Hashgraph-inspired consensus flavors (eng)

Here are some thoughts that emerged after watching and (finally) comprehending the following video:

Dr. Leemon Baird: How Hashgraph Works - A Simple Explanation w/ Pictures

https://www.youtube.com/watch?v=wgwYU1Zr9Tg

Hashgraph is a rather complex algorithm, but many of its moving parts, if not outright replaceable, can certainly be tweaked for the following reasons:

1. The algorithm does not enforce any specific low-level implementation details. As long as the fundamental principles outlined by the algorithm (or their equivalents) are adhered to, there are numerous ways it can be implemented, offering a high degree of flexibility.
2. Performance improvements: Simplifying the traversal of event sequences, optimizing the filtering of received information, and similar adjustments can enhance the system's efficiency.
3. Simplification of structures and utility algorithms when using the algorithm in a trusted environment. For instance, if we are creating a clustered database where all nodes are under our control, certain complexities can be reduced.
4. The ability to observe system behavior under different protocol settings, allowing for experimentation and optimization.

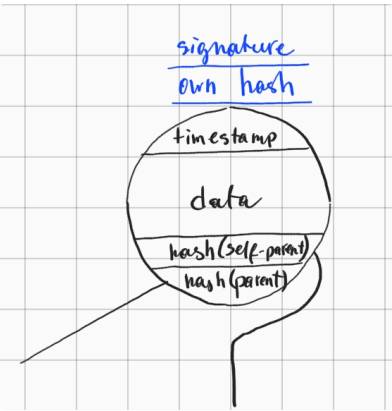
Events can be divided into two categories by their structure:

1. Events containing data, such as transactions in the case of a blockchain. We will refer to these, for simplicity, as **data/new**.
2. Events without data, containing only information about the propagation of gossip. We will refer to these as **gossip/reference**.

Also, conceptually, the data itself does not necessarily need to be embedded within the event structure. Instead, these could be references, such as object IDs stored in external storage, which would reduce the network traffic involved in gossip propagation.

My Perspective on Events

Let's take a closer look at the event structure they present in the video:



What is shown in black exists in the drawing from the video, while what is shown in blue is discussed but they seem not to consider it as part of the event itself, for reasons that remain unclear.

If we carefully consider what **self-parent-hash**, **other-parent-hash**, and **own-hash** represent, it becomes clear that these are nothing more than unique identifiers of an event and references to previous events by their respective identifiers. This approach has its advantages, as the ID is tightly coupled with the data, effectively functioning as a checksum (as any hash would). However, it also comes with some

drawbacks—for instance, the hashes are not monotonically increasing and do not contain additional metadata, such as a direct reference to a parent.

As a result, to determine where a specific event originated from, lookups will be necessary. Furthermore, to retrieve the sequence of events from a particular node, graph traversal will be required. This is because timestamps may coincide, and, if I understand correctly, nothing prevents timestamps from jumping backward or forward, even on a single node. Therefore, an ordered sequence of events is not guaranteed to form naturally based solely on timestamps, so we can't range-index.

Additionally, let's not forget that, in theory, hashes can experience collisions, as is the case with MD5.

From real life: a former colleague of mine once developed a system where he used MD5 hashes as primary keys, and they did eventually collide in production. This had a noticeable impact on his mood at the time. To resolve the issue, he replaced the single MD5 hash with a combination of hashes, like MD5+SHA1 or something similar. However, when I suggested using proper unique IDs instead, he looked at me like I was an idiot, made some dismissive remark about probabilities, and shut down the conversation. To this day, I'm not sure what I said wrong.

That being said, it's only fair to note that after the fix, his system no longer experienced collisions. Similarly, it's possible that Hashgraph at Hedera might never encounter collisions. The possibility of avoiding collisions does, of course, exist.

But let's move on to a practical question: is a hash truly irreplaceable in this case? Could it be substituted by a pair like **ID/checksum**? This brings up the idea that in an untrusted environment, like a blockchain, where nodes might be expected to send malicious data with the intent to break the distributed system, there could be advantages to having our ID tightly coupled with the data. It would be significantly harder for an attacker to generate maliciously formed or duplicated hashes compared to generating IDs.

At the same time, in a trusted environment like a **private database cluster**, optimizations can be made by avoiding the computational overhead of hash calculations. Instead, we could embed useful properties in our IDs, such as referencing the node that created the event, ensuring monotonically increasing values, and other such metadata that can enhance system performance with acceptable relaxation of security given the protected environment.

Additionally, for optimizing the performance of a trusted local cluster, we could even forgo the need for full cryptographic signatures. For instance, if we embed the creator node directly into the ID, this would also save computational resources.

I should note, as I've attempted to illustrate above (to the best of my ability), that the Hashgraph algorithm (at least as far as my theory goes) can be implemented without the use of hash functions. Let's count that as a pun!

If we do decide to replace the hash with an ID, we could re-label things in a way that's more intuitive (at least according to my perspective).

`own-hash`	`id` or `event_id`
`self-parent-hash`	`previous_event_id` - If we imagine the events originating from a single node as a linked list, the structure of this list is formed precisely by this field.
`other-parent-hash`	`@Nullable outer_event_id` - A reference to the external originating event, belonging to a sequence of IDs that exists on another node.

On Parent References

It is evident that, aside from the very first event (which marks the creation of a node), all subsequent events for that node will always have a **self-parent**. However, it's less clear whether there can be situations where the **other-parent** is NULL. If we look at their video, in their graphical representation, there are no nodes that only have a **self-parent** while lacking an **other-parent**. I'm unsure why this is the case. Adding to the confusion is the fact that it's unclear from their diagram which events contain data and which do not, but presumably, some of the events should probably contain data.

From my perspective, in this algorithm, any event classified as **data/new** will always have only a **self-parent** and will never have an **other-parent**. It's possible that their implementation operates on different assumptions.

My take on this is, for example, why should I be forced to wait for some external gossip event to create my own data? Even if gossip occurs frequently and regularly, why integrate it into the data creation process at all? In my view, it makes more sense to simply create the data whenever it's needed, and then propagate the gossip to others. This approach is clearer, simpler, and faster.

Monotonically Increasing IDs, Group Versions, and Their Role in Optimizing Graph Updates During Gossip Exchanges

If we replace hashes with monotonically increasing IDs, and considering that we are dealing with a group of sources (in this case, nodes), each of which generates its own sequence of objects/events identified by monotonically growing keys, this opens up the possibility of applying the concept of **group versions** from my earlier work, **VUnion**.

By using group versions, we can optimize the retrieval of subgraphs during gossip exchanges. Specifically, we can extract the "tails" of event sequences from each source (node), starting from the version present in the group version of the gossip partner. If we store these sequences locally in an ordered format, as in **VUnion**, we can quickly compute and send the "tails" of these sequences starting from any given version. In this way, the group version acts as an identifier of the overall state of the local graph representation.

For more details on group versions, refer to the **VUnion** documentation.

This approach enables efficient updates and minimizes redundant data transfer during gossip by leveraging the ordered nature of monotonically increasing IDs and the group version system.

Transiency of Event graphs for applications in Databases and Key-Value Storages

For Hedera's blockchain purposes, it is likely that they need to store the entire gossip graph in its original form, for reasons tied to trust, similar to why all blockchain systems, starting from Bitcoin, persist every record exactly as it was logged. (*Disclaimer: This is an assumption about Hedera, and I could be mistaken, but the core idea below remains unaffected*).

However, when applying Hashgraph-like algorithms in traditional databases, it's unnecessary to persist the entire gossip graph. In such cases, the key is to store only the **result** of the gossip, which is the unified sequence of data objects from the respective **data events**. Once all nodes agree on a full sequence for a given round, the associated gossip can be discarded. The result would resemble a **transaction log**, consisting of a sequence of the contents of data events.

In such scenarios where **data events** are more valuable than **gossip events**, one optimization for gossip using the **group version** concept could involve binding the group version to **decided** and **undecided** data events, rather than gossip events. This way, when a gossip round is initiated, unnecessary duplication of new events can be avoided, as the group version would clearly indicate that there are no new undecided data events to be shared through gossip. In such cases, gossip wouldn't need to be initiated—there would simply be nothing to discuss.

For a **data storage node** recovering from a crash or downtime, the recovery strategy could involve pulling the tail of the **decided data events** (essentially a transaction log) and then rejoining the ongoing gossip exchange. This approach would enable efficient state restoration without the overhead of reprocessing irrelevant gossip events.

Additional Optimizations in Trusted Clusters

In the context of **trusted clusters**, several potential optimizations can be made:

1. **Replacing XOR Hashing for Event Sequence Resolution**: In Hashgraph, techniques like XOR-ing a hash with a commonly known random value and using specific bits to resolve event sequence collisions can be simplified in a trusted environment. Instead, we can maintain a **circular buffer** of nodes that determines event priority. In this system, each new round moves the head of the buffer to the tail, ensuring balance and a fair distribution of priority. This avoids the complexity of using random bits or hashes for sequence resolution and simplifies event ordering.
2. **Replacing Supermajority with Quorum**: In trusted clusters, we can likely replace any instance of using a **supermajority** with a **quorum**-based system. A quorum, requiring a smaller consensus threshold, would likely be sufficient in a trusted environment where nodes are assumed to act in good faith. Intuitively, this should streamline the process and reduce communication overhead, but I would need to back this up with analysis to fully justify the change. For now, it's more of a gut feeling that a quorum would be enough in such cases.

These adjustments reflect the different assumptions in a trusted environment, where concerns over malicious nodes are minimized, allowing for performance-focused tweaks that simplify the algorithm without compromising correctness.