Creating examples of distributed system design problems for the Structural Simulation Toolkit Discrete-Event Simulator

Nick Schantz and Melissa Jost

Problem Overview

- Distributed system problems:
  - Deadlock, Livelock, Synchronization, Thundering Herd, TCP Global Synchronization, Congestive Collapse.
- Can we detect these problems during a simulation?
  - Detect it during simulation to prevent it from manifesting when the system is put into production.
- Increase the number of SST simulation examples not related to HPC systems.

What is the Structural Simulation Toolkit (SST)?

- Discrete-Event Simulator Framework written in C++ and utilizes MPI.

- Created by Sandia National Laboratories to simulate high-performance computers.

- Discrete-event simulator that can model more than just high-performance computing.

http://sst-simulator.org/

What is a discrete-event simulator?

- Time advances in steps based off of events.
    - Examples of events include a simulated component updating/running or a component receiving a message from another component.
    - Each component has ports, which allow for them to link up to other components to communicate with each other.

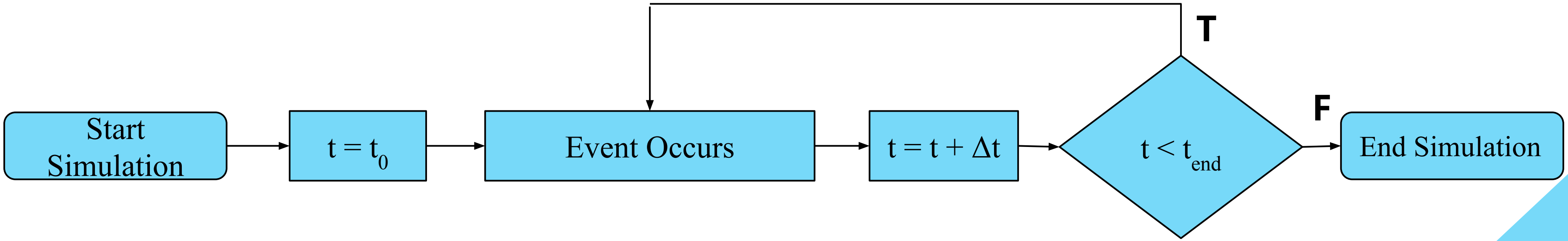


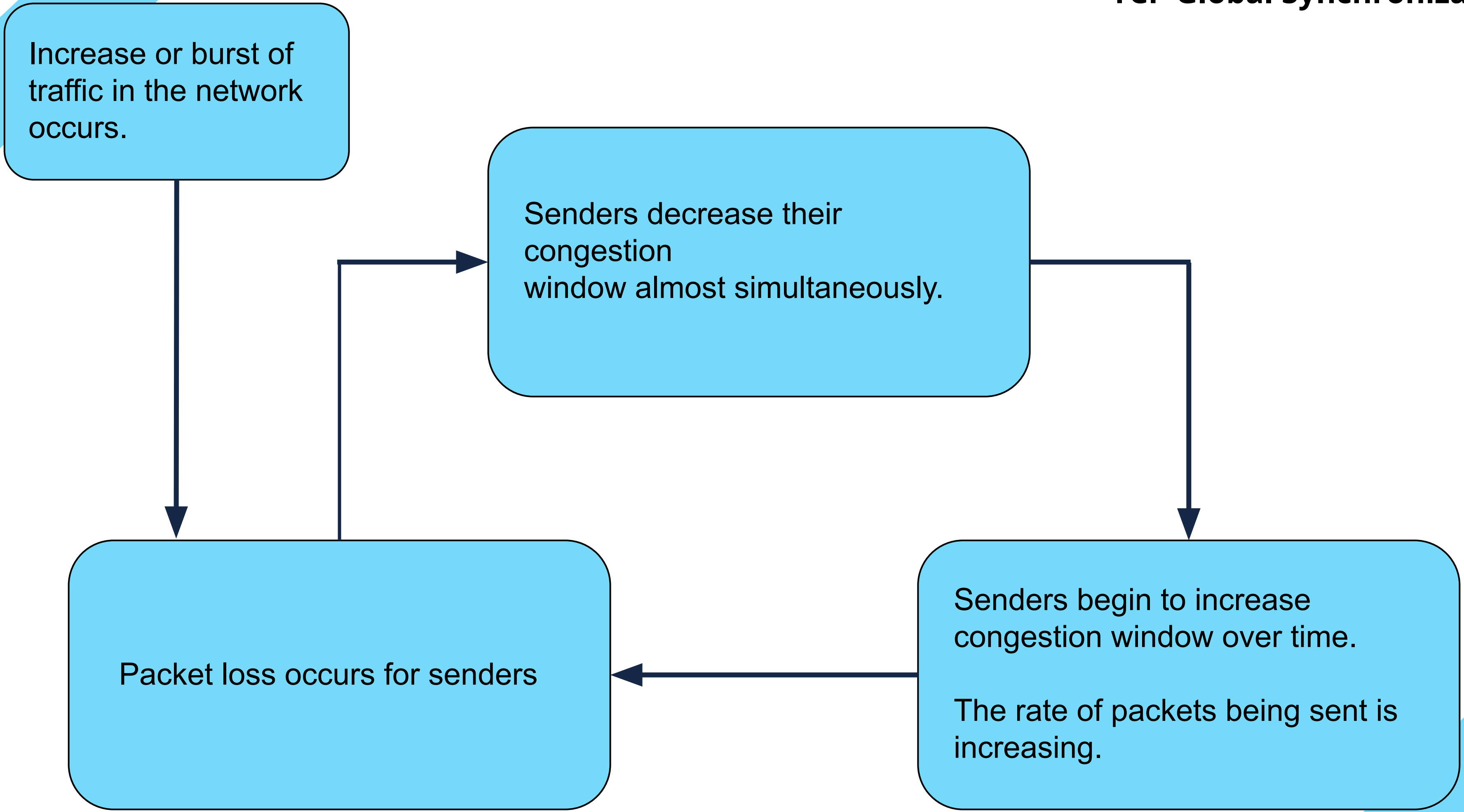
Diagram is modified from paper: http://cell-devs.sce.carleton.ca/publications/2013/GWK13/

TCP Global Synchronization

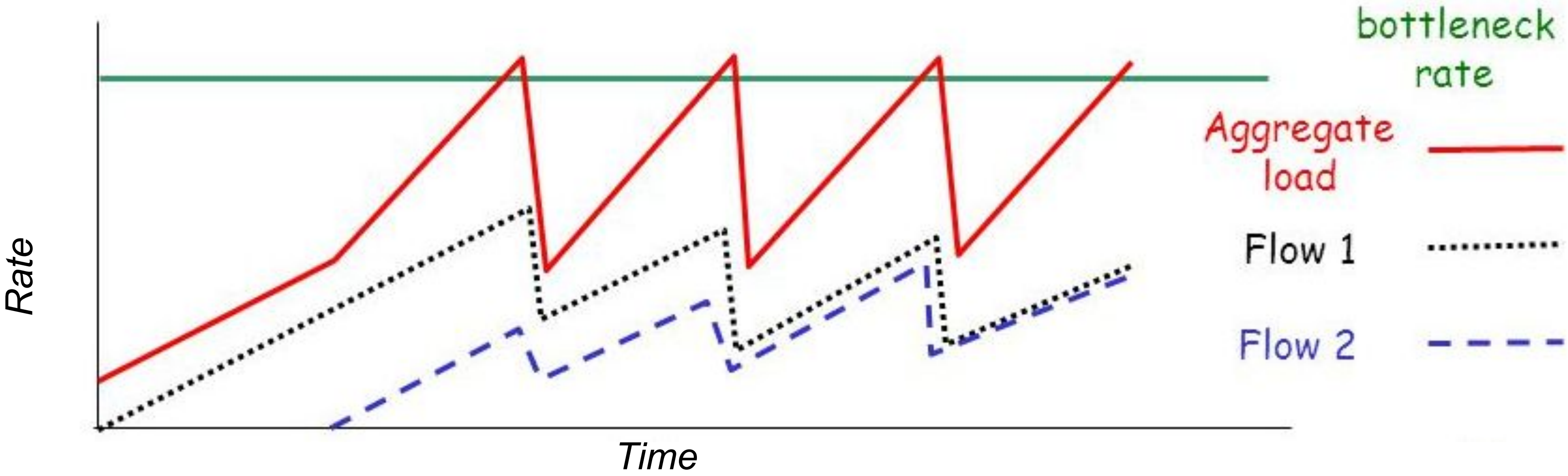
Problem occurs due to...

- TCP congestion control algorithm called “slow start”.
  - Sender starts with a small congestion window and increases it every time it receives an acknowledgement from a receiver.
- Queue dropping policy called “Tail drop”.
  - If the receiver’s queue is full, incoming packets will be dropped.

TCP Global Synchronization



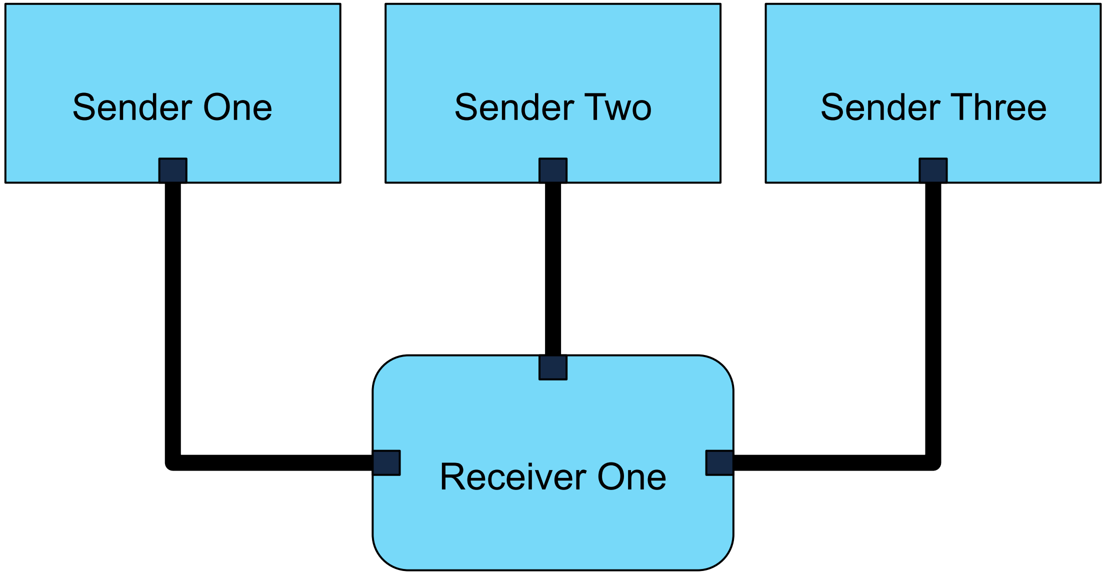
TCP Global Synchronization



Snippet of image from: https://techstat.net/1-1-e-vi-global-synchronization/

SST Model

There are *n* sending components that all send packets to one receiving component.



Model Assumptions

- Senders use the same protocol.
  - Transmission rates will be reduced to the same rate.
- The receiver uses a tail drop policy.
- The receiver notifies senders when their packet has been dropped.
- Senders increase their transmission rates linearly after each tick.
  - Transmission rate is treated as the senders congestion window.

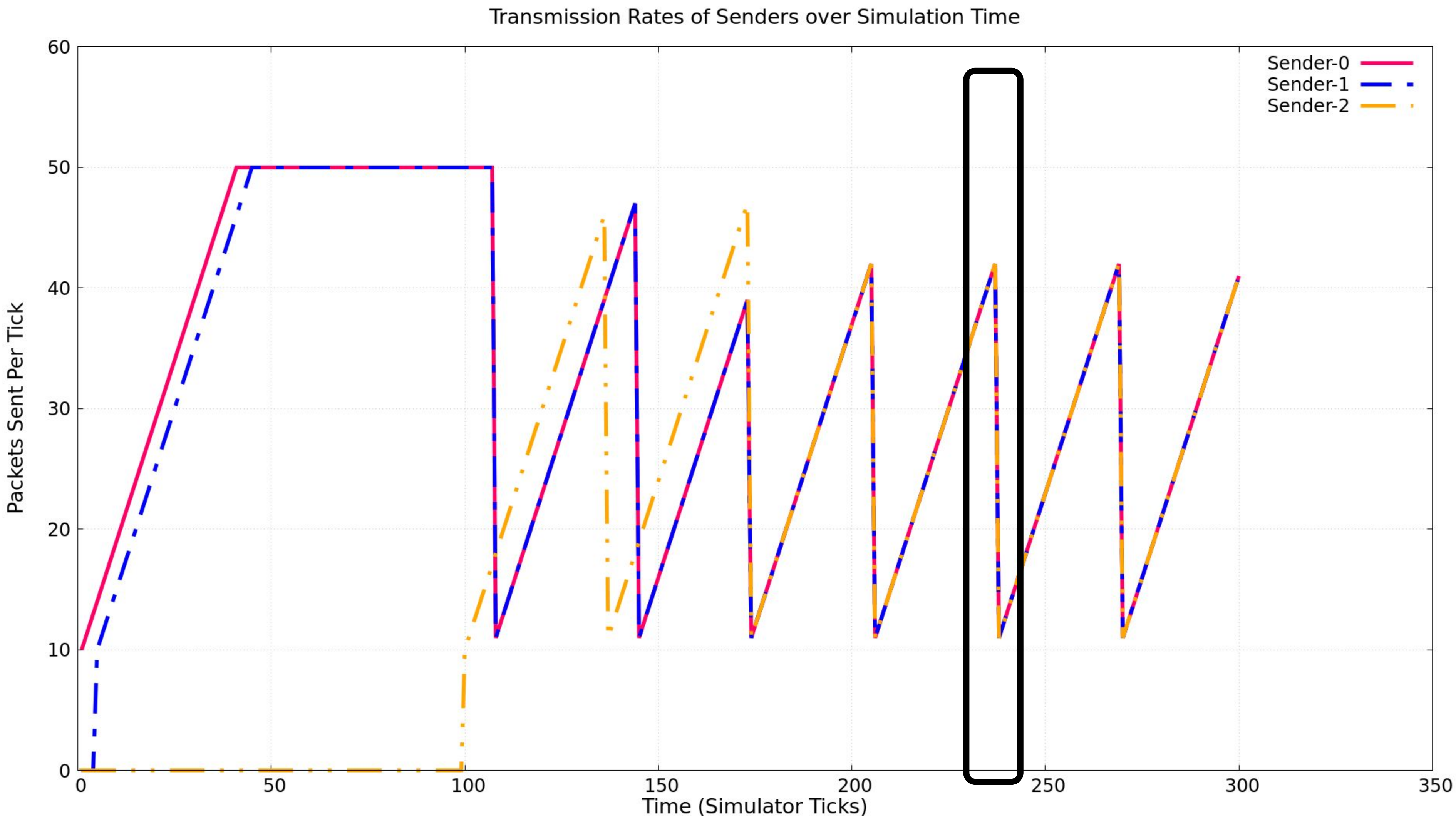
What to Measure for TCP Global Synchronization

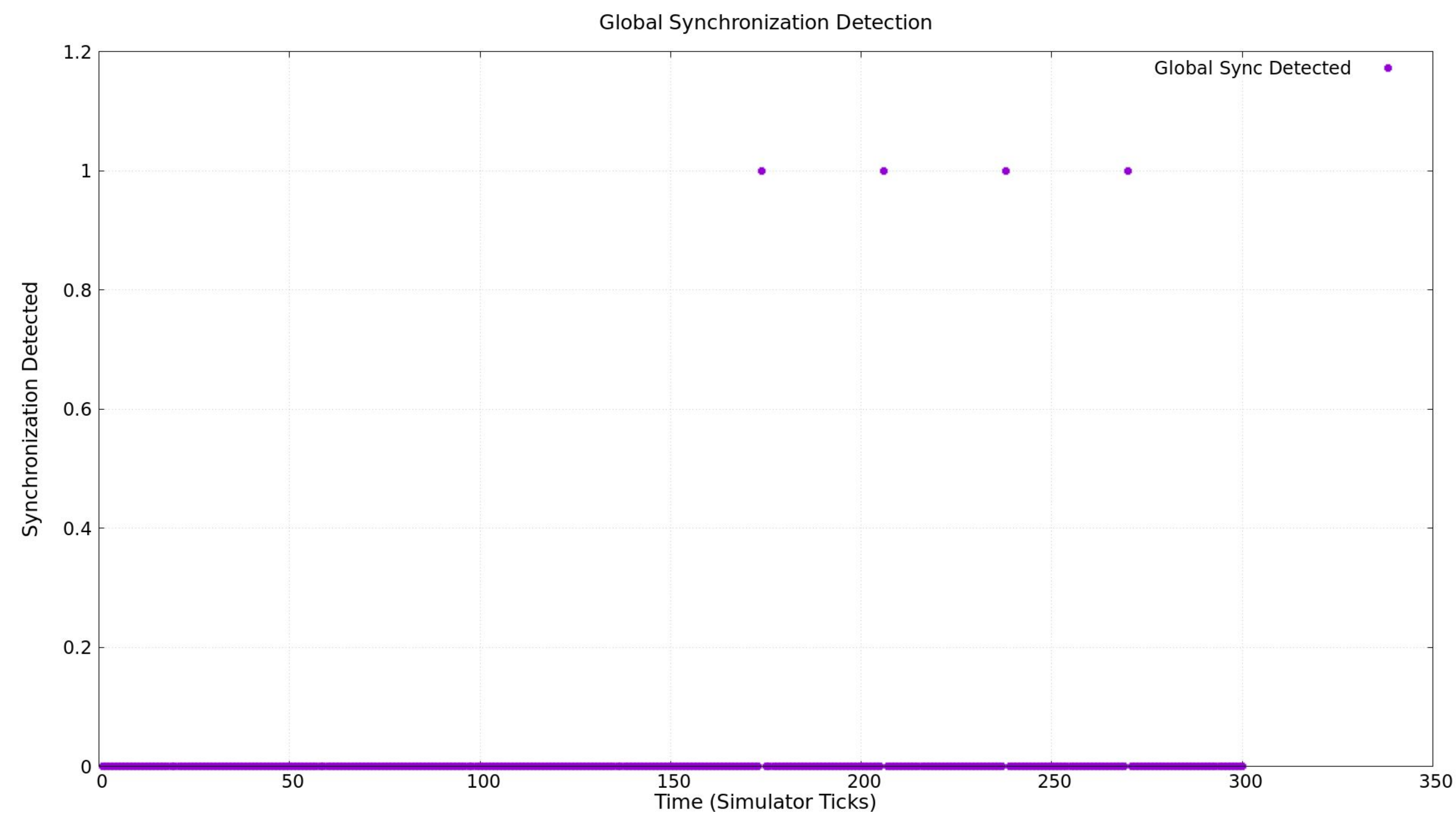
- Number of sending components that limit their transmission rates in a window of time.

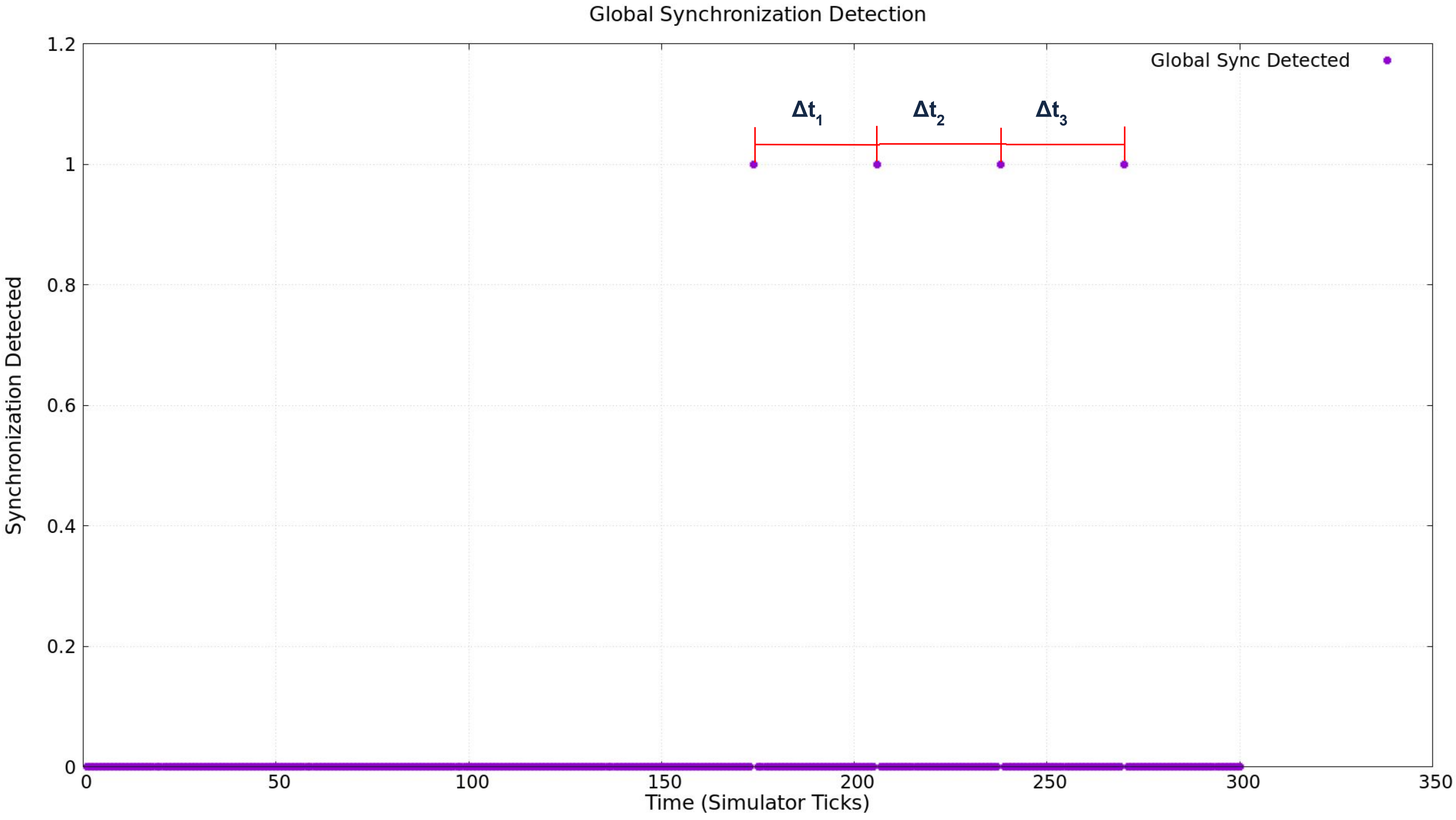
- Average difference between times in which the behavior above has occurred.

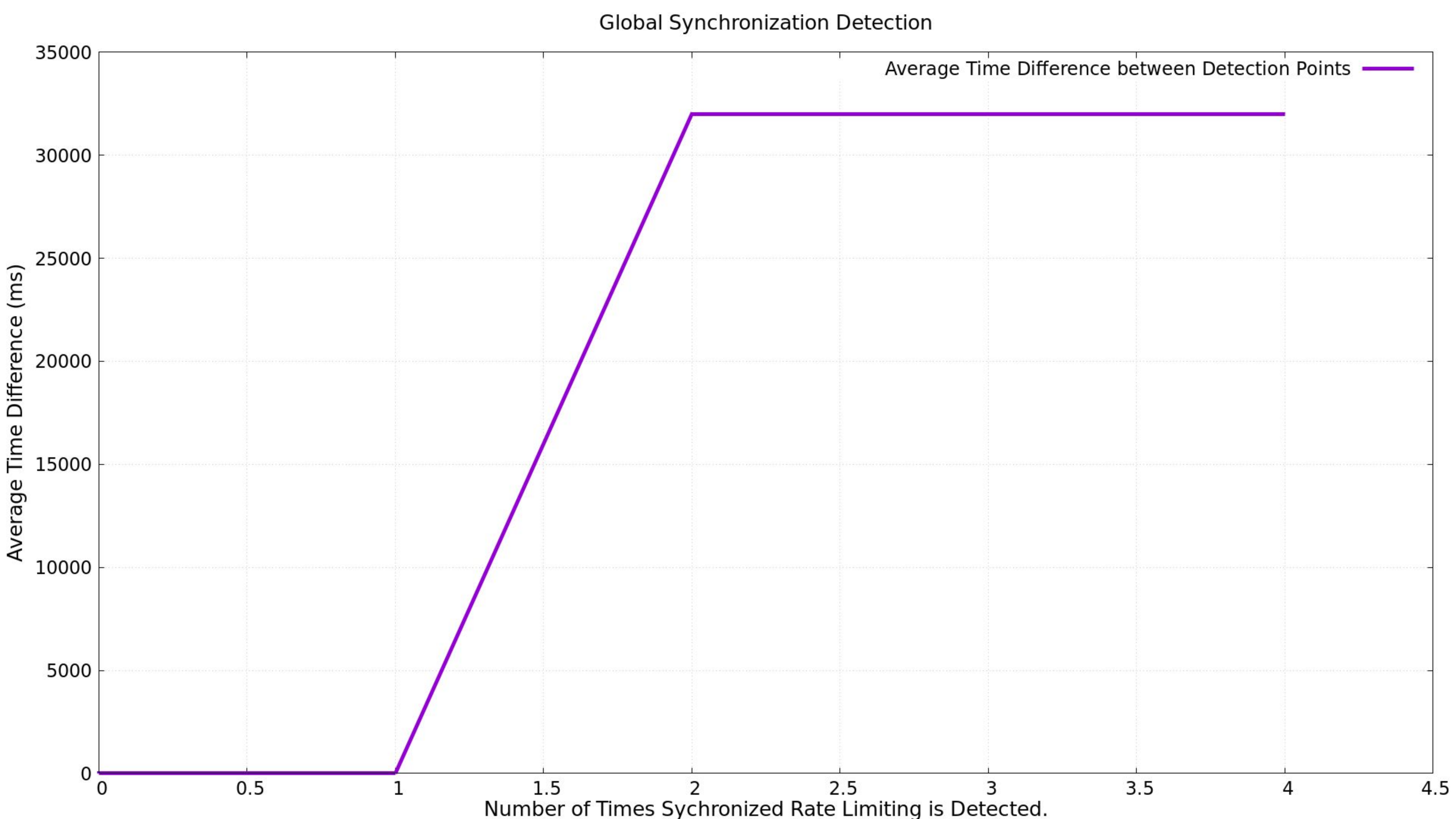
- Standard deviation.

Transmission Rates of Senders over Simulation Time
Packets Sent Per Tick
Time (Simulator Ticks)
Sender-0
Sender-1
Sender-2
0
10
20
30
40
50
60
0
50
100
150
200
250
300
350

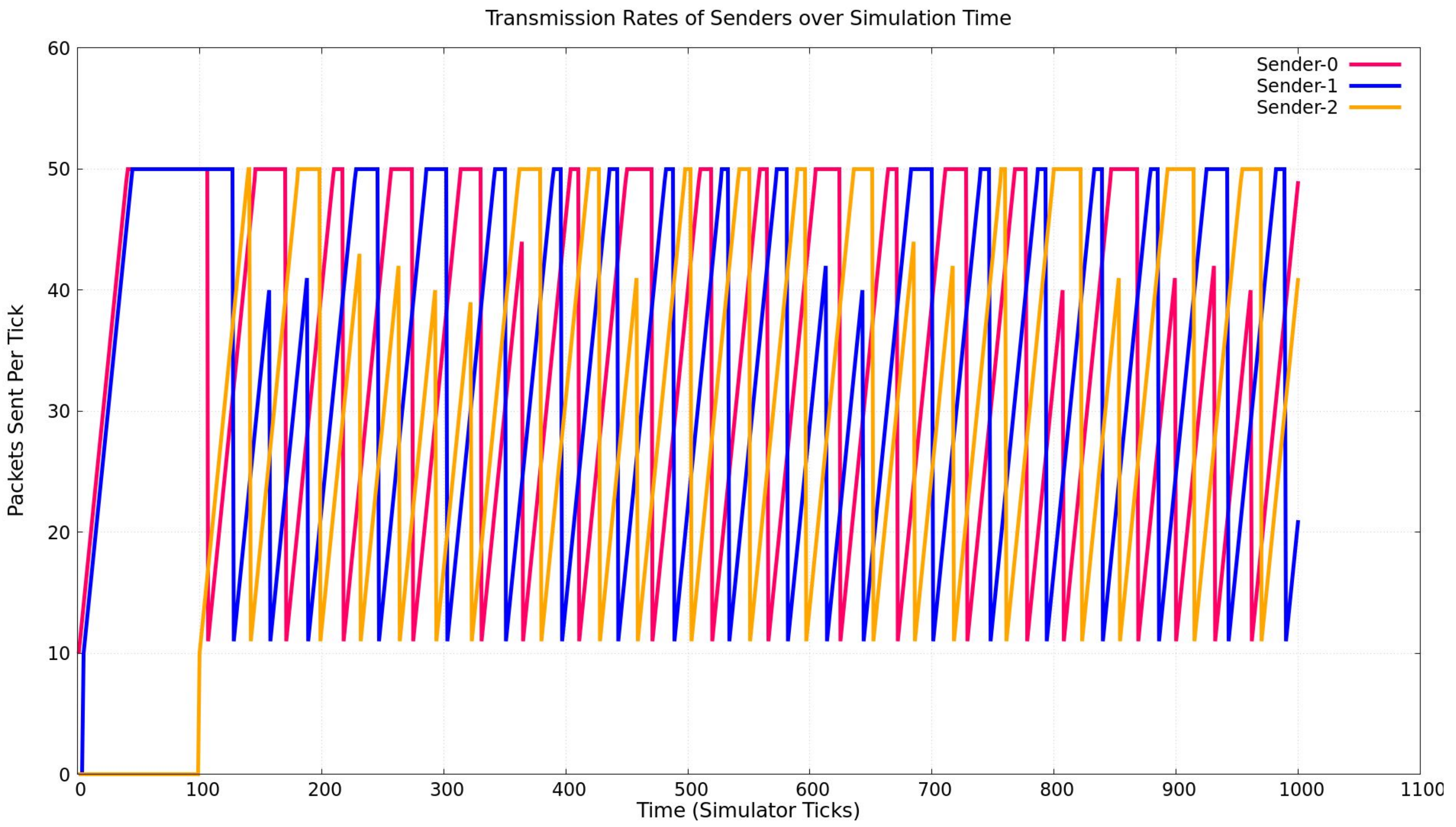

Transmission Rates of Senders over Simulation Time
Packets Sent Per Tick
Time (Simulator Ticks)
Sender-0
Sender-1
Sender-2
0
10
20
30
40
50
60
0
50
100
150
200
250
300
350

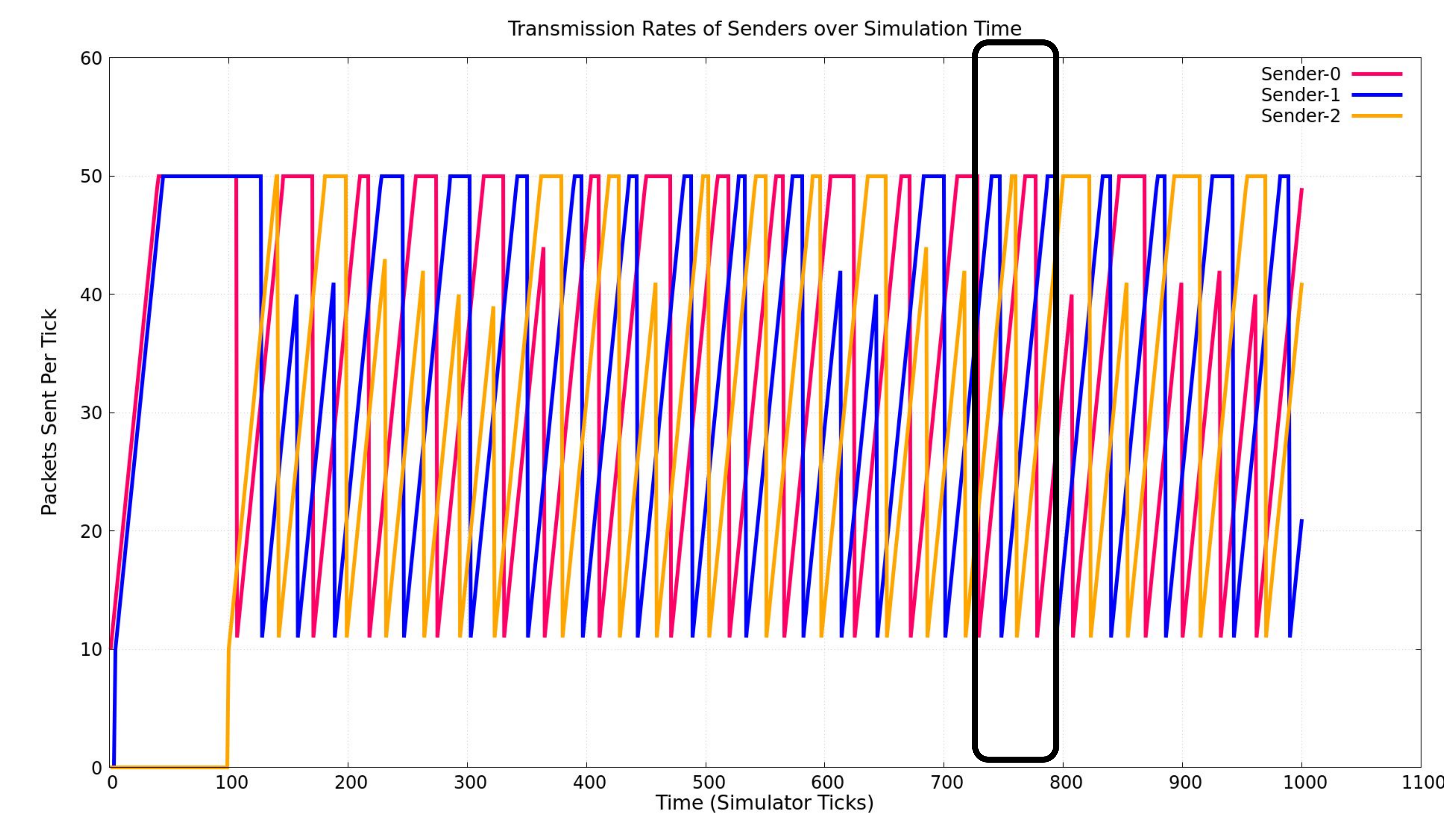

Global Synchronization Detection
Global Sync Detected
Synchronization Detected
Time (Simulator Ticks)
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350


Global Synchronization Detection
Global Sync Detected
Δt_1
Δt_2
Δt_3
Synchronization Detected
Time (Simulator Ticks)
1.2
1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
250
300
350

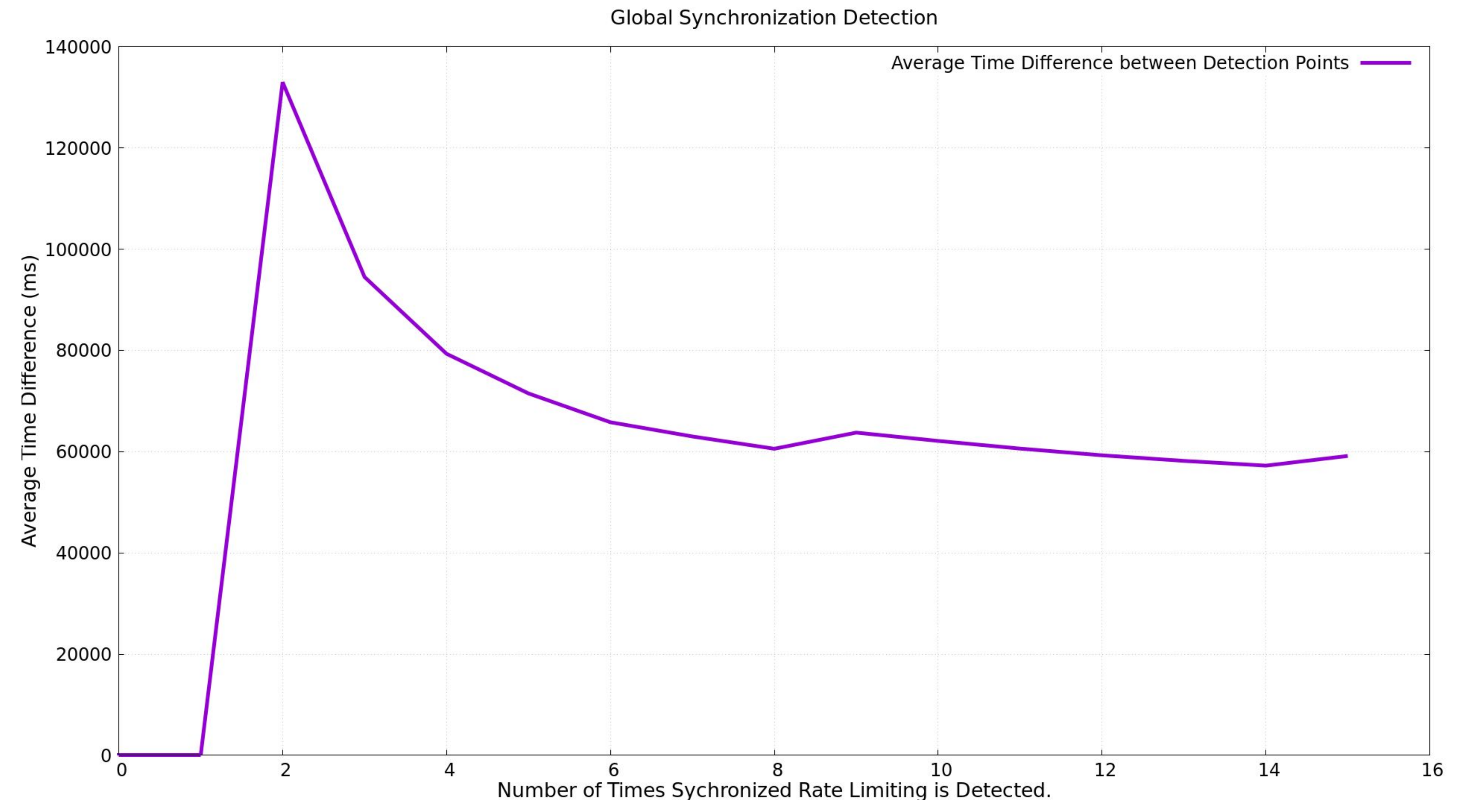


Global Synchronization Detection
Average Time Difference between Detection Points
Average Time Difference (ms)
Number of Times Sychronized Rate Limiting is Detected.
35000
30000
25000
20000
15000
10000
5000
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5


Transmission Rates of Senders over Simulation Time
Packets Sent Per Tick
Time (Simulator Ticks)
Sender-0
Sender-1
Sender-2
0
10
20
30
40
50
60
0
100
200
300
400
500
600
700
800
900
1000
1100



Transmission Rates of Senders over Simulation Time
Sender-0
Sender-1
Sender-2
Packets Sent Per Tick
Time (Simulator Ticks)
0
10
20
30
40
50
60
100
200
300
400
500
600
700
800
900
1000
1100


Global Synchronization Detection
Average Time Difference between Detection Points
Average Time Difference (ms)
Number of Times Sychronized Rate Limiting is Detected.
0
20000
40000
60000
80000
100000
120000
140000
0
2
4
6
8
10
12
14
16

Future Work

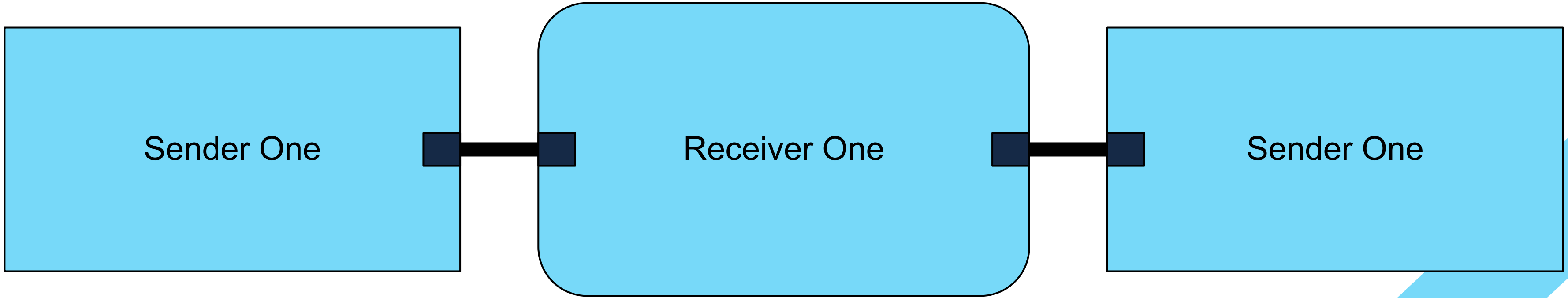
- Increase the fidelity of the simulation.
  - Will the metrics still hold up?
- Expand on the metric.
  - It can potentially produce false positives on a network flooded with internet traffic.

Congestive Collapse

- Non-useful data clogs a network and decreases useful throughput.
  - Increase in retransmitted data.

SST Model

There are *n* sending components that all send packets to one receiving component.



Model Assumptions

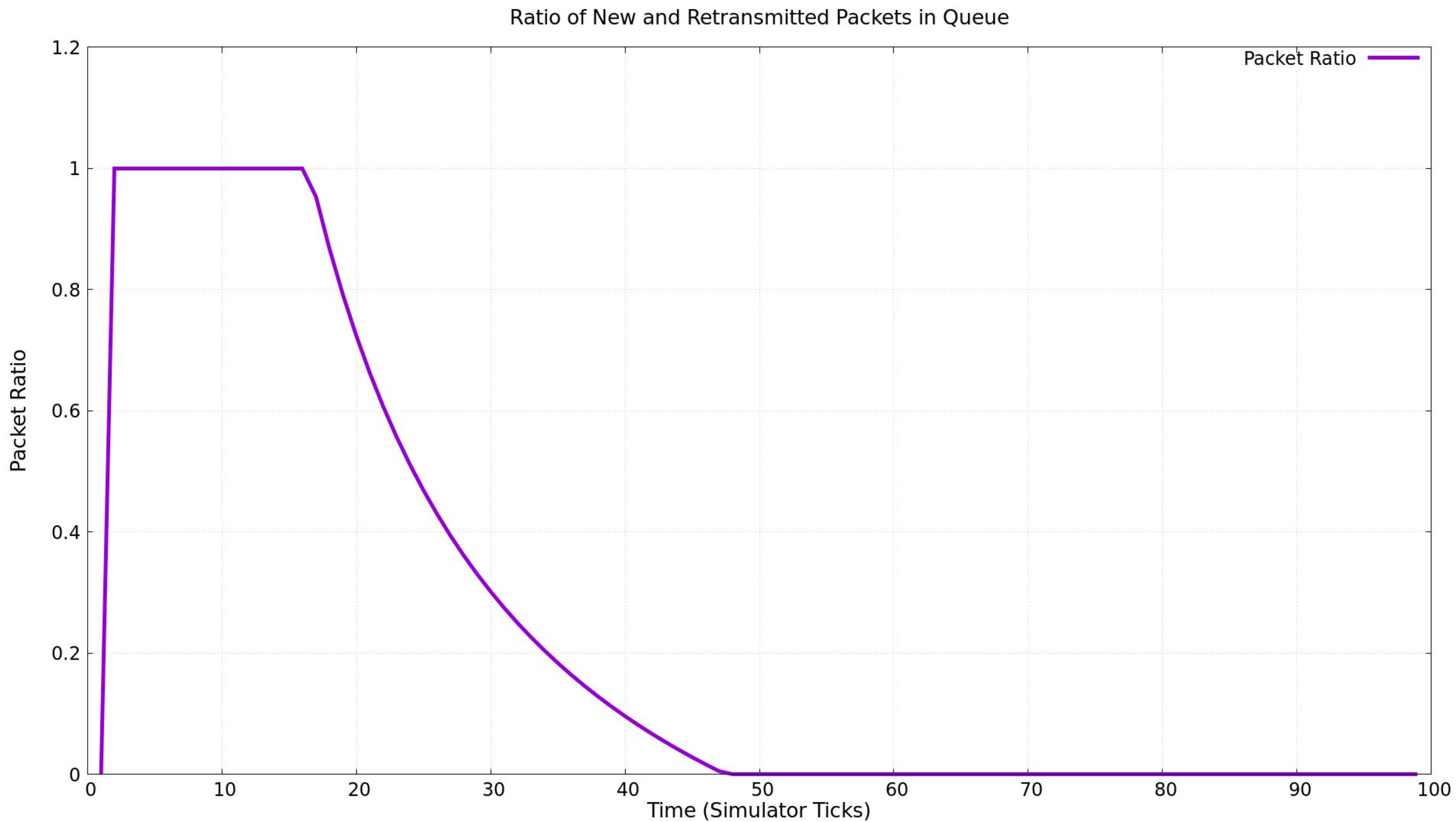
- Congestion Control algorithms are not being used.
    - “Conservation of packets” is not followed.
- Receiver uses an infinite queue. Packet loss does not occur.

What to Measure for Congestive Collapse

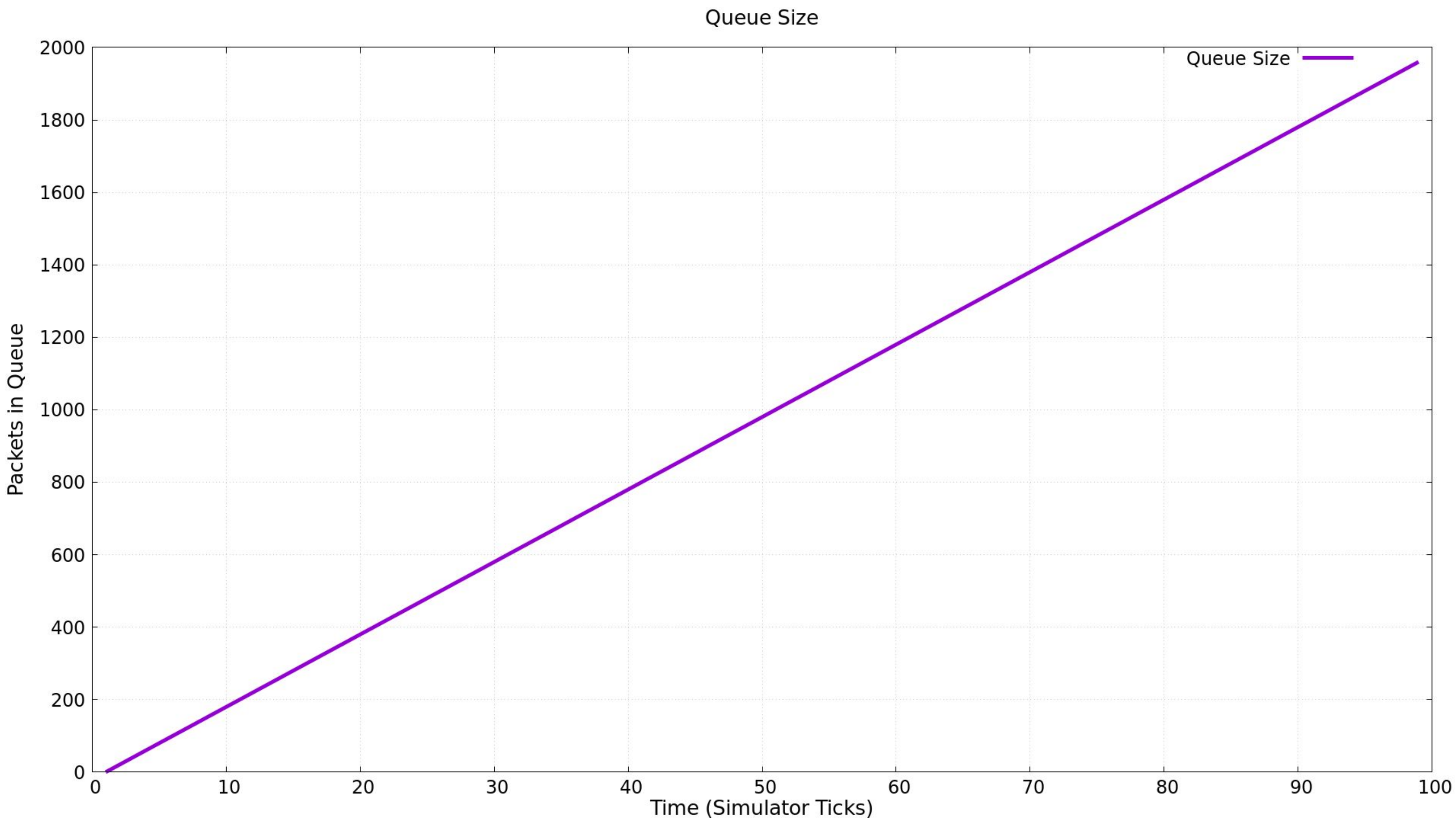
- Ratio of new packets to retransmitted packets in a receiver’s queue.

- Receiver’s queue depth.

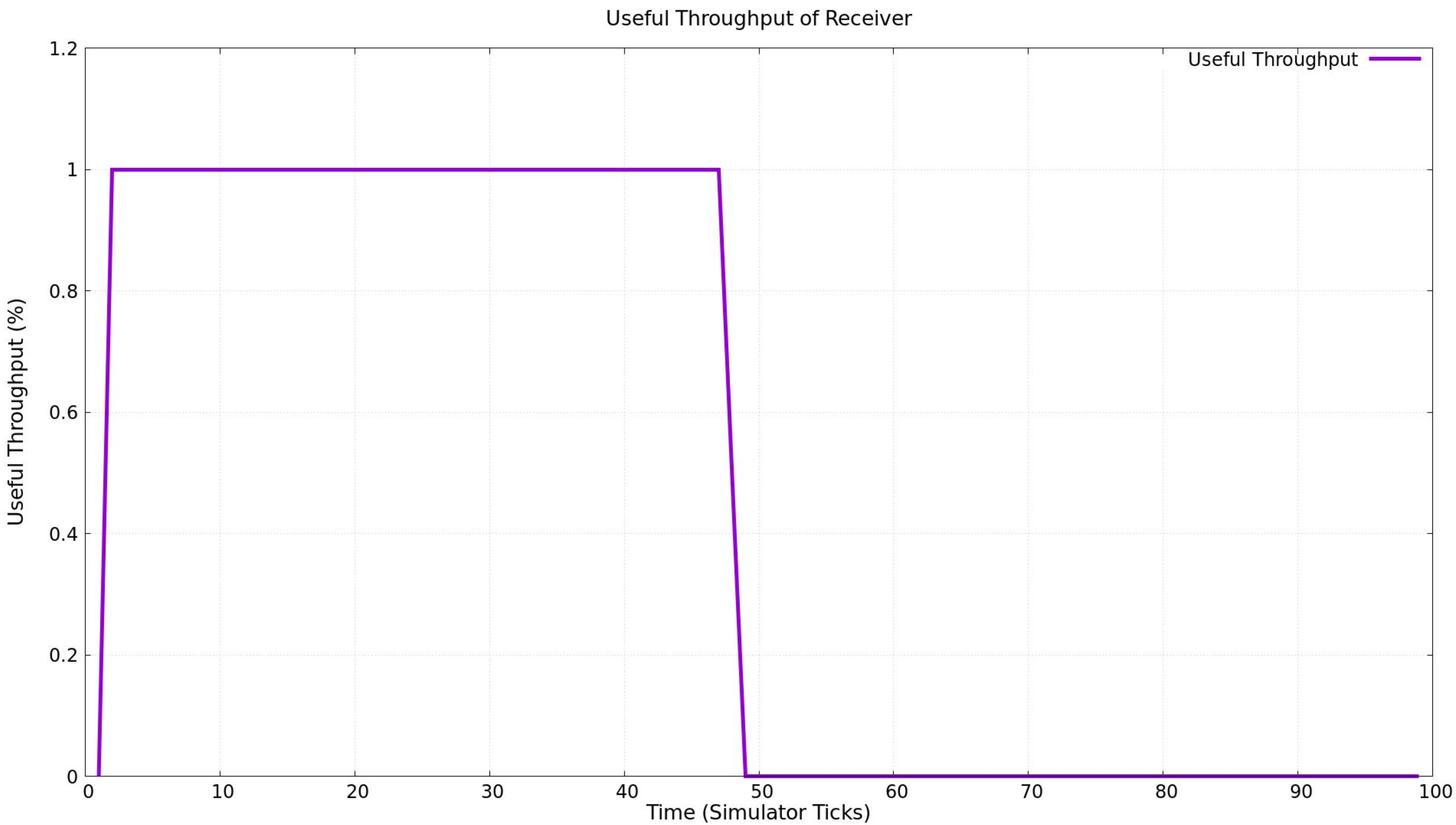
- Ratio of useful throughput to total throughput of the receiver.



Ratio of New and Retransmitted Packets in Queue
Packet Ratio
Packet Ratio
Time (Simulator Ticks)
1.2
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
50
60
70
80
90
100



Queue Size
Queue Size
Packets in Queue
Time (Simulator Ticks)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
10
20
30
40
50
60
70
80
90
100


Useful Throughput of Receiver
Useful Throughput
Useful Throughput (%)
Time (Simulator Ticks)
0
0.2
0.4
0.6
0.8
1
1.2
0
10
20
30
40
50
60
70
80
90
100

Future Work

- Increase the fidelity of the simulation.
    - Will the metrics still hold up?

Deadlock

- Multiple nodes/processes connected in a cycle proceed with any action because they are waiting for each other to take action as well.

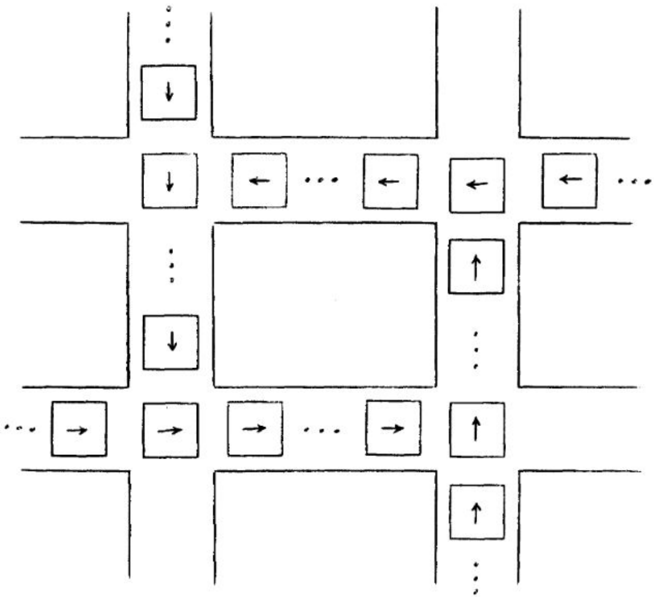


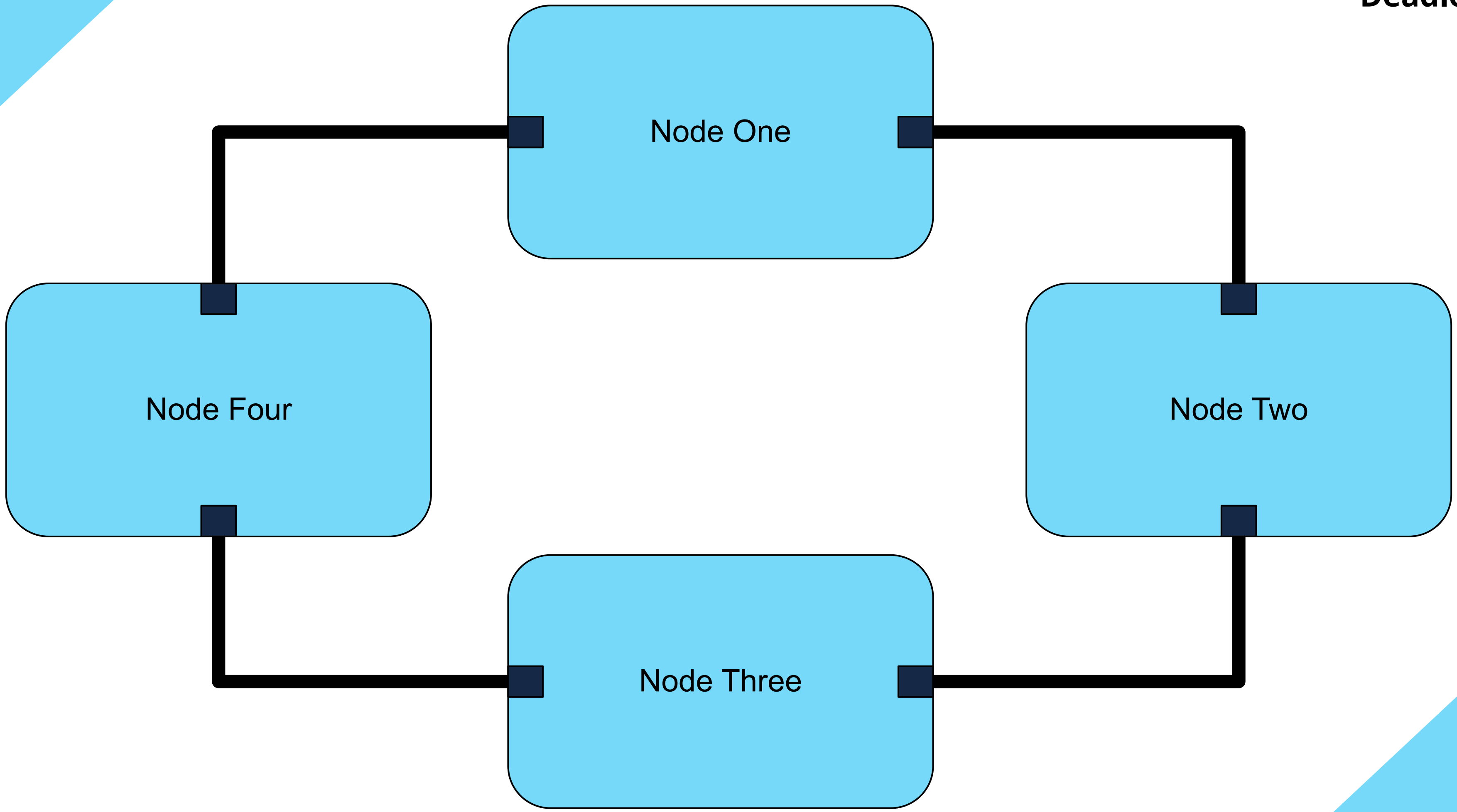
Image from https://dl.acm.org/doi/pdf/10.1145/356586.356588

First SST Model

- Model is *n* nodes in a ring topology that send messages unidirectionally.

- Nodes have queues to store messages in and use credits to tell other nodes how much space they have left.

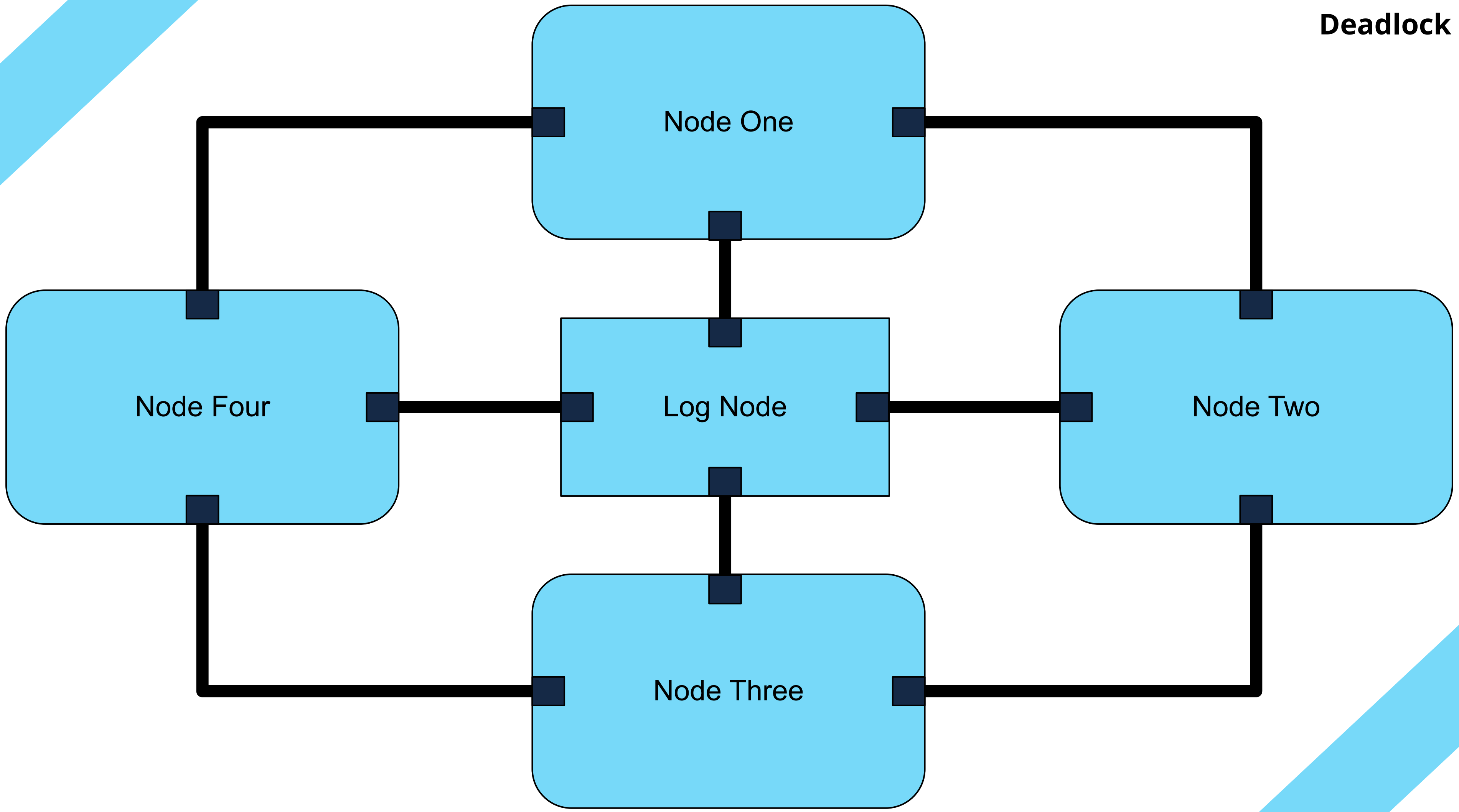
Deadlock



Second SST Model

- Similar to model one
    - Model is *n* nodes in a ring topology that send messages unidirectionally.
    - Nodes have queues to store messages in and use credits to tell other nodes how much space they have left.
- Introduces a node that logs data from all nodes in the ring topology.

Deadlock



What to Measure for Deadlock

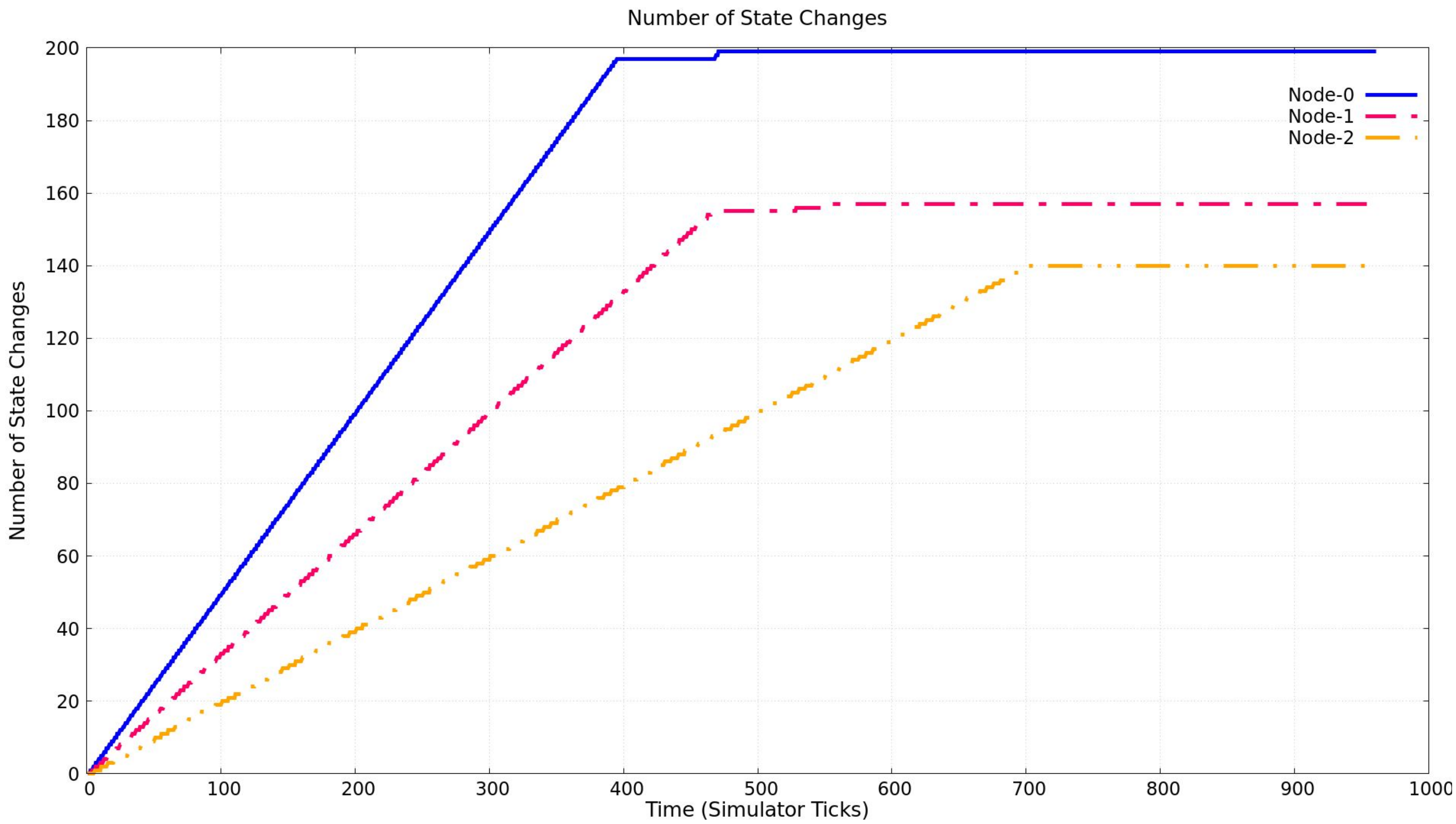
In both models, these metrics relate to system-scale deadlock.

In model one (no logging node):

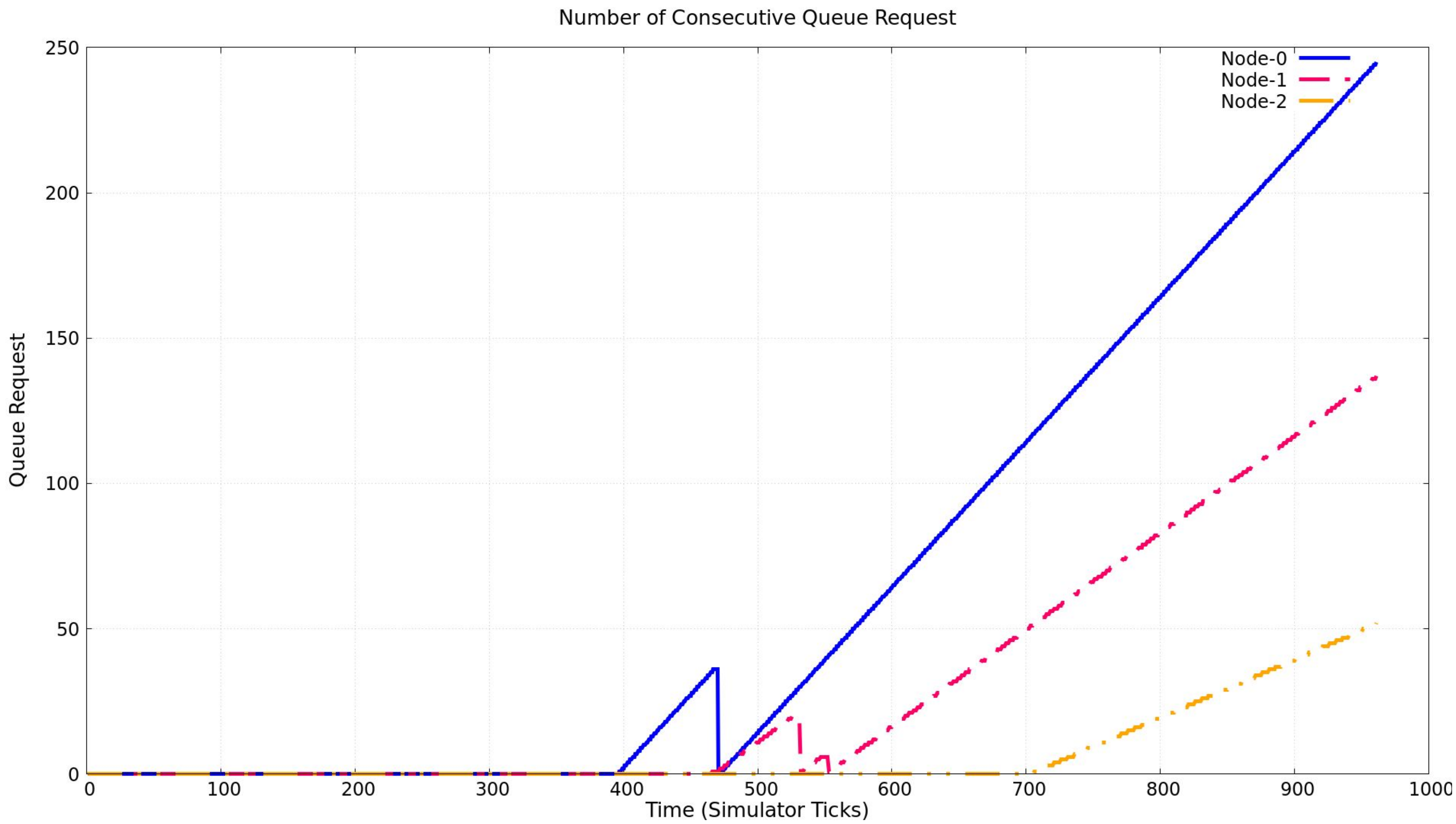
- Measure a circuit of blocked nodes in the system.

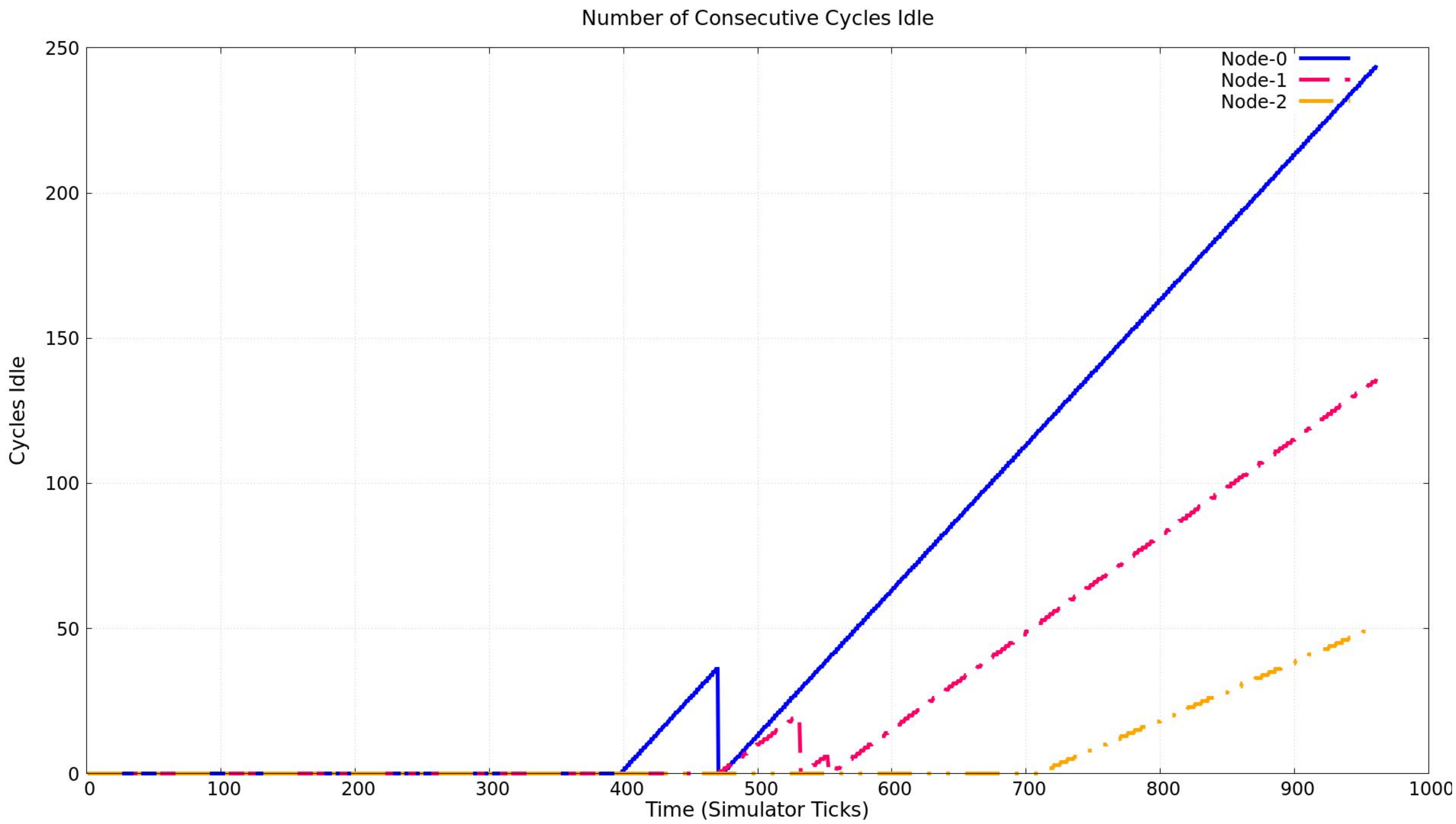
In model two:

- Measure number of state changes for a node over time.
- Measure time a component is idle.
- Measure the node’s number of consecutive request for a resource.



Number of State Changes
Number of State Changes
Time (Simulator Ticks)
Node-0
Node-1
Node-2
0
20
40
60
80
100
120
140
160
180
200
0
100
200
300
400
500
600
700
800
900
1000


Number of Consecutive Queue Request
Queue Request
Time (Simulator Ticks)
Node-0
Node-1
Node-2
0
50
100
150
200
250
0
100
200
300
400
500
600
700
800
900
1000


Number of Consecutive Cycles Idle
Cycles Idle
Time (Simulator Ticks)
Node-0
Node-1
Node-2
0
50
100
150
200
250
0
100
200
300
400
500
600
700
800
900
1000

Future Work

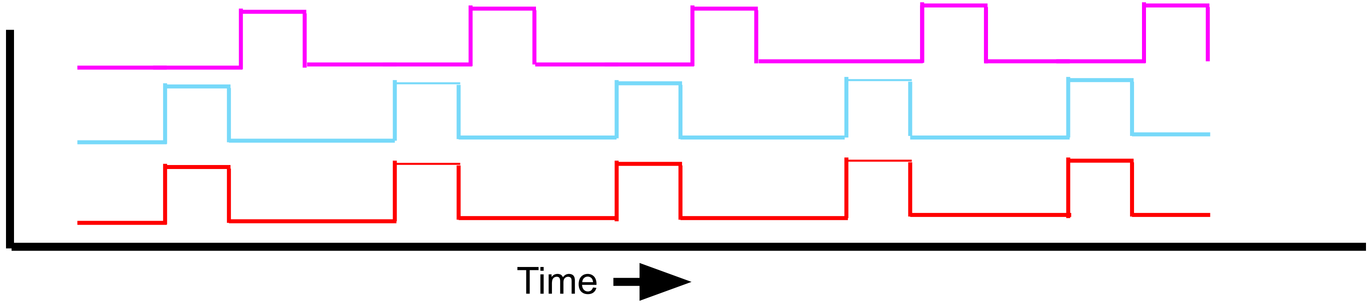
- Focus on determining a metric for component-scale deadlock.

- Attempt to adapt the conditions for deadlock into the SST model.

- Modify the model to support other topologies.

Problem of Synchronization

- What if oscillator synchronization is unwanted behavior?
  - Kuramoto Model
- How can we determine if oscillators in a system have synchronized?
- Can we detect which oscillators are synchronized and which are not?



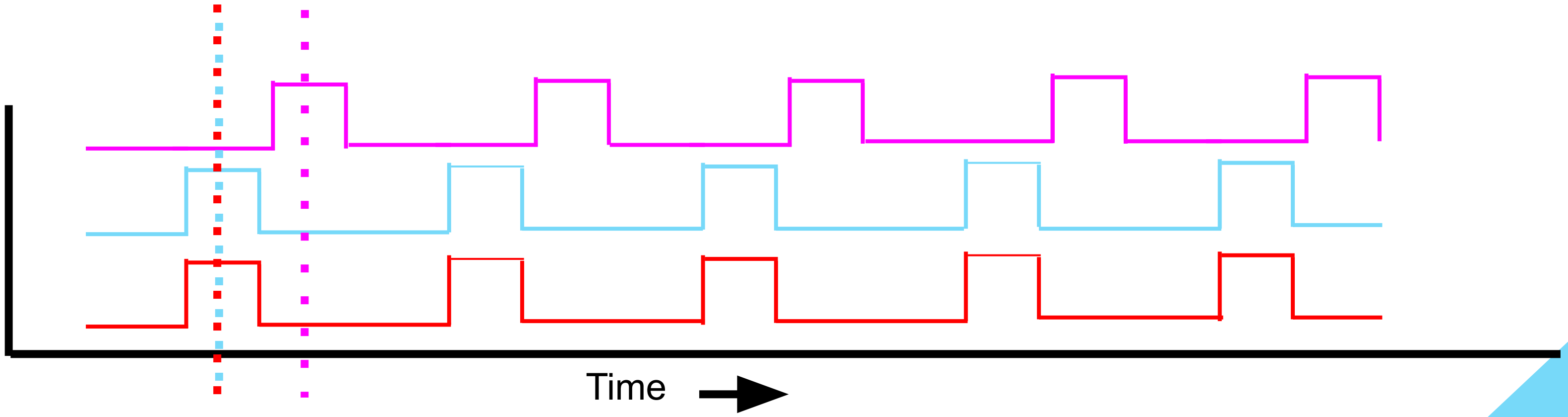
Progress

- Goal:
  - Find *n* oscillators in a set that have the same frequency and are in-phase or anti-phase and which do not.

- Theoretical assumptions if we were working with a discrete-event simulation:
  - Frequency of oscillator is known during simulation time.

Progress

- Current Implementation:
  - Use a reference oscillator and measure the difference of time between its first peak to all other oscillators' first peaks.
  - Determine the phase difference and compare the results with all oscillators.



Future Work

- Concrete implementation of synchronization detection algorithms in a discrete-event simulator.

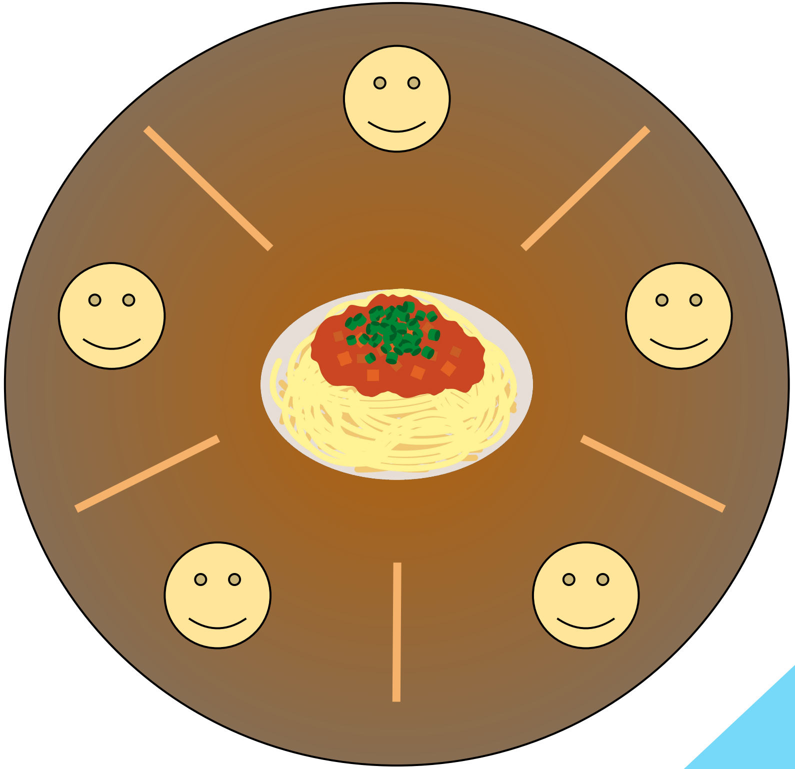
- Explore inter-site phase clustering.

Livelock

- “Livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.”
- Usually arises as a result of avoiding deadlock
- “Classic” example: imagine two people running into each other in a hallway

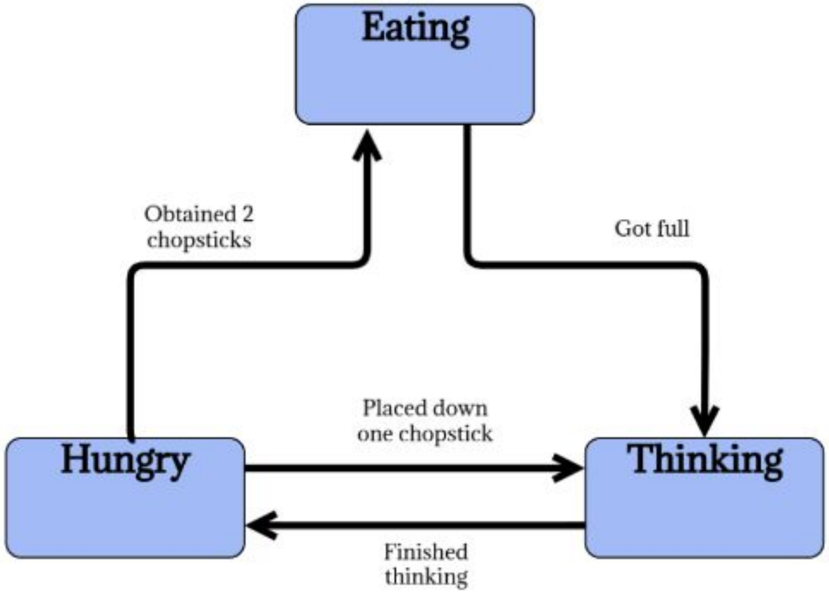
Example of Livelock →Dining Philosophers

- Lack of real world problems to model, so we chose this scenario
- Premise:
  - 5 philosophers, 5 chopsticks
  - You need 2 chopsticks to eat
  - You can only pick up one chopstick at a time



Demo!

- 3 participants needed
- Each will run on a 15 second cycle
- At 5 seconds, try to grab your left chopstick
- At 10 seconds, try to grab your right chopstick
- At 15 seconds, place your chopstick down if you're only holding one
- Can you eat?



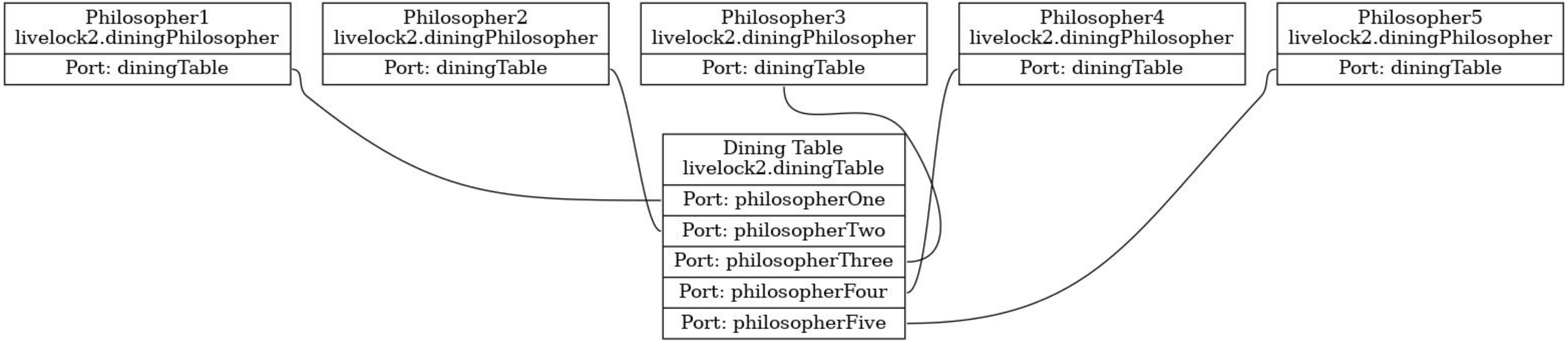
Demo!

Reminders!!

- 15 second cycle
- 5 sec: grab your left chopstick
- 10 sec: grab your right chopstick
- 15 sec, place your chopstick down if you're only holding one

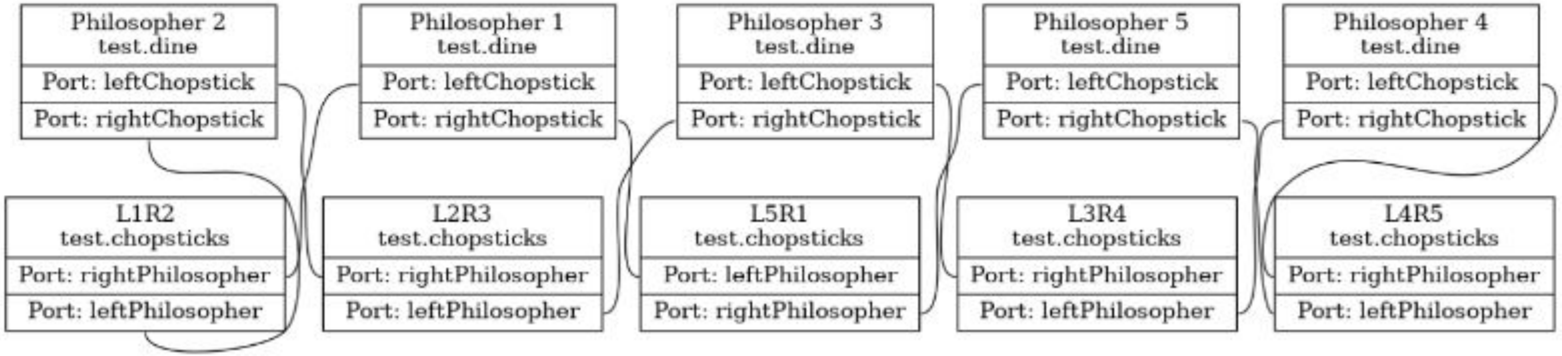
SST Model #1: Dining Table Model

- One central dining table to hold all 5 chopsticks
- Dining table has individual ports for each philosopher
- Each philosopher only has one port to the dining table



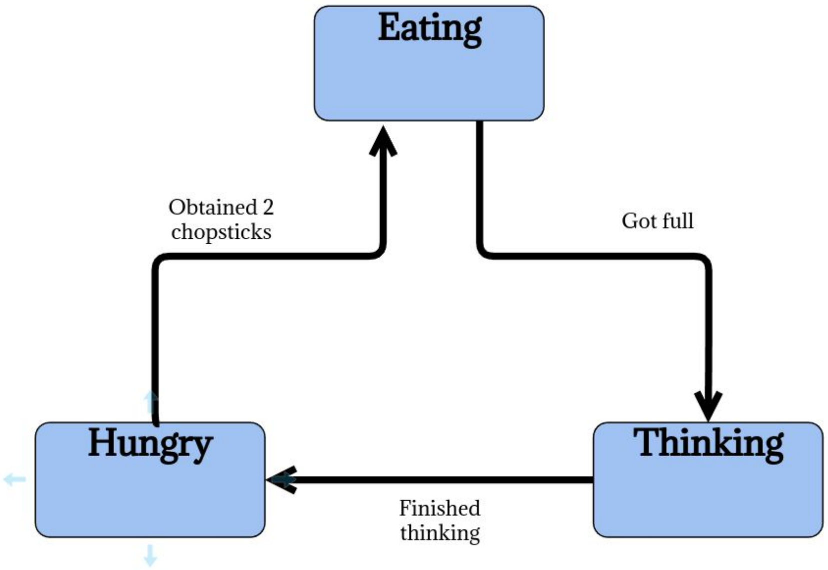
SST Model #2: Chopstick Model

- Chopsticks were the secondary component alongside the philosophers
- Both components had 2 links to their left and right components



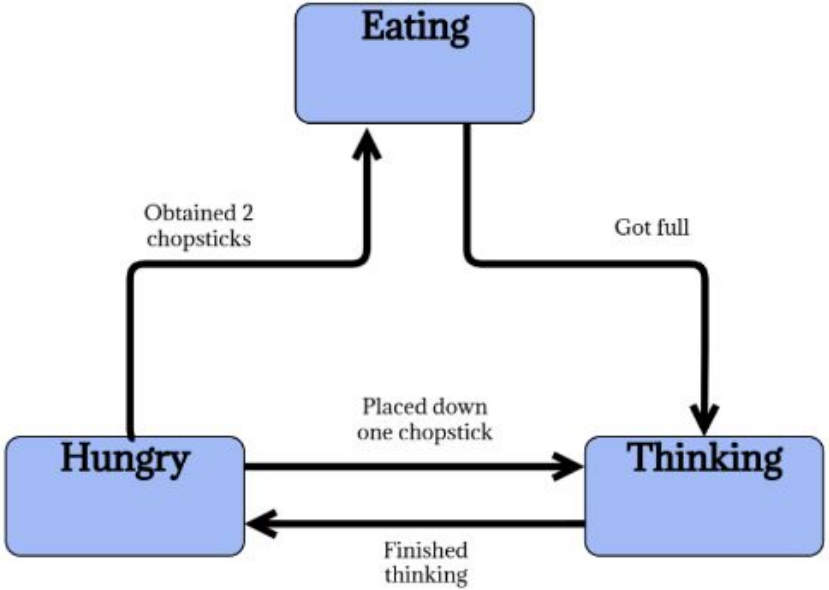
SST Model #3: Deadlock Model

- Based off of the dining table model, but either would suffice
- Removed the “polite” action of returning chopsticks if you were only holding one
- Helped us define livelock in a way that doesn’t falsely indicate livelock in a deadlock scenario



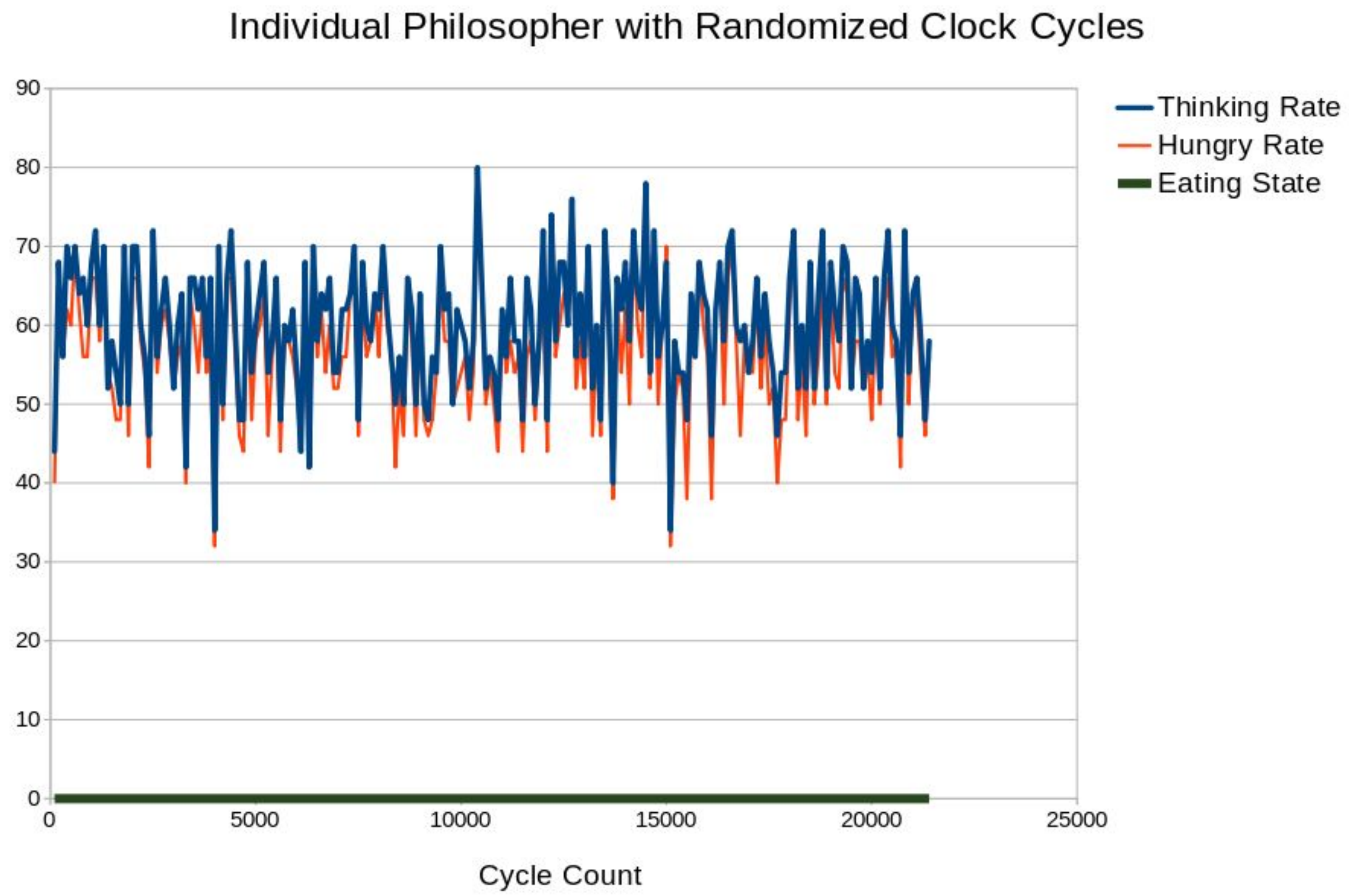
Metrics Chosen For Problem

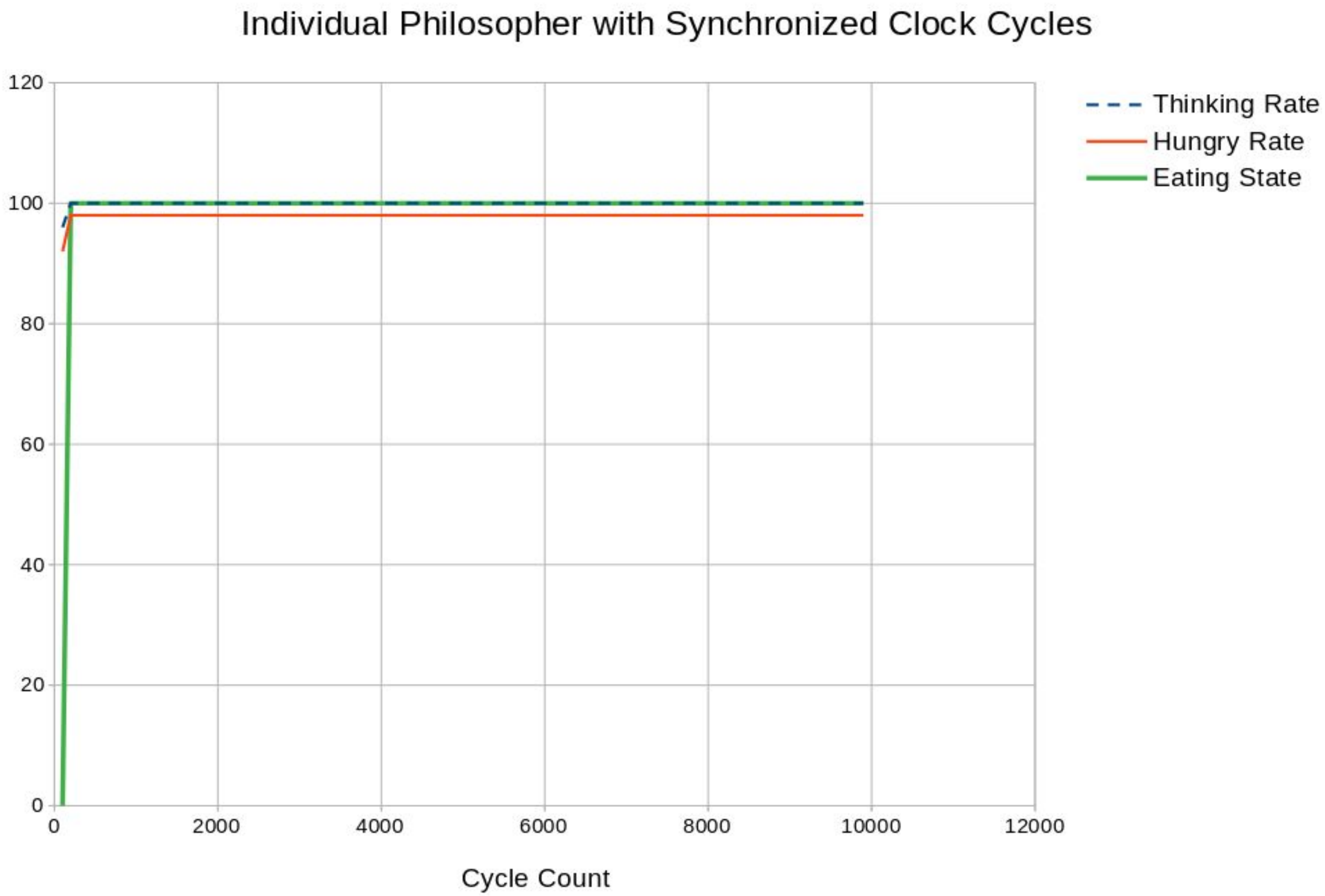
- Focused on using state transitions to identify livelock, and differentiatie from deadlock
- We selected a window size to check these states throughout the simulation
- For livelock, one only switches between hungry and thinking



What to Measure for Livelock

- Eating State: Boolean value of whether or not one ate in that window
- Thinking Rate: (actual time spent thinking) / (expected time spent thinking)
- Hungry Rate: (actual time spent hungry) / (expected time spent hungry)
- Both rates are shown as a percentage
- Our expected time for each is calculated as (window size/2), since a livelocked philosopher should only switch between those 2 states


Individual Philosopher with Randomized Clock Cycles
90
80
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
Cycle Count
Thinking Rate
Hungry Rate
Eating State


Individual Philosopher with Synchronized Clock Cycles
120
100
80
60
40
20
0
0
2000
4000
6000
8000
10000
12000
Cycle Count
Thinking Rate
Hungry Rate
Eating State

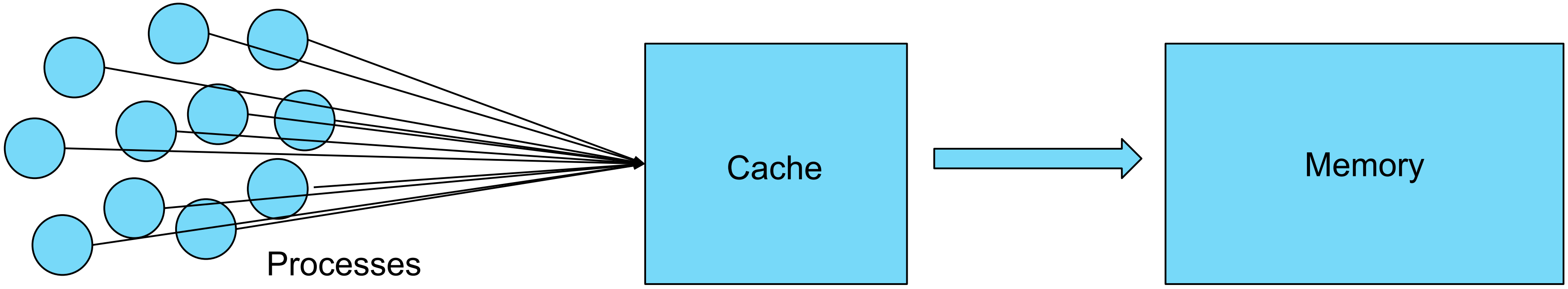
Future Work

- Simulate partial deadlock + transient livelock by varying the number of chopsticks
- Find more real-world examples of livelock to simulate
- Generalize the idea of tracking states to fit a wider range of problems

Thundering Herd

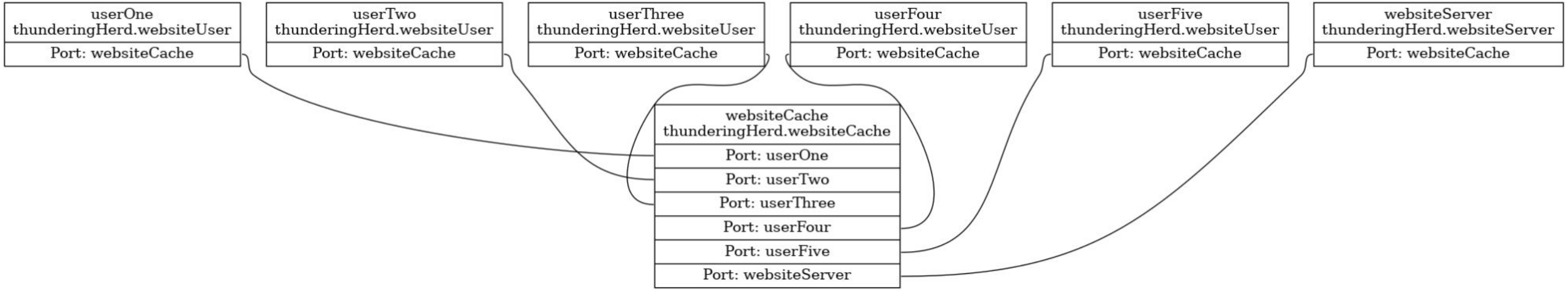
- Order of Events
  1. Many processes all request the same value in the cache
  2. The cache doesn’t have this value
  3. All the processes request directly from memory
  4. Memory isn’t capable of handling this many simultaneous requests
  5. If the process’ request timed out, they will send another request
- Typically leads to a huge slowdown in memory until one process finally gets a response, and the cache is updated

Thundering Herd



Model

- 3 Components: websiteUsers, websiteCache, and websiteServer
- Chosen to mirror real world instances of Thundering Herd crashes



Process of Events

- Users send requests to the cache for a website they want to access
- Cache either returns the site, or requests it from the server
- Both the cache and server have a queue to handle these requests
- Users can become impatient and spam requests if they don’t receive results
- If the server gets overloaded with requests, it goes down
- Goal is for cache to handle majority of requests (higher frequency)

Future Steps

- Refine simulation to more closely mirror real-world websites
  - Eg. Time spent processing requests, lengths of server failure during requests overload, etc.
- Clearly define metrics to detect this problem
- Refine simulation to allow for easier scalability

Future Computing Summer Internship

- Developed Five SST Models and three submodels this summer.
- Implementation for detecting five of six problems.
- Documentation of code and models.
- Check it out! https://github.com/lpsmodsim/2022HPCSummer

Questions?