2022 Future Computing Summer Internship Project: Modeling Congestive Collapse in a Discrete-Event Simulator to Measure its Existence in a Simulation.

Nicholas Schantz

August 4, 2022

Abstract

Congestive collapse is an issue involving reliable network protocols where a large amount of retransmitted data clogs a network. This causes the network's useful throughput to decline. 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 network simulations. The discrete-event simulator (DES) framework called Structural Simulation Toolkit (SST) is used to simulate this activity and find metrics to determine if it exist in a simulation. An SST model of a simple reliable network in which multiple nodes send packets to one receiving node. The receiving node is flooded with retransmitted data over time to create congestive collapse in the network. Data from the receiving node is analyzed and results reveal that a receiver's queue depth and ratio of first packets to duplicate packets directly measure the existence of congestive collapse. The ratio of goodput to total throughput can also measure congestive collapse.

1 Project Description

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

2 Motivation

Identifying the metrics for detecting congestive collapse will be vital for developing distributed systems that can avoid congestive collapse from occurring during communication. Network architects may be unaware of this problem when they begin to model and simulate a network. The metrics found in this research can help them catch this problem when it occurs during the simulation's development. The SST models created will demonstrate how the problem can occur in a simulation. These results allow for the problem to be avoided during simulation so it does not occur when the system is put into production.

A second outcome 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.

3 Prior work

Multiple research papers exist describing the problem of congestive collapse and how it was solved. One of the earlier reports by Van Jacobson[1] described the algorithms put into place after congestive collapse occurred

two years prior at the University of California, Berkeley. The information on what the new algorithms corrected in reliable networking is useful for figuring out if a network is prone for congestive collapse. The paper also explains how senders without congestion avoidance can increase congestion in the network by sending data above the wires bandwidth, which was useful information to determine that increased packet queuing can lead to packets being retransmitted which can spiral into congestive collapse (See [1, p.325]).

Similarly, in the tech report by J. Nagle[2], he reports that a primary cause for congestive collapse is due to spurious retransmissions in the network due to inefficient timeout delay algorithms. This information was used for the congestive collapse SST model to simulate congestive collapse occurring on a simplified reliable network.

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 on https://github.com/lpsmodsim/2022HPCSummer-CongestiveCollapse.

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
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

git clone https://github.com/lpsmodsim/2022HPCSummer-CongestiveCollapse
cd 2022HPCSummer-CongestiveCollapse
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 libcongestiveCollapse.so .build/receiver.o .build/sender.o
singularity exec /usr/local/bin/additions.sif sst-register congestiveCollapse
    ↪ congestiveCollapse_LIBDIR=/home/{USER}/sst-work/2022HPCSummer-CongestiveCollapse
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/congestiveCollapse.py
(Simulation console output)
```

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

```
2022HPCSummer-CongestiveCollapse/tests/congestionCollapse.py
```

5 The Model

The model is made up of sender components and receiver components who send and receive packets respectively. There is n senders connected to one receiver. The senders will send a set number of packets to the receiver and the receiver will enqueue the packets to an infinite queue. The receiver will process a set number of packets and respond to the senders with acknowledgments for the packets it has processed. The sender keeps track of every packet sent and if it does not receive an acknowledgment for that packet in a set time, it will retransmit the packet.

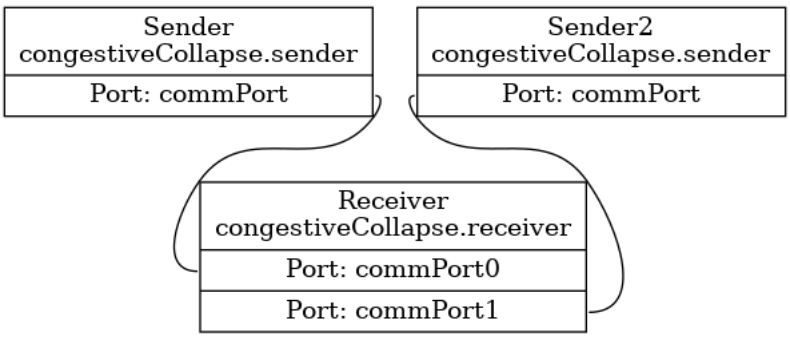

Figure 1: Example of model with two senders and one receiver component

Key components that are logged by the receiver component are as follows:

- Throughput - the number of new and retransmitted packets that are processed by the receiver per second.
- Goodput - the number of new packets that are processed by the receiver per second.
- Queue Depth - The number of packets in the queue.
- Queue Entries - Number of new and retransmitted packets entering the queue per second.

6 Result

To determine if congestive collapse exist in a network. Three candidate metrics were determined:

- Ratio of new packets to retransmitted packets in a receiver's queue.
- Receiver's queue depth.
- Ratio of goodput and throughput of the receiver.

The ratio of new packets and total packets in the receiver's queue will give an idea if more retransmitted packets are being enqueued by the receiver if the ratio decreases over time. However, only using this metric could produce false positives for detecting congestive collapse. For example, during a situation where there is low network traffic and the receiver's queue is close to empty, if one sender happens to retransmit packets that were previously lost, this will decrease the measurement even though congestive collapse did not occur. To resolve this issue, queue depth will also be measured to determine if there is a low ratio of new packets and total packets while the receiver's queue depth remains consistently high. In this case, a network with a high queue depth is clogged and if its ratio of packets insist of mostly retransmitted packets, the probability that congestive collapse exist in the network is high.

An example of the problem is simulated and gathered data is shown below. In this SST simulation, a sender component is sending packets at three times the rate that a receiver component can process them at. The receiver's queue fills quickly (Figure 3) and the receiver is unable to send acknowledgments under the sender's timeout time. Retransmissions are sent which cause more packets to fill the queue and the ratio of new packets to total packets in the queue exponentially decays to zero; see Figure 2.

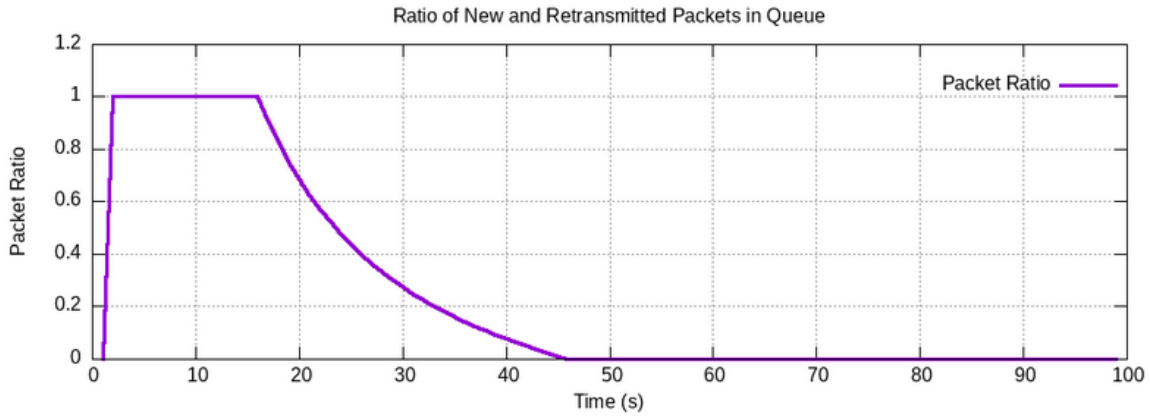

Figure 2: Amount of new packets entering the queue each second for one hundred second

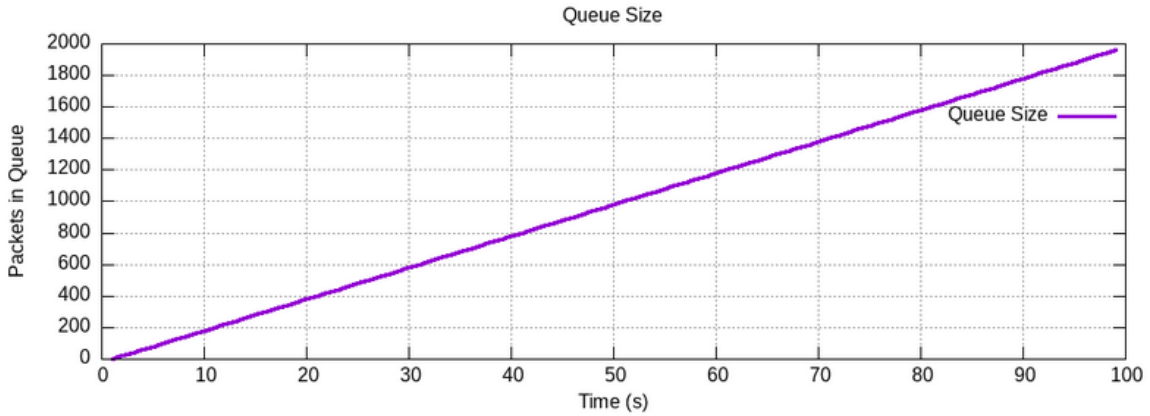

Figure 3: Infinite queue which increases over time.

As the queue grows, starting at around 15 seconds, the ratio of new and retransmitted packets in the queue begins to exponentially decay. In a short span of time, only retransmitted packets are being sent to the receiver since the receiver cannot process the packets in its queue. This will result in useful throughput dropping to zero over time; see Figure 4.

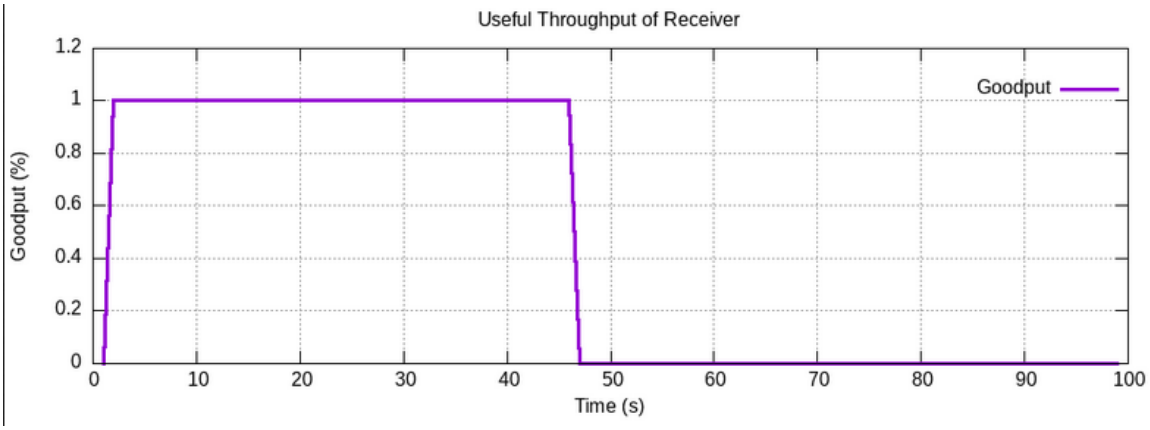

Figure 4: Ratio of goodput and total throughput over one hundred seconds

7 Future Work

Currently, the user must define some threshold to declare that the simulation has encountered congestive collapse. It might be better if the metric measures the trend of data during simulation. For instance, the ratio of new packets to retransmitted packets in the receiver's queue can be analyzed during runtime, and if the change in data appears to be following an exponential decay fit, then the probability that congestive collapse exist increases. Similarly, if the queue size is increasing over time, then this adds to the probability that congestive collapse exists.

References

[1] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, aug 1988.

[2] J. Nagle. Congestion control in ip/tcp internetworks. RFC 896, RFC Editor, January 1984.