A GOSSIPING

The gossiping module strengthens system robustness by further disseminating committed transactions to the network. This module does not affect the correctness of consensus and can be applied based on preferences. When gossiping is enabled, after the consensus module commits a transaction, the proposer sends a `Gossip` message piggybacking the transaction to other connected vehicles, which will keep propagating this message to their connected vehicles. Therefore, the transmission forms a *propagation tree* where the root node is the *proposer* and other nodes are *propagators*.

Each `Gossip` message has a *lifetime* (denoted by λ) that determines the number of propagators it traverses in a transmission link; i.e., the height of the propagation tree. For example, in Figure 11, V_1 is the proposer and has direct connections with V_2, V_3, and V_4. The gossip message has a lifetime of 2, and its propagation stops when a path has included 2 propagators; e.g., path <V_1, V_2, V_5>. Specifically, we describe the gossiping workflow as follows.

Proposer: The proposer (V_p) starts to disseminate committed transactions by taking the following actions.

(1) V_p creates a lifetime (e.g., $\lambda = 3$) for a `Commit` message and a set $\mathcal{G}$ for recording the traverse information of the message's transmission path.

(2) V_p hashes the `Commit` message and signs the combination of the hash result (h_c) and λ to obtain a signature σ_{V_p}. Then, it adds a *traverse entry* containing λ, σ_{V_p}, and C_{V_p} to $\mathcal{G}$; i.e., $\mathcal{G}.add([\lambda, \sigma_{V_p}, C_{V_p}])$, where C_{V_p} is V_p's address and public key.

(3) V_p sends $\langle$`Gossip`, $\langle$`Commit`$\rangle$, $h_c, tx, \mathcal{G}\rangle$ to connected vehicles that are not included in the consensus process of the `Commit` message and then creates a list for receiving `ack` messages from propagators.

Propagator: A propagator (V_i) stores a valid `Gossip` message and may further disseminate the message if there is lifetime remaining.

(1) V_i verifies a received `Gossip` message by four criteria: ① it has not previously received this message; ② signatures in $\mathcal{G}$ are valid; ③ λ in $\mathcal{G}$ is strictly monotonically decreasing; and ④ the message still has remaining lifetime; i.e., $\lambda_{min} = min\{\mathcal{G}.\lambda\} > 0$. If the verification succeeds, V_i sends an `ack` message to this message's proposer and the pivot validator, registering itself on the propagator list.

(2) V_i decrements this gossip message's lifetime; i.e., λ_{new}= $\max\{\lambda_{min} - 1, 0\}$. If $\lambda_{new} > 0$, then V_i prepares to further disseminate the message. It signs the combination of h_c and λ_{new} to obtain a signature σ_{V_i} and adds a new traverse entry into $\mathcal{G}$; i.e., $\mathcal{G}.add([\lambda_{new}, \sigma_{V_i}, V_i])$.

(3) V_i sends $\langle$`Gossip`, $\langle$`Commit`$\rangle$, $h_c, tx, \mathcal{G}\rangle$ to other connected vehicles, excluding those it receives this `Gossip` message from (in (1)).

The gossip module in V-Guard is an additional mechanism that provides an additional layer of redundancy. Since each propagator is registered to the proposer, the proposer can connect to propagators and read its own portion of data. This is especially important when consensus is conducted in small booth sizes, as the failure of vehicles can significantly affect the system's overall resilience. Although gossiping does not guarantee the delivery of all transactions to all participants in the network, it can help to increase the probability of transaction delivery even in the face of network partitions or failures, strengthening the system's robustness without interfering with the core consensus process.

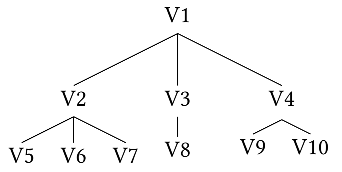

Figure 11: A propagation tree of a gossip message with $\lambda = 2$. V_1 is the proposer and others are propagators.

B STORAGE

V-Guard stores committed data batches in a *storage master*. The storage master creates storage master instances (SMIs) for transactions from different vehicles, as a vehicle may operate in different roles in different booths (i.e., $\gamma > 1$). For example, in Figure 12, the storage master has three instance zones containing proposer, validator, and gossiper SMIs (if the gossiping module is applied). A vehicle's storage master has at most one proposer SMI because it is the only proposer of its own V-Guard instance. When the vehicle joins other booths (operating as a validator), its storage master creates validator SMIs. The number of proposer and validator SMIs equals the vehicle's catering factor (γ). When gossiping is enabled, a vehicle may receive gossiping messages, and its storage master creates gossiper SMIs storing consensus results disseminated from other vehicles.

Each storage master instance has two layers: temporary and permanent, with the purpose of maximizing the usage of vehicles' limited storage space. Unlike the blockchain platforms working on servers (e.g., HyperLedger Fabric [8], CCF [108], and Diem [40]), V-Guard operates among vehicles, which usually have only limited storage capability.

The temporary storage temporarily stores transactions based on a predefined policy. V-Guard's implementation uses a timing policy that defines a time period (denoted by τ) that the storage master stores a transaction (e.g., τ = 24 hours). A transaction is registered in the temporary storage by default and is deleted after τ time if the user issues no further command. In addition, temporarily stored transactions can be moved to permanent storage and kept permanently per user request. This option hands over the control of storage to users. For example, when users suspect malfunctions of their vehicles, they may want to keep related transactions as evidence and move them to permanent storage. We introduce four SMI APIs for the layered design.

- `sm.RegisterToTemp(&tx, time.Now())` registers a transaction to the temporary storage layer.
- `sm.CleanUpTemp(time.Now())` is a daemon process that calculates the remaining time for transactions in the temporary storage and deletes expired ones.
- `sm.MoveToPerm(&tx)` is called by users, moving selected transactions from temporary to permanent storage.
- `sm.DeletePerm(&tx)` is called by users. A user may delete a permanently stored transaction after it has served the user's purpose.

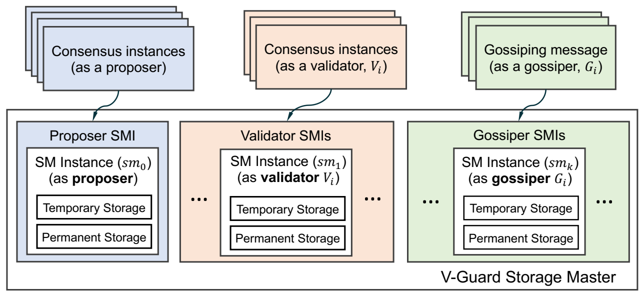

Figure 12: Storage master instances (SMIs). A vehicle operates a proposer SMI when it conducts consensus for itself, validator SMIs when it participates in other vehicles' consensus, and gossiper SMIs when it enables the gossip module.

The policy of temporary storage can be implemented differently. For example, a policy that defines a fixed size of storage space may apply. When the temporary storage exceeds the predefined size, it clears up old transactions in a first-in-first-out (FIFO) manner.

The policy-based storage module can significantly reduce the use of storage space as vehicles often have limited storage capability. Note that the pivot validator can permanently store all data as automobile manufacturers often operate on their cloud platforms with scalable storage devices.