The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers

Leemon Baird
Swirlds Inc. and Hedera Hashgraph
Dallas, Texas, USA
Email: leemon@hedera.com

Atul Luykx
Swirlds Inc.
San Francisco, California, USA
Email: atul.luykx@swirlds.com

***Abstract*—Atomic broadcast protocols are increasingly used to build distributed ledgers. The most robust protocols achieve byzantine fault tolerance (BFT) and operate in asynchronous networks. Recent proposals such as HoneyBadgerBFT (ACM CCS '16) and BEAT (ACM CCS '18) achieve optimal communication complexity, growing linearly as a function of the number of nodes present. Although asymptotically optimal, their practical performance precludes their use in demanding applications. Further performance improvements to HoneyBadgerBFT and BEAT are not obvious as they run two separate sub-protocols for broadcast and voting, each of which has already been optimized. We describe how hashgraph — an asynchronous BFT atomic broadcast protocol (ABFT) — departs in structure from prior work by not using communication to vote, only to broadcast transactions. We perform an extensive empirical study to understand how hashgraph's structure affects performance. We observe that hashgraph can improve latency by an order of magnitude over HoneyBadgerBFT and BEAT, while keeping throughput constant with the same number of nodes; similarly, throughput can increase by up to an order of magnitude while maintaining latency. Furthermore, we test hashgraph's capability for high performance, and conclude that it can achieve sufficiently high throughput and low latency to support demanding practical applications.**

KEYWORDS. Byzantine, Byzantine agreement, Byzantine fault tolerance, ABFT, replicated state machine, atomic broadcast, hashgraph, gossip about gossip, virtual voting

I. INTRODUCTION

Atomic broadcast protocols allow a network of nodes to reach consensus on an ordering of received transactions. They are often used to build distributed ledgers: users submit transactions to nodes, who in turn order the transactions by running an atomic broadcast protocol. The ordered output of the atomic broadcast protocol forms part of the "ledger".

Distributed ledger applications, such as payment networks and marketplaces, place stringent requirements on security and performance. When run over public networks, distributed ledgers can suffer from unreliable connections and denial-of-service attacks. Participants in the ledger might have incentives to deviate from the protocols. Furthermore, performance requires low latency to place transactions in the ledger, while supporting high transaction throughput.

Atomic broadcast protocols which are Byzantine Fault Tolerant (BFT) [20] operate in the presence of adversarial nodes, making them suitable for distributed ledger design. BFT protocols which can run in *asynchronous* networks are particularly robust against attacks that can be mounted over public networks. Such asynchronous BFT (ABFT) protocols ensure that transactions are processed without timing assumptions, a sensible property to have if adversaries can arbitrarily manipulate the network in a distributed ledger.

In contrast, BFT protocols that require network synchrony or partial synchrony only progress when communication is not delayed beyond a fixed amount of time, and could fail otherwise. Miller et al. [22] demonstrate situations where ABFT protocols advance where others cannot, and point out that ABFT protocols progress as soon as messages are delivered after network partitions, whereas non-ABFT protocols are slow to recover.

Although ABFT research has a long history [17], only recent atomic broadcast protocols achieve the optimal asymptotic communication complexity of $O(n)$ bits flowing through a network with n nodes, or $O(1)$ bits per node. At a high level, performance is achieved by optimizing two tasks:

1) reliably broadcasting transactions, and
2) reaching consensus on what transactions to output.

Often those two tasks take the form of sub-protocols, executed repeatedly. HoneyBadgerBFT [22] and BEAT [15] use either Bracha's broadcast [6] or one based on erasure codes [11] to broadcast transactions, and then optimize a Binary Byzantine Agreement (BBA) protocol to cast votes and reach agreement. Their networks proceed in rounds, where in each round n broadcast instances are executed alongside n BBA instances, the latter of which only conclude when sufficiently many broadcast instances have been completed to ensure progress.

Contributions

Although HoneyBadgerBFT and BEAT achieve optimal asymptotic communication complexity, performance can improve by executing broadcast and voting simultaneously. To that end, we demonstrate how the *hashgraph protocol*, an asynchronous BFT atomic broadcast protocol, improves throughput and latency.

Like prior work, the hashgraph protocol contains an underlying reliable broadcast protocol to disseminate transactions. Unlike prior work, each node in the protocol maintains the communication history among all nodes in the network — the *hashgraph* — and this history, together with the transactions, is propagated throughout the network.

Beyond sending the hashgraph, the protocol does not need further communication to reach consensus, as the nodes can base their decisions on the information they have. By optimizing communication of the hashgraph and removing the need for additional communication for Byzantine Agreement, the protocol achieves optimal asymptotic communication complexity and achieves high practical performance.

We describe the hashgraph protocol and illustrate its performance through an extensive empirical study. We evaluate hashgraph under the same settings as HoneyBadgerBFT by Miller et al. [22] and BEAT by Duan et al. [15], demonstrating that hashgraph performs as well as the state-of-the-art, and in some instances improves either latency or throughput by an order of magnitude. We also test hashgraph in high performance settings, running tests covering 4 to 128 nodes distributed over 1, 2, and 8 regions. For example, hashgraph achieves 20,000 transactions per second with 3.5 sec latency running over 32 nodes spread across 8 regions.

We only state the formal security results; the proofs are contained in the technical report [4]. Furthermore, the proofs have been

978-1-7281-6371-0/20/$31.00 ©2020 IEEE

formalized and verified by computer using the Coq proof assistant system [27].[1]

II. Related Work

We focus on describing related work within the context of asynchronous BFT protocols. More comprehensive overviews can be found in one of the many surveys, such as [14].

BFT consensus protocols have been designed under a wide range of network synchrony assumptions. Synchronous networks guarantee that messages are delivered within some fixed amount of time Δ. Asynchronous networks have unbounded message delays. Many well-known BFT protocols, such as PBFT [13] and Tendermint [7], assume that networks are partially synchronous [16], where networks behave asynchronously for some time, but eventually become synchronous.

By the FLP theorem [17], no deterministic Byzantine system can be asynchronous, with unbounded message delays, and still guarantee consensus. It is possible for a nondeterministic system to achieve consensus with probability one. Ben-Or [5] introduces a *local coin* mechanism, where nodes use local randomness to randomize the protocol. Rabin [25] uses a shared coin (or common coin) to ensure all nodes use the same randomness. Local coin algorithms typically terminate in an expected exponential number of communication steps. While shared coin algorithms can terminate in an expected constant number of steps [9], but there is no limit on how long each step can take. Hashgraph can use either the local or shared coin mechanisms.

A crucial component to many consensus protocols is reliable broadcast. Examples include Bracha's protocol [6] and Cachin and Tessaro's [11], both of which operate in ABFT settings. Protocols have been designed to achieve consensus on different types of inputs. Correia et al. [14] list some types, including binary, multi-valued, and vector agreement protocols, none of which output an order.

Atomic broadcast protocols guarantee agreement on a sequence of requests. Atomic broadcast is also known as total order, or simply consensus [12], and can be used to achieve *state machine replication*, and *ledger consensus* [18]. Most atomic broadcast protocols use other types of consensus protocols as building blocks; for example, Cachin et al. [8], [10], Miller et al. [22], and RITAS [23] all use variants of binary agreement and reliable broadcast.

Moser and Melliar-Smith [24] propose voting algorithms which can be run on a communication history to achieve BFT atomic broadcast, yet their algorithms only work in a weakly asynchronous network, where assumptions are made on the distribution of messages sent by the network and the attacker does not have full control. In contrast, the hashgraph protocol does not make any such assumption.

Prior proposals for asynchronous BFT atomic broadcast include those by Cachin et al. [8], [10], which do not achieve optimal communication complexity, and those by Kursawe and Shoup [19] and Ramasamy and Cachin [26], which do not communicate optimally when the network is not honest. There are few ABFT protocols other than hashgraph which achieve optimal communication complexity; HoneyBadgerBFT and BEAT are two examples. Recently, Lasy [21] studied variants of the hashgraph protocol.

III. System and Threat Model

Adversaries are computationally bounded, and cannot break the digital signature schemes and cryptographic hash functions used. Furthermore, each node has a key pair for digital signatures, and all nodes know the set of nodes and the public key for each node.

Throughout the paper, n denotes the number of nodes in the protocol. Out of the n nodes, we assume that more than $2n/3$ are honest, so fewer than $n/3$ are not honest[2]. The malicious nodes may collude, and we assume without loss of generality that they are coordinated by a single adversary.

For any two honest nodes A and B, A will eventually try to sync with B, and if A repeatedly tries to send B messages, A will eventually succeed. No other assumptions are made about network reliability, network speed, or timeout periods. Specifically, adversaries may delete and delay messages arbitrarily, subject to the constraint that a message between honest nodes that is sent repeatedly must eventually get through.

Following prior work [15], [22], we model how the protocols operate in practice where users continuously submit transactions to one or more nodes by assuming that adversaries give nodes a continual stream of transactions as inputs. During execution of the protocol, nodes will continuously *output* transactions, representing their view of the consensus order.

We follow the property-based description of atomic broadcast by Cachin et al. [8], Miller et al. [22], and Duan et al. [15].

Definition 1 (Atomic Broadcast)**.** Atomic broadcast protocols must satisfy the following properties in the face of an adversary:

Agreement if any honest node outputs a transaction, then every honest node outputs that transaction.

Total Order for all $i > 0$, if T is the ith output of an honest node and T' the ith output of another honest node, then $T = T'$.

Liveness if a transaction is input to an honest node, then it is eventually output by every honest node.

IV. The Hashgraph Protocol

The hashgraph consensus protocol is given by Algorithms 1, 2, 3, and 4. As mentioned in the introduction, a proof of the following theorem is provided in the technical report [4].

Theorem 1 (Asynchronous Byzantine Fault Tolerance Theorem)**.** *Each transaction submitted to honest nodes will eventually be assigned the same consensus position in the total order of events by each honest node, with probability 1, satisfying agreement, liveness, and total order.*

Fundamental to the operation of the hashgraph protocol is how it encodes and distributes the communication history — the *hashgraph* — among all the nodes, so that each node has a local copy. Section IV-A discusses how the hashgraph is communicated, and the subsequent subsections describe how the hashgraph is used to vote and achieve consensus without further communication.

A. Gossip About Gossip: Communicating the Hashgraph

Algorithm 1: Main procedure of the hashgraph protocol.

```
while True do
    select a node at random
    sync all events with that node
    create a new event
    divideRounds
    decideFame
    findOrder
```

[1]The Coq source code can be found at the following address: https://swirlds.com/downloads/hashgraph-coq.zip.

[2]In a *proof-of-stake* system, nodes have different weights in the voting called stake, and n represents the total amount of stake.

The hashgraph protocol uses gossip to communicate, as described in Algorithm 1. A node, Alice, chooses another node at random, say Bob, and then Alice sends Bob what she knows that he does not know, and vice versa. Alice repeats with a different random node, as do all other nodes. Information spreads through the network to every node.

The communication is stored in the hashgraph data structure illustrated in Figure 1. Nodes create *events* when they receive gossip syncs from others. An event is a tuple consisting of the hashes of two parent events, a list of transactions, a timestamp, and a signature for the rest of the tuple.

The tuples contains the timestamp of when it was created. The algorithm would work is this were removed, and a zero were used for every timestamp. However, the use of timestamps allows us to generate a consensus timestamp on every transaction which is important for practical applications.

If an event x created by node A contains a parent hash pointing to an event z created by A as well, we call z a *self*-parent of x. In Figure 1 A2 is a self-parent of A3, and B3 is a parent of A3. An event x is an *ancestor* of event y if x is y, or a parent of an ancestor of y. It is a *self-ancestor* of y if x is y, or a self-parent of a self-ancestor of y. We will also use the complementary terms *descendant* and *self-descendant*.

Nodes only accept events that have a valid signature and contain valid hashes referring to events they already have. For example in Figure 1, Carol receives A3 and can use the event's signature to verify that A3 was created by Alice when Bob synced with her. Carol can request the ancestors[3] of A3, which she verifies using the hashes contained in the event.

If two nodes both have the same event, then they can verifiably recover its ancestors, and will agree on the edges in the subgraph of those ancestors. Therefore, any two nodes have *consistent* hashgraphs: for any event x contained in hashgraphs A and B, both A and B contain the same set of ancestors for x, with the same parent edges between those ancestors.

B. Deriving Consistent Output From A Hashgraph

If all nodes have the same hashgraph, then they can calculate an ordering of events using a deterministic function of that hashgraph and reach consensus without further communication. However, different nodes may not have the same hashgraph; they will typically have the same older events, but differ in which recent ones they have. Therefore the algorithm must be designed so that a conclusion based on a deterministic function of the hashgraph at one moment will not change as the hashgraph grows.

For example, a node might have all the events in Figure 1 at one moment, and conclude that D1 is an ancestor of C6. Other nodes might grow this hashgraph differently: at one moment Bob might have D1 and not C6, and not yet have concluded whether D1 is an ancestor of C6. But when Bob accepts C6 via a sync, Bob must already have received all the ancestors of C6 so he will see the same paths between C6 and D1 and conclude that D1 is an ancestor of C6.

Thus, ancestry is an example of a deterministic function of a hashgraph that at any given moment either gives no conclusion at all, or it makes a conclusion that will never change, and all nodes eventually come to that same conclusion.

[3]The ancestors of A3 are A2, A1, B3, B2, B1, C3, C2, C1, D1, E2, and E1.

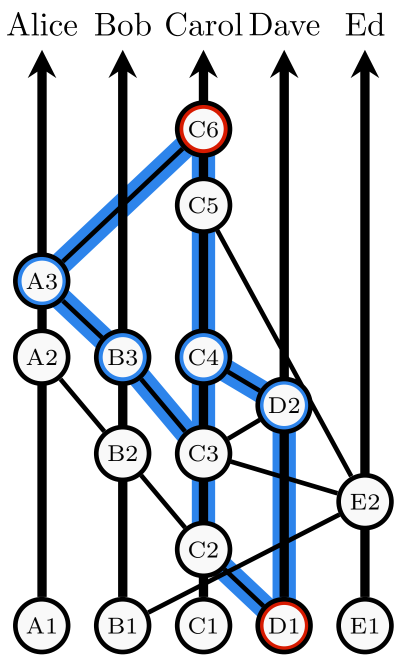

Fig. 1: The hashgraph data structure. Older events are located towards the bottom of the graph. C6 can strongly see D1 via A3, B3, C4, and D2.

C. Using Strongly Seeing To Handle Forks

Nodes might maliciously deviate from the protocol. For example, a node Alice could create two events y and z with the same self-parent x, that is, a *fork*: events y, z with the same creator, where neither is an ancestor of the other. If Alice gossips y to Bob and z to Carol, then Bob's hashgraph could contain y and not z, while Carol's hashgraph could have z and not y. To detect and prevent such forks, Bob and Carol need to know what other nodes' view is of the hashgraph.

Core to virtual voting are the concepts of an event x *seeing* or *strongly* seeing an event y. Seeing enhances the relationship "descendant of" by excluding forks: the event x sees y if x is a descendant of y, and y's ancestors do not include a fork by y's creator. Strongly seeing enhances seeing by having many other nodes "vouch for" y's lack of forks: the event x strongly sees y if x can see more than $2n/3$ events by different nodes, each of which can see y.

Strongly seeing is illustrated in Figure 1. Four events are needed for strongly seeing, because this example has $n = 5$ nodes, and the least integer greater than $2n/3$ is four. C6 can see D1 through four events by different nodes — A3, B3, C4, and D2. Alternatively, C6 could see through C6 and D1 because an event always sees itself.

Strongly seeing ensures that even if an attacker tries to cheat by forking, they cannot make different nodes strongly see different events. If y and z are on different branches of a fork, then x can strongly see either y or z, but not both. This is because the set of events through which x sees y and the set of events through which x sees z overlap in more than $n/3$ nodes. Since fewer than $n/3$ nodes are malicious, at least one node in the overlap must be honest, and that node cannot see both y and z since they are on a fork.

Therefore, strongly seeing is an example of a conclusion that is a deterministic function of a hashgraph that is guaranteed to eventually be reached by all nodes no matter what order they receive the hashgraph. All components of the hashgraph consensus algorithm have mathematical proofs that they have this consistency property.

D. Deciding Order

Virtual voting relies on strongly seeing to ensure nodes collect consistent votes. In Figure 1, if one node concludes that C6 collected a given vote from Dave by looking at event D1, then all other nodes are also guaranteed to conclude that C6 received the vote from D1 and not from a fork of D1.

To determine event ordering, each node performs the following operations on its own hashgraph as follows.

1) `divideRounds` *(Algorithm 2):* The hashgraph is divided into rounds, where an event is placed into a new round when it strongly sees more than $2n/3$ events in the prior round. Furthermore, rather than voting on all events, voting focuses on *witnesses*: the first event created by a node in a round.

2) `decideFame` *(Algorithm 3):* The voting protocol runs an election for each witness to determine if it is *famous*, meaning that event was received by many nodes by the start of the next round; we call the famous witness *unique* if there are no other famous witnesses from the same creator. For a witness x in round r, each witness in round $r + 1$ will vote that x is famous if it can see x.

If more than $2n/3$ witnesses agree on whether x 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 votes from 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. It is unlikely that a coin round will ever occur in a practical implementation, because convergence is fast when the honest nodes are allowed to communicate freely, even if some of them are shut down by attackers. But coin rounds are included here for theoretical completeness.

3) `findOrder` *(Algorithm 4):* Once consensus has been reached on whether each witness in a given round is famous, `findOrder` determines a consensus timestamp and a consensus total order on older events.

First, the *round received* of event x is calculated: the first round r in which all the unique famous witnesses are descendants of x, and all witnesses' fame is decided for rounds less than or equal to r.

Then, the *received time* is calculated. Say x has a received round of r, and Alice created a unique famous witness y in round r. The algorithm finds the earliest self-ancestor z of y that had learned of x. Then the timestamp that Alice assigns z is when 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 any arbitrary deterministic method. In Algorithm 4, the algorithm simply sorts by signature, after the signature is whitened by XORing with the signatures of all the unique famous witnesses in the received round.

Algorithm 2: The `divideRounds` procedure.

```
foreach event x do
    if x has parents then
        r ← max round of parents of x
    else
        r ← 1
    if x strongly sees > 2n/3 round r witnesses then
        x.round ← r + 1
    else
        x.round ← r
    x.witness ← (x has no self parent) or (x.round >
      x.selfParent.round)
```

Algorithm 3: The `decideFrame` procedure.

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

V. Performance

We illustrate that the hashgraph consensus algorithm performs competitively with state-of-the-art consensus protocols. Hashgraph can reach peak throughputs of hundreds of thousands of transactions per sec (tps), while maintaining low latency to reach confirmation.

A. Implementation Details

The hashgraph consensus algorithm uses two cryptographic primitives: digital signatures and hash functions. In the implementation tested here, which uses the Java standard crypto library, the digital signature is RSA with 3072 bit keys and the hash function is SHA-384. All communication is done over TLS 1.2 with RSA 3072, Diffie Hellman with ephemeral keys and AES256-GCM.

This implementation operates assuming the number of malicious nodes is set to the largest integer smaller than $n/3$, which is the

Algorithm 4: The `findOrder` procedure.

```
foreach event $x$ do
    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 $\leftarrow r$
        $s \leftarrow$ set of events $z$ where $z$ is a self-ancestor of a
          round $r$ unique famous witness, and $x$ is an ancestor
          of $z$ but not of $z$'s self-parent
        $x$.consensusTimestamp $\leftarrow$ median of the timestamps
          of the events in $s$
/* The whitened signature is the signature
   XORed with the signatures of all unique
   famous witnesses in the received round.
   */
return list of events $x$ where
  $x$.roundReceived $\neq$ UNDECIDED, sorted by roundReceived,
  then ties sorted by consensusTimestamp, then by whitened
  signature
```

recommended way to run the hashgraph protocol. The gossip protocol requires that nodes randomly sync with each other. Nodes can perform multiple syncs in parallel. In our implementation, each node performs up to fifteen parallel syncs. Furthermore, the events in the implementation contain up to 1024 transactions. Both the number of parallel syncs and transactions per event are chosen to optimize latency given a desired throughput.

B. Asymptotic Communication Complexity

If the transaction size is B bits, then the hashgraph protocol's communication cost to gossip the transaction to all nodes is $O(B)$ per node. The additional overhead for sending an event will include two hashes and a time created, so sending that event is $O(1)$ per node. If only a single transaction were sent to the network, then many empty events would have to be created and gossiped to reach consensus on that single transaction. However, hashgraph is designed for high-throughput settings, where there is a steady stream of transactions entering the network, and most events contain at least one transaction. In that case, the amortized communication complexity per transaction to reach consensus is just $O(B)$ per node. A single event, with just two hashes and a time, can act as a "vote" in many different consensus elections simultaneously, because of the virtual voting.

C. Evaluation

We evaluate hashgraph under two setups using Amazon EC2 [3]:

1) one to establish a baseline comparison, by recreating the conditions of the HoneyBadgerBFT and BEAT experimental results using t2.medium instances with two virtual CPUs, 4GB memory, and up to 1 Gbps network performance [2],
2) and to demonstrate high performance, using m4.4xlarge instances with 16 virtual CPUs, 64 GB memory [2], and up to 2 Gbps network performance [1].

The instances are distributed evenly across eight regions[4].

[4]The eight regions are US East, US West, Canada, Sao Paulo, Japan, Australia, South Korea, and Germany.

TABLE I: Hashgraph performance using Amazon t2.medium instances. Unless specified otherwise, nodes are distributed evenly across eight regions. HG stands for hashgraph and HB for HoneyBadgerBFT. Performance figures for HB and BEAT in the first table are from [15], and for HB in the second table are from [22].

Protocol	# nodes	tps	latency (sec)
HG	4	22,000	7
HB	4	1,500	7
BEAT0	4	2,000	7
BEAT1	4	600	7
BEAT2	4	700	7

	tps		latency (sec)	
# nodes	HB	HG	HB	HG
32	7,500	7,000	5	5.1
40	10,000	8,000	10	8.4
48	12,000	6,000	48	8.4
56	11,000	6,000	100	10
64	5,000	5,000	200	12.5
104	2,500	2,500	250	25

As in Miller et al. [22] and Duan et al. [15], throughput is measured in 250 byte tps, and latency in seconds. We compare with Duan et al.'s BEAT0, BEAT1, and BEAT2 protocols, which achieve general state machine replication. BEAT3 and BEAT4 are out of scope as they do not support general state machine replication.

As with prior work, these tests are for achieving consensus on transaction order and timestamps, and assumes nodes are honest. They do not include the time to process transactions.

Latency is measured as the average number of seconds from when a client first submits a transaction to a node until when the node knows the transaction's consensus order and timestamp.

Setup 1: Baseline Comparison with t2.medium. We recreate the experimental settings of Miller et al. [22] and Duan et al. [15] to compare throughput and latency, including the fact that nodes are honest. Furthermore, it is important to note that Miller et al.'s performance figures are for the protocol tuned to *a tolerance of up to $n/4$ malicious nodes*; in contrast, our figures for hashgraph are for a tolerance up to $n/3$ malicious nodes. When available, results for HoneyBadgerBFT (HB) and BEAT are estimated from their respective papers, in their favor (tps overestimated, latency underestimated).

Duan et al. [15] present performance figures for HoneyBadgerBFT and BEAT with four nodes all in one region, to simulate a LAN setting. HoneyBadgerBFT and BEAT achieve throughput of less than 10,000 tps; latency is not given. In our experiments, we find that hashgraph is able to achieve 27,000 tps with a latency of 0.35 sec, and 37,000 tps with a latency of 0.5 sec.

Table I gives performance results in settings where nodes are evenly distributed across regions. In all cases, hashgraph performs at least as well as HoneyBadgerBFT and BEAT. In some cases, hashgraph improves by an order of magnitude. For example, with four nodes across four regions, hashgraph is able to achieve 22,000 tps with latency of 7 sec, while HoneyBadgerBFT and BEAT achieve at most 2,000 tps. With 64 nodes across eight regions, hashgraph can achieve the same 5,000 tps as HoneyBadgerBFT, but with only 12.5 sec latency instead of 200 sec latency.

Setup 2: High Performance with m4.4xlarge. Results are given in

Figure 2. Each line is for a different number of nodes, shown to the right. The horizontal axis is the number of 250-byte transactions per sec (tps) placed in consensus order. In these experiments, throughput ranges from about 4,000 tps up to almost 250,000 tps. On most lines, the second dot from the left is 4,000 tps.

In nearly all experiments, latency was under 20 sec, and various experiments had latencies down to less than 0.04 sec.

The graphs illustrate tradeoffs between throughput, latency, number of nodes, and geographic distribution. For 32 nodes running at 20,000 tps, consensus finality is reached in 3.5 sec when the network is spread across 8 regions. When the network spans two regions, latency is 1.7 sec, and in a single region, it drops to 0.75 sec.

If latency needs to be kept under 7 seconds, as might be required for credit cards, while still achieving 50,000 transactions per second, one can use 32 computers in eight regions, 64 computers in two regions, or 128 computers in one region.

VI. Conclusions

As described earlier, hashgraph departs in structure from HoneyBadgerBFT and BEAT by not using communication to vote, only to broadcast transactions. Our performance evaluation shows that hashgraph significantly outperforms both BEAT and HoneyBadger by an order of magnitude in either latency or throughput, thereby supporting the claim that hashgraph's novel structure leads to concrete performance improvements.

Beyond a comparison with HoneyBadgerBFT and BEAT, we also tested hashgraph's capability for high performance. Based on our study, we conclude that hashgraph can achieve sufficient performance to support demanding practical applications, while still guaranteeing security in an ABFT setting.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and suggestions to improve the paper. We would like to thank Jeffrey Tang and Mehernosh (Nosh) Mody for their help in getting performance figures for hashgraph. We would also like to thank Mance Harmon for the many discussions about hashgraph.

References

[1] EC2 network performance demystified: m3 and m4. https://cloudonaut.io/ec2-network-performance-demystified-m3-m4/. Accessed: 2020-03-05.

[2] General Purpose Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html. Accessed: 2020-03-05.

[3] What Is Amazon EC2? https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed: 2020-03-05.

[4] L. Baird. Hashgraph consensus: fair, fast, byzantine fault tolerance. *Swirlds Tech Report, Tech. Rep.*, 2016.

[5] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In R. L. Probert, N. A. Lynch, and N. Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983.

[6] G. Bracha. An asynchronou [(n-1)/3]-resilient consensus protocol. In T. Kameda, J. Misra, J. G. Peters, and N. Santoro, editors, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, pages 154–162. ACM, 1984.

[7] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

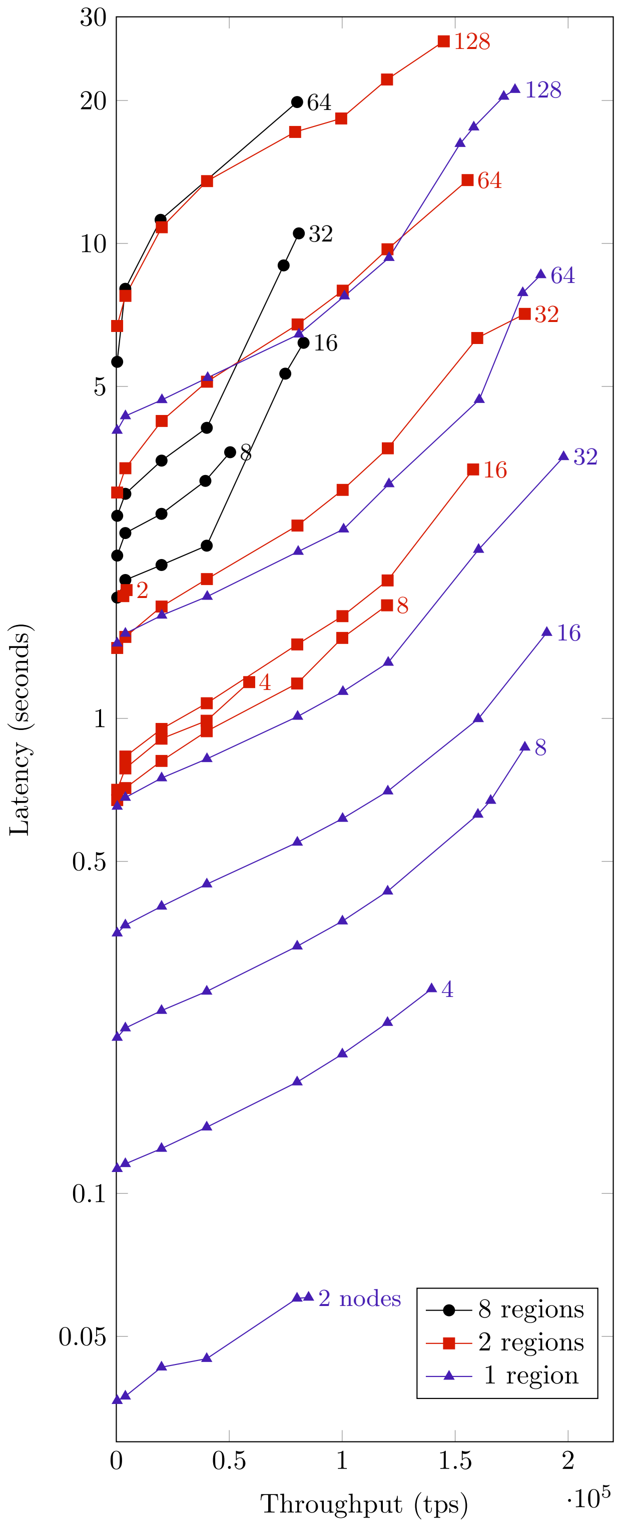

Fig. 2: Hashgraph latency versus throughput, where throughput is measured in 250-byte transactions per second.

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

[9] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In G. Neiger, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, pages 123–132. ACM, 2000.

[10] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 167–176. IEEE Computer Society, 2002.

[11] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In P. Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

[12] C. Cachin and M. Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[14] M. Correia, G. S. Veronese, N. F. Neves, and P. Veríssimo. Byzantine consensus in asynchronous message-passing systems: a survey. *IJCCBS*, 2(2):141–161, 2011.

[15] S. Duan, M. K. Reiter, and H. Zhang. BEAT: asynchronous BFT made practical. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041. ACM, 2018.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[18] J. A. Garay and A. Kiayias. Sok: A consensus taxonomy in the blockchain era. *IACR Cryptology ePrint Archive*, 2018:754, 2018.

[19] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2005.

[20] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[21] T. Lasy. From hashgraph to a family of atomic broadcast algorithms, 2019.

[22] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[23] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. RITAS: services for randomized intrusion tolerance. *IEEE Trans. Dependable Sec. Comput.*, 8(1):122–136, 2011.

[24] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Inf. Comput.*, 150(1):75–111, 1999.

[25] M. O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, page 403–409, USA, 1983. IEEE Computer Society.

[26] H. V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In J. H. Anderson, G. Prencipe, and R. Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2005.

[27] T. C. D. Team. The coq proof assistant, version 8.7.0, Oct. 2017.