COMP515 - Project Proposal
A Distributed Ledger Application using Hashgraph

Erhan Tezcan
Department of Computer Engineering
Koç University
Istanbul, Turkey
etezcan19@ku.edu.tr

Ahmet Uysal
Department of Computer Engineering
Koç University
Istanbul, Turkey
auysal16@ku.edu.tr

***Abstract*—In this project, we implement Hashgraph: a fast, fair, secure and asynchronously Byzantine Fault Tolerant distributed ledger technology. We implement this using Go, and evaluate our results using AWS EC2 t2.micro machines with 4 nodes in 1 region. We also implement a Hashgraph Visualizer, which works in parallel to the original distributed ledger and visualizes the underlying local hashgraph structure as it evolves throughout the program. Our evaluations and visualizations are satisfactory, and we provide algorithmic details regarding Hashgraph, with several optimizations we did in our code.**

***Index Terms*—hashgraph, consensus algorithms, byzantine fault tolerance, distributed ledger systems**

I. Introduction

Hashgraph [1] [2] is both a data structure and a fast, fair and secure consensus algorithm invented by Leemon Baird. Regarding the two cited papers, [1] is a technical report and is the original paper by the author, and has the proofs of the algorithm. [2] is the more formal paper, and is expected to be published in IEEE COINS 2020. In a consensus problem, we have a community of users where no single member is trusted by everyone, yet we require an overall consensus on the order of user generated transactions. In other words, in a system where no one trusts each other, we generate trust. Another very famous approach on this is Blockchain [4], however, Blockchain is orders of magnitude slower than what Hashgraph claims to be, and their mechanisms are different. In Blockchain, we mostly have Proof of Work (PoW) or Proof of Stake (PoS), but these approaches are not "fair" in the context of a distributed ledger system. A fair approach would be to instead do voting, but voting is a bandwidth demanding algorithm and it is not practical. However, Hashgraph wanted to do voting nevertheless, and for this they have two important tools:

- Gossip about Gossip
- Virtual Voting

In voting, every node must tell every other node how they are voting. This requirement was the main performance issue. To tackle this, nodes can gossip about gossip, in other words, nodes can gossip each other the overall conversation, so in the end, all nodes will know what every node knows. This uniform knowledge opens way for what is called "Virtual Voting". Unlike normal voting, nodes do not need to speak at the voting process, because they already know what every other knows. In other words, they already know what other nodes would be voting for. Leemon also proves that given n nodes where $2n/3$ of them are *honest*, which also means non-malicious, then this virtual voting is guaranteed to be safe and fault tolerant. Note that gossip itself is not a new protocol [3], however the idea of gossiping the gossip itself is a novelty by hashgraph.

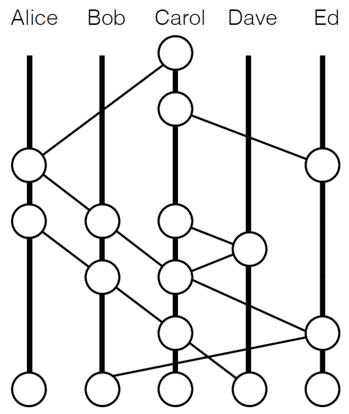


Fig. 1. Gossip history as a directed graph, taken from [1]. Time starts at the bottom of the graph, and flows forward as we go up in the graph.

An example Hashgraph is given in figure 1. When Alice receives gossip from Bob, this is represented as a vertex under the Alice column with two edges going downward, to the immediately preceding gossip events by Alice and Bob. The bottom of this Hashgraph can be thought of as $t = 0$, the starting point. As we see, all members (nodes) start by creating an *event* which is just a small data structure in memory. Each event is a container for zero or more *transactions*. The main goal of Hashgraph is for the members of the community to come to a *consensus* on the order of the events, as well as the order of the transactions inside the events, and agree on a timestamp for each event. Thanks to the security aspect of Hashgraph, it is hard for attackers to prevent consensus, or unfairly influence the global order and timestamp of events.

A. Hashgraph Applications and Related Work

Hashgraph, even though it is fairly new, has attracted some attention these past few years. In [7], the author tries to extend the hashgraph algorithm; in [5] and [6] the authors compare hashgraph to other distributed ledger algorithms; in [11] the authors investigate the claimed high performance of hashgraph, with connections to power grids. Given that it is a fast, fair and secure asynchronous Byzantine Fault tolerant (ABFT) consensus algorithm, hashgraph has already became an option to be considered for decentralized peer-to-peer applications. A recent study [8] employs hashgraph consensus in the domain of inter-vehicle (V2V) energy transfer systems. Another study uses it for the IoT domain [9], in their paper they present an IoT ecosystem, where each exchange of a service between a service provider and service consumer is logged as a transaction in a distributed ledger, and this distributed ledger is implemented with a hashgraph.

It should be noted that Hashgraph utilizes an asynchronous BFT, which is a fairly new subject compared to its synchronous counter-part. Other examples of ABFT consensus algorithms are HoneyBadgerBFT [14], a set of protocols collected under BEAT [15], and many more studies on ABFT such as [16] and [17].

II. Concepts and Implementation

We implemented Hashgraph using Go [10], as we have learned it in our class and it is particularly useful in handling distributed network applications. We have extensively studied the paper [2], the technical report [1] and an example by Leemon Baird [12] to be able to implement Hashgraph from scratch. We make several assumptions:

- Every node in the system knows the addresses of every other node in the system.
- No node joins the system at a later time.
- No node leaves the system at a later time.

We created a `package` for the hashgraph, which has 3 structures in it: Node, Transaction and Event. Every node in the system, is a data structure with the fields below:

- `Address`: Basically IP and port of this node.
- `Hashgraph`: The local copy of the actual hashgraph data structure. It is a map from address to a slice of event pointers.
- `Events`: Map from event signature to event pointers. This is done to access an event in $O(1)$, given it's signature.
- `Witnesses`: Map from node address to another map, which maps from *round* number to a single event pointer. We will explain what *round* and *witness* is later. The witness is an often used object in the whole algorithm, so to speed up the access and ease the calculations we have to store pointers to them for every round.
- `FirstRoundOfFameUndecided`: Witnesses can be *famous* or *not famous*, or undecided. This field is a map from address to the round number, such that that round is the first round with the witness where the fame is undecided.
- `FirstEventOfNotConsensusIndex`: This maps from the node address to the index of the first event that has not been reached to a consensus upon. The index is used by the event pointer slice of field `Hashgraph`, explained at the second bullet point.
- `ConsensusEvents`: The slice of event pointers which points to all events that are agreed upon. This is done because we may want to sort this array from time to time. This sorting will also be explained later.
- `TransactionBuffer`: This is a slice of transactions, which we will be receiving from the user from command line and attach to events during our gossip protocol.

A transaction is an object with 3 fields: sender address, receiver address and an amount. The event is the most important object, it has the fields below:

- `Owner`: We need to store the owner address in the event, as this information is required in the algorithm. It is possible to find the owner of an event by searching of course, but this reduces it to $O(1)$.
- `Signature`: Randomly generated sequence of bits by the owner of this event.
- `SelfParentHash`: The signature of the self-parent event. This is the parent that resides in the owner node.
- `OtherParentHash`: The signature of the other-parent event. This is the parent that resides in another node.
- `Timestamp`: Date-time of the creation of this event is timestamped to the object. It is given by `time.Now()` in Go.
- `Transactions`: If the `TransactionBuffer` of a node is empty, this becomes `nil`. Otherwise, we put the whole buffer here and empty the buffer for future events. This array of transactions are piggybacked on the events during gossip.
- `Round`: Every event has a calculated round on creation, and is stored in that event.
- `IsWitness`: A Boolean variable which indicates if an event is a witness or not.
- `IsFamous`: A Boolean variable which indicates if an event is a famous witness or not.
- `IsFameDecided`: A Boolean variable which indicates if an event had a decision regarding its fame.
- `RoundReceived`: This is the round that many nodes come to a consensus on for this event. It may be different than the calculated /codeRound.
- `ConsensusTimestamp`: This is the timestamp that many nodes come to a consensus on for this event.

The goal of the overall system is for every node to have a consensus on the total order of the events and the transactions within them. This order is achieved with respect to `RoundReceived` and within a round with respect to `ConsensusTimestamp`. How the order of transactions is achieved can be summarized in short as:

- Transaction order is the transactions sorted by *event* *order*,

- *Event order* is the events sorted by their *consensus timestamps*,
- *Consensus timestamp* of an event is it's median *received time*,
- *Received times* are for the active nodes only, where an active node means that the node **sent events** to **many nodes** at that **time**. To further explain the bold words:
  - **time** refers to the round,
  - **sent events** means that it has a witness in that round,
  - **many nodes** means that it is a famous witness (which is seen by many)

A. Gossip

Gossip is the heart of this program. Consider two nodes n_A and n_B in a set of nodes N with n nodes in it. When n_A wants to gossip to n_B, it will remotely call a procedure on n_B, which we implemented using `net/rpc` in Go. Every node is actively listening for such remote calls, and this is how gossip works here. The first procedure that is called from n_A at n_B is to learn how many events n_B does not know but n_A does. For this, we basically send the number of events per node we know to the target node, and they respond with the difference of theirs. A negative difference on a node would mean that the target node knows events that we do not know, but this is not an issue, because the gossip protocol ensures that at some point we will get those. For this call, we are interested in positive calls only, which indicates how many events we have to send to the target node. n_A makes a second remote call to n_B with the aforementioned events in the parameter, and this procedure is the most important one in the whole algorithm.

Algorithm 1 Sync Events

```
acquire read/write mutex on Node
fill the local hashgraph with received events
create newEvent
DivideRounds(&newEvent)
DecideFame()
FindOrder()
release read/write mutex on Node
return nil
```

In algorithm 1, we show how this happens. First, the receiving node updates it's local hashgraph. Then, this gossip itself becomes an event. Note that we return `nil` as the RPC call expects to return an error, and it would return `nil` if no error is present.

We need the mutual exclusion locks because several nodes may try to gossip to same node at once, however, their effects should be atomic. Therefore, on the outer scope of the RPC we lock the node, and we release it right before the end.

B. Event & `DivideRounds`

Every node, initially has a local hashgraph with one event in it. It is **their** own initial event. This event has no self-parent nor other-parent. However, it has *round* 1 and it is a *witness*. A round is something we calculate in `DivideRounds` function, for every new event. It gives us the ability to form groups for each round, and a sense of ordering. A witness is the first event with it's round number. Since initial event is the first even with round 1, it is by definition a *witness*. For other events, we show how to calculate the round in algorithm 2. Our algorithm is slightly different than the original, in a sense

Algorithm 2 `DivideRounds` for a new event `x`

```
$r \leftarrow \max(selfParent.Round, otherParent.Round)$
if $x$ strongly sees $> \frac{2n}{3}$ round $r$ witnesses then
    $x.Round \leftarrow r + 1$
else
    $x.Round \leftarrow r$
end if
$x.witness \leftarrow x.Round > x.selfParent.Round$
```

that in the original paper it is checked in this procedure if event x is initial or not, but we do not do that since all new events are non-initial by definition. We shall now explain what it means to *strongly see* or *see*.

C. Strongly See & See

These two characteristics are the most important characteristics among events, and they are used within all three functions `DivideRounds`, `DecideFame`, `FindOrder`.

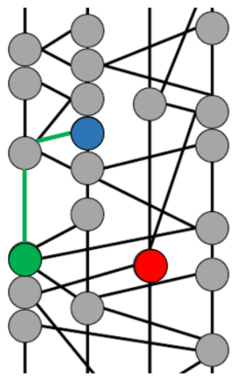


Fig. 2. An example of *see*. The blue event is able to *see* the green event, via the green path. However, blue event is not able to *see* the red event.

We will first explain what it means for an event to see another event. Basically, if an event p is able to *see* another event p, it means that there exists a downward path from p to q. In other words, p can reach q just by going through the either parent, iteratively. Since all nodes have a local copy of hashgraph, they can check this characteristic locally, without any requests to other nodes. This plays part in the "virtual voting" of the algorithm. An example of *see* is given in figure 2.

For an event p to *strongly see* another event q, the downward path from p to q must **go through** at least a *supermajority* of nodes. The supermajority is defined as at least $\frac{2n}{3}$ of nodes. This characteristic plays part in the Byzantine Fault Tolerant aspect of the Hashgraph. Note that in many cases,

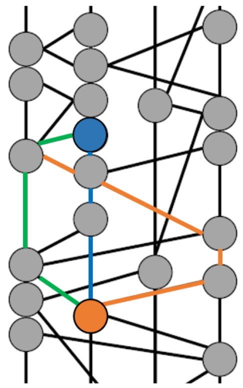


Fig. 3. An example of *strongly see*. The blue event can *strongly see* the orange event, through 3 nodes. Since $n = 4$ in this case and the supermajority is $\frac{2\times4}{3} = 2.66$ we would need to go through at least 3 nodes. Note that different paths are possible, but the existence of a valid combination would suffice.

fault tolerance is approached in a way that at most $\frac{n}{3}$ of the nodes may be malicious or faulty [13]. We show an example of *strongly see* in figure 3. In our implementation, our algorithm for *strongly see* is working in two parts: first, from the source event we do a breadth-first search to find the highest up ancestor of the source in other nodes, then we check if those parents and the source itself can *see* the target node. If the total number of successful *see*'s are supermajority, then we can say the source event can *strongly see* the target event.

D. `DecideFame`

Every witness in the hashgraph will eventually have a Boolean value regarding whether they are famous or not. This is decided by the `DecideFame` function, which we've first seen used in algorithm 1. The fame concept is used for the fault tolerance and consensus. Note that witness gave events a sense of ordering and importance, and fame can be thought of another layer of importance and power among other events. This would also imply that to decide on fame must not be exploitable, and as we would expect, this also requires a sense of consensus. In algorithm 3, we see how we calculate the fame.

Algorithm 3 `DecideFame` procedure

```
for each $x$ fame undecided from low to high do
    for each $y$ with round $> x.Round$ from low to high do
        $s \leftarrow$ set of witnesses in round $y.Round - 1$ that $y$ strongly sees
        $v \leftarrow$ the majority vote of witnesses in $s$
        $t \leftarrow$ number of events in $s$ who voted $v$
        if $t > \frac{2n}{3}$ then
            $x.Famous \leftarrow v$
            break out of the inner loop
        end if
    end for
end for
```

As we can see, the required fault tolerance is again obtained by the *strongly see* concept. We need a population of events to agree on this fame. The vote v we see here is to *see* in other words. If a witness event e is voting on a lower witness (in terms of round) w whether it should be famous or not, e checks if it can *see* w, and says "yes" if it can, "no" otherwise. We should further explain the actual rounds of witnesses that participate in the voting: let us say that we want to calculate whether a witness w is famous or not. A higher round witness will count the votes of the witnesses one round below it to decide if w is famous or not. When we say higher round, we mean at least 2 rounds bigger, because if the vote counter resides in round $r_w + 1$ then the voters would be in the same round with w. However, this would mean they would not be able to see w, thus vote "yes" at all. Therefore, in the original paper if the round is $r_w + 1$, which means the difference between r_w and the round of vote counter is 1, then the loop is continued. Another difference is that the votes are not stored in events here, which is an optimization that is done in the original paper rather than counting the votes again. However, this optimization would only be valid for the same set of witnesses in that round of voting, thus we do not store it.

We must also note that while `DivideRounds` was ran for one event on receipt, the `DecideFame` runs on all events that does not have their fame decided yet. This is because we require higher up round witnesses for this function to work, which would not work if we tried to decide the fame of a new event.

In the original paper by Leemon Baird [1], there is also a coin round in this function. If the difference between rounds of x and y is a multiple of c, where c is the coin round constant, then if $t \leq \frac{2n}{3}$, this y will vote with it's middle bit of signature rather than checking whether it can *see* x. In the paper it is shown that, with the coin rounds, ABFT is provided and the probability of arriving at a consesus is calculated as 1.

E. `FindOrder`

This is the function that makes the consensus happen. In algorithm 4 we show how we find such nodes. Note that we do not return anything, but instead update the slice itself from within the function.

The ordering is done with respect to round received values, and if there is a tie then with respect to consensus timestamps, if there is a tie again then we look at the middle bit of the whitened signature. The whitened signature is the signature XOR'ed with the signatures of all famous witnesses in the received round.

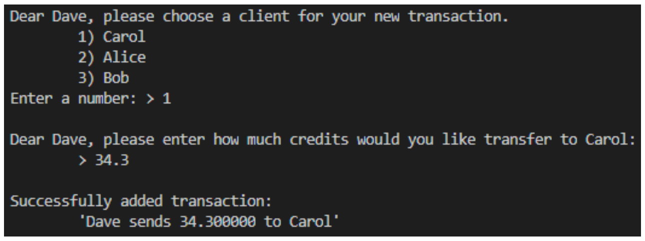
This concludes the implementation of our hashgraph. We shall now explain our user interface for the hashgraph visualization and for the transactions.

F. *Transaction Interface*

While the whole gossip protocol is taking place, another Go routine is expecting input from the command line. An example is given in figure 4.

Algorithm 4 `FindOrder` procedure

```
for each $x$ without a consensus do
    if $\exists r : \nexists$ witness $y$ without fame where $y.Round \leq r$
    $\land$ $x$ is ancestor of round $r$ famous witnesses then
        $x.RoundReceived \leftarrow r$
        $s \leftarrow$ set of events $z$ where $z$ is a self-ancestor of a round $r$ famous witness and $x$ is an ancestor of $z$ but not of $z$'s self-parent
        $x.ConsensusTimestamp \leftarrow$ median of timestamps in $s$
    end if
end for
order and update ConsensusEvents
```



```
Dear Dave, please choose a client for your new transaction.
        1) Carol
        2) Alice
        3) Bob
Enter a number: > 1

Dear Dave, please enter how much credits would you like transfer to Carol:
        > 34.3

Successfully added transaction:
        'Dave sends 34.300000 to Carol'
```

Fig. 4. Example menu with 4 nodes. In this case, nodes are Alice, Bob, Carol and Dave. The figure is from the perspective of Dave, where first chooses a node and then enters the amount for transaction.

The user is greeted with a menu, where they can choose another node. Then, they are asked to enter an amount to transfer to the selected node. What is meant by "successfully adding a transaction" is that the transaction will be added to the body of the next event created in this node. This event will be known by all nodes in a short matter of time, thanks to gossip protocol. The gossip is transparent to the user, so they will not know if their transactions are gossiped yet or not, however the protocol and it's randomness assures that everyone will be aware of that transaction, and the total ordering will determine the time and order of this transaction by the event it is stored within. When an event is sent, all the transactions that were loaded are stored there, and the transaction buffer is emptied for the next event and further transactions.

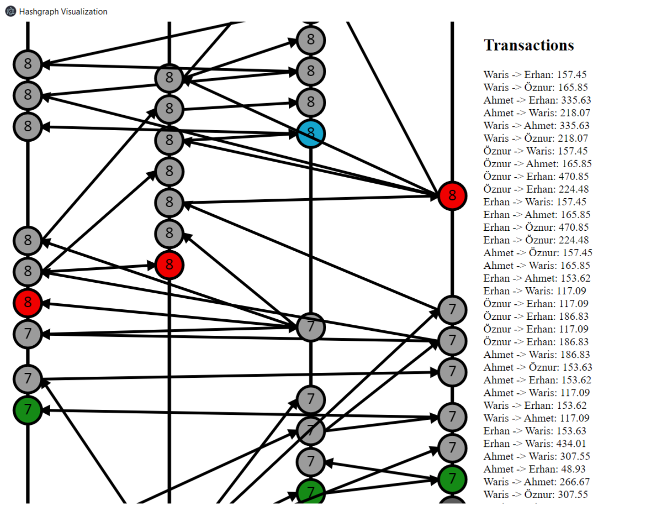
G. Hashgraph Visualization

In order to visualize our implementation, we created a cross-platform desktop application using Go and Electron [18]. Electron is based on Node.js and Chromium and lets you embed a web-application in a desktop application which can run on all major operating systems. Visualization application contains an instance of our *dledger* Go implementation, which is used for connecting to Hashgraph. Communication between hashgraph and our visualization frontend is handled by `go-astilectron` library. This library eases the communication between Electron applications and external Go scripts. We implemented another Go program which periodically checks the current state of Hashgraph and sends updates to the frontend.

In our visual Hashgraph, there are 8 possible colors for an event, they are split in 4 light colors, and 4 dark colors of the same hue. We will explain hue first:

- Gray: This is a normal event.
- Red: This is a witness without a decision regarding it's fame.
- Green: This is a famous witness.
- Blue: This is a not famous witness.

The light shade means that the event does not have a consensus on it yet, and the dark shade means that there has been a consensus on the event.



Transactions

Waris -> Erhan: 157.45
Waris -> Öznur: 165.85
Ahmet -> Erhan: 335.63
Ahmet -> Waris: 218.07
Waris -> Ahmet: 335.63
Waris -> Öznur: 218.07
Öznur -> Waris: 157.45
Öznur -> Ahmet: 165.85
Öznur -> Erhan: 470.85
Öznur -> Erhan: 224.48
Erhan -> Waris: 157.45
Erhan -> Ahmet: 165.85
Erhan -> Öznur: 470.85
Erhan -> Öznur: 224.48
Ahmet -> Öznur: 157.45
Ahmet -> Waris: 165.85
Erhan -> Ahmet: 153.62
Erhan -> Waris: 117.09
Öznur -> Erhan: 117.09
Öznur -> Erhan: 186.83
Öznur -> Erhan: 117.09
Öznur -> Erhan: 186.83
Ahmet -> Waris: 186.83
Ahmet -> Öznur: 153.63
Ahmet -> Erhan: 153.62
Ahmet -> Waris: 117.09
Waris -> Erhan: 153.62
Waris -> Ahmet: 117.09
Erhan -> Waris: 153.63
Erhan -> Waris: 434.01
Ahmet -> Waris: 307.55
Ahmet -> Erhan: 48.93
Waris -> Ahmet: 266.67
Waris -> Öznur: 307.55

Fig. 5. A screenshot from our Hashgraph Visualizer. Transactions were randomly generated for every event. There is a small delay between each gossip for the sake of visuals.

III. Results

We have used two metrics for our evaluations, seen from the papers [2], [14] and [15]:

1) Latency: The amount of time that passes between the creation of an event and it's time of consensus. Note that this is not `ConsensusTimestamp`, but rather the time it was assigned a consensus timestamp. This is simply measured by finding the duration between the creation time and time of consensus.
2) Transactions Per Second: This is also construed as throughput in the original paper. It is one of the main comparison metrics among distributed ledger systems. To calculate this, first we generate a constant amount of transactions per event, and then we look at how many gossip calls have been made from a node.

We tested using 4 AWS EC2 `t2.micro` nodes, as they are the only free ones available to us. They are very small, and in our evaluations the memory became a problem after a certain amount of runtime because each node holds many events in it's hashgraph, with many transactions with them. Combined with the algorithm that runs for every gossip in the node, and their own resource requirements, the `t2.micro` machines with 4 nodes in our implementation gave an *out of* *memory* error close to after 7000 events. We have collected the following results by choosing a specific checkpoint for evaluation: 1750 gossip calls from a node. Every node prints out several metrics for some amount of gossip calls, and we chose 1750 call mark for our evaluations. In figure 6 we can see the results.

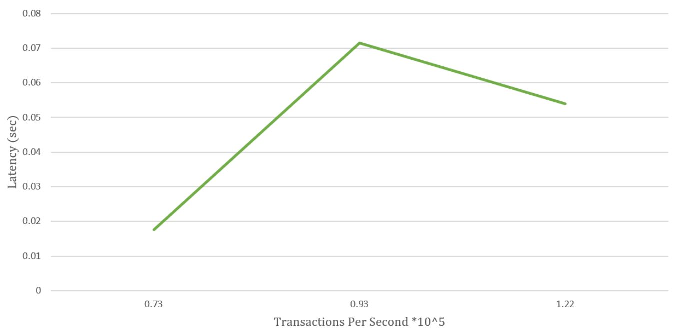


Fig. 6. Transactions per second versus Latency. From left to right, each event has 256, 512 and 1024 transactions in them. 4 `t2.micro` nodes in 1 region. The results are collected at the 1750th gossip call.

The values are at the same level of magnitude with the original paper [2], however the direction of the plot is different in ours. At this point, we should note that one of the main consideration for latency calculations is the encryption and decryption overheads for the transactions. However, in our implementation we did not do such encryptions or decryptions. We were mostly interested in the proof-of-concept, in other words, we wanted to see a working implementation with many events, and how they reach to consensus using Hashgraph algorithm, rather than all of the security considerations or attacks.

The latency is first calculated by every node locally, then they output that to the console, we take the average of all 4 for this report. A similar thing happens for the transaction per second calculation, every node locally calculates a *gossip per second* metric, where they basically divide the number of gossip calls they made to total runtime of the gossip loop. Since we used a constant amount of random transactions per event, and since every gossip call generates an event on the node that it is called upon, we can measure how many transactions are registered by these gossips by simply multiplying the *gossip per second* with the constant amount of randomly generated transactions.

Note that in figure 5 we said that there was a small delay between each gossip; in our evaluations there was no delay as it would degrade the overall performance for no reason.

Overall, the results are satisfyingly similar to that of the original paper in figure 2 of [2] for 4 nodes in 1 region, even with the implementation differences among our codes, as well as their environments, theirs being JAVA and ours being Go.

IV. Conclusions

We believe that Hashgraph is a powerful and promising algorithm. Even though our implementation lacks many considerations compared to the original SWIRLDS Hashgraph, we were able to observe its capabilities as a distributed ledger technology. The algorithm itself is ingenious, and as the author proves in it's original paper it is complete with proofs regarding correctness and security.

Our visual application provides a beautiful look over the progress of the algorithm and how the overall ledger system evolves, and these visualizations also match the figures in the technical report and paper of Hashgraph. Our evaluations also show that we have achieved satisfactory results in the same magnitudes with the original paper of Hashgraph [2].

As a final remark, we believe Hashgraph is a powerful contender for the novel ABFT distributed ledger algorithms, especially considering it's high throughput and low latency.

Our code is available on Github [19].

References

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

[2] L. Baird, A. Luykx, ”The Hashgraph Protocol: Efficient Asynchronous BFT for High-Throughput Distributed Ledgers”, *to appear in IEEE COINS 2020*

[3] A. Demers, D. Greene, C. Hauser, W. Wes, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry (1987-01-01). ”Epidemic Algorithms for Replicated Database Maintenance.” *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing. PODC '87.* New York, NY, USA: ACM. pp. 1–12. doi:10.1145/41840.41841

[4] S. Nakomoto, ”Bitcoin: A peer-to-peer electronic cash system. Whitepaper”, 2009.

[5] B. Cao et al., ”When Internet of Things Meets Blockchain: Challenges in Distributed Consensus”, *IEEE Network*, vol. 33, no. 6, pp. 133-139, Nov.-Dec. 2019, doi: 10.1109/MNET.2019.1900002.

[6] N. Živić, E. Kadušić and K. Kadušić, ”Directed Acyclic Graph as Hashgraph: an Alternative DLT to Blockchains and Tangles,” *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH)*, East Sarajevo, Bosnia and Herzegovina, 2020, pp. 1-4, doi: 10.1109/INFOTEH48170.2020.9066312.

[7] T. Lasy, ”From Hashgraph to a Family of Atomic Broadcast Algorithms”, *arXiv e-prints*, page arXiv:1907.02900, 2019

[8] G. Bansal and A. Bhatia, ”A Fast, Secure and Distributed Consensus Mechanism for Energy Trading Among Vehicles using Hashgraph,” *2020 International Conference on Information Networking (ICOIN)*, Barcelona, Spain, 2020, pp. 772-777, doi: 10.1109/ICOIN48656.2020.9016440.

[9] U. Timalsina and A. Wang, ”Incentivizing Services Sharing in IoT with OSGi and HashGraph,” *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, South Padre Island, TX, USA, 2019, pp. 48-52, doi: 10.1109/ICDIS.2019.00015.

[10] The Go programming language http://golang.org/

[11] J. James, D. Hawthorne, K. Duncan, A. St. Leger, J. Sagisi, M. Collins. ”An experimental framework for investigating hashgraph algorithm transaction speed”, *Proceedings of the 2nd Workshop on Blockchain-enabled Networked Sensor (BlockSys'19)*. Association for Computing Machinery, New York, NY, USA, 15–21. 2019 DOI:https://doi.org/10.1145/3362744.3363342

[12] SWIRLDS Hashgraph: Detailed Examples https://www.swirlds.com/downloads/SWIRLDS-TR-2016-02.pdf, *accessed 06/02/2020*

[13] M. Pease, R. Shostak, and L. Lamport. 1980. ”Reaching Agreement in the Presence of Faults” *J. ACM 27, 2 (April 1980)*, 228–234. DOI:https://doi.org/10.1145/322186.322188

[14] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. 2016. ”The Honey Badger of BFT Protocols.” *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16).* Association for Computing Machinery, New York, NY, USA, 31–42. DOI:https://doi.org/10.1145/2976749.2978399

[15] S. Duan, M. K. Reiter, and H. Zhang. 2018. "BEAT: Asynchronous BFT Made Practical." *In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* Association for Computing Machinery, New York, NY, USA, 2028–2041. DOI:https://doi.org/10.1145/3243734.3243812

[16] A. Shareef and C. P. Rangan. 2008. "On optimal probabilistic asynchronous Byzantine agreement." *In Proceedings of the 9th international conference on Distributed computing and networking (ICDCN'08).* Springer-Verlag, Berlin, Heidelberg, 86–98.

[17] C. Cachin, K. Kursawe, and V. Shoup. 2000. "Random oracles in constantipole: practical asynchronous Byzantine agreement using cryptography (extended abstract)." *In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC '00).* Association for Computing Machinery, New York, NY, USA, 123–132. DOI:https://doi.org/10.1145/343477.343531

[18] Electron: A Framework for building cross-platform desktop apps using JavaScript, HTML, and CSS https://www.electronjs.org/

[19] Erhan Tezcan, & Ahmet Uysal. (2020, June 6). ahmetuysal/hashgraph: Initial release (Version v0.1.0). Zenodo. http://doi.org/10.5281/zenodo.3882970