Time-Sensitive Networking Visualization in OMNeT++ Sequence Chart

Elliot A. Gorman

[a] *McMaster University*

[b] *McMaster Centre for Software Certification – Department of Computing and Software*

[c] *Supervisor: Dr. Victor Bandur*

Abstract

Time-Sensitive Networking (TSN) networks, particularly those with a multitude of switches, are complicated to test and debug. Assessing a TSN network's behavior is much easier done in simulation than with a physical network. OMNeT++ Discrete Event Simulator paired with the INET framework (which recently implemented a subset of the TSN-related IEEE standards) proves effective for creating Ethernet-based TSN network simulations. The Sequence Chart, OMNeT++'s offering for visualizing the runtime of simulations, provides highly customizable visuals. However, it lacks TSN-specific visuals which would complete the overall toolchain's compatibility with TSN development. To address this, we extend the Sequence Chart by implementing TSN-specific visualization settings.

Keywords: Time-Sensitive Networks, TSN, OMNeT++, INET, Network Visualization

1. Introduction

Note, Time-Sensitive Networking (TSN) refers to the set of standards from the Time-Sensitive Networking Task Group part of the IEEE 802.1 Working Group[1]. Hereon a TSN network shall refer to any Ethernet-based network which makes uses of the bounded latency time-aware shaping feature specified in IEEE Std 802.1Q-2022[2], or the superseded IEEE Std 02.1Q-2018[3] and IEEE Std 802.1Qbv-2015[4].

Switches in TSN networks transmit traffic based on traffic shaper mechanisms. Relevant to this project is the time-aware shaper, which operates on user-configured cyclical schedules. The schedules define time slots in which specified classes of traffic are selected for egress. Each traffic class has an associated egress queue within a switch's Ethernet interface in which frames of that traffic class are queued until the queue's transmission gate is open (meaning the schedule is allowing transmission).

The key improvement over non-TSN Ethernet networks is that the timing of packet delivery throughout the network is more deterministic, that is, if the schedules are properly configured.

For a TSN network to function properly, the schedules must be carefully predetermined and encoded into the Ethernet interface of each switch. Improperly configured schedules can lead to traffic bottlenecks. Furthermore, it is a requirement that each node in the network be synchronized to the same global time base using the Generalized Precision Time Protocol (gPTP). Any misalignment in synchronization in a switch can disturb traffic flow, possibly also causing bottlenecks or starvation.

Diagnosing such issues and ensuring that they do not occur during the network development stage is much simpler done in a simulated environment than with physical hardware. In simulation it is easy to probe into the lower layers of switches and see the status of queues, gates, and clocks.

OMNeT++ Discrete Event Simulator, distributed under the Academic Public License, is an Eclipse distribution for creating module-based event-driven simulations. Simulation models are built by first defining the nodes of the network, referred to as modules, using the NED language and C++. The

[1] https://1.ieee802.org/tsn/
[2] https://standards.ieee.org/ieee/802.1Q/10323/
[3] https://standards.ieee.org/ieee/802.1Q/6844/
[4] https://standards.ieee.org/ieee/802.1Qbv/6068/

connection between modules is then specified using the NED language, forming a network topology of interacting modules that communicate with each other through message passing. In addition, the INET framework for OMNeT++ provides modules for the development of communication network simulations (such as Ethernet), including TSN.

Visualization of simulations, once the runtime has ended, is provided in the form of the Sequence Chart, an OMNeT++ Eclipse plugin. It plots events and traffic between modules in chronological order. The user can filter and select exactly which events, traffic types, and modules are displayed. However, while one could plot the status of gates and queues within each switch, the result is not visually intuitive, since it would consist of plotting single points in which a relevant event happened. Trying to understand the status of a TSN switch based on this would involve mentally piecing together each plotted event.

2. Objective

We want to extend the OMNeT++ Sequence Chart to include TSN-specific visualization capabilities. Such an addition would increase the value of the OMNeT++ toolchain for TSN development and research.

The scope of this project is limited to the visualization of gates and egress queue fills for each traffic class in use, per switch Ethernet interface. However, this extension can be easily modified to add other visualizations for additional insight into the status of TSN switches.

3. Method

OMNeT++ (and by extension INET) is event-driven, so any action of interest (e.g. frame egress) is considered an event and has a timestamp and unique event number. During simulation runtime, events pertaining to a particular module can be logged into a vector — essentially a dynamic array — which is then accessible via a data file after the runtime has ended. Aside from vectors, which are created by modules voluntarily, an event log is generated that stores all events and related information. It is this log that the Sequence Chart takes as input.

Each module plotted in the Sequence Chart has a horizontal time axis spanning the screen. Any events related to a module are then plotted using the module's axis as a starting/ending point. For example, if a module sends a message to itself, a line with an arrow tip is drawn from the module's axis back to the same axis, starting at the time of egress and ending at the time of ingress.

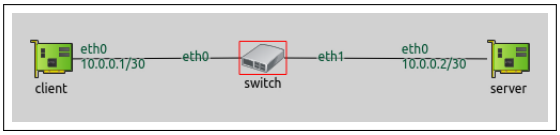
To visualize the gates and queues, the gate state changes (tracked by the `PeriodicGate` INET class) and queue fill (tracked by the `PacketQueue` INET class) events must be recorded into a vector during runtime. Fortunately, both classes already populate vectors with this data by default — the `gateStates` vector and `queueLength` vector contain the respective information in enough detail for our purposes.

A feature that is already part of the Sequence Chart is the ability to attach a vector to an axis. This functionality is what we extend to overlay TSN visuals. The `AxisVectorBarRenderer` class can display a vector chosen by the user across the axis of a chosen module. To display the gate schedules for each traffic class in use by a switch's Ethernet interface (resulting in anywhere from one to eight vectors being displayed), we implement the `AxisMultiVectorBarRenderer` class, which can render multiple vectors across a single axis. Furthermore, we also implement the `TrafficClassInfo` class and `EthIf` class. The `TrafficClassInfo` class stores the `gateState` and `queueLength` vectors for a given gate. The `EthIf` class represents a switch's Ethernet interface and stores the `TrafficClassInfo` for all gates in the switch being represented.

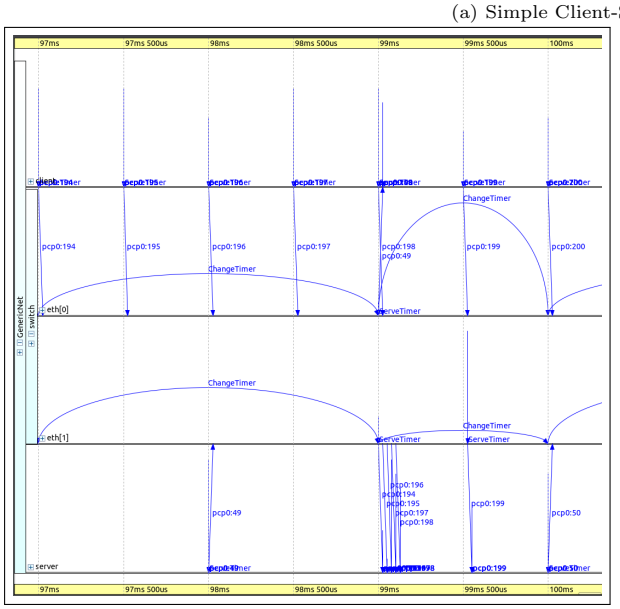
The attach vector to axis menu is then modified such that if the user selects any `gateState` vector to be displayed across an axis, Sequence Chart will automatically find all other `gateState` and `queueLength` vectors part of the same Ethernet interface, and pass them to the `AxisMultiVectorBarRenderer` which displays the schedules as contiguous rectangles across the axis.

4. Results

For demonstration, we start with a simple network with only one traffic class.



(a) Simple Client-Server Network



(b) Sequence Chart snippet for (a) without TSN visuals

(c) Sequence Chart snippet for (a) with TSN visuals

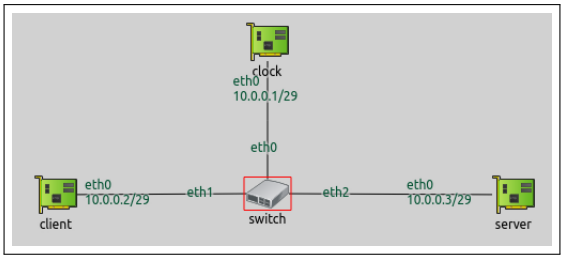
Figure 1: Simple Client-Server Network

In Figure 1 a simulation was run using the simple client-server network shown in (a). Frames were sent from the `client` module every 500 microseconds to the `server` module, and from the `server` to the `client` every 2000 microseconds. All traffic has the same traffic class, and synchronization via gPTP is omitted for simplicity. The `switch` module's Ethernet interfaces open the gate (written as `GATE 0` in (c)) that controls the one traffic class's flow for 1 millisecond and then closes it for 2 milliseconds.

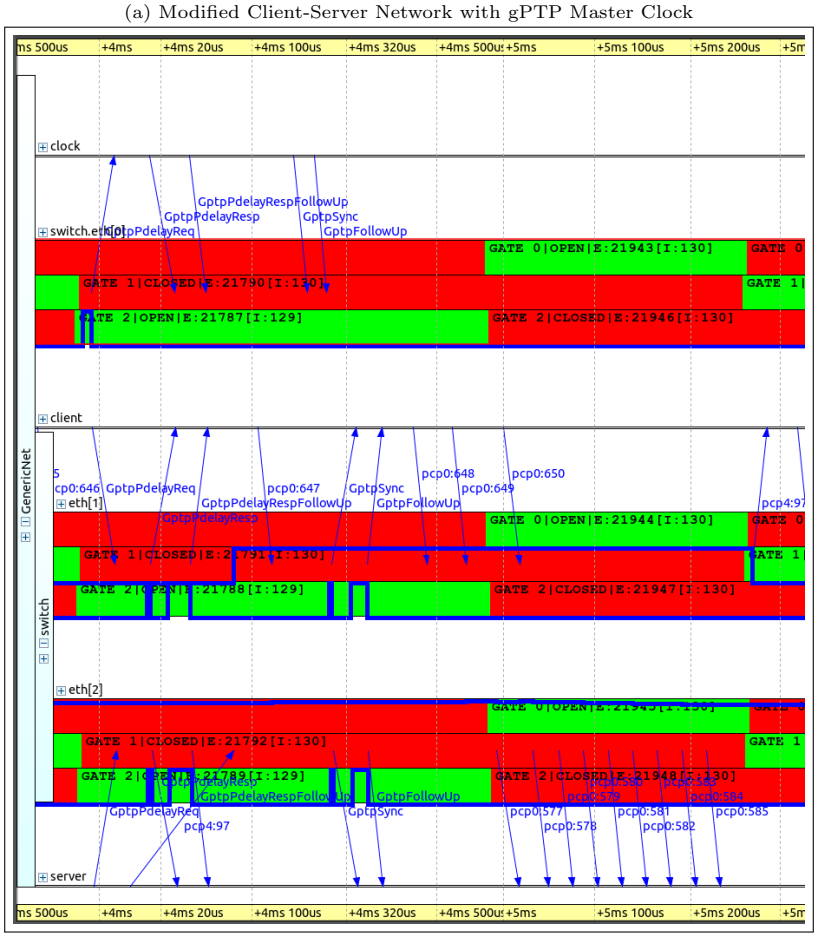
Part (b) of Figure 1 shows a snippet of the resulting Sequence Chart without TSN visuals. All events are filtered out except for the frames sent between `client` and `server` (displayed as `pcp0:{frame number}`), and the timer controlling the gate (displayed as `ChangeTimer`). The modules/axes shown from top to bottom are: the `client` module, the `switch` Ethernet interface connected to the `client` (`switch.eth[0]`), the `switch` Ethernet interface connected to the `server` (`switch.eth[1]`), and the `server` module. The `ChangeTimer` is a self-message clock event that is sent and received at each opening or closing of the gate, and by default the only insight into the state of the gate.

Finally, part (c) of Figure 1 shows the sequence chart with our TSN visuals. The chart has the same filtering as (b) (except for the `ChangeTimer` events), and the order of the modules is kept. The schedule for `GATE 0` is drawn on the axis for `switch.eth[0]` and `switch.eth[1]`. The solid red component of the rectangular schedule indicates that the gate is closed, and the green component indicates that it is open. The varying blue line is the queue fill — the line increases in verticality the more a queue is populated. Since there is no specified limit to queue length, the queue fill line is linearly interpolated to the height of the rectangle, so a line at the bottom of the rectangle indicates an empty queue and a line at the top indicates that the queue fill is at its highest for the simulation run.

Extending the simple client-server network showcases the utility of the TSN visuals for detecting issues with TSN configuration.



(a) Modified Client-Server Network with gPTP Master Clock



(b) Sequence Chart snippet for (a) with TSN visuals

Figure 2: Modified Client-Server Network

Here, the client-server network has been extended with gPTP functionality (the master clock is the `clock` module) and slightly modified. Frames from the `client` to the `server` are produced every 300 microseconds, and traffic from the `server` to the `client` is now of a different traffic class (written as `GATE 1` in (b)). The gPTP traffic also has its own traffic class (written as `GATE 2` in (b)), resulting in a total of three different classes. Lastly, the schedule is as follows:

Table 1: Gate timings used in the `switch` module

Time	Gate 0	Gate 1	Gate 2
0ms to 0.25ms	open	closed	closed
0.25ms to 2ms	closed	open	closed
2ms to 3ms	closed	closed	open

It is immediately evident from the Sequence Chart visuals that there is a significant bottleneck in traffic passing through gate 0. The queue fill line for gate 0 hardly descends at all when the gate is open,

meaning that the switch cannot empty the queue fast enough before the gate is closed, resulting in an infinitely increasing queue fill. On a real system, this would result in a sizable delay of traffic for the affected traffic class, on top of dropped frames.

5. Further Work

There are several quality-of-life improvements that this extension could benefit from. For example, exact queue fill numbers would be helpful to see, but drawing them directly onto the schedule is bound to increase clutter, so a hover-over tooltip displaying precise info may be ideal.

Furthermore, some granularity as to which visuals are displayed would be an improvement. Currently, if the user selects a `gateState` vector to display over an axis, both the gate schedule and queue fill are displayed, whereas showing only one (or neither) could be valuable to the user.

The source code of this extension is available on GitHub [5].

[5] https://github.com/gormae1/SequenceChartTSN-ify