Riley Deal
CSCI 53700 Distributed Computing
Assignment 4

Overview

My implementation assignment 4 was mostly similar to assignment 1 on the Detection node side of things. The system consists of four Detection Nodes, each of which can communicate to the other Detection Nodes. The possible events and probabilities of each are listed in figure 1. Each second, data was gathered from every Detection Node, with each test run lasting 5 minutes. Three queues were used in each Detection Node, an output queue for messages which have been generated and are waiting to be sent, an input queue for messages which have been sent and are waiting to be received, and a work queue for messages which have been received which are waiting to be checked for anomalies. One type of byzantine behavior was implemented; if an anomaly is detected, immediate drop all messages within the work queue.

Event Type	Event Explanation
Send Message	1. Current Detection Node, Di, randomly selects another Detection Node, Dj 2. Remove a message from Di output queue 3. Place it in Dj input queue (Via RMI on Node stub)
Receive Message	1. Take message from Di input queue 2. Add it to Di work queue 3. Update logical clock if necessary
Generate Message	1. Add a message to Di output queue
Detect Anomaly	1. Remove an item from Di work queue 2. Check for anomaly (message value of 0) 3. Anomalies result in clearing of the work queue

Figure 1, event information

Interaction Model

The big differences from assignment 1 are in the Interaction Model. In the first assignment, each Detection Node had a list containing references to the other Detection Nodes, which meant they had direct access to each other. Since the Nodes are now distributed across multiple JVMs this is no longer possible without some kind of communication framework. For this we were tasked to use Java Remote Method Invocation (RMI). I took inspiration from how some peer-to-peer chat clients work by having clients "register" to a service, but communicate directly with each other once registered. This was implemented via running a Server process acting as a registration service. This server had 3 key objectives to fulfill, which was done via the methods listed below:

1. Clients can register to the service [**register(RemoteNode)**]
2. Clients can unregister, or disconnect from the service [**disconnect(RemoteNode)**]
3. Clients can query for everyone currently connected to the service [**getDetectionNodeList()**]

This approach meant that each Detection Node was required to acquire the server stub via a registry lookup, register a stub of itself with the server, and then request the stubs of the other connected Detection Nodes. The Detection Node stub only needs to expose one function in order to accept a message from another Node.

Failure Model

One very common failure with this design compared to the previous one is that nodes can drop out of the system or fail independently of one another. If another node has not updated its list of who it can send messages to, it can potentially be attempting to send messages to a node who no longer exists, which could result in failures. This kind of failure is currently handled by hiding it from the user, as it almost exclusively happens during the ending of the execution where one node is sending just as the other has shut down. The general case of this error could be alleviated to some extent by retrieving the current nodes from the server every time T, but would also be slow. Another failure point is if the server went down, so would the RMI registry, which would crash the entire system. I maintained the same byzantine failure as assignment one, in the event of an anomaly being detected, all messages in the work queue will be dropped. This was done in order to simulate a process omitting intended processing steps. In a network this issue could potentially be solved if you were expecting acknowledgement of the work done by the server to be sent back. If the client did not receive acknowledgement after a certain time frame the request could be resent.

Result Analysis

I will be focusing on the standard deviation between the Detection Node logical clocks for this part of the report. Figure 2 shows the standard deviation from project 1. The main takeaway from assignment 1 was the hypothesis that the standard deviation of the Detection Node logical clocks should remain stable. My reasoning was based on the math behind my original probabilities: 16.6% chance to generate a message, 33.3% chance to send a message, 33.3% chance to receive a message, and 16.6% chance to detect anomalies. This math essentially meant that sending a message was bounded by the number of messages generated (16.6%). Receiving a message was bounded by the chance of having a message sent to you (16.6 / 3 * 3 =

16.6%). This meant receiving a message and detect anomalies were evenly bounded. As a result of this there was an inability to keep up on messages generated, but saw the clocks sync up regularly. This syncing results in a stable standard deviation.

Unfortunately, the byzantine behavior used will not affect this statistic much since the clock time is already updated in receive, whereas the byzantine behavior is present during the detect anomalies stage.

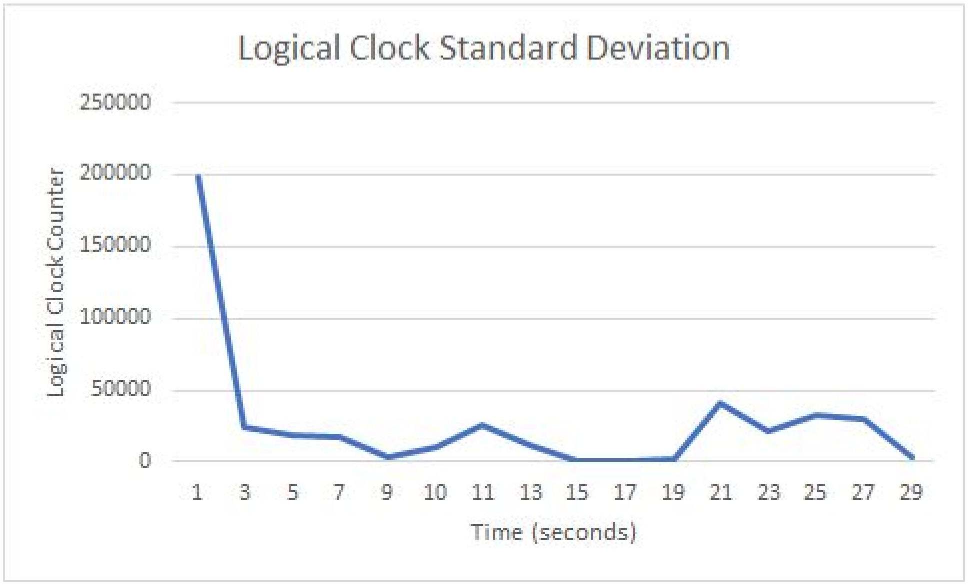


Figure 2, Assignment 1 Logical Clock Standard Deviation

The remainder of the graphs show 4 tests. Each ran for 5 minutes each, with the probabilities of the different events specified in the graph header.

Test 1 results are shown in Figure 3. The goal of test 1 was to prove that assignment 4 had similar results to assignment 1. Other than the deviation value being significantly higher, we see very similar results where the deviation quickly reaches and maintains stability.

Test 2 results are shown in figure 4. Since Test 1 further supported the result of being bound by the number of generated messages, the goal of Test 2 was to generate more messages, while still being bound by the generation step. The hypothesis is that there should still be a fairly stable deviation, but it should stabilize at a lower value since there are messages sent more often for clock updates. Since Figure 4 stabilized at a value ~2-4 million below Figure 3, I would say these results support the hypothesis.

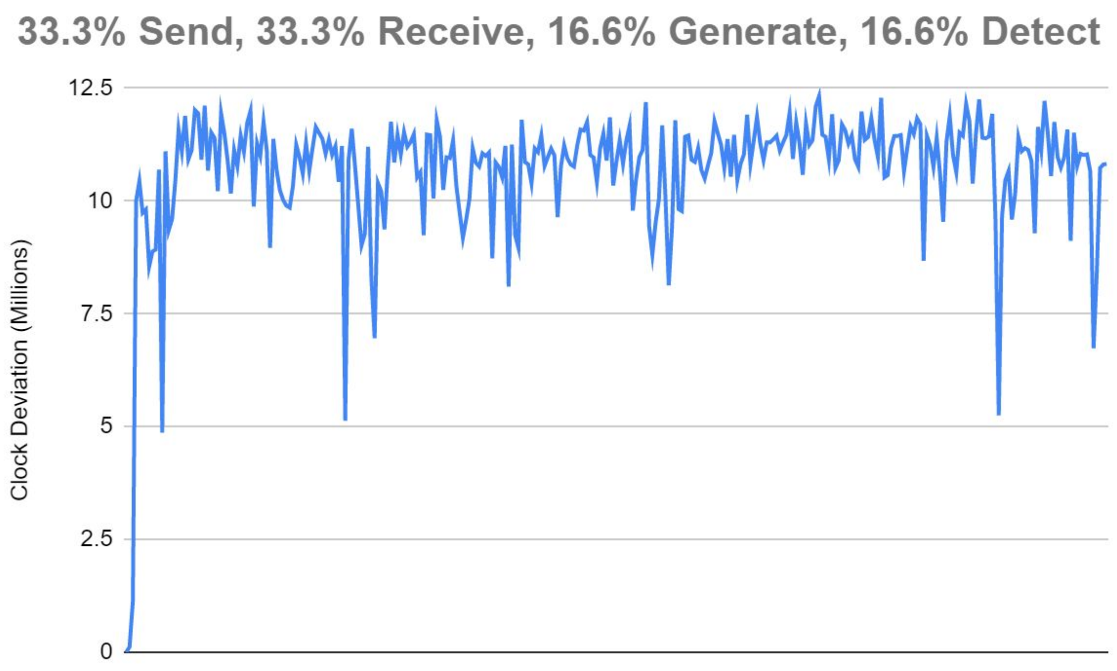


Figure 3, Assignment 4 Logical Clock Standard Deviation Test 1

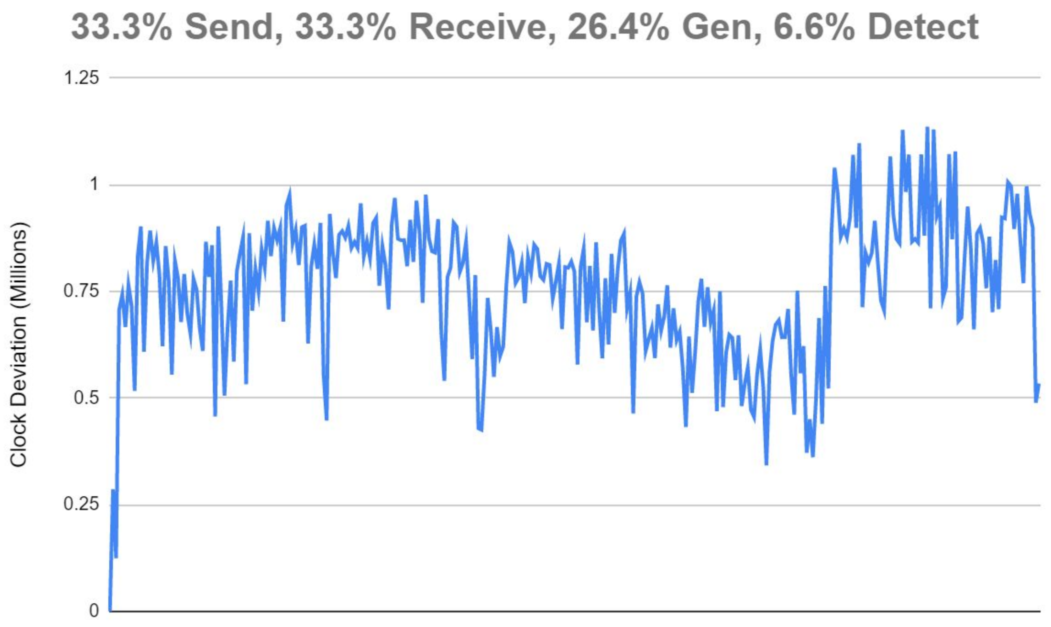


Figure 4, Assignment 4 Logical Clock Standard Deviation Test 2

For further testing, the next step was to place even more reliance on message generation by lowering the number of messages generated, but send even more. This was done in Test 3, as shown in Figure 5. The theory is clock drift should not happen as fast since we are sending significantly more frequently, even though generation happens slightly less. Due to this we see slightly more erratic deviation, which takes significantly longer to get to a stable point. I had originally expected higher deviation because the bounding is still on the message generation. I believe more tests need to be done with these values as its possible the beginning of the test was abnormal.

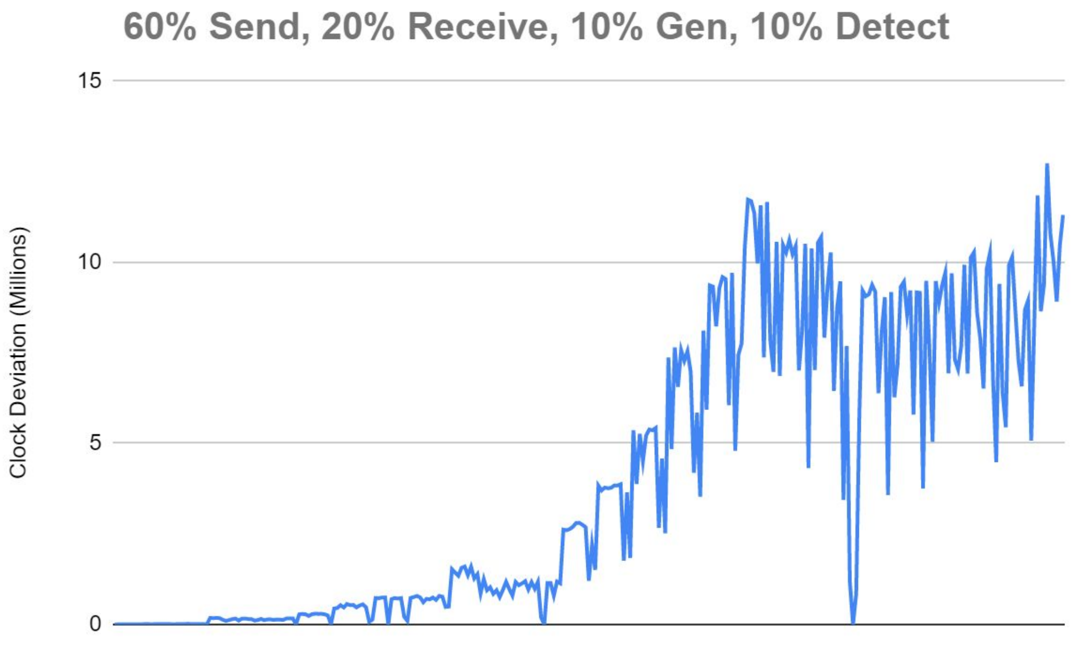


Figure 5, Assignment 4 Logical Clock Standard Deviation Test 3

Finally, I wanted to try a simulation which was bounded by something other than the message generation. In Test 4, I used numbers which should be bound on send/receive. My first test here seemed abnormal, as I had deviation values significantly higher than any test before (as shown in Figure 6); however, even after a second test with the same values we see similar results (as shown in Figure 7). This does make sense, as we are sending less messages than before which means the drift can potentially be much higher, and we do see the standard deviation continue to grow over time. In these tests, it's possible that 5 minutes was not enough to know if the graph had stabilized.

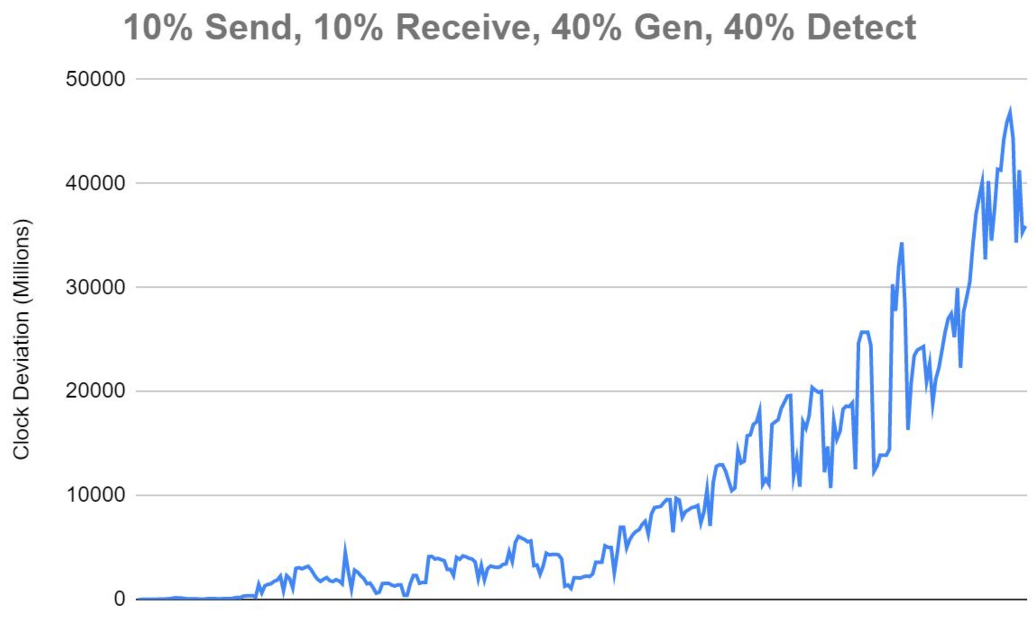


Figure 6, Assignment 4 Logical Clock Standard Deviation Test 4 Part 1

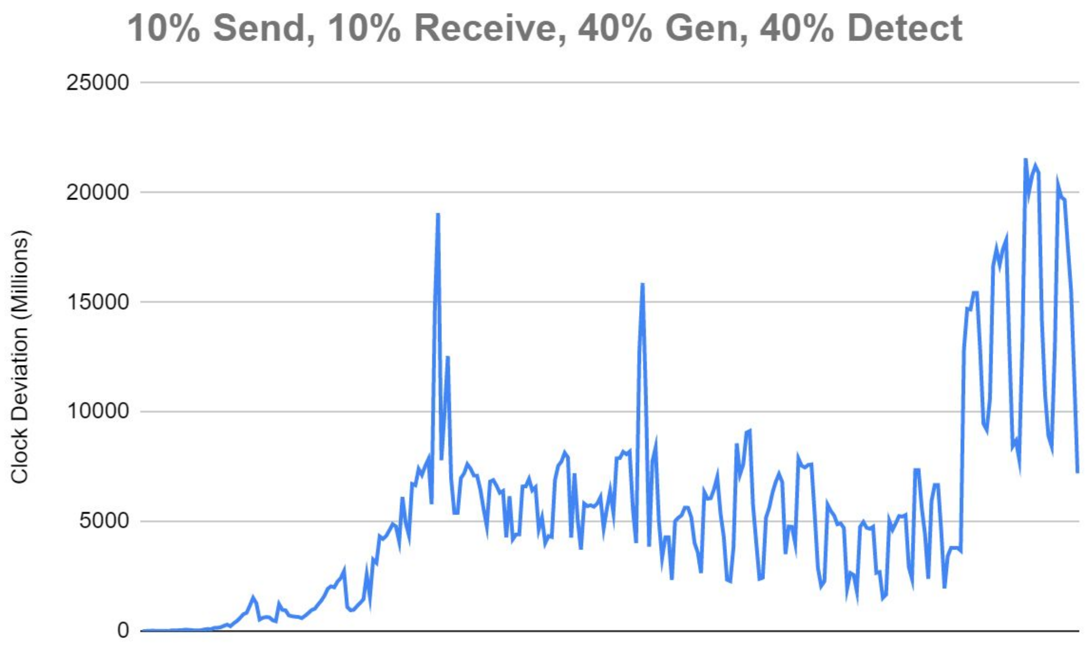
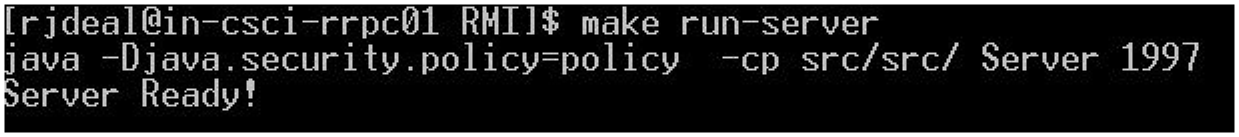
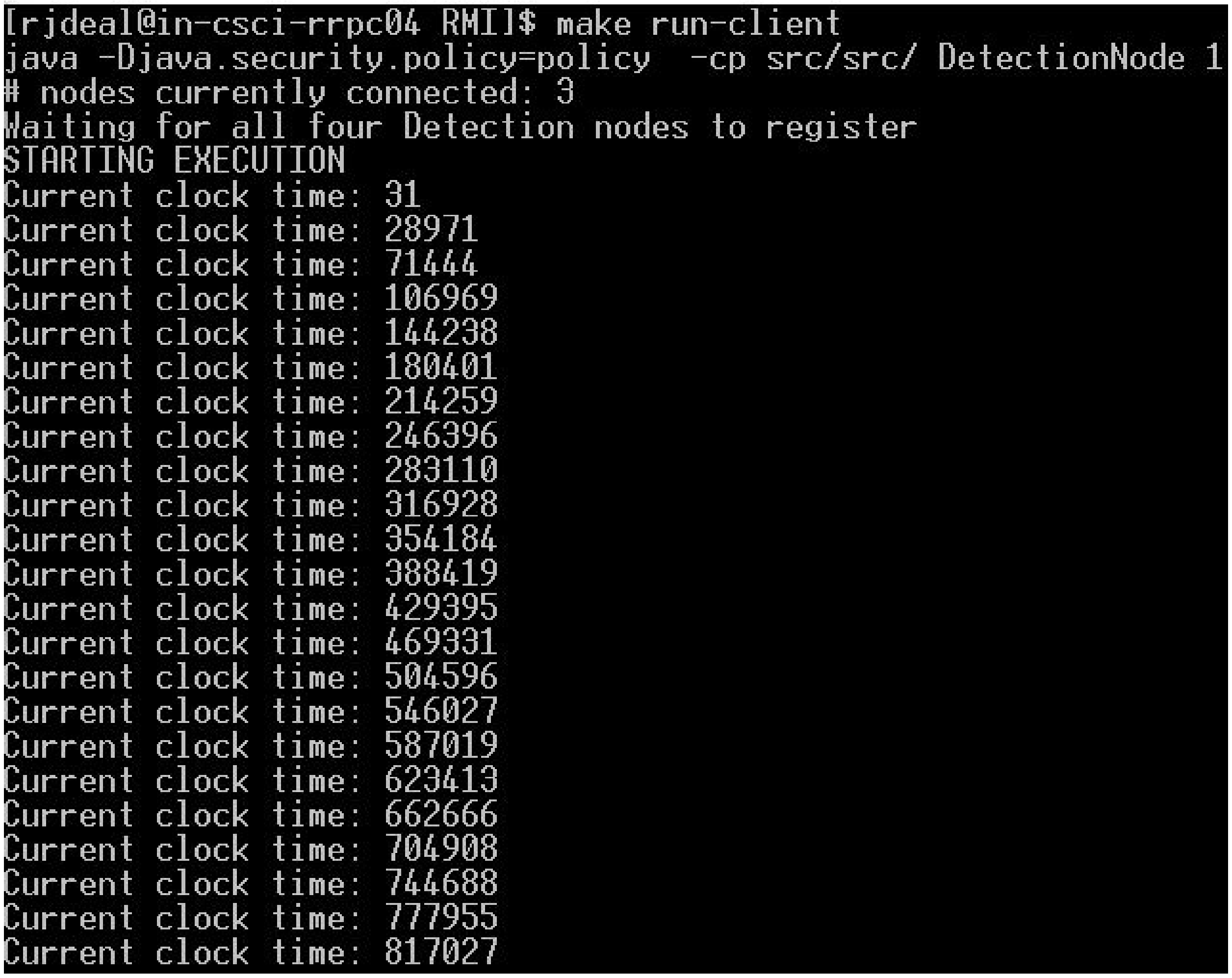


Figure 7, Assignment 4 Logical Clock Standard Deviation Test 4 Part 2

Execution



```
[rjdeal@in-csci-rrpc01 RMI]$ make run-server
java -Djava.security.policy=policy  -cp src/src/ Server 1997
Server Ready!
```



```
[rjdeal@in-csci-rrpc04 RMI]$ make run-client
java -Djava.security.policy=policy  -cp src/src/ DetectionNode 1
# nodes currently connected: 3
Waiting for all four Detection nodes to register
STARTING EXECUTION
Current clock time: 31
Current clock time: 28971
Current clock time: 71444
Current clock time: 106969
Current clock time: 144238
Current clock time: 180401
Current clock time: 214259
Current clock time: 246396
Current clock time: 283110
Current clock time: 316928
Current clock time: 354184
Current clock time: 388419
Current clock time: 429395
Current clock time: 469331
Current clock time: 504596
Current clock time: 546027
Current clock time: 587019
Current clock time: 623413
Current clock time: 662666
Current clock time: 704908
Current clock time: 744688
Current clock time: 777955
Current clock time: 817027
```