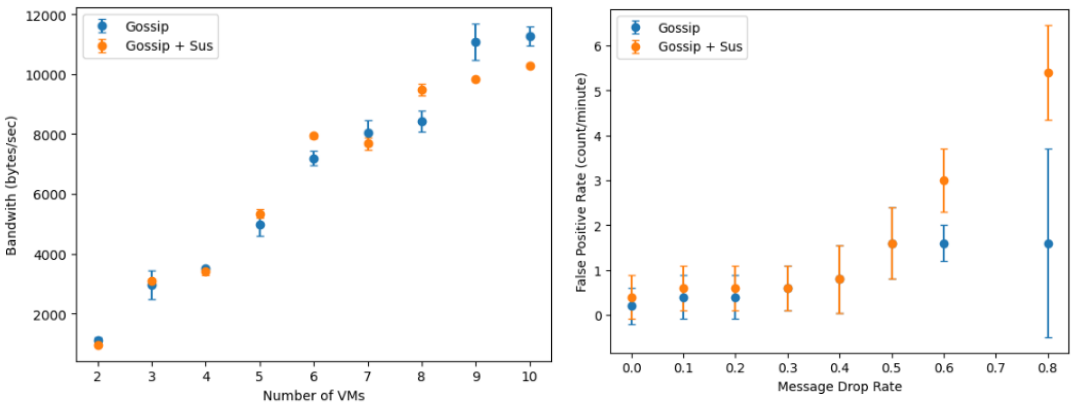
Summary:
We were able to implement a distributed group membership system using Go. The system supports the following: Logging when a new member joins or leaves the group, changing between pure gossip and gossip with suspicion enabled, listing current membership list, leaving and rejoining.

Design Choice:
There are two major components to the system, the regular node and an introducer. In our case the introducer has the same functionality as a regular node with the exception that it is always running a different listening port using TCP that handles the initial joining of a new node by providing it with its current membership list. Apart from this distinction, every node consists of the following go routines: a user input routine that constantly checks for user input and performs accordingly, a listener that constantly listens for incoming gossip and updates the local clientlist accordingly, a sender that regularly gossips to members in the membership list about it's current updated list, and another routine that is in charge of timeouts and cleanups.

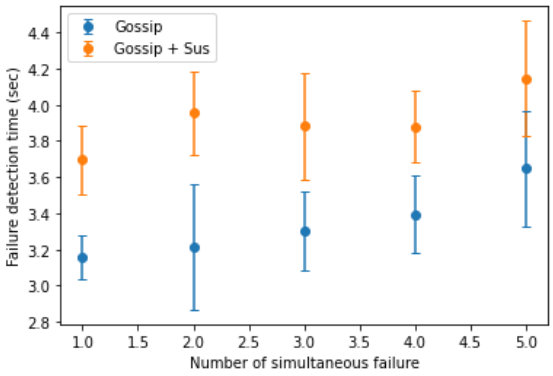
To guarantee that our design met the 5 second completeness, for every iteration we shuffle the clientlist, gossip to three clients, sleep for 500 millisecond, and repeat until we go through the current clientlist back to the reshuffle process. We also experimented and setup the timeouts accordingly so that changes are reflected among the entire member nodes in under 5 seconds

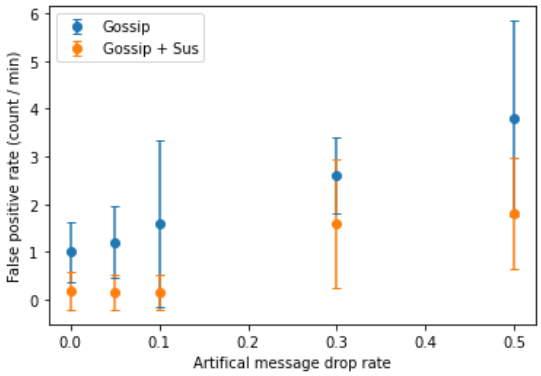
Plots:
1.

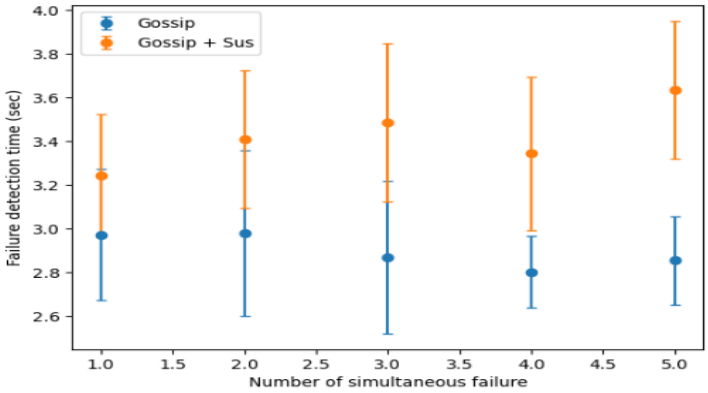


a. Detection times as a function of number of failures

2. These plots are generated with 7 running VMs, with a fixed cap of 8000 bytes/sec per machine as average base bandwidth (sum of outgoing & incoming).







a. Failure detection time (sec) vs. Number of simultaneous failures: The increase in the number of simultaneous failures increases the failure detection time. This is reasonable as the more nodes stop responding, the more a node needs to rely on its own timeout to detect failures instead of by receiving frequent gossip.
b. False positive rate (count / min) vs Artificial message drop rate: The increase in message drop rate increases false positive rate. This is reasonable as timeouts are more likely to occur as messages are dropped, thus causing false positives.
c. Measured bandwidth: Gossip = 7277 bytes/sec, Gossip+sus = 7344 bytes/sec. These are within 5% of each other as 5% of the average (7277+7344)/2*0.05 = 365.525 is much greater than the difference between the two numbers.

MP1:

We modified MP1 accordingly so that we were able to use the distributed grep to validate the changes in the local logs generated by the membership list changes such as new nodes adding, failing, etc.