MP1: Event Ordering Report

Paul Pok Hym P Ng (ppng2)
Bruno Seo (sbseo2)

March 10, 2020

1 Cluster number / Repository

Our cluster number is 15.
Below is the URL of our git repository.
URL: https://gitlab.engr.illinois.edu/ppng2/ece428/tree/master
(Revision number: a0d9563aeac6e449cfd7989147bc73ed51029adb)

2 Design Document

In this MP we implement the ISIS algorithm. This algorithm is an asynchronous and decentralized algorithm that allows for messages to be totally ordered. All nodes in the system must be connected to each other.

The algorithm can be described in several simple steps.

1. A node A multicasts its desired message to all other nodes.
2. $\forall n \in Nodes\ s.t.\ n \neq A$ will then propose a sequence number and send it back to A
3. The node A then chooses the largest of all the sequence numbers proposed to it (including its own) and multicasts this number back out (including itself).
4. When all nodes receive the sequence number, they assign it to the message and mark it as deliverable
5. Each node is now to sort sort the holdback and deliver any messages at the front which are marked deliverable (lowest sequence number)

Our design comprises of several files.

- account.py: Unused as we store accounts in the process as a dictionary
- isis.py: Utitily functions for sequence number generation
- gentx.py: A file to automatically generate random input messages
- main.py: The main loop which spawns threads and handles things such as heartbeats and messsage passing

- message.py: Contains a class and utility functions for how we define a message
- process.py: Contains our definition of a "node"

Here we will now describe our main execution path. Firstly, main spawns threads used for connections between all nodes. We space ports out in increments of 100. Therefore a node will occupy ports between `BASE_PORT + SPACING * OFFSET` and `BASE_PORT + SPACING * OFFSET + NUM_NODES` for accept ports and `BASE_PORT + OFFSET - 1` to `NUM_NODE * SPACING + OFFSET` for connect ports, where `OFFSET` represents the Node's number (VM1 = Offset 0, VM2 = Offset 1). Additionally, we have an offset for our heartbeats these will then will take ports between `BASE_PORT HB_SPACING + SPACING * OFFSET` and `BASE_PORT + HB_SPACING + SPACING * OFFSET + NUM_NODES` for accept and `BASE_PORT + OFFSET - 1` to `NUM_NODE * SPACING + OFFSET` for connect.

We then spawn threads to make the connection. With this, we have a total of $4*(numNodes-1)$ ports dedicated to network traffic per process.

Now we discuss what threads we have. After connection, the threads for accept/connect will be closed. The persistent threads are as follows

- A thread performing the ISIS algorithm
- $2 * (N - 1)$ threads for sending messages from `stdin` to other processes
- $2 * (N - 1)$ threads to send heartbeats to each other
- One heartbeat detection thread which is used to detect when a process dies
- One `stdin` thread to constantly read the input from `gentx.py`

To detect a crash we use two functions, namely `heartbeatSend`, `heartbeatRecv`, `heartbeatDetect` and `heartbeatParse`.

The heartbeat send and receive functions do as their name implies, they send and receive heartbeat messages. We use an all to all heartbeat protocol to ensure that each process is able to detect a crash without relying on a central node (which is not allowed). The delay between heartbeats is a global variable set ahead of time (0.5 seconds). Finally we also use a predefined K variable. This is used during crash detection.

The heartbeat parsing function is part of a heartbeat receiver thread and takes a heartbeat message as input and updates global variables containing the RTT between each node and last observed times. Using these numbers we average every heartbeat we get to obtain an average RTT. This will be used in heartbeat detection to determine when a process has timed out.

The heartbeat detection function is a separate thread and is used to check if a process has crashed. The jist is as follows. If $currentTime - lastHeartbeat > K * averageRTT$

then we have a crash. Update a global list of crashed process IDs and kill the sender process to that process.

With this crashed process ID information we can then proceed to decrements counters such as `process.numOtherProc` or `NUM_NODES`. Furthermore, we are then able to purge stale messages (unable to receive an SRep) or change the required RReps for a message a process own's. This way a process will not infinitely wait on a S/RRep that may never come.

2.1 Classes

2.1.1 Messages

A **Message** is defined by the following parameters

- source: The originating process
- target: The target process of a receiver reply (RRep)
- messageId: A hash of the message content a global message counter and the process id this message originated from
- proposedSeq: The proposed sequence attached to the message
- type: The type of message (deposit or transfer)
- sourceAcc: The source account the money is being transferred from
- targetAcc: The target account the money is being transferred to
- amount: The amount of money to be transferred
- isDeliverable: This marks whether we have received a sender reply (SRep) representing whether this message's sequence number has been finalized
- isRRep: A boolean telling us if this message is a receiver reply
- isSRep: A boolean telling us if this message is a receiver reply
- Message.counter: A global message counter used to differentiate between messages. This is required because we can envision the same process sending the same message twice in a row leading to the same hashed message id. Therefore everytime a new message is created (per process) we increment this counter

We have two classes that inherit from **Message**. These are **ReceiverReply** and **SenderReply**. These do not have extra fields but rather set `isRRep` and isSRep to true by default.

In addition to these fields we also include several function. These functions allow us to preprocess raw string data and set values in our class.

2.1.2 Process

A **Process** contains the following fields

1. pid: Process id
2. numOtherProc: The number of process in the cluster (other than yourself aka NUM_-1)
3. proposedSeq: The current proposed sequence number
4. agreedSeq: The current agreed sequence nubmer
5. holdback: A list of messages in our holdback queue
6. delivered: A list of messages which have been delivered. This is never used as we process and discard the message immediately after updating accoutn information
7. accountDict: An account dictionary. The key is the account name and the value is the balance of the account
8. socketsDict: Unused but was originally designed for each process to keep a dictionary of what port number represents what node. In actuality we can just pass in this as a parameter during initialization to our threaded accept/receive threads.
9. selfMessageRepDict: A dictionary where we store RReps to messages we own and have multicast. The key is the message id and the value is a list of RRep messages

In addition to setter functions, we also place our receiveReply and createReply functions in the process class. These create new instances of the correct type of message and return them for the main loop in `main.py` to send off.

2.1.3 ISIS

While not a class, the `isis.py` file contains utility functions for generating the correct proposed sequence or largest proposed sequence from a list of messages. These are then used when creating a new message, creating a receiver reply, or when receiving a sender reply. In addition to those functions, we also placed a reordering function here to help us reorder the messages in the holdback queue of our process.

2.1.4 blockingRecvThread and blockingSendThread

These are class wrappers around `multiprocessing.connection.Client()` and `multiprocessing.connection.Listener()`. There are two things of note. We chose to use the `multiprocessing.connection` library because it was designed for sending serialized objects. In our design we do not send a text string representing the reply or message we wish to convey. But rather we send an object. Therefore we `pickle.dumps()` and `pickle.loads()` our object on each end. Moreover, `multiprocessing.connection` guarantees that an object will be sent fully. This allows us to not have to deal with an object spanning two separate buffers (which could happen when using the basic socket library).

3 Ensuring Total Ordering and Reliability

We have run into problems when implementing and guaranteeing total ordering in Python. We have managed to achieve both scenario 1 (3 nodes 0.5Hz), 2 (8 nodes 5Hz), and 3 (3 nodes 0.5Hz with 1 failure).

To achieve this we implemented the ISIS algorithm together with multicast. In order to ensure that we maintain data coherency we use `threading.Lock()`s and `queue.Queues()`. As discussed above, the protocol (steps) are implemented in `main.py`. In order to ensure that our multicast is reliable we use `multiprocessing.connection` which is built on top of TCP therefore guaranteeing us reliability.

Scenario 1 and 2 (Refer to figures 1, 2 for delay and bandwidth graphs) were run on a different version of the code without the heartbeat implementation and process crash detection and handling. Scenario 3 (Figure 3) was run on a different version of the code with the heartbeat and process crash detection and handling. We also verified that **3 nodes at 5Hz** also works.

We are not sure why scenario 4 (8 nodes 5Hz with 3 crashes) is not able to be achieved. The problem is not reliability (we use TCP) but rather the ordering itself. We have not been able to pinpoint the problem after many hours but we think that there may be an addition lock that we are missing or have not placed in a particular place to guarantee that our hold-back queue is not written to during some critical point.

We believe such issues may also be due to Python itself. For example, `multiprocessing.connection` does not return an `addr` structure which we can index to obtain port numbers and other information. Or `holdback.sort()` throws errors saying we are editing while sorting even while locked. Therefore it may be better to implement future MPs in C/C++ due to the large number of mature libraries and low level control C/C++ provides.

4 Instructions for running code

Please follow the instructions written as below.

Processes

There will be two folders containing two versions of code. This is due to the issues we mentioned previously. There are two zips which will be named `scenario12.zip` and one will be named `scenario34.zip` containing code for both scenarios. The usage is the same. `scenario34.zip` is configured to not print the accounts every 5 seconds but rather `[len(holdback), len(totalOrdered), print(HEARTBEAT_DEAD), count)]`. This is debugging information which allows us to see the length of our holdback queue, the number of totally ordered delivered messages, the dead processes, and a counter for number of iterations of our ISIS thread.

1. Please open **config.py** and configure your ip addresses

2. Please obtain a copy of the repository on each node

3. at the command prompt please enter the following command

```
# python3 -u gentx.py <Hz> | python3 main.py <N> <M>
```

$\langle Hz \rangle$ represents the maximum Hertz at which messages are being sent
$\langle N \rangle$ represents the total number of nodes in the system
$\langle M \rangle$ represents the node number (Note: this number is **ZERO INDEXED**)

3. Press enter on each VM to begin execution.

4. The execution will run forever until `Ctrl+C` is pressed. This will produce a few broken pipe errors on screen but do not worry. This is because we handle our data output in the signal handler.

5. Output data will be dumped on each VM as: `(<N>)node<M>.txt`

5 Data Analysis Methodology

Data we collected via our evaluation process include total ordering verification, final balances of accounts, bandwidth per each second, and transaction delay.
Total ordering is verified by unique message ids generated from hash ids.
Bandwidth is collected by detecting the size of the data downloaded from the sockets. We checked the size of the data and summed it by each second.
Transaction delay is obtained by recording the timestamp when each message is delivered. To guarantee the synchronization, we applied external synchronization method called NTP in our local setting. Since the cluster provided does not allow install NTP library, however, we assume clients to be synchronized for future usage. We obtained the transaction delay by the following equation.

$$Delay = Max(N_0, N_i) - Min(N_0, N_i)$$

N denotes the timestamp of the synchronized node and i denotes the number of nodes connected to the group.

6 Delay and Bandwidth Measurement

Graphs of the evaluation

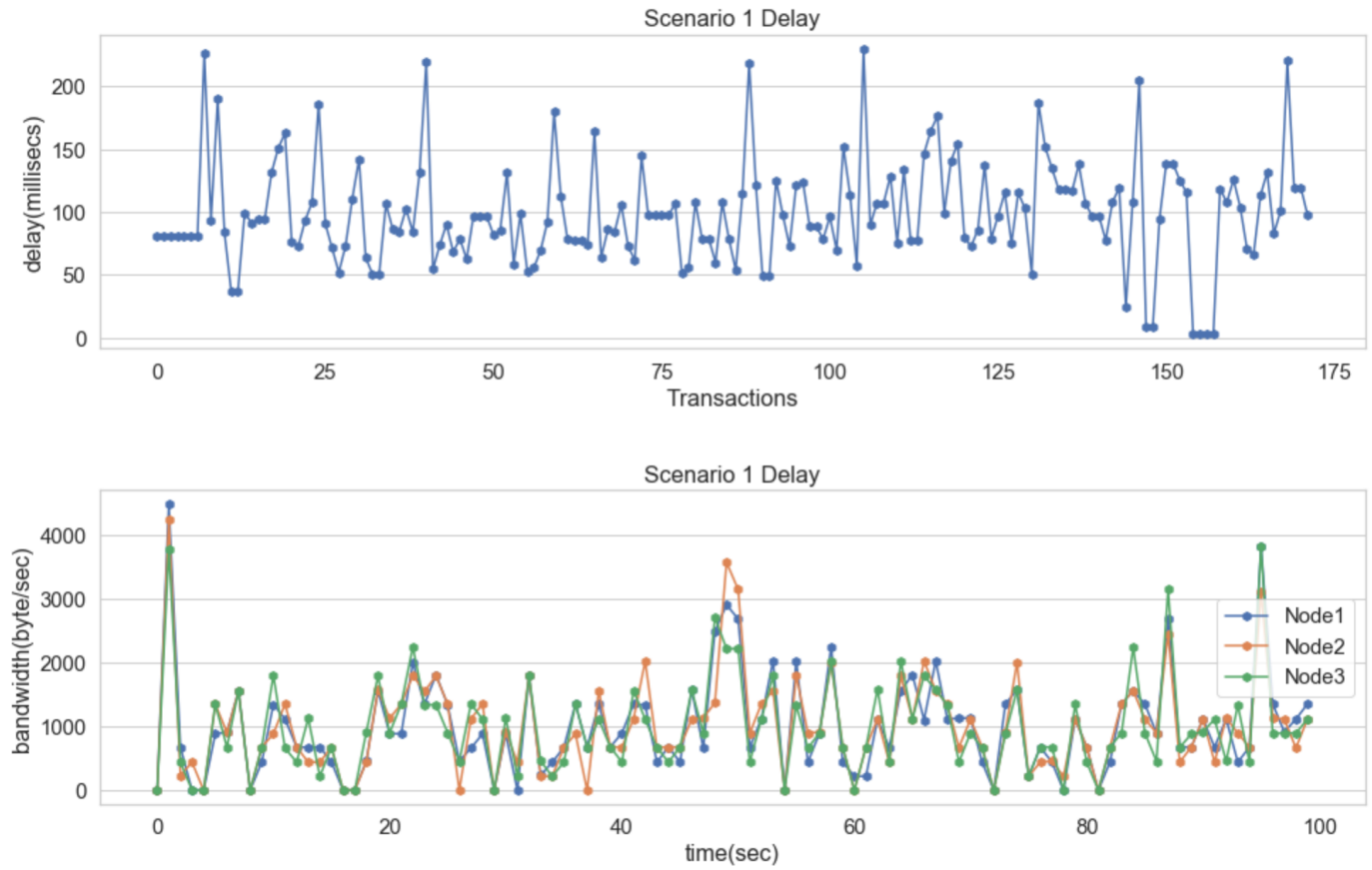

Figure 1: 3 nodes, 0.5 Hz each, running for 100 seconds. * First plot is Bandwidth

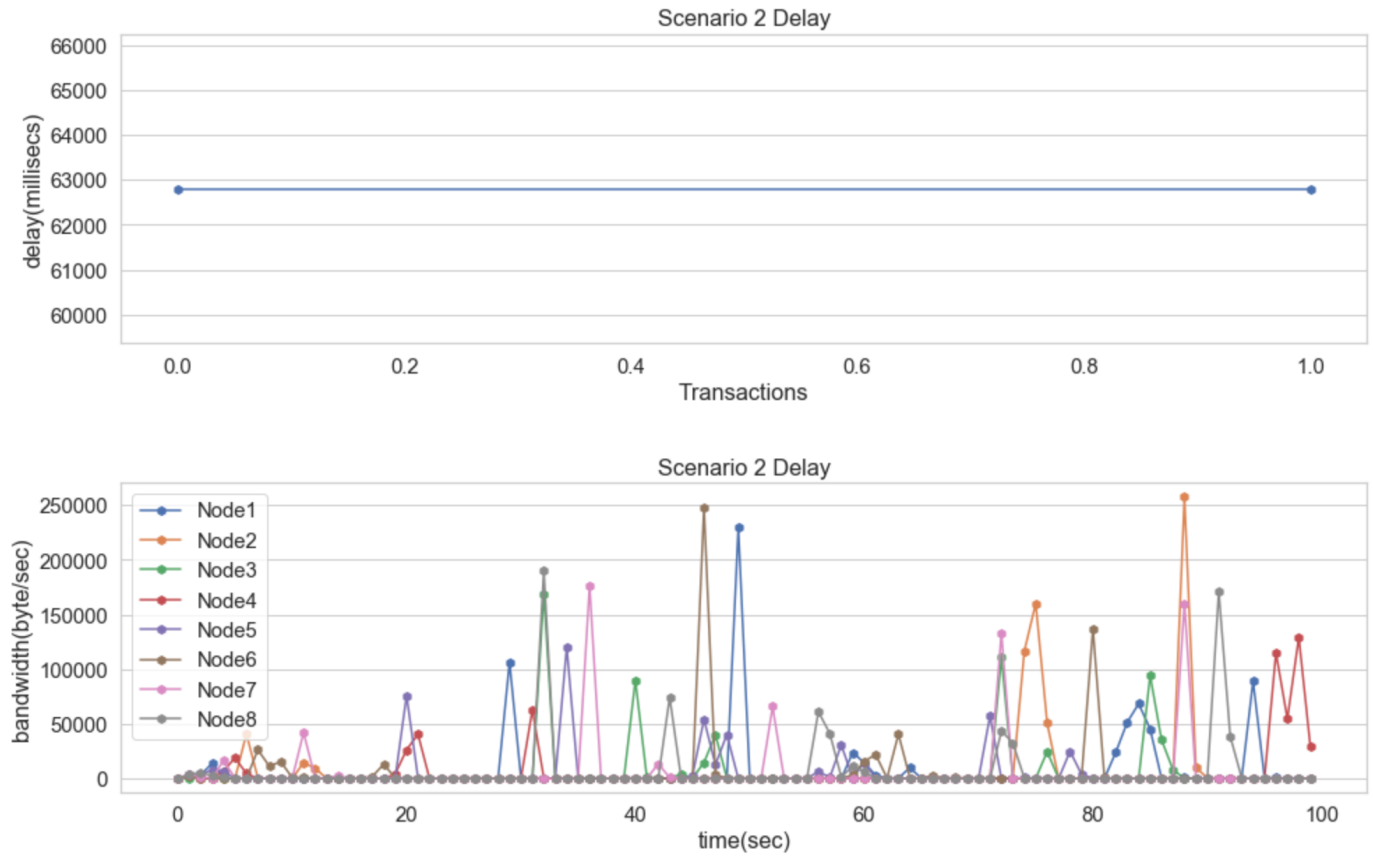

Figure 2: 8 nodes, 5 Hz each, running for 100 seconds. . * First plot is Bandwidth

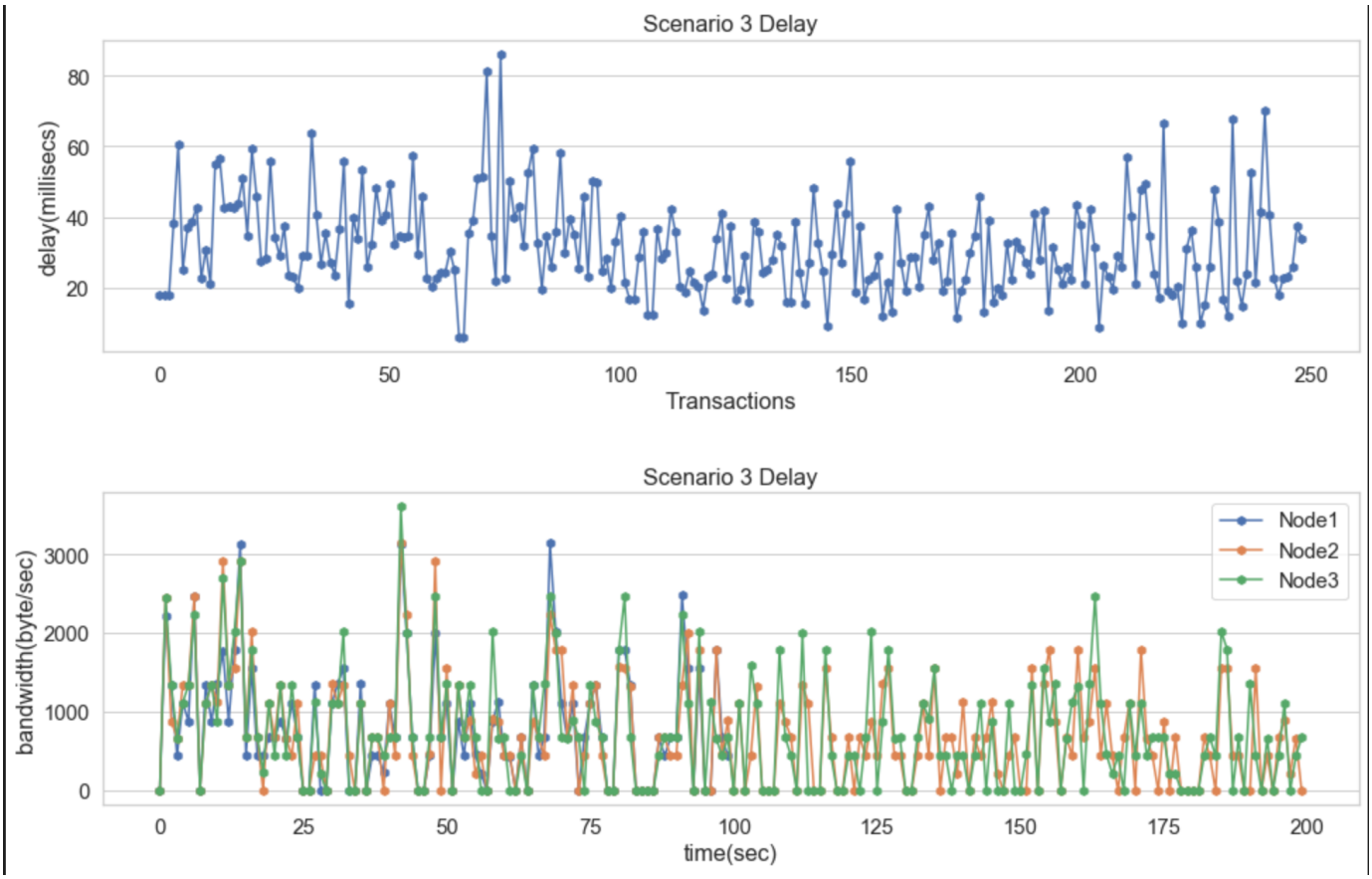

Figure 3: 3 nodes, 0.5 Hz each, running for 100 seconds, then one node fails, and the rest continue to run for 100 seconds.* First plot is Bandwidth

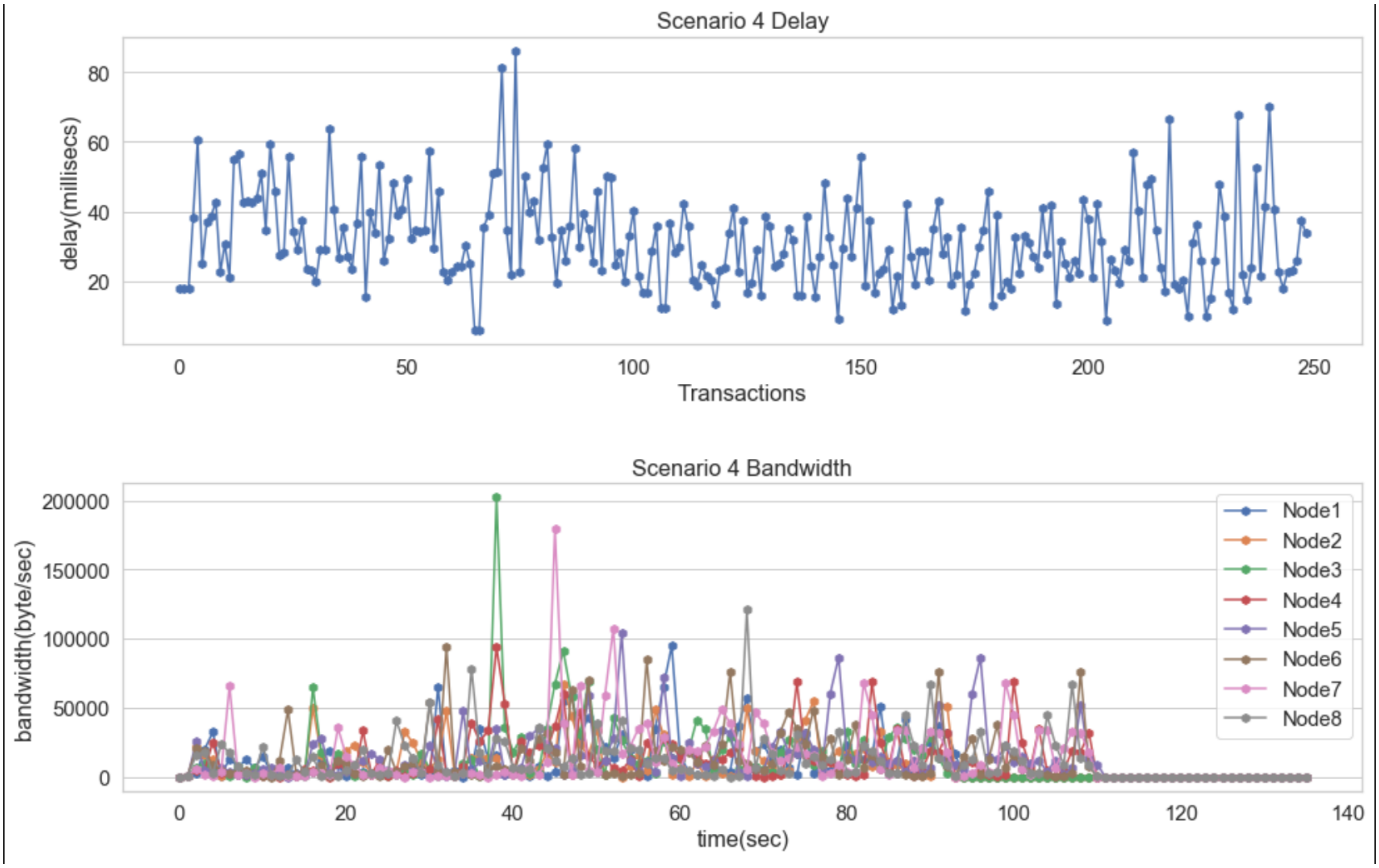

Figure 4: 8 nodes, 5 Hz each, running for 100 seconds, then one node fails, and the rest continue to run for 100 seconds