2019-11

Congestion Control in Software-Defined Networks: A Simulation Study

Gholizadeh, Reza

UNIVERSITY OF CALGARY

Congestion Control in Software-Defined Networks: A Simulation Study

by

Reza Gholizadeh

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA
NOVEMBER, 2019

Abstract

Congestion is an underlying reason for performance degradation in computer networks. Current TCP congestion control has no information about the network. Hence, it increases the sending window to overflow the bottleneck link buffer, and backs off when packet drops are detected. Software-Defined Networking (*SDN*) is a new paradigm, which provides information about the network. In this thesis, we propose a novel centralized congestion control scheme for *SDN*. Our solution exploits the information provided by the *SDN* controller to prevent formation of persistent queues in bottleneck links. Also, we introduce an *SDN Simulation Tool* developed in Java, which facilitates simulation experiments. We used our tool to evaluate the proposed solution. The preliminary results shows the potential scalability and flexibility of the protocol.

Acknowledgements

I wish to thank my supervisor, Dr. Carey Williamson, for his supporting and guiding me throughout this program. He is a great supervisor and teacher. But most importantly, he is an inspirational human being who never stops believing in people and what they are capable of doing. I truly enjoyed being his student, and from now on I will enjoy being his friend.

Table of Contents

List of Figures and Illustrations

List of Tables

List of Acronyms

Symbol or abbreviation	Definition
API	Application Programming Interface
AvgQL	Average Queue Length
BDP	Bandwidth Delay Product
DES	Discrete-Event Simulator
FCT	Flow Completion Time
FSD	Flow Setup Delay
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
LAN	Local Area Network
MaxQL	Maximum Queue Length
RVG	Random Variate Generator
RTO	Average Queue Length
RTT	Round-Trip Time
SDN	Software-Defined Network
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

1.1 Historical Context and Motivation

The history of computer networks dates back to the early 1970s, when proprietary local computer networks were deployed in different institutions around the world [2, 30]. As the number of these local networks grew, the idea of inter-connecting them resulted in the creation of the first version of the *Transmission Control Protocol* (TCP) [12]. This version of TCP had both end-to-end reliable data delivery and forwarding functions. Later, the forwarding functionality was separated from TCP, and the first layered architecture of the Internet we know today was formed. This layered architecture has enabled the introduction of new protocols, and the modification of existing ones without the need to change the whole architecture of the network. Hence, the layered architecture of the network has not changed since.

The early versions of TCP did not consider the state of the network links and buffers for adjusting the sending window. On the other hand, the number of networks connecting to the Internet grew quickly. As a result, a major collapse of the Internet happened in the late 1980s due to congestion. The TCP congestion control mechanism was designed as a result of this incident.

By that time, it was already shown that the optimal sending rate for the sender can be determined by the physical properties of the bottleneck link [32]. However, TCP was defined in the transport layer, and did not have any information about the network layer (i.e., the physical link properties and buffer state). Hence, the first TCP congestion control mechanism [25] was designed based on this fact. The basic idea of the algorithm was a control loop, which increases the sending rate until packet losses happen then reduces the sending rate and starts to increase it again. This algorithm became a standard feature of TCP [5]. However, as the Internet kept growing, the performance issues of TCP congestion control became a popular topic in the research community.

The prior research to resolve the performance issues of TCP congestion control can be divided into the following categories:

- Delay-based Approaches: These solutions use variations in *Round-Trip Time* (RTT) to detect congestion sooner than loss-based solutions [9, 23, 52, 55].

- Rate-Based Approaches: These approaches reduce the queuing delay generated due to the burstiness of TCP traffic, using pacing methods [15, 19, 56]. This approach has solid foundations in queuing theory, since reducing the variability of the packet arrival process reduces the average queuing delay.

- Network-specific Solutions: These solutions modify the congestion control mechanism to make it suitable for a specific type of network, such as wireless [38, 51, 54], high-speed networks [10, 22, 35, 58], and data-center networks [4, 49, 57].

Most of these solutions try to make up for the fact that TCP's congestion control mechanism operates with no explicit information about the network.

1.2 Software-Defined Networks

Software-Defined Networking (SDN) is a rather new paradigm, which redefines the network layer by separating control and forwarding planes from each other [39]. In a legacy network layer, both control and forwarding functionalities are implemented in the router. The forwarding plane is responsible for forwarding network packets according to the rules determined by the control plane. The control plane is defined by introducing a new entity called the SDN Controller. The controller is in charge of controlling the forwarding plane by setting rules.

An important feature of the controller is having a central view of the network. It can determine all the physical properties of network links and switches. Also, it can actively update its knowledge of the traffic traversing the network by monitoring switches.

Since the introduction of SDN, the research community has tried to exploit its features to improve TCP congestion control. Most of the proposed solutions try to improve the performance with minimal change in the TCP congestion control algorithm. They use the controller to detect congestion, and notify the TCP agents to reduce their sending rates.

TCP congestion control was designed at a time that there was no entity in the network possessing information, and the transport agents were completely agnostic about the network layer information. To exploit the full potential of SDN, the congestion control mechanism should be re-designed in a centralized manner.

1.3 Research Objectives

The first research objective in this thesis is to design a centralized congestion control mechanism for SDN. To this end, we define new congestion control parameters and provide analytic models for them. In our protocol, the controller is in charge of determining the value of congestion control parameters. Also, we enable the controller to communicate with transport agents throughout the network, to provide them with congestion control parameters. The transport agents are responsible for operating based on the congestion control parameters provided by the controller.

The proposed protocol tries to eliminate two main reasons for formation of queues in network buffers. The first reason is the mismatch between the transport agent sending rate and the bottleneck rate. The second one is the arrival of packets from multiple inbound links at the same time, which share the same outbound link.

The second research objective is to evaluate the performance of the proposed solution. To this end, we need a fully controlled environment to be able to perform the evaluation according to our assumptions. Hence, we use simulation experiments as our evaluation methodology.

The third research objective of this thesis is to develop an *SDN* simulator that models transport and network layers. Also, we aim to develop an integrative framework for our simulator, which facilitates multi-factor simulation experiments. Our framework provides a traffic and topology generator, and provides the experiment results in spreadsheet files with tables and graph based on pre-defined metrics.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents the background knowledge and related work for the thesis. It provides an overview of the network protocol stack. Then it introduces TCP along with its reliable data transfer and congestion control mechanisms, and discusses some performance issues. Also, it overviews SDN architecture, and discusses some of the challenges and opportunities. Moreover, it provides a description of related work.

Chapter 3 presents the SDN Simulation Tool developed and used for evaluation of the proposed solution. Specifically, it discusses the architecture of the simulator and verification tests.

In Chapter 4, the centralized congestion control scheme designed for SDN is introduced in concept and examples. The analytical and simulation models along with the verification tests are described in detail.

Chapter 5 presents the simulation evaluation of the proposed solution. It describes the experimental design, including the performance metrics, and provides the simulation experiments along with their results and analysis.

Finally, Chapter 6 provides the conclusion of the thesis and suggests possible future directions for further research.

Chapter 2

Background and Related Work

This chapter presents the background knowledge and related work for the thesis. Section 2.1 explains the network protocol stack and how the Internet works. Section 2.2 describes TCP and its features. Section 2.3 reviews the SDN architecture along with its benefits and challenges. Section 2.4 introduces work done by other researchers related to the solution presented in the thesis. Section 2.5 provides a summary of research contributions of the proposed solution in comparison with the related work. Finally, Section 2.5 summarizes the chapter.

2.1 Network Protocol Stack

Computer networks are designed using a modular layered architecture to facilitate updating protocols and system components without changing other parts. Each protocol is designed specifically for a single layer to provide services to an upper layer. However, there are some potential drawbacks to this design such as service duplication and the need for information from other layers. Figure 2.1 illustrates the five layers of the Internet protocol stack [34].

All network applications[1] and their application-layer protocols are defined in the application layer. Application-layer protocols enable an application in one end system to exchange

[1] An application that uses networking features.

packets of information with the application in another one. The packet of information in the application layer is called a **message**. *HyperText Transfer Protocol* (HTTP) [8], *File Transfer Protocol* (FTP) [48], and *Simple Mail Transfer Protocol* (SMTP) [45] are examples of application-layer protocols providing services to the end system applications.

The transport layer provides the application layer with a service for transporting messages between end systems. The packet in the transport layer is called a **segment**. *Transmission Control Protocol* (TCP) [47] and *User Datagram Protocol* (UDP) [44] are the dominant transport layer protocols in current network systems. TCP offers a connection-oriented service with guaranteed end-to-end delivery, while UDP is connectionless with no reliability. This thesis focuses on TCP performance issues.

The network layer takes a segment from the transport-layer protocol in the source host and transforms it into a **datagram**. Then, it transfers the datagram between network nodes and delivers it to the transport-layer protocol in the receiver end host. On the Internet, the *Internet Protocol* (IP) [46] is the single network-layer protocol, which works alongside multiple routing protocols. All hosts and routers implement IP and the routing protocols to be able to communicate with each other.

The link layer obtains a datagram from the network layer in the sender node, transfers it to the next node, and gives it to the network layer of that node. Link-layer protocols may provide different delivery services in terms of reliability. Ethernet and WiFi are two well-known link-layer protocols. The packet of information in the link layer is called a **frame**.

The physical layer is responsible for delivering individual bits of a link-layer frame via the transmission medium. A physical-layer protocol determines the way that bits are transferred across the transmission medium. Hence, the protocols are defined based on the type of the actual transmission medium.

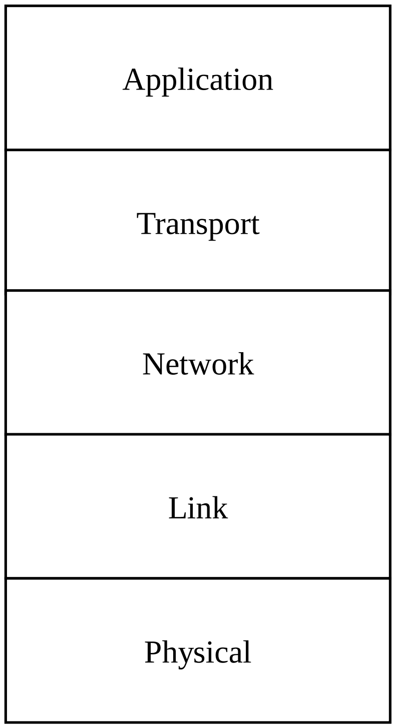


Figure 2.1: Five-layer Internet protocol stack.

2.2 TCP

TCP is an end-to-end transport-layer protocol, responsible for transferring information from a source host to the destination. This protocol was initially introduced in 1973 as a part of the US *Advanced Research Project Agency* (ARPA) network system. Alongside the expansion of the Internet, new mechanisms and functionalities have been added to TCP. One recent version of TCP is called NewReno. Reliable data transfer and congestion control are two main concepts that are implemented in it.

2.2.1 Reliable Data Transfer

TCP operates on top of an unreliable network layer. In fact, all the layers beneath TCP can be modeled as an unreliable point-to-point channel. TCP provides end-to-end reliability on top of this channel by implementing multiple mechanisms.

Sequence Number and Acknowledgment

To make sure that every segment sent is received by the end host, TCP implements an acknowledgment mechanism. When a data segment is delivered to the receiver TCP agent, it sends back an acknowledgment segment to the sender TCP agent. In addition, to guarantee that all the segments are delivered in the correct order, the sender assigns a sequence number to each data segment it sends. Also, this mechanism helps TCP avoid duplicate data delivery at the receiver. The sequence number and acknowledgment mechanism enables TCP to determine whether a data segment has been delivered to the receiver or not.

Timer and Retransmission

TCP uses a timer and retransmission mechanism for identification and recovery of lost segments. The sender TCP agent sets a timer when it sends a segment to the network. If the timer expires and the acknowledgment for the data segment has not been received, the sender retransmits the data segment. The value of the timer is determined by the estimation of the round trip time for the end-to-end path.

Flow Control

TCP has a pipelined sending mechanism to improve utilization of the network links. There is a sending window parameter at the sender TCP agent that determines the number of in-flight segments allowed for a TCP connection. However, if there is no constraint on the value of the sending window, the receiver's buffer may overflow. To avoid this issue, TCP uses a flow control mechanism that sets the value of the sending window according to the value advertised by the receiver.

2.2.2 Congestion Control

A congestion control algorithm was added to TCP in 1988 [25], as a result of a major collapse of the Internet. This mechanism ensures that the network does not stay in a congested state by introducing another constraint, called the congestion window, on the sending window. The algorithm uses a finite state machine that uses three states for the TCP sender agent.

The initial state is called *Slow Start*. This state increases the value of the congestion window every time a new data segment is acknowledged, until it reaches a specific threshold. After reaching the threshold, the algorithm transitions to the *Congestion Avoidance* state.

The *Congestion Avoidance* state keeps increasing the congestion window more slowly, until it receives a loss indication, such as three duplicate acknowledgments from the receiver. The algorithm considers this segment drop as a sign of congestion. Hence, it reduces the threshold for the congestion window and updates its value accordingly. Then, it retransmits the missing data segment, and transitions to the *Fast Recovery* state.

The *Fast Recovery* state keeps sending missing segments while increasing the congestion window. As soon as the sender receives a new acknowledgment, it goes back to *Congestion Avoidance* until the next segment loss happens.

This algorithm is called a *loss-based* congestion control mechanism. TCP is a transport-layer protocol, which means it does not possess any information about the network infrastructure proprieties, such as bottleneck link bandwidth. Hence, to determine the proper sending rate, the algorithm keeps increasing the sending window until the bottleneck buffer overflows and a segment is dropped. This approach is the underlying cause of many performance issues of TCP [21, 41, 42, 43, 53].

2.3 SDN

SDN introduces a new architecture for the network layer. Figure 2.2 illustrates the logical view of the SDN architecture [34]. Decoupling the data and control planes is the main characteristic of SDN.

This architecture enables SDN to introduce flow-based forwarding, which forwards the packets based on any combination of transport-layer, network-layer, and link-layer header fields. It also re-defines the network functions (e.g., routing) as software that operates separately from the forwarding devices.

2.3.1 Control Plane

The control plane's responsibility is to control the packet forwarding by configuration and management of the forwarding plane devices. An SDN controller, which can be considered as a network operating system, and SDN network-control applications are two main components of this layer.

An SDN controller provides an interface, which is called the *southbound interface*, for communication with data plane devices. OpenFlow is the dominant protocol to standardize the southbound interface. The SDN controller uses the OpenFlow [39] protocol to configure the SDN-enabled switches, and get network-state information from them. The controller uses the information received from the data plane devices to maintain a central view of the

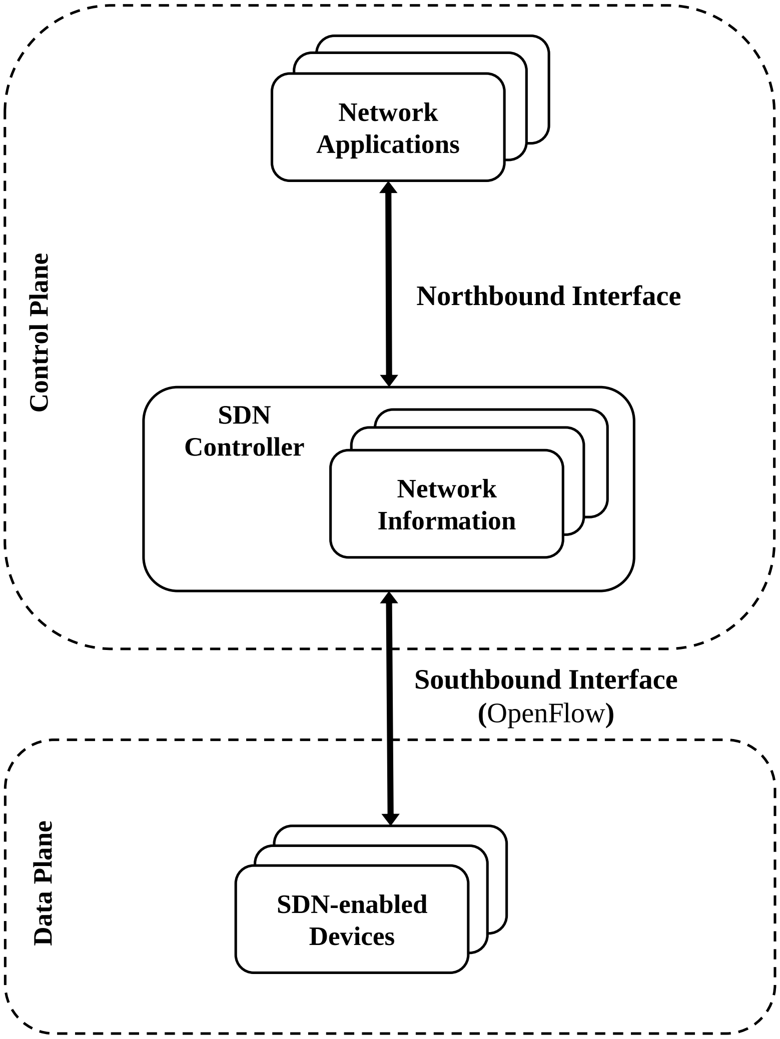


Figure 2.2: Logical view of SDN architecture.

network and the traffic. Also, it provides another interface, called the *northbound interface*, to pass the network-state information to different network-control applications. There is no single standard northbound API defined for controllers. However, most of the existing SDN controllers implement a RESTful API as the northbound interface [7, 16, 40, 50].

A network-state application is a network functionality, such as routing, implemented as software. These applications obtain network information, and apply their functionality to the data plane via the northbound interface. Common network functionalities, such as routing, can be implemented as actual controller modules with direct access to the network information.

2.3.2 Data Plane

The data plane contains SDN-enabled network devices operating with the OpenFlow protocol. An SDN-enabled switch has a flow table that is maintained by the control plane.

The flow table is defined by the OpenFlow protocol, and operates based on *match-action* logic [34]. Each entry in the table is called an OpenFlow rule. The rules are defined based on any combination of the packet header information. For instance, "$IP\ Src = 10.1.*.*$" will match any packet that comes from a source with an IP address starting with "10.1.".

When a packet arrives to the switch and one of the flow table entries matches it, the switch performs the action corresponding to the rule. Some of the actions that the switch can perform are *forwarding*, *dropping*, and *modify-field*.

2.3.3 Challenges and Opportunities

The separation of the control plane from the data plane in SDN adds programmability to the network, which creates a vast opportunity for re-imagining the network management and operation. The SDN control plane offers a multi-layer central view of the network and the traffic, which can be exploited to improve the performance of network protocols such as TCP.

Scalability is the most important challenge for SDN [18, 27]. In SDN, the controller maintains active communication with all the switches in the data plane. Hence, increasing the size of the network adds more complexity to this task.

Also, there are concerns about the integration of SDN into the Internet [27]. Replacing all legacy forwarding devices with SDN-enabled hardware is a challenging requirement. Another challenge is designing a fault-tolerant and efficient control plane for SDN.

2.4 Related Work

The design of an ideal transport-layer protocol has been one of the most popular research areas since the advent of the Internet. There are many research works concentrating on performance improvement of congestion control mechanisms for TCP. Figure 2.3 illustrates the classification for congestion control solutions.

There are many solutions that try to improve the congestion control mechanism of TCP, regardless of the network layer architecture. Hence, we only mention the most influential solutions in this category. These works are classified based on the congestion control approach. Also, with the advent of SDN, various research works have tried to exploit its potential benefits to improve the congestion control mechanism of TCP.

2.4.1 Generic Network Solutions

The first version of TCP [12] was introduced in 1974. The Internet was different back then. It was essentially a small network of small networks. There was no congestion control mechanism in TCP's design. As a result of a major congestion collapse in the Internet in 1986, Jacobson [25] introduced the first congestion control mechanism for TCP, called *Tahoe*.

After the basic problem was solved by the Tahoe congestion control mechanism, subsequent research focused on the performance and efficiency of TCP.

Jacobson [26] modified Tahoe to Reno in 1990. The main differences between Tahoe

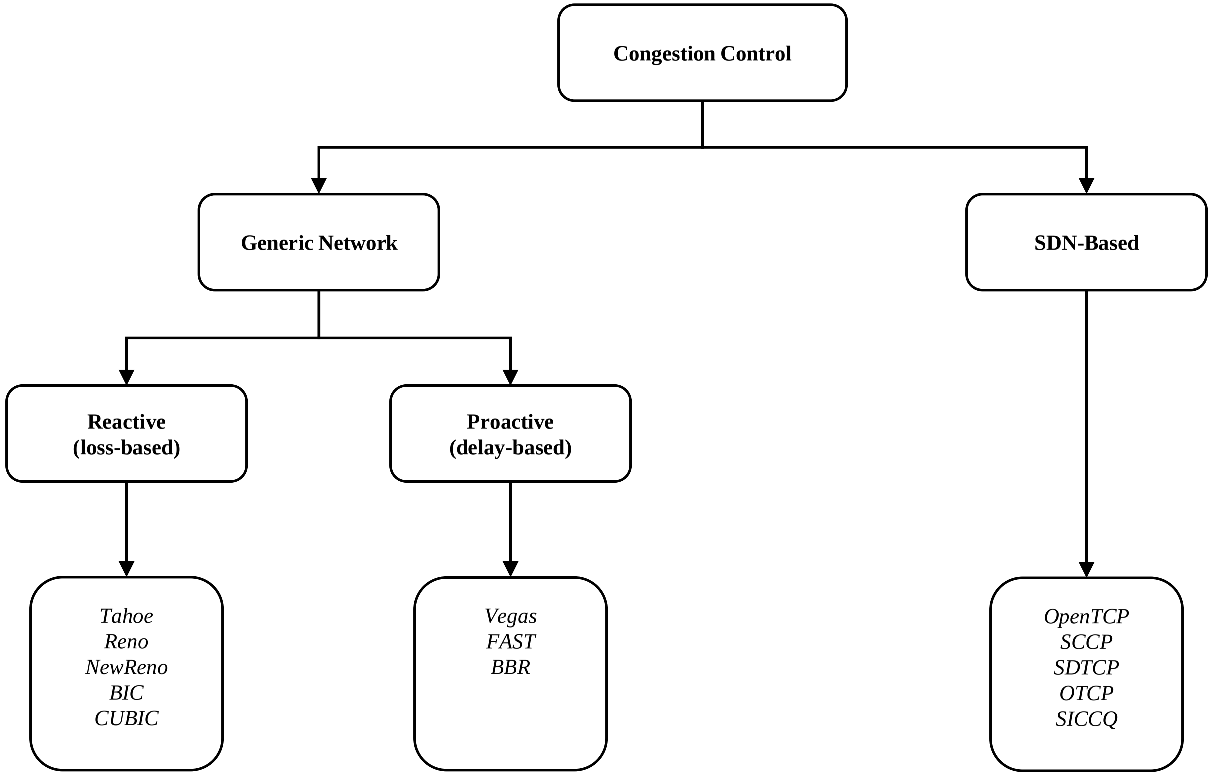


Figure 2.3: Categories of congestion control solutions.

and Reno are as follows. Upon receiving three duplicate *ACKs*, Tahoe retransmits the lost segment. Then, it sets the slow start threshold to half of the congestion window, and reduces the congestion window to 1 MSS before transitioning to the *Slow Start* state [34]. On the other hand, Reno resends the lost segment, updates the congestion window to half, and sets the slow start threshold to the congestion window. Then, it transitions to the *Fast Recovery* state. In this way, Reno needs less time to reestablish a proper congestion window size.

Floyd [17] introduced NewReno to improve upon the *Fast Recovery* mechanism of Reno. NewReno keeps the sending window full during retransmission in the *Fast Recovery* state by sending additional new segments using an expanded congestion window. This results in better link utilization after segment losses happen. NewReno is the most recent standard TCP variant [17].

The solutions in this category try to improve NewReno performance for specific types of networks, such as high-speed networks, data centers, and wireless networks. The main challenge for these solutions is estimating the state of the network links and buffers to either prevent or alleviate congestion in the network.

Reactive (Loss-Based)

These solutions are designed based on NewReno, and consider segment loss as an indication of congestion in the network. Each solution tries to introduce a congestion control scheme suitable for a specific type of network. In particular, there are several research works that improve bandwidth utilization in high speed networks with high latency.

Binary Increase Congestion Control (BIC) [58] improves the bandwidth utilization in wide area networks with high link bandwidths. It introduces an adaptive scheme that varies the congestion window growth rate depending on network conditions. Based on different conditions, it determines the congestion window size using either binary search, additive increase, or slow start approaches.

CUBIC [22] is a derivative of BIC, which determines the window size with a cubic function of time since the last congestion event. CUBIC is the default TCP implementation in Windows 10 and Linux kernels.

Although these solutions successfully improve TCP performance in specific types of networks, all of them use a reactive congestion control approach. The lack of information about the network properties (e.g., bottleneck bandwidth) forces these solutions to rely upon estimation mechanisms. These mechanisms gradually saturate the bottleneck links and buffer until a segment drop happens. The segment drop is considered as an indication of congestion in the network, and used to update the estimation of network properties. These algorithms always reach an operating point in which the bottleneck buffer fills to overflowing, which is sub-optimal according to Kleinrock [32].

Proactive (Delay-Based)

These solutions implement a proactive approach to network congestion. They utilize the *Round Trip Time* (RTT) values of the segments to detect network congestion before losses happen.

TCP Vegas [9] calculates the maximum connection throughput based on the RTT samples of the segments. Then it determines the difference between expected and actual throughputs, and adjusts this value by specifying the congestion window size.

FAST TCP [28] is a delay-based solution for high-speed long-latency networks. It maintains a queuing delay estimate for the segments, and utilizes it to prevent congestion in the network. Both Vegas and FAST try to maintain a constant number of packets queued in the buffers of network links.

BBR [11] is Google's proactive congestion control mechanism. It utilizes segment RTT samples to estimate the bottleneck bandwidth (BB) and round-trip (R) propagation time. Unlike Vegas and FAST, BBR tries to prevent accumulation of segments in the network buffers. The authors claim that BBR converges with high probability to Kleinrock's optimal operating point [11].

These solutions have the advantage of preventing congestion, compared to loss-based approaches. However, their performance is highly dependent on the accuracy of their estimation of the network properties. Also, the fact that these solutions detect congestion and reduce their sending window size earlier than loss-based approaches could be detrimental when both types of solutions exist in the same network.

2.4.2 SDN-Based Solutions

SDN provides a central view of the network information, which can be used to enhance TCP's congestion control mechanism. Data center networks can exploit SDN features for better operation. Hence, most of the research works in this category try to use SDN to improve the performance of TCP in data centers.

Ghobadi *et al.* [20] introduced OpenTCP for SDN. The controller monitors the utilization of core links, and notifies the congestion control agents at end hosts to tune TCP congestion control parameters accordingly. Also, the hosts can switch to different TCP variants implemented in the operating system kernel. The solution does not introduce a new congestion control mechanism, and the computational overhead of monitoring link utilization can be high.

Hwang *et al.* [24] used SDN to introduce a congestion control protocol for data centers. The solution uses OpenFlow functionalities to change the advertised window of ACK segments to control the sending rate of senders based on bottleneck bandwidth. It uses a fixed variable, determined by the network administrator, to calculate the BDP of the network path. The fact that this value is static results in either under-utilization or congestion of the bottleneck, depending on the actual RTT of each flow. Our solution does not rely on a static value to calculate the BDP of the path. The other problem with SCCP [24] is that it requires large switch buffers in order to eliminate packet losses. Our solution requires only minimal buffering in the switches.

Lu *et al.* [36, 37] introduced another SDN-based TCP congestion control for data center networks, called SDTCP. The OpenFlow switch sends a congestion notification message to the controller as soon as the occupancy of its buffer is higher than a pre-defined threshold. The controller chooses the oldest flow and reduces the advertised window of the next ACK for it. Hence, the mechanism prevents segment losses at the buffer. The solution is only for the scenario that all the flows share the same path and have equal RTTs. The criteria of choosing the flow to punish is not necessarily fair. The control loop could potentially become large if the chosen flow is not suitable for reducing the sending rate (e.g., it leaves the network soon).

Jouet *et al.* [29] used SDN's central view to give precise values to congestion control parameters of TCP. The controller collects information about the network and traffic properties, and calculates congestion control parameters such as retransmission timers, RTOmin, RTOmax, and congestion window bounds. The controller sends the calculated values to the sender host using JSON/REST northbound interface, upon arrival of the flow in the host. Although the solution reduces flow completion time and packet loss rate, compared to regular TCP, it heavily relies on enough switch buffers. Also, the operation of the protocol requires direct connection between hosts and the controller, which is not necessarily practical in all scenarios. The main problem with OTCP is that it cannot dynamically control the flows.

Abdelmoniem *et al.* [1] designed an SDN-based solution to mitigate the TCP in-cast problem in data centers. The controller monitors the switch buffers in pre-defined intervals. When the occupancy of a buffer reaches a pre-defined threshold, the controller transitions to in-cast-On state and notifies the hosts (through their hyper-visors) to reduce their sending window. The system stays in in-cast-On state until the occupancy of the buffer reduces to another pre-defined threshold. The solution reduces the segment drops related to TCP in-cast problem. However, it does not prevent congestion completely, and it heavily relies on the pre-defined value of the monitoring intervals.

Bao *et al.* [6] introduced ECTCP, which actively controls the sending rate of flows to ensure the link is fully utilized with low latency. The controller updates bottleneck bandwidth allocation and ACK sending rates of the flows when it receives a notification of SYN or FIN arrival to a switch. Although the solution reduces flow completion time and segment drops compared to TCP, it still relies on buffers in the switches.

2.5 Research Contributions

The main challenge for generic network solutions is the lack of information about the network. The loss-based solutions gradually saturate the bottleneck links and buffers to create packet drops and detect congestion. Although the delay-based solutions prevent congestion in earlier stages, compared to loss-based approaches, their estimation of the states of the network may be incorrect. One contribution the proposed solution is to use the information provided by the SDN controller to make congestion control decisions more efficiently.

The SDN-based solutions enhance the transport-layer protocol performance by either tuning the congestion control mechanisms or improving the congestion detection. However, none of these remove the main impacts of congestion, which is queue build-up in the switch buffers. The proposed solution exploits SDN's view of the network to avoid queue build-ups, hence preventing the congestion.

2.6 Summary

This chapter presented background knowledge and related work for the research done in this thesis. It reviewed the network protocol stack and the operation of the Internet. In particular, it explained TCP and its features. Also, it described the SDN architecture as well as its benefits and challenges. In addition, it explained prior research work related to the proposed solution in the thesis. The next chapter presents the SDN simulation tool developed for evaluating the proposed solution.

Chapter 3

SDN Simulation Tool

This chapter discusses the SDN simulation tool that I developed. Section 3.1 presents an overview of the system. Section 3.2 focuses on the design and implementation of the network simulator module. Section 3.3 presents the verification test setup and results. Finally, Section 3.4 summarizes the chapter.

3.1 System Architecture

The SDN simulation tool is an object-oriented application developed in Java. It performs simulation studies with up to two factors and outputs the results in a spreadsheet file with table and graph formats (see Appendix A).

Figure 3.1 illustrates the logical architecture of the simulator. Each experiment, defined in a Scenario Handler, uses Traffic and Topology Generators to determine the Network Simulator properties. The Scenario Handler uses nested loops defined via study factors and keeps the results of the simulation for each combination. After running the simulation for all levels of the factors studied, the Scenario Handler passes the results to the Output Handler, which generates the output as tables and graphs in a spreadsheet file. Table 3.1 represents the list of available study factors for experiments. Each of the numerical factors are defined by distribution type, mean, and standard deviation.

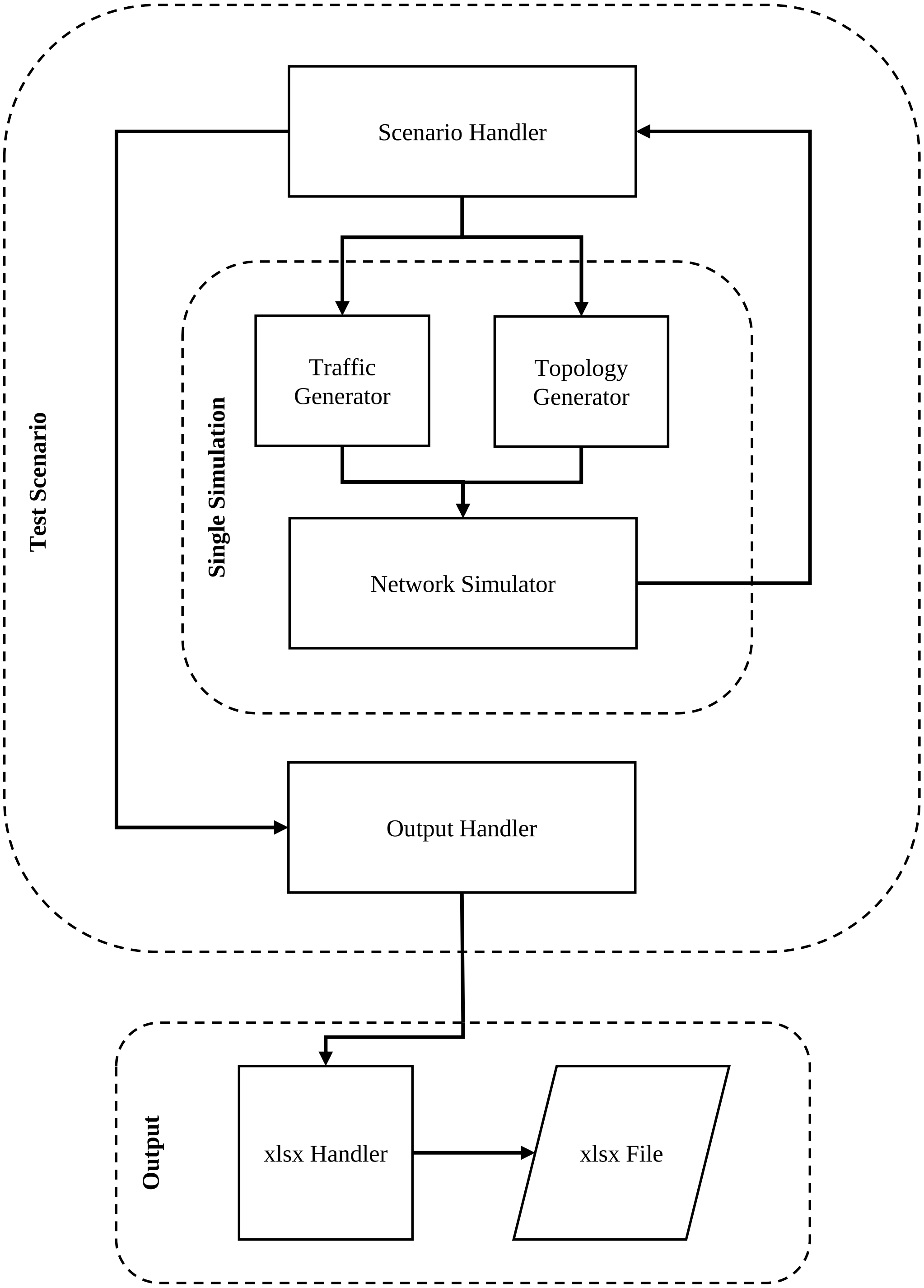


Figure 3.1: Logical architecture of the system.

Table 3.1: List of the available factors for simulation studies.

Traffic Factors	Topology Factors
Number of Flows	Network Type (Dumbbell, Data Center, Parking Lot)
Flow Sizes	Access-Link Propagation Delay
Flow Inter-Arrival Times	Network-Link Delay Type (LAN, WAN)

3.1.1 Topology Generator

The user can use the Topology Generator to specify the desired network topology for the experiment. This module provides the user with the choice among different topology types. For each type of topology, the user can choose the link delay category. The access link propagation delay is the only random variable in the topology generator, and the user can use the *Random Variate Generator* (RVG) module to generate the access link delays with the desired distribution for the hosts.

The current version of the simulator implements Dumbbell, Parking-Lot, and Data-Center topologies. For each of them, the user can choose the network delay category between Local and Wide Area. The former sets the propagation delay of network links in the range of $1 - 10$ μs while the latter sets them in the range of $1 - 10$ ms.

3.1.2 Traffic Generator

This module generates the network traffic specified by the user for the simulation experiment. The traffic is characterized by *Number of Flows*, *Size of Flows*, and *Flow Inter-Arrival Times*. The user can use the RVG module to generate the value for each of these parameters randomly with the desired distribution. The RVG module takes distribution type, mean, and standard deviation as input parameters, and provides a randomly generated value as output. The current version can generate random variables with Constant, Uniform, Exponential, Gaussian, and Log-Normal distributions. It also has a seed control mechanism to enable reproducible random values.

3.2 Network Simulator

The Network Simulator class is the core module of the SDN simulation tool. It is a *Discrete-Event Simulator* (DES) that models the transport and network layers of an SDN. This class provides the user with entity creation methods alongside a method named *run*, which implements the generic DES loop represented in Figure 3.2.

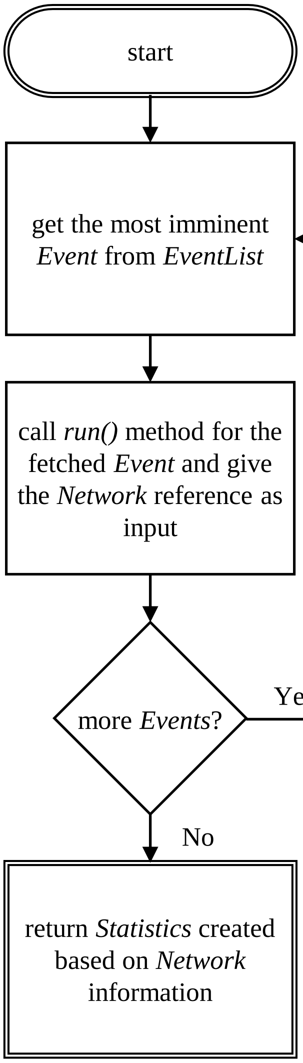

start
get the most imminent *Event* from *EventList*
call *run()* method for the fetched *Event* and give the *Network* reference as input
more *Events*?
Yes
No
return *Statistics* created based on *Network* information

Figure 3.2: Network Simulator workflow.

Figure 3.3 shows a simplified class diagram of the Network Simulator module. The *Simulator* class contains a *Network* object that holds the network entities and the *EventList*. In each loop, the most imminent event is fetched from the *EventList* and its *execute* method is called, which triggers the call-sequence for updating the packet and network states and creating possible future events. The *Segment* and *Packet* classes model the transport and network layer, respectively. All the network entities are defined as abstract classes to facilitate developing new versions of them without the need to change the interfaces.

3.2.1 Events

The abstract class *Event* represents spontaneous occurrences that change the state of the Network Simulator. The *Event* class has an abstract method named *execute*, which must be implemented by any type of event inherited from it. The *execute* method gets the *Network* object reference as input and triggers a sequence of method calls to update its state. Also, the *Event* has a single attribute named `eventTime`, which determines the time at which it must be executed.

As Figure 3.4 illustrates, the abstract class *PacketEvent* extends the *Event* class and represents any event in the network related to a single packet. Hence, it has an attribute with the type *Packet* to represent the packet associated with it. *ArrivalToNode* and *DepartureFromNode*, both extending *PacketEvent*, are the only event types implemented in the current version of the simulator.

The *ArrivalToNode* class models the behavior of a network node when a packet arrives. The `nodeID` is an attribute in this class, and gets its value when the event is constructed. This class implements the *execute* method by determining the node type and calling the *recvPacket* method of the corresponding node from the *Network*.

The *DepartureFromNode* class models the behavior of a node when a packet is sent. The departure of a packet is considered to be the same as the start of transmission of the packet by the egress link. For this reason, `linkID` is defined as an attribute to this type of event. The *DepartureFromNode* class implements the *execute* method by calling the *trasmitPacket* method for the link corresponding to the `linkID` in the *Network*.

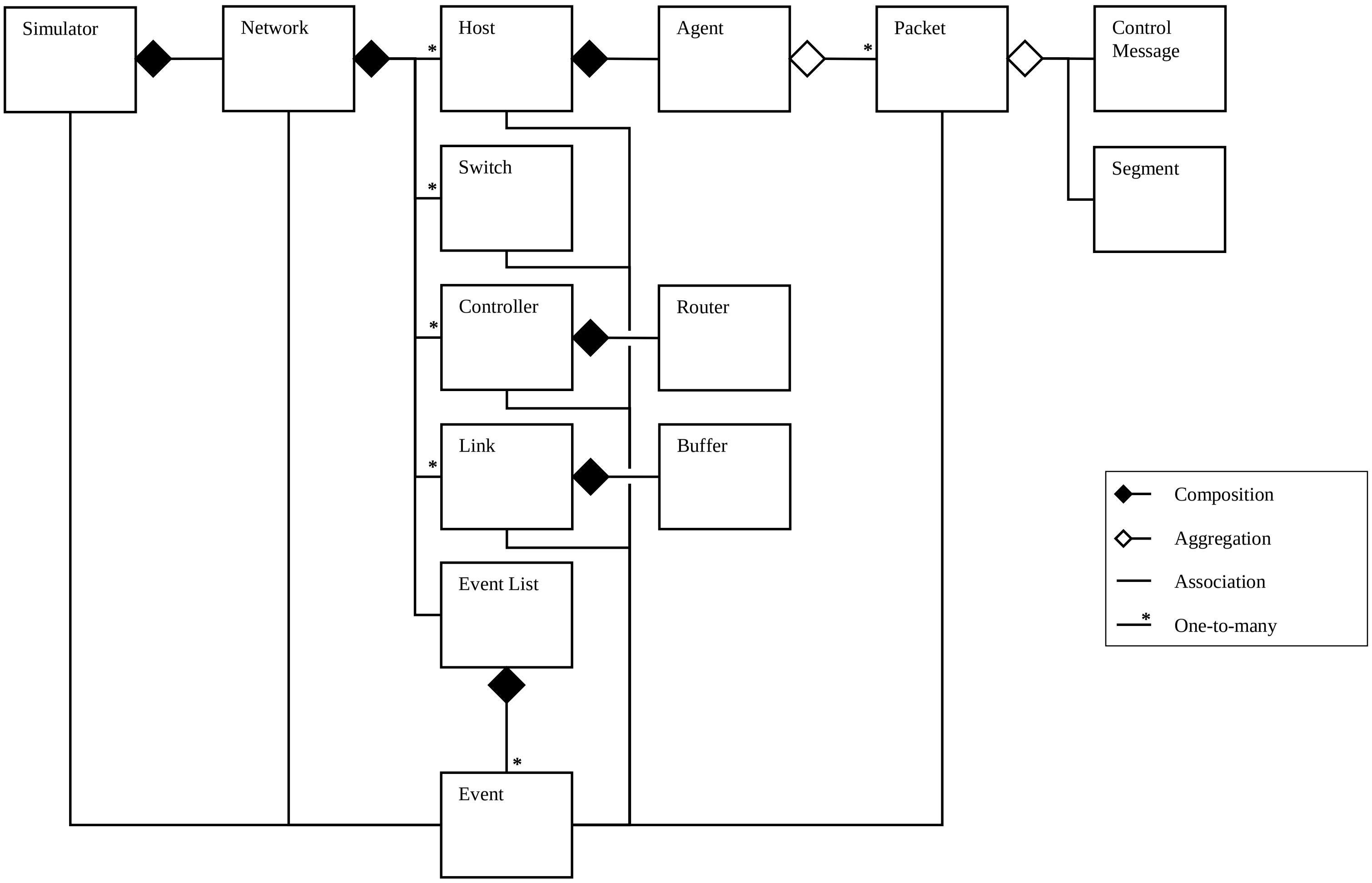


Figure 3.3: Simplified class diagram of the Network Simulator.

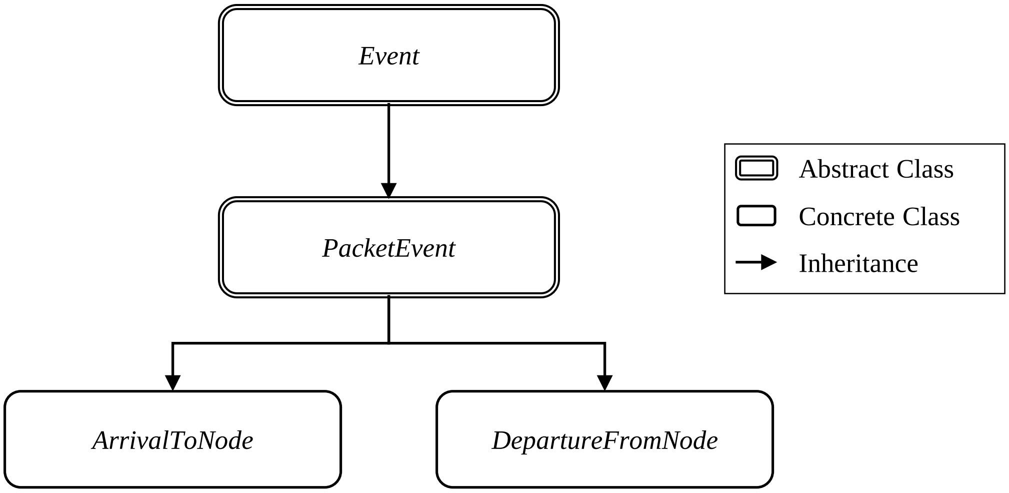


Figure 3.4: The type hierarchy of simulator events.

3.2.2 Entities

Figure 3.5 illustrates the type hierarchy of the entities in the Network Simulator. All network entities extend the abstract class *Entity*, which has a single attribute called `ID`. In the current design of the simulator, *Node*, *Link*, and *Buffer* are abstract types extending *Entity*. The abstract type *Node* itself has three abstract sub-classes named *Controller*, *Switch*, and *Host*. The remainder of this section describes each of these types separately.

Controller

The abstract class *Controller* models the generic properties and functionalities of an SDN controller. It contains a *Router* module that is in charge of finding the best path for the flows in the network. This class models the SDN southbound communication via implementation of a method named *sendPacketToSwitch*.

The concrete class *DefaultController* extends the abstract class *Controller*. It can send two types of SDN control messages (encapsulated in *Packets*) to a specific *SDNSwitch* in the network. The *Router* module in *DefaultController* uses Dijkstra's algorithm for finding the shortest path for each flow in the network.

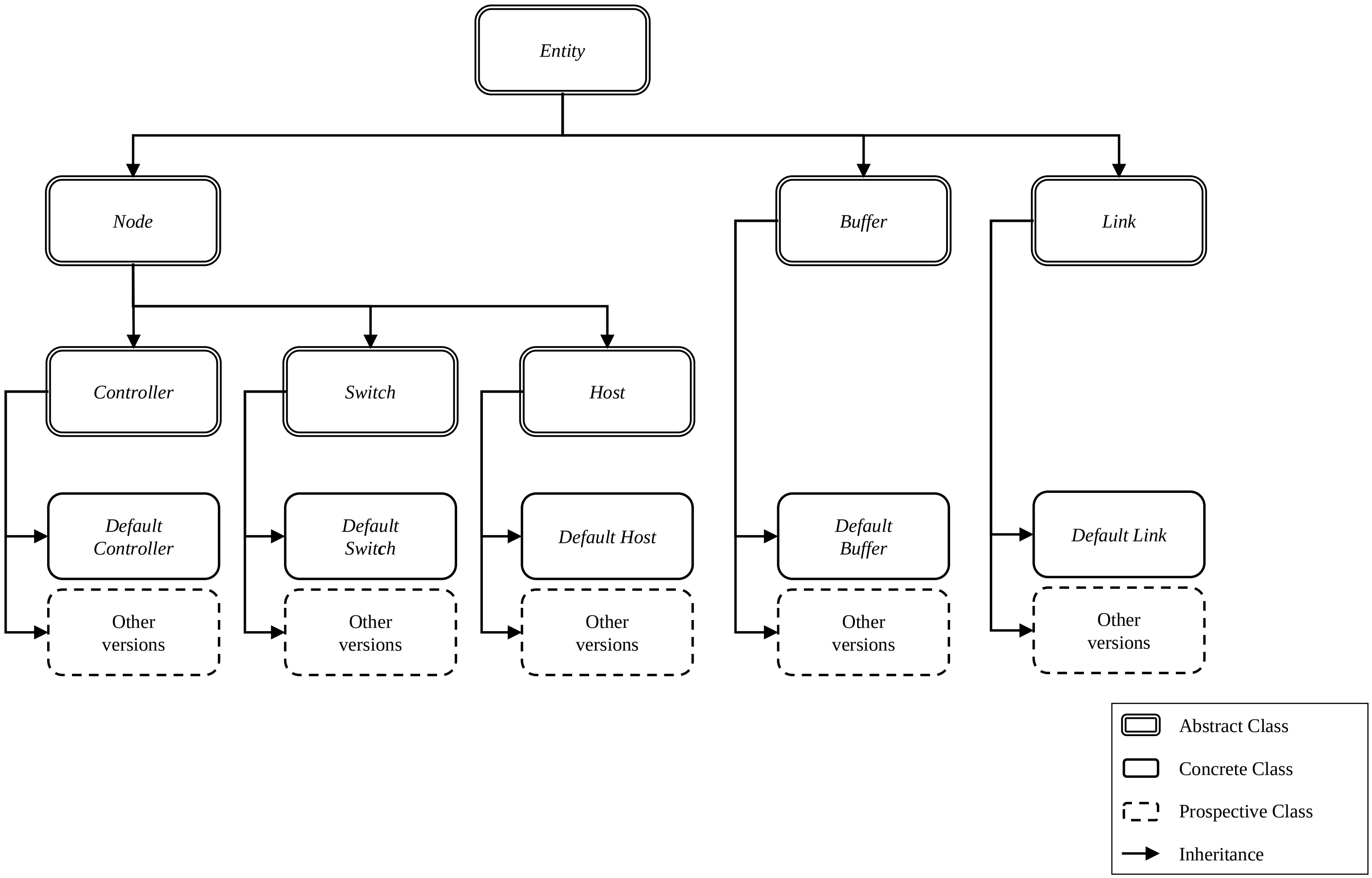


Figure 3.5: Entity type hierarchy in the Network Simulator.

The *DefaultController* implements the *recvPacket* method based on the flow-chart depicted in Figure 3.6. Upon arrival of a packet, the controller checks the `segmentType`. If it is a *SYN* segment, the controller obtains the shortest path from the *Router* module. Then it sends the `pathSetup` control messages to the corresponding switches in the flow path. Once the flow path is established, it sends the *SYN* segment back to the switch that it originally came from. If the segment is a *FIN*, the controller updates its network information and sends the *FIN* segment back to the switch that it came from. Then, it sends a `pathRemoval` control message to all the switches associated with the flow.

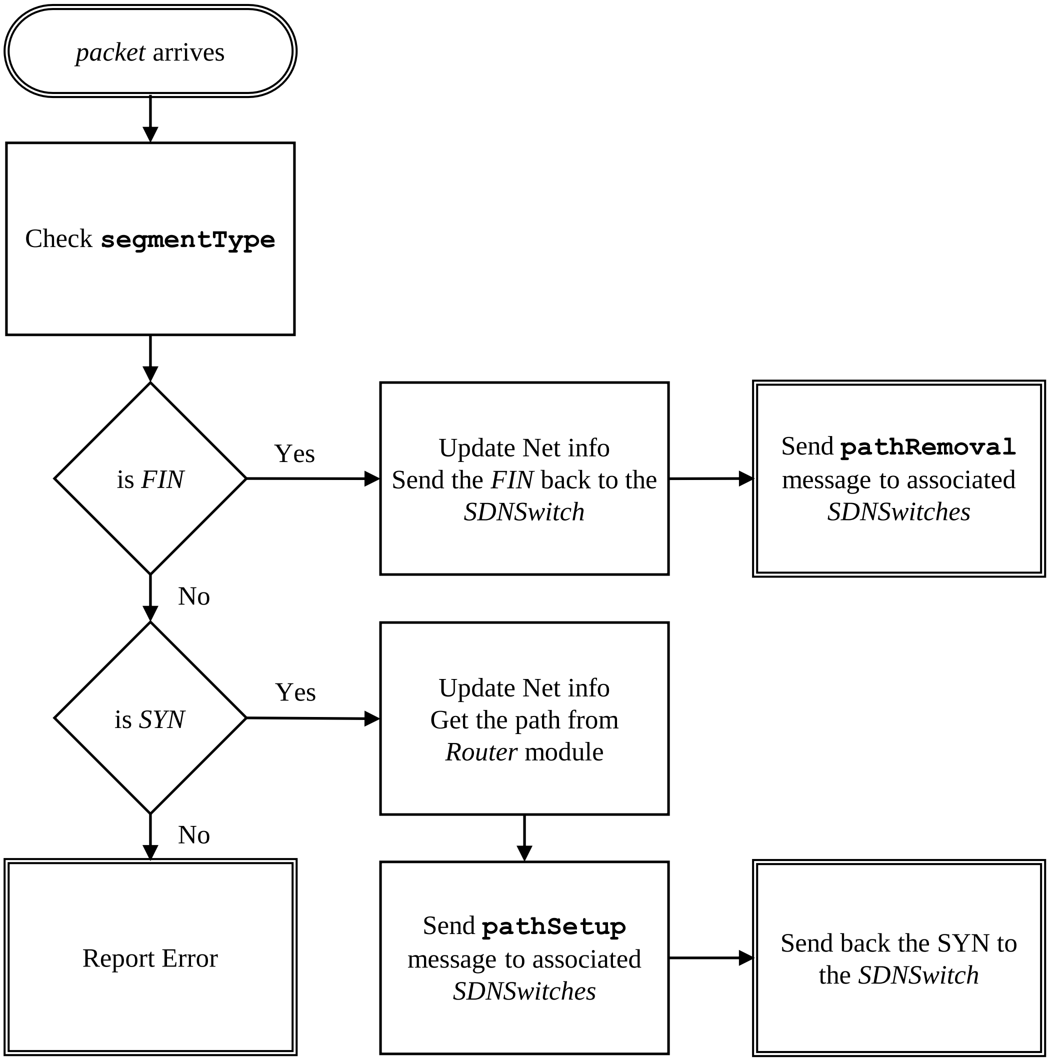

packet arrives
Check segmentType
is FIN
Yes
Update Net info
Send the FIN back to the SDNSwitch
Send pathRemoval message to associated SDNSwitches
No
is SYN
Yes
Update Net info
Get the path from Router module
No
Report Error
Send pathSetup message to associated SDNSwitches
Send back the SYN to the SDNSwitch

Figure 3.6: The *recvPacket* flow-chart in *DefaultController*.

SDNSwitch

The abstract class *SDNSwitch* extends the abstract type *Node*. It represents the general properties and functionalities of an OpenFlow switch. It contains a *flowTable* that holds the *egress linkID* for each *flowID*. It models the forwarding functionality of the OpenFlow switch by implementing *forwardToHost*, *forwardToSwitch* and *forwardToController* methods.

The concrete class *DefaultSDNSwitch* extends the abstract class *SDNSwitch*. It implements *recvPacket* based on the flow-chart presented in Figure 3.7. When a packet arrives, it checks the `packetType`. If it is a `CtrlPacket` coming from the controller, the switch updates its *flowTable* based on it. When the type is a `FlwPacket`, if the destination host is connected to the switch, the packet is forwarded to the host. Otherwise, if the *flowTable* has an entry for the *flowID*, it forwards the packet to the next switch via the egress link. In the case that the *flowTable* does not have an entry for the *flowID*, the packet is forwarded to the controller.

Host

The abstract class *Host* extends the class *Node* and models a physical host capable of sending and receiving packets. It contains an attribute called `transportAgent` from the type *Agent* to model the transport protocol. The class provides the *sendSegment* method, which encapsulates a *Segment* into a *Packet* and sends it to the network via the access link.

In the current version of the simulator, *DefaultHost* is the concrete class that extends the abstract class *Host*. The *recvPacket* method gets the packet from the network, decapsulates the segment from it, and passes the segment to the `transportAgent`.

Agent

The abstract class *Agent* models the transport layer agent in a host. The `sourceHostID` and `destinationHostID` are the attributes of this abstract class. It also contains a *Flow* object that holds the flow information such as flow size, arrival time, and statistical counters.

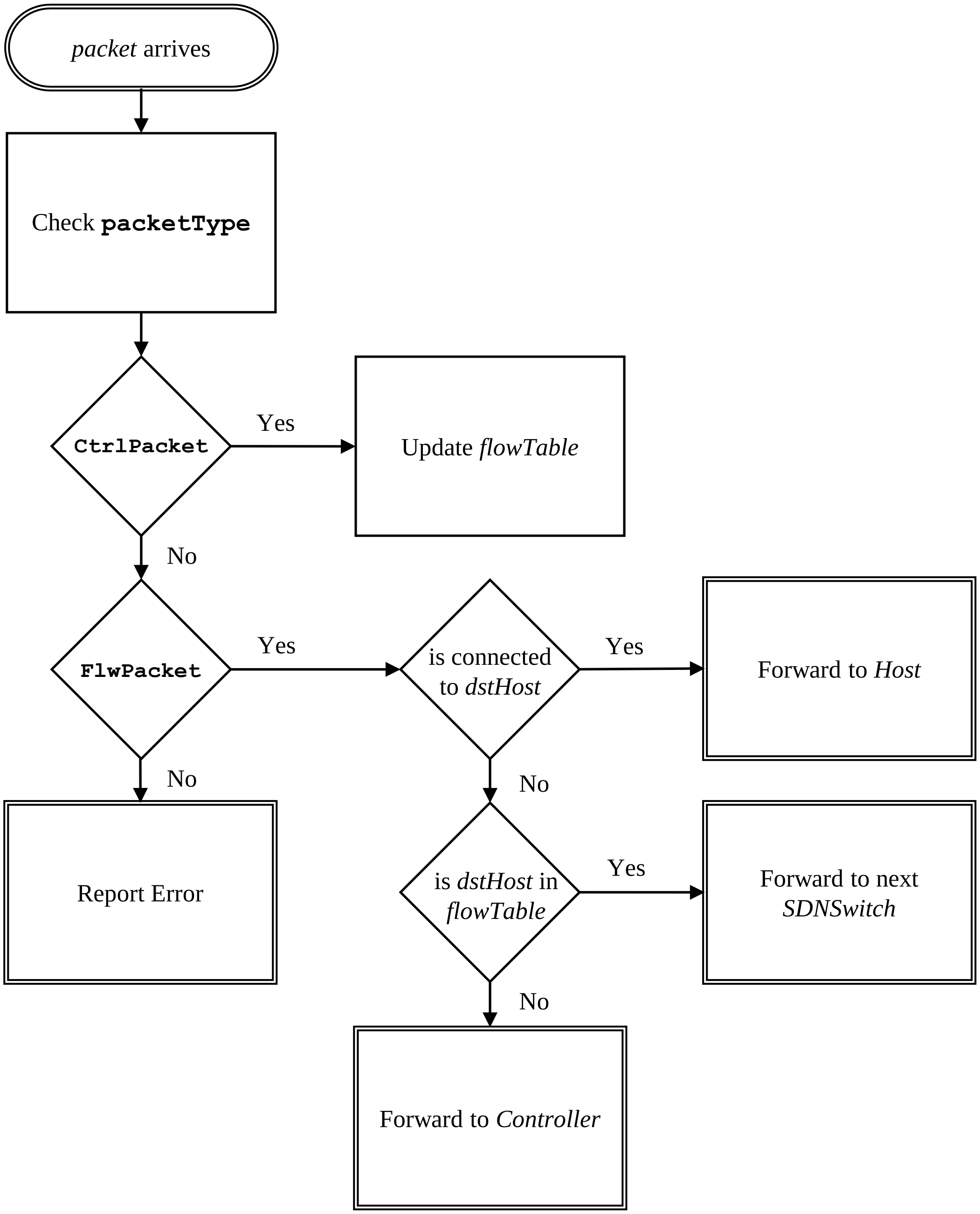


Figure 3.7: The *recvPacket* flow-chart in *DefaultSDNSwitch*.

DefaultSender and *DefaultReceiver*, both extending abstract type *Agent*, are the concrete classes implementing the default transport protocol implemented in the current version of the simulator. This protocol has a client-server and connection-oriented design, and uses sliding window and sequence number mechanisms without any congestion control algorithm.

Link

The abstract type *Link* represents the transmission medium of a network. This class models the physical link properties with `bandwidth`, `propagationDelay`, and `lossProbability`. It also contains the `sourceNodeID` and `destinationNodeID` representing the end nodes of the link. The buffer is modeled as an attribute of the *Link* with the type *Buffer*. This class models the functionality of the actual network link with *bufferPacket* and *transmitPacket* methods.

The concrete class *DefaultLink* extends the abstract type *Link*, and models a simple wired link with zero loss. It implements the *bufferPacket* method by getting the `bufferTime`, from its *Buffer* attribute, for each arriving packet and creating the *DepartureFromNode* event based on it. The *transmitPacket* method, which is called during the execution of *DepartureFromNode* event, updates the buffer occupancy, then calculates the transmission and propagation delays based on the properties of the link, and creates the *ArrivalToNode* event for the packet.

Buffer

The abstract class *Buffer* is the model for a network buffer. It models the properties of the buffer with `capacity` and `occupancy` (both in number of packets) and the `bufferingPolicy`. This class has an abstract method named *getBufferTime*, which is called in the *Link* object.

The concrete class *DefaultBuffer* extends the abstract type *Buffer*. It uses FIFO as the default buffering policy in the current version of the simulator. The implementation of the *getBufferTime* method follows the flow-chart depicted in Figure 3.8. When the *Link* object calls *getBufferTime*, the method checks the `occupany`. If the buffer is full, it will return `Minus_Infinity` to signify packet drop. Otherwise, the `bufferTime` is calculated based on the departure time of the previous packet buffered. Before returning the `bufferTime`, the method updates the departure time of the most recent packet and increases the `occupancy`.

3.3 Verification

This section describes the verification test scenario for the network simulator module and reports the results. The test scenario objectives focus on:

- Default transport protocol implemented by *DefaultSender* and *DefaultReceiver*;
- Routing module and flow setup mechanisms of the controller; and
- Forwarding plane functionalities and link-level delay calculations.

Figure 3.9 shows the network topology for the verification test. There is a single pair of hosts in the network. One of them contains a *DefaultSender* agent and the other one has a *DefaultReceiver*. The *Sending Window* (*Swnd*) is set to five. The only traffic flow in this scenario arrives at time zero and has 10 data segments to send. There are four *SDNSwitches* in the network providing two possible paths from the sender to the receiver. However, based on the link properties in Table 3.2, the shortest path is: $Sw_0 \rightarrow Sw_1 \rightarrow Sw_3$

Table 3.2: Link properties of the topology.

Name	Bandwidth(Gb/s)	Propagation Delay(μs)
SL_0	8	1
RL_0	8	1
CL_0	8	1
NL_0	8	1
NL_1	8	1
NL_2	4	1
NL_3	4	1

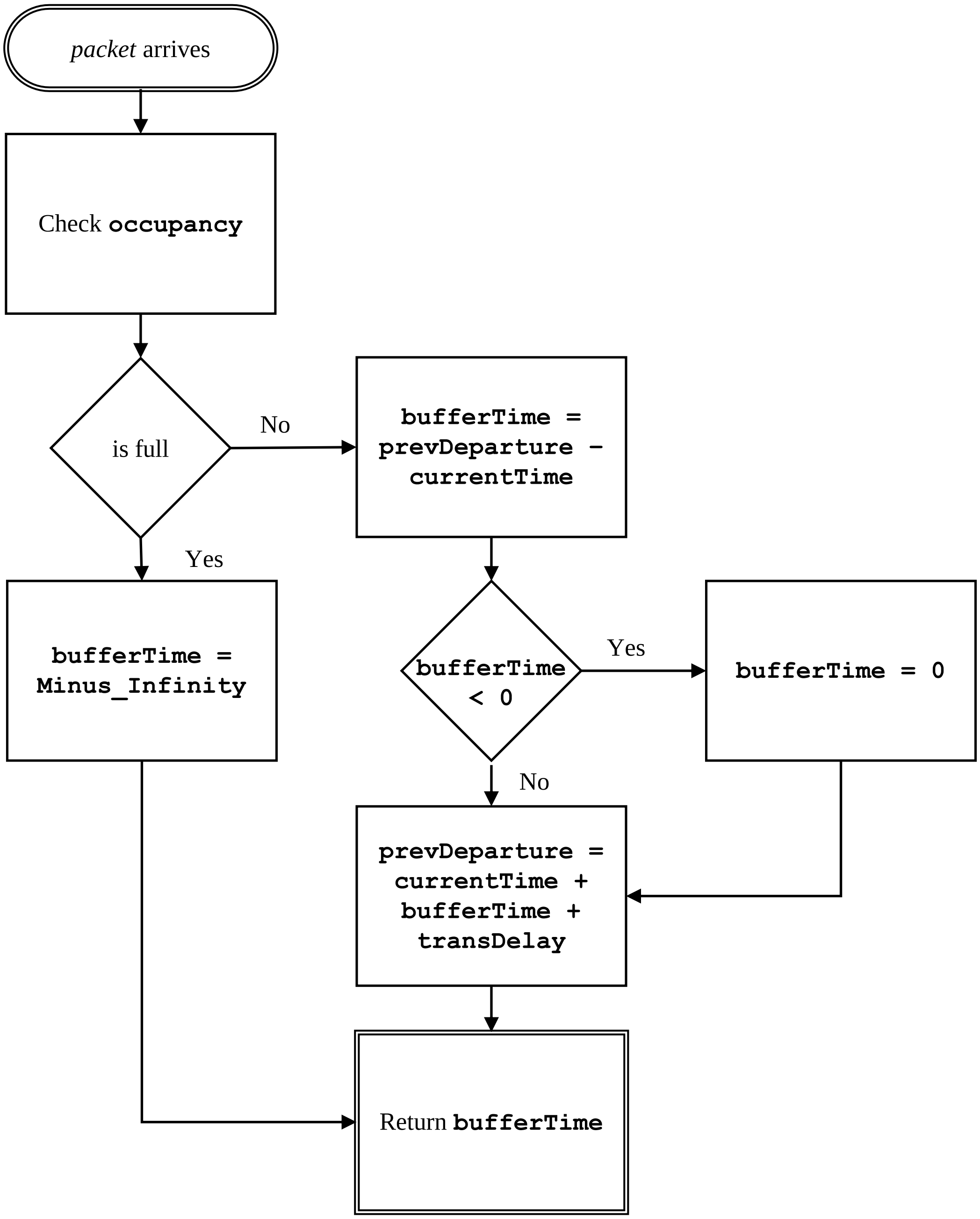

packet arrives
Check occupancy
is full
No
bufferTime = prevDeparture - currentTime
Yes
bufferTime = Minus_Infinity
bufferTime < 0
Yes
bufferTime = 0
No
prevDeparture = currentTime + bufferTime + transDelay
Return bufferTime

Figure 3.8: Buffering algorithm implemented in *DefaultBuffer*.

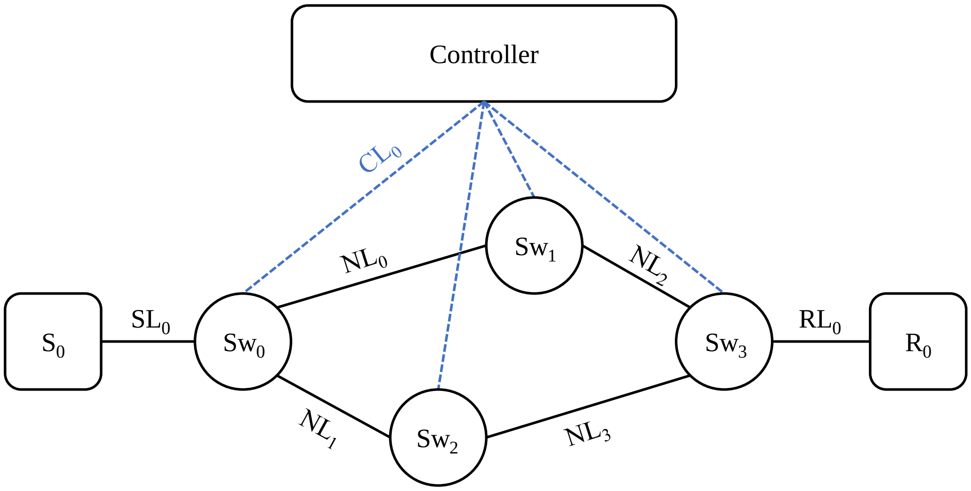


Figure 3.9: Network setup for verification test.

The first metric is *FlowSetupDelay*, defined as the amount of time between sending the *SYN* segment and receiving the *SYNACK* at the sender transport agent. The expected value for this metric is 10.44 μs, based on the propagation and transmission delays of a *SYN* segment of size 40 *Bytes* traversing the path:

$S_0 \rightarrow Sw_0 \rightarrow Controller \rightarrow Sw_0 \rightarrow Sw_1 \rightarrow Sw_3 \rightarrow R_0 \rightarrow Sw_3 \rightarrow Sw_1 \rightarrow Sw_0 \rightarrow S_0$

The second metric is *rtt*, defined as the amount of time between sending a *Data* segment and receiving its *ACK* at the sender transport agent. The expected value for this metric is 12.16 μs, based on the propagation and transmission delays of *Data* and *ACK* segments, with the size 1000 and 40 *Bytes* respectively, traversing the path:

$S_0 \rightarrow Sw_0 \rightarrow Sw_1 \rightarrow Sw_3 \rightarrow R_0 \rightarrow Sw_3 \rightarrow Sw_1 \rightarrow Sw_0 \rightarrow S_0$

Figure 3.10 illustrates the simulation results for the single flow scenario in form of transport agent sequence number plot. The values for *FlowSetupDelay* and *rtt* are 10.44 μs and 12.16 μs, respectively which validates:

- Transport protocol 3-way-handshake mechanism;
- Controller routing and flow setup mechanisms; and
- Forwarding plane functionality and link-level delay model.

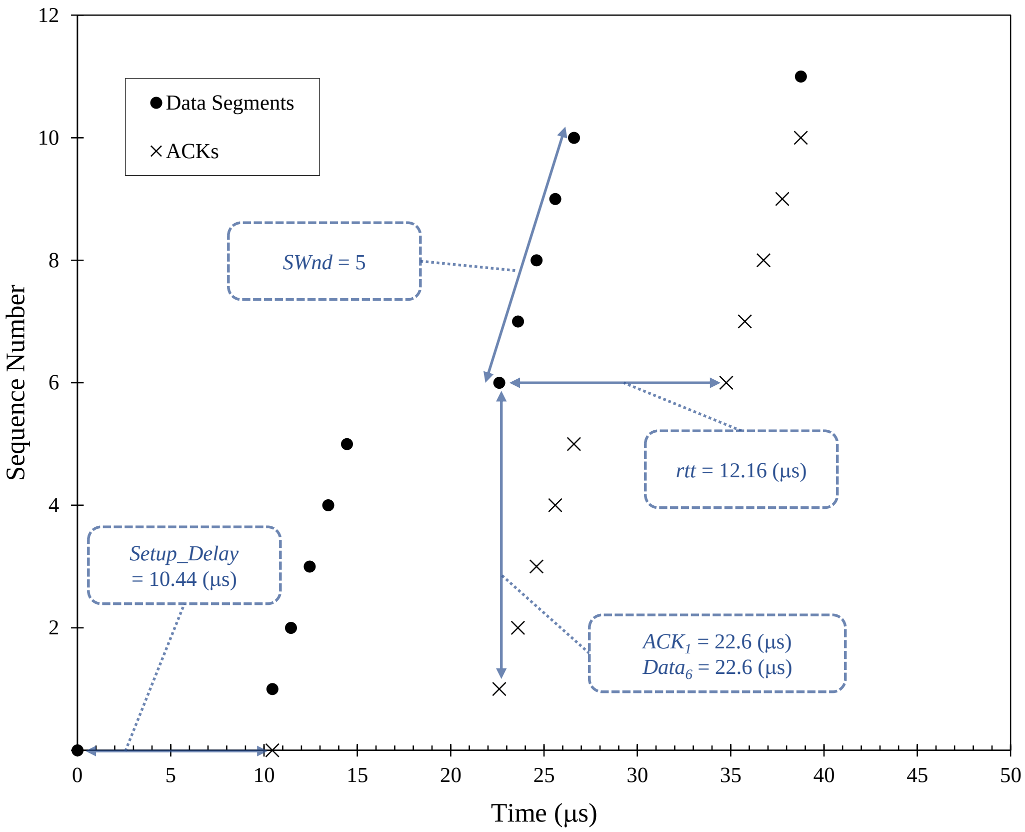


Figure 3.10: verification test result for single flow scenario.

The number of in-flight *Data* segments is equal to the pre-defined value of $Swnd = 5$. Also, upon receiving the ACK for the *Data* segment with the lowest sequence number, the sender agent slides the sending window and sends the next *Data* segment. These observations validate the sliding-window and sequence number mechanisms of the transport agent.

3.4 Summary

This chapter discussed the architecture of the developed SDN simulation tool. In particular, it described the design and implementation of the network simulator module. In addition, the verification test setup and results were presented. The next chapter discusses the SDN congestion control solution proposed in this thesis.

Chapter 4

Proposed Solution

This chapter describes the proposed solution. Section 4.1 gives an overview of the solution. Section 4.2 focuses on the analytical model for the proposed solution. Section 4.3 represents the simulation model implemented from the analytical model. Section 4.4 describes the verification test scenarios and results. Finally, Section 4.5 summarizes the chapter.

4.1 Solution Overview

Persistent queues are one of the manifestations of network congestion. There are two main scenarios that result in the formation of queues in network buffers.

The first scenario is when the inbound arrival rate of segments exceeds the outbound departure rate. This situation happens because the transport agents send bursts of data segments and do not have any information about the bottleneck bandwidth along the path. Hence, the segment arrival rate to the bottleneck link is determined by the access link bandwidth of the sender host. As Figure 4.1 depicts, if $AccessLinkBw > btlBw$, a queue of segments forms in the bottleneck link buffer. This queue is usually transient. However, other flows in the network that share that bottleneck link might send bursts of segments too, which can result in a persistent queue, and eventually packet drops.

To reduce the likelihood of persistent queues and segment drops, the arrival rate of

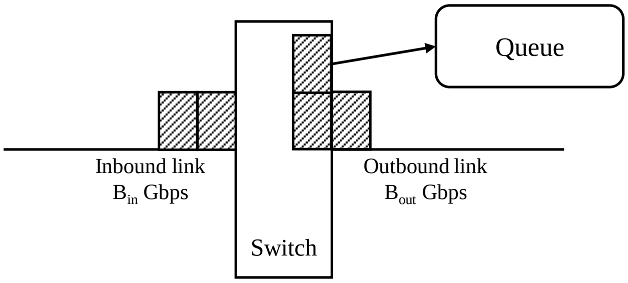


Figure 4.1: Queue forms when $B_{in} > B_{out}$.

segments in a path should not exceed the transmission rate of the bottleneck link. This can be done using inter-segment delays, as shown in Figure 4.2.

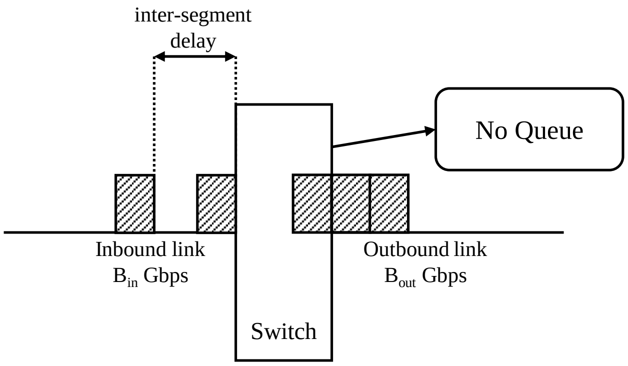


Figure 4.2: Inter-segment delay eliminates the queue.

The second scenario that forms queues in the buffers is the coincident arrival of segments on multiple inbound links that share the same outbound link. As Figure 4.3 shows, if the bursts of segments on these inbound links arrive at the same time (or have significant overlap), then a queue forms at the outbound link. One example of this scenario is the TCP in-cast problem in data centers [4, 13], in which many segments from multiple flows arrive at the access switches at the same time, due to parallel requests for data from multiple servers.

Figure 4.4 shows how the addition of inter-flow delays solves the problem. When the bursts of segments from each host are phased to arrive in series, rather than parallel, there will be no queue formed at the buffer of the shared outbound link.

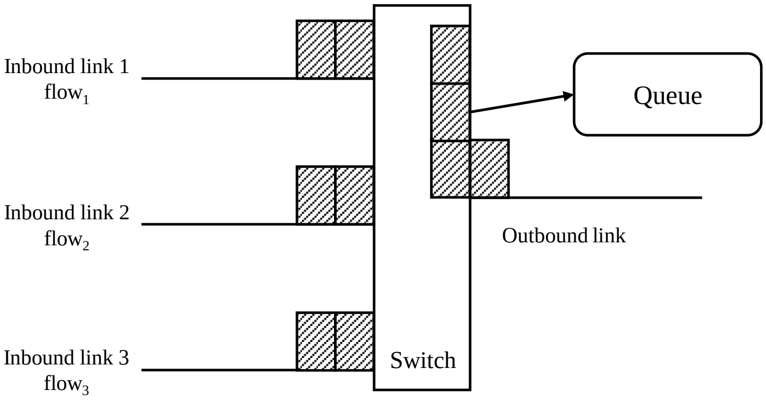


Figure 4.3: Multiple inbound links and single outbound link.

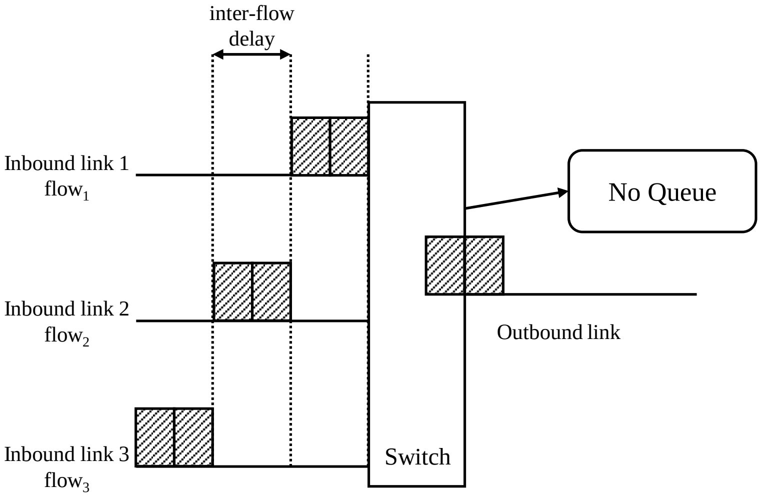


Figure 4.4: Inter-flow delay eliminates the queue.

These scenarios that result in the formation of queues in the network are well-known to the research community. Multiple research works have been done to resolve this issue [19, 31, 56]. However, the lack of information about the network, such as the bottleneck bandwidth for each flow, prevented these solutions from resolving the issue effectively [3, 33].

In this thesis, we exploit the properties of SDN to propose a solution that implements the inter-segment and inter-flow spacing ideas depicted in Figure 4.2 and Figure 4.4. We use the simplified network model shown in Figure 4.5 for analytical modeling of the solution. We make the following simplifying assumptions:

- Each network switch has exactly one inbound and one outbound link.
- Data transfer always happens in a single direction, from senders to receivers.
- The processing delays for hosts, switches, and controller are negligible.
- The controller is capable of communicating with sender agents through the network.

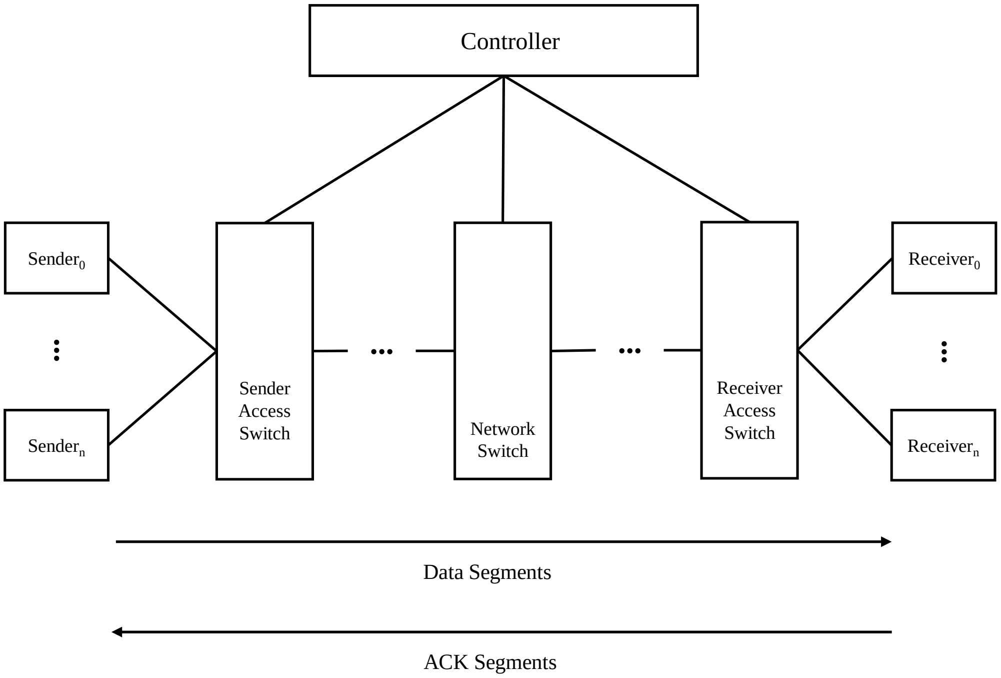


Figure 4.5: The network topology used for analysis.

4.2 Analytical Model

The proposed solution aims to prevent queues forming at bottleneck links. The SDN controller plays a central control role, and provides sender agents with necessary information. The protocol aims to eliminate queues forming at shared outbound links by adding inter-flow delays to all the flows traversing the same access switch. Also, it ensures that each flow adds inter-segment delays according to the bottleneck link of the path.

To reach the above mentioned goals, five congestion control parameters are defined for the protocol. The controller calculates these values based upon arrival of *SYN* or *FIN* segments from an access switch, and notifies the sender agents connected to that access switch with the updated values for the parameters.

The protocol defines a Sending Cycle (*sCycle*) for all flows connected to the same access switch. The lifetime of a *sCycle* is determined based on the arrival of new flows or departures of existing ones. Each *sCycle* consists of multiple Sending Intervals (*sInterval*). A *sInterval* is a period of time that is partitioned and dedicated to the flows connected to the same access switch. Each flow is allowed to send a specific number of segments with a specific inter-segment delay in its share of an *sInterval.* Table 4.1 defines the parameters used in the analytical model.

Table 4.1: Parameters and Definitions.

Parameter Name	Definition
$rtt_{i(s)}$	The round-trip time of a segment (with size s) for flow i
F_i	The set of flows for sender access switch i
$linkTotalDelay_{(s)}$	The transmission and propagation delay of a link for a segment with size $= s$
$btlBw_i$	The lowest link bandwidth in the path of *flow* i
$TransDelay$	The transmission delay
$PropDelay$	The propagation delay
$QDelay$	The queuing delay
CL_i	Control link corresponding to *flow* i
AL_i	Access link corresponding to *flow* i
NL	Network link

4.2.1 DelayToNextCycle

$DelayToNextCycle_i$ is the wait time for starting the new *sCycle* for flow i after receiving a *CTRL* segment from the controller. As Equation 4.1 represents, to calculate $DelayToNextCycle_i$, we need to determine the time to the start of the next *sCycle* (*CycleStartDelay*), and the time needed for a *CTRL* segment to arrive to the sender agent of *flow* i ($CTRLDelay_i$).

$$DelayToNextCycle_i = CycleStartDelay - CTRLDelay_i \tag{4.1}$$

Calculating CycleStartDelay

The calculation of *CycleStartDelay* varies depending upon the arrival of *SYN* or *FIN* segments. If a *SYN* segment arrives, the *sCycle* starts when the *SYNACK* segment of the corresponding flow arrives to the sender agent. On the other hand, if the received segment is a *FIN*, the new *sCycle* starts when all the *CTRL* segments are received by the sender agents.

If the segment arriving to the controller is a *SYN* segment, the controller calculates the *SYNACKDelay* of *flow j* with:

$$SYNACKDelay_j = rtt_{j(SYN)} + CLTotalDelay_{j(SYN)} - ALTotalDelay_{j(SYN)}$$

where:

$$CLTotalDelay_j = CLQDelay_j + CLTransDelay_j + CLPropDelay_j$$

$$ALTotalDelay_j = ALQDelay_j + ALTransDelay_j + ALPropDelay_j$$

$ALTotalDelay_i$ is deterministic, because we know $ALQDelay_j = 0$, and transmission and propagation delays can be calculated based on the physical properties of the link. However, calculating $rtt_{j(SYN)}$ and $CLTotalDelay_j$ requires information about the state of buffers involved.

When the *SYN* segment arrives, the controller finds the best path and sends a *flowSetupMessage* to the corresponding switches, including the sender access switch. Then, it calculates the congestion control parameters and sends the *CTRL* segments to the sender agents. Finally, it sends the *SYN* segment back to the sender's access switch. Hence, the $CLQDelay_j$ can be calculated by:

$$CLQDelay_j = CLTransDelay_{j(flowSetupMessag)} + N \times CLTransDelay_{i(CTRL)}$$

where N is the number of active flows at the sender's access switch. However, if other *SYN* or *FIN* segments have been received by the controller recently, the control link buffer may have non-zero occupancy before the start of the current control loop, and we need to bound this value. The maximum number of SYN or FIN segments that can arrive to the controller from the same access switch at any given time is less than or equal to the number of active flows connected to that switch. Hence, we can use Equation 4.2 as an upper bound for $CLQDelay_j$.

$$CLQDelay_j = N \times (CLTransDelay_{j(flowSetupMessag)} + N \times CLTransDelay_{i(CTRL)}) \quad (4.2)$$

Calculating $rtt_{j(SYN)}$ requires the analysis of the link buffers in the path of flow j. When the SYN segment leaves the controller, it should traverse the links to the receiver host in the $Data$ direction. Figure 4.6 represents the scenario with the maximum queuing delay for the SYN segment on its way to the receiver host. The maximum queuing delay happens when the SYN segment and a $Data$ segment from another flow arrive to the sender access switch at the same time. In this case, the SYN segment is transmitted right after the $Data$ segment en route to the receiver access switch. The queuing delay for the SYN segment is

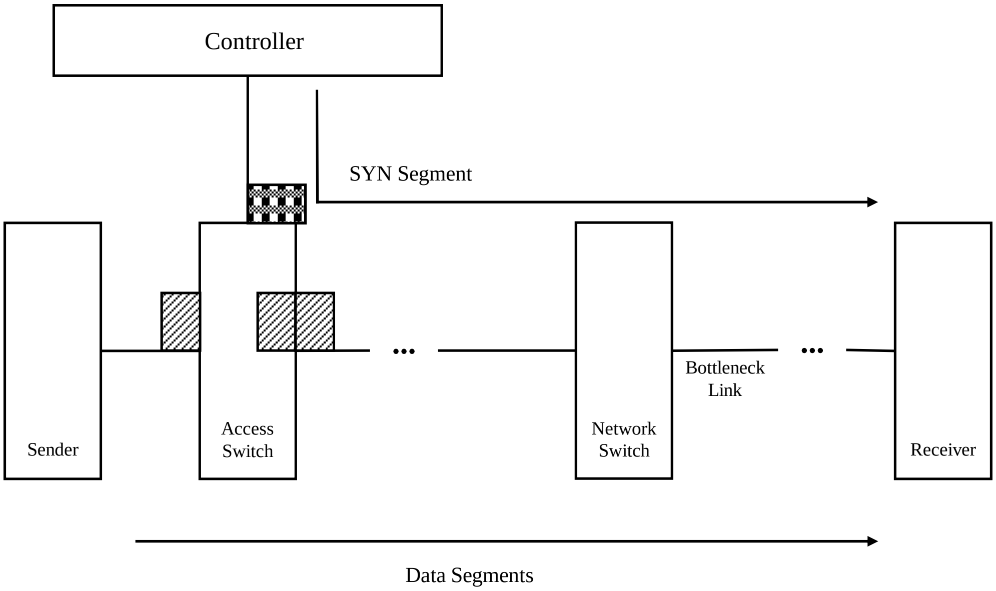


Figure 4.6: The maximum queuing delay for SYN and $SYNACK$ segments.

equal to the transmission time of the *Data* segment:

$$\frac{DataSegSize}{SharedBtlBw_{Data}}$$

where $sharedBtlBw_{Data}$ is the lowest bandwidth among the shared link on the *Data* stream before the receiver access switch. With the same rationale, the maximum queuing delay for the *SYNACK* segment is equal to:

$$\frac{ACKSegSize}{sharedBtlBw_{ACK}}$$

where $sharedBtlBw_{ACK}$ is the lowest bandwidth among the shared links on the *ACK* stream before the sender access switch. Hence, Equation 4.3 gives an upper bound for the non-deterministic part of the $rtt_{j(SYN)}$.

$$maxQDelay = \frac{DataSegSize}{SharedBtlBw_{Data}} + \frac{ACKSegSize}{sharedBtlBw_{ACK}} \tag{4.3}$$

Equation 4.4 presents the final formula for the calculation of the $SYNACKDelay_j$. Although all the maximum values for the non-deterministic delays are taken into consideration, we use the over-estimation factor $\beta \geq 1$, to ensure that the *SYNACK* segment always arrives no later than expected by the sender agents.

$$SYNACKDelay_j = \beta \times (rtt_{j(SYN)} + CLTotalDelay_{j(SYN)}) - ALTotalDelay_{j(SYN)} \tag{4.4}$$

When a *FIN* segment arrives, the controller calculates *CTRLDelay* for all the flows and uses the maximum of these values as *CycleStartDelay*. The calculation of *CTRLDelay* for each flow is discussed later in this section. In summary, Equation 4.5 represents the value of

CycleStartDelay upon arrival of *SYN* or *FIN* segments to the controller.

$$CycleStartDelay = \begin{cases} SYNACKDelay & SYN\ arrival \\ maxCTRLDelay & FIN\ arrival \end{cases} \tag{4.5}$$

Calculating CTRLDelay

$CTRLDelay_i$ is calculated by:

$$CTRLDelay_i = CLTotalDelay_i + ALTotalDelay_i$$

where:

$$CLTotalDelay_i = CLQDelay_i + CLTransDelay_i + CLPropDelay_i$$

$$ALTotalDelay_i = ALQDelay_i + ALTransDelay_i + ALPropDelay_i$$

Transmission and propagation delays are deterministic and can be calculated using physical properties of the control and access links corresponding to flow i. However, the queuing delays are non-deterministic. Hence we should determine bounds on these delays.

As Figure 4.7 represents, the maximum queuing delay happens when the *CTRL* segment (coming from the controller) and an *ACK* segment (coming from the receiver of a flow) arrive to the access switch at exactly the same time. In this case, the queuing delay for the *CTRL* segment is equal to the transmission delay of the ACK segment on the access link. Also, the minimum queuing delay for the *CTRL* segment is equal to zero, which happens when its arrival does not overlap with the arrival of any ACK segment. Hence, we can claim that the actual value of $ALQDelay_i$ has the following bounds:

$$0 \leq ALQDelay_i \leq ALTransDelay_{i(ACK)}$$

Equation 4.6 uses the upper bound to over-estimate the queuing delay.

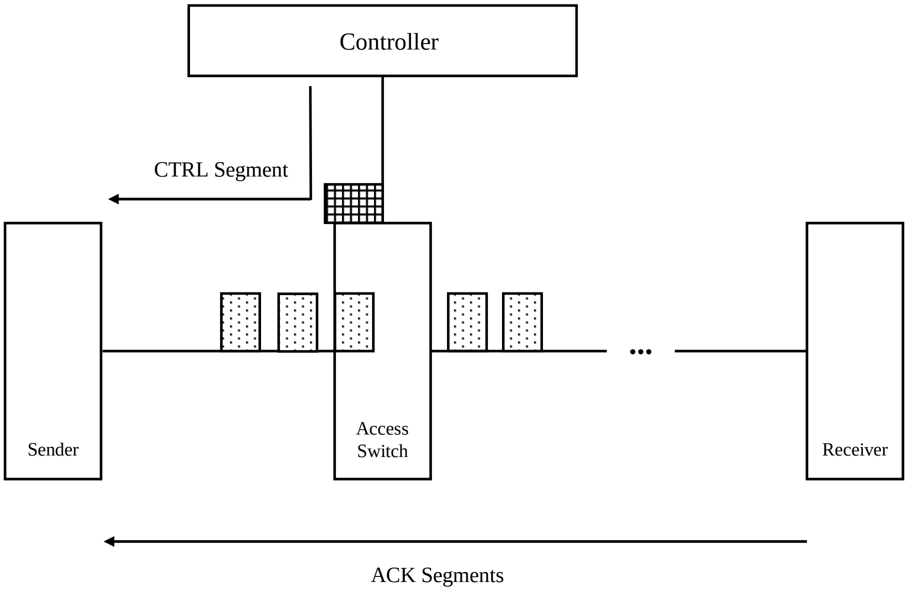


Figure 4.7: *CTRL* and *ACK* segments arrivals to the access switch.

$$ALQDelay_i = ALTransDelay_{i(ACK)} \tag{4.6}$$

For some flows, depending on whether their ACK stream overlaps with the *CTRL* segment or not, the *CTRLDelay* is over-estimated. Hence, those flows start their *sCycle* sooner than they should with an error of ϵ. Considering the bounds on the $ALQDelay_i$, Equation 4.7 represents the bounds for ϵ. This error is handled in the *sWnd* calculation.

$$0 \leq \epsilon \leq ALTransDelay_{i(ACK)} \tag{4.7}$$

Similar to the calculation of *CycleStartDelay*, determining the bounds on $CLQDelay_i$ varies depending upon the arrival of *SYN* or *FIN* segments to the controller. We need to analyze the behavior of the controller in each control loop to determine the state of the control link buffer.

When a *SYN* segment arrives, the controller finds the best path for the flow and sends a *flowSetupMessage* to the corresponding switches, including the sender access switch. Then, the controller prepares *CTRL* segments to send to the corresponding sender agents. Hence, the $CLQDelay_i$ can be calculated by:

$$\begin{aligned} CLQDelay_i = {} & CLTransDelay_{i(flowSetupMessage)} \\ & + flowCounter \times CLTransDelay_{i(CTRL)} \end{aligned} \tag{4.8}$$

where *flowCounter* is the number of flows for which the *CTRL* segment is ready to send.

If the arriving segment is a *FIN*, the controller calculates the congestion control parameters, and sends the *CTRL* segments to the corresponding sender hosts. For this, the $CLQDelay_i$ is calculated by:

$$CLQDelay_i = flowCounter \times CLTransDelay_{i(CTRL)} \tag{4.9}$$

where *flowCounter* is the number of flows for which the *CTRL* segment is ready to send.

4.2.2 sInterval

Equation 4.10 is used to calculate the value for *sInterval* in each *sCycle*. The factor $\gamma \geq 1$ is used for overestimating the maximum *rtt* among the flows to account for non-deterministic buffering delays. Using this value ensures that each interval starts after the *ACK* segments of the previous *sInterval* have arrived back at the senders.

$$sInterval = \gamma \times \max(rtt_i), \forall i \in F, \gamma \geq 1 \tag{4.10}$$

4.2.3 sWnd

Each *sInterval* is divided into equal periods of time dedicated to each flow:

$$\frac{sInterval}{N}$$

The maximum number of segments that the bottleneck link of flow i can transmit in the dedicated time period is equal to:

$$\lfloor \frac{sInterval \times btlBw_i}{N \times DataSegSize} \rfloor$$

Equation 4.11 calculates the *sWnd*. We add an aggressiveness factor, $0 < \alpha \leq 1$, to control the utilization of the shared network link. To handle the error mentioned in Equation 4.7, the calculated *sWnd* is reduced by one, so each sender agent can skip one segment at the start of each *sInterval*. Finally, to ensure that all flows have the chance to send in each *sInterval*, the minimum value for *sWnd* is one.

$$sWnd_i = \max(1, \lfloor \alpha \times \frac{sInterval \times btlBw_i}{N \times DataSegSize} \rfloor - 1) \tag{4.11}$$

4.2.4 sInitialDelay

sInitialDelay is added at the start of each *sCycle* to create the *interFlowDelay* needed to make sure the access switch outbound link buffer is not overloaded by segments. The controller allocates an index to each flow connected to the same access switch, starting from zero. Equation 4.12 ensures that there is no overlap between the arrival of segments of different flows at the sender access switch.

$$sInitialDelay_i = i \times \frac{sInterval}{N} + (ALTotalDelay_0 - ALTotalDelay_i) \tag{4.12}$$

4.2.5 sInterSegmentDelay

sInterSegmentDelay is added after the transmission of each segment in a single *sInterval*, to ensure there is no need for buffering at the bottleneck link due to arrival and departure rate mismatch. Equation 4.13 represents the formula for this parameter.

$$sInterSegmentDelay_i = \frac{DataSegSize}{btlBw_i} \tag{4.13}$$

4.3 Simulation Model

The simulation model is designed based on the analytical model presented in the previous section. The protocol involves three entities in the network:

- Controller
- Sender agent
- Receiver agent

The controller is capable of communicating with the sender agent by sending a new type of segment called *CTRL*. The remainder of this section describes the model implemented for each entity in the SDN Simulation Tool.

4.3.1 Controller

The main role of the controller in the proposed solution is to update the Congestion Control Parameters (*CCParams*) based on the analytical model, and initiate the next *sCycle* when a flow arrives or departs. Figure 4.8 represents the behavior of the controller upon arrival of a *SYN* or *FIN* segment.

When a *SYN* segment arrives, the controller updates the network information database. After that, it finds the best path for the new flow, and sends the flow setup messages to the corresponding switches. Then, it calculates the new *CCParams*, and generates the *CTRL* segments for all the flows connected to the corresponding access switch. After sending the *CTRL* segments to sender agents, the controller sends the *SYN* segment back to the access switch.

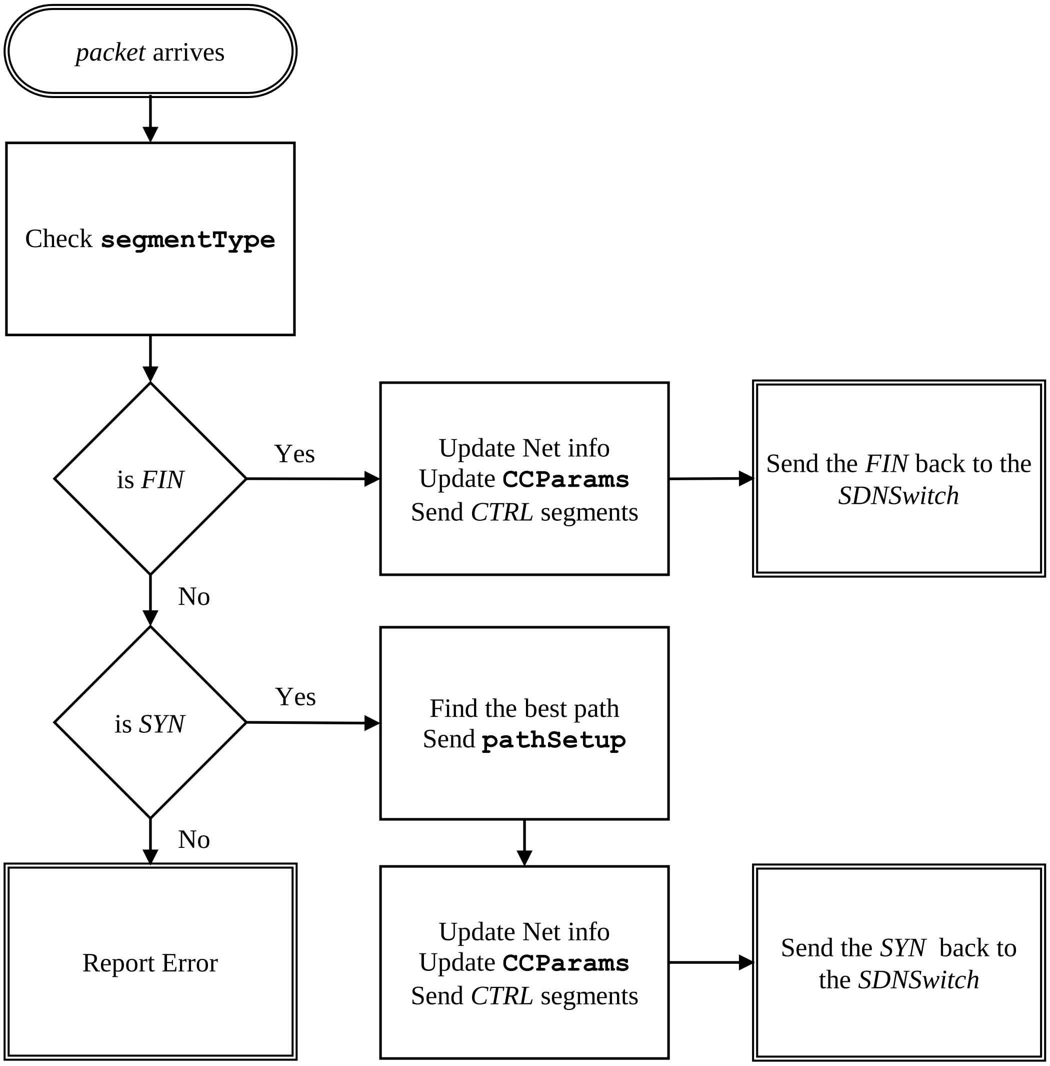


Figure 4.8: Controller flowchart for *recvPacket* method.

When a *FIN* segment arrives, the controller updates its information database about the network. Then, it calculates the new *CCParams*, generates the *CTRL* segments for each sender agent connected to the corresponding access switch, and sends them. Finally, it sends the *FIN* segment back to the access switch.

4.3.2 Sender

The role of the sender agent is to send its *Data* segments to the network based on the *CCParams* that it receives from the controller. To this end, the sender uses several types of timers. Table 4.2 represents the different types of timers used in the sender agent.

Table 4.2: Different types of timers in sender agent.

Timer Name	Definition
NextCycleTimer	Indicates the start of the new *sCycle*
InitialDelayTimer	Indicates the start of the first *sInterval*
IntervalTimer	Indicates the start of next *sInterval*
InterSegmentDelayTimer	Indicates the sending time of the next segment

As Figure 4.9 depicts, the sender agent can receive three types of segments. When a *CTRL* segment arrives from the controller, the sender stores the new *CCParams* without changing the current ones, and initiates a *NextCycleTimer*. When the *SYNACK* segment arrives, the connection is established, and the sender updates the information about acknowledged sequence numbers. When an *ACK* segment arrives, the sender checks to see if this acknowledgment is for the last *Data* segment of the flow. If true, it will send the *FIN* segment. Otherwise, it updates the information about acknowledged sequence numbers.

When the *NextCycleTimer* times out, the sender stops all the active timers associated with the current *sCycle*. Then, it uses the previously stored *CCParams* to update the current parameters. Finally, it initiates the new *sCycle* by activating an *InitialDelayTimer*.

When the *InitialDelayTimer* times out, the sender starts the first *sInterval*, which sends *Data* segments for the current sending window, using *InterSegmentDelayTimer*. It also starts an *IntervalTimer* for the next *sInterval* right after sending the first *Data* segment. When the *IntervalTimer* times out, the sender repeats the same routine as the *InitialDelayTimer* timeout.

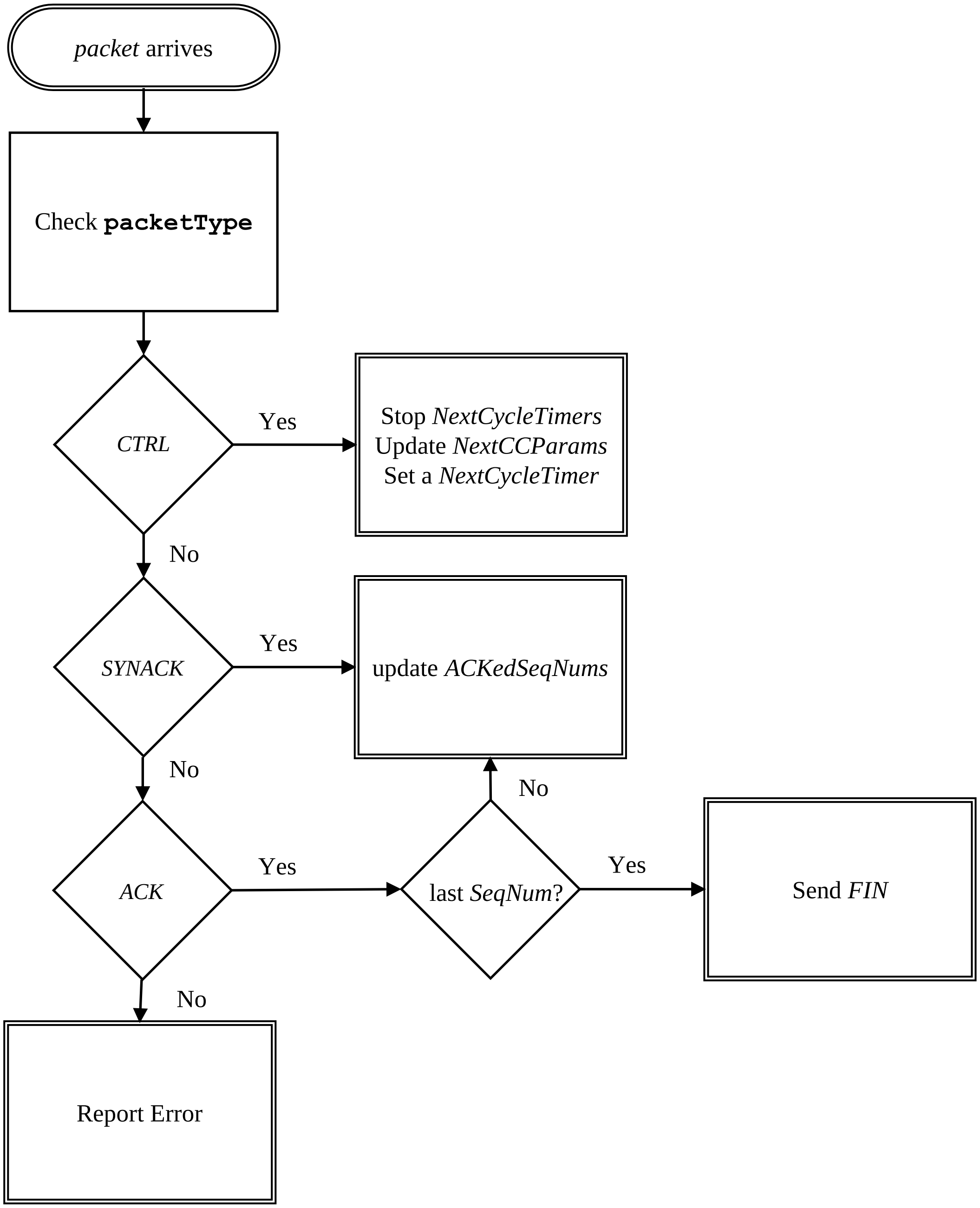


Figure 4.9: Sender agent flowchart for *recvPacket* method.

4.3.3 Receiver

The receiver agent has a simple acknowledgment generation mechanism. When a SYN segment arrives, it generates a $SYNACK$ segment and sends it back into the network. When a *Data* segment arrives, it generates an *ACK* segment with the same sequence number and sends it back to the sender agent.

4.4 Verification

To validate the implemented simulation model against the analytical model, two test scenarios have been designed. Both scenarios use a Dumbbell topology, and focus on the functionality of the controller and sender agents in the proposed protocol. The three congestion control parameters of the controller are $\alpha = 1$, $\beta = 1.5$, and $\gamma = 1.5$. The simulation results are shown in the form of sequence number graphs for each flow in the test.

4.4.1 Single Flow

The purpose of this test is to validate the functionality of the solution with a single flow in the network. The flow size is 50 *Data* segments, each of them 1000 *Bytes*, and arrives to the network at time zero. Figure 4.10 depicts the network topology used for this test scenario. Table 4.3 presents the link properties of the topology used for the test. All link bandwidths are set to $8(Gb/s)$. Hence, the transmission delays for *Data* (1000 *Bytes*) and *ACK* (40 *Bytes*) segments are $1(\mu s)$ and $0.04(\mu s)$ respectively.

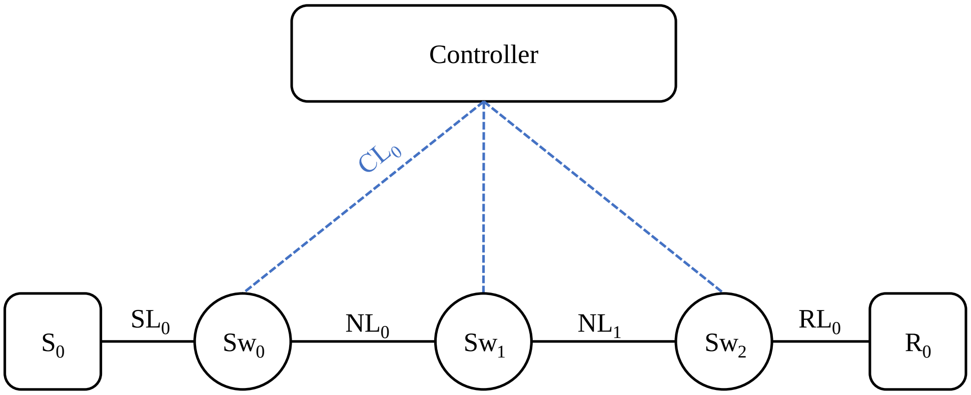


Figure 4.10: Network setup for single flow verification test.

Table 4.3: Link properties of the topology.

Name	Bandwidth(Gb/s)	Propagation Delay(μs)
SL_0	8	1
RL_0	8	1
CL_0	8	1
NL_0	8	1
NL_1	8	1
NL_2	8	1

Table 4.4 shows the calculated analytical values for the parameters, using equations in Section 4.2.

Table 4.4: Analytical values.

Parameter	Analytical Value
$sInterval$	18.24 (μs)
$sInterSegmentDelay$	1 (μs)
$SYNACKDelay$	14.68 (μs)
$CTRLDelay$	2.16 (μs)
$sInitialDelay$	0 (μs)
rtt	12.16 (μs)
$sWnd$	17 (segments)

Figure 4.11 depicts the simulation result in the form of a sequence number graph. The simulation values for *sInterval*, *sInterSegmentDelay*, *sInitialDelay*, *rtt*, and $sWnd$ match the analytical values.

The simulation values of $SYNACKDelay = 14.68$ (μs) and $CTRLDelay = 2.16$ (μs) are consistent with the analytical values that the controller calculates. However, the equations for calculating them are over-estimations of the actual values. Based on the simulation results from measurement on the sender agent, $SYNACKDelay = 8.4$ (μs) and $CTRLDelay = 2.12$ (μs), which shows that the analysis mentioned in Section 4.2 is correct.

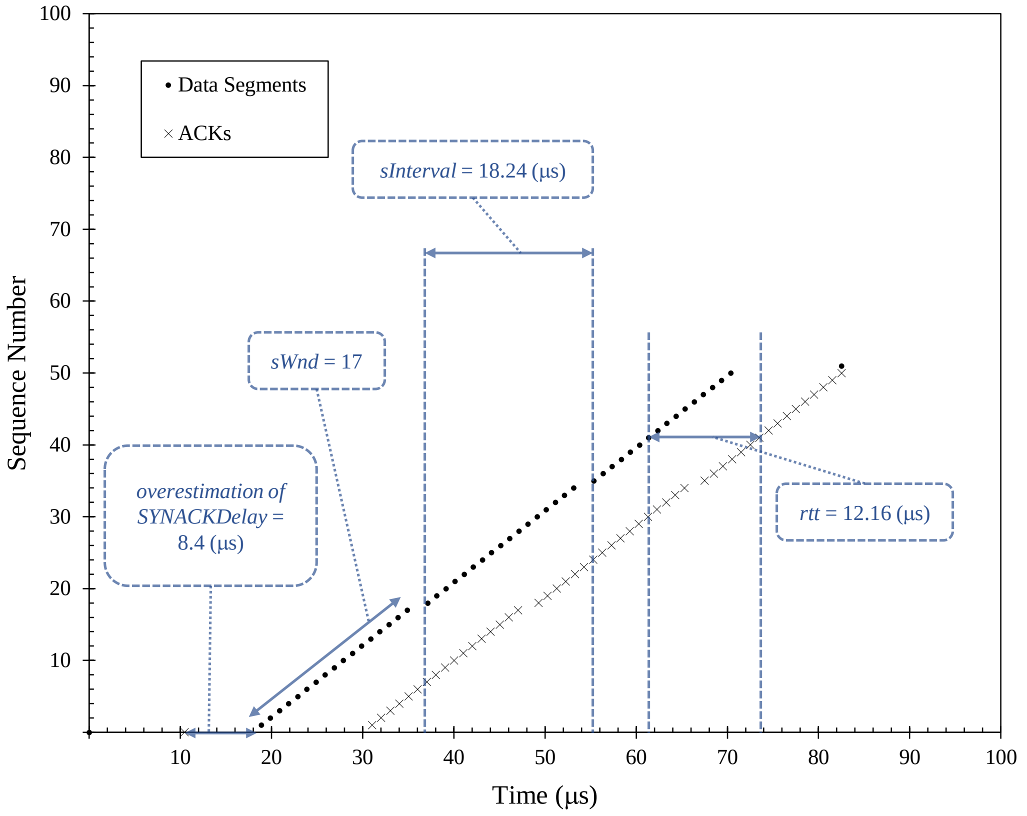


Figure 4.11: Verification test result for single flow scenario.

4.4.2 Multiple Flows

The purpose of this scenario is to test the functionality of the proposed solution when there is more than one flow in the network. Figure 4.12 depicts the topology used for the test. Table 4.5 presents the arrival time and the size for each flow in the test. Two cases of homogeneous and heterogeneous access links are tested with this setup.

Table 4.5: Traffic properties of the verification test for multiple flows.

Name	Arrival Time (μs)	Size (*Data* segments)
$flow_0$	0	150
$flow_1$	30	80
$flow_2$	100	20

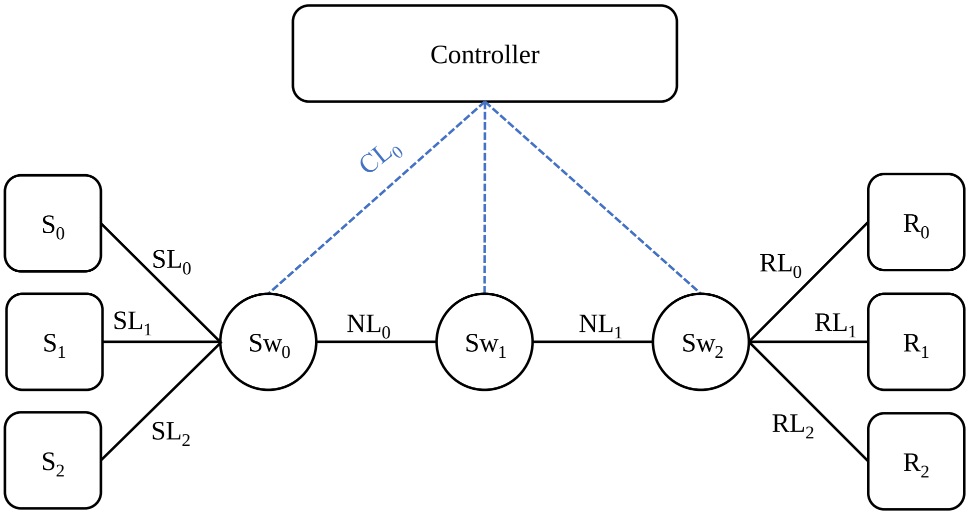


Figure 4.12: Network setup for multiple flow verification test.

Homogeneous Access Links

Table 4.6 presents the link properties of the homogeneous test. All link bandwidth are set to 8(Gb/s), so the transmission delay for *Data* and *ACK* segments are equal to 1(μs) and 0.04(μs), respectively.

Table 4.6: Link properties of the topology.

Name	Bandwidth(Gb/s)	Propagation Delay(μs)
SL_0	8	1
SL_1	8	1
SL_2	8	1
RL_0	8	1
RL_1	8	1
RL_2	8	1
CL_0	8	1
NL_0	8	1
NL_1	8	1

Figure 4.13 depicts the simulation result for $flow_0$, in the form of a sequence number graph. When $flow_0$ arrives at time zero, it is the only flow in the network. Hence, the controller updates the congestion control parameters and $sCycle_0$ starts with $flow_0$ utilizing the network with $sWnd = 17$ (*segments*).

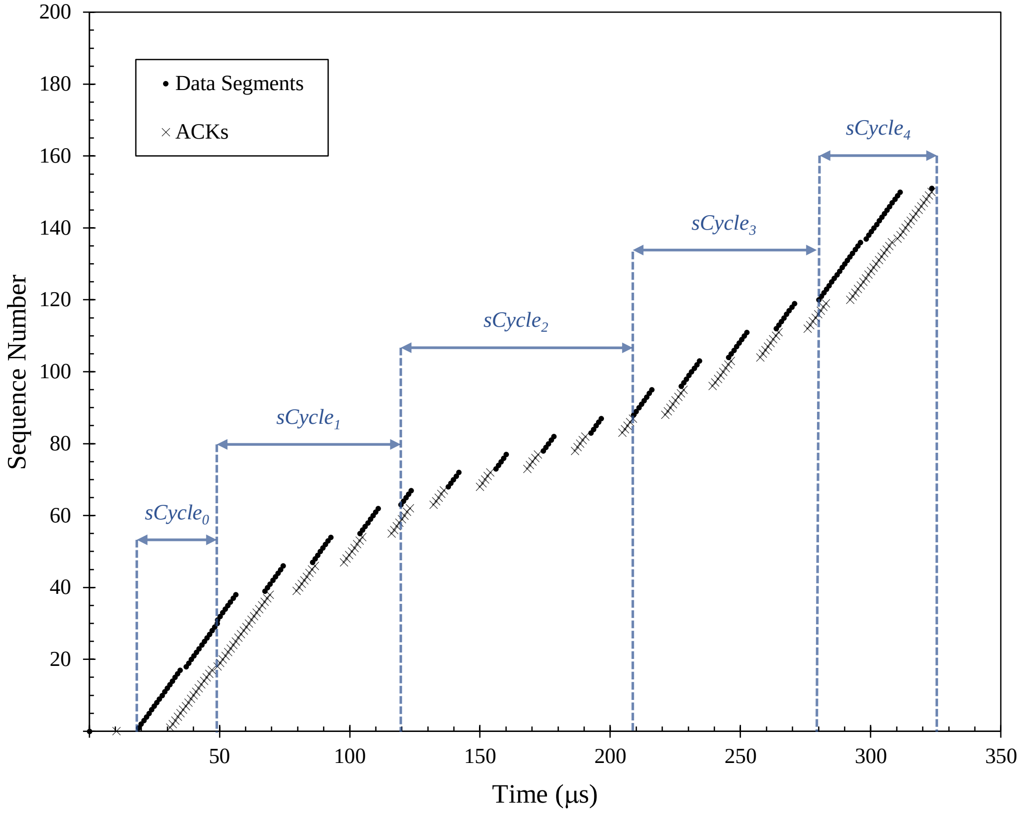


Figure 4.13: Verification test result for multiple flows with homogeneous access links ($flow_0$).

When the controller receives the SYN segment of $flow_1$, it informs both flows about the newly calculated congestion control parameters. $sCycle_1$ starts at $time = 49.16$ (μs), with $sWnd = 8$. $flow_0$ starts the new cycle without any initial delay. As Figure 4.14 shows, $flow_1$ waits for $sInitialDelay_1 = 9.12$ (μs), then sends segments.

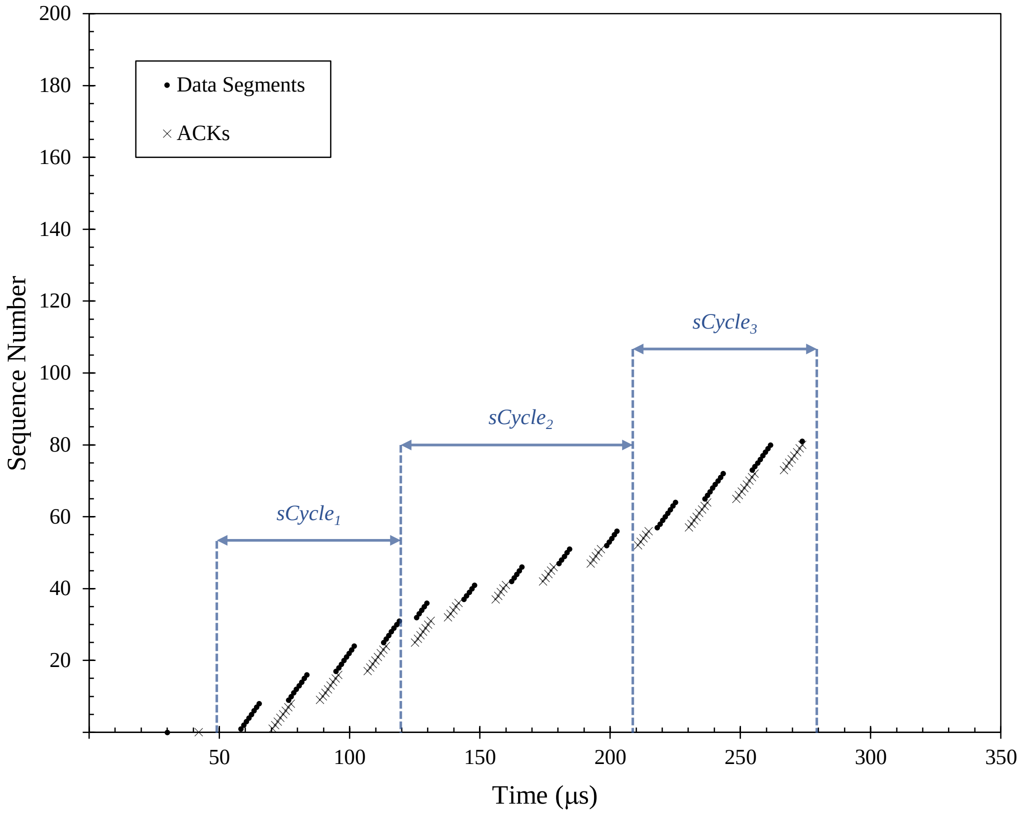


Figure 4.14: Verification test result for multiple flows with homogeneous access links ($flow_1$).

When $flow_2$ arrives, the controller notifies the flows, and $sCycle_2$ starts at $Time = 119.56(\mu s)$. The $sWnd$ is reduced to 5 ($segments$), and all flows start to send data according to their *sInitialDelay*, as seen in Figure 4.15.

Upon the departure of $flow_2$, the controller triggers $sCycle_3$, which starts at $Time = 208.96$ (μs). The $sWnd$ is set back to 8 ($segments$), and both remaining flows start sending data segments according to their $sInitialDelay$.

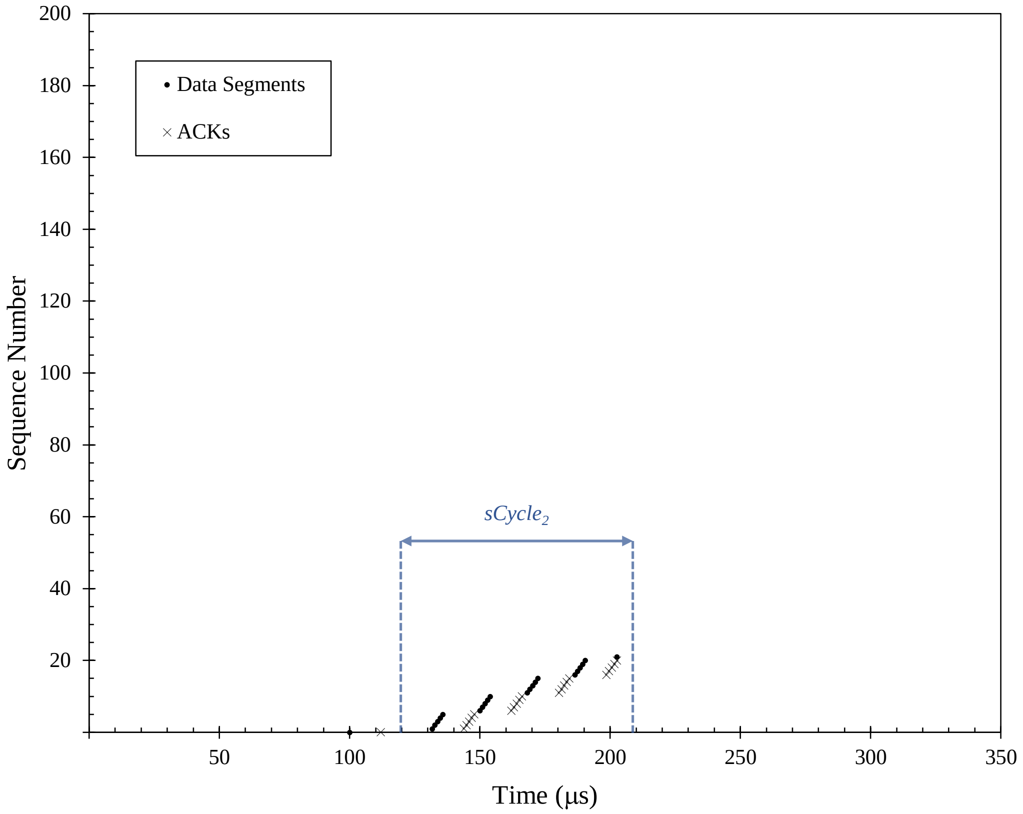


Figure 4.15: Verification test result for multiple flows with homogeneous access links ($flow_2$).

Finally, when $flow_1$ departs, the controller notifies $flow_0$, and $sCycle_4$ starts at $Time =$ 280 (μs). The $sWnd$ for this cycle is set to 17 (*segments*), and $flow_0$ sends its remaining $Data$ segments.

Heterogeneous Access Links

Table 4.7 presents the link properties of the heterogeneous test. In this case, the propagation delays for the sender access links are heterogeneous.

Figure 4.16 depicts the simulation result for $flow_0$, as a sequence number graph. The order of flow arrival and departure events is similar to the homogeneous test case. There are five sending cycles, and the $sWnd$ and *sInitialDelay* for each flow is set by the controller in each one of them.

Table 4.7: Link properties of the topology.

Name	Bandwidth(Gb/s)	Propagation Delay(μs)
SL_0	8	2
SL_1	8	4
SL_2	8	1
RL_0	8	1
RL_1	8	1
RL_2	8	1
CL_0	8	1
NL_0	8	1
NL_1	8	1

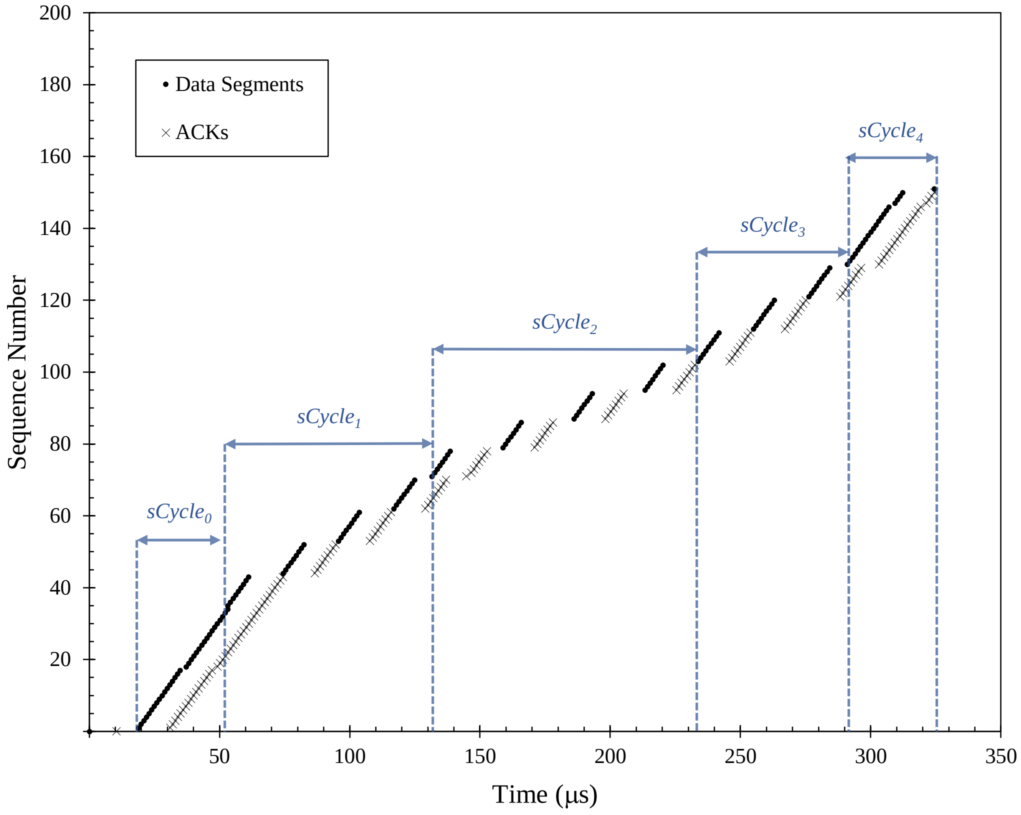


Figure 4.16: Verification test result for multiple flows with heterogeneous access links ($flow_0$).

4.5 Summary

This chapter discussed the proposed solution. Specifically, it provided an overview of the basic idea of the solution and the problems that it solves. Also, it presented the analytical and simulation models of the solution. Finally, it described the verification test scenarios, and presented the results for them. The next chapter discusses the simulation evaluation of the proposed solution.

Chapter 5

Simulation Evaluation

This chapter presents the simulation evaluation of the proposed solution. Section 5.1 gives an overview of the performance metrics and network topology used for the evaluation. Section 5.2 presents the simulation results and analysis of multiple experiments. Finally, Section 5.3 summarizes the chapter.

5.1 Experimental Design

The proposed solution is evaluated by simulation studies, using the developed SDN Simulation Tool. The purpose of the evaluation is to study the behavior of the proposed protocol in multiple scenarios.

5.1.1 Performance Metrics

Several metrics are used in the simulation evaluation of the proposed solution. The performance metrics are:

Flow Setup Delay

Flow Setup Delay (FSD) is the elapsed time between the generation of the *SYN* segment, and the time the first *Data* segment is sent.

Flow Completion Time

Flow Completion Time (FCT) is the elapsed time between the generation of the *SYN* and the *FIN* segments for a given flow.

Flow Throughput

As Equation 5.1 represents, *Flow Throughput* is defined as the ratio of the total transmission time of the flow, to the total lifetime of it. The unit of the metric is percentage (%):

$$Troughput = 100 \times \frac{totalTransmissionTime}{(FlowCompletionTime - FlowArrivalTime)} \tag{5.1}$$

Link Utilization

Equation 5.2 shows the formula for *Link Utilization*. It is defined as amount of acknowledged data (*ACKedSegments*) transferred over a link during the time it is active (*totalUpTime*).

$$LinkUtilization = \frac{ACKedSegments \times segmentSize}{linkBandwidth \times totalUpTime} \tag{5.2}$$

Fairness

Equation 5.3 represents Chiu and Jain's *fairness index* [14].

$$f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} (x_i)^2} \tag{5.3}$$

where x_i is the throughput achieved by flow i, and n is the total number of flows. The value of f ranges between $\frac{1}{n}$ and 1. $f = 1$ is the highest fairness achievable, and it means all flows have identical throughput. When f is close to $\dfrac{1}{n}$, it means one flow is monopolizing the network throughput.

Average Queue Length

Equation 5.4 shows the formula for *Average Queue Length* (AvgQL) of a given link in the network.

$$f = \frac{\sum_{i=1}^{n-1} l_i \times (t_{i+1} - t_i)}{t_n - t_0} \tag{5.4}$$

where l_i is the instantaneous queue length at time t_i, and n is the number of entries in the queue length times series.

Maximum Queue Length

Maximum Queue Length (MaxQL) metric shows the highest measured buffer occupancy during the simulation for a given link.

5.1.2 Simulation Setup

Figure 5.1 depicts the network topology used for simulation evaluation of the proposed solution. Table 5.1 shows the default link properties. The link properties are similar to a *Local Area Network* (LAN). All bandwidths and propagation delays are set to 10 (Gb/s) and 5 (μs), respectively. Also, Table 5.2 presents the default traffic specifications. All simulation studies use these settings, unless mentioned otherwise.

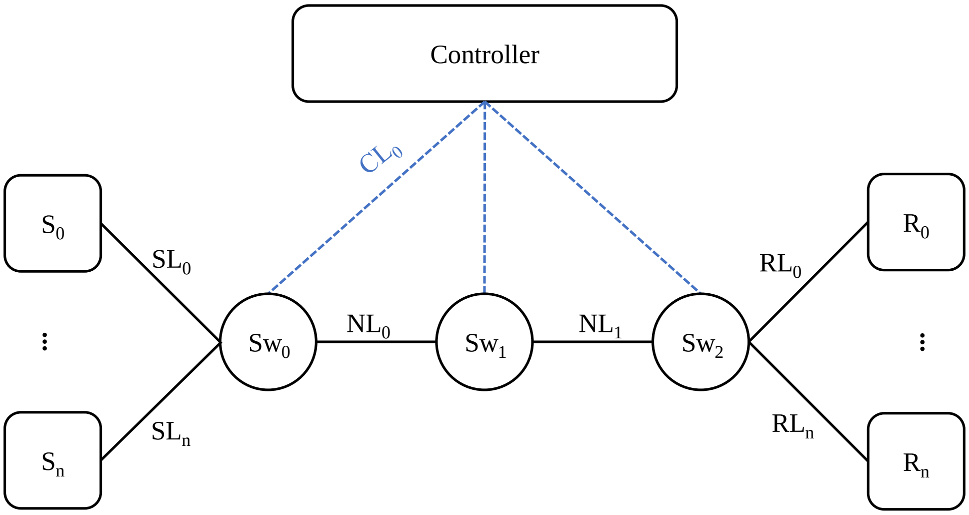


Figure 5.1: Default network topology for simulation experiments.

Table 5.1: Default link properties.

Parameters	Bandwidth(Gb/s)	Propagation Delay(μs)
CLTransDelays	10	5
NLTransDelays	10	5
SLTransDelays	10	5
RLTransDelays	10	5

Table 5.2: Default traffic specifications.

Parameter	Value
Flow Inter-Arrival Time (*Int-AT*)	0.5 (ms)
Flow Sizes (S)	1 MB
Number of Flows (N)	10

5.2 Simulation Studies

The purpose of these simulation experiments is to study the behavior of the proposed solution. Each section explains the scenario studied, and then provides the results along with their analysis.

5.2.1 Flow Inter-Arrival Times

The purpose of this study is to investigate the effect of *Int-AT* on the performance of the proposed solution. Table 5.3 presents the traffic specification for the study. *Int-AT* is the main factor, and it is identical for all the flows in a single simulation run. To see the effect of size of the flows at the same time, S is chosen as the secondary factor, and all the flows in the simulation runs have identical sizes.

Table 5.3: Traffic specifications for *Flow Inter-Arrival Times* study.

Parameter	Value	Comments
Int-AT	0 to 2 (ms)	step size = 0.1 (ms)
S	100 KB, 500 KB, 1 MB	NA

To help understanding the behavior of the protocol in this study, we define the following parameters:

- *Degree of Overlapping* (d_o): Shows the number of flows overlapping in a period of time
- *Single Flow Completion Time* (*singleFCT*): The completion time of a flow when it is alone in the network.

Varying *Int-AT* in the study results in three special cases in terms of overlapping of the flows. Figure 5.2 shows *full-overlapping*, *half-overlapping*, and *non-overlapping* cases for flow lifetimes. When *Int-AT*$= 0$, all flows arrive at the same time, and share the same *sCycle*. When *Int-AT*$= \dfrac{singleFCT}{2}$, exactly two flows share the same *sCycle* most of the time. When *Int-AT* $\geq$ *singleFCT*, each flow sends all its data throughout a single *sCycle* shared by no other flow.

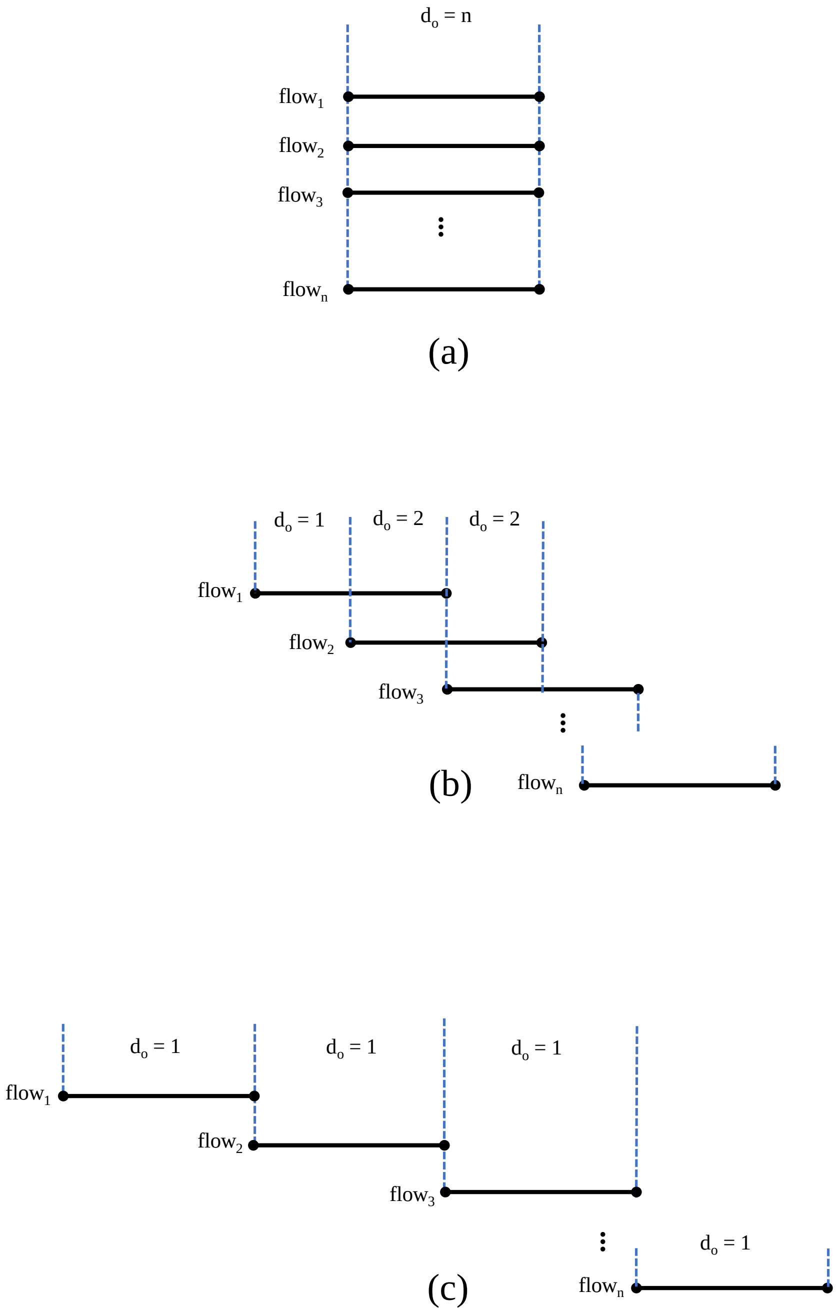


Figure 5.2: Flow lifetime overlapping cases: (a) full-overlapping. (b) half-overlapping. (c) non-overlapping.

Figure 5.3 depicts two intermediate cases between the thresholds discussed in Figure 5.2. In both cases, when *Int-AT* increases, the average degree of overlapping decreases. When

$0 < \textit{Int-AT} < \frac{singleFCT}{2}$, the average degree of overlapping is high (i.e., greater than two). When $\frac{singleFCT}{2} < \textit{Int-AT} < singleFCT$, the average degree of overlapping is low (i.e., less than two). Also, increasing the size of the flows increases the $singleFCT$. Hence, both Average FCT and the threshold for entering non-overlapping state increases.

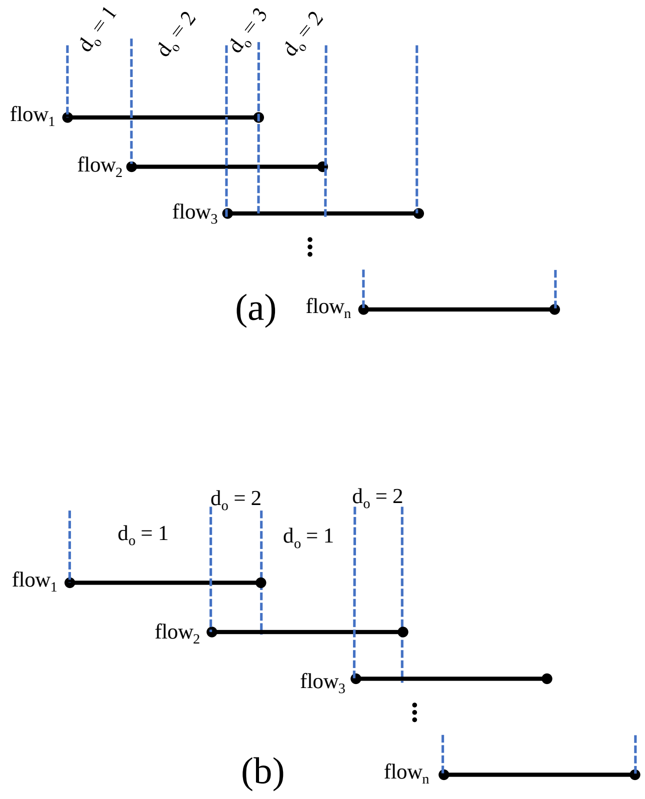


Figure 5.3: Flow lifetime overlapping cases: (a) high-overlapping. (b) low-overlapping.

In the simulation experiments that follow, the *Int-AT* is used as the main factor to vary the structural characteristics of the network workload, as explained above. An understanding of this workload pattern is essential to interpreting the results properly.

Figure 5.4 depicts the Average FCT for three different flow sizes. Considering flow size of 1 (MB), when $\textit{Int-AT} = 0$ the flows are in *full-overlapping* state. Hence, the Average FCT is at its peak. As the *Int-AT* increases to $singleFCT = 0.9$ (μs), the degree of overlapping decreases. Hence, the Average FCT decreases too. When $\textit{Int-AT} \geq singleFCT$, the flows become *non-overlapping* and Average FCT is constant.

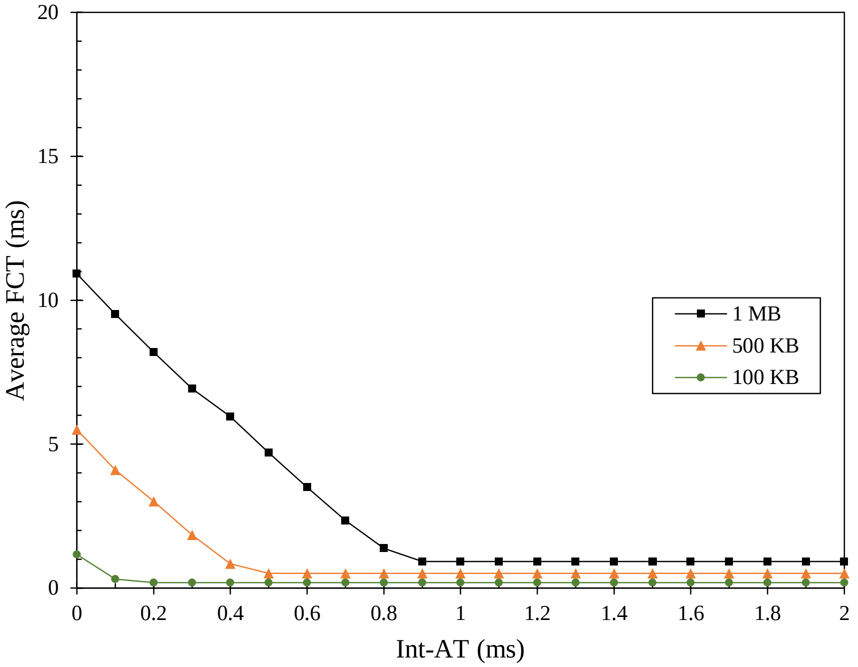


Figure 5.4: Average FCT for *Flow Inter-Arrival Time* study.

Figure 5.5 depicts the Fairness Index for different flow sizes. Considering the flow size of 1 (MB), when *Int-AT*$= 0$, all flows are in *full-overlapping* state. In this situation, all of the *SYN* segments arrive to the controller within a small period of time. Hence, all flows start sending at the same *sCycle*, which means all of them get an identical share of each *sInterval*, and Fairness Index is equal to one.

When $0 <$*Int-AT*< 0.45 the flows are in *high-overlapping* state, which means the controller receives the *SYN* segments within a larger period of time compared to the previous state. Hence, the earlier flows have to wait for a larger amount of time to start sending compared to the later ones. As a results, the *Fairness Index* decreases.

When $0.45 <$*Int-AT*< 0.9, the flows are in *low-overlapping* state, which means the controller receives the *SYN* segments within a much larger period of time compared to the previous state. Hence, the earlier flows get the chance to start sending segments while the controller receives the *SYN* segments of the new flows and triggers the new *sCycle*. As a results, the *Fairness Index* increases.

When *Int-AT* ≥ 0.9, flows are in *non-overlapping* state, and they have identical Flow Throughput. Hence the Fairness Index is constant and equal to one.

Also, the size of flows does not affect the period of time the *SYN* segments arrive to the controller. Hence, the Fairness Index is the same for different flow sizes.

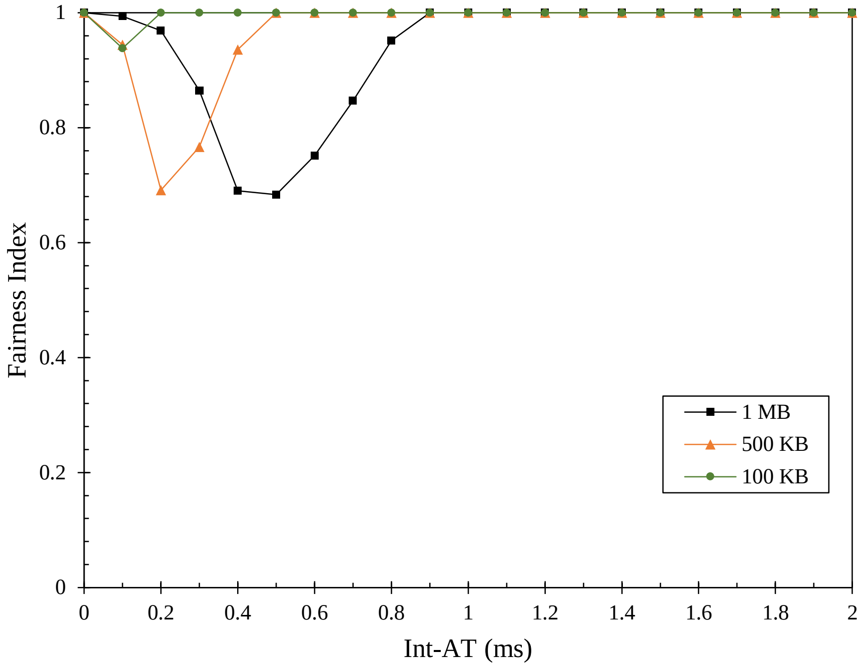


Figure 5.5: Fairness Index for *Flow Inter-Arrival Time* study.

Figure 5.6 shows the Average FSD for different flow sizes. FSD is a function of *DelayToNextCycle* and *sInitialDelay*. Considering flow size of 1 (*MB*), when the flows are *full-overlapping*, the value of *sInitialDelay* is high. Hence, the Average FSD is high too. However, in this scenario the controller receives all the *SYN* segments within a short period of time. As a result, the delay to start the first *sCycle* is not high.

When *Int-AT* ≤ 0.45, the flows are in *high-overlapping* state, and the value of *sInitialDelay* is still high. Also, because the controller receives all the *SYN* segment within a larger period of time compared to *full-overlapping* case, the Average FSD increases.

When *Int-AT* ≥ 0.45, the flows enter *low-overlapping* state, and the value of *sInitialDelay* decreases. Hence, the Average FSD decreases too.

When *Int-AT* ≥ 0.9, the flows are in *non-overlapping* state, and the Average FSD is constant and minimal. Increasing the size of flows does not affect the *DelayToNextCycle* or *sInitialDelay*, though it increases the thresholds for the overlapping cases. Hence, the Average FSD does not change.

Figure 5.7 and Figure 5.8 represent the Average Flow Throughput and the Bottleneck Link Utilization, respectively. Similar to FSD, both Flow Throughput and the Bottleneck Link Utilization are functions of *DelayToNextCycle* and *sInitialDelay*. Hence, the same analysis applies for them too.

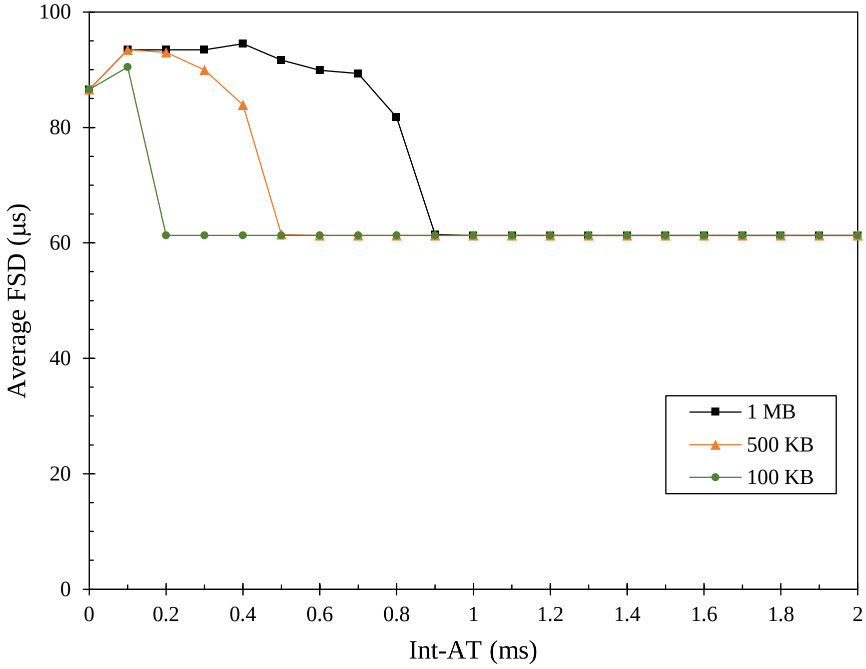


Figure 5.6: Average FSD for *Flow Inter-Arrival Time* study.

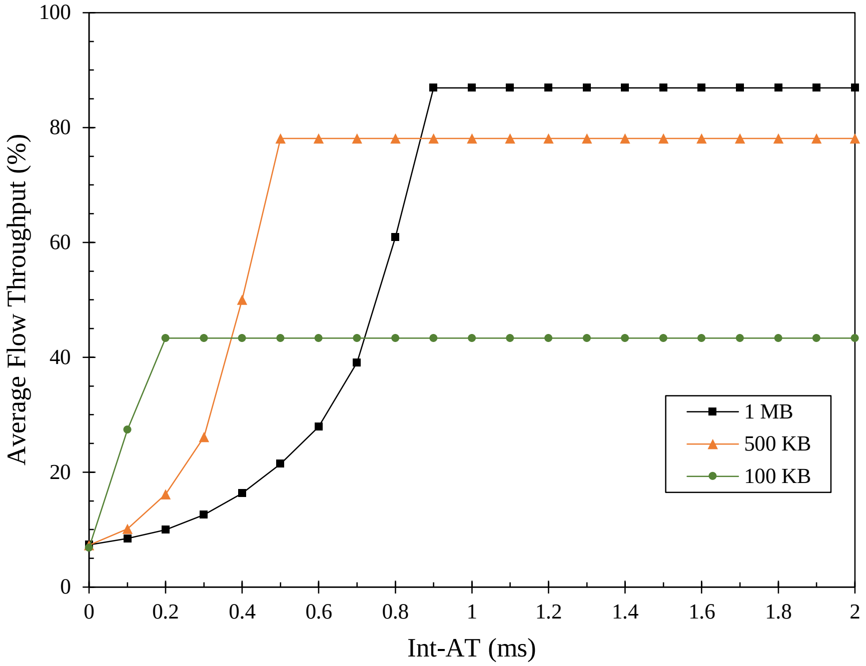


Figure 5.7: Average Flow Throughput for *Flow Inter-Arrival Time* study.

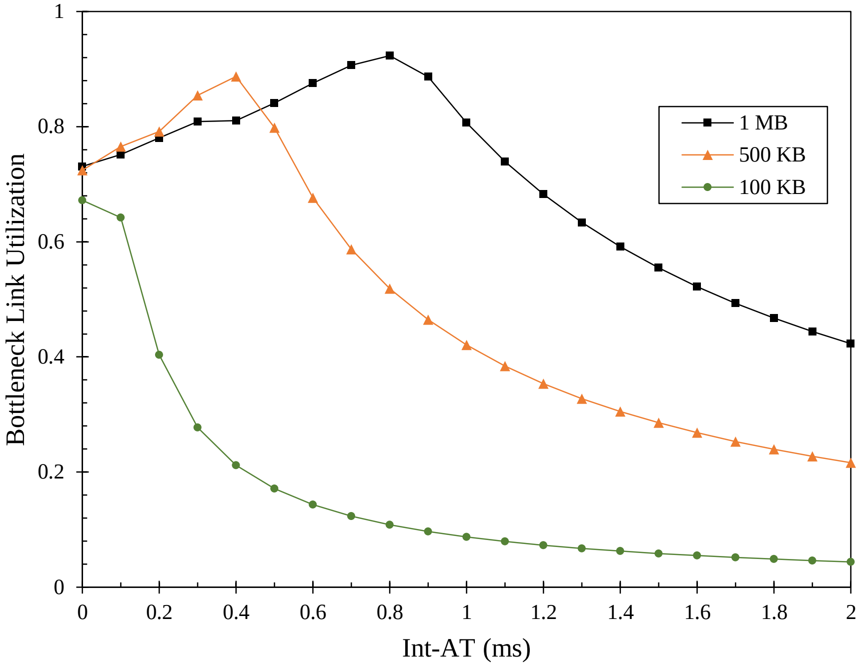


Figure 5.8: Bottleneck Link Utilization for *Flow Inter-Arrival Time* study.

5.2.2 Number of Flows

This study examines the effect of N on the performance of the proposed protocol. Table 5.4 presents the traffic specification used for this study.

According to the analytical model of the proposed solution, when the number of flows passes the point that $sWnd < 1$, the protocol will allow all flows to send at least one segment per each *sInterval*. As a result, a persistent queue starts to form. In this experiment, we set the size of the bottleneck link buffer to infinity to see the behavior of the protocol when it reaches the breaking point.

Table 5.4: Traffic specifications for N study.

Parameter	Value	Comments
N	1 to 100	step size = 5
S	100 KB, 500 KB, 1 MB	NA
Int-AT	0.1 (ms)	To avoid non-overlapping flows

Figure 5.9 depicts the Average FCT. When $1 \leq N \leq 20$, as N increases, more flows share the same *sCycle*, and $sWnd$ decreases drastically. Hence, Average FCT increases, and Average Flow Throughput decreases (see Figure 5.11). As Figure 5.12 shows, because of the increase in *sInitialDelay*, Bottleneck Link Utilization decreases in this range.

For $20 < N < 60$, since the controller divides the *sInterval* by N to determine the $sWnd$, the changes in $sWnd$ is modest. Hence, Average FCT and Average Flow Throughput do not change that much either.

When $N > 60$, even if all flows use $sWnd = 1$, the number of segments in each *sInterval* exceeds the capacity of the bottleneck link. The Bottleneck AvgQL shown in Figure 5.13 shows that a persistent queue starts to form at the bottleneck link buffer at this point. As a result, Average FCT starts to increase drastically.

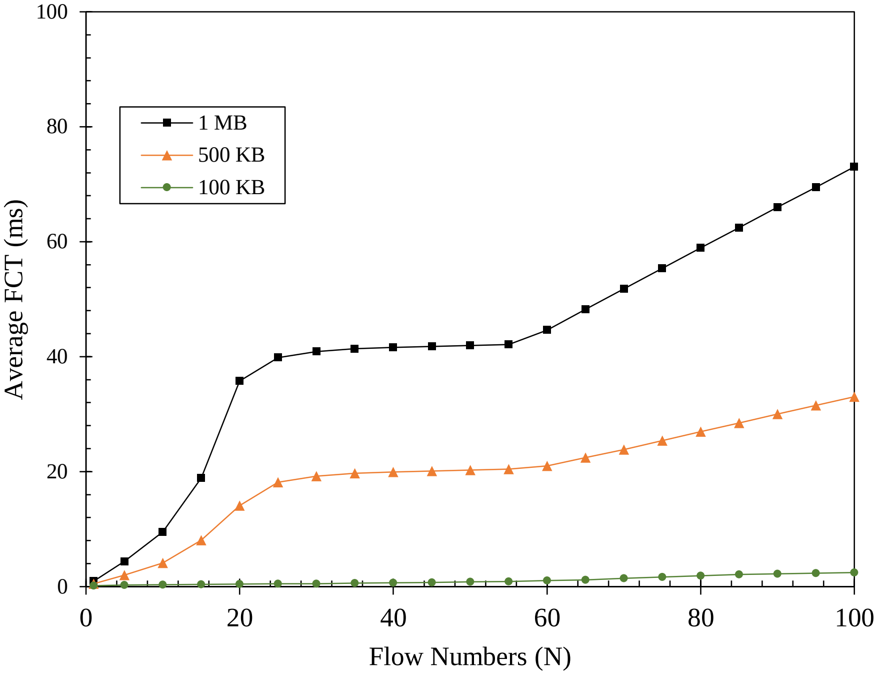


Figure 5.9: Average FCT for *Number of Flows* study.

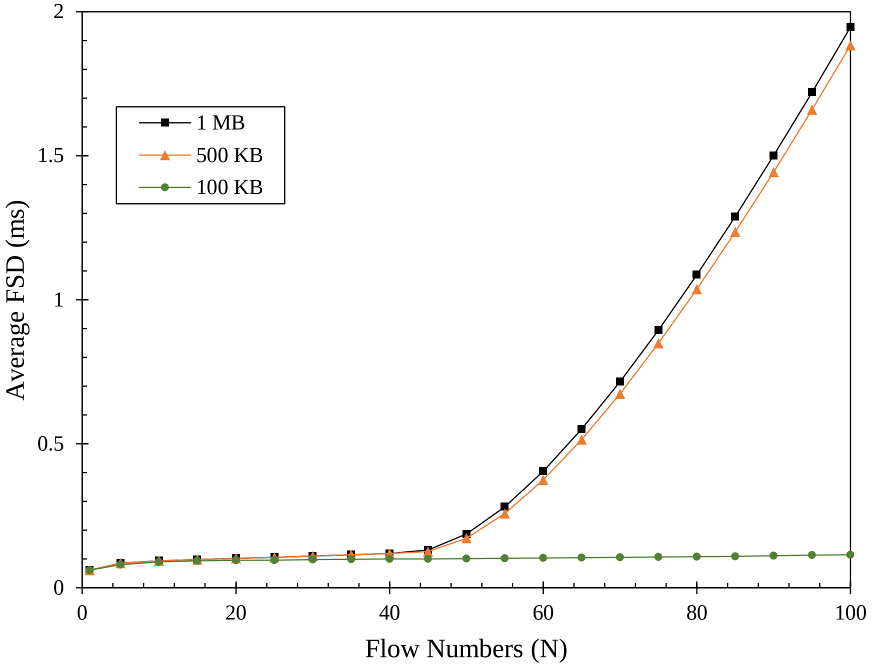


Figure 5.10: Average FSD for *Number of Flows* study.

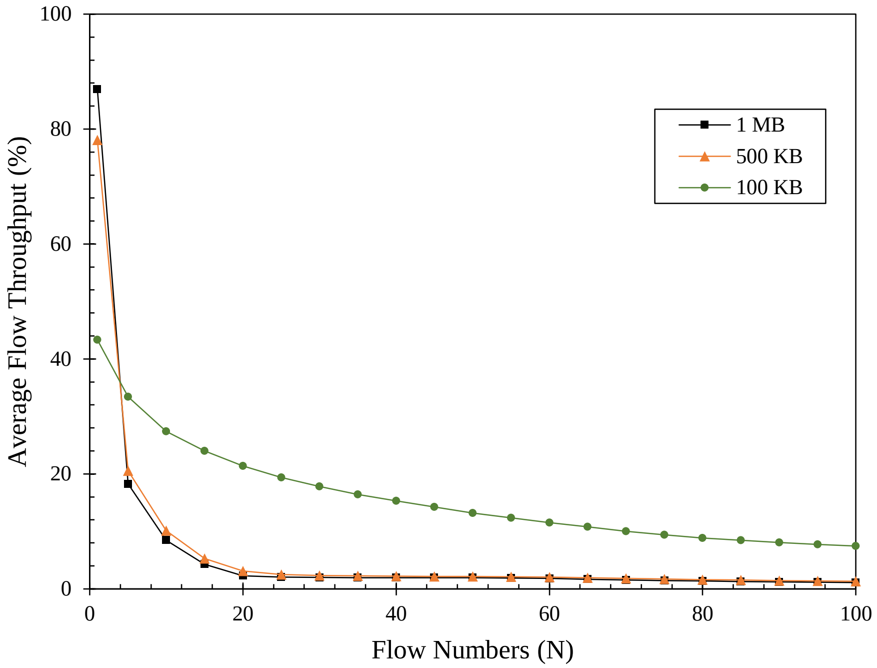


Figure 5.11: Average Flow Throughput for *Number of Flows* study.

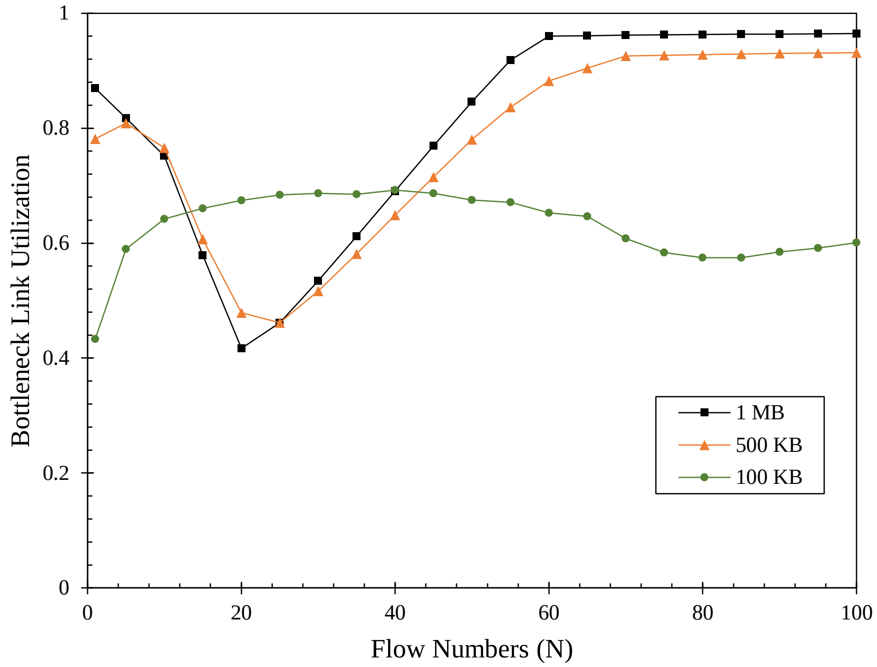


Figure 5.12: Bottleneck Link Utilization for *Number of Flows* study.

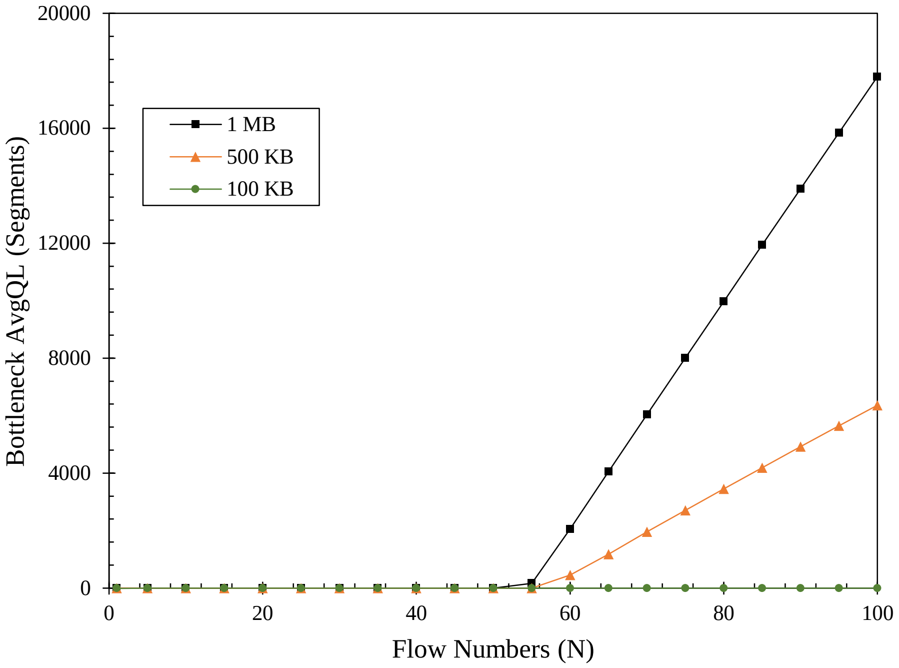


Figure 5.13: Bottleneck AvgQL for *Number of Flows* study.

5.2.3 In-cast Problem

This scenario aims to study the TCP in-cast problem [13], and investigate the performance of the proposed solution mentioned in the *Number of Flows* study. The previous study showed that the protocol performance degrades drastically when it reaches the breaking point. The number of flows (connected to the same access switch) needed to break the solution was 50. This means the protocol is not quite scalable for a typical LAN network. However, according to the analytical model for the solution, increasing the value of γ increases the length of each *sInterval*. Hence, the $sWnd$ increases too, and the breaking point should happen for a greater number of flows. For this reason, γ is considered as the secondary factor of this study to see if it can improve the scalability of the protocol.

Figure 5.14, Figure 5.15, and Figure 5.16 show the Bottleneck AvgQL, Bottleneck Link Utilization, and Average FCT, respectively.

When $\gamma = 1$, and N exceeds the maximum number of segments the bottleneck can

Table 5.5: Traffic specifications for *In-cast Problem* study.

Parameter	Value	Comments
N	1 to 100	step size = 5
γ	1, 1.5, 2	NA
S	1 MB	NA
Int-AT	0	All flows arrive at the same time

transmit in *sInterval*, *Average FCT* and *Bottleneck AvgQL* increase very quickly. Increasing γ increases the *sInterval*. Hence, the upper bound for N increases too. As Figure 5.16 depicts, for $\gamma = 1$ the maximum value for N without a persistent queue at the bottleneck link is 55. Using $\gamma = 1.5$, this value extends to 80. For $\gamma = 2$, the proposed protocol can handle 100 flows arriving at the same time to the access switch. The results of this experiment shows that by tuning γ the scalability of the protocol improves.

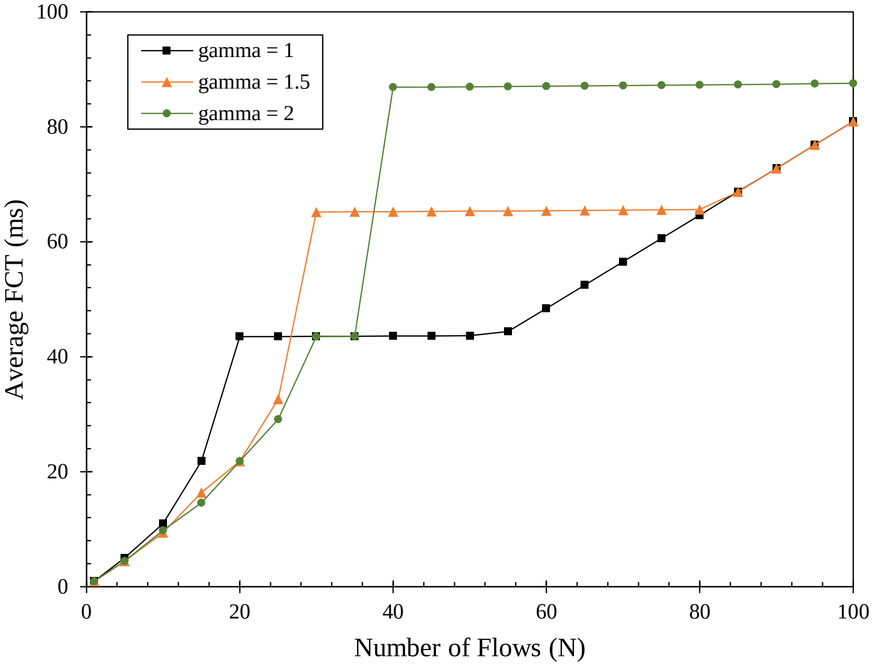


Figure 5.14: Average FCT for *In-cast Problem* study.

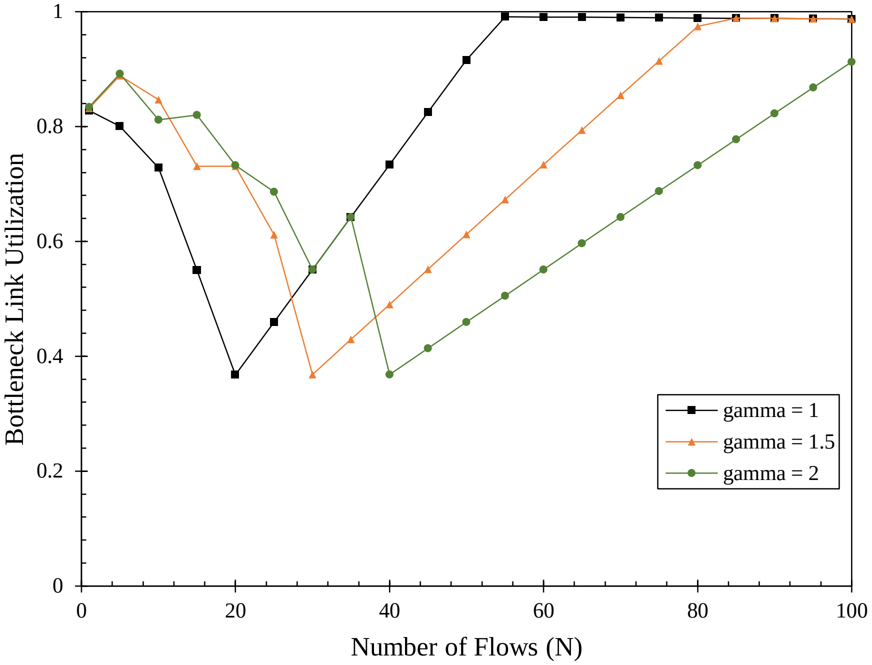


Figure 5.15: Bottleneck Link Utilization for *In-cast Problem* study.

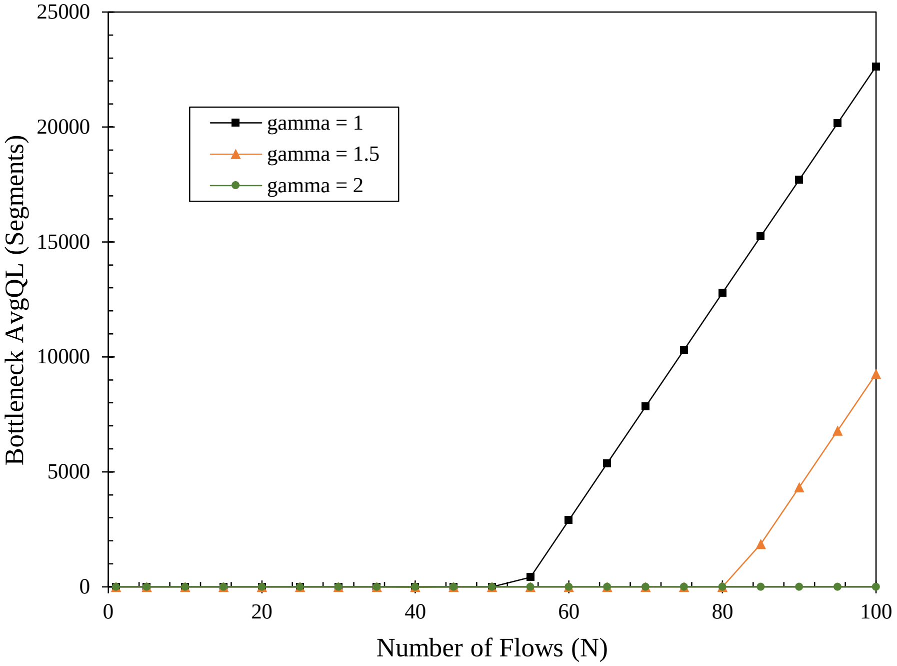


Figure 5.16: Bottleneck AvgQL for *In-cast Problem* study.

5.2.4 Gamma

The purpose of this study is to investigate the effect of varying γ on the performance of the protocol. Table 5.6 shows the factors, with their levels, used in this study.

Table 5.6: The first and the second factors for the *Gamma* study.

Parameter	Value	Comments
γ	1 to 200	step size = 1
Int-AT	0, 0.5, 1 (ms)	NA

The parameter γ is defined to over-estimate the *sInterval*. Increasing γ increases the *sInterval*. Hence, the *sWnd* for each flow in the *sCycle* increases too. When the *Int-AT* is equal to 1 (ms), none of the flows overlap with each other. Hence, changing the value of γ does not affect any of the performance metrics.

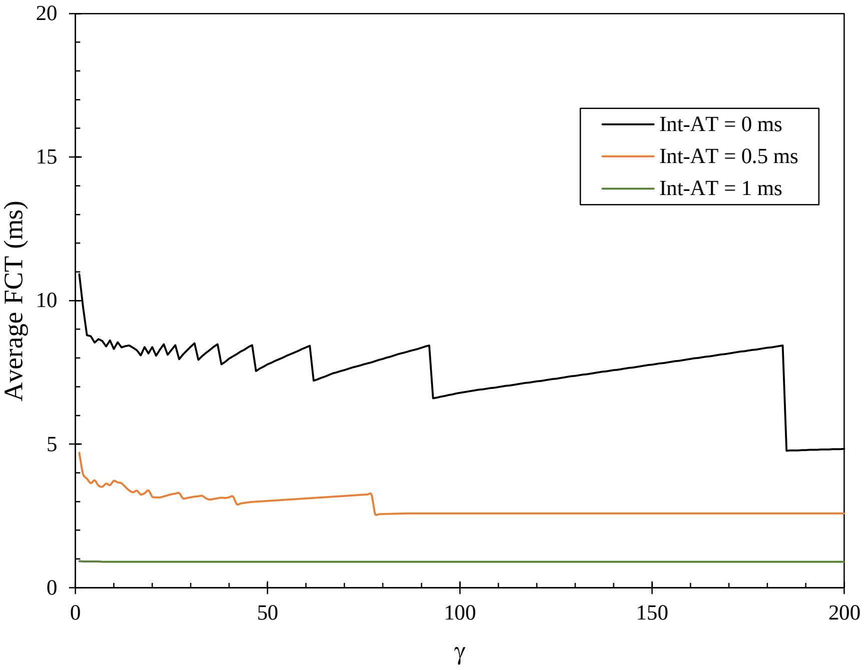


Figure 5.17: Average FCT for *Gamma* study.

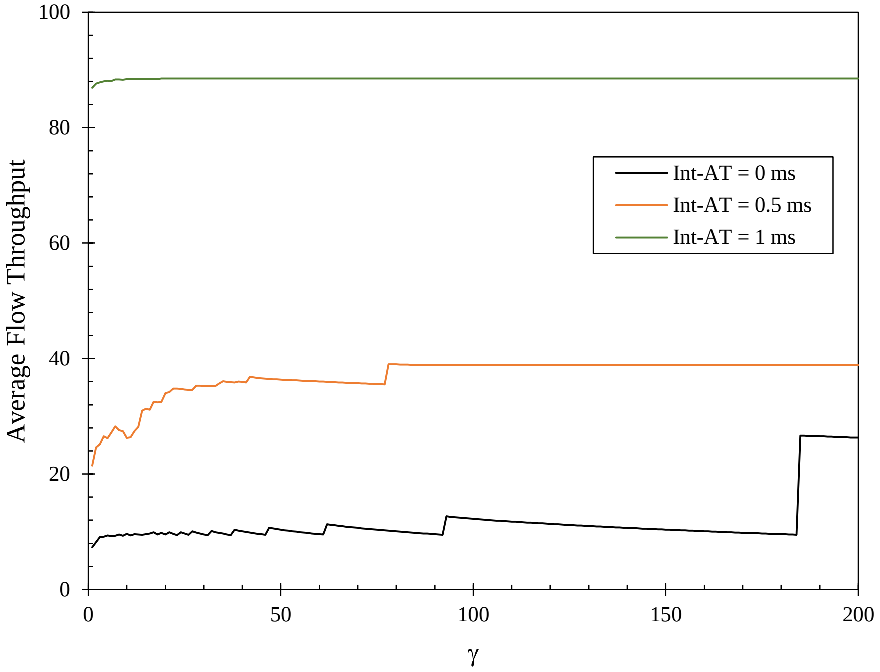


Figure 5.18: Average Flow Throughput for *Gamma* study.

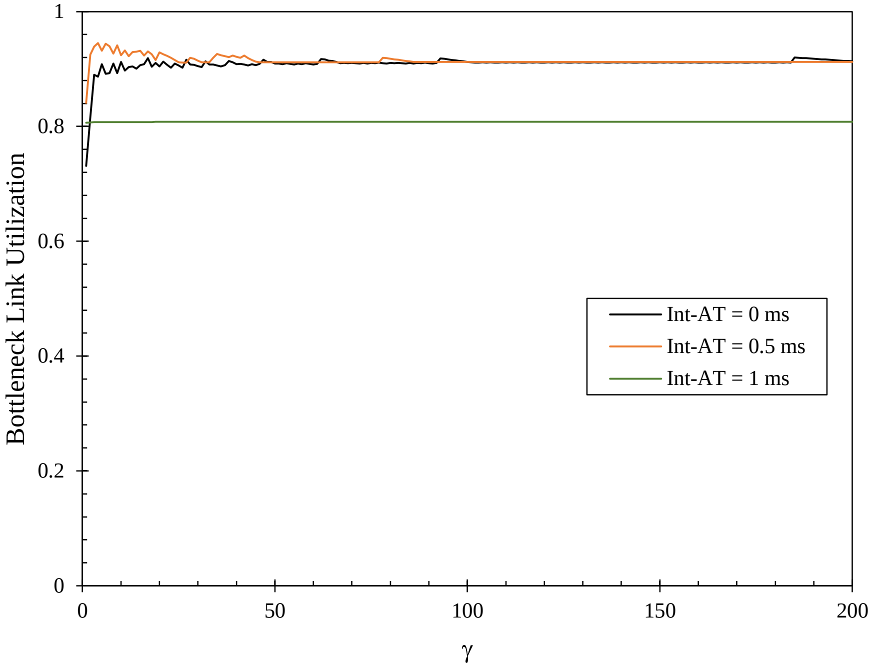


Figure 5.19: Bottleneck Link Utilization for *Gamma* study.

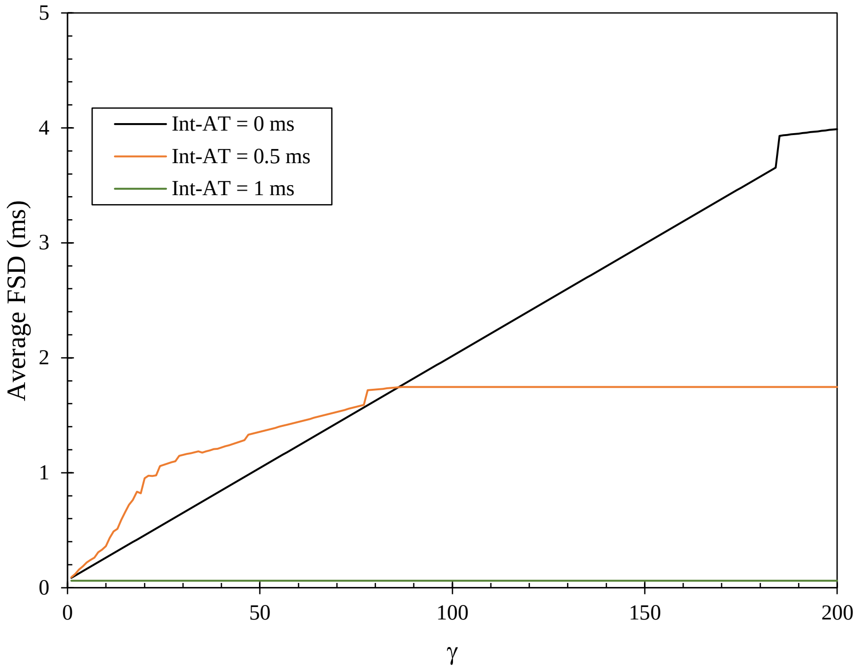


Figure 5.20: Average FSD for *Gamma* Study.

As Figure 5.17, Figure 5.18, and Figure 5.19 depict, when the *Int-AT* is low, increasing γ improves Average FCT, Average Flow Throughput, and Bottleneck Link Utilization, respectively. However, as Figure 5.20 shows, increasing γ increases Average FSD. The lower the *Int-AT* is, the more linear this increment is.

In summary, the study suggests that increasing γ to some extent, when *Int-AT* is low, improves the performance of the proposed solution.

5.3 Summary

This chapter presented the simulation evaluation of the proposed solution. In particular, it provided an overview of the performance metrics and simulation setup. It also presented the simulation studies along with their results and analysis. The next chapter concludes the thesis and discusses future work.

Chapter 6

Conclusions and Future Work

This chapter provides a summary of the thesis, conclusions, and future work. Section 6.1 summarizes the thesis. Section 6.2 discusses the conclusions of the research work. Section 6.3 suggests some directions for future work.

6.1 Thesis Summary

A summary of the thesis is as follows:

- Chapter 2 provided the background knowledge and related work. It described TCP along with its reliability and congestion control mechanisms. Also, it provided an overview about SDN, and explained the related work for the thesis.

- Chapter 3 discussed the SDN Simulation Tool. It described the components of the simulator in detail. Also, it explained the verification test along with its results.

- Chapter 4 introduced the proposed solution. It described the analytical and simulation model of the protocol. Moreover, it discussed the verification test for implementation of the simulation model.

- Chapter 5 presented simulation experiments and results. It discussed the experimental methodology. Also, it explained simulation experiments along with their results.

6.2 Conclusions

The thesis has the following objectives:

- To design a centralized congestion control scheme for SDN: We used a simplified network to create the analytical model for the solution. The analytical model defined equations for newly introduced congestion control parameters by doing a fine-grained analysis of the network.

- To develop an SDN simulation tool: The simulator models the transport and network layers of SDN. Moreover, the integrated framework facilitates simulation experiments by providing traffic and topology generators. Also, it formats the simulation output in the form of tables and graphs for desired performance metrics. The tool is developed in Java and benefits from a modular object-oriented design, which enhances its reusability and extensibility.

- To evaluate the proposed solution: We used the analytical model of the proposed solution to create the simulation model for our SDN Simulation Tool. Then the implementation of the simulation model was verified by multiple verification test scenarios. Also, multiple simulation experiments are designed and executed to investigate the behavior of the proposed solution in different scenarios.

The simulation experiments focused on a simple LAN network. The results and analysis of the experiments provided the following insights:

- The proposed protocol maintains minimal queue length at the bottleneck link.

- The proposed protocol maintains a high level of fairness in different circumstances. Specifically the *Int-AT* study showed that the lowest degradation of Fairness Index was 30% for a specific set of values for *Int-AT*.

- Average Flow Completion time increases linearly (before the breaking point) for most of the situations.

- The tuning parameter γ increases the scalability of the protocol. In particular, for *In-cast Problem* experiment, increasing the value of γ by 50% increased the number of flows needed for reaching the breaking point by 60%.

6.3 Future Work

The possible future directions for this thesis can be categorized as follows:

- Evaluation of the protocol: The simulation experiments done in this thesis can be extended to more complicated scenarios. In particular, using a more complex network topology will shed more light on the behavior of the proposed protocol. Also, the possible comparison of the proposed solution with well-known TCP variants such as NewReno, CUBIC, and BBR will provide more comprehensive insights about its strengths and limitations.

- Design of the protocol: Even though the results of the experiments demonstrated the functionality of the solution, it is not a complete protocol. By analyzing a more complex model of the network, a more comprehensive analytical model can be designed for the solution.

- SDN Simulation Tool: The simulation tool is designed and implemented in a way to be highly modular and extensible. The current version of the simulator uses zero processing delays for SDN entities such as switches and the controller. A simple measurement study of a small real-world SDN network can improves the simulator's model of the network layer. Moreover, the model for the TCP variants and UDP protocol can be easily integrated to the simulator.

- Actual implementation and experimental evaluation: Implementing a prototype of the solution and evaluating it on a real SDN network could provide more insights about its practicality.

Bibliography

[1] A. Abdelmoniem, B. Bensaou, and A. Abu, "Mitigating Incast-TCP Congestion in Data Centers with SDN", *Annals of Telecommunications*, Vol. 73, No. 4, pp. 263–277, April 2018.

[2] N. Abramson, "The Aloha System: Another Alternative for Computer Communications", In *Proceedings of the November 17-19, 1970, fall joint computer conference*, Houston, Texas, pp. 281–285, November 1970.

[3] A. Agrawal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing", In *Proceedings of IEEE INFOCOM '2000* , Tel Aviv, Israel, Vol. 3, pp. 1157–1165, March 2000.

[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)", In *Proceedings of ACM SIGCOMM '10*, New Delhi, India, pp. 63–74, September 2010.

[5] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control", *RFC 2581*, April 1999.

[6] J. Bao, J. Wang, Q. Qi, and J. Liao, "ECTCP: An Explicit Centralized Congestion Avoidance for TCP in SDN-based Data Center", In *Proceedings of IEEE Symposium on Computers and Communications (ISCC) '18*, Natal, Brazil, pp 347–353, November 2018.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, and W. Snow, "ONOS: Towards an Open, Distributed SDN OS", In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN) '14*, Chicago, Illinois, pp. 1–6, August 2014.

[8] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol–HTTP/1.1", *RFC 2616*, June 1999.

[9] L. Brakmo and L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", *IEEE Journal on Selected Areas in Communications*, Vol. 13, NO.8, pp. 1465–1480, October 1995.

[10] C. Caini and R. Firrincieli, "TCP Hybla: a TCP Enhancement for Heterogeneous Networks", *International Journal of Satellite Communications and Networking*, Vol. 22, No. 5, pp. 547–566, September 2004.

[11] N. Cardwell, Y. Cheng, C. S. Gunn, S. Yeganeh, and V. Jacobson, "BBR: Congestion-based Congestion Control", *ACM Queue* , Vol. 14, pp. 20–53, December 2016.

[12] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication", *IEEE Transactions on Communications*, Vol. 22, No. 5, pp. 637–648, May 1974.

[13] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph, "Understanding TCP Incast Throughput Collapse in Data Center Networks", In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN) '09*, Barcelona, Spain, pp. 73–82. August 2009.

[14] D. Chiu and R. Jain, "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks", *Computer Networks and ISDN Systems*, Vol. 17, No. 1, pp. 1–14, June 1989.

[15] S. ElRakabawy, A. Klemm, and C. Lindemann, "TCP with Adaptive Pacing for Multihop Wireless Networks", In *Proceedings of the 6th ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc) '05*, Urbana-Champaign, IL, pp. 288–299, May 2005.

[16] D. Erickson, "The Beacon Openflow Controller", In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN) '13*, Hong Kong, China, pp. 13–18. August 2013.

[17] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", *RFC 2582*, April 1999.

[18] A. Gelberger, N. Yemini, and R. Giladi, "Performance Analysis of Software-Defined Networking (SDN)", In *Proceedings of IEEE MASCOTS '13*, San Francisco, CA, pp 389–393, August 2013.

[19] M. Ghobadi and Y. Ganjali, "TCP Pacing in Data Center Networks", In *Proceedings of IEEE 21st Annual Symposium on High-Performance Interconnects*, San Jose, CA, pp. 25–32. October 2013.

[20] M. Ghobadi, S. Yeganeh, and Y. Ganjali, "Rethinking End-to-End Congestion Control in Software-Defined Networks", In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets) '12*, Redmond, Washington, pp. 61–66. October 2012.

[21] L. Grieco and S. Mascolo, "Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control", *ACM SIGCOMM Computer Communication Review*, Vol. 34, No. 2, pp. 25–38, April 2004.

[22] S. Ha, I. Rhee, and L. Xu, "Cubic: A New TCP-friendly High-speed TCP Variant", *ACM SIGOPS Operating Systems Review*, Vol. 42, No. 5, pp. 64–74, July 2008.

[23] G. Hasegawa, K. Kurata, and M. Murata, "Analysis and Improvement of Fairness Between TCP Reno and Vegas for Deployment of TCP Vegas to the Internet", In *Proceedings of the International Conference on Network Protocols (ICNP) '2000*, Osaka, Japan, pp. 177–186, August 2000.

[24] J. Hwang, J. Yoo, S. Lee, and H. Jin, "Scalable Congestion Control Protocol Based on SDN in Data Center Networks", In *Proceedings of IEEE Global Communications Conference (GLOBECOM) '15*, San Diego, CA, pp. 1–6, December 2015.

[25] V. Jacobson, "Congestion Avoidance and Control", In *Proceedings of ACM SIGCOMM '88*, Standford, CA, pp 314–329, August 1988.

[26] V. Jacobson, "Berkley TCP Evolution from 4.3-Tahoe to 4.3-Reno", In *Proceedings of the 18th Internet Engineering Task Force*, Vancouver, Canada, August 1990.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined WAN", In *Proceedings of ACM SIGCOMM '13*, Hong Kong, China, pp. 3–14, August 2013.

[28] C. Jin, D. Wei, and S. Low, "Fast TCP: Motivation, Architecture, Algorithms, Performance", In *Proceedings of IEEE INFOCOM '04*, Hong Kong, China, Vol. 4, pp. 2490–2501, November 2004.

[29] S. Jouet, C. Perkins, and D. Pezaros, "OTCP: SDN-managed Congestion Control for Data Center Networks", In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS) '16*, Istanbul, Turkey, pp. 171–179, July 2016.

[30] R. Kahn, S. Gronemeyer, J. Burchfiel, and R. Kunzelman, "Advances in Packet Radio Technology", *Proceedings of the IEEE*, Vol. 66, No. 11, pp. 1468–1496, November 1978.

[31] J. Ke and C. Williamson, "Towards a Rate-based TCP Protocol for the Web", In *Proceedings of IEEE MASCOTS '2000*, San Francisco, CA, pp 36–45. August 2000.

[32] L. Kleinrock, "Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications", In *Proceedings of the International Conference on Communications (ICC) '79*, Boston, Mass, pp. 43.1.1–43.1.10, June 1979.

[33] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge., "A Simulation Study of Paced TCP", CR-2000-209416, NASA, Tech. Rep, January 2000.

[34] J. Kurose and K. Ross, Computer Networking: A Top Down Approach, Seventh Edition, Pearson, April 2016.

[35] S. Liu, T. Başar, and R. Srikant, "TCP-Illinois: A Loss and Delay-based Congestion Control Algorithm for High-speed Networks", *Performance Evaluation*, Vol. 65, No. 6-7, pp. 417–440, June 2008.

[36] Y. Lu, Z. Ling, S. Zhu, and L. Tang, "SDTCP: Towards Data Center TCP Congestion Control with SDN for IoT Applications", *Sensors*, Vol. 17, No. 1, pp. 109, January 2017.

[37] Y. Lu and S. Zhu, "SDN-based TCP Congestion Control in Data Center Networks", In *Proceedings of IEEE 34th International Performance Computing and Communications Conference (IPCCC) '15*, Nanjing, China, pp. 1–7, February 2015.

[38] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang, "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links", In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom) '01*, Rome, Italy, pp. 287–297, July 2001.

[39] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling Innovation in Campus Networks",

ACM SIGCOMM Computer Communication Review, Vol. 38, No. 2, pp. 69–74, April 2008.

[40] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-driven SDN Controller Architecture", In *Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks '14*, Sydney, Australia, pp. 1–6. October 2014.

[41] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Reno Performance: A Simple Model and its Empirical Validation", *IEEE/ACM Transactions on Networking*, Vol. 8, No. 2, pp. 133–145, April 2000.

[42] J. Pahdye and S. Floyd, "On Inferring TCP Behavior", In *Proceedings of ACM SIGCOMM '01*, San Diego, CA, pp. 287–298, August 2001.

[43] N. Parvez, A. Mahanti, and C. Williamson, "TCP New Reno: Slow-but-steady or Impatient?", In *Proceedings of IEEE International Conference on Communications (ICC) '06*, Istanbul, Turkey, pp. 716–722, December 2006.

[44] J. Postel, "User Datagram Protocol", *RFC 768*, August 1980.

[45] J. Postel, "Simple Mail Transfer Protocol", *RFC 5321*, October 2008.

[46] J. Postel, "Internet Protocol", *RFC 791*, September 1981.

[47] J. Postel, "Transmission Control Protocol", *RFC 793*, September 1981.

[48] J. Postel and J. Reynolds, "File Transfer Protocol", *RFC 959*, October 1985.

[49] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving data Center Performance and Robustness with Multipath TCP", In *Proceedings of ACM SIGCOMM '11*, Toronto, Ontario, pp. 266–277, August 2011.

[50] O. Salman, I. Elhajj, A. Kayssi, and A. Chehab, "SDN Controllers: A Comparative Study", In *18th Mediterranean Electrotechnical Conference (MELECON) '16*, Lemesos, Cyprus, pp 1–6. June 2016.

[51] H. Shimonishi, M. Sanadidi, and M. Gerla, "Improving Efficiency-Friendliness Tradeoffs of TCP in Wired-Wireless Combined Networks", In *Proceedings of IEEE International Conference on Communications (ICC) '05*, Seoul, South Korea, August 2005.

[52] K. Srijith, L. Jacob, and A. Ananda, "TCP Vegas-a: Improving the Performance of TCP Vegas", *Computer Communications*, Vol. 28, No. 4, pp. 429–440, March 2005.

[53] A. Veres and M. Boda, "The Chaotic Nature of TCP Congestion Control", In *Proceedings IEEE INFOCOM '2000*, Tel Aviv, Israel, Vol. 3, pp. 1715–1723, March 2000.

[54] R. Wang, M. Valla, M. Sanadidi, and M. Gerla, "Adaptive Bandwidth Share Estimation in TCP Westwood", In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM) '02*, Taipei, Taiwan, pp. 2604–2608 , November 2002.

[55] Z. Wang and J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe TCP Congestion Control Algorithm", *ACM SIGCOMM Computer Communication Review*, Vol. 22, No. 2, pp. 9–16, April 1992.

[56] D. Wei, P. Cao, S. Low, and C. EAS, "TCP Pacing Revisited", In *Proceedings of IEEE INFOCOM '06*, Barcelona, Spain, Vol. 2, page 3–14, April 2006.

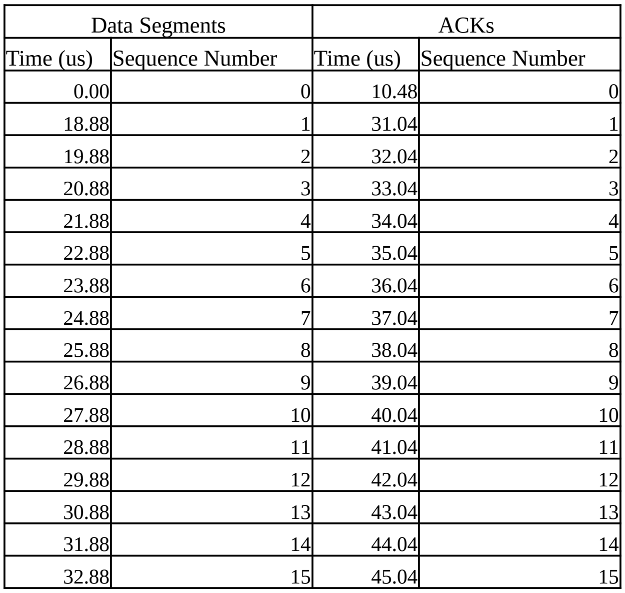
[57] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks", *IEEE/ACM Transactions on Networking*, Vol. 21, No. 2, pp. 345–358, April 2013.

[58] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control (BIC) for Fast Long-distance Networks", In *Proceedings of IEEE INFOCOM '04*, Hong Kong, China, Vol. 4, pp. 2514–2524, March 2004.

Appendix A

A.1 Debugging Output

The simulation tool is capable of generating sequence number output for any desired flow in the scenario. This output is used for debugging purposes. Figure A.1 shows the output data generated in form of a table. Figure A.2 shows the generated graph for the sequence number data.



Data Segments		ACKs	
Time (us)	Sequence Number	Time (us)	Sequence Number
0.00	0	10.48	0
18.88	1	31.04	1
19.88	2	32.04	2
20.88	3	33.04	3
21.88	4	34.04	4
22.88	5	35.04	5
23.88	6	36.04	6
24.88	7	37.04	7
25.88	8	38.04	8
26.88	9	39.04	9
27.88	10	40.04	10
28.88	11	41.04	11
29.88	12	42.04	12
30.88	13	43.04	13
31.88	14	44.04	14
32.88	15	45.04	15

Figure A.1: The sequence number output in form of table.

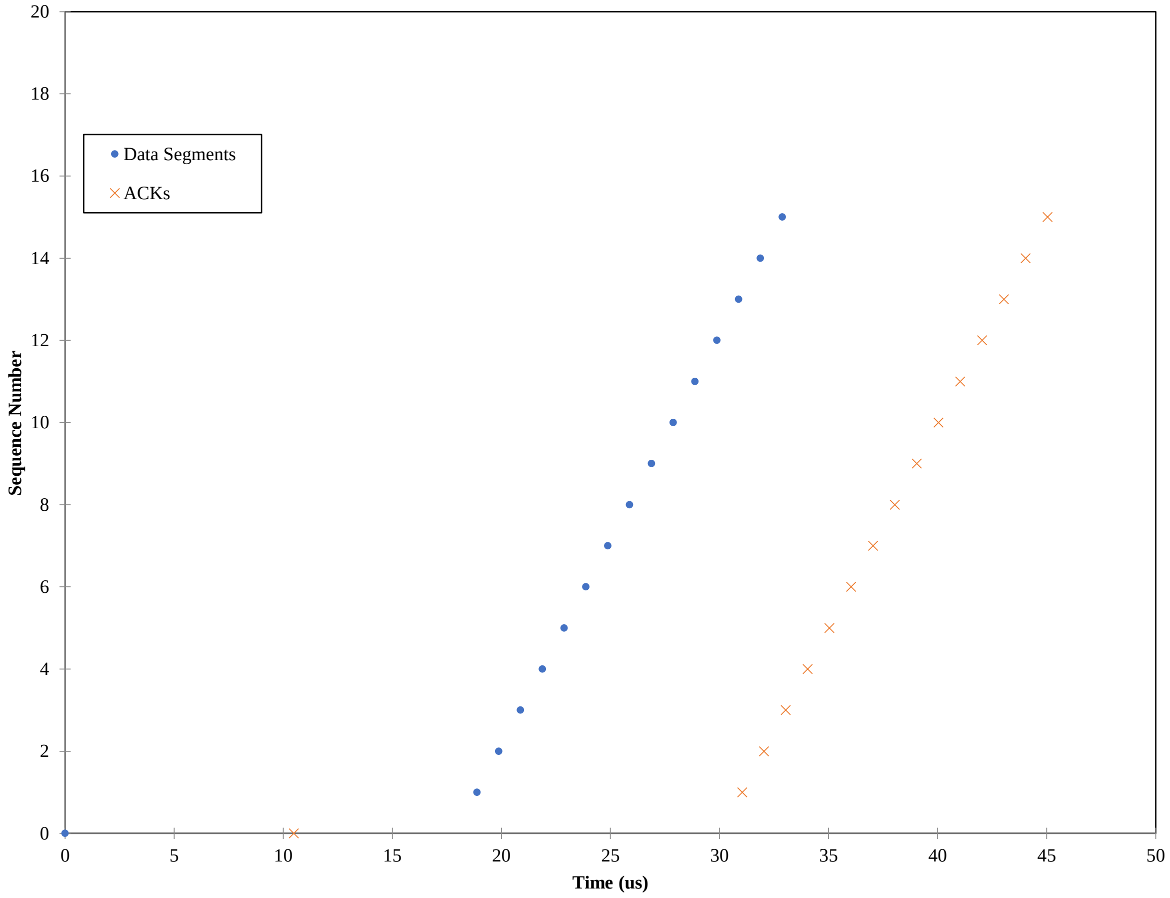
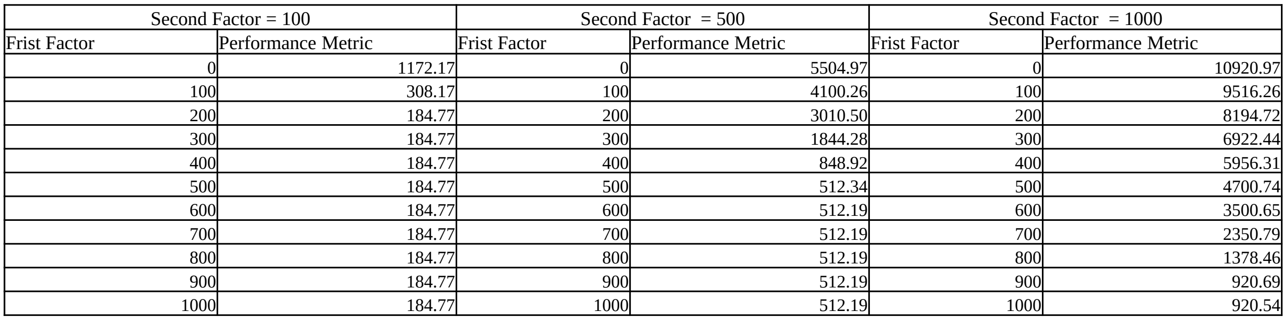


Figure A.2: The sequence number graph.

A.2 Study Output

The simulation tool is capable of generating the output for a double factor simulation study in form of table and graphs. Figure A.3 represents the table generated by the tool for a performance metric. Figure A.4 shows the line graph generated for a double factor study. This type of graph is suitable for studies with a continuous first factor. Figure A.5 shows the bar graph generated for a double factor study. This type of graph is recommended for studies with categorical first factor.



Second Factor = 100		Second Factor = 500		Second Factor = 1000	
Frist Factor	Performance Metric	Frist Factor	Performance Metric	Frist Factor	Performance Metric
0	1172.17	0	5504.97	0	10920.97
100	308.17	100	4100.26	100	9516.26
200	184.77	200	3010.50	200	8194.72
300	184.77	300	1844.28	300	6922.44
400	184.77	400	848.92	400	5956.31
500	184.77	500	512.34	500	4700.74
600	184.77	600	512.19	600	3500.65
700	184.77	700	512.19	700	2350.79
800	184.77	800	512.19	800	1378.46
900	184.77	900	512.19	900	920.69
1000	184.77	1000	512.19	1000	920.54

Figure A.3: The double factor study output in form of table.

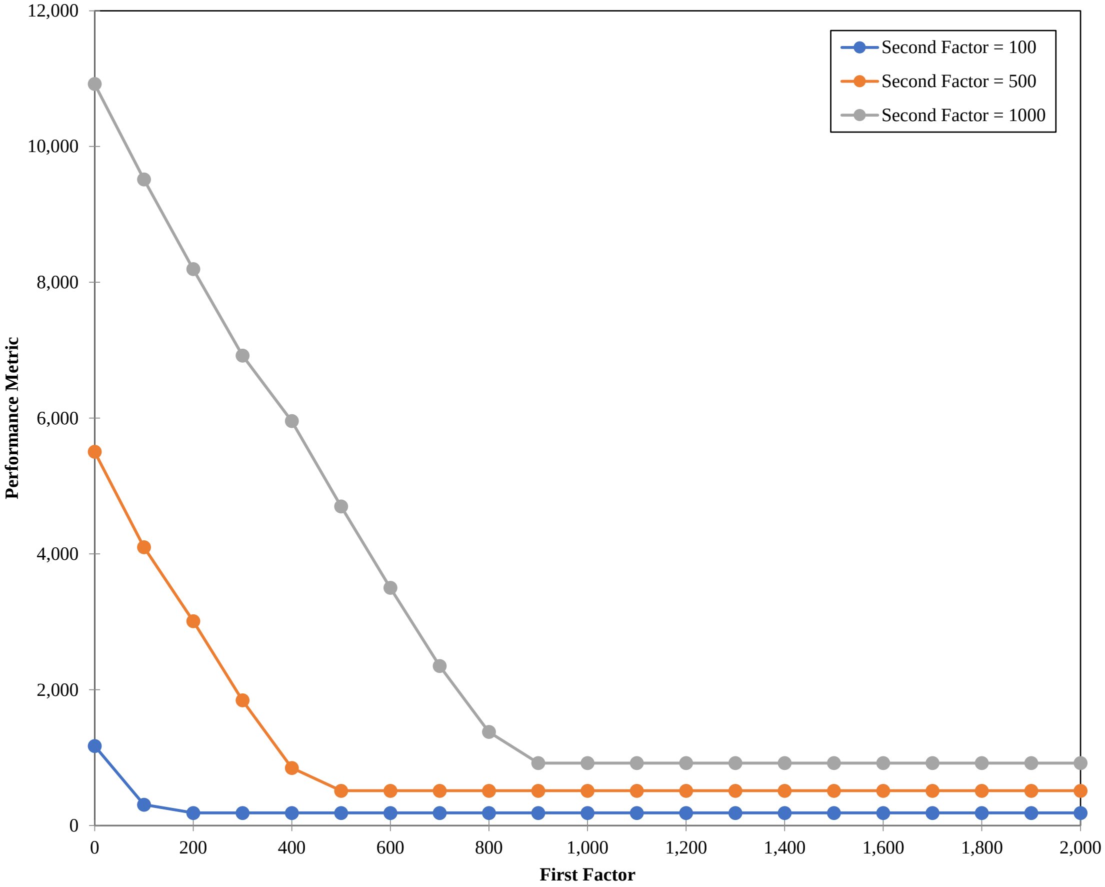


Figure A.4: The double factor study output in form of line graph.

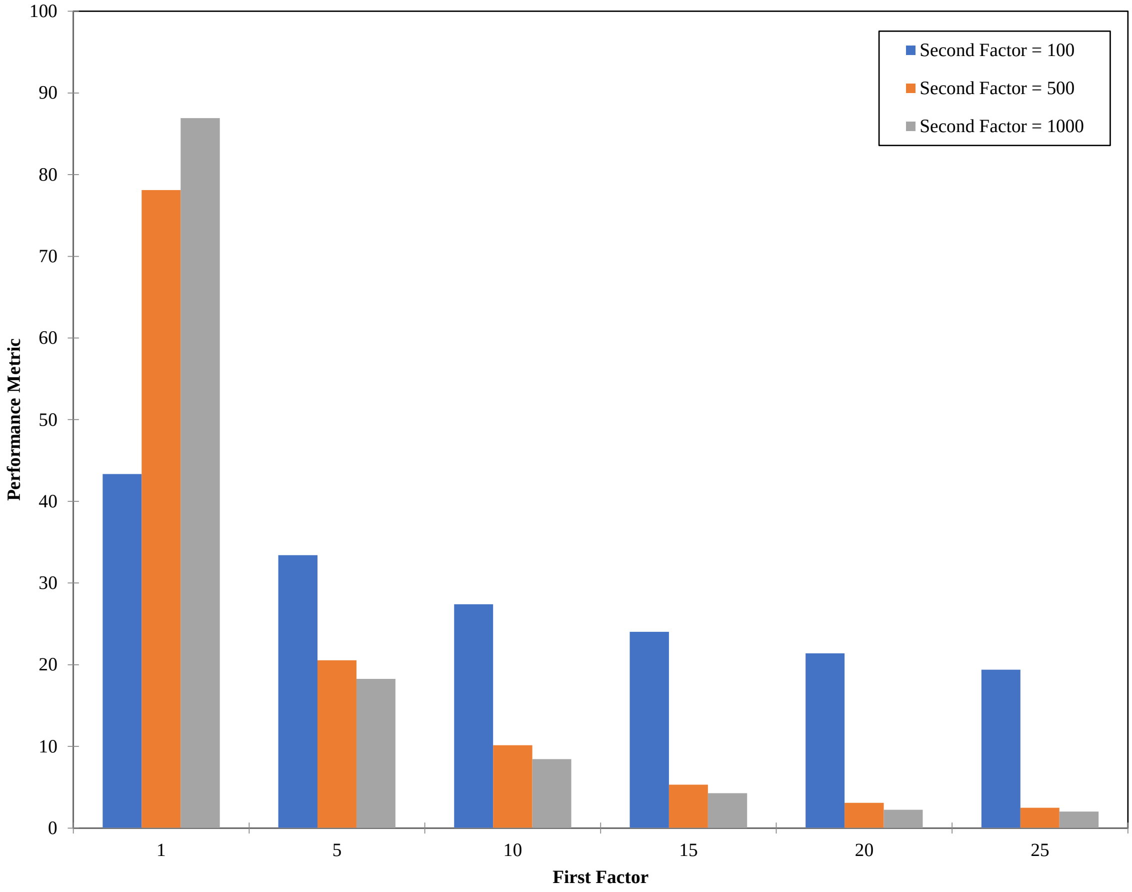


Figure A.5: The double factor study output in form of bar graph.