2022 Future Computing Summer Internship Project: Modeling TCP Global Synchronization in a Discrete-Event Simulator to Find Metrics to Measure its Existence in a Simulation

Nicholas Schantz*

August 5, 2022

Abstract

TCP Global Synchronization is a networking problem in which a burst of traffic in a network causes multiple clients to drop packets and limit their transmission rates. Afterwards, the clients begin to increase their transmission rates consecutively which results in more packet loss and transmission limiting, which creates a loop of this activity. This research addresses the question as to whether metrics exist to determine if this problem has occurred in a simulated network. These metrics are useful for network architects who are unaware of this problem, because they can better understand how to avoid causing this problem in a network simulation. The discrete-event simulator (DES) framework named Structural Simulation Toolkit (SST) is used to simulate this activity and find a metric. An SST model is created of a simple reliable network where components send data to a receiving component who will drop data when its queue is filled. Global synchronization is caused in the simulation and data is collected from the model's components to determine metrics for detecting global synchronization. A metric is to look in a window of activity when packet loss occurs and measure the number of sending components that have reduced their transmission rates.

1 Project Description

The challenge addressed by this work is to map the networking problem TCP global synchronization to a discrete-event simulator. Sandia National Laboratories' Structural Simulation Toolkit is a discrete-event simulator framework which is used to simulate the problem. Simulation output is collected to help identify mathematical or logical conditions that can cause this problem. This information is used to develop metrics to identify that the problem has occurred in simulated distributed systems.

2 Motivation

Identifying the metrics to detect TCP Global Synchronization will be vital for developing distributed systems that can avoid global synchronization from occurring during network communication. Network architects may be unaware of this problem when they begin to model and simulate a network. Therefore, the metrics found in this research can help network architects catch this problem when synchronization occurs during the simulation's development. These results allow for the problem to be prevented during simulation so it does not occur when the system is put into production.

A second goal of this work is to develop examples and documentation of SST simulations that new users can utilize. Currently, SST is primarily used for High-Performance Computing (HPC) simulations and the majority of SST examples revolve around modeling HPC systems. However, the SST framework utilizes a powerful discrete-event simulator that can model more than just HPC systems. The models and documentation created for this project show off the discrete-event simulator for simulations not relating to HPC systems.

*nschantz3@gatech.edu

3 Prior work

Prior research by Bashi and Al-Hammouri [1] explains in detail on how the global synchronization can occur in a network. The authors simulate network traffic to show synchronization occurring when all senders decrease their congestion window at the same time. Congestion window is the number of packets a sender will send before needing an acknowledgment. This information was used to create a simple SST model explained in this report that can detect if global synchronization is occurring by measuring the number of clients that limit their transmission rates in an interval of time.

4 Running the Model

The software developed for this challenge was run on one laptop running an Ubuntu-Based Linux operating system. The software is available at https://github.com/lpsmodsim/2022HPCSummer-TCPGlobalSynchronization.

Assuming the user is on a system running a Ubuntu-Based Linux Distro. To run the software:

Prerequisites:

```
sudo apt install singularity black mypi
git clone https://github.com/tactcomplabs/sst-containers.git
```

Follow the instructions in the git repo to build the container "sstpackage-11.1.0-ubuntu-20.04.sif".

```
cp sst-containers/singularity/sstpackage-11.1.0-ubuntu-20.04.sif /usr/local/bin/
git clone https://github.com/lpsmodsim/2022HPCSummer-SST.git
sudo . /2022HPCSummer-SST/additions.def.sh
```

Obtaining and running the model:

```
git clone https://github.com/lpsmodsim/2022HPCSummer-TCPGlobalSynchronization
cd 2022HPCSummer-TCPGlobalSynchronization
make
```

To re-run the software:

```
make clean
make
```

Expected output:

```
mkdir -p .build
singularity exec /usr/local/bin/additions.sif g++ -std=c++1y -D\_\_STDC\_FORMAT\_MACROS -
    ↪ fPIC -DHAVE\_CONFIG\_H -I/opt/SST/11.1.0/include -MMD -c receiver.cc -o .build/
    ↪ receiver.o
mkdir -p .build
singularity exec /usr/local/bin/additions.sif g++ -std=c++1y -D\_\_STDC\_FORMAT\_MACROS -
    ↪ fPIC -DHAVE\_CONFIG\_H -I/opt/SST/11.1.0/include -MMD -c sender.cc -o .build/
    ↪ sender.o
singularity exec /usr/local/bin/additions.sif g++ -std=c++1y -D\_\_STDC\_FORMAT\_MACROS -
    ↪ fPIC -DHAVE\_CONFIG\_H -I/opt/SST/11.1.0/include -shared -fno-common -Wl,-
    ↪ undefined -Wl,dynamic\_lookup -o libtcpGlobSync.so .build/receiver.o .build/sender
    ↪ .o
singularity exec /usr/local/bin/additions.sif sst-register tcpGlobSync tcpGlobSync\
    ↪ _LIBDIR=/home/{USER}/sst-work/2022HPCSummer-TCPGlobalSynchronization
singularity exec /usr/local/bin/additions.sif black tests/*.py
singularity exec /usr/local/bin/additions.sif mypy tests/*.py
Success: no issues found in 2 source files
singularity exec /usr/local/bin/additions.sif sst tests/tcpGlobSync.py
(Simulator console output)
```

The simulation can be modified by editing the python driver file, which is located at:

```
2022HPCSummer-TCPGlobalSynchronization/tests/tcpGlobSync.py
```

5 SST Model

The model is a simplified version of a reliable network protocol with a tail drop queue management policy. It is made up of sender components and receiver components. The following assumptions are used in the model:

- Sender components use the same protocol.
- Sender components limit their transmission rates to the same value.
- Transmission rate limiting occurs immediately after a sender is notified that its packet was dropped.
- Sender components increase their transmission rates linearly after each tick.
- Sender components continuously send messages during the entire simulation.

The model involves a receiver component that has n ports which connect to n sender components. The sender components will send packets to the receiver's first in, first out (FIFO) queue, and the receiver will process a set number of packets in the queue per tick.

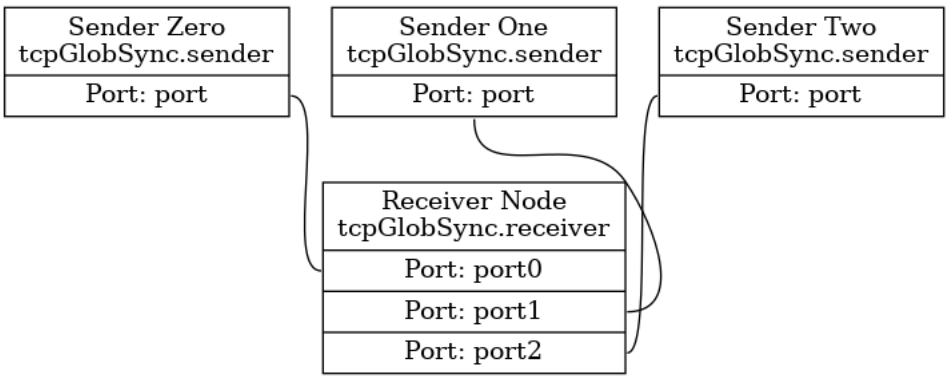

Figure 1: Connection between 3 sender components and 1 receiver component.

6 Result

The metric capable of detecting TCP global synchronization is to first look at the time in which senders limit their transmission rates. If all senders limit their transmission rates in a set interval of time (which will be referred to as a window), the time in which this behavior occurred is logged. The difference in time between these points is calculated and averaged resulting in the metric.

In the simulation, the receiver component measures this behavior. The receiver will monitor the senders connected to it. When a sender limits its transmission rates, the receiver will begin sampling input for a window of time. If all sender limit their transmission rates in the window, the problem's behavior is detected in the simulation. This metric is demonstrated on three simulations which simulate three different scenarios of network traffic:

The first simulation is of three senders that have a combined transmission rate that is under the receiver's capacity. The receiver's queue will not fill so packet loss will not occur, and senders will not limit their transmission rates; see Figure 2. In this scenario, global synchronization is not occurring so the metric will not detect this behavior in the simulation; see Figure 3.

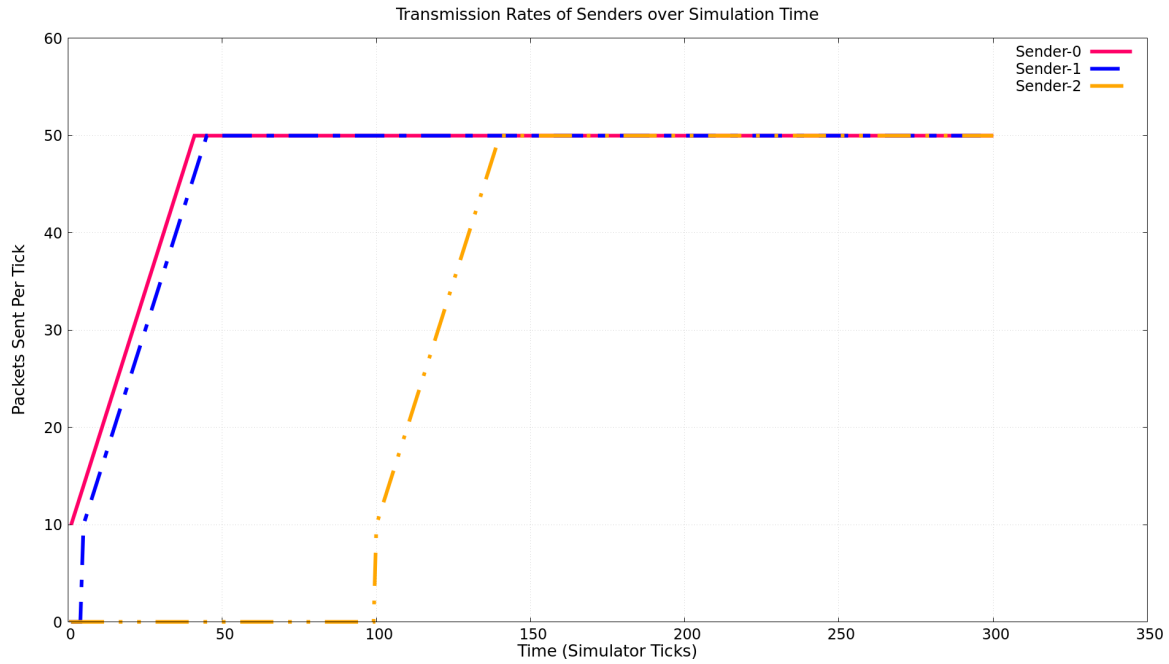

Figure 2: Transmission rates of the three senders. Transmission rate limiting does not occur so global synchronization is not detected.

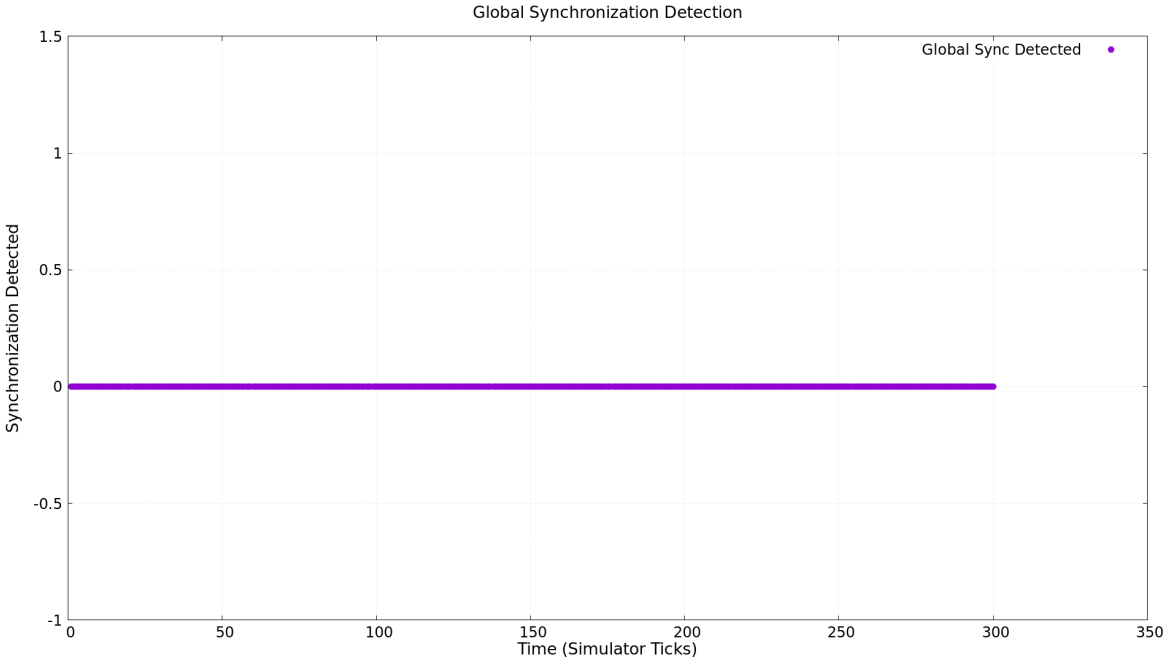

Figure 3: Global Synchronization is not detected in simulation one.

The second simulation contains two senders that send packets to a receiver at a rate in which the queue does not fill up so packet loss can not occur. A third client is introduced after one hundred seconds and disrupts the network by sending packets which causes the queue to fill and packets to be dropped. All clients send packets consecutively when the packet loss occurs so they all encounter packet loss and limit their transmission rates consecutively; see Figure 4. This creates the problem of global synchronization as they attempt to increase their transmission rates back over time which leads to more packet loss and rate limiting. In this scenario, global synchronization occurs and the metric detects it; see Figure 5. The difference in time

between each of the pairs of points is calculated and the average is calculated for all points in the simulation output; see Figure 6.

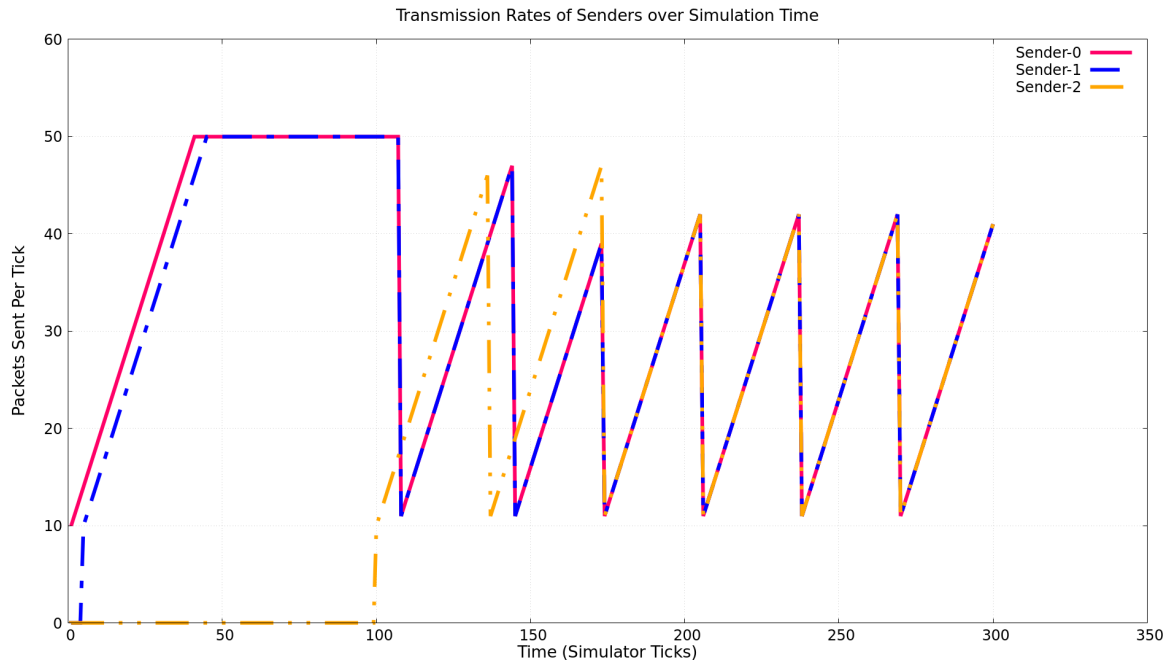

Figure 4: Transmission rates of the three senders over time.

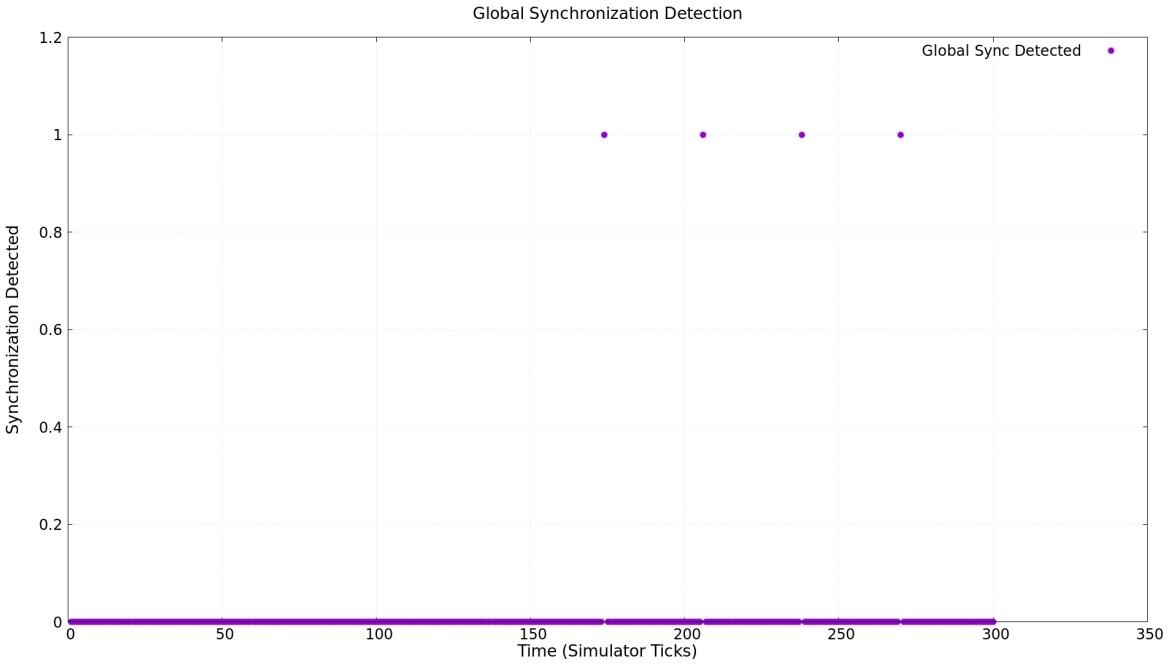

Figure 5: Metric detecting the existence of the problem in simulation two.

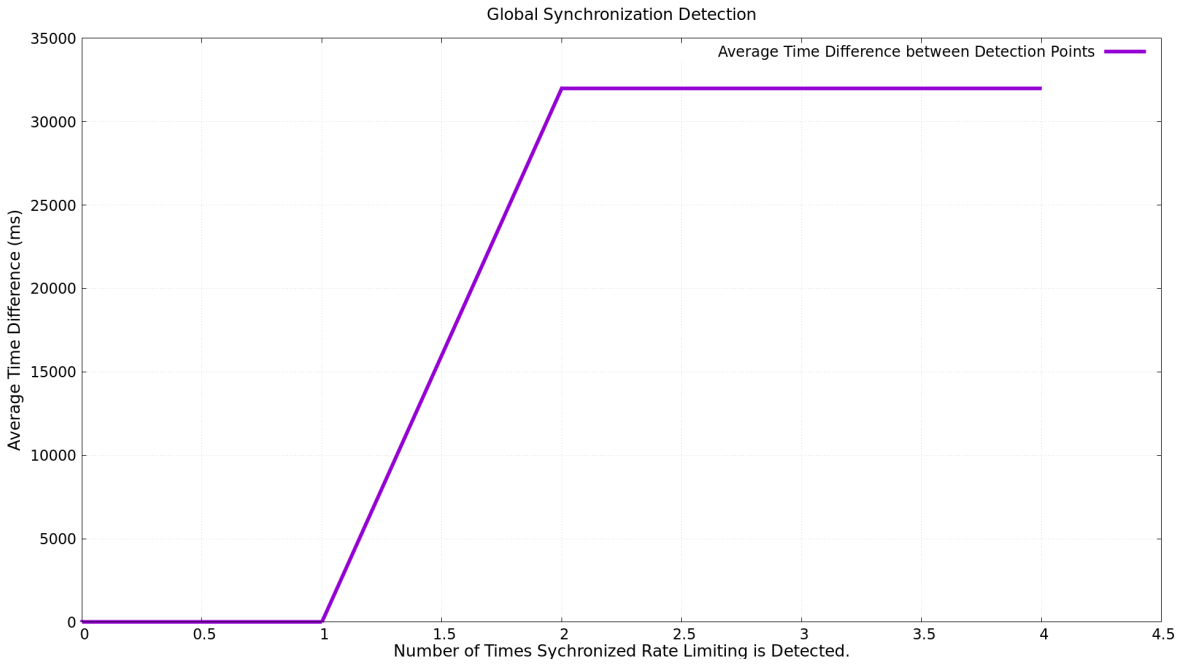

Figure 6: Average difference in time between global synchronized rate limiting during simulation runtime.

The third simulation contains the same parameters as simulation two. However, a simple algorithm inspired by Random Early Detection (RED) is incorporated into the simulation to prevent global synchronization from occurring. This algorithm runs every time the receiver receives a packet and works by dropping random packets after the queue size becomes greater than an user-defined size. This will cause senders to limit their transmission rates early and at random intervals; see Figure 7. This is used to demonstrate the metric in an environment where global synchronization could potentially occur but does not. This is to test if the metric can produce false positives; see Figure 8.

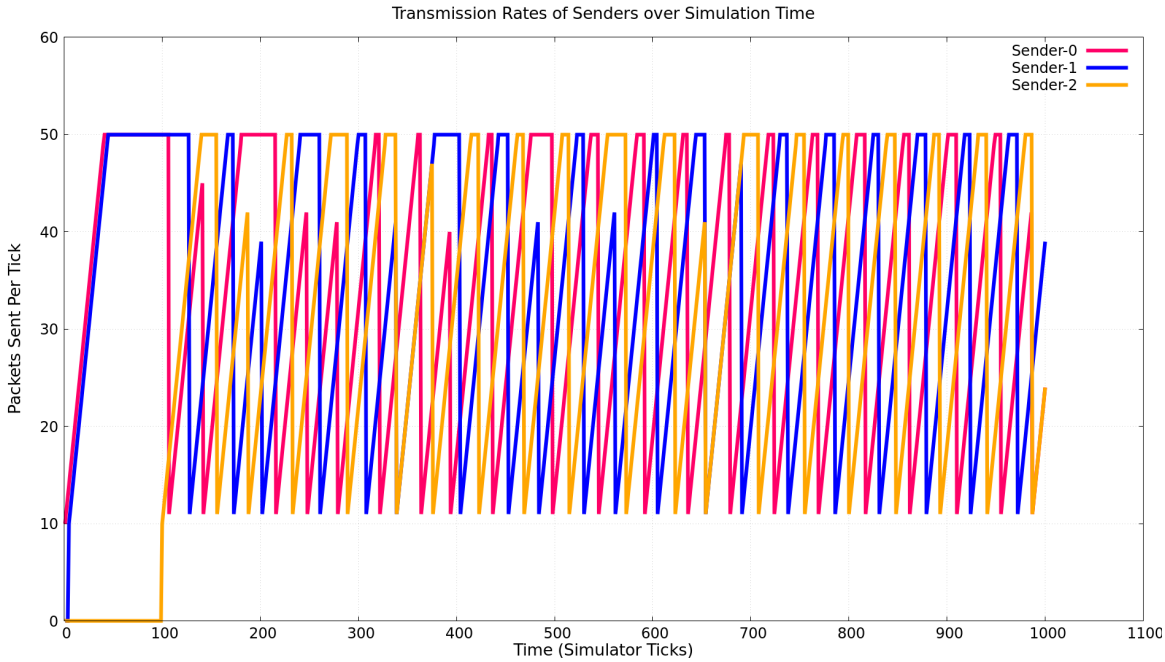

Figure 7: Transmission rates of the three senders over time.

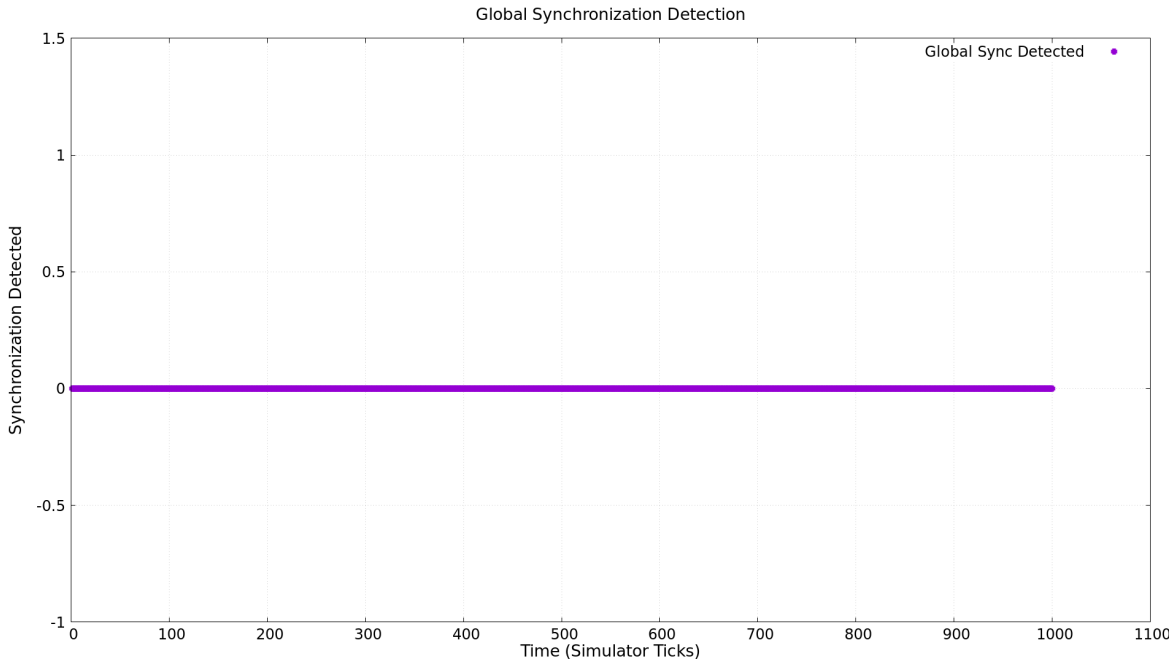

Figure 8: Metric does not detect global synchronization in simulation three.

Expanding the window size for sampling will cause the metric to produce false positives. The following simulation has a sampling window size of forty seconds. Visually, the transmission rates of the senders do not synchronize in Figure 9. However the metric produces unintended results; see Figure 10.

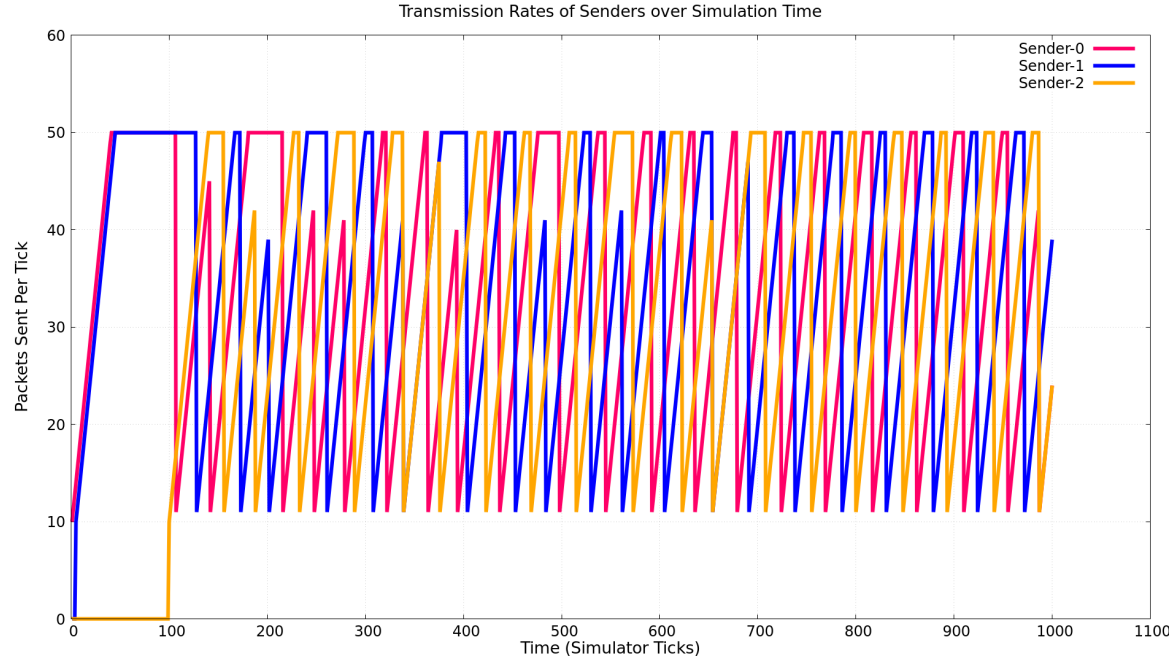

Figure 9: Transmission rates of the three senders over time.

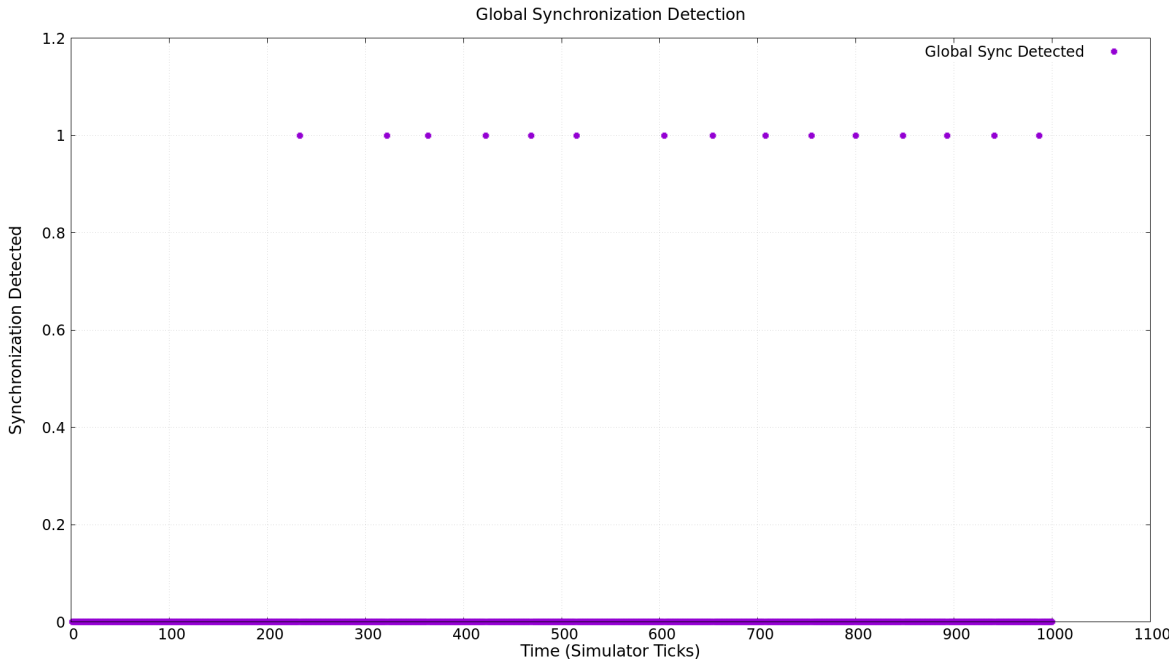

Figure 10: A large window size (forty seconds) causes false positives in detection.

Notice that the average time difference between points will fluctuate during a false positive; see Figure 11. This is useful to differentiate false positives from true positives where the average time difference is constant after the first two detections; see Figure 6.

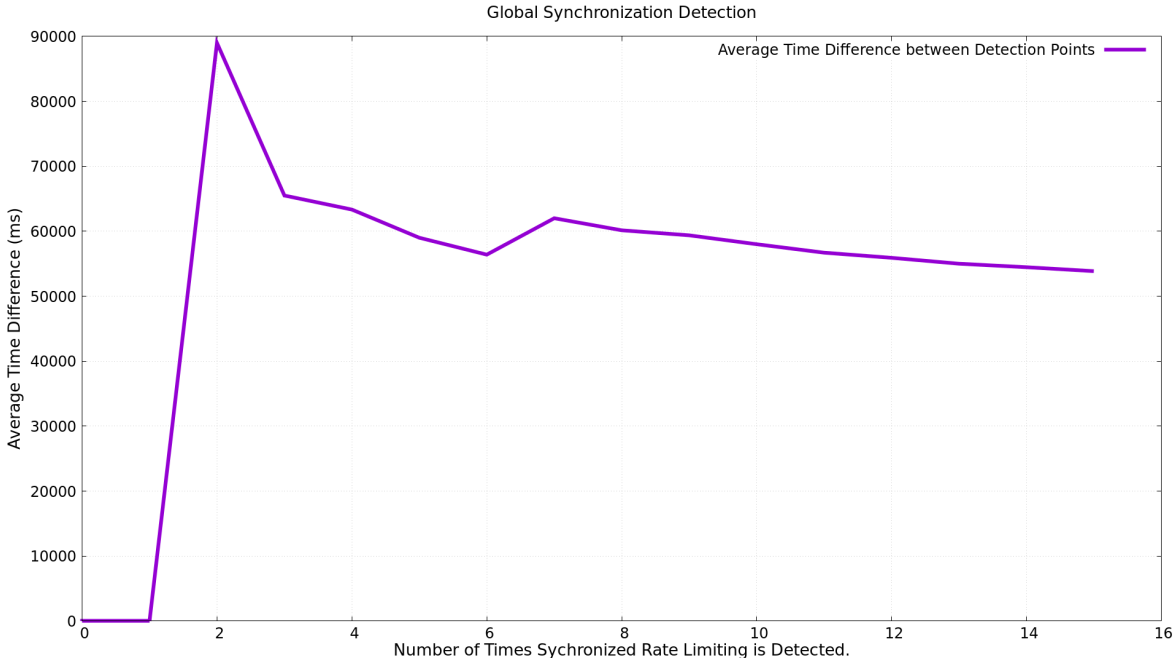

Figure 11: Average time between detection points fluctuates during a false positive.

The metric results from an algorithm ran during the simulation, so it can be analyzed or used during runtime. The data it produces is also output to a log file for analysis after the simulation has ended.

The receiver component runs the algorithm in the SST model, but it could be generalized to any node capable of collecting information on all senders that are connected to a receiver. The following C++ code snippets show the implementation of the algorithm that results in the metric:

In the node's setup function:

```
void receiver::setup() {
        ...
        // Dynamically allocate an array to keep track of what sender components have limited
            ↪ transmission rates during a window of time.
        tracked_senders = (int*) calloc(num_senders, sizeof(int));
        ...
}
```

In the node's tick/update function:

```
bool receiver::tick(SST::Cycle_t currentCycle) {
        ...
        // Check if the window time for sampling has ended.
        if (sampling_status == true && (getCurrentSimTimeMilli() >= sampling_start_time +
            ↪ window_size)) {
                already_sampled = false;
                sampling_status = false;

                // Reset variable/array data.
                sampling_start_time = 0;
                senders_limited = 0;
                for (int i = 0; i < num_nodes; i++) {
                        tracked_senders[i] = 0;
                }
        }
        ...
}
```

In the node's event handler that captures when senders have limited their transmission rate:

```
void receiver::eventHandler(SST::Event *ev) {
        ...
        // Receives an event that a sender has limited their transmission rates.
        case LIMIT:

        /**
                Background:
                "pe" is event sent by a sender component that limited their transmission rate.
                "pe->pack.node_id" is the specific sender that sent the event.
        */

        // When the node is not sampling and receives an event that a sender is limiting transmission
            ↪ rates,
        // it begins sampling for other transmission rate limiting in the set window time.
        if (sampling_status == false && already_sampled == false) {
                sampling_start_time = getCurrentSimTimeMilli(); // Set start time of sampling.
                sampling_status = true;  // Begin sampling.
                tracked_senders[pe->pack.node_id] = 1; // Set the sender that caused sampling to
                    ↪ begin to
                senders_limited++; // Increase the number of senders that have limited their
                    ↪ transmission rates.
        }

        // During sampling, check if the sender that limited their transmission rate is new, and that
            ↪ sampling has not already ended early (behavior has already been detected in sampling
            ↪ window).
```

```
        if (sampling_status == true && tracked_senders[pe->pack.node_id] == 0 &&
            ↪ already_sampled == false) {
                tracked_senders[pe->pack.node_id] = 1;
                senders_limited++;

                // Check if all seperate senders have limited their transmission rates in the sampling
                    ↪ window time.
                if (senders_limited == num_nodes) {
                        // Global sychronization behavior has occured.

                        // Log time sychronization was detected
                        new_globsync_time = getCurrentSimTimeMilli();
                        num_globsync++; // Increment count of detections

                        // For detections after the first one, take the difference in times between
                            ↪ detection points and average them.
                        if (num_globsync != 1) {
                                globsync_time_diff_avg = (total_time_diff + (
                                    ↪ new_globsync_time - prev_globsync_time)) / (
                                    ↪ num_globsync - 1);
                                total_time_diff = total_time_diff + (new_globsync_time -
                                    ↪ prev_globsync_time);
                        }

                        prev_globsync_time = new_globsync_time;

                        globsync_detect = 1; // Set boolean metric to true.
                        already_sampled = true;  // End sampling early.

                        // Reset variable/array data.
                        senders_limited = 0;
                        for (int i = 0; i < num_nodes; i++) {
                                tracked_senders[i] = 0;
                        }
                }
        }
        break;
        ...
}
```

7 Future Work

It would be useful to try increasing the fidelity of the simulation and comparing if the detection metric is still accurate. Examples include incorporating sliding window protocol, different queue dropping policies such as WRED, congestion window, and traffic prioritization.

References

[1] Yahya Bashi and Ahmad Al-Hammouri. Transmission control protocol global synchronization problem in wide area monitoring and control systems. 08 2017.