The Preserve: Lehigh Library Digital Collections

An Enhanced Tool for Simulating the End-to-End Network Delays of Cyber Physical Systems

Citation

Gray, Dylan M., and Liang Cheng. *An Enhanced Tool for Simulating the End-to-End Network Delays of Cyber Physical Systems*. 2019, https://preserve.lehigh.edu/lehigh-scholarship/graduate-publications-theses-dissertations/theses-dissertations/enhanced-tool-0.

Find more at https://preserve.lehigh.edu/

This document is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

An Enhanced Tool for Simulating the End-to-End Network Delays of Cyber Physical Systems

by

Dylan Gray

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science in Computer Science

Lehigh University

August 2019

© Copyright by Dylan Gray 2019

All Rights Reserved

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____

Date

_____

Thesis Advisor: Liang Cheng

_____

Chairperson of Department: Jeff Trinkle

Acknowledgements

I would like to thank …

Professor Liang Cheng for sharing his advice, guidance, and help throughout this project and the rest of the time during my studies. Ph.D. students Huan Yang and Isaac Howenstine for their ideas and feedback that was invaluable for my work. The faculty and staff of Lehigh University's CSE Department for providing a great learning environment and helping me achieve educational goals. Most importantly, my family for loving and supporting me throughout this entire experience.

Table of Contents

List of Tables

List of Figures

Abstract

In this paper, we present improvements to a network simulation tool that allows us to more accurately determine the maximum end-to-end delay for data flows of a cyber physical system. We discuss the existing solutions for determining end-to-end delays of a network and the flaws in these methods. Then we explain how the development of a simulation tool can enhance the transition from traditional systems, with a requirement for the maximum end-to-end delay, to safe cyber physical systems. We briefly discuss the field of mathematics called network calculus to show the ideas behind the simulation. We then discuss the base simulation tool and its uses in calculating end-to-end delay. We also mention the existing and potential optimizations to the simulation tool concerning the use of advanced data structures. These include the use of Fibonacci heaps and b-trees as a way to improve performance and add new functionality. Our first contribution is a simulation modal that is an improvement on the existing tool and is more accurate and better able to module the traffic flows of a network. We do this by providing a queueing implementation that allows us greater control over the service flows of network switches. The improvement also offers additional support for different network disciplines and a starting point for future network discipline implementations. Our second contribution is a simulation tool that can simulate time sensitive networks using a deficit round robin scheduler. We then present a case study using the simulation tool on a tandem network and compare the results against the existing methods of calculating end-to-end network delay. We show that the results generated are more precise and tighter bounded than other methods.

1. Introduction

Recent advancements in ethernet technology have led to a desire to replace traditional embedded systems where one component is connected directly to another component via a wire, with a more advanced cyber physical system, where all components are connected to a network of switches. This transition has provided many benefits but does create an increased risk and complexity of calculating the maximum end-to-end delay of these systems which need to meet industry requirements to prevent failures.

These new cyber physical systems are affected by the flows of other devices on the network, unlike their counterpart in traditional systems. In traditional systems, the network delay can be estimated easily. As pictured in figure 1, the end-to-end delay of a traditional system can be calculated by taking the processing time of component 1, the processing time of component 2, and the time over the wire. The total delay of the flow is $t = t_w+t_{c1}+t_{c2}$, which is constant and easy to calculate.

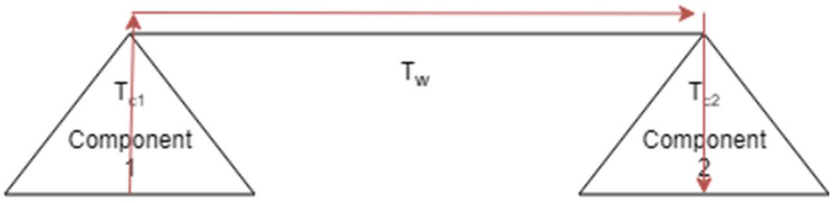

Figure 1: Layout of a traditional embed system with two components connected via a wire.

In cyber physical systems, the connection between the two components is more complicated. As shown in figure 2, the components are connected through a network. The total delay of the red flow is $t = t_{c1}+t_{w1}+t_n+t_{w2}+t_{c2}$, which excluding t_n were part of the original system and are easy to calculate. There are also multiple components on the network that have flows of their own.

These additional flows might interact with and delay messages that we send. This makes calculating t_n more difficult as it depends on what the state of the network is at any given time. Determining an accurate bound for T_n is the focus of many researchers in this area. One solution is to use a simulation tool to attempt to determine the maximum end-to-end delay of any given flow in the network, given the characteristics of the network.

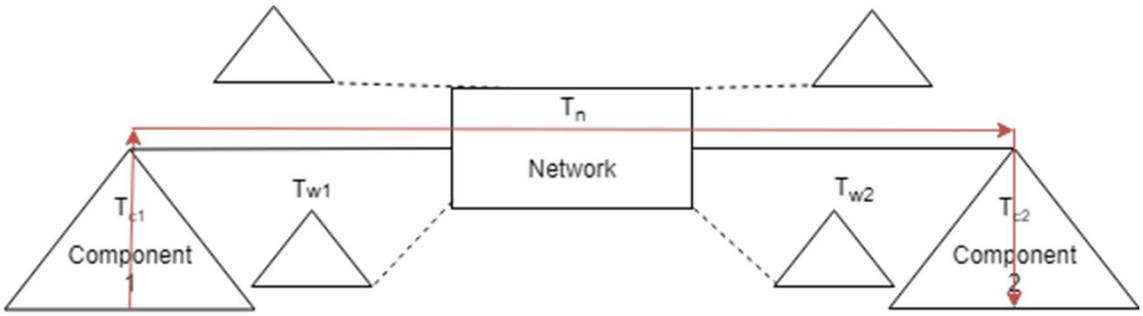

Figure 2: Layout of a cyber physical system with the flow between two components connected highlighted.

Our contributions focus around enhancing the network switches modals to increase the accuracy of simulating T_n in our simulation tool. A switch influences every packet that travels through it. Figure 3 shows the total delay a packet encounters in a switch. The total time spent in a switch by any packet is $t_s=t_a+t_{qa}+t_i+t_{qd}+t_d$, where t_a and t_d are the times required to process the arrival and departure of the packet from the wire. These times depend on packet size. Next, t_i is an inherent delay all packets traveling through the switch incur, and t_{qa} and t_{qd} are the time a packet spends in the arrival and departure queues. These last two times vary on the current situation of the network, and our contribution focuses on simulating t_{qd} more accurately. We assumed that the packets would arrive and be processed at the same rate, and thus no queueing would occur on arrival. Therefore, t_{qa} would not be impactful on the total time. However, much of the work

presented here can be applied to t_{qa} in the future if this assumption no longer holds.

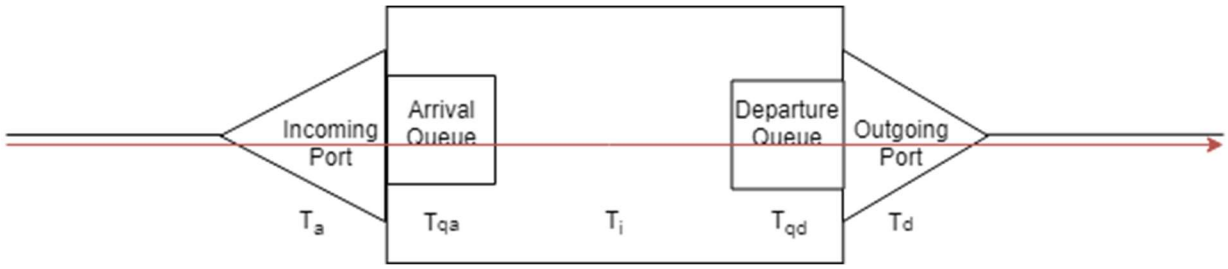

Figure 3: Layout of a switch part of a network for a cyber physical system. A real switch would have multiple incoming and outgoing ports and queues.

The departure queue can operate in a wide range of modes, and the network traffic can have different requirements. The simplest versions of queues operate in a standard first in first out procedure. More complicated networks that require specific flows be handled extremely fast but allow other flows to be handled slowly can make use of some sort of priority scheduling the queue such as deficient round-robin or credit-based shaping.

Our first contribution is an improvement to the quality of the simulation tool that allows us to have more control over the departure queue and the way it behaves. We also add the potential to configure the inherent delay t_i and the departure delay t_d, which was not previously offered. Our second contribution extends on the first and is an enhanced version of the switch that supports deficit round-robin and best effort broadcast that can be used in the simulation of time sensitive networks, which is a network that consists of delay requirements for some flows that are stricter than they are for other flows. For this reason, at least one flow is usually prioritized over others.

2. Methods of calculating end-to-end delay

Network calculus is a way to examine a network mathematically by determining the arrival and service curves of flows through a series of switches in a network. A simple example can be seen in figure 4. The arrival curve on the left consists of two components a burst, the maximum burst that can occur at any one time, and an average arrival rate, the rate at which data arrives over a period. This is an upper bound on the amount of data arriving at a switch. The service curve on the right consists of two components a warm-up delay, a time that a switch requires to warm up and begin servicing packets after the first packet arrives, and a service rate, the rate at which data is serviced once the switch is warmed up. This is a lower bound on the amount of data processed by the switch. The horizontal distance between a service and arrival curve is the amount of time a piece of data is in the switch, assuming first in first out order.

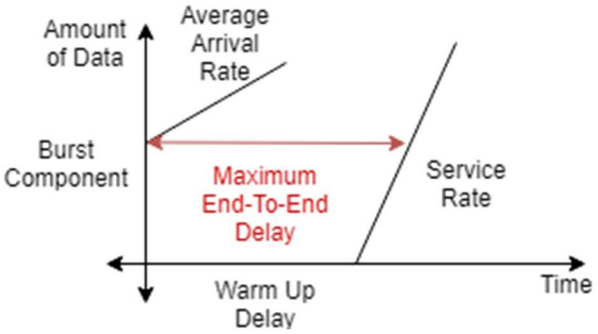

Figure 4: Graph showing service and arrival curve used to calculate maximum end-to-end

[1] and [2] show some of the work that has been done in this field to be able to compute the convolution and deconvolution of much more advanced networks that results in more complicated service and arrival curves. [3] is a tool that easily allows users to perform total flow analysis, partial flow analysis, and PMOO a phenomenon introduced in [4]. Many of the topics covered are outside the scope of this paper. However, the general problem with this approach is

that while it is always mathematically right, it often results in maximum end-to-end delays that are too loosely bounded to be applicable in a real cyber physical system.

Another approach to calculating maximum end-to-end delay is to replicate your network in a testbed as described in [5]. This approach gives more realistic results than the theoretical approach. However, there is no guarantee that the results cover the maximum end-to-end delay. It is also costly to buy enough switches and time consuming to configure a more extensive network. Another limitation is the time taken to run the testbed. It is possible you can run the testbed for a month and still not get the result that causes the network not to meet some performance requirement. The testbed can give us results that closely match a real cyber physical system but does not guarantee an upper bound on the maximum end-to-end delay as well as other limitations on the size of the network.

The two previous methods provide different ways to approach the problem of calculating this maximum end-to-end delay but also come with shortcomings. The third approach attempts to bridge the gap between these two methods and provide results that can both bound maximum end-to-end delay and accurately compare to a real cyber physical system. The idea is to use a simulation tool to accurately be able to put a bound on end-to-end delay while being able to simulate complex networks in a reasonable time. [6] uses a simulation tool to calculate the network performance of wired and wireless networks. Additionally, a simulation tool allows us to analyze the network and determine the exact cause of the slowdown, which can help us create a network that better meets performance requirements.

2.1 OMNeT++ and INET Framework

To perform these simulations, we started using a tool named OMNeT++ [7]. OMNeT++ is a discrete event simulate that allows users to build network topographies, run a simulation kernel, display the simulation, and to generate statistics. It is a general-purpose, open-sourced, discrete event simulator that provides modular components that are coded in C++. Networks can be created using a custom NED file to describe the components in the network, the connection between the components, and some general parameters. OMNeT also relies on INI files to provide information about the flows of the network such as destination, packet size, and the rate at which to send packets. While OMNeT provides a general environment to perform simulations, often, the components it provides are not enough. For that reason, we use the INET Framework [8] for OMNeT to provide more enhanced components, specifically the EtherSwitch and EtherHost components.

The OMNeT simulator works by keeping a global simulation time, which would equate to the time the network was running in the real world. Using these simulation time, OMNeT coordinates events by scheduling messages to the component with events occurring at certain simulation times. Components can create these messages for themselves or other components at any time greater than or equal to the current simulation time. One example of where this would occur is a component sending a packet to another component. A component could decide to send a packet to the next component in line. If both components were inside the same switch, there could be no delay. The component would schedule a message for the destination component at the current simulation time. The OMNeT simulator would notify this component a message has arrived and then deliver the message, which would probably be the packet, and the component could decide how to proceed.

The component we focus our contribution is called EtherSwitch and represents a network switch. The EtherSwitch is made up of three main components an encapsulation component, which is responsible for encapsulation and decapsulation of packets, ethernet interface component, which is responsible for handling incoming and outgoing connection from the switch, and a MacRelayUnit which is responsible for routing the packets to the correct destination. The MacRelayUnit performs lookups against a MacAddressTable that is either given to the switch or collected by the switch during its operations.

2.2 Advanced data structures

One of the main reasons for using simulation as opposed to other methods of calculating maximum end-to-end delay is the ability to simulate a large amount of traffic in a short time. The route pathing component of the OMNeT++ simulator used Dijkstra's algorithm to find the shortest weighted or unweighted path from the source node to the destination node. Being able to optimize this algorithm improves the performance of the simulator significantly. One such method is to use Fibonacci heaps to improve the performance of Dijkstra's algorithm. A second method is to use fuzzy numbers stored in a b-tree that allows us to run an algorithm that considers the possible variation of weights on each of the paths of the network.

2.2.1 Fibonacci heaps

One way to improve the performance of algorithms is to ensure that you use the best data structure for your algorithm. In [9], the authors experiment using Dijkstra's algorithm with a Fibonacci heap in the optimization of networks. For our simulation tool, Dijkstra's algorithm of finding the shortest path through a network is critical to the simulation total run time and using a

Fibonacci heap is the best choice to achieve excellent performance. Fibonacci heaps work with Dijkstra's algorithm by allowing insertion, extracting the minimum, and decreasing a key. These functions are used by Dijkstra's algorithm to keep track of the labeled but not scanned nodes. Dijkstra's algorithm first extracts a node from the heap, then insert or potentially decrease the keys of all neighboring nodes. A Fibonacci heap is a data structure that is a forest of min-heaps linked to each other such that no heaps at the same level have the same height. A Fibonacci heap is an ideal choice for this algorithm firstly because it allows inserting by merely adding a new node to the Fibonacci heap, with a cost O(1). A decrease key is performed by decreasing the desired key; if a min-heap property is violated, we cut the heap and create a new heap and mark the parent. If the parent is already marked, then we perform cascading cuts and marks until we have an unmarked parent and mark it. Despite this, the amortized cost of a decrease key is O(1) due to the potential lost by changing from marked nodes to root heaps in the worst case and the increased potential of marking a node in the ideal case. Finally, an extract min is performed by extracting the min pointer and creating new heaps for the children. Then all heaps of the same height are melded together until no two heaps have the same height. This operation is performed in O(log n) since the rank of the heap is bound to O(log n) due to the decrease key function. This allows us to perform Dijkstra's algorithm by inserting all the nodes once at O(1), deleting all vertices once at O(log n) and for all edges that do not add a new vertex potentially decrease key at O(1). This results in a run time of O(V log V) which allows for the algorithm to run extremely fast and is beneficial for our simulation.

2.2.2 B-trees

One of the issues we encountered while working with the OMNeT simulator was being able to route flows through the network properly. Figure 5 shows a network that highlights such a

problem. Using Dijkstra's algorithm, we would find one path through the network, and both flows would follow the same path as opposed to the desired one through the top switch and one through the bottom switch.

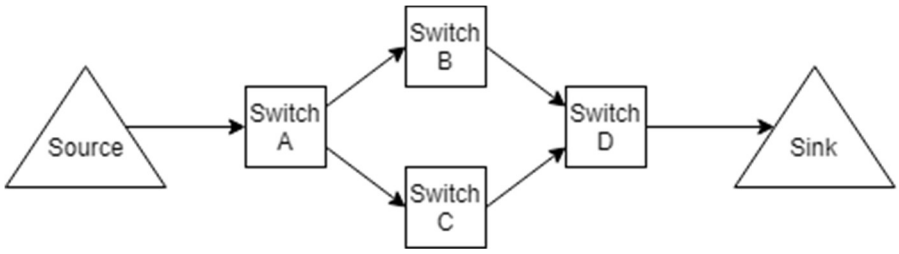

Figure 5: A network diagram showing a potential problematic flow for Dijkstra's algorithm. With two different flows from source to sink.

One solution to this problem is to use fuzzy numbers, a concept examined in [10], to assign imprecise edge weights to a graph and calculate the shortest path. This could be done to ensure that flows after the initial flow choose the best path available by providing the edges with a fuzzy weight to them for the existing flows. This could be done using a linked list to hold potential paths from source to sink before they are flagged. However, it is often better done using a b-tree since our networks tend to be large enough and complicated enough to warrant it. B-tree's are a tree structure with specific requirements that there is a minimum of n elements on each node, and the height of each leaf is the same. With pre-emptive splitting and merging, we can assure worst-case insertion, deletion, and lookup in O(log n). This allows the algorithm to be fast enough to warrant use in more complicated networks where a traditional Dijkstra algorithm approach would not produce the best results.

3. Improvements to simulation

In this section, we demonstrate the improvements made to the simulation and how they result in a more accurate and complete simulation tool. The first section covers the improvements made to a simple first in first out switch. The second section covers the improvements made to allow for the simulation of decisive round-robin and first in first out as part of a time sensitive network.

3.1 Basic queue enhancements

The first improvement made to the existing simulation tool was focused on enhancing the queueing infrastructure and allow the user to have more control over the service curves of the system. Our focus for this enhancement was to more accurately reflect the departure time ($t_{d)}$ and the queueing delay on outbound packets (t_{qd}) as well as introduce an inherent delay to the switch (t_i). We do not attempt to change the arrival curves, and we assume that every packet in the system encounters a constant t_i delay. We designed a system using a first in first out procedure with enqueue and dequeue methods as the only interaction between the unit and the underlying queue. This way, in the future, we would be able to change the underlying queue if it could operate with an enqueue and a dequeue message. Our implementation modifies EtherSwitch by placing a new component named queueUnit between the relayUnit and each of the output interfaces of a switch. Figure 6 shows the OMNeT EtherSwitch with and without our added queueUnit. The switch works by having packets arrive at the eth interface which would represent a port. The packet then travels through the relay unit which routes the packet. Then the encapsulation units to the bottom layer before reversing and heading back towards the eth interface to depart. In our implementation the, packet will travel to the queue unit associated with

the outgoing port the packet is traveling towards before departing. As shown, we only place a queueUnit between packets coming from the relayUnit and the outgoing port, not the incoming port.

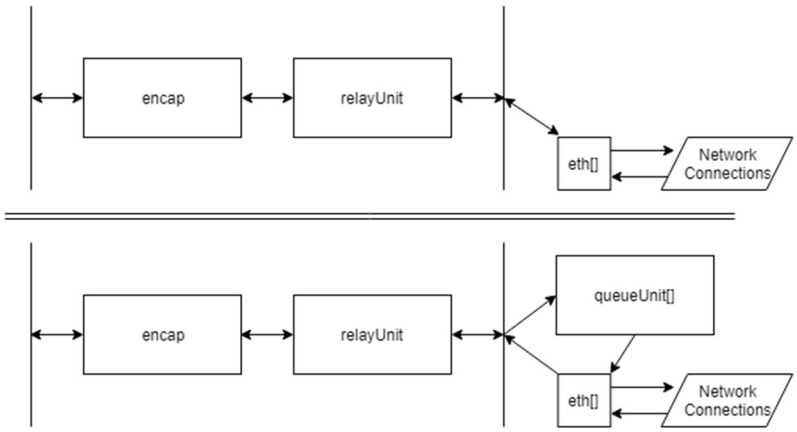

Figure 6: Diagram showing the submodules and connections of an EtherSwitch. On the top the existing implementation and on the bottom our enhanced EtherSwitch implementation.

We implemented the queueUnit to respond to two different signals, a message's arrival and a self-message performed by the queueUnit. We allow the queueUnit to operate in two distinct modes, a mode for when a message is being sent and a mode for when no message is being sent. This is controlled by a boolean variable. We require two different parameters, the delayTime, which controls the inherent delay all packets experience and the outputRate, which controls the speed at which packets are processed, leaving the switch. When a packet arrives at the queueUnit we check to see if a packet is currently being sent, if one is, we enqueue the packet. If a packet is not being sent, we switch the unit to be in the sending mode, send the packet with a delay, by providing OMNeT the simulation time to delay sending the packet, equal to delayTime, and schedule an alert at the current simulation time plus the length of the packet divided by the outputRate, this represents when the packet should be sent and when we can start processing a new packet. Note that this time doesn't include an inherent delay component, this delay is factored in later, but the delay can be counterfeited if the switch is already warmed up and

sending packets as a real switch would. The other process occurs when we receive a self-message. When we receive a self-message, we know that a new packet can be sent. The unit first checks if there is a message in the queue if there are no messages, it switches to standby mode. If there is a message in the queue the unit takes the message out of the queue, it sends the packet with a delay of delayTime and schedules a new self-message at the current time plus the length of the packet divided by the outputRate. This is the same procedure as when a new message arrives and is sent right away. The system of self-messages ensures that if any packets are in the queue, we keep sending packets until the queue is empty. We also ensure that the ordering of the queue can change, but the unit behaves in the same way so that any queue can be used. The next section covers enhancements made to this structure, and section 4 shows the results of these changes.

3.2 Time sensitive network enhancements

Our next contribution was to create a switch that would be able to simulate time sensitive networks, specifically a switch that operated in deficit round-robin for some high priority flows and first in first out for a low priority flow. We used the queueUnit implementation explained in the last section as a starting point with some minor changes. First, a priority queue was used instead of a regular queue, if the packet was from the deficit round-robin the packet was given high priority and if the packet was from the regular first in first out flow is was given low priority. The unit operated in the same way as the original unit with one mode for sending and one mode for waiting, along with having two events cause operations, the arrival of new packets and self-messages by the unit. Additionally, the new unit has a deficit counter and a deficit queue, which is just a normal first in first out queue, for each incoming flow, which is

determined by source address, for the deficit round-robin flows. Figure 7 shows the layout of these new features and the flows through the switch.

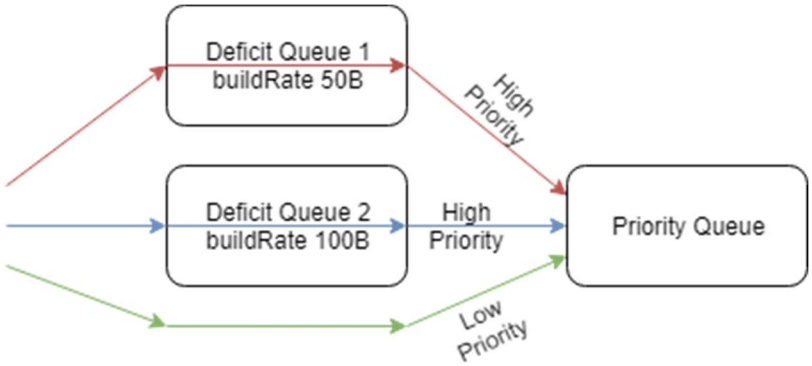

Figure 7: Example of our deficit round-robin queue implementation. The top flow is serviced approximately have as fast as the middle flow based on the buildRates. The bottom flow is only serviced when there is no other traffic.

Each of these flows also requires a buildRate which controls how fast the flow builds up the deficit counter. When a packet arrives if the queueUnit is in a waiting state, then we switch it to a sending state and send whatever packet has arrived using the same method as previously described. If a packet arrives when the queueUnit is in a sending state, we enqueue the packet in deficit queue if it is used in the deficit round-robin or the regular priority queue with low priority if it is from the first in first out flow. When we receive a self-message, we first check to see if there are any high priority messages in the queue, if there are, we send the message using the method previously described and finish. If that is not the case, we perform a round or multiple rounds of the deficit round-robin. We perform a round by first going through the queues; if any are empty, we set the deficit counter to zero. If any of the queues have packets at the front that are smaller than their associated deficit counter, we remove it from the queue and add it to the priority queue with high priority. We also subtract the packet size from the deficit counter and then redo this check on the queue. Once we get through all deficit queues for this step if we moved any packets from the deficit queues to the priority queue we are done, otherwise we

increment all the deficit counters that have items in their queue by the build rate for that specific queue and perform another round until eventually at least one packet is moved. We then send the highest priority packet in a queue using the method described earlier. This is a packet moved from the deficit queues if any were moved or a packet from the first in first out queue only if no high priority packets were in the system. If there are no low priority packets in the system either, then we switch the mode to stand by.

4. Tandem network analysis

In this section, we show the results of our simulation tool using the improved first in first out method. We use a tandem network that consists of a line of switches and one flow through all the switches. Additionally, each switch has two flows interfering with the main flow. This can be seen in figure 8 with the top and bottom flows interfering with the main flow. The pattern can be extended indefinitely.

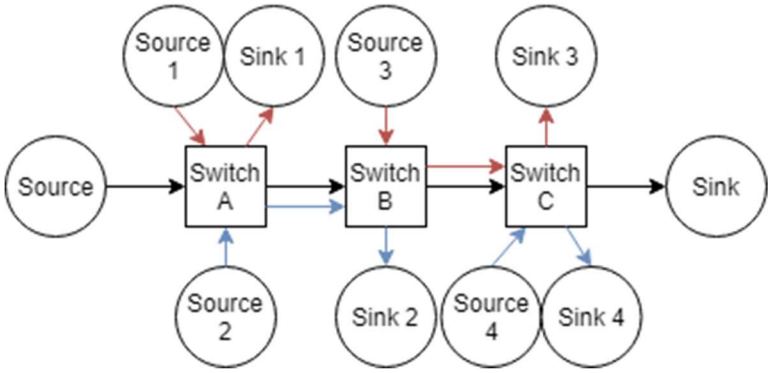

Figure 8: Diagram showing the layout of a 3-switch tandem network. Main flow in the middle with conflicting top and bottom flows.

We compare our result to those gathered in [11] as a way to prove that our simulation can model network traffic. Our simulation uses two source components instead of one, one for the burstiness and one for the average rate, however, since we are not prioritizing the incoming queues the flows merge into one in the first switch and should not affect results. Additionally, we set each packet size to be the maximum 1500B. The rate is 0.67Mbps with a burstiness of 1Mbps. The switches have a latency of 0.1Mbps and service rate of 10Mbps.

4.1 Simulation results

We performed our simulation on a 4-switch tandem network with 6 flows. We performed this simulation using both our enhanced first in first out version as well as the default OMNeT implementation. Table 1 shows the resultant maximum end-to-end delays of all flows through the network.

	Default OMNeT	Enhanced OMNeT
Main Flow	1.45828s	1.75562s
Conflicting Flow 1	0.24034s	0.45054s
Conflicting Flow 2	0.70568s	0.86171s
Conflicting Flow 3	0.72024s	0.89413s
Conflicting Flow 4	0.72080s	0.91532s
Conflicting Flow 5	0.47179s	0.49132s

Table 1: Results of a 4-switch tandem network with 6 total flows. The main flow is the straight flow through all switches whereas conflicting flows are through one or two switches.

As you can see, our enhancements result in a significantly longer delay than the default OMNeT implementation. We believe that this delay is, however, more accurate under the worst-case performance of a network. This is due to the added inherent delay and proper queueing of packets that were not provided by the original OMNeT implementation but more accurately model a real-world switch. We also calculated the maximum end-to-end delay of the main flow for a 1 and 20 switch network. Table 2 shows the results of the experiment which are consistent with the results found in [11].

	1 Switch	4 Switch	20 Switch
Maximum End-to-End Delay	0.92385s	1.75562s	24.52364s

Table 2: Results of maximum end-to-end delay of the main flow through 1, 4, and 20 switch tandem networks.

4.2 Future work

Some ways to enhance this experiment, as well as the OMNeT implementation, are discussed here. Our results can be improved by performing tests of more networks other than the tandem network. We could also develop a way to perform tests of a time sensitive network and be able to compare the results to something. We create a system to analyze networks by combining the network calculus calculators, network simulators, and network testbeds together for a complete analysis of the results of the system. Also, developments can be made to the implemented queue unit to support more advanced network policies for time sensitive networks such as creating a credit-based implementation in OMNeT. We designed our implementations in a way that would hopefully allow for these improvements to be made efficiently in the future and hope that many new networking paradigms are created.

5. Conclusion

In conclusion, our contributions are made up of two separate implementations that serve to enhance the functionality of the OMNeT++ simulator with the INET Framework. The first implementation allows more control over the service curve by allowing the users to set parameters to control the output rate and the inherent delay of the queue. It also provides an easy way to change and implement more queue types in the future. Our second contribution was to create an implementation based on the previous work to be able to simulate time sensitive networks. We implemented an algorithm that would perform deficit round-robin at a high priority and first in first out at a low priority. The work done can be expanded to allow even more complicated time sensitive networks. Overall, we demonstrated that our contributions were able to increase the accuracy of the simulation tool and allow for the simulation of more diverse networks. We hope this is useful as more cyber physical systems are introduced and simulating the maximum end-to-end delay is even more critical.

Bibliography

1. J.-Y. L. Boudec and P. Thiran. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Springer, 2001.
2. Y. Jiang. Network calculus and queueing theory: Two sides of one coin. In ICST ValueTools, 2009.
3. S. Bondorf and J. B. Schmitt, "The DiscoDNC v2: A comprehensive tool for deterministic network calculus," in Proc. 8th Int. Conf. Perform. Eval. Methodologies Tools. Västerås, Sweden: ICST, 2014, pp. 44–49.
4. J. B. Schmitt, F. A. Zdarsky, and I. Martinovic. Improving Performance Bounds in Feed-Forward Networks by Paying Multiplexing Only Once. In GI/ITG MMB, 2008.
5. Lerch, Sean. IEC 61850 - The Future of Substation Automation. March, 2015, https://www.lehigh.edu/inesei/public/www-data/images/posterpdfs/14-15_Sean%20Lerch.pdf. [accessed on August 2, 2019]
6. Dhobale, I. "Wired and Wireless Computer Network Performance Evaluation Using OMNeT++ Simulation Environment." (2014).
7. OMNeT++ Discrete Event Simulator. http://www.omnetpp.org [accessed on August 2, 2019]
8. INET Framework. https://inet.omnetpp.org [accessed on August 2, 2019]
9. M. FREDMAN AND R. TARJAN, Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithm, Journal of ACM, 34 (1987), pp. 596-615.

10. Nayeem, Sk & Pal, Madhumangal. (2005). Shortest Path Problem on a Network with Imprecise Edge Weight. Fuzzy Optimization and Decision Making. 4. 293-312. 10.1007/s10700-005-3665-2.
11. Bouillard, Anne & Jouhet, Laurent & Thierry, Eric. (2010). Tight Performance Bounds in the Worst-Case Analysis of Feed-Forward Networks. Proceedings - IEEE INFOCOM. 1 - 9. 10.1109/INFCOM.2010.5461912.