Carleton University

SYSC5104W – Methodologies for Discrete Event Modeling

Assignment #1

March 7th, 2025

Author	Student Number	Email
Paul Desaulniers	101174195	pauldesaulniers@cmail.carleton.ca

Table of Contents

Conceptual model Design:

System Components:

- Nodes: A set of participating entities in the consensus protocol.
- RAFT: An algorithm that selects a leader node amongst of set of nodes in a network and gathers information from peering nodes to form a consensus.
- Network: An infrastructure that facilitates direct communication and message exchange between devices, including peer-to-peer interactions.

Model Structure:

Atomic Models

- Message Parser: The parsing logic for directing the messages to the proper models.
- RAFT: Implements the RAFT consensus logic.
- Network: Implements a simple networking delay and packet forwarding mechanism.
- Buffer: Stores inbound messages, and dispatches them to the atomic model
- PacketMaker: Creates packets that the network atomic model can interface with.

Coupled Models

- Raft: A tightly integrated model that combines buffering mechanisms with Raft consensus logic, ensuring efficient state management while maintaining separation of concerns.
- Node: A strategic combination of MessageProcesser, Raft, and PacketProcesser that enables node-specific tasks.
- Simulation: An aggregation of n-node models, one network model, and one Client iestream model for event injection. Nodes have a bidirectional connection to the network, while the client also maintains a bidirectional connection with the network. This structure allows the simulation to be extendable by adding n-nodes while maintaining consistent logic across other atomic components.

Transitions & Events:

- RAFT: Nodes transition between follower, candidate, and leader states according to the RAFT consensus protocol.
- Network: Introduces delays and forwards messages to the appropriate nodes.
- Client: Generates events based on a predefined file and sends them to the network, enabling interaction with the RAFT consensus

Assumptions & Simplifications:

- Cryptographic aspects of time-signing are abstracted away for now.
- Network latency and message loss are not explicitly modeled in this phase.

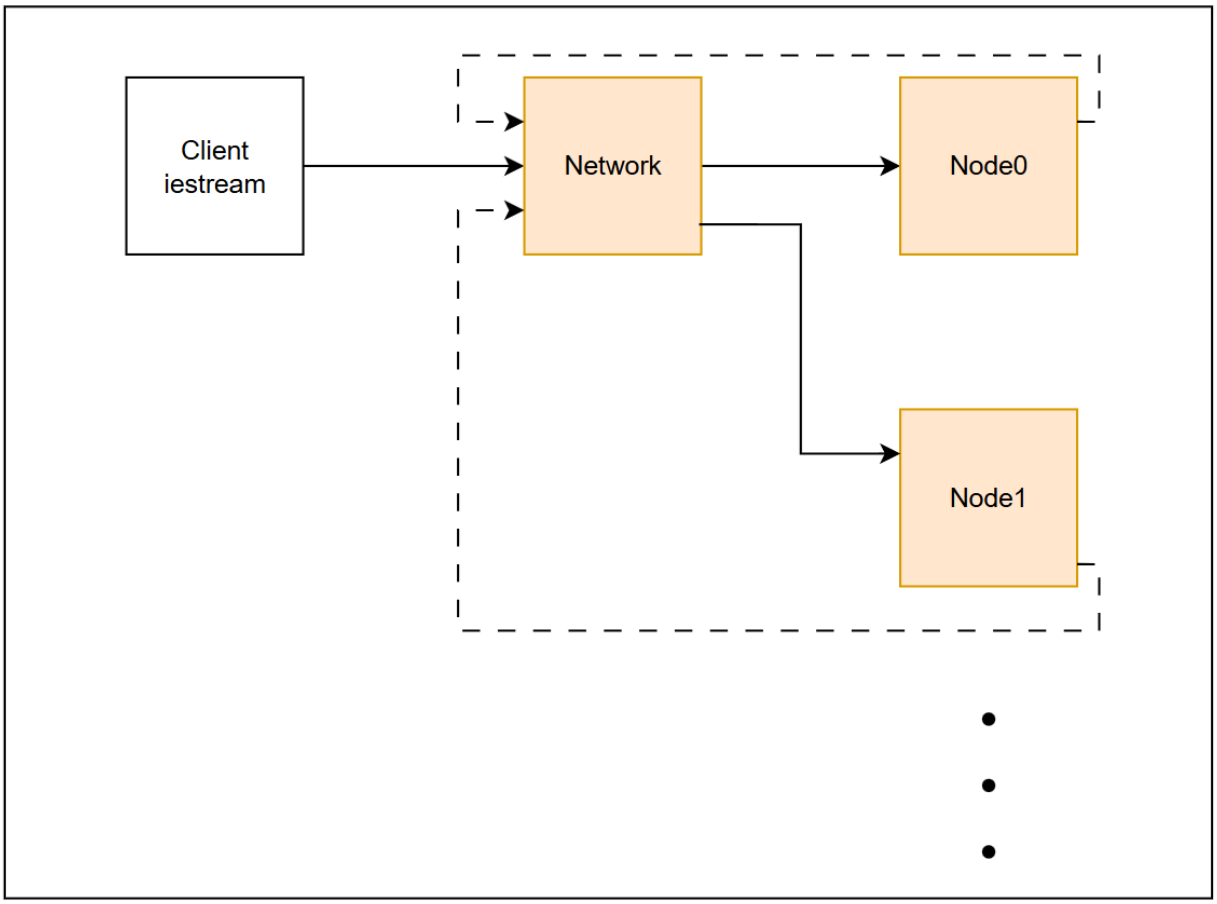
Introduction

This assignment focuses on the RAFT protocol, which is broken into three components: the Buffer, RaftController, and HeartbeatController. Future assignments will extend this work by building the remaining components of the node. The primary objective of this assignment is to develop a RAFT model.

The application logic will be implemented in the final project, with more details to come. For now, the node's coupled model will include a RAFT coupled model, a Packet Processor atomic model, and a Message Processor atomic model, which interface with the external environment. The Packet Processor encapsulates message logic and interacts with the network atomic model, ensuring that the network model focuses solely on processing rather than handling message types. Similarly, the Message Processor atomic model receives packets and dispatches messages to the appropriate entity.

GitHub Repository: https://github.com/PDesa16/RAFT-DEVS.git

Simulation - the topmost coupled model



$$< X,\ Y,\ D,\ \{M_i\},\ EIC,\ \ EOC\ ,\ IC,\ SELECT >_{\text{Consensus Simulation}}$$

$$X = \{None\}$$

$$Y = \{None\}$$

$$EIC = \{None\}$$

$$EOC = \{None\}$$

$$IC\ = \begin{Bmatrix} \text{Node}_0.\,\text{output\_network}\ ,\ \text{Network.\ input\_packet} \\ \dots \\ \text{Node}_n.\,\text{output\_network}\ ,\ \text{Network.\ input\_packet} \\ \text{Network.\ output\_packet},\ \text{Node}_0.\,\text{input\_network} \\ \dots \\ \text{Network.\ output\_packet},\ \text{Node}_n.\,\text{input\_network} \\ \text{Client.\ output\_network},\ \text{Network.\ input\_packet} \\ \text{Network.\ output\_packet},\ \text{Client.\ input\_network} \end{Bmatrix}$$

$$D\ =\ \{\text{Node}_0 \dots \text{Node}_n\ ,\ \text{Network},\ \text{Client}\}$$

$$M_i\ =\ \begin{Bmatrix} \langle X, Y, D, M_i, EIC, EOC, IC, SELECT\rangle_{\text{Node}_0} \\ \dots \\ \langle X, Y, D, M_i, EIC, EOC, IC, SELECT\rangle_{\text{Node}_n} \\ \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{Network}} \\ \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{Client}} \end{Bmatrix}$$

$$SEL\ =\ \begin{Bmatrix} \text{Network} & \text{if Network}\ \in\ \text{SEL} \\ \text{Client} & \text{if Client}\ \in\ \text{SEL and Network}\ \in/\ \text{SEL} \\ \text{Min}(\text{Node}_i \mid \text{Node}_i\ \in\ \text{SEL}) & \text{otherwise} \end{Bmatrix}$$

Network model

$$\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{Network}}$$

$$S\ =\begin{Bmatrix} \text{packetEventQueue}\ \in\ \text{PE} \\ \text{currentTime}\ \in\ \mathbb{R}^+ \\ activeNodes\ \in\ \text{NODES} \end{Bmatrix}$$

- PE: A priority queue containing packet events.
  - A packet events = (packet ∈ Packet, delay ∈ $\mathbb{R}^+$, timestamp ∈ $\mathbb{R}^+$)
- NODES: A set of active network nodes.

$$X = \{packet \mid packet \in Packet\}$$

$$Y = \{packet \mid packet \in Packet\}$$

A packet is defined as:

$$Packet = (Src, Dest, Payload)$$

$$Src \in \Sigma^*, \quad Dest \in \Sigma^*, \quad Payload \in M$$

- Σ^* represents the entire string space.
- $M = \{msg \mid msg \text{ implements IMessageInterface}\}$

The packet subspace is then:

$$Packet \subseteq \Sigma^* \times \Sigma^* \times M$$

$$\delta_{int}(s) = \begin{cases} packetEventQueue.pop() & \text{if } packetEventQueue \neq \emptyset \\ s & \text{otherwise} \end{cases}$$

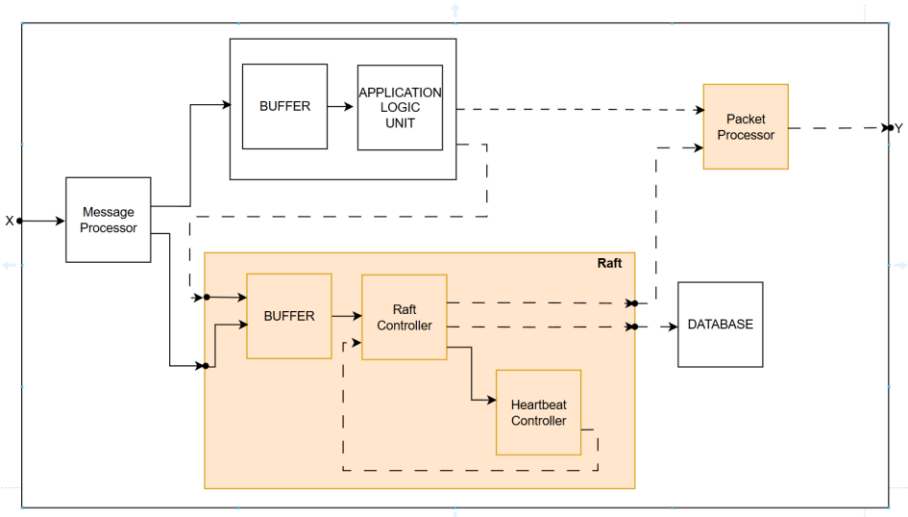
$$\lambda(s) = \begin{cases} Network.out \leftarrow packetEventQueue.top() \rightarrow Packet & \text{if } packetQueue \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$ta(s) = \begin{cases} packetEventQueue.top().delay & \text{if } packetEventQueue \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

$$\delta_{ext}(s, e) = \begin{cases} s.currentTime \leftarrow s.currentTime + e, \\ s.packetEventQueue \leftarrow Network.in \end{cases}$$

Node Model

$$< X, Y, D, Mi, EIC, EOC, IC, SELECT >_{Node}$$



$$X = \{packet \mid packet \in Packet\}$$

$$Y = \{packet \mid packet \in Packet\}$$

$$EIC = \{Node.in, MessageProcessor.in\_packet\}$$

$$EOC = \{PacketProcessor.output\_packet, Node.out\}$$

$$IC = \begin{Bmatrix} MessageProcessor.output\_raft, Raft.input\_external \\ raft.output\_external, PacketProcessor.input\_raft \end{Bmatrix}$$

$$D = \{MessageProcessor, Raft, PacketProcessor\}$$

$$M_i = \begin{Bmatrix} \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle_{MessageProcessor} \\ \langle X, Y, D, M_i, EIC, EOC, IC, SELECT \rangle_{Raft} \\ \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle_{PacketProcessor} \end{Bmatrix}$$

$$SEL = \begin{Bmatrix} MessageProcessor & \text{if } MessageProcessor \in SEL \\ Raft & \text{if } Raft \in SEL \text{ and } MessageProcessor \notin SEL \\ PacketProcessor & \text{otherwise} \end{Bmatrix}$$

A Brief Overview of the RAFT Protocol

RAFT is a distributed consensus protocol that ensures a consistent log across nodes. However, it is not fully Byzantine fault-tolerant, meaning that if a node is actively malicious, RAFT does not provide a recovery mechanism. Nonetheless, it offers fault tolerance by dynamically electing a new leader in the event of failures. Alternative protocols, such as PBFT, provide Byzantine fault tolerance, but for the purposes of this experiment, RAFT offers sufficient guarantees.

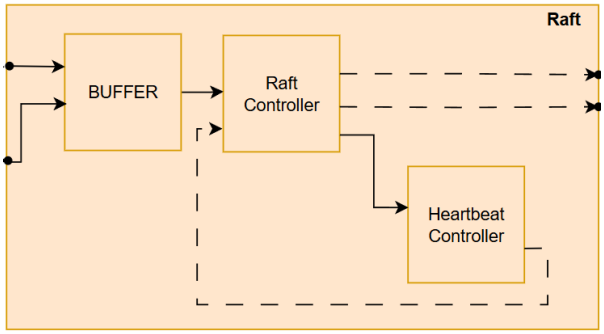
Main Concepts

Nodes in RAFT can be in one of three states: Leader, Candidate, or Follower. A node becomes a Candidate when it does not receive a heartbeat message from the Leader. To prevent all nodes from becoming Candidates simultaneously, each node, upon startup or upon receiving a heartbeat message, selects a timeout value from a uniform distribution ranging from 150ms to 300ms.

A Candidate becomes a Leader when it receives votes from at least n/2 nodes (where n is the total number of nodes). Nodes that are neither Candidates nor Leaders remain Followers, replicating the Leader's log entries. The Leader interfaces with clients and serves as the primary driver for Follower nodes. This is a high-level overview of the RAFT protocol.

One of the advantages of RAFT is its simplicity, which allows for easy integration of application-specific logic. In my final project, I will implement a custom consensus protocol and use it to perform blockchain operations. The objective is to achieve higher throughput and real-time blocking, targeting a latency range of 100ms to 300ms.

Raft Model



$$< X,\ Y,\ D,\ \{M_i\},\ EIC,\ \ EOC\ ,\ IC,\ SELECT >_{\text{Buffer}-\text{RAFT}}$$

$$X = \{msg \mid msg\ \in\ RaftMessage\}$$

$$Y = \{msg \mid msg\ \in RaftMessage\}$$

$$EIC = \{\ Raft.\,in, Buffer.\,in\ \}$$

$$EOC = \{RaftController.\,out,\qquad Raft.\,out\}$$

$$IC\ =\ \begin{Bmatrix} Buffer.\,out, RaftController.\,in \\ RaftController.\,out\_heartbeat, HeartbeatController.\,in \\ HeartbeatController.\,out, RaftController.\,in\_heartbeat \end{Bmatrix}$$

$$D\ =\ \{Buffer, RaftController, HeartbeatController\}$$

$$M_i\ =\ \begin{Bmatrix} \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{HeartbeatController}} \\ \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{Buffer}} \\ \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta\rangle_{\text{RaftController}} \end{Bmatrix}$$

$$SEL\ =\ \begin{Bmatrix} RAFT & \text{if } RAFT\ \in\ SEL \\ BUFFER & \text{otherwise} \end{Bmatrix}$$

$$SEL\ =\ \begin{Bmatrix} HeartbeatController & \text{if } eartbeatController\ \in\ SEL \\ RaftController & \text{if RaftController}\ \in\ SEL \text{ and } HeartbeatController\ \in/\ SEL \\ Buffer & \text{otherwise} \end{Bmatrix}$$

Buffer Atomic model

$$\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle_{\text{BUFFER}}$$

$$S = \begin{Bmatrix} \text{buffer} \in \mathbb{Q} \\ \text{state} \in \{\text{IDLE}, \text{BUSY}\} \end{Bmatrix}$$

$$X = \{msg \mid msg \in \text{RaftMessage}\}$$

$$Y = \{msg \mid msg \in \text{RaftMessage}\}$$

$$\delta_{int}(s) = \begin{Bmatrix} (s.\text{buffer} - \{s.\text{buffer.front}()\}, \text{BUSY}) & \text{if } |\text{buffer}| > 1 \\ (\emptyset, \text{IDLE}) & \text{otherwise} \end{Bmatrix}$$

$$\delta_{ext}(s, e) = (s.\text{buffer} \cup \{m \mid m \in \text{Buffer.input\_port}\}, \text{BUSY})$$

$$\lambda(s) = \begin{Bmatrix} \text{Buffer.out\_port} \leftarrow s.\text{buffer.front}() & \text{if BUSY} \\ \text{Buffer.out\_port} \leftarrow \emptyset & \text{otherwise} \end{Bmatrix}$$

$$ta(s) = \begin{Bmatrix} 0.05 & \text{if BUSY} \\ \infty & \text{otherwise} \end{Bmatrix}$$

HeartbeatController Atomic model

$$\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle_{\text{HeartbeatController}}$$

$$S = \begin{Bmatrix} \text{status} \in \{\text{ALIVE}, \text{TIMEOUT}, \text{UPDATE}\} \\ \text{heartbeatTimeout} \in \mathbb{R}^+ \cup \{\infty\} \end{Bmatrix}$$

$$X = \{\text{status} \mid \text{status} \in \{\text{ALIVE}, \text{TIMEOUT}, \text{UPDATE}\}\}$$

$$Y = \{\text{status} \mid \text{status} \in \{\text{ALIVE}, \text{TIMEOUT}, \text{UPDATE}\}\}$$

$$\delta_{int}(s) = \begin{Bmatrix} \text{heartbeatTimeout} \leftarrow \infty & \text{if status} \neq \text{UPDATE} \\ \text{heartbeatTimeout} \leftarrow 0.05 & \text{if status} = \text{UPDATE} \end{Bmatrix}$$

$$\delta_{ext}(s, e) = \begin{Bmatrix} \text{status} \leftarrow \text{HeartbeatController.in}, & \text{if status} = \text{ALIVE} \\ heartbeatTimeout \leftarrow \sim U(0.150, 0.300), & \text{if status} = \text{UPDATE} \\ heartbeatTimeout \leftarrow 0.05 & \end{Bmatrix}$$

$$\lambda(s) = \begin{Bmatrix} \text{TIMEOUT} & \text{iif status} = \text{ALIVE} \\ \text{UPDATE} & \text{if status} = \text{UPDATE} \end{Bmatrix}$$

$$ta(s) = heartbeatTimeout$$

RaftController Atomic Model

$$\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle_{\text{RaftController}}$$

$$X = \{msg \mid msg \in \text{RaftMessage}\}$$

$$Y = \{packet \mid packet \in \text{Packet}\}$$

$$S_{\text{RaftController}} = \left\{ \begin{array}{c} \text{raftState} \in \{\text{FOLLOWER}, \text{CANDIDATE}, \text{LEADER}\} \\ \text{currentTerm} \in \mathbb{Z}^+ \\ \text{votedStatus} \in \{\text{VOTE\_NOT\_YET\_SUBMITTED}, \text{VOTED}\} \\ \text{commitIndex, lastApplied, logIndex} \in \mathbb{Z}^+ \\ \text{heartbeatTimeout, currentTime} \in \mathbb{R}^+ \\ \text{privateKey} \in S \text{ (String representation)} \\ \text{publicKeys} \in PK \text{ (Set of all public keys)} \\ \text{peers} \in PS \text{ (Set of all peers in the cluster)} \\ \text{electionTimeout} \in \mathbb{R}^+ \\ \text{leaderID} \in S \text{ (string identifier)} \\ \text{messageLog} = [\text{LogEntry}_1, \text{LogEntry}_2, \dots, \text{LogEntry}_n]^1 \\ \text{tempMessageStorage} \in Q \text{ (Queue)} \\ \text{databaseOutMessages} \in Q \\ \text{raftOutMessages} \in Q \end{array} \right\}$$

$\text{LogEntry}_i{}^1 \in L$, where L represents the set of possible log entries.

$$\delta_{int}(s) = \begin{cases} s.\text{heartbeatStatus} \leftarrow \text{ALIVE}, \\ s.databaseOutMessages \leftarrow \emptyset \\ s.\text{raft}OutMessages \leftarrow \emptyset \end{cases}$$

$$\delta_{ext}(s, e) = \begin{cases} \forall x \in \text{in\_port} & \text{apply } f(s, x) \\ s & \text{otherwise} \end{cases}$$

$$f(s, x) = \left\{ \begin{array}{cc} \text{HandleRequest}(s, x) & \text{if } x.\text{getType}() = \text{VOTE\_REQUEST} \\ \text{HandleResponse}(s, x) & \text{if } x.\text{getType}() = \text{VOTE\_RESPONSE} \\ \text{HandleAppendEntries}(s, x) & \text{if } x.\text{getType}() = \text{APPEND\_ENTRIES} \end{array} \right\}$$

HandleRequest(s,x)

```
Function HandleRequest(s: RaftState, requestMessage: RequestVote)
  voteGranted ← (requestMessage.termNumber > s.currentTerm) OR
  (requestMessage.termNumber == s.currentTerm AND s.votedStatus == VOTE_NOT_YET_SUBMITTED)

  responseMetadata ← CreateResponseMetadata(requestMessage, voteGranted, id)
  signedDigest ← SignData(responseMetadata, s.privateKey)

  response ← New ResponseVote(responseMetadata, signedDigest)
  raftMessage ← New RaftMessage(response)

  Append s.raftOutMessages with raftMessage
End Function
```

HandleResponse(s,x)

```
Function HandleResponse(s: RaftState, responseMessage: ResponseVote)
  If responseMessage.voteGranted == true AND responseMessage valid Then
    Append responseMessage to s.tempMessageStorage
  End If
End Function
```

HandleAppendEntries(s,x)

```
Function HandleAppendEntries(s: RaftState, msg: AppendEntries)
  If msg.term < s.currentTerm Then Return

  For each entry in msg.entries:
    If entry.type == RAFT AND entry is valid
      Then s.leaderID = msg.leaderID
      Append entry to s.messageLog
    Else If entry.type == HEARTBEAT AND msg.leaderID is valid Then
      Append entry to s.messageLog
      s.lastHeartbeatUpdate = s.currentTime
End Function
```

$$\lambda(s) = \begin{cases} \text{RAFT.out\_database} \leftarrow \{msg \mid msg \in s.\text{databaseOutMessage}\} \\ \text{RAFT.out\_raftMessage} \leftarrow \{msg \mid msg \in s.\text{raftOutMessage}\} \\ \text{RAFT.out\_raftMessage} \leftarrow \emptyset \\ \text{RAFT.out\_database} \leftarrow \emptyset \end{cases} \quad \begin{array}{c} \text{if } s.\text{databaseOutMessage} \neq \emptyset \text{ or } s.\text{raftOutMessage} \neq \emptyset \\ \text{otherwise} \end{array}$$

$$ta(s) = \begin{cases} \left(\sum_{msg \in s.\text{raftOutMessage}} -\left(\frac{\ln(U_{msg})}{\lambda_{msg}}\right), U_{msg} \sim Uniform(0,1) \quad \begin{array}{c} \lambda = 10^4 \text{ if VOTE\_REQUEST} \\ \lambda = 10^4 \text{ if VOTE\_RESPONSE} \\ \lambda = 10^6 \text{ if APPEND\_ENTRIES} \end{array} \right) & \text{if } s.\text{raftOutMessage} = \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Reasoning:

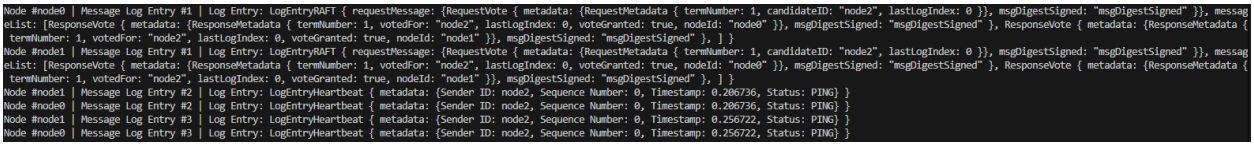
The expected value of an exponential distribution is given by:

Logic: $E[X] = \frac{1}{\lambda}$

Therefore, we can choose a mean that makes sense for the workload. In future iterations of the simulation, all these parameters will be specified in simulation_config.yaml. Since APPEND_ENTRIES involves more processing, an average time of 100 µs is used for λ. For VOTE_REQUEST and VOTE_RESPONSE, which require fewer computations, an average time of 10 µs is used for λ.

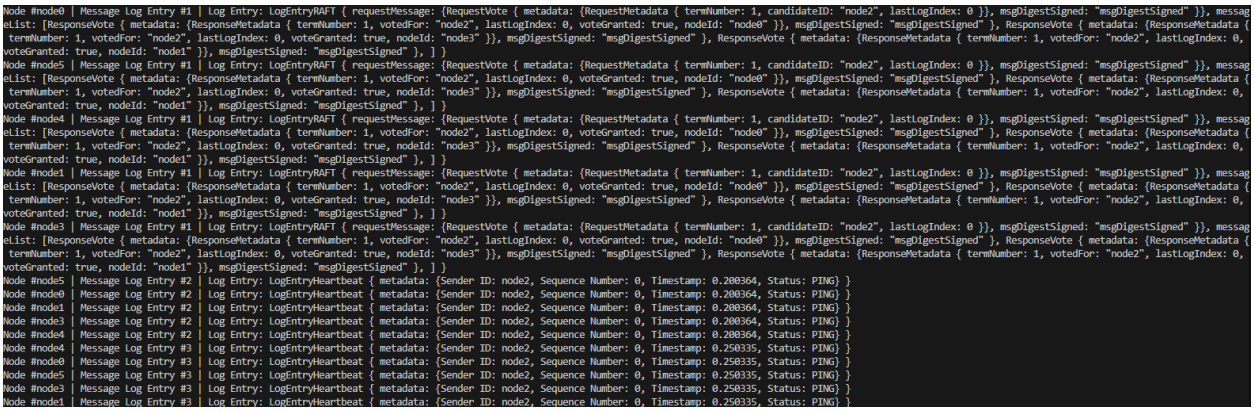
Experimentation:

In this experimental setup, we examine the operation of the RAFT protocol using three nodes. We aim to observe key aspects of the protocol, including leader election, message storage, and the maintenance of the leader's activity through periodic heartbeat messages. The simulation is designed to run for 300ms, and the final model should include a single AppendEntries message for leader election and n heartbeat messages.



The results show that node2 was elected as the leader. Each log entry contains proofs in the form of messages exchanged between nodes. Ideally, these messages would be hashed, signed, and verified. In this case, we observe "msgDigestSigned" as a placeholder for this verification process.

For greater flexibility, we aim to allow users to specify the number of nodes in the network. To explore this, the next test will simulate a network with up to six nodes, running for 300ms. We expect to see consistent behavior, including a proper leader election and n heartbeat messages.



In this scenario, all five nodes correctly follow the elected leader, which remains node2.

Conclusion

This first assignment provided a fantastic introduction to DEVS. While implementing the details of this distributed algorithm presented several challenges, it was a worthwhile endeavor. For future work, I plan to expand this simulation by incorporating a client atomic model and application-specific logic, as outlined in my conceptual design.

References

[1] "Raft Consensus Algorithm," geeksforgeeks, 07 03 2024. [Online]. Available: https://www.geeksforgeeks.org/raft-consensus-algorithm/.

[2] "practical Byzantine Fault Tolerance(pBFT)," geeksforgeeks, 23 08 2024. [Online]. Available: https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/.