Improvements to gossip exchanges with monotonically increasing versions

- We assume monotonically increasing version from each node, similarly to VUnion, and version exchange in the same manner.

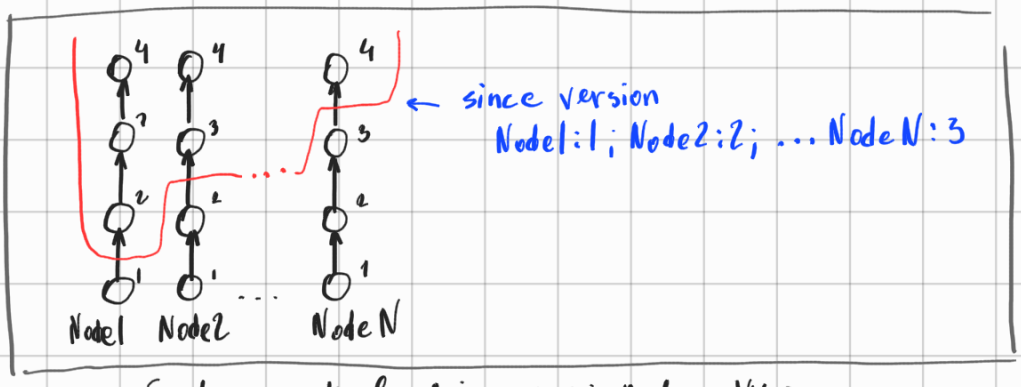

fig 1: concept of "since version" from VUnion

- In order to efficiently manage gossip exchanges, we maintain multiple versions on each Node

1) gossip version - classical high WM version, like in VUnion, considers all events
2) data version - version that only considers data events, decided and undecided
3) decided data version - version that only considers decided data events

- We use gossip version to exchange events efficiently based on "since version" idea that we borrow from VUnion
- We use data version and decided data version to minimize the amount of gossip created.

Thus, if incoming data version is <= own decided data version, by definition there is nothing meaningful to gossip about.

However, can we further minimize the communication? How can we ensure that redundant exchanges are completely eliminated?

Let's think about what is important to communicate apart from data itself? The only other important communication other than ourselves obtaining gossip about new data is when other nodes are obtaining gossip about new data. We want to get the new data itself, and see how it spreads to be able to meaningfully decide that everybody has it.

In order to keep that information efficiently, we can maintain apart from own data version, data versions from all other nodes.

a.k.a. multi data version

Let's say I'm Node 3 out of 4 Nodes in the cluster.

	Node 1	Node 2	Node 3	Node 4
Node 1	1	2	0	3
Node 2	0	1	0	3
Node 3	1	2	3	4
Node 4	1	1	2	2

While being Node 3, this is what I know Node 1's data version is, as far as the gossip goes.

I'm Node 3, this is my data version. It's the latest data I've got from gossiping

Here we also note, that our own data version is always the highest in this structure, i.e. no single component of any outer data version can be > than the corresponding component in our version.

Fig 2. multi-data version

Effects on propagation

We're not immediately sure that this can stack healthily with original hashgraph decision strategy. Let me remind you that it takes several rounds of gossip to finalize decisions: event's own round, Event's Witness round, Voters round, and then the beginning of Vote Counters round.

In a situation when we reject redundant gossip, and don't have constant data pressure, those rounds in theory may never come to be. Imagine edge case where we have just 1 data event submitted - the growth of Hashgraph will halt before the rounds required to decide on it will be finalized. For that reason it's possible to state, that redundant gossip is an important part of the original Hashgraph algorithm, that can't be altered without undesirable side-effects.

However, the proposed optimization might be valuable enough to revise the decision-making process to guarantee resolution of single or rare Data Events. Unless we alter the algorithm, a simple practical approach could be to spawn dummy events / force gossip under certain low traffic conditions.

Limited optimization compatible with original Hashgraph decision-making

As it was mentioned above, we can relax the restriction on redundant gossip by comparing incoming data version with own decided data version to allow all gossip until there are no undecided data events.

This, however, will produce the same amount of gossip as the original unrestricted approach under constant data load, so the optimisation in regard to reducing the amount of gossip generated will only apply in low traffic / low load situations and would therefore be superficial.

1) To update gossip version
   - find new top gossip graph version,
     combining top gossip versions for each Node

2) To update multi-version / cluster version
   - find new top data graph version,
     combining top data versions for each Node
   - traverse graph to find assumed / virtual versions for each new top data events (BFS?)
     • if reached the end of graph - version didn't change

3) Round commensing
   - round is commenced N>5 rounds after last event was decided (5)
   - once a round is commenced, we remove it from RAM
   - round can be partially recreated in split brain situation, but it will commence all the same - since its events will be decided at a later round in this case

4) Startup flow
   - We call gossip with our version
     • If our gossip version is behind we're asked to load transaction log first. After that, we use transaction log data version as our gossip version to join the gossip.