ECE 428 MP1 Report

Name: Qihao Wang, Xianfan Gu
Netid: qw2, xianfan2

Repository URL: https://gitlab.engr.illinois.edu/qw2/cs425.git
Version number: de28d7bd65d4c421c329d10d3a1ecb92915969ef

the code with graph:
in the "gotest" branch with version number: 0d00c5b00fb9d3840ce16e0af0ccb78ec8d7d12d

- The cluster number you are working on

For the small scenario: 3 nodes, we use machines “01”, “02” and “03” as my nodes.

For the large scenario: 8 nodes, I use machines from “01” to “08” as my nodes.

For the experiment part, from the output files, We can know the delay for each timestamps by computing the difference of current receiving time and the timestamps in receiving packets. The delay = T receive_node −T packet (at
the same event). The bandwidth for each seconds is measured by the bits of receiving message
during the delay. Hence we can know the bandwidths, which is

$$Bandwidth = \frac{bits\ of\ the\ receive\ packets\ at\ the\ delay}{delay\ intervals}$$

The amount of process time is the time = T process_node −T packet. First node process time is the earliest process time in each message, the last node process time is latest process time in each message.

- Instructions for building and running your code

In each home directory, type git clone https://gitlab.engr.illinois.edu/qw2/cs425.git and “cd mp1” to the directory for this MP. The source code is node.go. Type “make” to compile all files we need for this mp or “make clean” to remove all executables.

In each machine, type “python3 -u gentx.py” + frequency + “ | ./mp1 ” + number of nodes in the network + the port number each node listens.

The balance information will be sent to standard out every 5 seconds

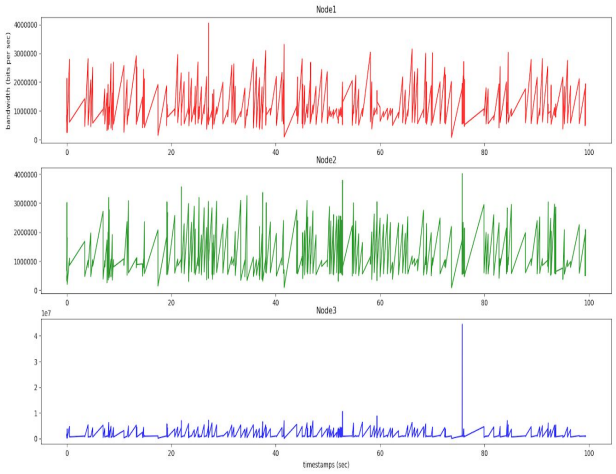
- Design Document

We get inspiration from the Campuswire Post and comments in #233 “A starter approach for MP1” and based on ISIS algorithm. For each node, we first use a go routine to accept all the incoming connections and use a handler function to handle each incoming connection. Then, we will add 1 second delay and construct n-1 outgoing connections to all other nodes and store these outgoing connections in a global variable. After that, we will add 2 seconds delay for nodes construction and read the message from the standard input and pack the message into a “packet” struct to multicast to all other nodes and push this package into the local priority queue. Once the incoming connection handler receives a packet, it will first check if it is already received. if it is a duplicate message, we will skip it. The way we check the duplicate package is that we combine the sequence number + reply from + isfinalpriority as the key and mark this key received if we do not see it before. There are two possibilities of the incoming package, it is either request or reply from the other nodes. If it is a request from the other nodes, we will check if it is the final priority. If it is the final priority, we will multicast the final priority and push it into the local queue and deliver any deliverable messages at front of priority queue. Otherwise, we will reply and push it into the local queue with priority max(agreed priority, proposed priority) + 1. If it is a reply from the other nodes, we will update the priority for that package and check if we can deliver it and multicast the final priority.

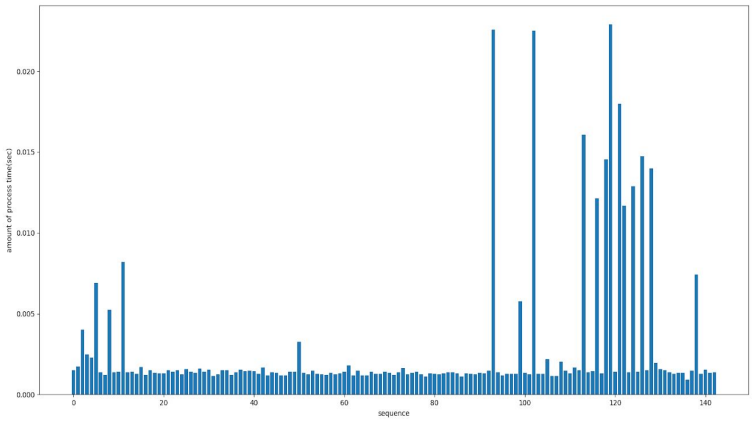
The ISIS algorithm guarantees the total order delivery and the re-multicast for the final-priority package at each node guarantees the reliable transmission. We do not need to multicast the request from other nodes if it is not final-priority package because the TCP protocol guarantees that the package will not drop in the middle and if the sender nodes is dead in the middle, it will not receive all the reply from other node and will never deliver this message. The way we handle the node failure is that once we find the first package in the priority queue is sent from the dead node and not deliverable, we will pop it from the queue. If the first package in the queue still needs the reply from dead nodes, we will deliver this package.

- Graphs of the evaluation

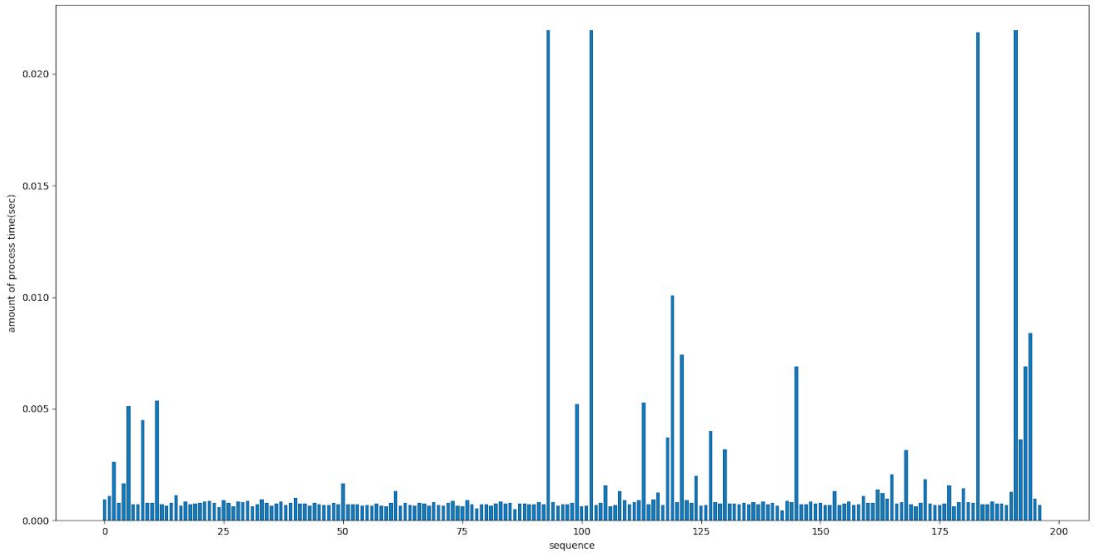
To plot the graph, we use the matplot library in python for plotting. Fig. 1(a) shows the real-time bandwidth in the scenario of 3 nodes, 0.5 Hz. The result reflects that two of three nodes performed similar bandwidth, the rest of the nodes gained the bandwidth(~4000000 bits per sec) which is nearly the sum of the bandwidths of the other two nodes(~2000000 bits per sec) .
Fig. 1(b) shows the amount of process time in the scenario of 3 nodes, 0.5 Hz.The amount of process time increased to an extent as the sequences are increased. Mostly, the amount of time stayed at near 0.002 sec.



(a) Bandwidth among three nodes at frequency = 0.5Hz for 100 secs

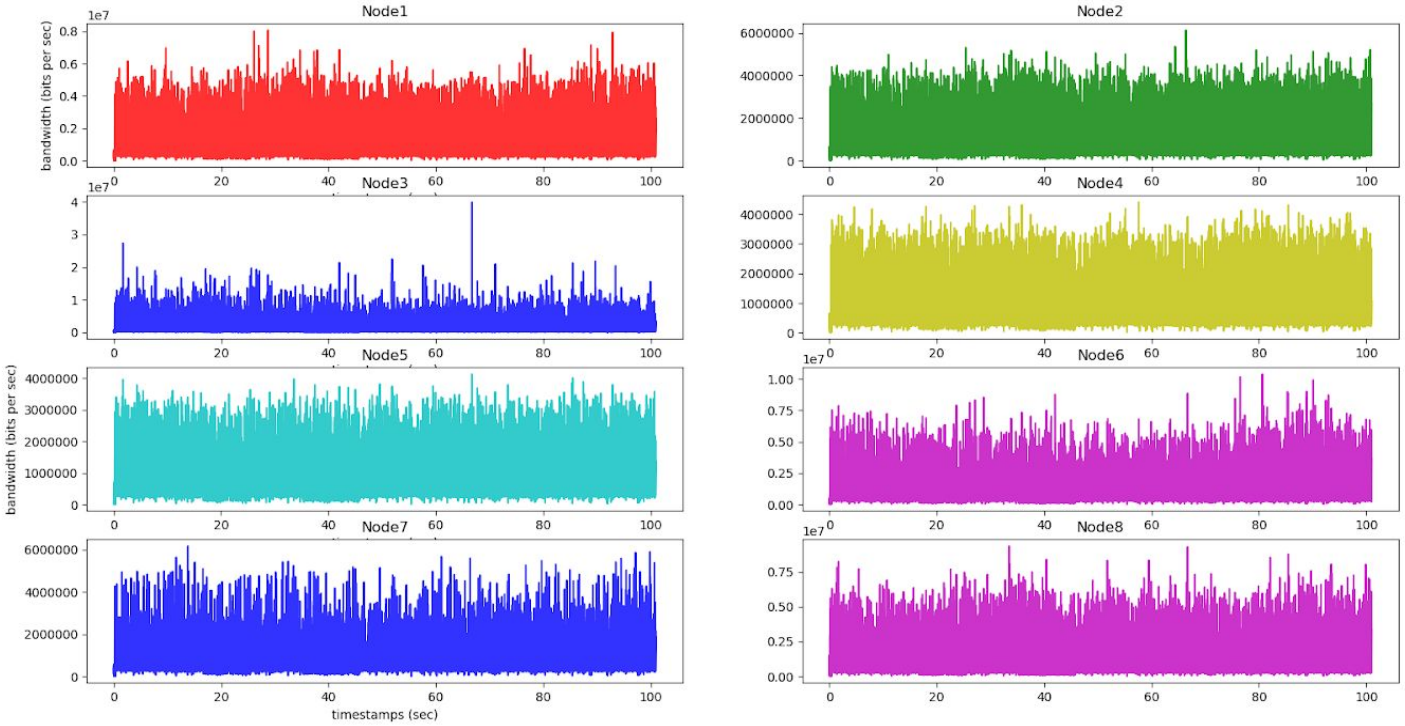


(b) The amount of the last node process time for 100 secs

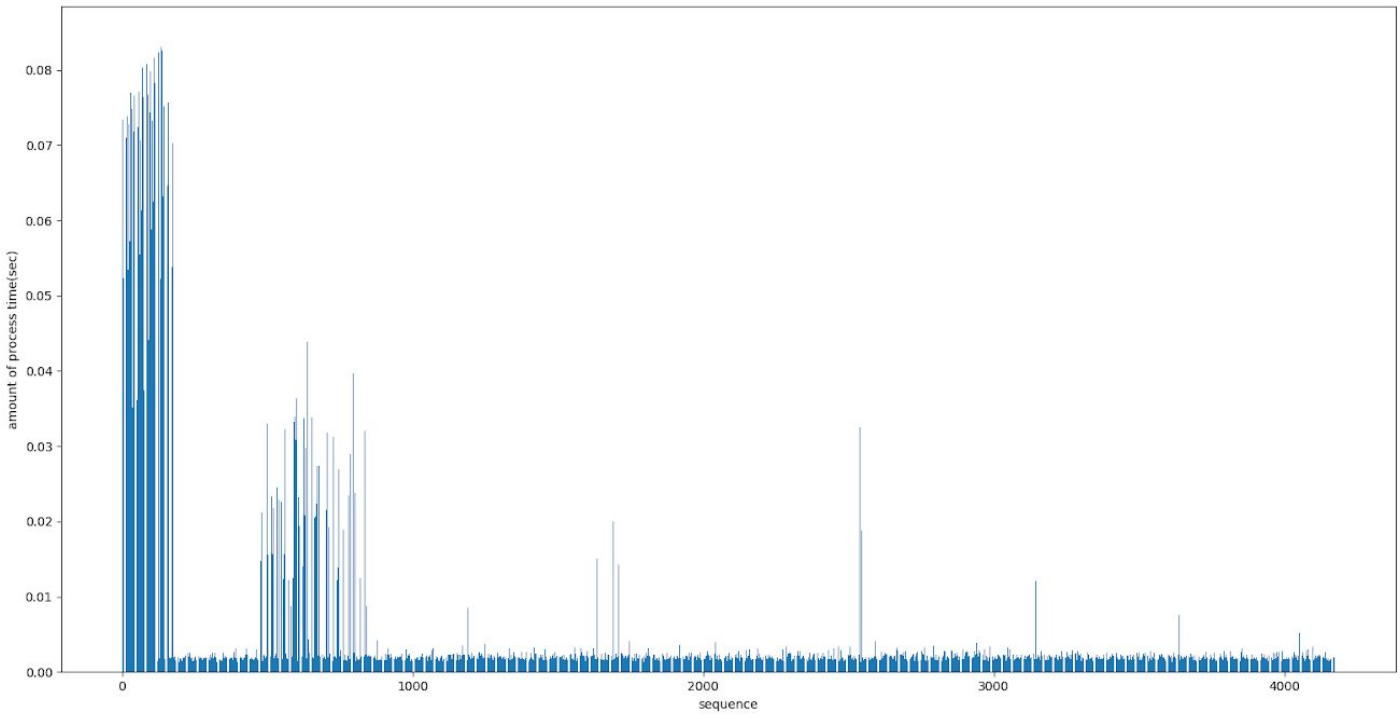


(c) The amount of the first node process time for 100 secs

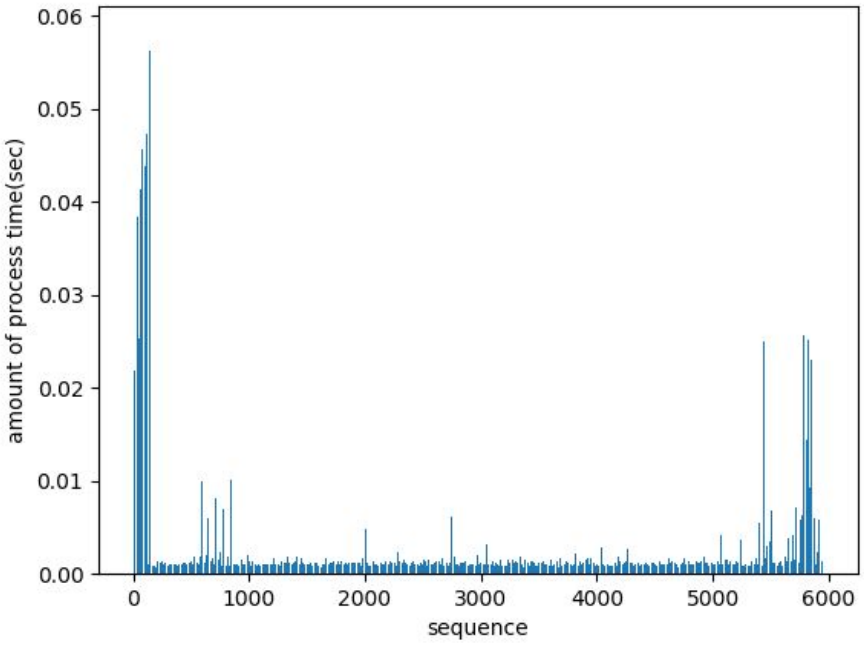
Figure 1: scenario 1 - 3 nodes, 0.5 Hz each



(a) Bandwidth among eight nodes at frequency = 5Hz for 100 secs

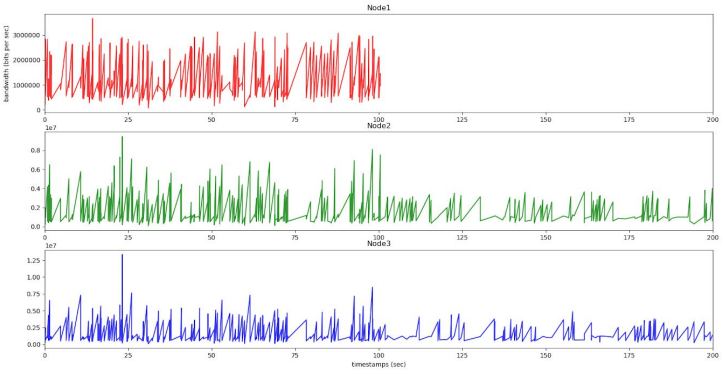


(b) The amount of the last node process time for 100 secs

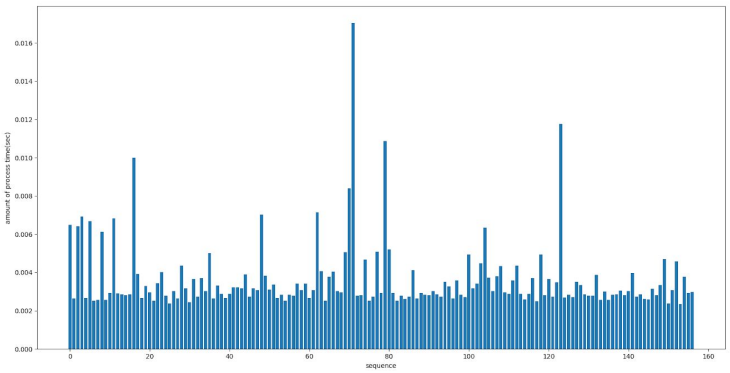


(c) The amount of the first node process time for 100 secs

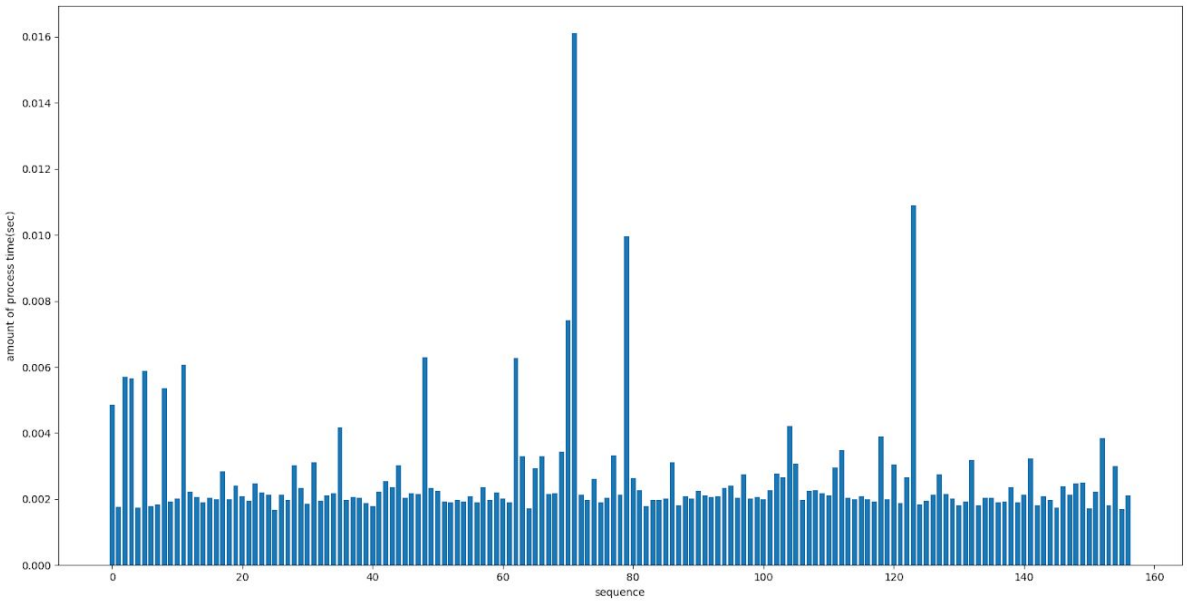
Figure 2: scenario 2 - 8 nodes, 5 Hz each



(a) Bandwidth among three nodes at frequency = 0.5Hz for 200 secs

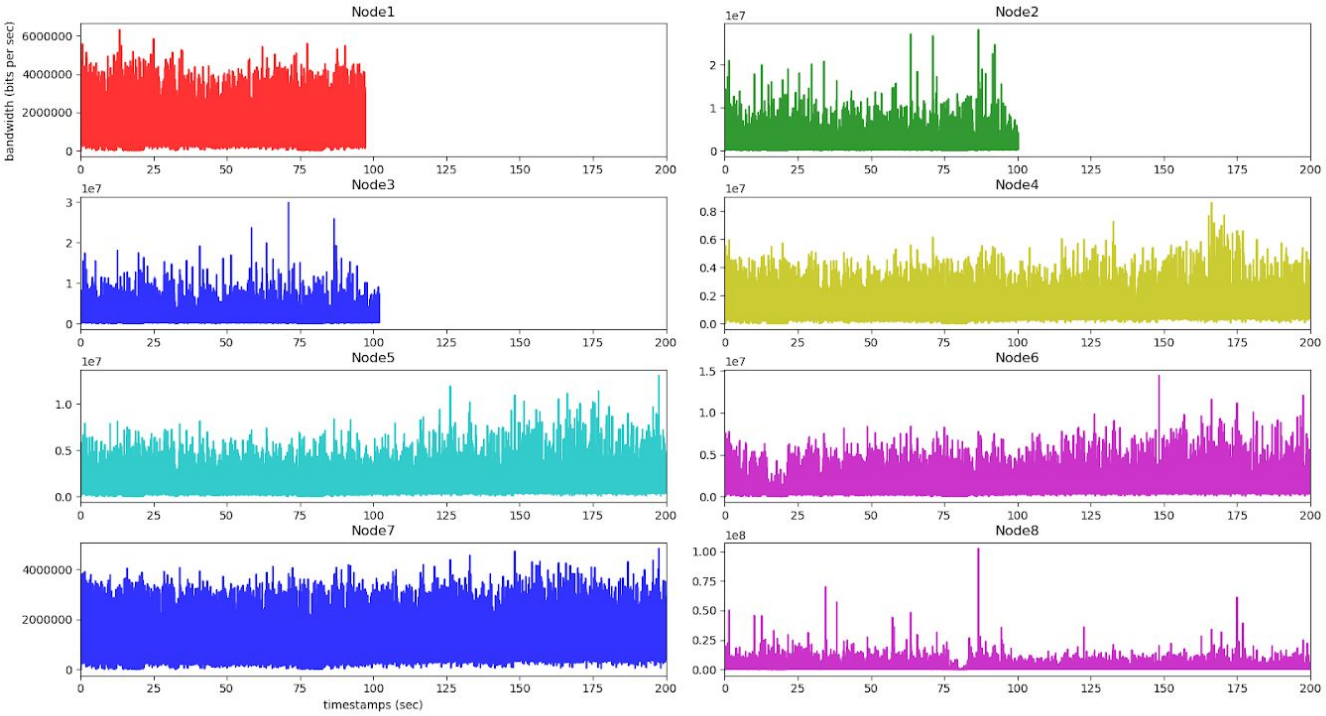


(b) The amount of the last node process time for 200 secs

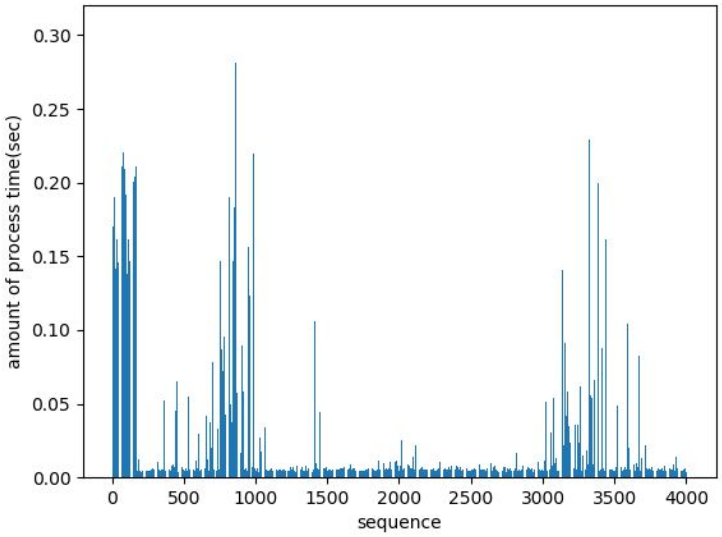


(c) The amount of the first node process time for 200 secs

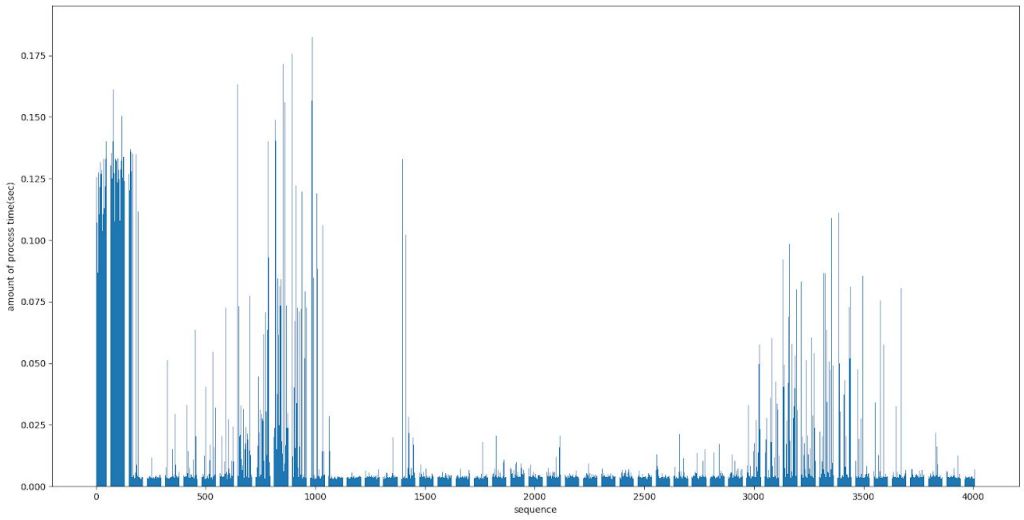
Figure 3: scenario 3 - 3 nodes, 0.5 Hz each (one node failed)



(a) Bandwidth among eight nodes at frequency = 5Hz for 200 secs



(b) The amount of the last node process time for 200 secs



(c) The amount of the first node process time for 200 secs

Figure 4: scenario 4 - 8 nodes, 5 Hz each (three nodes failed)

Fig. 2(a) shows the real-time bandwidth in the scenario of 8 nodes, 5 Hz for 100 seconds. All the eight nodes gained the consecutive higher bandwidth than scenario 1(~5000000 bits per sec, ~4000000 bits per sec, ~3500000 bits per sec, ~10000000 bits per sec, ~3000000 bits per sec, ~4900000 bits per sec, ~5000000 bits per sec, ~4000000 bits per sec), only one node have 2 times difference to the other nodes.

Fig. 2(b) shows the amount of process time in the scenario of 8 nodes, 5 Hz.The amount of process time reached the peak at the beginning time of the test, which was nearly 0.07 sec. Mostly, the amount of time stayed at near 0.002 sec.

Fig. 3(a) shows the real-time bandwidth in the scenario of 3 nodes, 0.5 Hz for 200 seconds when one of the three nodes failed at 100 seconds. The bandwidth of the other two nodes apparently decreased when the first node failed. The bandwidth of the second node was the sum of the other two nodes before 100 seconds.
Fig. 3(b) shows the amount of process time in the scenario of 3 nodes, 0.5 Hz when one of the three nodes failed at 100 seconds.The amount of time stayed at nearly 0.002 sec.

Fig. 4(a) shows the real-time bandwidth in the scenario of 8 nodes, 5 Hz for 200 seconds when three of the nodes failed at 100 seconds. The bandwidth of the other five nodes stayed at the stable bandwidth (~3000000 bits per sec, ~4900000 bits per sec, ~5000000 bits per sec, ~4000000 bits per sec, ~12000000 bits per sec) when the first node failed, only one node have 2 times difference to the other nodes.
Fig. 3(b)shows the amount of process time in the scenario of 8 nodes, 5 Hz when three of the nodes failed at 100 seconds.The amount of process time reached the peak at the beginning time of the test, which was nearly 0.15 sec at 0~150 sec. Mostly, the amount of time stayed at near 0.003 sec.

Reference:
“A starter approach for MP1”: https://campuswire.com/c/G86520A5B/feed/233