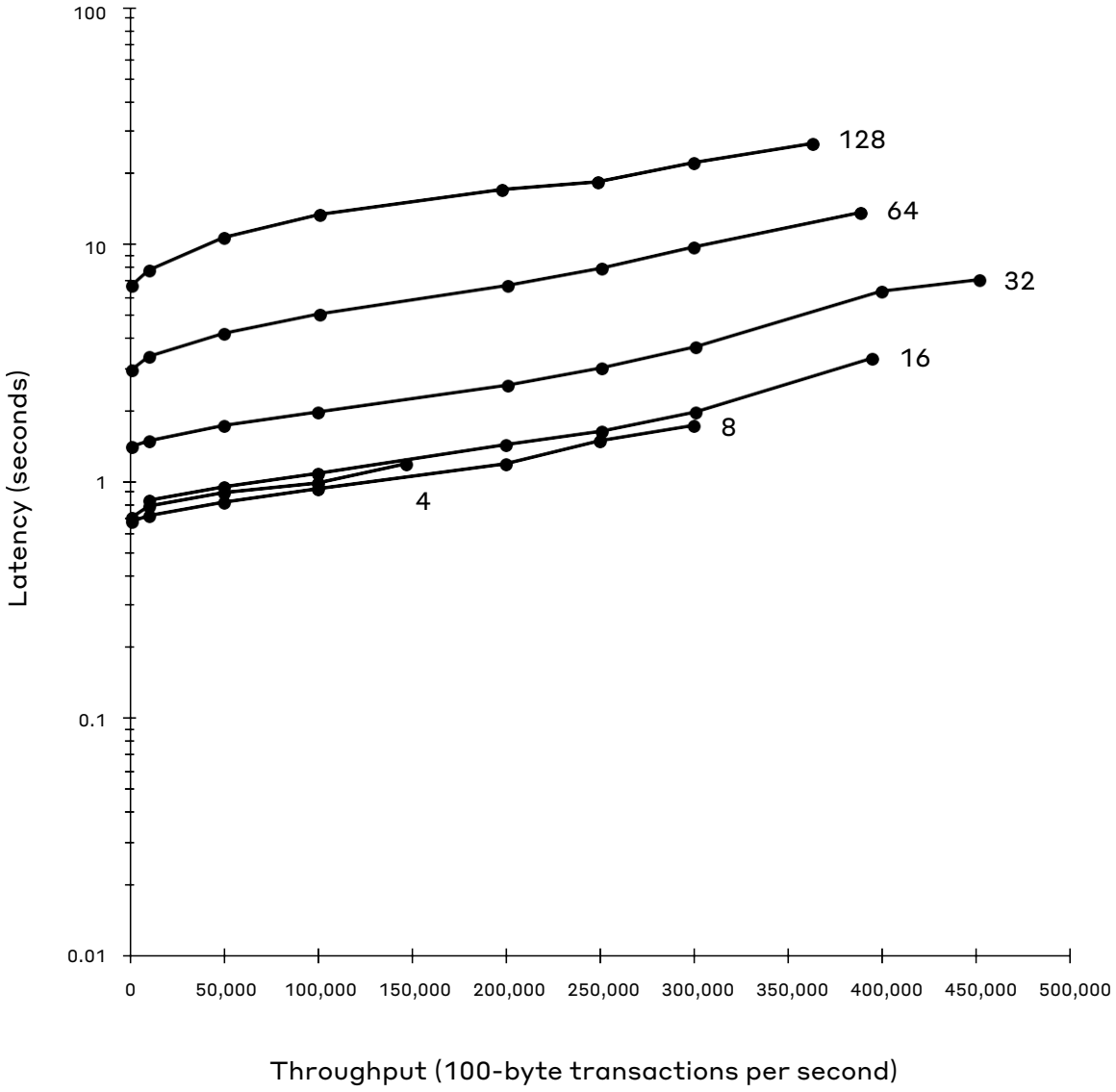
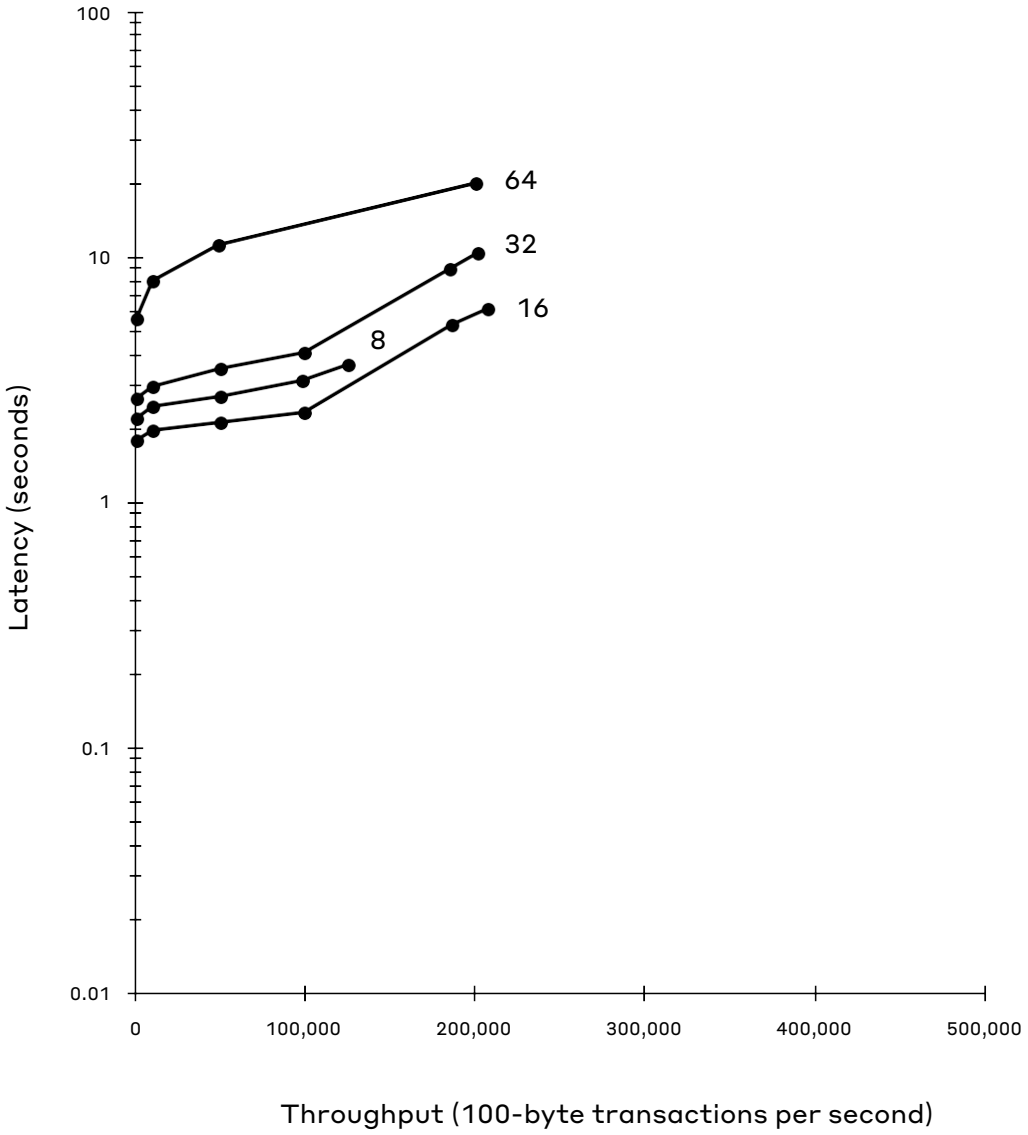


Figure 3

Hashgraph Latency vs Throughput
8 regions, m4.4xlarge



These tests were performed using Amazon AWS m4.4xlarge instances. Figure 1 shows performance for a single region (Virginia), while Figure 2 shows results for two regions on opposite sides of the continental United States (Virginia and Oregon), over 2,000 miles apart. Figure 3 shows results for 8 regions (Virginia, Oregon, Canada, Sao Paulo, Australia, Seoul, Tokyo, and Frankfurt).

Each line on the chart is for a different number of instances (computers), which is shown to the right of the line. In every case, the instances were distributed evenly across the number of regions being used.

The horizontal axis is the number of 100-byte transactions per second for which the ledger achieved consensus. In these experiments, this throughput ranges from less than 50,000 tps up to almost 500,000 tps. On most of the lines, the second dot from the left is 10,000 tps.

The vertical axis is the average number of seconds from when a node first creates a transaction until it knows the exact consensus order and consensus timestamp for it. This isn't just a time to a first confirmation—it is the time until a 100% certain finality is reached.

In all of the experiments, this latency was under 11 seconds. And various experiments had latencies down to less than 0.04 seconds.

In the graphs, there are clear tradeoffs between throughput, latency, number of computers, and geographic distribution. For 32 computers running at 50,000 transactions per second, consensus finality is reached in 3 seconds when the network is spread across 8 regions spanning the globe. When the network stretches only 2,000 miles across the US, this drops to 1.5 seconds. In a single region, it drops to 0.75 seconds.

If it is desired to keep the latency under the 7 seconds required by credit cards, while still achieving 200,000 transactions per second, it is possible to use 32 computers in eight regions, or use 64 computers in two regions, or use 128 computers in one region.

It is important to note that these tests are purely for achieving consensus on transaction order and timestamps. They do not include the time to process transactions. For example, if every transaction is digitally signed, then these results suggest that a great deal of processing power might be needed to verify hundreds of thousands of digital signatures per second. It is possible that GPU implementations could be helpful.

In addition, if a transaction is of the form "store this gigabyte file", then bandwidth limitations would greatly slow down the system.

STATE EFFICIENCY

Once a transaction occurs on the network, within **seconds** all the nodes in the network will know where it should be placed in history with **100% certainty**. More importantly, everyone will know that everyone else knows this. At that point, they can just incorporate the effects of the transaction and, unless needed for future audit or compliance, then discard it. So in a minimal cryptocurrency system, each member would only need to store the current balance of each account that isn't empty. They wouldn't need to remember the full history of the transactions that resulted in those balances all the way back to 'genesis'.

Security

CRYPTOGRAPHY

All communications are encrypted with TLS 1.2, all transactions are digitally signed, and the hashgraph is constructed using cryptographic hashes. All the algorithms and key sizes were chosen to be compliant with the CNSA Suite security standard. This is the standard required for protecting US government Top Secret information. It specifies using AES-256, RSA 3072, SHA-384, and ECDSA and ECDH with p-384 and using ephemeral keys for perfect forward secrecy.

ASYNCHRONOUS BYZANTINE FAULT TOLERANCE

The hashgraph is asynchronous Byzantine Fault Tolerant. This is a technical term meaning that no single member (or small group of members) can prevent the community from reaching a consensus. Nor can they change the consensus once it has been reached. Each member will eventually reach a point where they know for sure that they have reached consensus. Blockchain does not have a guarantee of Byzantine agreement, because a member never reaches certainty that agreement has been achieved (there's just a probability that rises over time). Blockchain is also non-Byzantine because it doesn't automatically deal with network partitions. If a group of miners is isolated from the rest of the internet, that can allow multiple chains to grow, which conflict with each other on the order of transactions.

It is worth noting that the term "Byzantine Fault Tolerant" (BFT) is sometimes used in a weaker sense by other consensus algorithms. But here, it is used in its original, stronger sense that (1) every member eventually knows consensus has been reached, (2) attackers may collude, and (3) attackers even control the internet itself (with some limits). Hashgraph is Byzantine, even by this stronger definition.

There are different degrees of BFT, depending on the assumptions made about the network and transmission of messages.

The strongest form of BFT is asynchronous BFT- meaning that it can achieve consensus even if malicious actors are able to control the network and delete or slow down messages of their choosing. The only assumptions made are that more than ⅔ are following the protocol correctly, and that if messages are repeatedly sent from one node to another over the internet, eventually one will get through, and then eventually another will, and so on. Some systems are partially asynchronous, which are secure only if the attackers do not have too much power and do not manipulate the timing of messages too much. For instance, a partially asynchronous system could prove Byzantine under the assumption that messages get passed over the internet in ten seconds. This assumption ignores the reality of botnets, Distributed Denial of Service attacks, and malicious firewalls.

A full technical report describing the hashgraph data structure and algorithm, including mathematical proofs that Hashgraph is asynchronous BFT, is included in the Appendix.

ACID COMPLIANCE

The hashgraph is **ACID compliant**. ACID (Atomicity, Consistency, Isolation, Durability) is a database term, and applies to the hashgraph when it is used as a distributed database. A community of nodes uses it to reach a consensus on the order in which transactions occurred. After reaching consensus, each node feeds those transactions to that node's local copy of the database, sending in each one in the consensus order. If the local database has all the standard properties of a database (ACID), then the community as a whole can be said to have a single, distributed database with those same properties. In blockchain, there is never a moment when you know that consensus has been reached, so it would not be ACID compliant.

DISTRIBUTED DENIAL OF SERVICE ATTACK RESILIENCE

One form of Denial of Service (DoS) attack occurs when an attacker is able to flood an honest node on a network with meaningless messages, preventing that node from performing other (valid) duties and roles. A Distributed Denial of Service (DDoS) uses public services or devices to unwittingly amplify that DoS attack—making them an even greater threat.

In a DLT network, a DDoS attack could target the nodes that contribute to the definition of consensus and, potentially, prevent that consensus from being established.

The hashgraph is DDoS resilient as it empowers no single node or small number of nodes with special rights or responsibilities in establishing consensus. Both Bitcoin and hashgraph are distributed in a way that resists DDoS attacks. An attacker might flood one member or miner with packets, to temporarily disconnect them from the internet. But the community as a whole will continue to operate normally. An attack on the system as a whole would require flooding a large fraction of the members with packets, which is more difficult. There have been a number of proposed alternatives to blockchain based on "leaders" or "round robin" models. These have been proposed to avoid the proof-of-work costs of Bitcoin. But they have the drawback of being sensitive to DDoS attacks. If the attacker attacks the current leader, and switches to attacking the new leader as soon as one is chosen, then the attacker can freeze the entire system while still attacking only one computer at a time. Hashgraph avoids this problem, while still not needing proof-of-work.

Fairness

Hashgraph is fair because there is no leader or miner given special permissions for determining the consensus timestamp assigned to a transaction. Instead, the consensus timestamp for transactions are calculated via a voting process in which the nodes collectively and democratically establish the consensus. We can distinguish between three aspects of fairness.

FAIR ACCESS

Hashgraph is fundamentally fair because no individual can stop a transaction from entering the system, or even delay it very much. If one (or few) malicious node attempts to prevent a given transaction from being delivered to the rest of the community and so be added into consensus, then the random nature of the hashgraph gossip protocol through which nodes communicate messages to each other will ensure that the transaction flows around that blockage.

FAIR TIMESTAMPS

Hashgraph gives each transaction a consensus timestamp that is based on when the majority of the network members received that transaction. This consensus timestamp is fair, because it is not possible for a malicious node to corrupt it and make it differ by very much from that time.
Every transaction is assigned a consensus time, which is the median of the times at which each member says it first received it. Received here refers to the time that a given node was first passed the transaction from another node through gossip. This is part of the consensus, and so has all the guarantees of being Byzantine. If more than two-thirds of participating members are honest and have reliable clocks on their computer, then the timestamp itself will be honest and reliable, because it is generated by an honest and reliable member or falls between two times that were generated by honest and reliable members. Because hashgraph takes the median of all these times, the consensus timestamp is robust. Even if a few of the clocks are a bit off, or even if a few of the nodes maliciously give times that are far off, the consensus timestamp is not significantly impacted.
This consensus timestamping is useful for things such as a legal obligation to perform some action by a particular time. There will be a consensus on whether an event happened by a deadline, and the timestamp is resistant to manipulation by an attacker. In blockchain, each block contains a timestamp, but it reflects only a single clock: the one on the computer of the miner who mined that block.

FAIR TRANSACTION ORDER

Transactions are put into order according to their timestamps. Because the timestamps assigned to individual transactions are fair, so is the resulting order. This is critically important for some use cases. For example, imagine a stock market, where Alice and Bob both try to buy the last available share of a stock at the same moment for the same price. In blockchain, a miner might put both of those transactions in a single block, and have complete freedom to choose what order they occur. Or the miner might choose to only include Alice's transaction, and delay Bob's to a future block. In hashgraph, there is no way for an individual to unduly affect the consensus order of those transactions. The best Alice can do is to invest in a better internet connection so that her transaction reaches everyone before Bob's. That's the fair way to compete.

Governance

A governance model for a public ledger will define the rules and policies that control the evolution of the node software, issuance of coins, and the reward model that incentivizes network participants. The stakeholders whose interests and motivations must be balanced are those running the network nodes, those building applications on the platform, those businesses relying on those applications, the end-users of those applications, and relevant regulatory bodies.

The Hedera Hashgraph Council is a limited liability company that will have up to 39 members, who will be well-known enterprises from diverse industries and geographies. Hedera's licensing and governance model protects network users by eliminating the risk of forks, guaranteeing the integrity of the codebase, and providing open access to review the underlying software code. Under the governance model, all Governing Members will have equal voting rights and each Governing Member (with the exception of Swirlds) will serve a limited term, ensuring that no single Governing Member or group of Governing Members has centralized control.

Hedera has a Permissioned Governance Model with Permissionless or Open Consensus.

The Council establishes policy for council membership, regulates the network rules and coins, and approves changes to the platform codebase. Our governance model is based on the original model used by National BankAmericard Inc., founded in 1968, which was later renamed VISA. We are designing our governance model in a way that ensures the Governing Members can be trusted to do what's in the best interest of the Hedera platform, and not be unduly influenced by individual Council members or node operators. In addition to Governing Members, Hedera will have a set of Advisory Members that contribute by providing advisory services as appropriate, but do not have voting privileges.

The Open Consensus model relates to the process by which the nodes join the network and reach a consensus on the order of transactions in the platform. The model is designed to prevent consolidation of power over consensus by encouraging the emergence of a decentralized network with, eventually, millions of nodes. It prevents collusion by a few to attack the system such as by counterfeiting the cryptocurrency, modifying the ledger inappropriately, or influencing the consensus order of transactions. We inhibit collusion by weighting the votes within the hashgraph algorithm of a particular node based on the node's stake. Loosely stated, each node casts one vote for each coin of the Hedera native currency (hbars) it owns. New node operators will join the network and be paid for their services in maintaining the hashgraph. The number of nodes is expected to grow large very quickly, ensuring consensus voting privileges are distributed to many nodes. A full discussion of the staking model is included in the section below.

This system of Permissioned Governance with Open Consensus will build more public trust than a purely closed system. This is essential to the success of a global cryptocurrency.

PERMISSIONED GOVERNANCE

We designed the Hedera governance model to ensure that the organization can be trusted. Council members will have equal governing rights and limited terms, ensuring that governance is decentralized. Deliberation and debate will be open to all and controlled by none.

The Governing Members will also elect or appoint members to committees that provide oversight of Hedera operations. Committees will include but are not limited to a Technical Steering Committee, a Finance Committee, and a Legal & Regulatory Committee. The Governing Members are organizations that span a broad range of business sectors, and our objective is that, collectively, the membership will contribute industry-leading representation to the range of Hedera committees.

Stability

The hard forks that Bitcoin and Ethereum have experienced have arguably damaged the network effect of their corresponding currencies, creating confusion and uncertainty in the marketplace. Similarly, the explosion of altcoins (and the dubious legitimacy and value of many of them) does not engender the necessary confidence in businesses and consumers considering adopting cryptocurrencies.

Historically, open source software developers have recognized the value of maintaining a single baseline, and ensuring that the best ideas from the community are included for the benefit of the whole. However, when combining an open source project with a cryptocurrency, the traditional incentive structure is turned upside down. The distributed ledger technologies that have been most widely adopted are also those that have split the most. This dynamic causes chaos in the industry, and directly impedes the adoption of public ledgers by mainstream markets.

Hedera technical and legal controls ensure the platform will not fork into a competing platform and cryptocurrency.

Technical controls

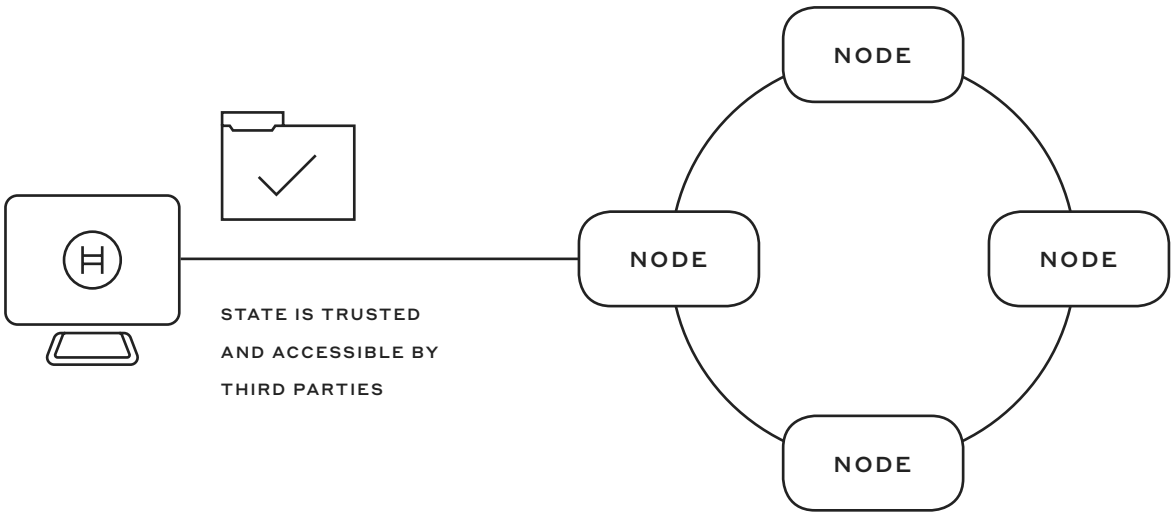
SIGNED STATE PROOFS

There is a shared state that is maintained by every node in the ledger (or in the shard, when the ledger is sharded). At the end of each round, each node calculates the shared state after processing all transactions that were received in that round and before. It then digitally signs a hash of that shared state, puts it in a transaction, and gossips it out to the community. Then it collects those signatures from all other nodes.

In this way, a node can have a copy of the state with a set of signatures that proves to a third party that this is the true, consensus state. This allows the node to construct a small file which is a verifiable proof that the state was truly the consensus.

The state is organized as a Merkle tree, so a third party can be given a proof that consists of a small part of the state, plus the path from there to the root of the Merkle tree (including siblings of those vertices in the tree), plus the signatures, and an address book history for the public keys.

The diagram below represents how a third party can be confident that the state it receives from one of the nodes does indeed represent the consensus state of the full network.



LEDGER ID

The proof must also include an "address book", which is list of the public keys of all the members, along with each member's stake (owned directly or by proxy). A third party will need this address book in order to check the signatures on the state (or portion of state).

The proof must also include an "address book history". This is a sequence of address books, where each address book is signed by members from the previous address book. Any given address book must be signed by a set of members that own more than 2/3 of the stake, according to the membership and stake from the previous address book. This chain of address books extends back to the genesis address book, which is the initial members who created the ledger at the beginning.

The hash of the genesis address book is important. It serves as a unique identifier of the ledger. It is the "name" of the ledger.

HANDLING FORKS

If a small number of members want to split off from the group and create a new ledger that is a fork of the current one, they have the technical ability to do so, and can even create the initial state of their new ledger to be identical to the old ledger. So it is a fork. However, they will not be able to create an address book history reaching back to the genesis address book, with the members of each address book signing the next one, because the majority of members (who are not forking) will not sign the address book for the minority of members who are forking. This forces the new fork to have a new genesis address book, and therefore a new unique identifier, and therefore a new name. Consequently, those creating the fork will be unable to fool anybody into thinking the fork is the legitimate ledger.

When a client submits a transaction to a node to send to the ledger, the client receives in response from the node the cryptographic proof that their transaction has affected the shared state correctly. When Alice transfers cryptocurrency to Bob, both of them can receive a cryptographic proof that the transaction succeeded. This proof includes the signatures reaching back to the genesis address book. So they not only verify that the transfer occurred, they verify that it occurred on the correct ledger. If a ledger forks, no client will ever be confused about which ledger they are dealing with because only one ledger at a time can have that name.

Furthermore, if a 50/50 split were to happen, then neither side would be able to prove a connection to the genesis address book. It wouldn't be a fork; it would be the complete destruction of one ledger, and the creation of two unrelated ledgers. This would greatly reduce the value to the nodes, because they would no longer be able to earn fees from the clients who want to access the original ledger. And all of the original cryptocurrency would, in a very real sense, cease to exist. This creates an enormous disincentive to forking.

In this way, confusing forks simply become impossible. Non-confusing forks become unappealing for the nodes. So there are strong incentives to avoid forks, even aside from any legal incentives.

The cryptographic proofs and unique identifiers are also critically important for secure sharding. They allow shards to send each other messages, with assurance that the message from a given shard was truly the consensus of that shard.

Legal Controls and Transparency

The Hedera codebase will be governed by the Hedera Hashgraph Council, and will be released for public review with Version 1.0. The codebase will be open review, meaning that anyone will be able to read the source code, recompile it, and verify that it is correct.

No license will be required to use the Hedera platform. No license will be required to write software that uses the services of the Hedera platform. No license will be required to build smart contracts on top of the Hedera platform. Applications built upon the Hedera platform can be open source or proprietary. They do not require any license or any approval from Hedera. Software developed using the platform APIs will not be encumbered in any way. Software developers will have complete ownership and discretion on the licensing they choose for their applications that use the Hedera platform.

Swirlds owns the intellectual property rights in the hashgraph consensus algorithm. Hedera Hashgraph Council has a license from Swirlds to use the hashgraph consensus algorithm and associated technology for the Hedera distributed public ledger platform. As part of that license, Hedera Hashgraph Council will pay Swirlds 10% of revenue (with monthly minimums) and Swirlds will own 5% of Hedera coins. Swirlds will continue to require licenses for use of the hashgraph technology in permissioned networks, but no license will be required for distributed applications that run on Hedera's public platform. Hedera and Swirlds will use the patent rights associated with the hashgraph algorithm defensively to legally prohibit the forking of the codebase and the creation of a competing platform and currency. Developers are free to build distributed applications on top of the Hedera platform with associated native tokens.

In summary, Hedera will simultaneously embrace open review, while bringing stability to the platform and cryptocurrency by controlling the license. In this way, Hedera will provide a transparent codebase that will provide the stability that markets demand for mainstream adoption.

Regulatory Compliance

We expect that governments will soon require comparable visibility and oversight into public ledger financial transactions that they have now for traditional banking. In our view, for a public ledger to be adopted widely, it must be capable of enabling appropriate Know Your Customer (KYC) and Anti Money Laundering (AML) checks.

Hedera will enable KYC and AML through an Opt-In Escrowed Identity system.

The Escrowed Identity system allows a user's real identity (as asserted by an accredited party acting as a Certificate Authority) to be logically bound to their ledger account so that, when using that account to move funds, appropriate KYC checks can be performed and AML protections enforced. The identity is escrowed because only on defined criteria or schedule need the government be informed.

The system is opt-in, in that a user must explicitly choose to avail themselves of the mechanism and, if they do not, their account transactions will remain anonymous. This choice however may prevent them from engaging in certain financial transactions.

The system is designed to provide the appropriate balance of

1. **Government visibility**
2. **Security**
3. **User privacy**

Comparable to showing a driver's license when creating a new bank account, the model has a user attach a hash of a digital certificate created by a recognized identity provider to their account. This attachment will take the form of a transaction sent out to the network.

This transaction:

1 . May need to be signed by both the user's private key and that of the identity provider.
2. Can stipulate what parties are authorized to subsequently detach the hash (and so revoke the binding between the identity and the account).

As long as the attachment between account and certificate has not been revoked, by either the user or the identity provider, it can be used to establish that the account is bound to a known user whenever funds move in or out of that account. If and when appropriate, the identity provider can revoke the binding simply by sending a signed transaction to the network.

As an example of how it might work, consider a user trying to send money from their Hedera account to a US bank. The user would provide to the bank the certificate as well as their account address. The bank would look up the account and confirm that the account had the corresponding hash for the certificate, and that the certificate was issued by a trusted identity provider. Only if all these checks were confirmed would the bank authorize the transaction and accept the funds.The bank might be asked by the corresponding government to send the certificate and transaction details, either in real-time (perhaps based on the amount of the transfer) or on a schedule.

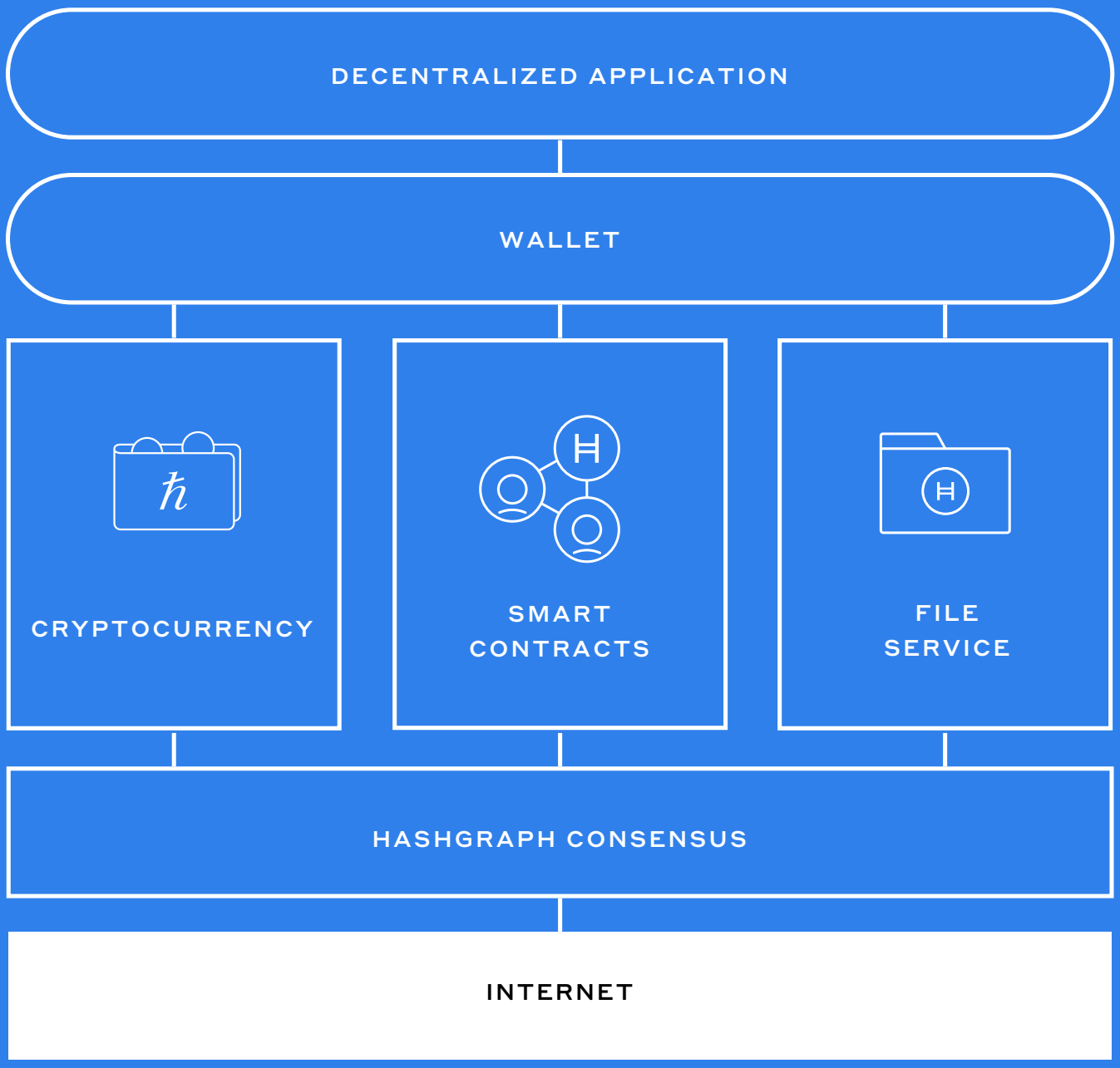
Critically for privacy, the user can revoke the binding at any time as well - removing the binding between their identity and the account. Doing so might prevent them from using that account in certain situations but that would be their choice.

Hedera is a founding member of the Distributed Ledger Foundation and will work with the broader DLT community and governments there to ensure that regulatory requirements can be satisfied, while maintaining privacy and security.

Conclusion

Hedera directly resolves the five fundamental obstacles to mainstream market adoption of public ledger technology: Performance, Security, Stability, Governance, and Regulatory Compliance. The hashgraph data structure and consensus algorithm provides a best-in-class, unmatched combination of performance and security. The Hedera platform and governance council will provide transparency, open innovation with platform stability, tools to enable opt-in KYC and AML, and global, cross-industry expertise to provide governance and decision making for a globally distributed network and cryptocurrency.

Part 2
Architecture


DECENTRALIZED APPLICATION
WALLET
CRYPTOCURRENCY
SMART CONTRACTS
FILE SERVICE
HASHGRAPH CONSENSUS
INTERNET

INTERNET LAYER

The nodes are all computers on the internet, communicating by TCP/IP connections protected by TLS encryption with ephemeral keys for perfect forward secrecy. Nodes are addressed by IP address and port, rather than by symbolic names, so attacks on the DNS system will not affect the network.

HASHGRAPH CONSENSUS LAYER

The nodes take transactions from clients and share them throughout the network with a gossip protocol. Then all nodes run the hashgraph consensus algorithm to reach agreement on a consensus timestamp for each transaction and its consensus order in history. Each node then applies the effects of the transactions in consensus order to modify its copy of the shared state. In this way, all nodes maintain an identical consensus state (within any given shard).

SERVICES LAYER

CRYPTOCURRENCY

The cryptocurrency is designed to be fast, which leads to low network fees, making very small microtransactions practical. When the Hedera platform is running at scale, any user will be able to run a node in the network and earn cryptocurrency payments for doing so. Any user can create an account by simply creating a key pair, without any name or address attached to it. Optionally, provisions are made to allow a user to attach hashes of identity certificates. These can come from any third party certificate authority or identity authority that the user chooses. This is intended to allow regulatory compliance, for cryptocurrency accounts that will be used in a jurisdiction with Know Your Customer (KYC) or Anti-Money Laundering (AML) laws. More detail is given in the Regulatory Compliance section.

FILE SERVICE

The file system allows users to store information, with consensus on exactly what is stored and what is not stored. Every node in the shard stores the same files, so they will not be lost if one of the nodes crashes. Stored information can only be deleted by those that were given permission. In this way, the file system can act as a revocation service. For example, in the future, a user might be issued a driver's license from the Department of Motor Vehicles (DMV), and both the user and the DMV digitally sign the transaction that puts a hash of it into the ledger. Both have the right to remove the hash of the license. The user can choose to prove to someone that they have a valid license, by giving that person a copy of the license file, so the person can check whether the hash is still stored in the ledger. If the DMV revokes the license, it would also delete the hash, to show the world that the license is no longer valid. If the user tries to store the hash again, without a signature from the DMV, it will be evident that the hash was stored only by the user without DMV cooperation, and would not be considered valid evidence of the user's right to drive.

Files are actually stored as Merkle trees, but we provide Java classes to allow developers to manipulate them.

We give developers Java code to manipulate a Merkle tree as if it were a file system. They see directories, subdirectories, files, and they can change the contents of files, and names of directories, and move things around, and copy and paste, and yet, underneath, it's all being stored as a Merkle tree automatically. This allows us to give proofs that a file is part of the consensus state. Also, users can store an entire directory in the Hedera file system.

We not only store Merkle trees, we store Merkle DAGs, which means that if two files have some bytes in common, we might only store one copy of the common bytes.

A file can be accessed by its hash, so people can rely on the fact that it is immutable. But it also has a File ID. Its owner can create a new file, and make the File ID to be associated with the new file instead of the old one. In this way, it is possible for users to always find the latest version of a file. They just access the File ID instead of the hash. So files are both securely immutable and securely non-immutable, at the same time. If a file is accessed by its hash, then it never changes. If it is accessed by its File ID, then the latest version is found.

SMART CONTRACTS

The Hedera ledger can run smart contracts written in Solidity. There currently exist large libraries of Solidity smart contract code, which can be run unchanged on Hedera. These allow for distributed applications to be easily built on top of Hedera.

Sharding

Initially, the Hedera network will likely be a small number of nodes all in a single shard. As Hedera grows, it will gain a sufficient number of nodes to justify multiple shards. Sharding can offer performance advantages as every node need not process every transaction. Consensus can consequently proceed in parallel. Shards trust each other, so one shard will honor requests to move cryptocurrency or to put a hold on various resources made by another shard - as long as those requests can be proven to reflect the consensus of the requesting shard. This allows the multi-shard ledger as a whole to achieve asynchronous Byzantine fault tolerance, and to prevent double spends or other illegal states, because each individual shard has those properties, and because all messages between them contain proofs that they are the consensus of that shard.

Nodes will be randomly grouped into different shards, within which consensus on transactions will be established as normal. Each shard is made up of a subset of the nodes, all of which share the same state, which is a subset of the state of the entire ledger. Transactions are placed into consensus order within individual shards in the normal manner - all nodes within a shard contribute only to the consensus for transactions that originate in that shard. The assignment of nodes to shards is performed randomly by a master shard, which assigns new nodes to a shard once a day, and also moves nodes between shards as necessary to ensure that each shard has a large total amount being staked, and that no one member of a shard has a large fraction of that amount.

Shards communicate through the exchange of messages between members of the different shards. All such messages are push (rather than pull). Each shard (its members) maintains a queue of outgoing messages to each of the other shards. Each shard remembers the sequence number of the last message it processed from each of the other shards. A message is sent from shard Alpha to shard Beta by nodes in Alpha randomly contacting nodes in Beta, to transfer the message, along with a proof that it is part of the consensus state of the Alpha shard. They continue doing this until one of the Beta members replies with a proof that the Beta shard shared state includes a sequence number indicating that this message was received and processed. In this manner, transactions that impact addresses in different shards can be appropriately recorded into each shards state, and so the entire state of the entire ledger.

More details are given in the Sharding appendix.

Part 3
Cryptoeconomics

Staking and proxy staking

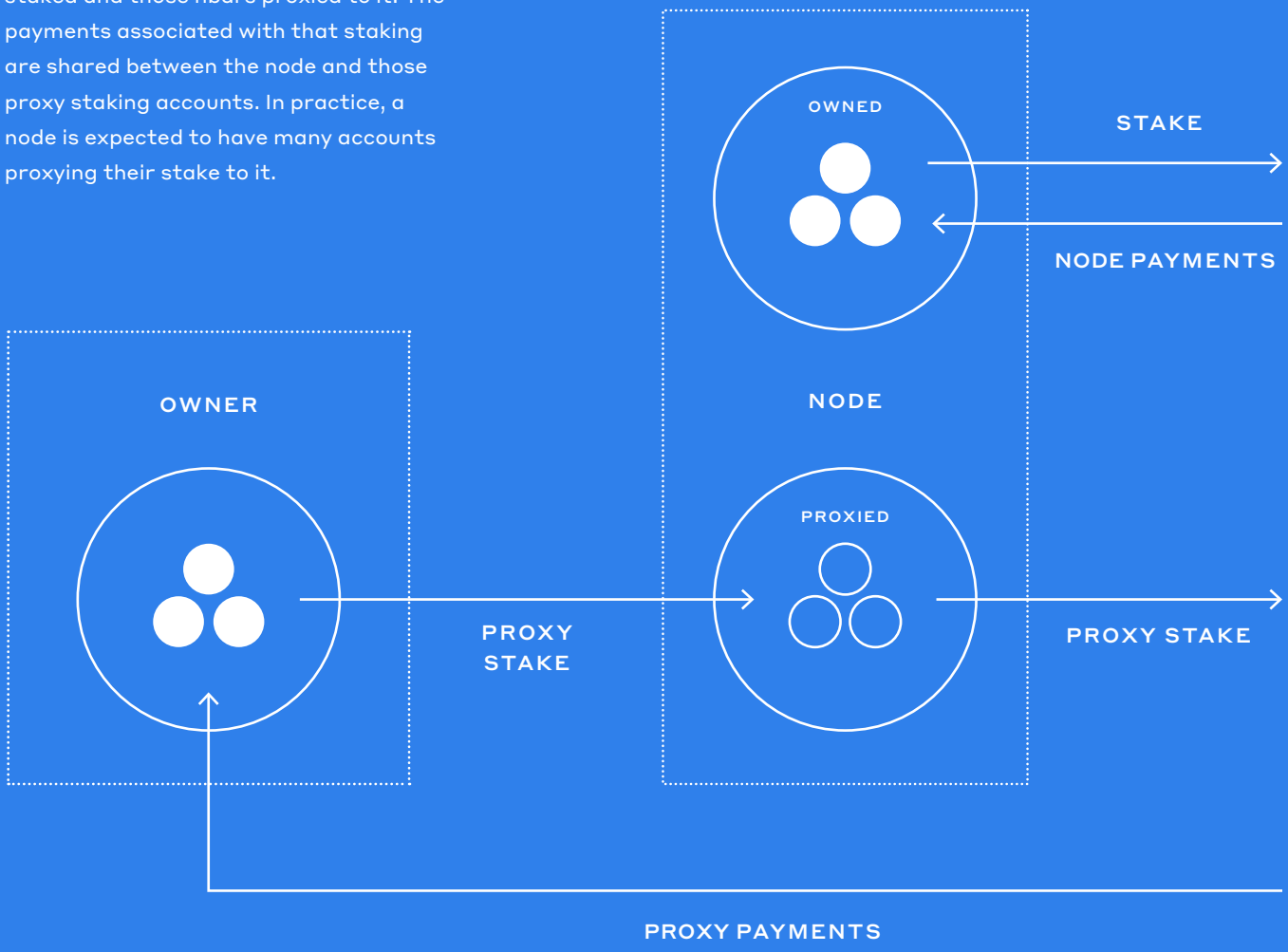
To achieve transparency, and the performance advantages of sharding, it is necessary to allow anonymous individuals to become nodes in the network. Then, to avoid Sybil attacks, we implement a system where each node has an influence on the consensus that is proportional to the amount of cryptocurrency that it owns. Then, it becomes important to ensure that most of the cryptocurrency is actually being staked, so that the network continues to run.

The Hedera ledger will use proof-of-stake. When a node joins the system, it must declare one or more accounts that it can control, and prove that it has the private keys for those accounts. From then on, the amount of hbars in those accounts will be used to weight its votes in the hashgraph virtual voting algorithm. Additionally, it will be paid to serve as a node, with that payment proportional to the amount of hbars in those accounts - the stake effectively earning interest. It is still free to spend those hbars at any time. Consequently, a potential disincentive of bonded proof of stake models - that of nodes unwilling to stake for fear of the associated loss of liquidity - is avoided.

In addition, a mechanism called proxy staking allows a person who owns hbars but does not run a node to nevertheless stake those hbars and so earn interest by “proxy staking” their account to a node. That means giving another account credit for their hbars, and allowing the node to use that stake. The payments for running the node (proportional to the amount staked) are then split between the node and the owner of the hbars being proxy staked. The hbars that are being proxy staked still remain under the control of their owner. The owner can turn off or redirect the proxy staking to another node at any time. They can also spend the hbars at any time, though again that will reduce the amount they receive in payment for staking. Note that the node to which the hbars are proxy staked can not spend those hbars.

A node must have at least some hbars in its account for it to be able to influence consensus or to receive payment for operating as a node, or to pay fees associated with sending transactions to the ledger.

The proxy staking model is shown below. A node's stake towards consensus reflects both the hbars it owns and has staked and those hbars proxied to it. The payments associated with that staking are shared between the node and those proxy staking accounts. In practice, a node is expected to have many accounts proxying their stake to it.



Proxy staking allows those who do not run nodes to still be able to earn some interest on their hbars and by encouraging this practice raises the bar for some actor being able to gain influence over a third of the entire stake. And those who do run nodes will be able to increase their revenue. The barrier to entry is very low for running a node and attracting proxy stakers. So it is expected that there will be a large ecosystem of nodes that are competing for proxy stakers. Consequently, it will be difficult for one attacker to gain proxy staking control of a third of the entire stake.

Payments and fees

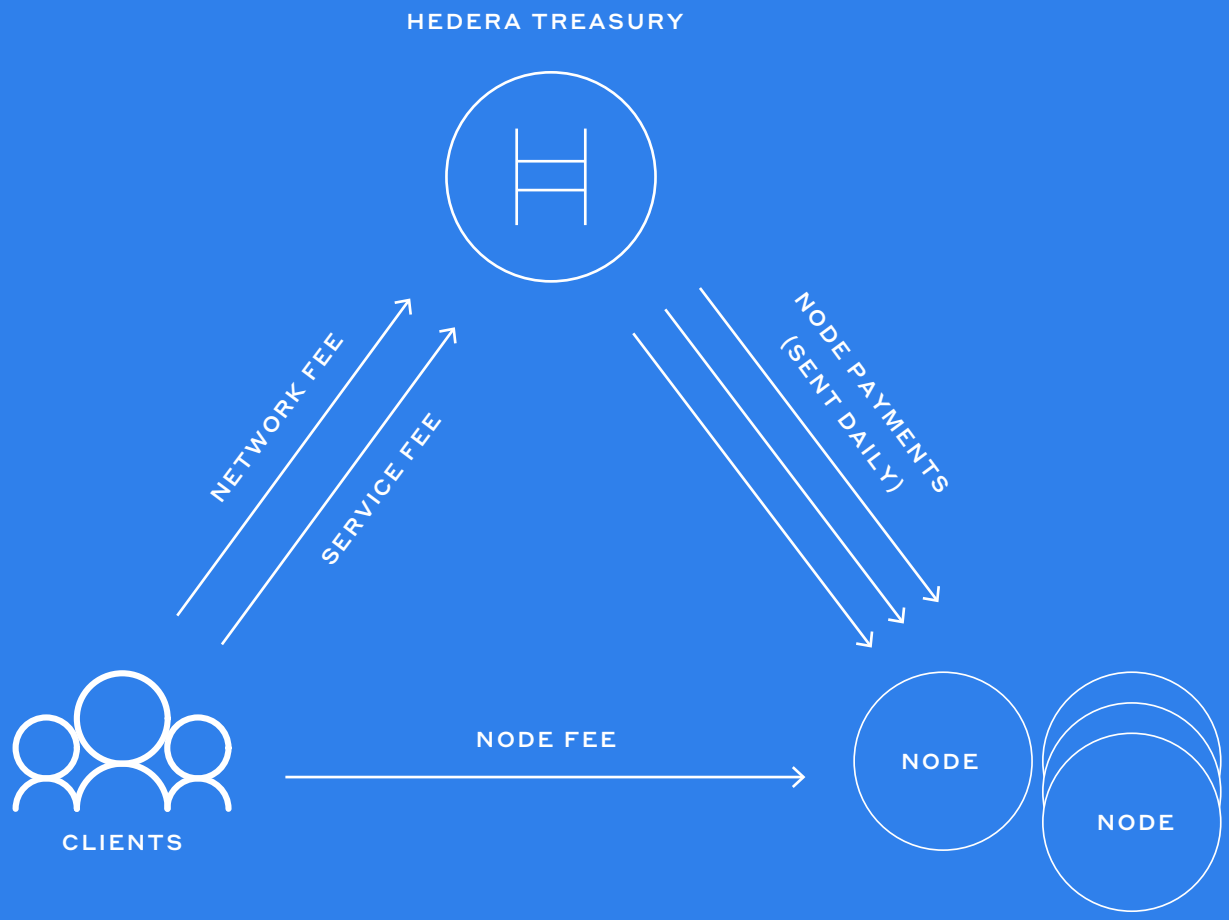
Users pay fees to use the platform, such as when they transfer cryptocurrency coins, or add items to the ledger. Because the Hedera network has high throughput and doesn't require proof-of-work, we anticipate the fees to be a small fraction of other public platforms in the market today.
Nodes in the Hedera ledger are compensated for the computing, bandwidth, and storage resources they use in establishing consensus and providing services. There are several types of payments and fees:

NODE FEE – A client can use the services of the platform by contacting a node, which will submit transactions on the client's behalf. For example, if a client wants to transfer cryptocurrency from their account to another, they will contact a node, and give it the signed transaction. The node will then put that transaction into the next event it creates, and gossip it out to the network so that it can be entered into consensus. The client reimburses the node for this effort by giving it a node fee. This fee is negotiated between client and node, and can be set by market forces as nodes set their fees. This is the only fee that is not set by Hedera.

SERVICE FEE – A client will pay a fee for any Hedera service. For example, if a client submits a transaction to store a file in the ledger, the fee will be calculated according to a schedule determined by Hedera. This is calculated as a fee per file plus an amount per byte per second that the file will be stored. A single transaction both requests the service and authorizes paying for it. If the client's account has insufficient funds at the point the transaction takes effect in the consensus order, then the client is not charged, and the file is not stored. But if there are sufficient funds, then simultaneously the client is charged and the file is stored.

NETWORK FEE – There is a fee for each transaction handled by the network, to cover the cost to nodes of gossiping it, temporarily storing it in memory, and calculating the consensus on the event containing it. The fee is calculated as an amount per transaction plus an amount per byte within the transaction. When a node includes a transaction in an event that it creates the node will be charged the network fee when consensus is reached on that transaction. If the transaction was initiated by a client, that client will compensate the node for that network fee the node paid.

The three different fee types are shown in the diagram below, indicating from which account they are taken, and to which they are added. Clients pay nodes fees directly to the node that they requested to process their transaction. Clients pay network fees to the same node, but those fees will be passed onto Hedera. Clients pay service fees direct to Hedera. All fee payments are made through the network coming to consensus on a gossiped transaction that authorized the payment.



Hedera collects services and transaction fees on behalf of all the nodes processing the transactions and performing the services. Hedera uses those collected fees to fund incentive payments to nodes:

INCENTIVE PAYMENT – Once a day, payments are made from the Hedera account to nodes, to incentivize them to serve as nodes. To be paid, a node must have been online for the full day, according to thresholds defined by Hedera (e.g., requiring that the node contribute at least one event each to at least 90% of the rounds during that 24 hour period). A node is paid proportional to the amount of cryptocurrency it is staking (both owned by itself, and proxy staked to it by others).

The fee model is designed to allocate costs and risks appropriately.

The biggest resource costs are paid for by service fees, and those resource costs (e.g., storing a large file) are never incurred until proper payment has been made by the client.

The smaller resource costs of gossiping and reaching consensus on the transactions themselves are paid for by the network fees. The ledger as a whole is sure to get the fees, because they are paid by the node, using the cryptocurrency being staked. And the node can do this without risk, by requiring that the client's payment for it be completely processed before the node submits it. (Or, the node can choose to submit it immediately, with a small risk of not being paid).

The smallest resource cost is the cost for the node to submit the tiny transaction that does nothing but transfer a fee from the client to the node. The risk of an attack seems low, because it requires careful timing on the part of the attacker, and will require the attacker to actually pay fees which might exceed the amount the attacker is tricking the node into spending. Furthermore, nodes can always use other mechanism, such as charging reduced fees to repeat customers, to further reduce the probability of this attack being frequent.

So at each level of the system, costs are paid for, and economic incentives are aligned.

When a client contacts a node for help submitting a transaction to the network to perform some service for the client, the client gives the node two transactions:

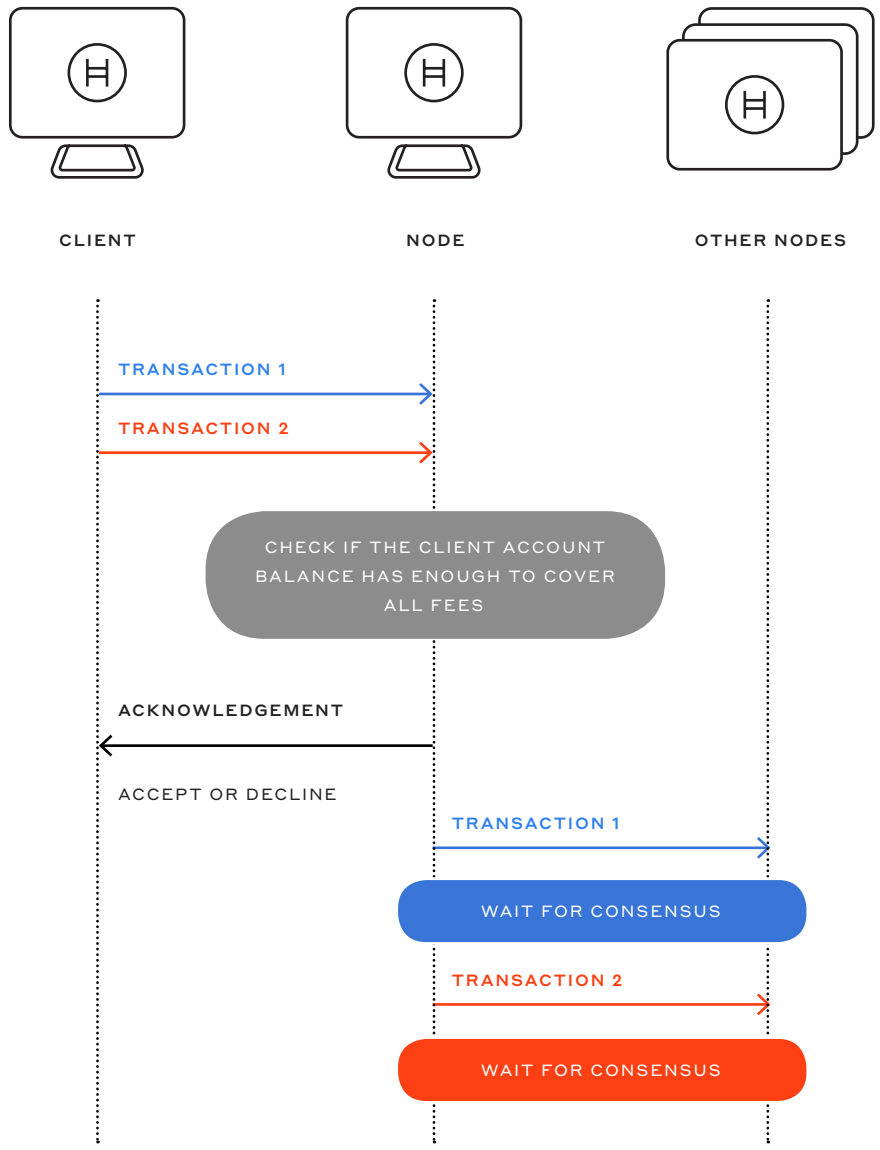
1. A service transaction that includes both the requested service and an authorization to pay the associated service fee.

2. A payment transaction that will pay the node an amount equal to the the sum of:

- **a.** The node fee
- **b.** A network fee for the service transaction and
- **c.** A network fee for the payment transaction.

The node first checks that the client's account has sufficient funds to pay the payment transaction. If so, it submits the payment transaction to the network. After that payment transaction has reached consensus, if it is valid, only then will the node submit the service transaction to the network.

This two phase processing is shown below. The blue arrow represents the payment transaction, the green arrow the (fundamental) service transaction.



Consider how the above model minimizes risks for all participants.

By sending out the payment transaction first, the node minimizes its risk of non-payment from the client. The node risks only the chance of paying the network fee for this single, very small transaction. Even in the case that the client's account is depleted during the short period between when the node first checks the client's account balance until the node submits the transaction, the node loses only that.

The client risks that it pays the two network fees and node fees without the node actually submitting the service transaction to the network. But the client does not risk the service fee, since that is only paid if the node submits the service transaction to the network. The service fee may be the largest of all these, so the client is likely only risking a very small amount. The client can also include in the second transaction a second node fee, to incentivize the node to submit the service transaction. This reduces the risk for the client.

Because the service is only performed if the payment occurs, there should not be a risk to Hedera.

An optimization is possible. If the actual service is a transaction with a small number of bytes, then the two transactions can be merged into a single transaction. For example, if the client wants to transfer cryptocurrency to another account, the client might create a single transaction that both performs that transfer, and also pays the node the sum of the network fee and node fee. Then, the client has little risk, because the node is paid for submitting the transaction. If the node fails to submit it, then the node is not paid. The fees incentivize the node to submit it correctly.

The Roadmap to Scale

The following are the steps by which we expect the network will grow from concentrated nodes and stake to widespread nodes and stake. These steps will not be distinct. They simply represent a smooth growth evolution of the network and currency.

STEP 1 The central Hedera treasury holds most of the coins, and proxy stakes these to the Hedera Governing Members. The members operate nodes with the treasury coins proxied to them. Coins held by individuals that use the software wallet from Hedera also proxy their coins to the member nodes. During this step, some coins from the Hedera treasury are distributed to the general population.

STEP 2 Advisory Members and other trusted parties are able to stand up nodes with the coins being proxied to them in addition to the Governing Members. The treasury and the Hedera wallet (by default) proxies to both the Governing Members and the Advisory Members. Over time the distribution of the staking becomes more even across all nodes. More coins continue to be distributed from the Hedera treasury to the general population.

STEP 3 Individuals who are interested may go through a Know-Your-Customer process and then also receive staking of coins from the Hedera treasury and the default Hedera wallet software. More coins continue to be distributed out of Hedera to the general population. Anonymous individuals can also run nodes and receive staking of coins. The Hedera wallet software will not proxy to them, but they may be able to receive proxying from 3rd-party wallet software.

STEP 4 As the coins are distributed widely, and competing wallet software programs arise, there will be a market for proxying that is independent of Hedera. Eventually all of the coins are widely distributed, there is a market of wallet software, and a market of nodes competing for the proxy staking.

In this way, all of the coins start in one account, and the initial Hedera wallet software defaults to proxying just to Governing Members, but over time both the coins and the proxying gain wider and wider distribution until they are distributed across millions of nodes and accounts.

Acknowledgements

We gratefully acknowledge the contributions and help from our advisors, Natalie Furman, Tom Trowbridge, Edgar Seah, Jordan Fried, Christian Hasker, Arlan Harris, Paul Bugeja, Alex Godwin, Ken Anderson, Patrick Harding, Zenobia Godschalk, and George Samman.

Appendices

Appendix 1: Team

DR. LEEMON BAIRD, CO-FOUNDER & CHIEF SCIENTIST

Leemon is the inventor of the hashgraph distributed consensus algorithm, and is the Co-Founder and Chief Scientist of Hedera. With over 20 years of technology and startup experience, he has held positions as a Professor of Computer Science at the US Air Force Academy and as a senior scientist in several labs. He has been the Co-Founder of several startups, including two identity-related startups, both of which were acquired. Leemon received his PhD in Computer Science from Carnegie Mellon University and has multiple patents and publications in peer-reviewed journals and conferences in computer security, machine learning, and mathematics.

MANCE HARMON, CO-FOUNDER & CEO

Mance is an experienced technology executive and entrepreneur with more than 20 years of strategic leadership experience in multi-national corporations, government agencies, and high-tech startups, and is Co-Founder and CEO of Hedera. His prior experience includes serving as the Head of Architecture and Labs at Ping Identity, Founder and CEO of two tech startups, the senior executive for product security at a $1.7B revenue organization, Program Manager for a very-large scale software program for the Missile Defense Agency, the Course Director for Cybersecurity at the US Air Force Academy, and research scientist in Machine Learning at Wright Laboratory. Mance received a MS in Computer Science from the University of Massachusetts and a BS in Computer Science from Mississippi State University.

TOM TROWBRIDGE, PRESIDENT

Tom was part of the founding team and has broad responsibility across the business, and is the President of Hedera. He has been advising and investing in technology companies since 1996 when started his career as an investment banker in the telecom group of Bear, Stearns & Co. He began investing in early stage companies in 1998 when he joined the private equity firm Alta Communications and over three and a half years executed ten deals in technology, telecom, and media and served on multiple boards. After Alta Communications, he spent almost four years at Goldman Sachs before holding various positions at Lombard Odier and Atticus Capital. Before joining Hedera, he started and ran the New York office for UK-based Odey Asset Management. Tom has a BA from Yale University and an MBA from Columbia University with honors (Beta Gamma Sigma).

JORDAN FRIED, VP OF BUSINESS DEVELOPMENT

Jordan is a DLT evangelist and self-professed crypto-capitalist, and is the Vice-President of Global Business Development of Hedera. He was previously the Co-Founder and CEO of Buffered VPN, the fastest growing personal VPN service online, which was acquired in Q1 2017. Jordan is an investor in companies such as Hive.org and Buffer App and has been featured in Entrepreneur Magazine, Inc.co, Wired.com, Time Magazine, and Success.com.

NATALIE FURMAN, GENERAL COUNSEL

Natalie is the General Counsel for Hedera. She was previously a senior associate at Paul Hastings LLP, where her practice focused on intellectual property, unfair competition, and rights of privacy and publicity. Prior to law school, she started her career in Silicon Valley, providing strategic advice to high tech startups. She was Director of Strategy and Business Development at an online group communication startup acquired by Yahoo! Inc. and Director of Business Development for a technology startup developing a global collaborative supply chain platform. Natalie received her BA in Anthropology with honors from Stanford University and her JD from Columbia University School of Law.

Appendix 2: Sharding

Initially, the network may consist of a smaller number of nodes in a single shard. As the network grows, it will gain sufficient nodes to support multiple shards. Those shards will work in the following way.

A transaction is always submitted to a specific shard. Within a shard, every node receives all of that shard's transactions, and every node maintains an identical shared state. Each shard can store both cryptocurrency accounts and files. Every shard can run smart contracts.

A shard uses the hashgraph consensus algorithm to reach a consensus order for its transactions. Each shard must be able to trust the consensus decision of each of the other shards. Therefore, each shard must be composed of randomly-chosen members, and must be large enough so that it can be trusted to never have 1/3 of its staked cryptocurrency being owned by malicious nodes.

If a transaction involves only resources within a given shard, then when that point in the consensus order is reached, the transaction performs its effect. For example, a transaction might move cryptocurrency between two accounts within the same shard. Or it might save a file within that shard, and pay for it with an account in that shard. In those cases, the cryptocurrency transfers or the file is stored immediately, at the point where the transaction occurs in the consensus order.

If a transaction involves resources in different shards, then it will trigger inter-shard messages. For example, if the cryptocurrency account Alice is in shard Alpha, and account Bob is in shard Beta, then Alice creates and signs a transaction to move cryptocurrency from Alice to Bob. She submits that transaction to a node in the Alpha shard, and all of the nodes in Alpha reach consensus on its order. At the point where this event occurs in the consensus order, Alice's account balance is decreased by the amount being sent, and a message to the Beta shard is generated. Each shard maintains a queue of outgoing messages to be sent to each of the other shards. So this new message is added to the queue that Alpha maintains for messages to send to Beta. Each message in a given queue has a 64-bit sequence number, which starts at zero when the network is first created, and then increments with each new message sent.

Each member of Alpha will, at random intervals, check to see if there are any messages in any outgoing queues, and attempt to send one of the queues. When they see that there are messages intended for Beta, they will call a random member of Beta, and give them all messages in the queue, along with the proof that this queue is part of the current signed state for Alpha.

When a member of Beta receives such a list of messages from the member of Alpha, the Beta member submits a transaction to Beta that has the messages and the proof that they are part of the signed state. If they see that a message has already been submitted, then they won't submit it again. Though sometimes the same message may be submitted twice at the same time. In that case, the sequence numbers will match, so the duplicate will be ignored, and no harm is done.

All messages between two particular shards will be processed in order of sequence number. So, if Alpha sends a message to Beta, and it is put into a transaction within Beta, then when that transaction reaches consensus, at that point in the consensus order, its sequence number will be checked. If it is the next

message in sequence, then its effect is performed immediately. If the sequence number shows that one or more other messages have been skipped, then it has no effect and is ignored. In that case, the other messages will eventually reach consensus, and then the skipped message will be submitted again, and will have an effect.

When Beta processes a message with Alpha with the expected next sequence number from Alpha, then it increments the count of the number of messages from Alpha that have been processed. So each shard maintains a single number for each of the other shards, which is the latest sequence number from that other shard that has been processed.

After Alpha has sent to the message to Beta, that message remains in the outgoing queue, and attempts will repeatedly be made to send it. Eventually, a member of Apha will contact a member of Beta to send that message, but will receive back a proof that the message has already been processed. That proof shows that the signed state contains Beta's sequence count for messages received from Alpha, and that the count is now higher than the message in the queue. At that point, the member of Alpha wraps the proof in a transaction, and gossips it out to Alpha. When it reaches consensus order, at that point the message is deleted from the outgoing queue in the shared state.

For the example of transferring cryptocurrency from Alice to Bob, we could say that there is "finality" when we know that the transfer is valid, that Alice had sufficient funds, and that Bob will certainly receive the funds. If this transfer is for Alice to buy a product from Bob, then finality is point in time where it is safe for Bob to give the product to Alice. The time to finality is actually as short as the consensus time for a single shard. Because once consensus has been reached on that initial transaction, it is certain that Alpha will send a message to Beta, and that Beta will process it, and that Bob's account will receive the transfer. So finality is as fast as consensus.

If a transfer is from one source account to two destination accounts, finality is still just as fast. As soon as the initial transaction reaches consensus, it will be known whether it had sufficient funds, and so whether the two messages will be sent.

However, if a single transaction is to transfer from two source accounts to one destination account, and the source accounts are in different shards, then finality will be slower. Because it will have to involve another type of message: a "hold", which is later followed by a "release".

For example, suppose a transaction is created to transfer 2 coins from Alice in Alpha shard and 3 coins from Bob in Beta shard, with the 5 coins being transferred to Gina in Gamma shard. This is intended to be atomic, so that nothing will happen unless Alice and Bob both have sufficient funds for the transfer.

To achieve this, the transaction must be signed by both Alice and Bob, and must be submitted to the Alpha shard. When it reaches consensus, it causes a "hold" to be put on 2 coins in Alice's account. This means that 2 coin's worth of the account is temporarily frozen. While it is frozen, Alice is still free to receive funds and to transfer out funds, but can't do any transfer that would decrease her balance to less than 2 coins.

At the same time that Alpha puts a hold on 2 coins for Alice, it also sends a message to Beta requesting a hold of 3 coins for Bob. This message does not need to be signed by Bob, because it is coming from Alpha, and Alpha has already checked that Bob had signed the transaction.

When the message is received and reaches consensus, Beta will attempt to put a hold on Bob's account for 3 coins. If he has sufficient funds, it succeeds. If he has less than 3 coins, then it fails, and no hold is put on him at all. Beta then sends back to Alpha a message saying whether the hold was successful.

When Alpha receives a reply that the hold was successful, Alpha then decrements Alice's account by 2 coins (which also removes the hold), and sends a message to Beta saying to decrement Bob's account by 3 coins (removing his hold) and sending a message to Gamma to increment Gina's account by 5 coins.

On the other hand, if Beta's message said the hold failed because Bob did not have sufficient coins, then Alpha simply released the hold on Alice's account, and considers the entire transaction to have failed. None of the three balances change.

Note that when this all started, with Alpha processing the initial transaction, it was possible for Alpha to calculate how many messages would be involved in the entire process: 4 messages in this example. Alpha will therefore check that the transaction included authorization of a service fee that included the fee for the service of sending those 4 messages. Hedera then automatically makes payments to the nodes that created each of the message transactions that were handled (and not ignored as duplicates). This acts as incentive for nodes to do the work of sending messages to other shards, receiving confirmations of receipt from them, and creating the transactions that contain those messages and confirmations.

Appendix 3: Hashgraph

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

LEEMON BAIRD
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: February 16, 2018

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.

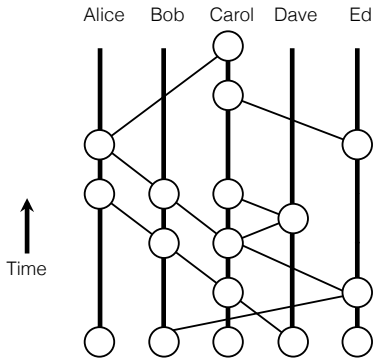


FIGURE 1. Gossip history as a directed graph. The history of any gossip protocol can be represented by a graph where each member is one column of vertices. When Alice receives gossip from Bob, telling her everything he knows, that gossip event is represented by a vertex in the Alice column, with two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

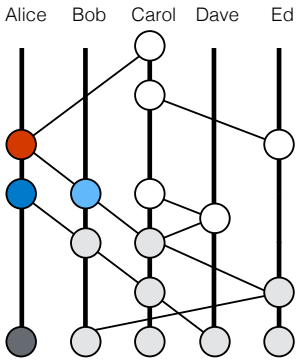


FIGURE 2. The hashgraph data structure. Alice creates an *event* (red) recording the occurrence of Bob doing a gossip sync to her and telling her everything he knows. The event contains a hash of two parent events (blue): the self-parent (dark blue) by the same creator Alice, and the other-parent (light 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

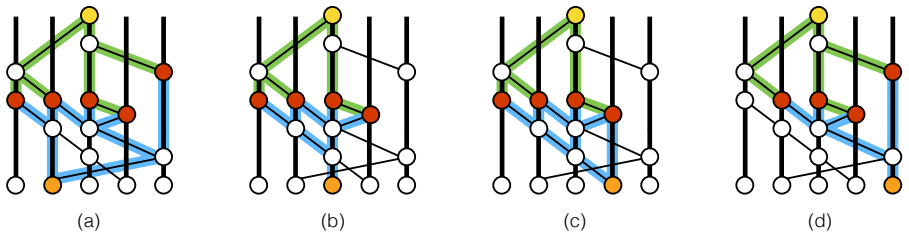


FIGURE 3. Illustration of strongly seeing. In each hashgraph, the yellow event at the top can *strongly see* one of the orange events on the bottom row. There are $n = 5$ members, so the least integer greater than $2n/3$ is 4. In (d), one event (orange) is an ancestor of each of 4 intermediate events 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. (The set of *unique famous witnesses* in a round is defined to be the same as the set of famous witnesses, except that all famous witness from a given member are removed if that member had more than one famous witness in that round).

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 $e = \{p, h, t, i, s\}$ where:

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

$$
\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 (\text{witness}(x) \wedge \text{witness}(y) \\
&\quad \wedge \text{diff}(x,y) > d \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) \\ \text{see}(x,y) & \text{if } \neg\,\text{copyVote}(x) \wedge \text{diff}(x,y) = d \\ 1 = \text{middleBit}(\text{signature}(x)) & \text{if } \neg\,\text{copyVote}(x) \wedge \text{diff}(x,y) \neq d \\ & \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}
$$