CS 425 MP1 Report

Name (netid): Tony Joo (djoo5), Dion Hiananto (hianant2)
Cluster Number: g47
Github Repo: https://github.com/cs425-ece428/mp1-mp1-td
Language: Python3, ver 3.6.8
Libraries:

- protobuf
- matplotlib and numpy (both are only needed if you want to run the graphing script, graphing.py)

To Run a node:
python3 gentx.py [hz] | ./mp1_node [nodeName] [port] [config_file]

To run the graphing script after all nodes are terminated:
python3 graphing.py [number of nodes]

Protocol Description

We are using the ISIS algorithm which keeps total ordering as described in the class lecture and as proven in the lecture slides. It achieves the total ordering by ensuring that the sender process multicasts message to every other process. Once the process receives the multicast message, it replies back with its proposed sequence number, which will always be larger than all observed agreed priorities and any previously proposed priorities.

Each time there is a new proposed sequence for a message, it is stored into the priority queue of each process sorted by the message priority.

The sender process will then wait for everyone in the group to reply with their proposed priorities. After the sender process receives every process' proposed priority, the sender process will find the maximum of all of those and multicast the agreed sequence to everyone in the group.

Every process will then get the agreed sequence of the message, update the priority queue and deliver the message from the priority queue if it is at the top of the priority queue.

Implementation details

We are using TCP Protocol to ensure reliable message delivery. Each process also has a counter starting from 1 that will be incremented each time it receives or multicasts a new message. Each process holds a priority queue (heapq), which is ordered by (the proposed

sequence of its process, process ID). To break ties, we are using string comparison in which the “lower value” of String(process id) will be processed first.

Each entry in the priority queue holds the is_agreed boolean field to tell if the message has been agreed so that the agreed message that’s on top of the priority queue can be processed immediately. Also, we use threading.Lock() whenever we try to modify the priority queue, the sequence counter and the hashmap of the proposed sequences for each message to ensure that threads will not be in a race condition.

Failure Handling

We R-multicasted agreed sequences to every other node in the group to handle failure scenarios. We rely on the multicast that is sent between intervals(from gentx.py) to detect failures and once a node identifies that another node failed(by catching exceptions when trying to send a message).

Once a failure has been detected, the node will then sleep/wait for 15 seconds before deleting all the messages where the originator is the failed node AND didn’t have an agreed sequence yet in the priority queue. We need to delete these messages to make sure that the top of the priority queue is not stuck with an unprocessable message(since after 15 secs, we can assume that the node that originally multicasted the message has died and no other nodes didn’t receive the agreed sequence yet for the node) and continue processing.

Every node will then also go over all messages sent by the node(itself) in the priority queue which is waiting for the dead node’s proposed sequence reply. It will then determine and multicast the agreed sequences if the message is only lacking the dead node’s proposed sequence(it received the proposed sequence from everyone else in the group except from the node that just died).

We decided to use 15 seconds since we expect in the worst case the message will arrive within 2 * max OWD (1 max OWD for a message to arrive at a separate process other than us and another 1 max OWD for the worst time another node takes to R multicast it to us). In the MP spec, it is mentioned that the expected OWD will be 5 seconds, we need to at least wait for **10 seconds**. We added an extra **5 secs(**during graphing we used 3 seconds extra instead of 5) as precaution and buffer for processing time/locks/other unexpected delays.

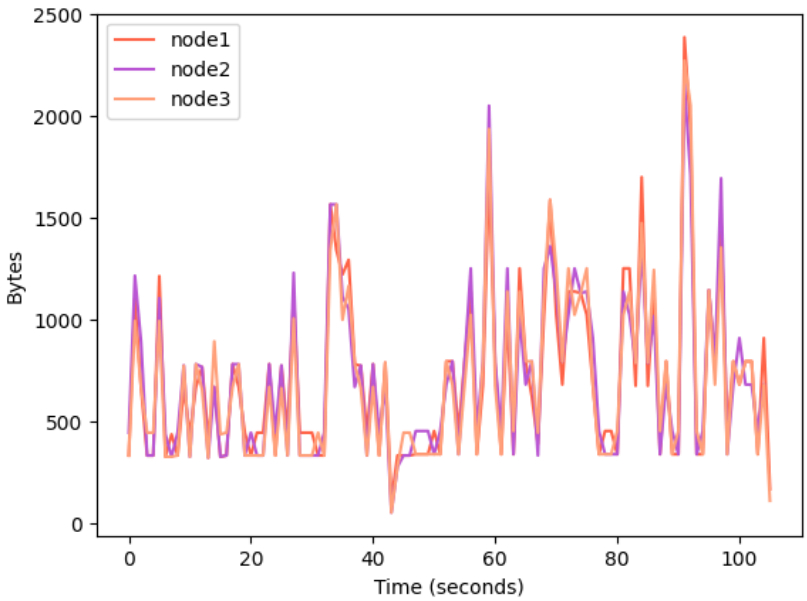
Graph Notes:

- There used to be like a big spike in all of our graphs but we cut that out of the graph. (The spike is because we are manually running the script on each node manually and packets might be piled up causing a big spike. We also recorded for around 120 secs for each scenario so we have extra seconds to trim)
- We used 13 second waiting time in the delay failure instead of the mentioned 15 seconds in the document. (We decided to add extra 2 seconds, to be more safe)
- In Scenario 3 and 4, there is a huge spike in the middle of the delay graph(when we killed 3 nodes) because when a node detected a failure our node would wait for ~13 seconds. (In case of 3 nodes, it might wait for 3 * 13 seconds). There is also a spike in the bandwidth because there are messages piled up when the nodes are sleeping and the nodes will all get processed once we finish sleeping.

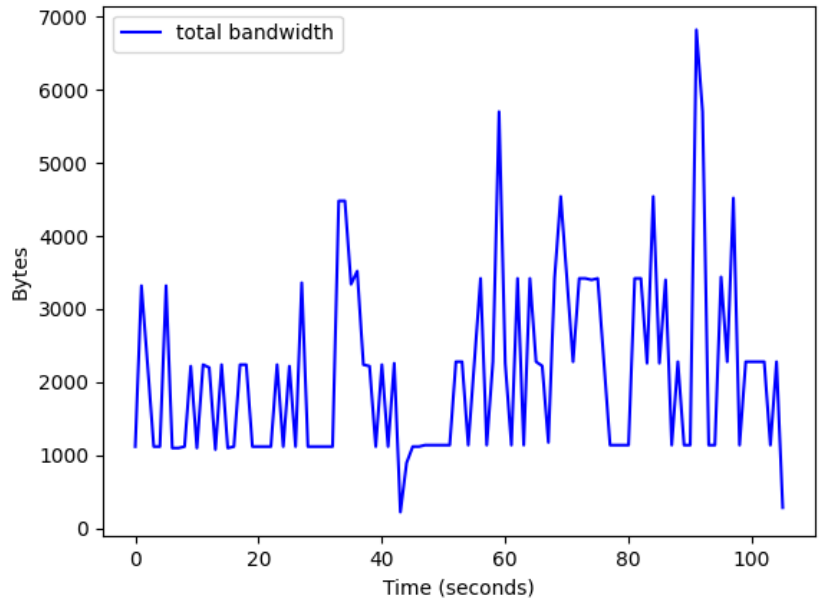
Scenario 1 Graphs

3 nodes, 0.5 Hz each, running for 100 seconds

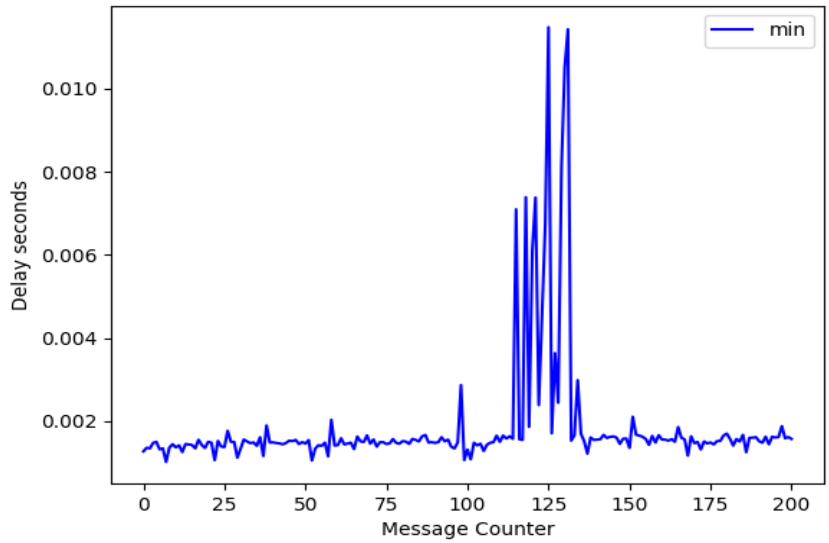
Bandwidth per node



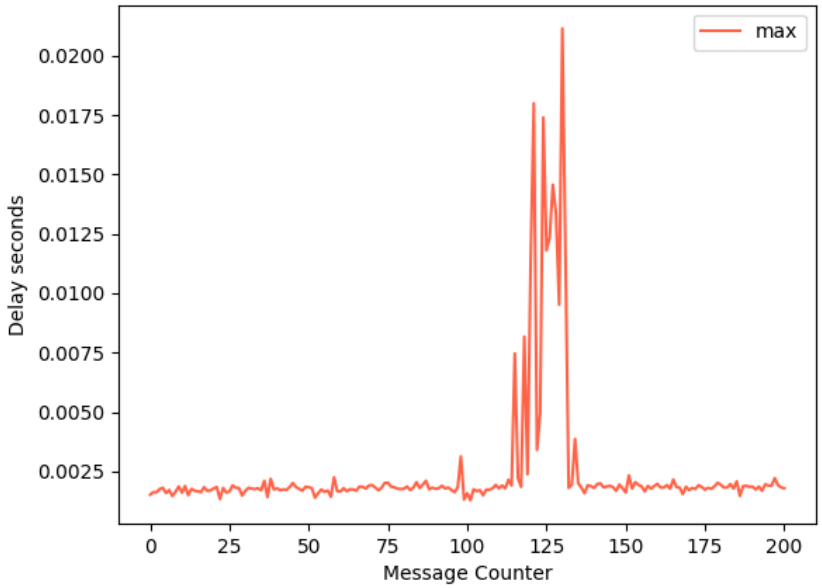
Total bandwidth



Min Delay



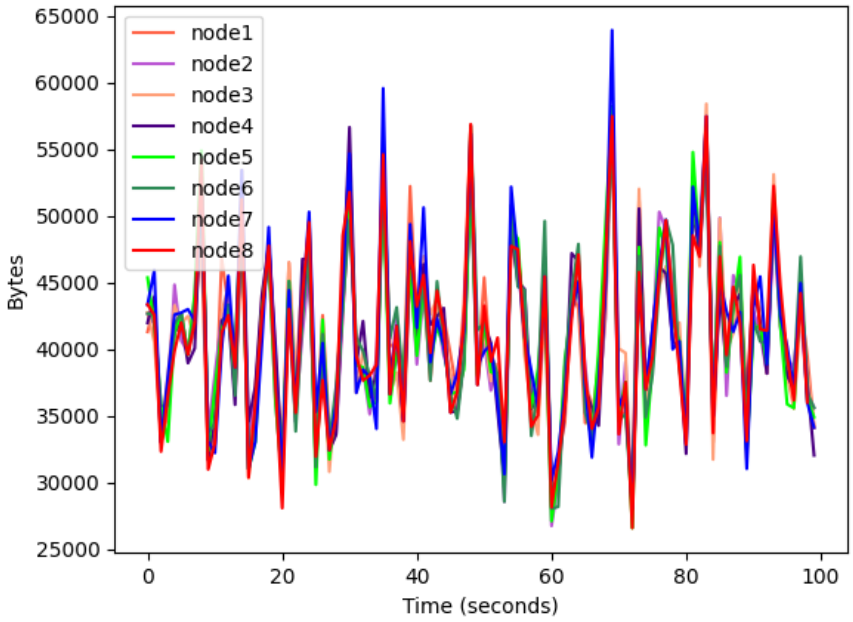
Max Delay


max
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
Delay seconds
0
25
50
75
100
125
150
175
200
Message Counter

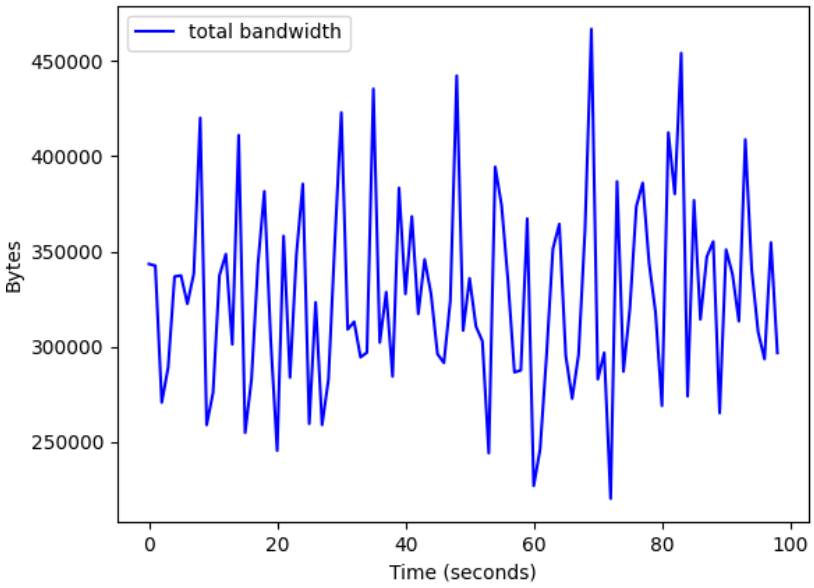
Scenario 2 Graphs

8 nodes, 5 Hz each, running for 100 seconds

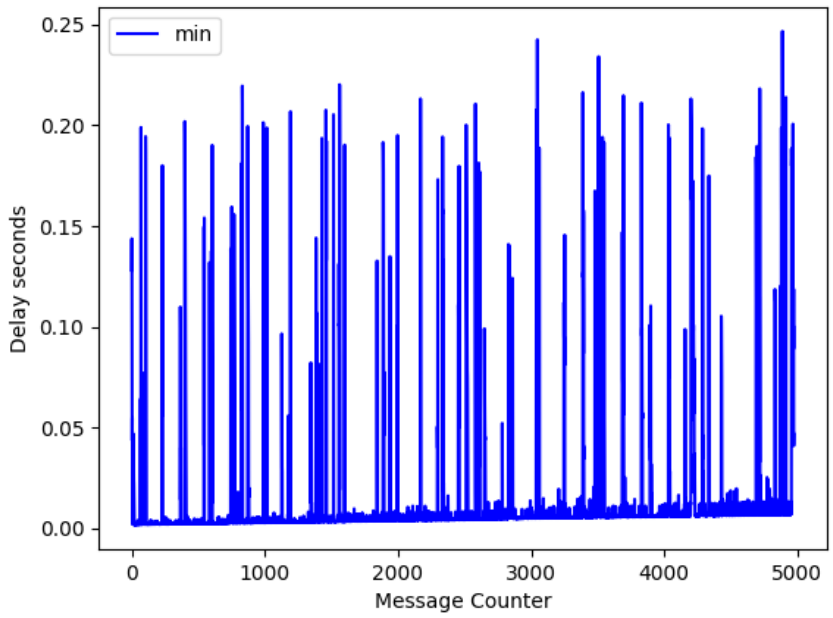
Bandwidth per node



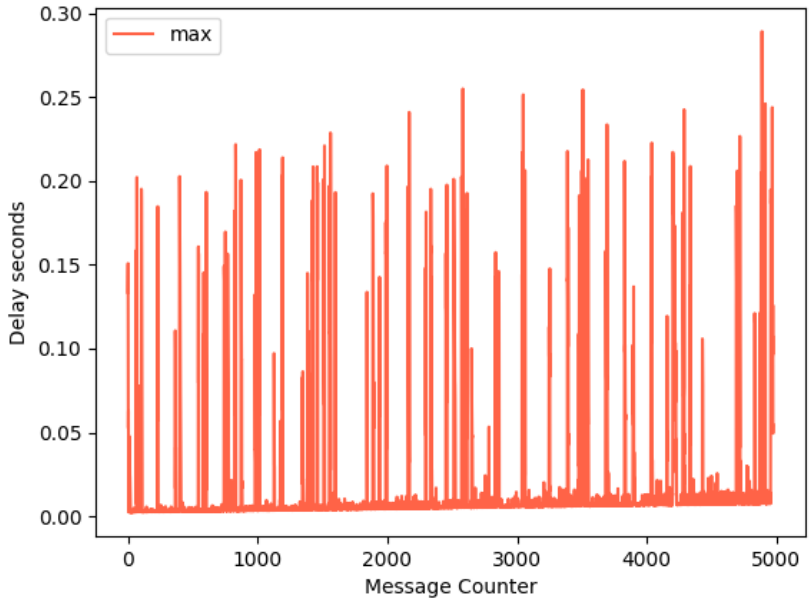
Total bandwidth



Min Delay



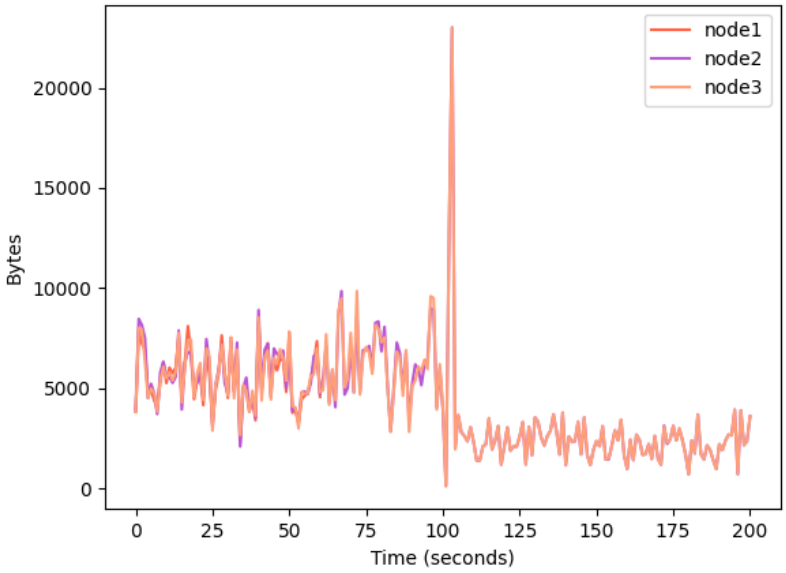
Max Delay



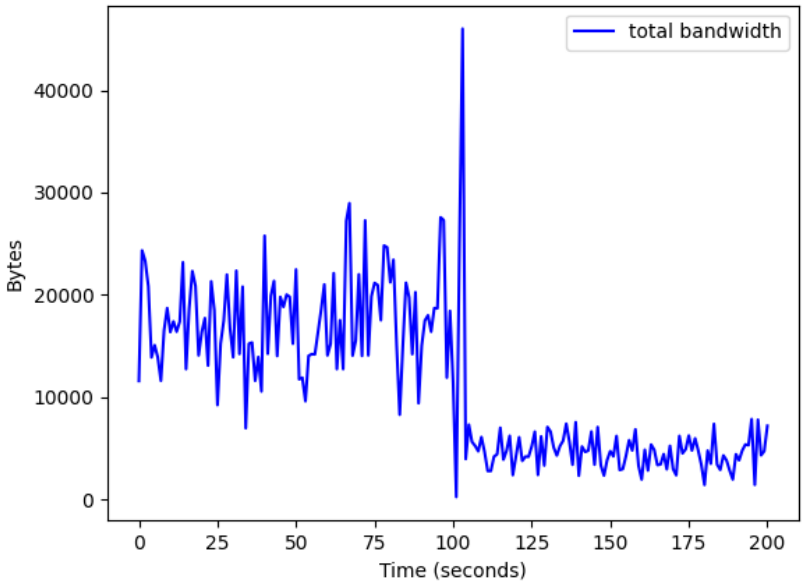
Scenario 3 Graphs

3 nodes, 0.5 Hz each, running for 100 seconds, then one node fails, and the rest continue to run for 100 seconds

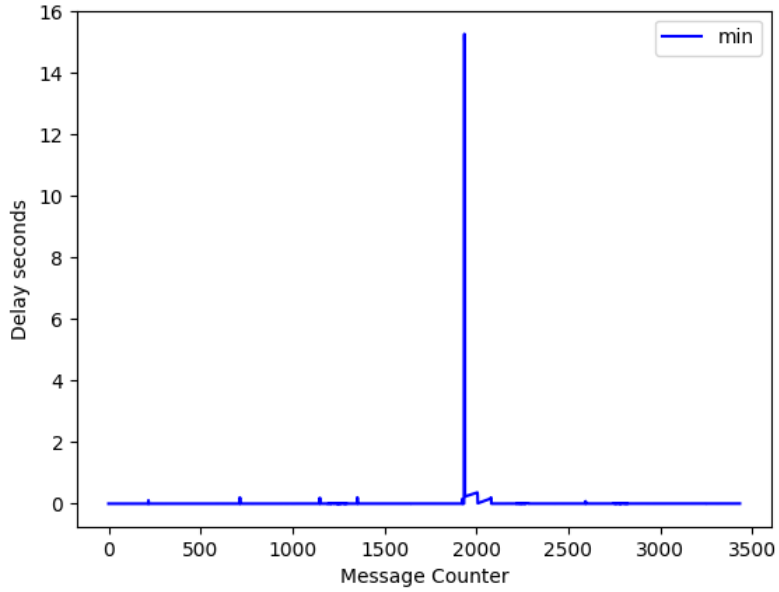
Bandwidth per node



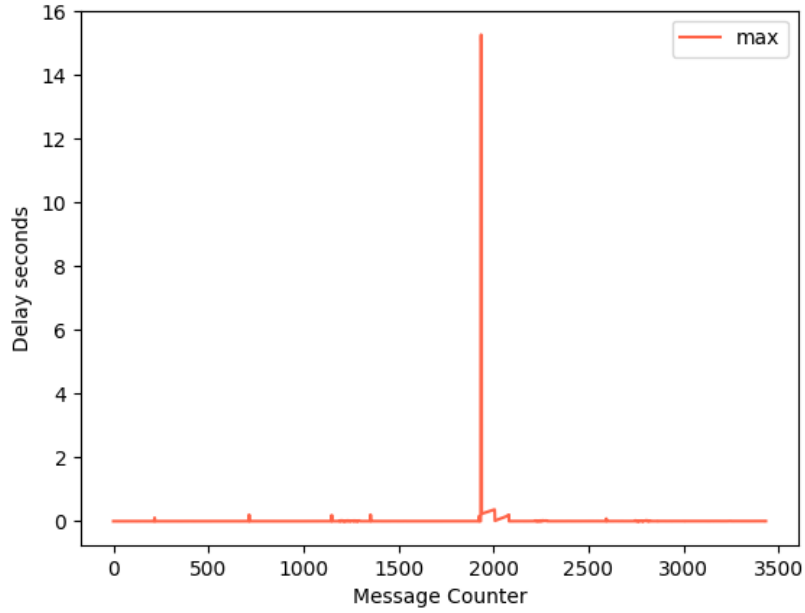
Total bandwidth



Min Delay



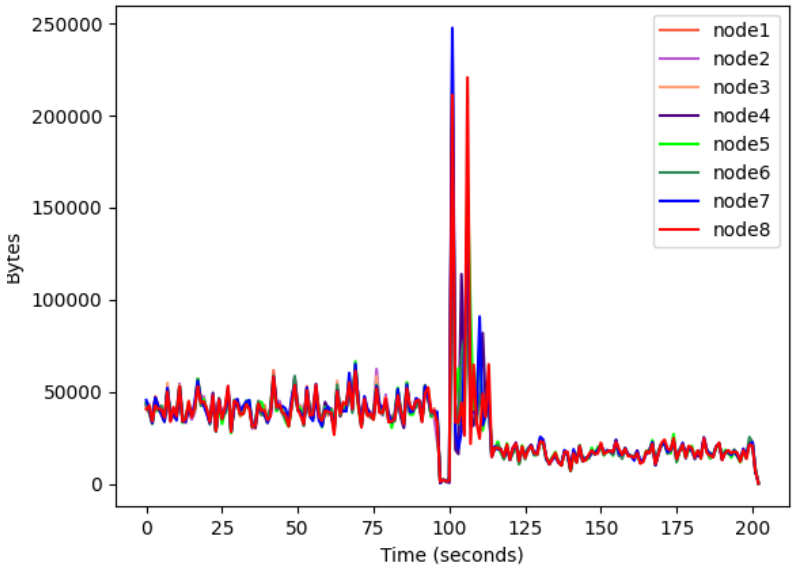
Max Delay



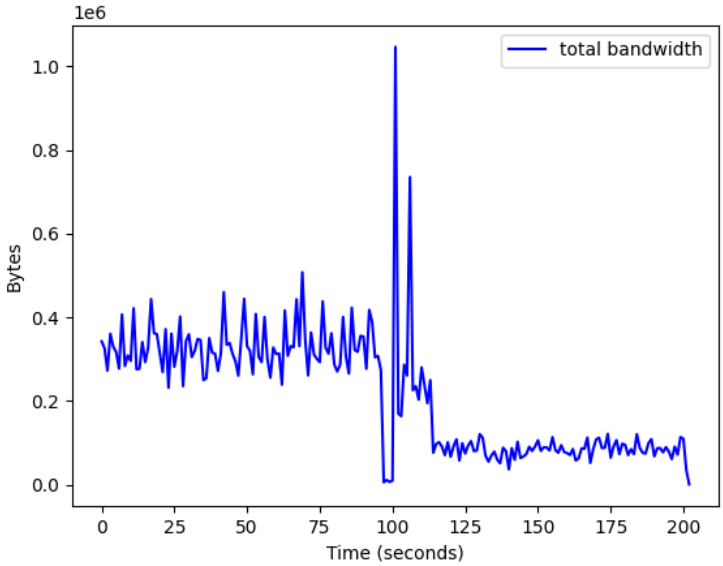
Scenario 4 Graphs

8 nodes, 5 Hz each, running for 100 seconds, then 3 nodes fail simultaneously, and the rest continue to run for 100 seconds.

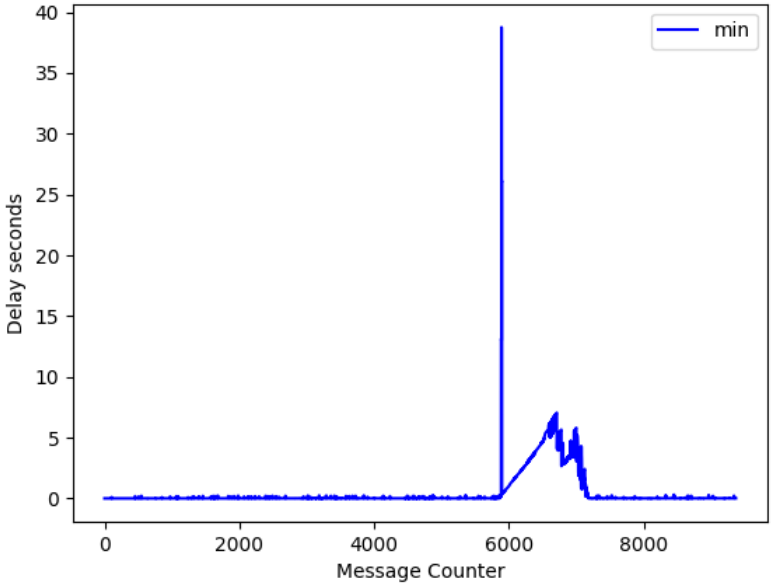
Bandwidth per node



Total bandwidth



Min Delay



Max Delay