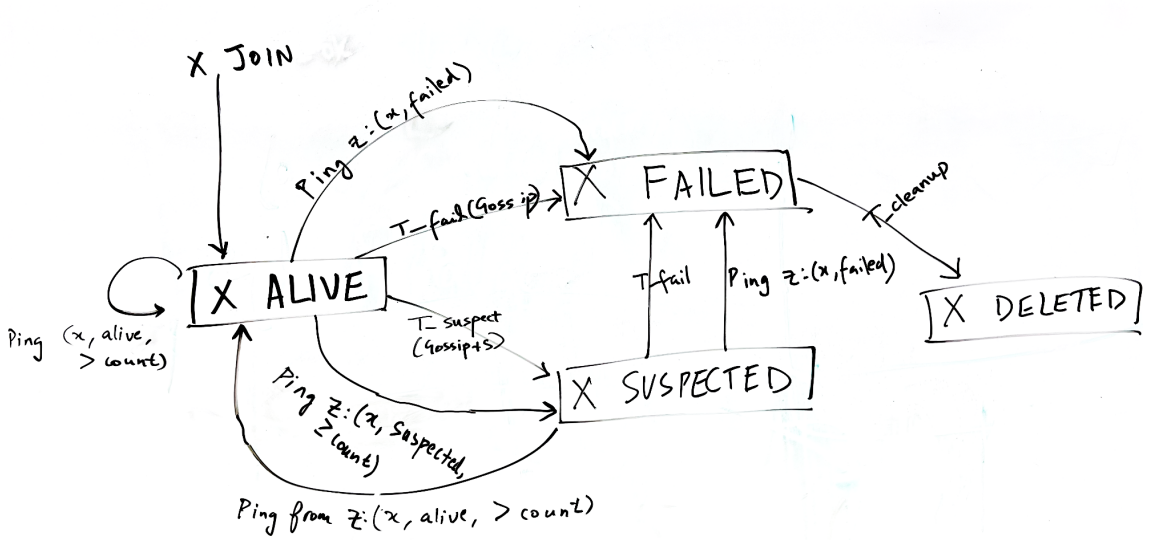
Failure Detector Architecture

Our server is written in Go. Each node runs a UDP server on a specified port, containing multiple Go-routines which are synchronized by the main thread using Go channels. Additionally, each node maintains a full membership list of all the nodes. Within each server, a receiver thread listens for gossip messages from peers and a sender periodically sends gossip messages to “b” (branching factor) randomly selected peers. The gossip frequency is determined by a constant “t_gossip”. A node always increments its own “heartbeat counter” in each round of gossip. A timer thread is launched for each peer, which is either restarted by the server after receiving an update, or it sends a timeout message to the server.

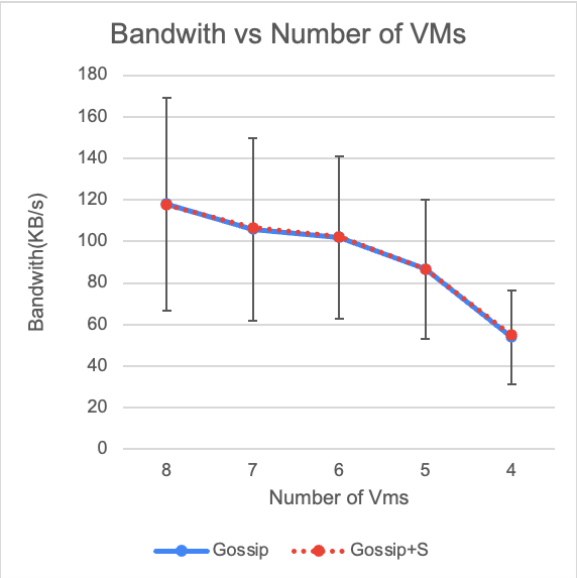
Algorithm

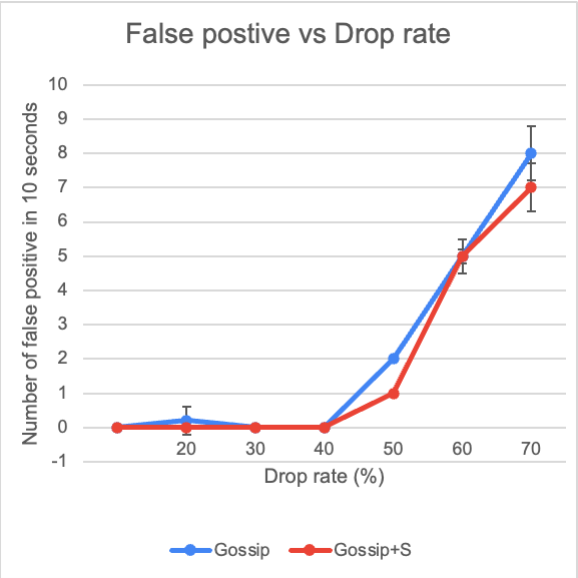
Each node attempts to join the system with a unique ID by sending a JOIN request to the introducer. The introducer responds with a JOIN_OK message plus the current membership list. We assign the first VM in our cluster to act as the introducer for all the nodes. A node can be in one of three states: Alive, Suspected or Failed. A node is initially Alive. After a timeout of “t_suspect”, the node moves into the Suspected stage. After another timeout of “t_fail”, the node moves into the Failed stage. At this stage, the node must rejoin the system to continue. We still keep the entry for this failed node in our membership list for “t_cleanup” time, so that the system has time to propagate the failure information and avoid ghost entries. In the gossip protocol, the node moves directly from Alive to Failed unlike in the gossip+s protocol. A suspected node can come back to life if it successfully pings a peer before it is marked as failed by some node.

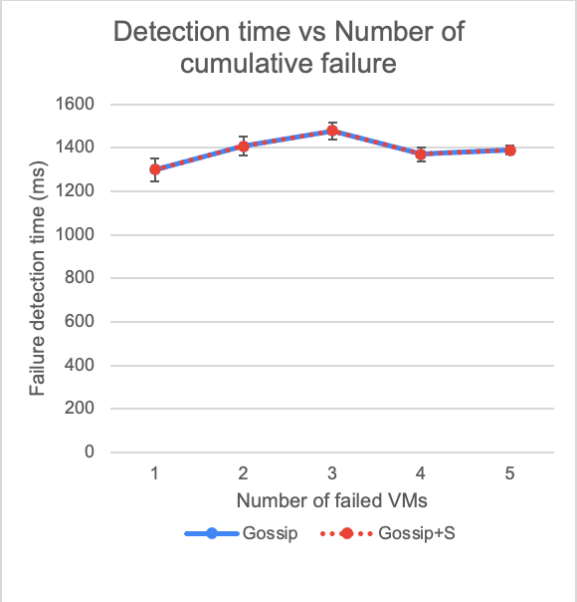


Experiment & Discussion

We tunned the parameters of both protocol so that both protocol will detect the failed node within 1.5 seconds with an additional 3 seconds for the whole system to remove the failed node from the membership list. For part2, we set the maximum bandwidth (inbound + outbound) to be 120KB/s. We also increase the T_suspect and T_timeout for Gossip + S. Since we used same T_gossip and branch factor for both part1 and part2, part2-c plot will look very similar to part1-a so we don’t re-include that in our report.







In part1, with the fixed detection time cap, the bandwidth is negatively related to the number of online VMs. The bandwidth between two protocols are similar because our Gossip+S implementation does not send extra information to achieve suspect mechnaism. The false positive rate increase significantly after 50% drop rate because we ensure 3 rounds gossip with branch factor of 4 before a node is declared failed. Therefore when drop rate is low, it is likely that the heartbeat will be gossiped in time. The number of failed node does not affect the detection time significantly due to our low T_gossip and high branch factor.

In part2, with bandwidth cap only, Gossip+S can have longer T_timeout so that it's more likely to recover a suspected node from false detection. However, the trade-off to this is the failure detection time is larger than Gossip protocol. The number of simultaneous failure does not affect the detection time significantly due to our low T_gossip and high branch factor.