MP0 - Event Logging

Names:

Ali Husain (alijh2), Satej Sukthankar (satejrs2)

Cluster Numbers:

4201 - 4210

Git URL:

https://gitlab.engr.illinois.edu/cs4251732105/mp1-event-ordering

Git Revision Number:

6acf8c8694e801422544dd44177c37e165f15c25

Design Protocol

Our design ensures total ordering and reliable multicast under any number of node failures. Our design utilizes the ISIS algorithm to ensure total ordering. Our design also ensures that each transaction is delivered to each node at most once, any multicast message by a node wil also be delivered to the sending node, and if a correct process delivers a message, then all other correct processes will also deliver that message.

We start by starting up all the nodes in our system. Once all the nodes are running, you can enter START into any node. This will establish a connection with each node in the configuration file and begin generating transactions on that specific node. We don't need all the nodes to begin generating transactions at once, since the ISIS algorithm only requires the transaction generating node to multicast to the rest of the system, and the nodes in the system to unicast back to the original sender. The user must go to each node and type in START for that node to begin generating transactions and multicasting to the rest of the system. Upon typing START. the handleConfiguration function initiates a connection with any remaining unconnected nodes in the configuration file. We store information about all the active node connections in a global nodes variable.

Once handleConfiguration is completed, we call the generateTransactions function. This function runs gentx.py and pipelines it's output into a local buffer. We read each written into the buffer and create a transaction object which includes a unique transactionId, the original sender of the transaction, it's priority, the state of the messages deliverability, the message type, and a timestamp. When a transaction is generated, we start the proposedPriorityFinaliser go routine. This routine see's the entire lifecycle of the transaction. It multicasts the transactions to all live nodes including itself, awaits the proposed priorities from all the nodes, finalizes the priority, and then multicasts the final priority to all live nodes, including itself. This function ensures that any multicast message by a node will also be delivered to the sending node.

When a node initiates a connection or receives an incoming connection, the dispatchTransactions go routine will be called. This routine is unique to each connection, and will properly handle where a transaction goes at any step of the ISIS algorithm. Each message can either be "init", "proposed", or "final". "Init" indicates that the sender is requesting a proposed priority from the recipient and should add this transaction to the recipient's priority queue. "Proposed" indicates that a node is sending the original sender it's proposed priority. "Final" indicates that this is the final, agreed upon priority and that the transaction can be delivered. When this function receives a transaction marked with "Init" it will change the priority on the transaction to it's proposed priority, mark the state of the transaction as "proposed" and unicast it back to the original sender. When this function receives a transaction marked as "proposed" it will dispatch the transaction to the appropriate proposedPriorityFinaliser function that is handling this transaction. When this function receives a transaction marked as "final," it will deliver the message, and remove the message from the priority queue.

Transactions are only added to a nodes priority queue when a transaction is generated by a sender or received with the “init” message tag. Each node will do either of these events, only once. Transactions are only removed from the priority queue upon delivery. This ensures that each transaction is delivered at most once.

There are a handful of cases with failing nodes. 1) A process multicasts a final priority and then dies before delivering to itself. In this situation, there is no additional overhead to implement as all live nodes will still receive and subsequently deliver the transaction. 2) A sender is waiting for a proposed priority from a dead node. In this situation, we detect a broken pipe in dispatchConnections and call the handleFailure function. This function will enact all necessary cleanup for this node and allow the final priority for this transaction to be determined without the proposed priority of the failed node. We can then proceed with multicasting the transaction with the final priority. 3) A node is waiting for the final priority from a dead node. This failure is handled in processTransactions, where it compares the timestamp of the top most node (when it received the initial transaction message) to the current time and, if it’s been more than 5 seconds, it removes it from the queue. We assume that, if the dead node couldn’t multicast it to this node, it didn’t multicast to the rest of the nodes.

Getting started

Starting node

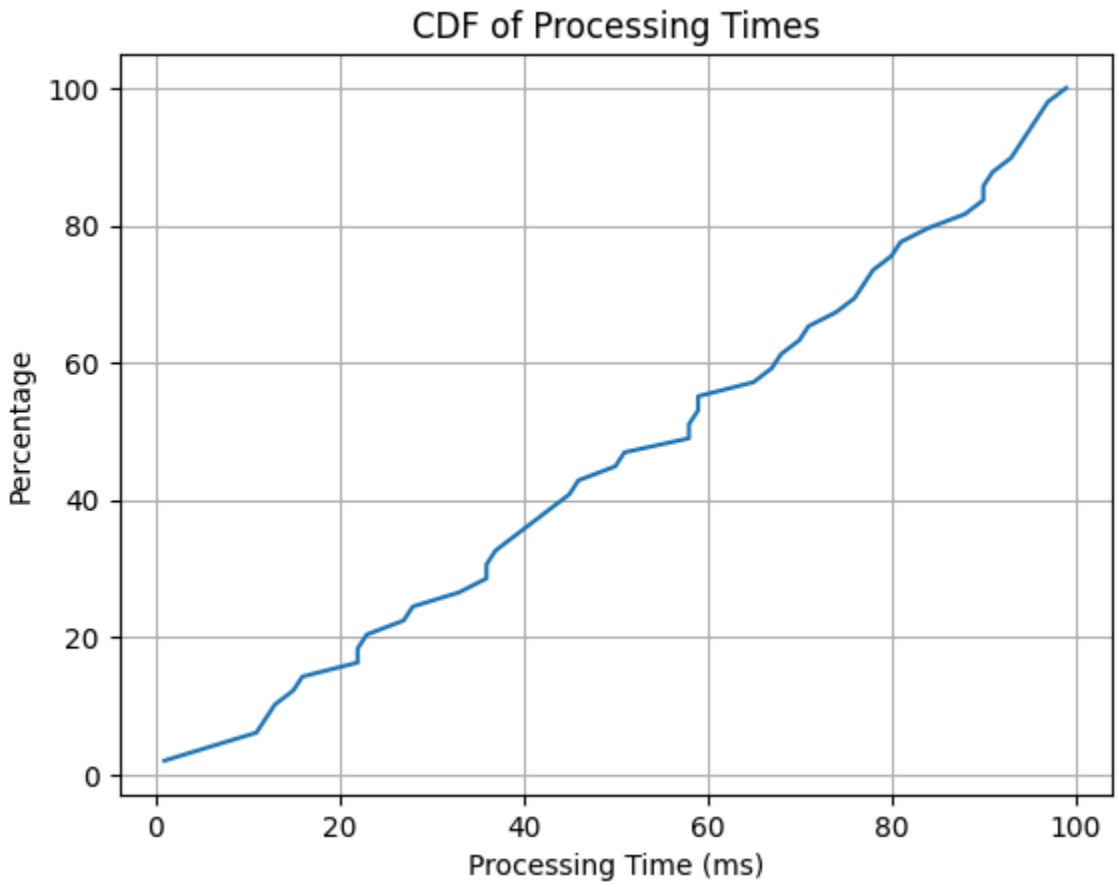
1. `make build` to compile the node.go file.
2. `go run node.go {node_name} {config.txt} {frequency}` to start TCP connections across nodes with the desired frequencies. `3_node_config.txt` can be used to test 3 nodes, 0.5 Hz each, running for 100 seconds, and `8_node_config.txt` can be used to test 8 nodes, 5 Hz each, running for 100 seconds
3. Once all node connections are made, the phrase “`START`” has to be typed into each individual node’s terminal to start generating transactions from each node.
4. Balances will start printing to standard output in all connected node terminals.

Conducting Analytics

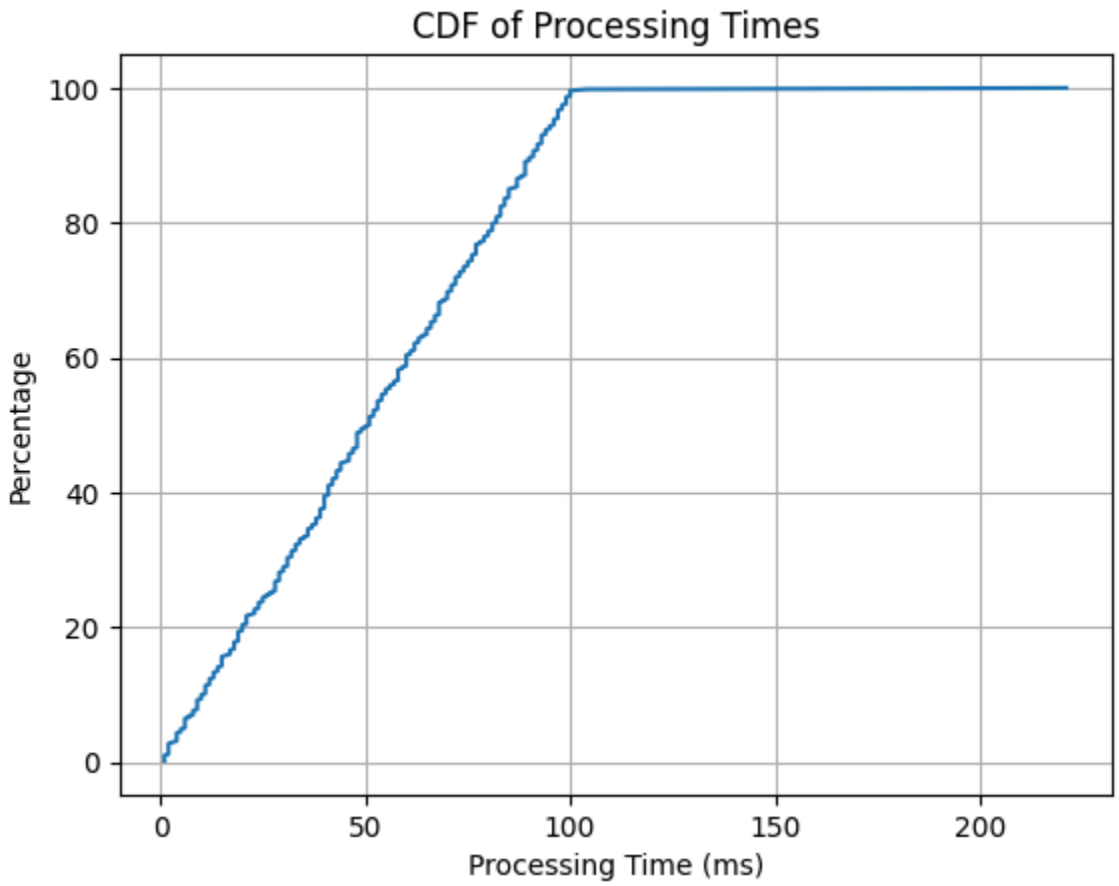
Ensure you have `matplotlib`*,* and `numpy` *installed before running analytics.*

1. Upon terminating the all node connections, a `processing_times.txt` file will be created in the current directory.
2. Run the code block within `analysis.ipynb` after running one of the 4 desired scenarios for their corresponding analytic.
3. A graph will be generated depicting the appropriate cumulative distribution function of transaction processing time.

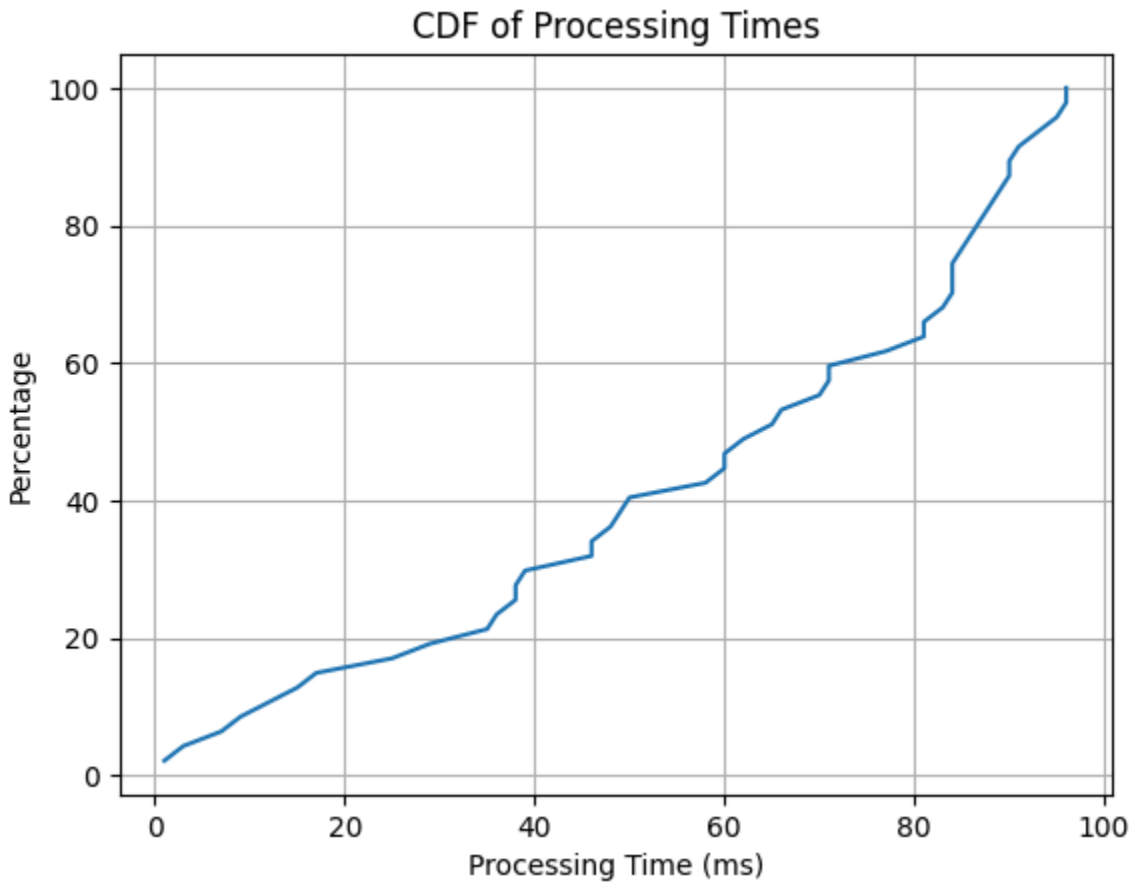
Test 1:


CDF of Processing Times
Percentage
Processing Time (ms)
0
20
40
60
80
100

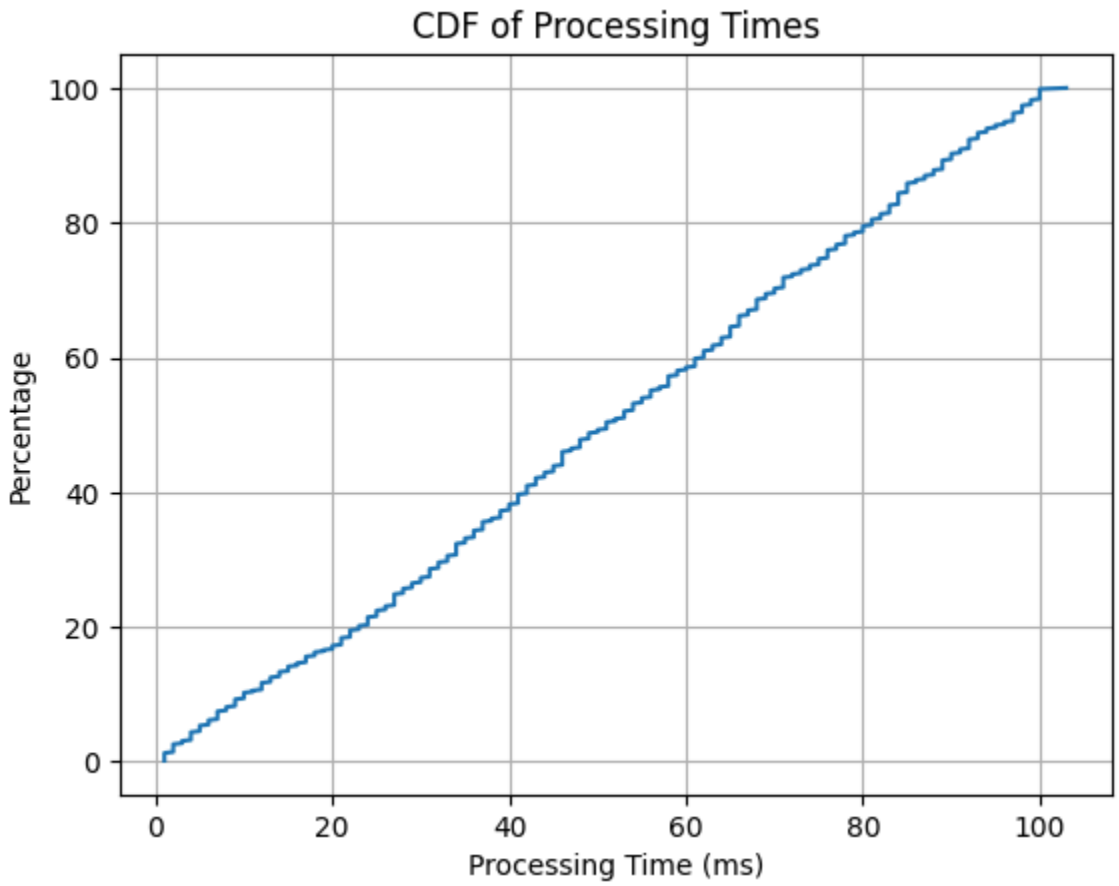
Test 2:


CDF of Processing Times
Percentage
Processing Time (ms)
0
20
40
60
80
100
0
50
100
150
200

Test 3:


CDF of Processing Times
Percentage
Processing Time (ms)
0
20
40
60
80
100

Test 4:


CDF of Processing Times
Percentage
Processing Time (ms)
0
20
40
60
80
100