CS425, Distributed Systems: Fall 2023
Machine Programming 2 - Distributed Group Membership

Group No: 58 **Names (netId):** Risheek Rakshit S K(rrs7), Siddharth Lal (sl203)
Repository link: *https://gitlab.engr.illinois.edu/sl203/mp2-cs425-group58.git*

The distributed group membership is built using Golang. The first machine to be started is the introducer and we assume that it never crashes. Whenever a new node is started, the node id is set to the format <machine-name>#<timestamp> as we are implementing fail-stop model. Once the node is up, it sends a introduce message to the Introducer and the introducer on receiving this message, updates its own membership list with the entry of node that sent the message and responds to the node with a subset of its membership list. The subset consists of the requesting node's entry and last three member entries of the introducer. When a node receives the response from the introducer, it simultaneously starts sending gossips and starts listening for gossips. When a node receives a gossip it checks for the increased heartbeats of its own member table entries and updates the heartbeat and the local time at which it was updated. A function continuously checks for the entries which has crossed the Tfail seconds without any updates to its heartbeat and marks the entry for failure. If the function finds a member with its heartbeat not updated for another Tcleanup seconds, its entry is removed from the table. When the node is about to gossip it updates its own heartbeat and sends the membership list across to *b* other nodes picked at random. The membership list sent across does not carry the entries for the nodes which it marked as Failed in Tfail sec. This way the bandwidth in the network is optimised. Each Node has its own version number which is used to avoid continuous switching between GOSSIP and GOSSIP + SUSPICION modes. When we command a node to switch the mode, it increases its own version number. When its gossip is received by other nodes, they check whether the change in mode is received from a node with higher version number. In that case they also change their version number and starts gossiping. So after log(N) time the entire network is switched to the desired mode. When run on GOSSIP+SUSPICION mode, the node marks a member as suspected when Tfail time has crossed since the last update. When a node receives a gossip which states a member to be suspected with the same incarnation number as in its own membership list, then it marks the node as suspected. Eventually within log(N) time the node which was suspected, if alive will know that it is being suspected. It increases its incarnation number and sends gossip. So when others which have marked a node as suspected, receives a entry of the same node with a higher incarnation number, the suspicion for the node is turned off. It also ideally takes log(N) time to be communicated. The suspected node increases its incarnation number only when its current incarnation number is suspected. If higher incarnation number is not received within Tfail + Tcleanup then the node is marked as failed and it can only receive gossip from that node only when it starts over again. So the slowest node will eventually be cut off from the network.

Part1: We have set the number of nodes to gossip in each round *b* = 3 and time period for gossip Tgossip = 1 sec, and since we have 10 nodes in the environment, we set the Tfail = 3 sec and Tcleanup = 2 . These values are chosen because log(10) on base 3 = 2.0959. Since Tfail + Tcleanup = 5 sec and we gossip for every 1 second, we can be sure that the failure is detected at least in one node in a span of 5 seconds.

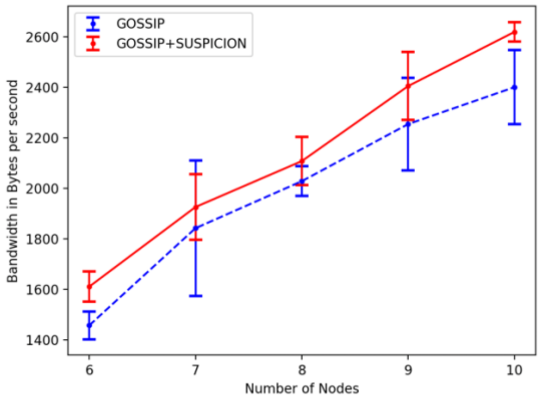

Figure 1.1: Average Bandwidth.

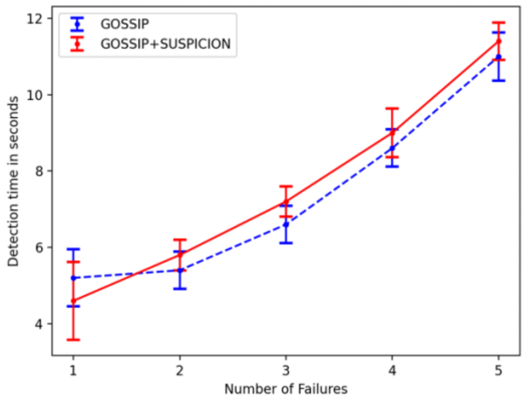

Figure 1.2: Average Detection in Seconds

From the Figure 1.1 we can see that as the number of nodes increases the bandwidth in terms of bytes per seconds increases for both GOSSIP and GOSSIP+SUSPICION modes as the number of nodes increase. And we also observe that GOSSIP+SUSPICION takes more bandwidth on average, as the extra information of the node which is suspected is also sent across the network. Figure 1.2 shows that the average detection time for both the modes remain almost the same and stays within the 5% margin even when it deviates. As expected, the time increases when detecting a higher number of node failures. Figure 1.3 explains the trend in false positives as the packet drop rates increases. As the packets are dropped, the gossips aren't reached as expected in a ideal scenario and hence the nodes mark more nodes as suspected even when they are up and active, and even remove them from membership list.

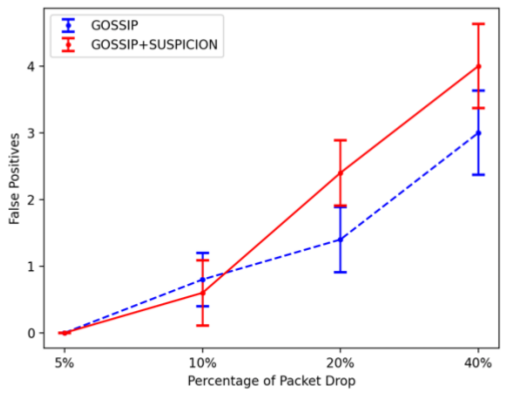

Figure 1.3

Part 2: To maintain the base bandwith we have slightly increased the gossip period and observed that Tgossip = 1.2 seconds matches the needs. We also set the Tfail = 4 seconds and Tcleanup = 3 seconds. This is done only for GOSSIP+SUSPICION, to match the bandwidth of GOSSIP within 5% error. GOSSIP still has Tfail as 3 sec and Tcleanup as 2 sec. From the figure 2.1 we can see that since we had to increase Tfail for GOSSIP+SUSPICION, we can see an increase in the detection time v/s GOSSIP. For figure 2.2, we measured the bandwidths, and as evident from the figures, we can see that they are almost the same. Since we increased Tgossip for GOSSIP+SUSPICION, we essentially made sure that per seconds the number of messages are reduced to compensate for the increase in information being sent. In Figure 2.3, We see the false positive rate v/s number of packets dropped, which is again in expected lines. Here, GOSSIP+SUSPICION we note that the number of fall positive rates is less than GOSSIP.

(For above experiments, we measured bandwidth using IFTOP and to implement packet drops we randomly selected an integer from 1-100 and if it is inside the drop% we drop the packet)

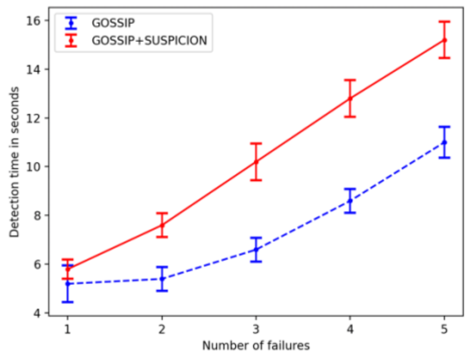

Figure 2. 1

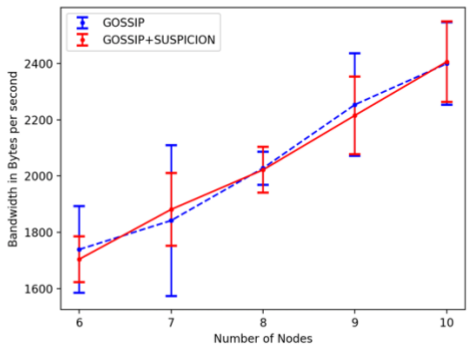

Figure 2.2

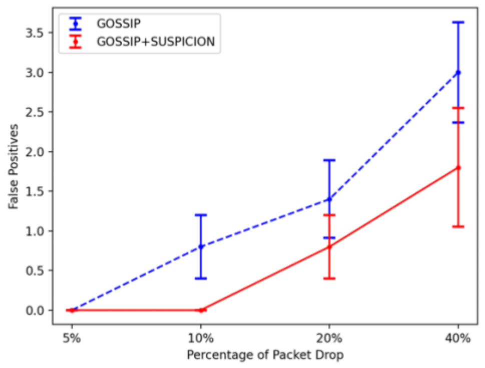

Figure 2.3