Department of Informatics

Master Thesis

Exploring MultiPath TCP Through Discrete Event Simulation

Henrik Libeck Hexeberg
Supervisor: Øyvind Ytrehus

June 1, 2021

Abstract

Global internet usage is rapidly becoming more mobile. Devices are, as a consequence, equipped with multiple network interfaces to meet the demand for mobility. With the emergence of 5G-technology, this trend will most likely continue. However, most internet traffic makes use of standard TCP. TCP does not allow a connection to span several interfaces, even though multiple network interfaces are available. MPTCP is a proposed protocol that enables the use of several interfaces. By distributing data through several paths, MPTCP aims at making internet usage more robust while providing higher throughput and multihoming features. In this thesis, we develop a discrete event simulator to simulate a network. Using this simulator, we analyze and evaluate the performance of MPTCP in various situations. Our findings show that MPTCP has several pitfalls and highlight possible mitigations.

Acknowledgements

I want to express my special thanks of gratitude to my supervisor Øyvind Ytrehus and everyone at Simula@UiB. Thank you for all the mentoring, supervision, and patience. Besides my supervisor, I would like to thank my family and Mariah, my friends Christopher, Jonas, Håkon, Anders, and Sondre, in addition to all the people I have gotten to know during my time at the University in Bergen. Your help, support, and encouragement have truly been appreciated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter will give an introduction to the thesis, including motivation, goal, and overview.

1.1 Motivation

The internet was created in a time with no mobile internet devices. Today, mobile devices are widespread, but the fundamental internet protocols are essentially the same. Most devices have the capability to use multiple network interfaces at once but rarely do. However, utilizing multiple interfaces to communicate could effectively eliminate many of the challenges wireless, and mobile devices have.

Firstly, mobile devices move around and will often change IP addresses, which identifies where the data should be sent. The change of IP address can result in a lost connection and the need to establish a new connection with the new IP. The process is relatively time-consuming and is not acceptable in latency-sensitive applications such as Voice over Internet Protocol (VoIP) or real-time video applications.

Secondly, wireless devices are prone to more losses than wired devices due to signal degradation. Using multiple channels combined makes it possible to offload a faulty channel when a failure is detected, making the device more reliable.

Thirdly, applications are becoming more data demanding. Aggregating the potential bandwidth of two or more interfaces could significantly increase throughput while also improving resource usage within the network.

MPTCP is a protocol that enables the use of multiple paths during communication, with the potential to solve the problems mentioned above. In practice, the protocol allows for simultaneously using 4G and WiFi or other combinations. With the emergence of 5G, MPTCP becomes increasingly relevant.

1.2 Goal

This thesis explores the possible benefits and obstacles of using MPTCP. By using a discrete event simulator, we hope to gain the insight needed to test and suggest improvements to MPTCP.

1.3 Overview

Chapter 1 Introduction - Outline of this thesis.

Chapter 2 Background - Introducing concepts, notations, and terminology.

Chapter 3 Related Work - Summary of related research in the field.

Chapter 4 Method - Description of the developed simulator and how it is used.

Chapter 5 Results - Presentation of the obtained results.

Chapter 6 Conclusion - Conclusion and future work.

Chapter 2

Background

This chapter will introduce concepts, notations, and terminology needed to understand the thesis. The internet protocol suite and TCP are sections that describe how the internet works. Then, an introduction to MPTCP is given. At last queueing theory and discrete event, simulation is presented.

2.1 The Internet Protocol Suite

The Internet protocol suite is a number of protocols used to communicate over computer networks. It is often referred to as TCP/IP because of the two fundamental protocols Transmission Control Protocol (TCP) and Internet Protocol (IP). TCP/IP's key characteristic is the use of encapsulation through the concept of network layers. Several different models describe the internet in a layered fashion. In the context of TCP/IP, the model specified by RFC1122 [1] is often used:

1. Application layer
2. Transport layer
3. Internet layer
4. Link layer

All layers depend on the layers beneath, and should in theory, not interfere with each other. Information is only passed between adjacent layers. Each layer has layer-specific protocols that handle the responsibilities assigned to the layer. For example, the transport layer should not handle the routing of packets; this is the internet layer's responsibility. Equally, the application layer should not affect the rate at which packets are sent, as it is the transport layer's responsibility. [2]

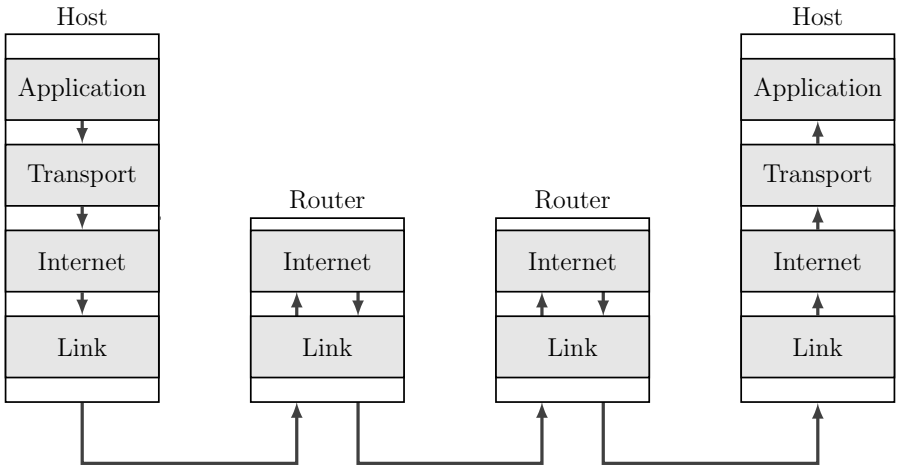


Figure 2.1: TCP/IP Data Flow

2.1.1 Internet Layer

The internet layer is responsible for communication between hosts through the use of IP addresses. IP addresses allow for communication between hosts on the same network as well as communication between networks through connected gateways. The Internet Protocol (IP) is the protocol used to find the next host on the path to the destination of the packet. When the next host is found, the packet is passed to the link layer. Here the packet is sent to the given host. The next host will, in turn, do the same until the destination is reached. The Internet layer provides best-effort delivery and therefore does not guarantee the eventual delivery of the sent packets. [2]

IP Fragmentation

IP fragmentation is the capability of splitting and merging packets in order not to exceed the links Maximum Transmission Unit (MTU), i.e., the maximum size of data that can be transmitted at once. Different links on the Internet have different MTU, which affects the number and size of the packets to be sent. All protocols that directly or indirectly use IP must consider this as IP fragmentation can cause higher propagation and processing delay. Fragments of packets can be lost, leading to retransmission of the whole packet. [2]

2.1.2 Transport Layer

The Transport layer is situated between the Application layer and the Internet Layer in the TCP/IP model. In contrast to the Internet layer, the Transport layer is only operating on endpoints in the communication. The layer provides host-to-host communication for the application layer to use. This layer contains the two protocols TCP, which provides connected and reliable communication, and User Datagram Protocol (UDP) that provides connectionless and best-effort communication. The layer can facilitate the following

features:

1. Connection based communication
2. Reliable communication
3. Fairness to other users of the Internet

[2]

2.2 TCP

The Transmission Control Protocol (TCP) is a connection-based and reliable protocol situated in the Transport layer. It features flow control and congestion control mechanisms that ideally should be fair to other internet users. However, the fairness of the mechanisms are disputed; connections with longer paths can, for instance, experience lower throughput than competing connections with shorter path [3]. Also, there are multiple versions of TCP that mainly differ in the way the congestion control mechanism is implemented. This thesis will only concentrate on the TCP Tahoe and TCP Reno implementations.

TCP rely on a virtual connection to communicate with the other part. The virtual connection is only maintained by the endpoints and is needed in the process of making TCP reliable. TCP connections are established with a three-way handshake as shown in figure 2.2.

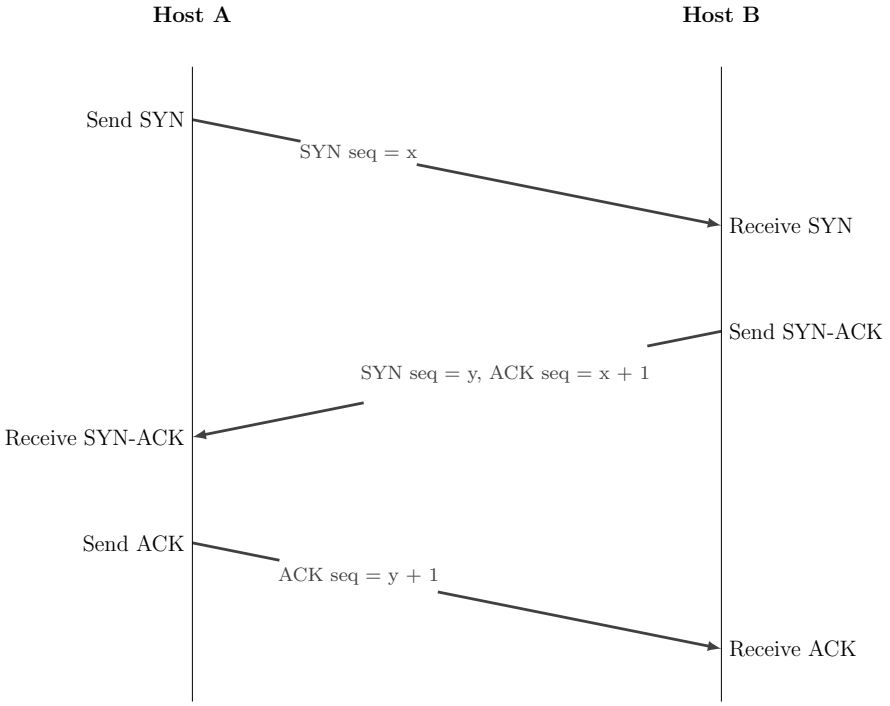


Figure 2.2: TCP Three-Way Handshake

Host A sends a Synchronize (SYN) packet to host B. The SYN packet is what initiates a connection. Upon receiving the SYN, host B will acknowledge (ACK) the SYN while opening a connection with Host A. this packet is called SYN-ACK. Host A will, in turn, acknowledge the incoming connection, and the two parts have established a connection. [2]

2.2.1 Flow Control

Flow Control is a mechanism in TCP for ordered and reliable data transfer. Flow control established the maximum number of packets that can be sent and controls what sending discipline is used to make communication reliable. The mechanism often uses a form of Automatic Repeat Request (ARQ), but Forward Error Correction (FEC) can also be used. ARQ's and FEC's are error control mechanisms used to either notify the sender that an error has occurred or assume an error rate and add redundancy. In ARQ's this means acknowledging received packets (ACK). The absence of an ACK tells the sender that retransmission is needed. For FEC's, the redundant packets are added in the hopes of recreating missing packets. However, this scheme must know the packet loss rate in forehand to know the redundancy needed [4]. Hybrids of the two error control mechanisms have also been implemented, with the benefit being that the loss rate can be unknown and adjusted [5]. Further, the TCP's flow control will notify the sender how many packets can be sent at any given time. This limit is called the Receiver Window (RWND). It prohibits packet loss caused by congestion at the receiver's end. Since sending one packet at a time is slow, different methods of sending multiple packets are often used. The technique must, however, still respect the restriction given by the RWND. [2]

Retransmission

Retransmissions occur when packets are lost, and an ARQ or ARQ-FEC hybrid mechanism is used. Packet loss can happen due to network congestion or link degradation caused by interference or other conditions. Regardless, the occurrence of a lost packet is detected by either not receiving an ACK or by getting multiple ACKs called DupACKs on a specific packet (caused by the receiver getting multiple packets out of order). A DupACK signals the sender to retransmit a specific packet. Not receiving an ACK will, after a timeout, trigger retransmission. [2]

ARQ

Automatic Repeat Request (ARQ) can be implemented in several different ways. The most common are Stop-and-wait, Go-Back-N, and Selective Repeat. The Stop-and-wait ARQ is the simplest. It sends only one packet at a time and waits for the acknowledgment before sending a new packet. A lost packet will trigger a retransmission of the packet sent.

The Go-Back-N ARQ is reasonably easy to implement while significantly improving the sending rate compared to Stop-and-wait. It features a sliding window of size N on the

sending side of the communication. The sender sends N packets before requiring an ACK, acknowledging all the N packets. If one of the N packets or the ACK is lost, all N packets will be retransmitted. While this ARQ is more efficient compared to Stop-and-wait, it can cause excessive retransmissions.

The Selective Repeat ARQ can reduce the number of retransmissions needed but at the cost of higher complexity and higher memory requirements. It uses a sliding window with size N at both sides of the communication. N packets are sent, and all packets are ACKed individually. When a packet is lost, the packets out of order, i.e., the packets that in sequence are after the lost packet, are stored in the receiver's buffer. Either through a Duplicate Acknowledgement (DupACK) or through a Retransmission Timeout (RTO). The lost packet is retransmitted, and the stored packets can be sent to the application layer in the correct order. [2]

2.2.2 Congestion Control

Congestion control consists of four mechanisms that limit the number of packets that can be sent simultaneously through a scheme called Additive-Increase/Multiplicative-Decrease (AIMD) and a Congestion Window (CWND). The AIMD scheme increases CWND by adding a fixed integer a and decreases CWND by multiplying a fixed number b. This implies that a and b have the following properties. $a \in \mathbb{N}$, and $0 < b < 1$. Often, a is set to 1, and b is set to 0.5 but can have different values depending on implementation.

The main four congestion control mechanisms are slow start, congestion avoidance, fast retransmit, and fast recovery. They are aimed at limiting congestion while achieving high throughput and being fair to other users of the internet. By using the Congestion Window (CWND), the mechanisms can dynamically adjust the sending rate. It is important to note that while congestion control limits the number of packets that can be sent at a time, the size of the RWND will always work as an upper bound to the congestion control algorithms. $CWND \leq RWND$. [2]

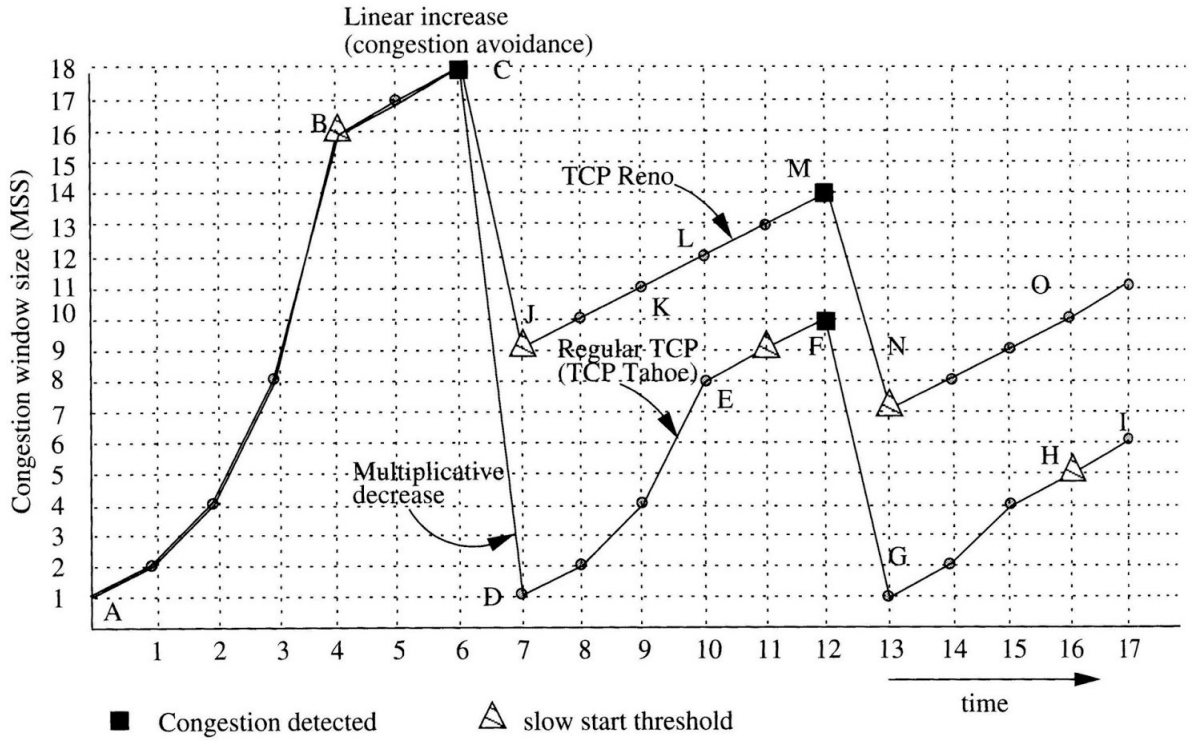


Figure 2.3: TCP Reno and Tahoe Congestion Control [6]

Slow Start

The slow start mechanism is initiated when a new connection is established and when retransmission caused by a Retransmission Timeout (RTO) occurs. It starts with a small initial congestion window and stops its operation when the slow start threshold (*ssthresh*) is hit or congestion is detected. Since the initial *ssthresh* value is set to an arbitrarily high value, i.e., as high as possible, the initial slow start operation is operating until congestion is detected. This can be thought of as testing the waters before diving in. Despite its name, slow start is increasing the sending rate aggressively. The congestion window size is increasing by one for every acknowledged packet. In practice the congestion window will double in size every Round Trip Time (RTT), making it an exponential growth. [7] [2]

Congestion Avoidance

Congestion avoidance is initiated when slow start hits *ssthresh*. The mechanism will continue to adjust the sending rate until congestion is detected, in which case the CWND is multiplicatively decreased. When operating without loss, the algorithm increases the CWND by adding $a/CWND$ for each acknowledged packet, i.e., adding a to CWND each RTT. Consequently, congestion avoidance operates with linear growth. When congestion occurs, the sender is notified in two ways, either explicitly through a DupACK or a RTO. This is where the two main implementations of TCP differ. TCP Tahoe will treat three

successive DupACK the same way as a normal RTO, effectively reducing the CWND to the initial value and starting the slow start mechanism. TCP Reno will, however, treat three successive DupACK's in another way. CWND will be multiplicatively decreased with the factor b, then a fast retransmit of the lost packet is triggered, and the fast recovery mechanism is initiated. If a RTO occurs, then the slow start mechanism starts with the initial CWND, just like in TCP Tahoe. [7] [8] [2]

Fast Restransmit

Fast retransmit is a mechanism that improves the throughput by fast-retransmitting packets that most likely are lost due to congestion. When a DupACK is received, it can mean that the packet is lost, but it can also mean that packets must be reordered. If reordering is needed, the retransmission of the packet is not necessary. However, if multiple DupACK's occur, the packet is more likely to have been lost in transmission. The fast retransmit mechanism is therefore initiated if three or more DupACK's are received. After the presumed lost packet is retransmitted, TCP initiates either slow start or fast recovery depending on implementation. [7] [2]

Fast Recovery

Fast recovery is a mechanism that, much like fast retransmit, exists to improve the throughput. It works as an alternative to slow start by going directly to the congestion avoidance mechanism and is deployed in TCP Reno. The reasoning behind this mechanism is that the occurrences of DupACKs are prof that the lost packet is not part of a major congestion problem in the network, but rather a rare event. DupACKs are only sent if packets are received out of order, essentially proving that no serious network congestion has occurred. [7] [2]

2.3 MultiPath TCP

MultiPath TCP (MPTCP) is an emerging protocol situated in the transport layer that aims to utilize multiple paths during communication. It is implemented using TCP extensions defined in RFC8684 [9]; however, it has been divided opinions on whether the implementation should be embedded in the payload instead. The goals of MPTCP are many, with the primary goal being to make the internet more robust with better performance and load-balancing. This should, however, be implemented in a way that presents itself as TCP to the layer above, namely the application layer. Other goals of MPTCP are to perform at least as good as standard TCP in terms of resource usage and reliability.

One of the main benefits of using multiple paths aside from higher throughput is the redundancy of multiple paths. If one path no longer exists, all data can be sent through the remaining paths. Mobile devices often have WiFi and 4G(LTE) available at the same

time. Using MPTCP it is possible to use both interfaces at the same time as shown in Figure 2.4.

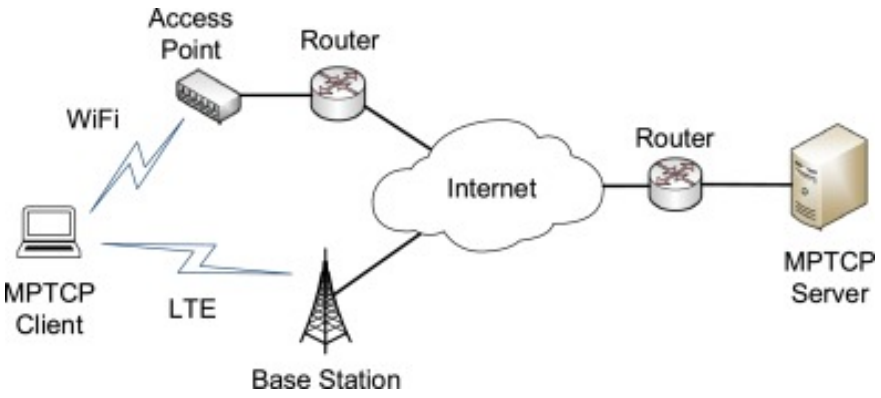


Figure 2.4: MPTCP [10]

Due to the many middleboxes on the internet, such as NAT and load-balancers, the implementation must touch all TCP functions like flow control, congestion control, and connection establishment and tear-down. This increases the complexity needed to deploy a working MPTCP implementation successfully. [11]

2.4 Queuing Theory

This section will go through the fundamentals of queueing theory and is influenced by the book Fundamentals of Queueing Theory [12]. This source is used throughout this section and is the used source if not stated otherwise.

2.4.1 Kendall Notation

In queuing theory, a node's queuing process can be described using the Kendall notation. This notation has six parameters and is divided by slashes like this: A/S/c/K/N/D. The position and value describe how the queuing process handles and experiences arrivals (A), Service (S), number of servers (c), capacity (K), source size (N), and queuing discipline (D). However, the three last parameters are often left unspecified. In which case the the values are assumed to be K = ∞ , N = ∞ and D = FIFO. Each parameter can take multiple different values.

Interarrival Time Distribution (A)

Interarrival time distribution (A) is a symbol that denotes the interarrival time distribution used in the queuing system and is the first parameter in the Kendall notation. See table 2.1

Symbol	Description	Explanation
M	Exponential	Poisson arrival process. Inherits the Markovian or memoryless property and is discrete.
M^X	Batch or Bulk Markovian	Poisson arrival process with a random X arrivals between 1 and ∞. Inherits the Markovian or memoryless property and is discrete.
D	Deterministic	Deterministic arrival process. Arrivals occur at a fixed rate.
G	General	Independent or unknown arrival process.
E_k	Erlang type-k	An Erlang distribution with k as shape parameter. i.e the average time it takes for k Poisson arrivals to occur ($E_1 = M$)

Table 2.1: Interarrival Time Distributions

Service Time Distribution (S)

Service time distribution (S) is a symbol that denotes the service time distribution used in the queuing system and is the second parameter in the Kendall notation. See table 2.2

Symbol	Description	Explanation
M	Exponential	Exponential service time. Has the Markovian or memoryless property.
M^X	Batch or Bulk Markovian	Exponential service time with a random X between 1 and ∞ customers being served at once. Has the Markovian or memoryless property.
D	Deterministic	Deterministic service time. The time it takes to serve a customer is fixed
G	General	Independent or unknown service time.
E_k	Erlang type-k	An Erlang distribution with k as shape parameter. i.e the average time it takes to serve k customers ($E_1 = M$)

Table 2.2: Service Time Distributions

Parallel Servers (c)

Parallel servers (c) is an integer value that denotes the number of parallel servers in the system and is the third parameter in the Kendall notation. Each parallel server has an independent but identical service distribution. For instance, an M/M/2 queue has a Poisson arrival process (section 2.4.3) and an exponential service rate (section 2.4.4). Both the arrival and service distributions are independent of the number of parallel servers in

the system. The arrival rate represents the arrivals to the system and not a server. The service rate describes the time a server uses to serve a customer and not the time the system uses.

System Capacity (K)

System capacity (K) is an integer value that denotes the capacity of the system and is the fourth parameter in the Kendall notation. It describes the maximum number of customers in the system at any time. An M/M/2/3 queue can have no more than one customer in the queue since this implies that two customers are being served.

Source Population (N)

Source population (N) is an integer value that denotes the source population and is the fifth parameter in the Kendall notation. Other values than ∞ mean that there is a finite source population which in turn will affect the arrival rate to the queuing system.

Queuing Discipline (D)

An acronym that denotes the queuing discipline used in the queuing system and is the sixth parameter in the Kendall notation. See table 2.3

Acronym	Description
FIFO/FCFS	Fist In Fist Out/First Come First Serve
LIFO/LCFS	Last In First Out/Last Come First Serve
SIRO/RSS	Service In Random Order/Random Selection for Service
PR	Priority
GD	General Discipline

Table 2.3: Queuing Disciplines

2.4.2 Notations

Table 2.4 present the key notations and results for G/G/c queues.

Notation	Description
A	Random arrival time
S	Random service time
$E[x]$	Expected value or mean of x
$\lambda \equiv 1/E[A]$	Average arrival rate
$\mu \equiv 1/E[S]$	Average service rate
c	Number of servers
$r \equiv \lambda/\mu$	Offered load
$\rho \equiv \lambda/c\mu$	Traffic intensity or utilization
T	Random time a customer spends in the system
T_q	Random time a customer spends in the queue
W	Average time a customer spends in the system
W_q	Average time a customer spends in the queue
N	Random number of customers in the system
N_q	Random number of customers in the queue
L	Average number of customers in the system
L_q	Average number of customers in the queue

Table 2.4: Key Queueing Theory Notations

2.4.3 Arrival

An arrival process defines the rate at which customers arrive at a queue. λ is used to denote the average arrival rate of the arrival process, which in turn is used in the calculation of several important metrics. Note that the term customer is used as a general term to describe an entity that enters a queueing system. Looking at a queue in a simplistic manner, a customer that arrives at a queue can either be served at once or must wait in queue for service. However, this assumption does not take into consideration that the queue might, for example, have a finite capacity or that customers can choose not to enqueue for some reason. Ether way, if the arrivals to the system happen at a higher rate than customers are being served, the queue can grow indefinitely.

Knowing the exact arrival process to a system is often impossible. For this reason, it is common to use a probability distribution to model the way customers arrive at the system. The model will never be perfect, but many natural processes have shown to follow certain distributions closely. Counting processes are used to model the arrival processes since arrivals are discrete events, i.e., events that occur in time.

The Poisson Process

The Poisson process is a counting process that is commonly used to model the arrivals to a queue. A counting process $N(t)$ is defined as a stochastic process in which $N(t)$ takes on non-negative integers and is non-decreasing in time. Counting processes are used to count the number of events that occur in a given time interval.

Definition 2.4.1. *A Poisson process with rate $\lambda > 0$ is a counting process $N(t)$ with the following properties:*

1. $N(0) = 0$.
2. $P\{$*1 event between t and $t + \Delta t$*$\} = \lambda \Delta t + o(\Delta t)$.
3. $P\{$*2 or more events between t and $t + \Delta t$*$\} = o(\Delta t)$
4. *The number of events in non-overlapping intervals are statically independent; that is, the process has independent increments.*

The quantity $o(\Delta t)$ can intuitively be seen as essentially zero when $\Delta t \to 0$. Formally the relation between Δt and $o(\Delta t)$ is:

$$\lim_{\Delta t \to 0} \frac{o(\Delta t)}{\Delta t} = 0$$

While definition 2.4.1 introduce the properties of a Poisson process, it does not give the probability distribution for the counting process $N(t)$. First, the Poisson random variable needs to be defined.

Definition 2.4.2. *A Poisson random variable is a discrete random variable with a probability mass function.*

$$p_n = e^{-A} \frac{A^n}{n!} \qquad (n = 0, 1, 2, \ldots)$$

where $A > 0$ is a constant.

Using the defined properties of a Poisson process 2.4.2 and the definition of a Poisson random variable 2.4.1, it is possible to obtain the probability distributing of the counting process $N(t)$

Theorem 2.4.1. *Let $N(t)$ be a Poisson process with rate $\lambda > 0$. The number of events occurring by the time t is a Poisson random variable with mean λt. That is,*

$$p_n = e^{-\lambda t} \frac{(\lambda t)^n}{n!} \qquad (n = 0, 1, 2, \ldots)$$

where $p_n(t) \equiv P\{N(t) = n\}$.

The Poisson properties have, as a bonus, some additional characteristics that make them unique and useful when modeling arrivals.

Theorem 2.4.2. *Let $N(t)$ be a Poisson process with rate λ. Then the times between successive events are independent and exponentially distributed with rate λ*

Theorem 2.4.2 states that the Poisson process has a relation to the exponential distribution. This results in the Poisson process inherently having the memoryless property (Theorem 2.4.7). Another interesting characteristic is that a Poisson process has stationary increments, which in essence means that the probability of an event's occurrence in

time is dependent on the length of the interval, and non-dependent on the exact moment the event occurs. This characteristic is defined as follows:

Theorem 2.4.3. *A Poisson process has stationary increments. That is, for $t > s$, $N(t) - N(s)$ is identically distributed as $N(t+h) - N(s+h)$ for any $h > 0$.*

The Poisson process can be thought of as a random process, due to it's exponentially distributed interarrival times. While the concept of randomness is hard to define, the experienced randomness of a Poisson process is a result of the following characteristic.

Theorem 2.4.4. *Let $N(t)$ be a Poisson process. Given that k events have occurred in a time interval $[0, T]$, the times $t_1 < t_2 < \ldots < t_k$ at which the events occurred are distributed as the order statistics of k independent uniform random variables on $[0, T]$*

A significant side effect of Theorem 2.4.4 is the Poisson Arrivals See Time Averages (PASTA) property [13]. This property states that the observations taken according to a stochastic process have the same probability of observing a system in a given state as observing the system at Poisson selected points. More intuitively, the PASTA property states that customers arriving in a queue will have the same probability of finding the queue at a certain length as an outsider looking at the queue at a random point in time.

The last characteristic of the process is that splitting a Poison process into sub-processes will result in several Poisson processes. Equivalently, the outcome of combining several Poisson processes will yield a Poisson process. This characteristic is formally defined in the following theorems.

Theorem 2.4.5 (Splitting)**.** *Let $N(t)$ be a Poisson process with rate λ. Suppose that each event is labeled a type-i event with probability p_i, independent of all else. Let $N_i(t)$ be the number of type-i events by time t. Then $N_i(t)$ is a Poisson process with rate $\lambda p_i, i = 1, \ldots, n$. Furthermore, $N_i(t)$ and $N_j(t)$ are independent, for all $i \neq j$*

Theorem 2.4.6 (Combining)**.** *Let $N_1(t), \ldots, N_n(t)$ be independent Poisson processes with rates $\lambda_1, \ldots, \lambda_n$, respectively. Then $N(t) \equiv N_1(t) + \ldots + N_n(t)$ is a Poisson process with rate $\lambda \equiv \lambda_1 + \ldots + \lambda_n$.*

2.4.4 Service

The service time of a queue is the time it takes for a customer to complete service. In other terms, the time a customer occupies a server. Once a customer has completed service, the next customer can be served. When dealing with queuing systems with multiple servers, the service rate is provided per server and not the service rate as a whole. For this reason, the traffic intensity (ρ) measure is a product of the number of servers while the offered load (r) is not Table 2.4. In many cases, the service time distributions can seem random and are often modeled as a stochastic process, much like the arrival process. However, unlike the arrival processes, the service patterns should model time and not events in time. The probability distributions must consequently be continuous.

The Exponential Distribution

The exponential distribution is continuous and is commonly used to model the service rate of a queueing system. Additionally, it is used as the continuous-time between two events in the Poisson process (Theorem 2.4.2) and is fundamental in continuous-time Markov chains (subsection 2.4.7). The probability distribution has the memoryless property, which states that the current and future states are independent of the previous. This property is formally defined as follows:

Definition 2.4.3. *A random variable T has the memoryless property if*

$$P\{T > t + s | T > s\} = P\{T > t\} \qquad (s, t \geq 0).$$

The exponential distribution is often used to model time. This could, for example, be the time between two successive events, i.e., the Poisson process, or the time an arbitrary process needs to complete, as is the case for service patterns. This pseudo-random value is defined as an exponential random variable.

Definition 2.4.4. *An exponential random variable is a continuous random variable with:*

$$PDF: \qquad f(t) = \lambda e^{-\lambda t} \tag{2.1}$$

$$CDF: \qquad f(x) = 1 - e^{-\lambda t} \tag{2.2}$$

$$CCDF: \qquad f(x) = e^{-\lambda t} \tag{2.3}$$

where $\lambda > 0$ is a constant and $t \geq 0$

Given the memoryless property (Definition 2.4.3) and Complementary Cumulative Distribution Function (CCDF) (Definition 2.4.4), it is possible to show that the exponential distribution is in fact memoryless.

$$\begin{aligned} P\{T > t + s | T > s\} &= \frac{P\{T > t + s, T > s\}}{P\{T > s\}} \\ &= \frac{P\{T > t + s\}}{P\{T > s\}} \\ &= \frac{e^{-\lambda(t+s)}}{e^{-\lambda s}} \\ &= e^{-\lambda t} \end{aligned} \tag{2.4}$$

Theorem 2.4.7. *An exponential random variable has the memoryless property*

2.4.5 General Results

Little's Law

Little's Law is a fundamental queueing theory result that provides a relationship between the average arrival rate λ, the average time a customer spends in the queueing system W, and the average number of customers in the system L. The result is general, meaning that it applies to all possible queues given some constraints. The relationship is based on the long-run averages; hence the three values must be finite when the time and the cumulative number of customers approach infinity. Little's Law is therefore not applicable with systems where the queue size increase without bound.

Theorem 2.4.8 (Little's Law)**.** *In a queuing process, let $1/\lambda$ be the mean time between the arrivals of two consecutive customers, L be the mean number of customers in the system, and W be the mean time spent by a customer in the system. It is shown that if the three means are finite and the corresponding stochastic processes strictly stationary, and if the arrival process is metrically transitive with nonzero mean [14].*

$$L = \lambda W$$

The quantities L, λ and W are the long-run averages and are defined as follows:

$$\lambda \equiv \lim_{t\to\infty} \frac{A(t)}{t}, \qquad W \equiv \lim_{k\to\infty} \frac{1}{k} \sum_{i=1}^{k} W^k, \qquad L \equiv \lim_{T\to\infty} \frac{1}{T} \int_0^T N(t)\, dt. \tag{2.5}$$

$A(t)$ is the cumulative number of arrivals at time t, W^k is the time customer number k spent in the system, and $N(t)$ is the number of customers in the system at the time t. Note that Little's Law is an intuitive result. For example, suppose customers arrive at a system exactly every second and customers finish service precisely every second, given that the system starts with one customer in the system. In that case, there will be precisely one customer in the system at all times. However, the law is dependent on several assumptions about the system. In the example given, it is not specified that a customer must be served upon entering the system. Likewise, it is not specified that service must be completed.

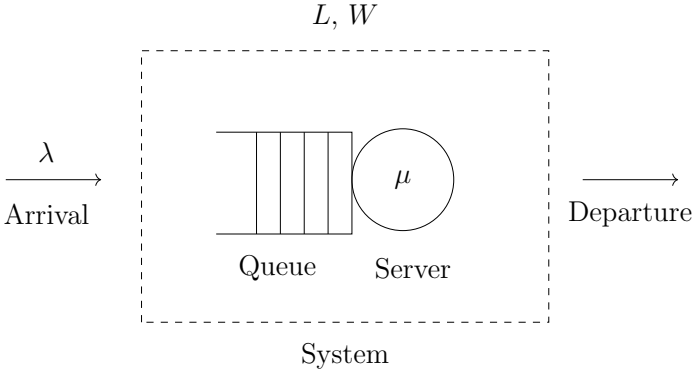


Figure 2.5: Little's Law

Figure 2.5 illustrates the law's usability, as well as its limitations. As shown, customers arrive at the system and depart from the system after a given time. No customer can leave in any other way. Thinking of the system as a black box can help in understanding its general nature. Consequently, Little's Law can be used in several different situations. Figure 2.6 and Equation 2.6 shows how the law can be used on the queue only, which will yield the time or number of customers in the queue, thereby excluding the customers being served. In fact, since Little's law is a relation among the variables L, W, L_q and W_q additional equations can be derived shown in Equation 2.7 and Equation 2.8

$$L_q = \lambda W_q \tag{2.6}$$

$$L = L_q + r \tag{2.7}$$

$$W = W_q + 1/\mu \tag{2.8}$$

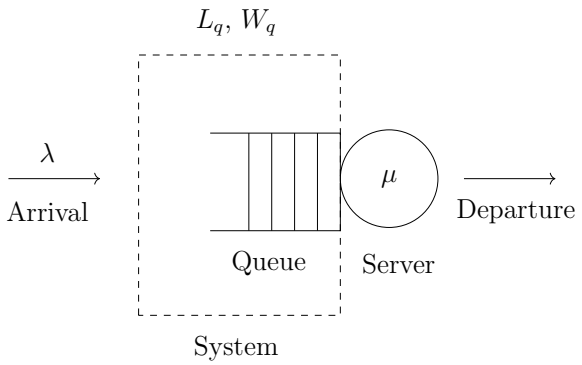


Figure 2.6: Little's Law Applied on the Queue

Lindley's equation

Lindley's equation is an essential general result in Queueing Theory. The equation provides a relation between the time spent in the queue of customer n ($W_q^{(n)}$), the time used to serve this customer ($S^{(n)}$), the time spent in the queue of the next customer $n+1$ ($W_q^{(n+1)}$), and the interarrival time between the two customers ($T^{(n)}$). However, the equation is not general when it comes to the number of servers and queueing discipline since it assumes one server and a FIFO discipline — a $G/G/1/FIFO$ queue. Note that the subtracting of the interarrival time can, if large enough, result in a negative waiting time. Since customers cannot wait a negative amount of time, a maximum of the equation and zero is performed [15]. Figure 2.7 illustrates the relation given by the equation. Formally Lindley's equation can be described as follows:

Theorem 2.4.9. *A customer arriving at a $G/G/1/FIFO$ queue will experience a waiting time equal to the max of the previous arrived customer plus the service time of this*

customer minus the interarrival time between the two customers, and zero. [15]

$$W_q^{(n+1)} = max\{W_q^n + S^n - T^n, 0\}$$

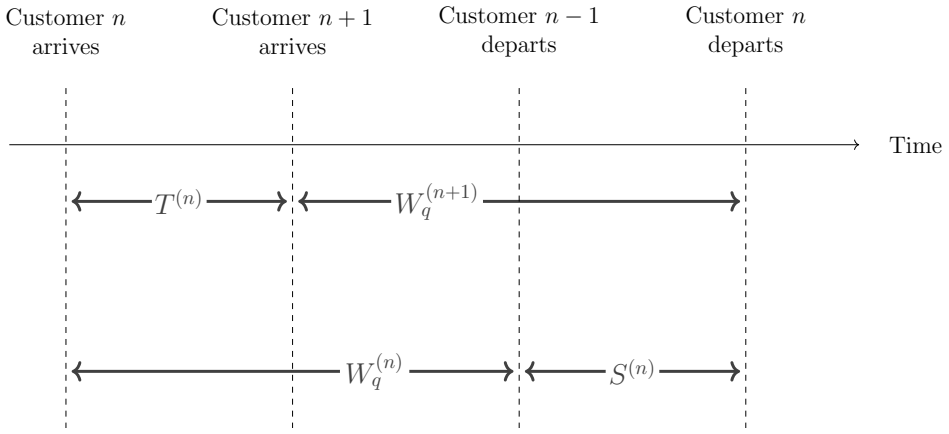


Figure 2.7: Lindley's Equation Visualized [12]

Pollaczek–Khinchine Formula

Until now, the general result has been general in both arrival and service patterns, with Little's Law being completely general and the fact that Lindley's equation is working under assumptions of a single server FIFO queue. An M/G/1 queue has general service distributing and a Poisson arrival process. This motivates the following set of formulas that can be said to extend Little's Law in an M/G/1 queue.

$$L_q = \frac{\lambda^2 E[S^2]}{2(1-\rho)} \tag{2.9}$$

$$W_q = \frac{\lambda E[S^2]}{2(1-\rho)} \tag{2.10}$$

$$W = \frac{\lambda E[S^2]}{2(1-\rho)} + \frac{1}{\mu} \tag{2.11}$$

$$L = \frac{\lambda^2 E[S^2]}{2(1-\rho)} + \rho \tag{2.12}$$

Consequently, it is possible to obtain results by only knowing the arrival process and the service process of the system. Recalling Table 2.4, the variables λ, μ, ρ, and S are all conditioned on either the service or arrival distribution. Simple justifications of the given formulas can be derived by altering Little's Law and using the PASTA property. Recall that in an arrival context, the PASTA property states that new arrivals will, on average, find the queue in the same state as the time-average state of the queue.

2.4.6 Event-Oriented Bookkeeping

Event-oriented bookkeeping is a method to secure that state change only happens when an event occurs. How queues behave in time is often the objective when analyzing a queue. Observing the arrival of customers and the time it takes to serve this customer is enough to calculate the interarrival time between customers, the time a customer starts service, the departure time of a customer, and the total time a customer used in the system (Table 2.5). However, calculating when the next customer starts service assumes a $G/G/1/FIFO$ queue since this equation uses the previous customer in service. Consequently, another observation is needed to calculate all variables in a $G/G/c$ queue.

Variable	Definition	Sample Relationship	Applicable Queue
$A^{(n)}$	Arrival time of customers n		$G/G/c$
$S^{(n)}$	Service time of customer n		$G/G/c$
$T^{(n)}$	Interarrival time between customer n and customer $n+1$	$T^{(n)} = A^{(n+1)} - A^{(n)}$	$G/G/c$
$U^{(n)}$	Time customer n starts service	$U^{(n+1)} = max\{D^{(n)},\ A^{(n+1)}\}$	$G/G/1/FIFO$
$D^{(n)}$	Departure time of customer n	$D^{(n)} = U^{(n)} + S^{(n)}$	$G/G/c$
$W_q^{(n)}$	Time customer n is in the queue	$W_q^{(n)} = U^{(n)} - A^{(n)}$	$G/G/c$
$W^{(n)}$	Time customer n is in the system	$W^{(n)} = W_q^{(n)} + S^{(n)}$	$G/G/c$

Table 2.5: Basic Bookkeeping Relations

2.4.7 Markov Chains

Markov chains are stochastic processes that describe the possible change of state in a model. They can be both discrete and continuous with respect to time. Discrete-time Markov chains can only change state at discrete points in time, unlike the continuous-time Markov chains that will remain in a state in an arbitrary amount of time before transitioning to another state. The fundamental basis of all Markov chains is the Markovian property.

Theorem 2.4.10 (Markovian property)**.** *A series of random variables is said to be a Markov chain if it has the following property:*

$$P\{X_{n+1} = j | X_0 = i_0, X_1 = i_1, \ldots, X_n = i_n\} = P\{X_{n+1} = j | X_n = i_n\}$$

This property states that the current state is independent of previous states. Knowing the probability of transitioning from the current state to any other state is the same

as knowing the whole history of prior states. There is a relation to the memoryless property 2.4.3 where the next variable is entirely independent of the past. However, Markov chains have a state, unlike the exponential distribution, which makes them more versatile. Further, the arbitrary time spent in a continuous-time Markov chain follows the exponential distribution.

Transition probabilities

The transition probabilities of a Markov chain is the probability of transitioning to a state once a state change happens. Notably, for every event in the case of discrete-time Markov chains, or after an arbitrary amount of time, as is the case for continuous-time Markov chains. Figure 2.8 is an example of a four-state Markov chain with transition probabilities. Note that the sum of all probabilities going out from a state must be equal to 1 as shown in Equation 2.13

$$\sum_{j=0}^{m} X_{ij} = 1 \tag{2.13}$$

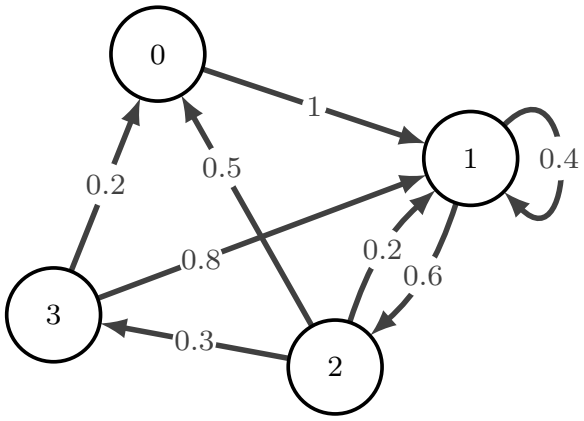


Figure 2.8: Markov Chain With Four States and Transition Probabilities

The single-step probabilities $P\{X_{n+1} = j|N_n = i\}$ is often assumed to be independent of n, which is the time of the system. In which case the Markov chain is homogeneous, and the transition probabilities can be written as $p_{ij} \equiv P\{X_{n+1} = j|N_n = i\}$. Figure 2.8 transition probabilities can be written in matrix form as shown in Equation 2.14 and is called the transition matrix.

$$\boldsymbol{P} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.4 & 0.6 & 0 \\ 0.5 & 0.2 & 0 & 0.3 \\ 0.2 & 0.8 & 0 & 0 \end{bmatrix} \tag{2.14}$$

Properties and Terminology

Although the defining property for all Markov chains is the Markovian property Theorem 2.4.10, Markov chains have several other properties or characteristics associated with them. The following definitions will be helpful in the next sections in deciding how to work with particular chains.

Definition 2.4.5 (Accessibility)**.** *A state j is* accessible *from a state i if there exist a valid path between j and i ($i \to j$).*

Definition 2.4.6 (Communication)**.** *The two states i and j communicate if i is accessible form j and j is accessible form i ($i \leftrightarrow j$). A Markov chain can have multiple* communication classes *which means that it is impossible to go back and forth between a set of states.*

Definition 2.4.7 (Reducible)**.** *A Markov chain is* irreducible *if all states communicate with each other, i.e., all states are within the same* communication class*. A Markov chain is said to be* reducible *if the opposite is true.*

Definition 2.4.8 (Recurrent)**.** *A state j is said to be* recurrent *if the probability of returning to state j, when staring at j, is 1. Formally this is defined as follows:*

$$f_{jj} = \sum_{n=1}^{\infty} f_{jj}^{(n)}$$

If $f_{jj} = 1$ the state is recurrent*. The state is* transient *if $f_{jj} < 1$*

Definition 2.4.9 (Positive Recurrent)**.** *A state is* positive recurrent *if the* mean recurrence time *m_{jj} is less than ∞.*

$$m_{jj} = \sum_{n=1}^{\infty} n f_{jj}^{(n)} < \infty$$

If $m_{jj} = \infty$ the state is null recurrent*.*

Definition 2.4.10 (Periodic)**.** *A state j is* periodic *if the chain is guranteed to transition back to state j after a fixed number of steps d, where d is the greatest common divisor of all possible m steps so that the probability of being in state j is $p_{jj}^{(m)} > 0$. A state with period $d = 1$ is* aperiodic

Long-Run Behavior

When working with Markov chains, it is often interesting to know how the chain behaves when the time approaches infinity ($n \to \infty$). The n-step transition matrix is the matrix that is formed when raising the transition matrix to the power of n, essentially finding the probability of being in certain states after time n. When n approaches infinity ($n \to \infty$), this matrix is called the limiting matrix. The limiting matrix of the transition matrix $\boldsymbol{P}$

Equation 2.14 is:

$$\lim_{n\to\infty} \boldsymbol{P}^n = \lim_{n\to\infty} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.4 & 0.6 & 0 \\ 0.5 & 0.2 & 0 & 0.3 \\ 0.2 & 0.8 & 0 & 0 \end{bmatrix}^n \approx \begin{bmatrix} 0.15 & 0.47 & 0.28 & 0.08 \\ 0.15 & 0.47 & 0.28 & 0.08 \\ 0.15 & 0.47 & 0.28 & 0.08 \\ 0.15 & 0.47 & 0.28 & 0.08 \end{bmatrix} \tag{2.15}$$

In Equation 2.15 all rows converge, meaning that in the long run, the probability of being in a state is independent of the starting state. The values can be seen as the fraction of time spent in the system states. Further, if the limiting matrix is converging with equal rows, and each row sum to one $\sum_j \pi_j = 1$, then π_j is a limiting distribution. It can be shown that instead of raising the transition matrix to a high degree, π_j can be found by solving one of the following equation:

$$\pi_j = \sum_i \pi_i p_{ij} \tag{2.16}$$

$$\boldsymbol{\pi} = \boldsymbol{\pi}\boldsymbol{P} \tag{2.17}$$

Where Equation 2.17 is the vector for of Equation 2.16

Theorem 2.4.11 (Stationary equation)**.** *An irreducible and positive recurrent discrete-time Markov chain has a unique solution to the stationary equations*

$$\boldsymbol{\pi} = \boldsymbol{\pi}\boldsymbol{P} \quad \textit{and} \quad \sum_j \pi_j = 1 \tag{2.18}$$

Furthermore, if the chain is aperiodic, the limiting probability distribution exists and is equal to the stationary distribution.

Another long-run behavior concept is the idea of ergodicity. A process is ergodic if the ensemble average of a process is equal to the average of a process. Intuitively, if measuring an infinite number of processes at a fixed point is equal to the average of a process, then the process is said to be ergodic. Often, processes are time-dependent, and so the approximation is a time average. In which case, ergodicity can be defined as follows:

Definition 2.4.11 (Ergodicity)**.** *A time-dependent process $X(t)$ is ergodic if time averages equal ensemble averages.*

The properties defined in section 2.4.7 decides some resulting properties and how to use the stationary distribution. If the Markov chain is irreducible and positive recurrent, there exists a unique stationary distribution as defined in Theorem 2.4.11. Further, if $\boldsymbol{\pi}^0 = \boldsymbol{\pi}$, i.e., the starting probability vector is equal to the stationary vector, then the process has a unique stationary distribution, and the process is stationary and ergodic. If, on the other hand, the chain is irreducible, positive recurrent, and aperiodic, there exist a limiting distribution equal to the unique stationary distribution. The process is also ergodic.

2.5 Discrete Event Simulator

Discrete event simulation is described in the book Discrete-Event System Simulation [16] to be a way of simulation a system in which the state of the system only changes at discrete points in time. These state changes are determined by the so-called events running in the simulator, hence the name discrete event simulator. Events are also tasked with creating and scheduling new events to run in the simulator at a later time. Since an event can create multiple events, a call queue is needed to store and order future events. Simulated time is often used to order the event; however, real-time is also used. [17]

The simulator output is composed of measurements gathered during the simulation, acting as an artificial history of the system generated by model assumptions. These measurements are used to analyze the performance of the modeled system, which in turn is used to estimate the true performance of the system of interest. Generally, simulation is used when it's impractical to experiment on the real system. There may be too high a risk associated with the experiments, or it is impossible to acquire the needed results from the experiments. Regardless of the reason, simulation can be an efficient and practical way of experimenting since all unknown factors can be eliminated, giving precise and reproducible results. Simplified assumptions can, however, lead to misleading results. A good model is therefore key to obtain usable results. [16]

2.5.1 Advantages

The advantages of using discrete event simulation, or just simulation, to analyze a system are many. Two of them are, as mentioned, the potential risk associated with performing tests on the real system and the feasibility of acquiring the needed results from the experiments. Simulation allows for risk-free testing of a system with control over all parameters. Further, insight can be obtained about the interaction and importance of variables in the system and how it affects performance. In addition, questions concerning potential changes to the system can be answered without the need to implement them. In summary, a simulation is a versatile tool that allows the user insight into every aspect of the simulated process. [16]

2.5.2 Disadvantages

Simulating has some clear disadvantages that affect the way one should interpret the obtained results. Simulators are used when complicated systems are analyzed. As a consequence, many of the processes happening within a system are too complicated to model correctly. This motivates the use of random components that can model the seemingly random processes that cannot be modeled effectively. Therefore it can be hard to know if the results are caused by inherent randomness or by interrelationships. Simulation cannot identically replicate the real-world environment. Consequently, the results must be analyzed with this in mind [16]

Chapter 3

Related Work

This chapter will introduce some of the related work previously done in the field of MPTCP and present the contribution of this thesis.

3.1 MPTCP Research

The effort towards developing MultiPath TCP is an active research topic with many issues that need solving to ensure good throughput and efficient use of the added available bandwidth. Ideally, the throughput of MPTCP should be the added potential throughput of all available TCP flows. In reality, this is is not achieved and can, in some cases, underperform the throughput of a normal TCP connection. This section will shed light on some of the work done in this field.

The Linux kernel implementation created by C. Paasch, S. Barre, et al. [18] is an effort to enable the simultaneous use of several TCP interfaces. This implementation is aiming at providing MPTCP in a way that resembles TCP to the application layer, while in fact, sending data across multiple paths. The project is a product of the research being done in the field, and the implantation is actively worked on.

Fountain Code-Based MultiPath Transmission Control Protocol (FMTCP) is proposed [19] to tackle the fact that subflows experiencing high loss and delay are becoming bottlenecks of the MPTCP connection. The researchers conducted an extensive simulation-based study, which found that the heterogeneity of paths can cause a significant reduction in the overall performance. The proposed implementation takes advantage of fountain codes to mitigate the negative impact of the heterogeneity of the different paths.

A MPTCP implementation using network coding for mobile devices in heterogeneous networks is presented in this paper [20]. Empirical data was collected to help understand the mobile environment when three heterogeneous networks are available to the mobile device. WiFi, WiMax, and an Iridium satellite network are used to gather the data. Further, a reliable multi-path protocol called MPTCP with Network Coding (MPTCP/NC)

is proposed. The results compared to MPTCP without network coding show that the proposed implementation gives users a higher quality of service since it can overcome packet losses due to lossy wireless network connections.

A research group has presented the experience in implementing MPTCP [11]. The paper focuses on receive buffer tuning and segment reordering to efficiently send data along different paths with different characteristics. The results are then compared to regular TCP. Based on the results and the improvements made through new algorithms that solve important MPTCP problems, it is concluded that MPTCP is ready for adaptation.

The performance of MPTCP over heterogeneous subpaths is the topic of another research paper [21]. The researchers are exploring the performance of MPTCP over real-world networks. The results mainly point to obstacles that need to be addressed if MPTCP is to be widespread. In addition, the results show how heterogeneous paths can reduce the overall performance of MPTCP.

The way packets are scheduled across the available subpaths is important for the performance of a MPTCP implementation. This paper [22] propose two schedulers that aim to optimize the latency to match an applications desired latency.

The asymmetric nature of heterogeneous paths is causing problems such as the out-of-order problem and the load imbalance problem. This paper [23] is proposing a Receive Buffer Pre-division based flow control mechanism (RBP) for MPTCP as an alternative to scheduling algorithms. The control mechanism divides the receive buffer according to the subflows characteristics and controls the data transmission in each subflow. The NS-3 simulator is used to verify the scheme, which produces results that show a significant increase in throughput.

3.2 Contribution

The focus of this thesis is to analyze a simulated implementation of MPTCP in various situations. The results obtained from the simulator can hopefully give valuable insight into problems and possible solutions. As part of this thesis, a discrete event simulator was developed to simulate a network in an event-based fashion. The TCP and MPTCP implementations were tested using the simulated network, and the results are analyzed and compared using results from queueing theory.

Chapter 4

Method

This chapter will describe how the discrete event simulator, developed as part of the work behind this thesis, is implemented and explain the simplifications and assumptions needed to understand the obtained results. The simulator is used to analyze simulated implementations of both TCP and MPTCP in various situations.

4.1 The Discrete Event Simulator

The simulator created for this thesis is written in java using an object-oriented programming paradigm to structure the code as objects in a similar manner to how the internet actually looks like. For this reason, logic is implemented in the various classes:

- Router
- ClassicTCP
- MPTCP
- Channel
- Packet

In addition, the Router, ClassicTCP, and MPTCP classes are implementing the NetworkNode interface that defines the common methods needed to communicate with each other through Channel objects. Furthermore, the ClassicTCP and MPTCP classes are implementing the TCP interface as well to define the common methods needed in TCP communication. The Packet and Channel classes are not implementing any interface but are used throughout the simulator.

The use of a discrete event simulator allows for reproducible results as it is deterministic if the random sequence is known. This is achieved by using a single random generator with a pre-defined seed. The reproducibility of the results is key during analyses and for the reader to verify the obtained results. Further, real-time is not a factor in the simulator,

nor are hardware concerns which could be a decisive factor in other simulator techniques. To run the simulator, the EventHandler class employs a PriorityQueue that sorts Event objects in simulated time. Together these classes form the core of the simulation loop and will be described further in the following sections.

4.1.1 Events

Events are created to run the objects in the simulator. This ensures a distinction between the simulator and the simulated network. The objects public methods are used to run the objects as well as generating new events that are added to the call queue. This is structured around two main methods that all events are implementing, namely the run method and the generateEvent method (see Algorithm 1). The objects are responsible for the actual state changes that happen in the run method. In other words, the state of the system is stored in the networking objects, while the events only call the objects and enqueue new events based on the feedback from the objects themself.

4.1.2 Event Handler

An event handler is the main controller of the simulation. It employs a call queue that keeps track of the next events to be called as well as storing future events. The queue is implemented using the standard PriorityQueue provided by java, with events sorted in time. Events are given timestamps (t) upon creation which is based on the objects delay-method (d), as well as the simulation time (T).

$$t = T + d$$

This operation ensures that events are scheduled in the correct order based on the type of event added and the time of creation. Further, if the next event is scheduled with a long delay, i.e., a long simulated time to the event should occur, the simulator can jump directly to this event without any waiting needed (see Algorithm 1 and Figure 4.1). This is doable due to the use of simulated time in place of actual time.

Algorithm 1: Main simulation loop

```
Function run()
    while True do
        var event = eventQueue.poll();
        if event == null then
            return;
        Util.tickTime(event);
        event.run();
        event.generateNextEvent(eventQueue);
```

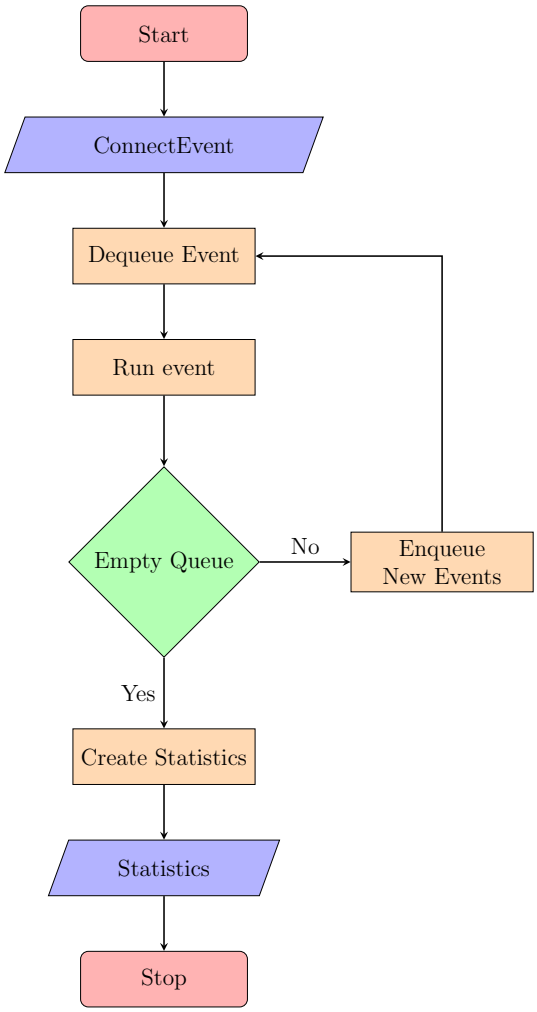


Figure 4.1: Simulator Flow Chart

4.1.3 Simulated Time

The simulator employs a logical clock in place of a physical clock. This logical clock ticks based on events and has no relation to real-world time. A logical clock is useful in simulation as hardware speeds compared to real-world time are eliminated, making results reproducible using different hardware. However, the delays described in the literature are based on real-world time. The equivalent to a real-world microsecond is set to 1 simulated time, meaning that the time unit used is microseconds. This is done in the effort to scale the delays according to each other. Consequently, all delays associated with events can be set to approximations of values in microseconds found in the literature.

Each time the event handler runs an event, the event timestamp is set to the new simulated time as shown in the main simulation loop algorithm 1. This ensures that events happen at the right time, as well as eliminating the possibility of scheduling events in the past. This also makes it easy to introduce randomness in the model without dramatically increasing

the complexity of the model [24].

4.1.4 Random Generator

A random generator is used throughout the model and is the only source of randomness. Randomly generated input is needed in modeling both the queue sizes and the loss in the model. It is important to generate the same random sequence in order to create reproducible results. This is achieved by using seeds given to the random generator and, in that way, reproducing the same pseudo-random generated sequence.

4.2 Router Modeling

The Router objects are tasked with the routing of packets, i.e., finding the right channel to send a Packet object through, as well as modeling the queuing and transmission delay. The actual routing operation is not important for the tests conducted in this thesis, as the networks created are linear paths. However, larger simulated experiments that needed this functionality were planed but are left for future work. Nevertheless, the queuing and transmission delays are what affect the output from the simulation. The following sections will describe how the two delays are modeled.

4.2.1 Queue

The presented model is using a simple Poisson process to model the router queue sizes to allow for comprehensible results. However, it is widely accepted that a Poisson process cannot accurately model the arrival rate of internet routers. The traffic seen in routers has a Long-Range Dependence (LRD) which means that the traffic has some sort of memory [25]. Considering that much of the traffic comes from TCP connections, it is not surprising that the perceived traffic distribution has some kind of memory. The TCP congestion control mechanisms are actively adjusting the sending rate to maximize throughput while not contributing to congestion. However, modeling the queues of routers has been shown to be difficult. Models that promise LRD properties and a closer fitting to real-life router queue sizes are generally complicated. They may have a lot of parameters and may only fit certain scenarios well [25]. Too realistic assumptions and complex models can make the analysis difficult.

Each time a new packet is arriving the modeled routers, the experienced queue size is equal to the Poisson random variable given by the Poisson process. Recall the PASTA property discussed in section 2.4.3. Since the Poisson process possesses this property, it is suitable to only generate the queue size when new arrivals from the modeled TCP endpoint arrive. That is, there is no need to keep track of the queue size in times outside of interesting arrivals because of the PASTA property.

4.2.2 Transmission and Queue Delay

There are three sources of delay that take place in the router of a network: transmission delay, queueing delay, and processing delay.

The transmission delay is the time it takes to push the bits in one packet onto the channel and is typically in the order of microseconds to milliseconds. The delay is proportional to the size of the packet L and the transmission rate R of the link, that is L/R [2]. The model does not differentiate the size of packets, nor does it have transmission rates. For this reason, the transmission delay is set to be a constant of 10.

The queueing delay is where the noticeable variation in packet delay originates from. This delay is the duration of time a packet waits in the queue while other packets are being transmitted. Arriving packets can experience queue sizes from zero to full; in the latter case, packets are dropped or lost. The varying queue size causes varying delays from zero to several milliseconds. For this reason, it is an interesting and important part of the simulator. Given the queue size experienced by the arriving packet $L_q^{(p)}$ and the previously mentioned transmission delay S the delay is set to be:

$$Delay = S + L_q^{(p)} \cdot S$$

where $L_q^{(p)}$ is modulated as a Poisson process with $\lambda = 100$.

The processing delay is commonly neglected because of the small, approximately constant delay in the order of microseconds or fewer [2]. However, in routers that perform complex payload modifications it can be of significance [26] [27]. This model will nonetheless assume a simple packet forwarding network, so the processing delay is omitted in the model.

4.3 Channel Modeling

The responsibility of the Channel objects is to deliver a Packet object to and from the linked NetworkNodes, in addition to model loss and propagation delay of the sent Packet object. NetworkNodes are objects that implement the NetworkNode interface, i.e., Router, ClassicTCP, or MPTCP objects. Loss and propagation delay manifest themselves in the output of the simulation and will be described in the following sections.

4.3.1 Propagation Delay

Propagation delay is the delay associated with a channel. This delay is closely approximated to be constant and is directly dependent on the length of the channel and the propagation speed of the link. The medium in which signals are sent is what decides the propagation speed, where fiber optics are the fastest. Regardless, the speed is a little less than the speed of light. In wide-area networks, propagation delay is said to be in the order of milliseconds. [2]. The model has a propagation delay set to be between 10

and 1000, depending on the cost associated with the channel. The cost is a parameter of the channel and can be thought of as both length and medium combined, where a high cost leads to a longer propagation delay. Although it is hard to exactly carry over the propagation delay to a simulation with a simulated time, the delay should be appropriate compared with the other delays in the network.

4.3.2 Packet Loss

Packet loss occurs when one or more packets of data fail to reach the destination. Loss can be caused by routing changes, link degradation, or congestion in the network [28]. The particular types of loss are often experienced as burst loss, i.e., loss follows loss with a high probability.

A 2-state hidden Markov model is used to model the loss that occurs in a channel, namely the Gilbert-Elliot Burst Model [29] [30]. This model is widely used for describing error patterns in transmission channels due to its flexibility and simplicity. The Gilbert-Elliot burst model achieves its burstiness by semantically having a "Good" and a "Bad" state (see Figure 4.2). Being in the good state means that loss rarely happens while being in the bad state means that loss occurs with a higher probability. Also, the probability of transitioning from the good state to the bad state is relatively low compared to the probability of going from the bad state to the good state, as shown in the figure. The properties as mentioned above, along with the Markovian property Theorem 2.4.10 gives the model its characteristics which have shown to be appropriate in emulating packet loss patterns [31] [32].

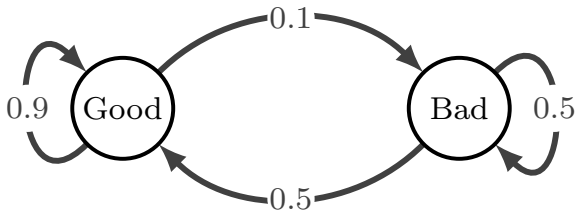


Figure 4.2: Gilbert-Elliot Burst Model

Implementation

Considering the simplicity of the Gilbert-Elliot burst model, the model is implemented with a simple check of the current state of the channel (see Algorithm 2). If the channel is in the Good-state, the channel loss parameter is used for both the transition probability of the Markov chain as well as the probability for loss. If the channel is in the bad state, then there is a 20% chance to stay in the Bad-state as well as packets being lost. This motivates the following transition matrix:

$$\begin{bmatrix} 1-p & p \\ 0.8 & 0.2 \end{bmatrix} \tag{4.1}$$

Where p is the *loss* parameter in each channel in the network, hence, it is possible to calculate the fraction of time spent in the different states by using the known *loss* and the stationary equations Theorem 2.4.11. This result can again be used to calculate the expected loss per channel while having a burst loss model.

Algorithm 2: Gilbert-Elliot burst model implementation

```
Function loss()
    if goodState then
        goodState = Random.double() ≥ loss;
        return Random.double() < loss;
    goodState = Random.double() ≥ 0.5;
    return Random.double() < 0.5;
```

4.4 TCP Modeling

TCP modeling is achieved by implementing a class called ClassicTCP that employs the needed functions to simulate a TCP communication. Some of the logic concerning sending and receiving are separated out as own classes to reduce the complexity of ClassicTCP class. The following sections will describe how the ReceivingWindow and the Sending-Window are implemented and how this affects the results, as well as describe the delays defined in the Implementation of ClassicTCP.

4.4.1 Receiving Window

The receiving window implements the Flow Control mechanism, Selective Repeat ARQ, in the TCP model. The TCP objects in the model hold a receiving window that should be interchangeable with various ARQ schemes; however, only the Selective Repeat ARQ is implemented. The receiving window's responsibility is to check if incoming packets are ordered and if the packet should be stored in the buffer. Also, it is deciding which packet to acknowledge. If a packet is out of order but inside the defined receiving window, the packet is stored in the buffer. This can happen in the model due to packet loss. When the packet or packets blocking the stored packet arrives, the stored packet is moved from the buffer to the list of received packets. A single ACK is sent, acknowledging the last packet added to the received packet list, i.e., acknowledging the stored packet if it was the only stored packet.

The TCP model also holds a sending window that also should be interchangeable with other ARQ schemes. This object is created when a connection is made and controls how fast TCP should send packets. Since the receiving window capacity is the upper bound of packets possible to send without receiving an acknowledgment, the receivers receiving window capacity is communicated during the connection phase. This ensures that the receiving window capacity is not exceeded.

4.4.2 Retransmission

Retransmission of packets is performed if the transmitter receives three Duplicate Acknowledgements or through a Retransmission Timeout (RTO). Usually, the TCP endpoints are run as events when packets arrive; however, retransmissions are events that should occur whiteout a packet triggering the event. Consequently, a retransmit event is scheduled with a delay equal to the RTO, which checks if the packet should be retransmitted or not. The event then triggers a retransmit if necessary. In contrast, retransmissions caused by duplicate ACKs can fast-retransmit the packet in the event triggered by the third DupACK.

RTO

RFC6298 [33] describe the initial calculation of the RTO value as follows:

$$SRTT = R \tag{4.2}$$

$$RTTVAR = \frac{R}{2} \tag{4.3}$$

$$RTO = SRTT + max\{G, K \cdot RTTVAR\} \tag{4.4}$$

Where R is the initial RTT measurement, $SRTT$ is a smoothed round-trip time, $RTTVAR$ is round-trip time variation, G is a clock granularity, and K is set to 4. The initial value of the RTT and RTO is calculated during connection, but in contrast to the real-world implementations, the RTT and RTO values are not updated. The $SRTT$ and $RTTVAR$ are consequently not needed. The clock granularity G is neglected, so the RTO is:

$$RTO = R + 4 \cdot \frac{R}{2} \tag{4.5}$$

$$RTO = 3 \cdot R \tag{4.6}$$

The RTO of the model is, for this reason, set to $3 \cdot RTT$.

4.4.3 Sending Window

A sending window is created when a connection is made to match the ARQ scheme of the receiver and to operate the congestion control mechanisms. The TCP model simply calls the sending window to send the appropriate number of packets according to the congestion control mechanism and the ARQ scheme which is operated within the sending window. Consequently, the sending window operates the sender-side TCP flow control mechanism, as well as the congestion control mechanism, which is implemented only on the sender's side.

Congestion Controll

The Congestion Control mechanism adjusts the sending window size or Congestion Window (CWND), effectively adjusting the sending rate. The adjustments are made according

to the ACKs received or by the absents of receivedACKand the eventual retransmit caused by the RTO. Therefore the model allows for communication between the sending window and the receiving window objects. Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery are all implemented, along with the possibility of using the Tahoe or Reno implementation

4.4.4 Transmission Delay

The transmission delay of the modeled TCP is set to be the same as for the routers. Since there is no concept of transmission rates, the transmission delay should be set equal to how to router's transmission delay is set. Further, since all packets have the same size, the transmission delay is constant at both routers and the TCP endpoints. The transmission delay is set to a constant of 10. 4.2.2

4.5 MPTCP Modeling

The MPTCP model must be able to send TCP-like traffic along different paths. For this reason, it was decided to implement the modeled MPTCP using The ClassicTCP class as the subflows, i.e., separated TCP flows per path. This approach is somewhat similar to using an existing TCP implementation on multiple interfaces. However, some modifications to the ClassicTCP implementation were needed. All the ClassicTCP subflows share a receiving window, while the sending window is isolated in the ClassicTCP subflows to separate the congestion control mechanisms as much as possible. The following sections will describe how the MPTCP implementation differs from the ClassicTCP implementation.

4.5.1 Sending Window

Since each subflow in MPTCP is operating on different paths, Each TCP subflow in the model has its own congestion control mechanisms. If congestion is detected in one path, it should not directly affect the sending rate of the other active subflows. The congestion control mechanisms are implemented in the sending window, so it follows that each TCP subflow in the MPTCP model has its own sending window. The congestion control mechanisms are therefore responding to ACK and RTO in the same way as the normal TCP implementation. In fact, using real-life excising TCP implementations as subflows limit the way data can be balanced over the available subflows. Packets are retained in the sending buffer until ACK is received and cannot directly be moved from one subflow to another to offload a subflow if serious congestion is detected [11].

Packet scheduler

The model does not implement a scheduler as is common in many MPTCP implementations [11] [21] [22]; however, the TCP subflows are all sending packets from a common list of packets to send. Subflows with shorter RTT and better throughput will consequently

send more packets, and the other subflows are offloaded. Then again, since all packets should be received in order at the shared receiver window, the lack of scheduler can result in sub-flows being bottle-necked by other slower subflows, i.e., forced to wait on a packet from another subflow.

4.5.2 Receiver Window

The receiver window is shared between all the subflows to ensure that packets are received in order. The Linux kernel implementation does the same, where all subflows compete against each other. This can cause unreasonable distribution of the receive buffer, which is undesirable. For this reason, a pre-division of the receive buffer is performed in the model [23]. However, unlike other pre-division methods that use the RTT to divide the receiving window, this model assigns equal receiving window size for all subflows.

4.5.3 Delay

The transmission delay is set to be the same as for the standard TCP and Router implementations, since the subflows of the MPTCP model are TCP subflows. which is a delay of 10. The processing delay is throughout this model neglected. This may be a simplification which is not suitable when analyzing MPTCP as the receiver window can, with benefit, be implemented with more complexity. Packets in normal TCP are assumed to be received in order [11], this assumption is not valid in the case of MPTCP where packets often arrive out of order. This can cause higher processing delay, and more complex data structures can help this problem. The processing delay is nevertheless neglected.

4.6 Collecting Results

To collect results, a statistics class is used to do event-based bookkeeping as described in subsection 2.4.6 and Table 2.5 To retrieve results from the simulation runs. The arrival and departure times are collected and used to calculate the interarrival times of packets, the time spent in the system per packet, and the number of packets in the system. Since the service time, i.e., the transmission delay is a constant 10, this is not collected. Also, the time in the system and time in the queue of a packet is only differing with a constant 10, so the time in the queue is not needed. Additionally, the time service starts (U) is reformulated as $U^{(n)} = D^{(n)} - S^{(n)}$ and is used to calculate the time in system using just departure and arrival time Table 4.1. Further, graphs are made to visualize arrival and departure times, the number of packets in the system, the time of packets in the system, and the interarrival times. In addition, TCP statistics such as goodput, which is the number of useful packets delivered in per time unit, and loss rate are collected, along with a graph of the sender's congestion window.

Variable	Definition	Calculation
$A^{(n)}$	Arrival time of packet n	Collected
$D^{(n)}$	Departure time of packet n	Collected
$L^{(t)}$	Number of packets in the system at time t	Collected
$S^{(n)}$	Service time of packet n	Constant 10
$T^{(n)}$	Interarrival time between packet n and packet $n+1$	$T^{(n)} = A^{(n+1)} - A^{(n)}$
$U^{(n)}$	Time packet n starts service	$U^{(n)} = D^{(n)} - S^{(n)}$
$W^{(n)}$	Time packet n is in the system	$W^{(n)} = D^{(n)} - A^{(n)}$
$W_q^{(n)}$	Time packet n is in the queue	$W_q^{(n)} = W^{(n)} - S^{(n)}$
T	Number of packets sent (Transmitted)	Collected
RT	Number of packets retransmitted	Collected
FRT	Number of packets fast retransmitted	Collected
Goodput	The information throughput	$Goodput = T/t$
Loss rate	The perceived loss	$Loss\ rate = (RT + FRT)/T$

Table 4.1: Collected and Calculated Results

Chapter 5

Results

In this chapter, the results gathered from the simulation are presented. The setup of the various tests is described in detail before the results from the simulated TCP and MPTCP implementations are presented.

5.1 Setup

The tests which led to the results are performed using equal values of the parameters throughout the different experiments with the exception of the paths being different as well as the use of both TCP and MPTCP. In this way, the importance of the parameters that changed from test to test could be observed. All routers are using a Poisson queue length process with different means to simulate the queues and consequently the queueing delay per packet. During development, it was found that 2000 packets is a good amount of packets to send since it is enough that loss can occur, even in environments that have a low probability of loss, and allows for clearer graphs as more packets will compact the timeline and consequently obscure the results. Further, it simulates downloading a file of size roughly equal to 2-3GB, if the average packet size is assumed to be 1300 byte [34]. Even though the measure of packet size does not affect the simulator in any way, it can give the reader and the results something tangible to relate to. For this reason, the average packet size of 1300 byte is used whenever packet size is referenced in the results. The simulated time is divided by 1000, making the results a measure in "milliseconds" instead of "microseconds" to make the results more readable. Additionally, only the TCP Reno implementation is presented since the Tahoe and Reno results did not differ in a substantial way.

5.1.1 Path Variations

Four variations of paths is the main difference between all the test done on MPTCP and TCP. Each path has its own properties that will show how the implementations work in different environments. The properties that are different in the four path variation are the

length and loss parameters, giving four different variations. When testing the MPTCP implementation, the heterogeneous path tests are using combinations of the declared tests, while the homogeneous tests are using two of the same paths. The results can, in turn, be compared to the TCP implementation used on the same path. The properties of the variation of the paths will be presented in the following sections.

Short Path Low Loss

This path is using four routers, with router 1 and 2 having a 50% queue utilization while router 2 and 3 have a 98% queue utilization. The loss parameter is set in only one channel to 0.001, which should give a loss of a little more 0.12% when calculating the fraction of time being at the two-state the Gilbert-Elliot channel Equation 4.1 and multiplying the probability of loss in each state.

$$0.999 \cdot 0.001 + 0.001 \cdot 0.2 = 0.001199 \tag{5.1}$$

$$\approx 0.12\% \tag{5.2}$$

However, in the results that will be presented, the loss will be higher since this is the implementation's perceived loss. This is the number of retransmissions divided by the number of packets sent as shown in Table 4.1. The cost of the channels is set to 1 in the first and last channel, while the inner channels have a cost of 10.

Short Path High Loss

This path is equal to the short path with low loss (section 5.1.1) except that the loss parameter is set to 0.01 giving the channel a loss of approximately 1.2% when taking the Gilbert-Elliot burst channel into consideration.

$$0.988 \cdot 0.01 + 0.012 \cdot 0.2 = 0.01228 \tag{5.3}$$

$$\approx 1.2\% \tag{5.4}$$

Long Path Low Loss

This path is set up using ten routers, with the outermost routers at both ends being at 50% queue utilization while the rest is at 98%. This makes it similar to the short path but with extra routers in between. Additionally, channels 1 and 11 have a cost of 1, channels 2 and 10 have a cost of 10, while the rest have a cost of 50, making the path substantially longer. Further, 3 of the channels connecting the inner routers are lossy, in contrast to the single lossy channel in the short paths. This loss is set to the same as in the short path low loss test, but this time in 3 channels making it more probable that loss will occur. Using the Gilbert-Elliot burst channel transition matrix from Equation 4.1 and the stationary equations Theorem 2.4.11, the fraction of time in each state is found. Further, since loss can occur three places on the path, the overall probability of loss is the reverse of not having loss at all:

$$1 - (0.999 \cdot 0.999 + 0.001 \cdot 0.8)^3 = 0.00359 \tag{5.5}$$

$$\approx 0.36\% \tag{5.6}$$

The higher loss compared with the short path with low loss section 5.1.1 is expected in real life since longer paths have more possibilities of experiencing interference, congestion, and other situations leading to loss.

Long Path High Loss

This path is equal to the long path with low loss section 5.1.1 with the exception of the loss parameters. Instead of having three channels with the same loss of 0.001, two of the channels now have a loss of 0.001 and one with a loss of 0.01, which is the same loss as in the short path high loss section 5.1.1. The loss for this channel is:

$$1 - ((0.999 \cdot 0.999 + 0.001 \cdot 0.8)^2(0.988 \cdot 0.99 + 0.01 \cdot 0.8)) = 0.01465 \tag{5.7}$$

$$\approx 1.5\% \tag{5.8}$$

5.1.2 Simulation Results

General results are presented in separate tables per test. The arrival plot, departure plot, interarrival plot, number of packets in system plot, time in system plot, and the congestion window size plot are all graphs that visualize how the implementation behaves in time in the given environment. The arrival plot and departure plot are describing the arrival and departure time of the n-th packet. This is interesting because it highlights the fact that arrivals and departures are essentially the same in TCP. The interarrival plot is plotting the interarrival time between packets received. This is interesting because it should be easy to spot when loss occurs as well as seeing the effect of the sending window. The number of packets in the system and the amount of packets in the system is of interest because it shows how full the receiving window is, as well as highlighting the effect lost packets has on the time out of order packets are held in the system. Lastly, the congestion window plot is showing the congestion control mechanisms. In light of the other graphs, the congestion window is especially interesting as it is a key part of how TCP works. The MPTCP results are presented as separate subflows since the performance difference or lack of difference is interesting to analyze. Consequently, there are two tables and double the number of graphs when MPTCP results are presented.

5.2 TCP

Tests were performed on simulated environments to validate the TCP implementation used in the MPTCP model. The tests included environments with high and low loss, as well as long and short paths, resulting in four tests described in the following sections.

5.2.1 Short Path Low Loss

This test is using the declared short path with low loss section 5.1.1. From the graphs, it can be seen that loss was detected two times. The congestion window is perhaps the best

indicator Figure 5.6. The first experienced loss is serious enough to trigger the RTO seen by the appearance of the slow start mechanism, i.e., an exponential growth until ssthresh is reached. The second experienced loss is not as serious as seen by the lack of slow start, but rather the fast recovery mechanism being initiated due to 3 or more DupACKs. All the other graphs are observations made at the receivers end and are especially interesting to look at in the light of the sender's congestion window.

From the graphs listed below, it is clear that loss is a vital part of the performance of TCP. In Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, and Figure 5.6, a packet loss is evident at the same simulated time represented by the x-axis. Given the loss of 0.12%, the expected amount of packets to be lost when 2000 packets are sent is $2000 \cdot 0.0012 = 2.4$. Looking at the graphs, at least two loss bursts can be seen.

Variable	Result
λ	3.093 t
W	0.053 t
L	0.164 *Packets*
Packets Sent	2000 *Packets*
ACKs Received	1804 *Packets*
RT	32 *Packets*
FRT	9 *Packets*
Goodput	3.641 *Packets*/t
Loss rate	2%

Table 5.1: TCP Short Path Low Loss Results

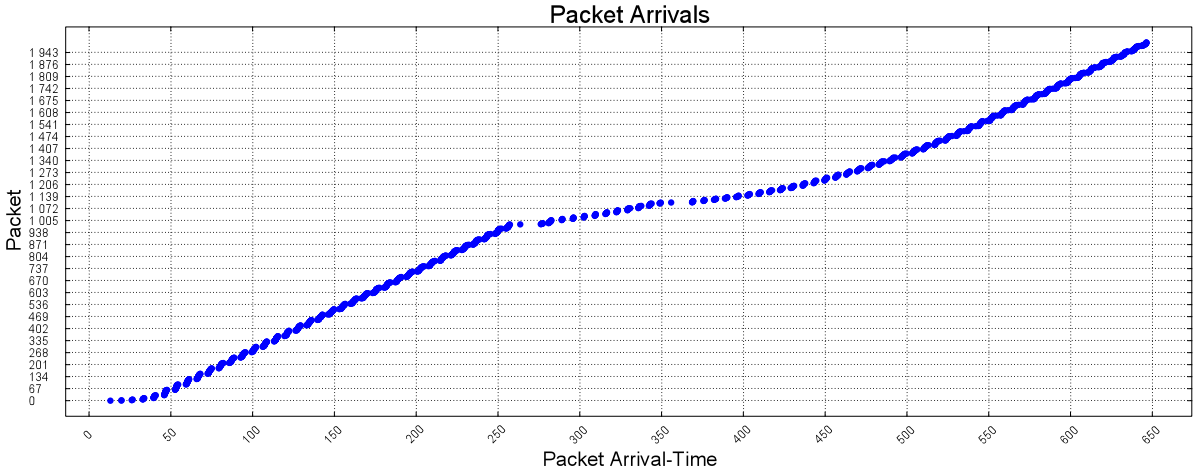


Figure 5.1: TCP Short Path Low Loss Packet Arrivals

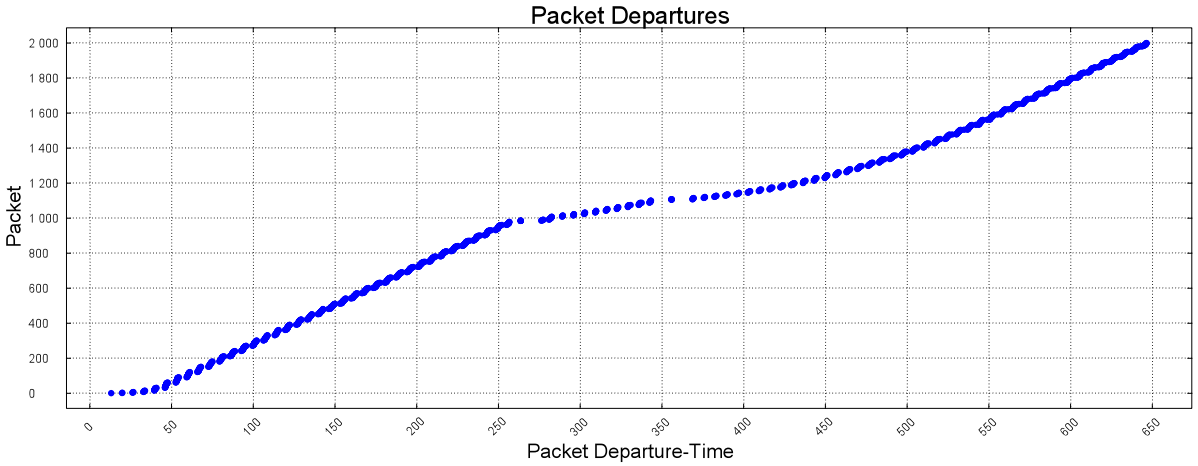


Figure 5.2: TCP Short Path Low Loss Packet Departures

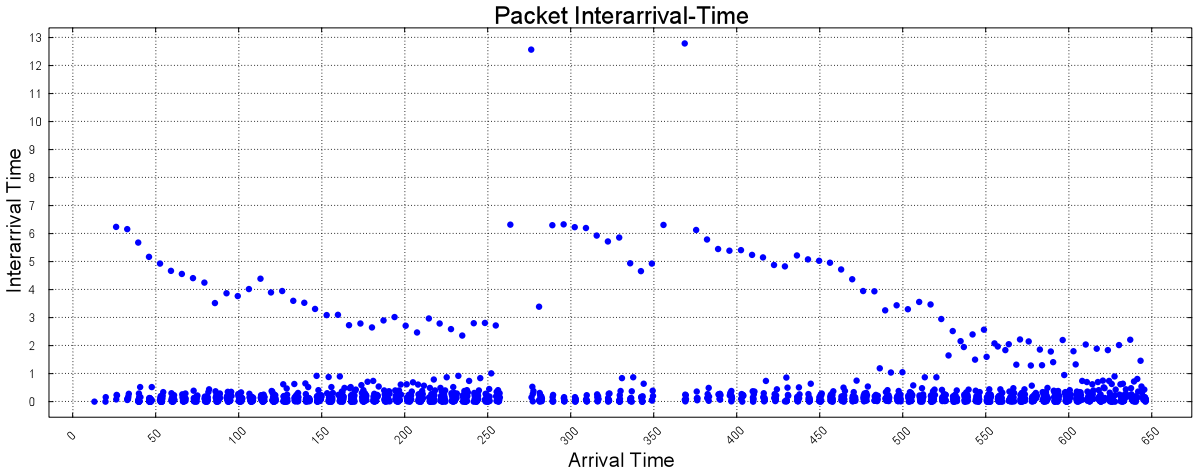


Figure 5.3: TCP Short Path Low Loss Packet Interarrival Time

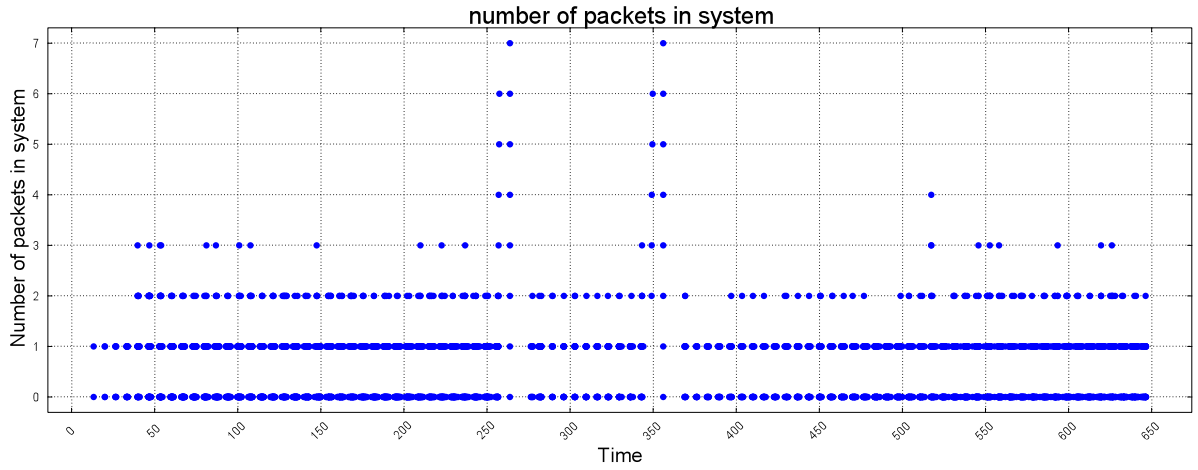


Figure 5.4: TCP Short Path Low Loss Packets in System

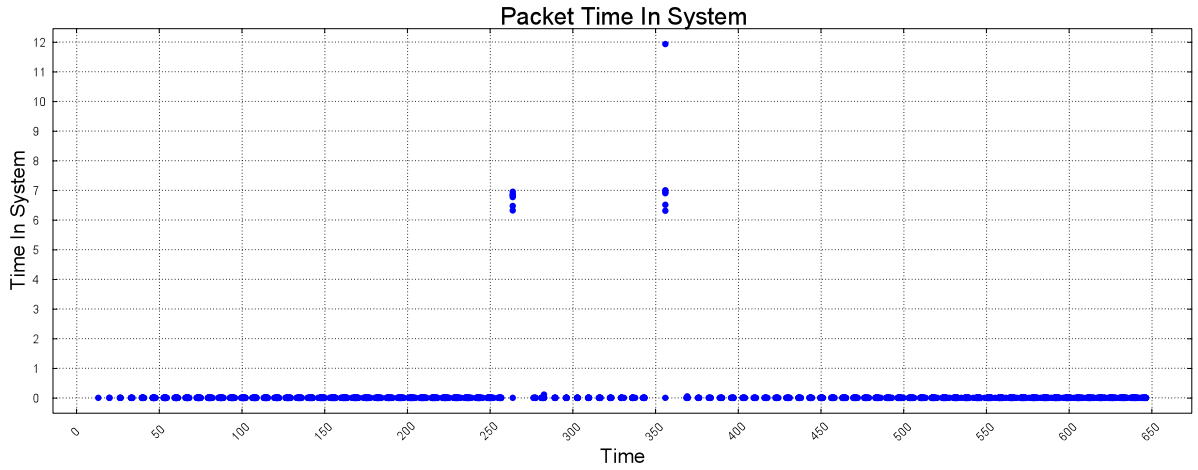


Figure 5.5: TCP Short Path Low Loss Packet Time in System

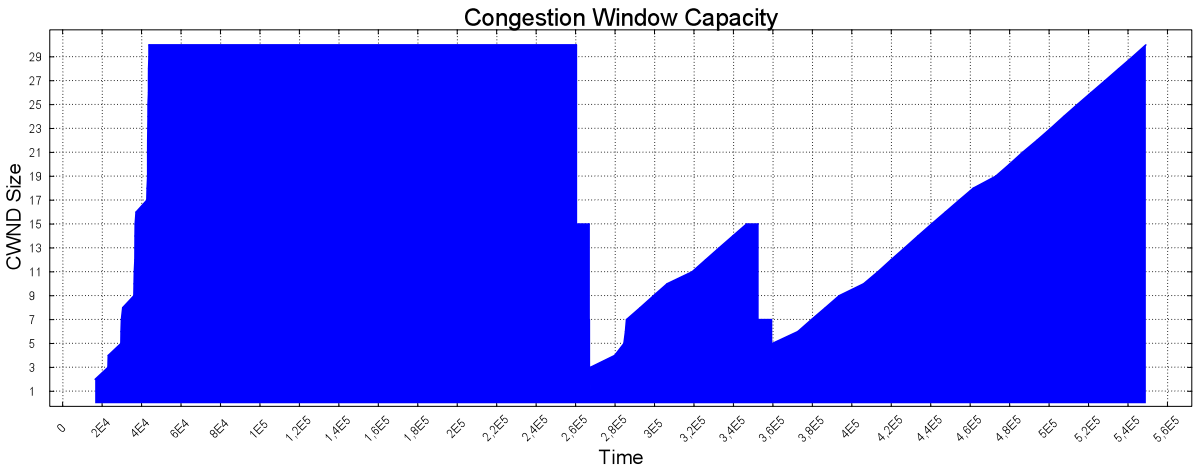


Figure 5.6: TCP Short Path Low Loss Congestion Window

5.2.2 Short Path High Loss

This test is using the short path with high loss section 5.1.1. Comparing this test with the test with low loss, the congestion window in this test does not stabilize at the RWND upper bound, apart from the small time frame during slow start. This is because loss often occurs, resulting in the reduction of the congestion window. The fast recovery mechanism can be observed in the congestion window plot Figure 5.12. Other points of interest are the increase in both retransmissions and fast retransmissions, resulting in a higher loss as well as more ACKs received compared with the previous test. Additionally, the average time in the system W and the average number of packets in system L are also increased. This is due to more packets being lost, and consequently, the packets need to be retransmitted, causing a block in the receivers end, resulting in a higher average time and number of packets in the system. This is can be seen by looking at the Packets in system graph Figure 5.10 and the time in system graph Figure 5.11. The interarrival times shown in Figure 5.9 are as expected with less predictability caused by the increase in loss.

Variable	Result
λ	1.11 t
W	0.59 t
L	0.658 *Packets*
Packets Sent	2000 *Packets*
ACKs Received	1995 *Packets*
RT	133 *Packets*
FRT	55 *Packets*
Goodput	1.109 *Packets*/t
Loss rate	9.4%

Table 5.2: TCP Short Path High Loss Results

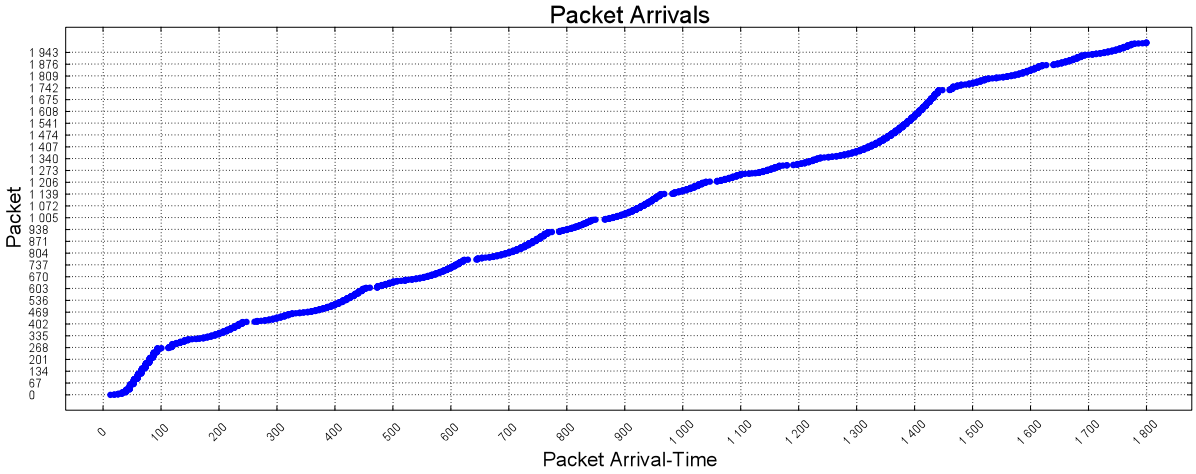


Figure 5.7: TCP Short Path High Loss Packet Arrivals

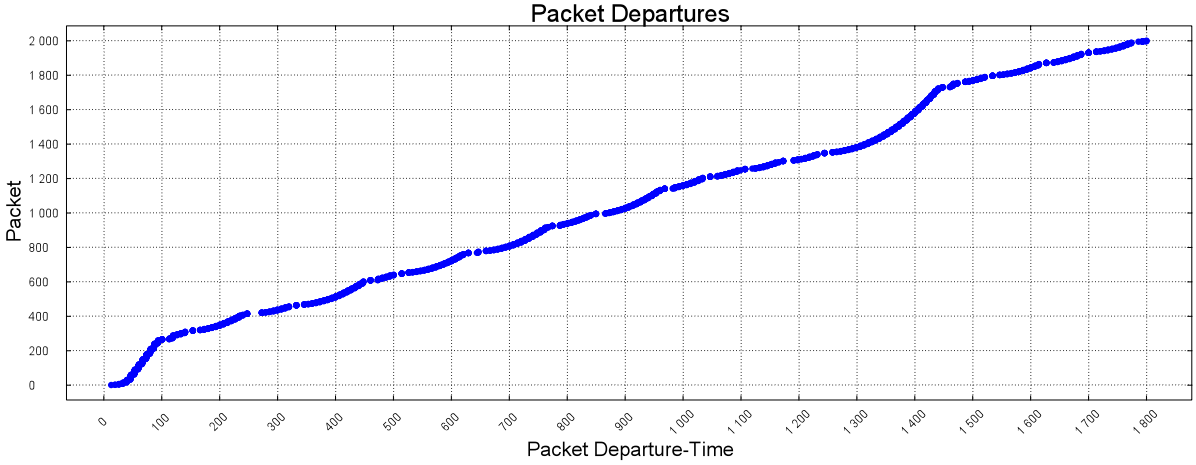


Figure 5.8: TCP Short Path High Loss Packet Departures

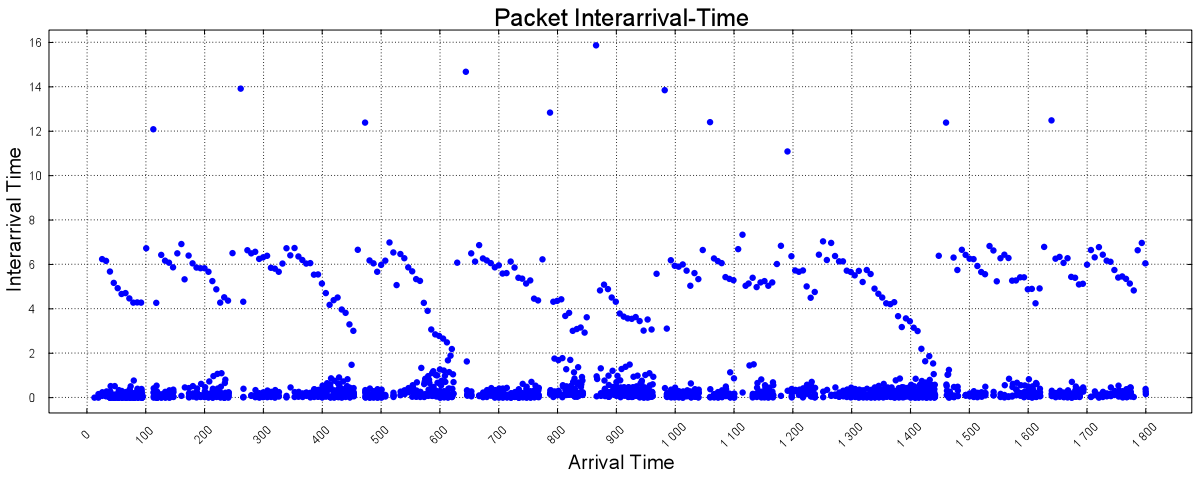


Figure 5.9: TCP Short Path High Loss Packet Interarrival Time

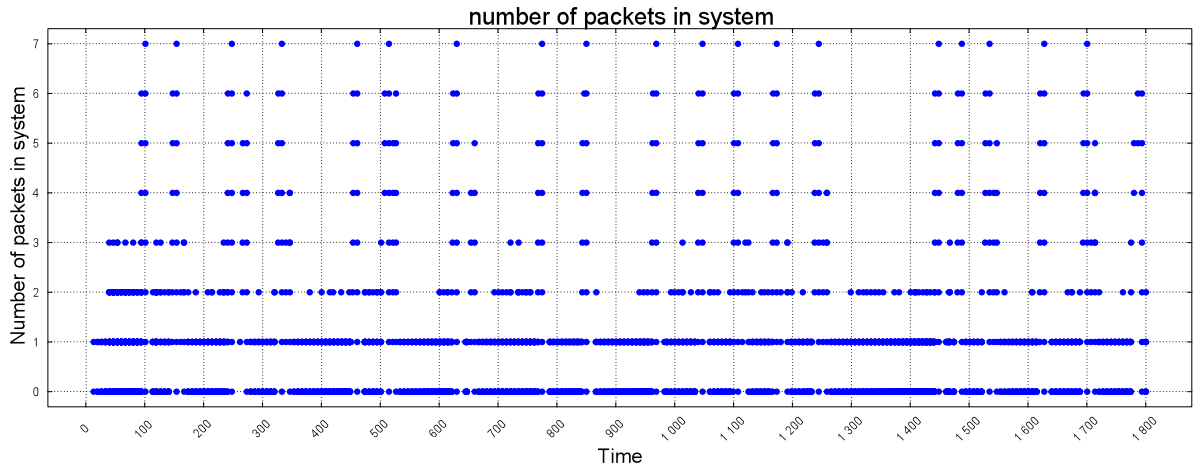


Figure 5.10: TCP Short Path High Loss Packets in System

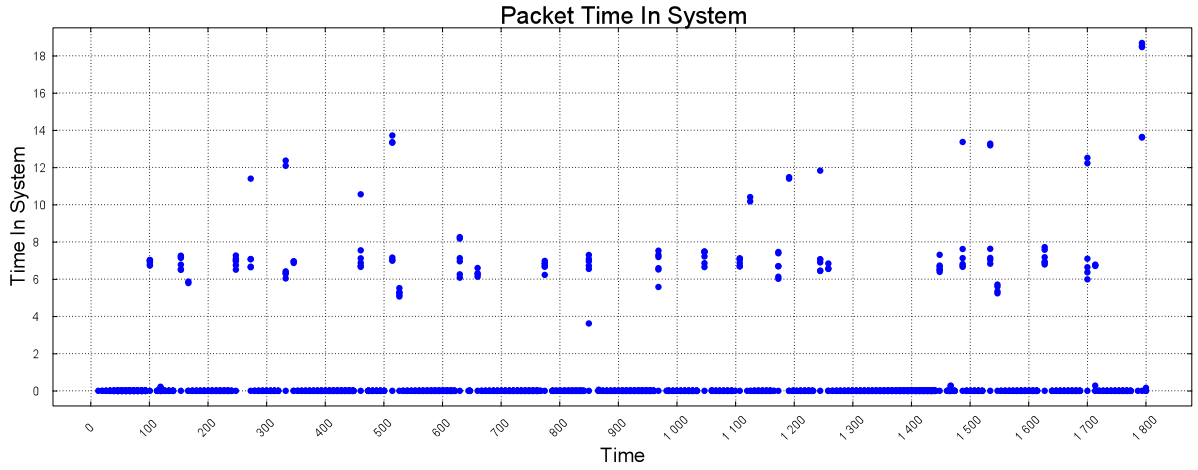


Figure 5.11: TCP Short Path High Loss Packet Time in System

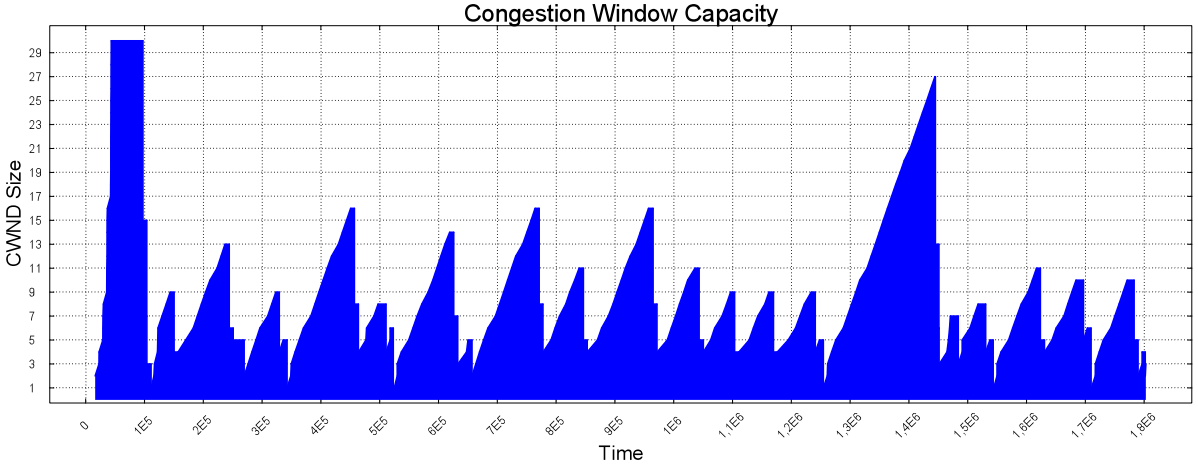


Figure 5.12: TCP Short Path High Loss Congestion Window

5.2.3 Long Path Low Loss

This test is using the long path with low loss section 5.1.1. Again looking at just the congestion window plot Figure 5.18 it is clear that there are four bursts of loss, where four have been serious enough to trigger RTO, while fast retransmission has solved one loss burst. Numerical results are presented in Table 5.3. One important point of interest is the dramatic decrease in goodput. This is caused by the longer path inherently reducing the number of packets being sent per time unit.

Variable	Result
λ	0.623 t
W	0.317 t
L	0.197 *Packets*
Packets Sent	2000 *Packets*
ACKs Received	1985 *Packets*
RT	67 *Packets*
FRT	24 *Packets*
Goodput	0.632 *Packets*/t
Loss rate	4.6%

Table 5.3: TCP Long Path Low Loss Results

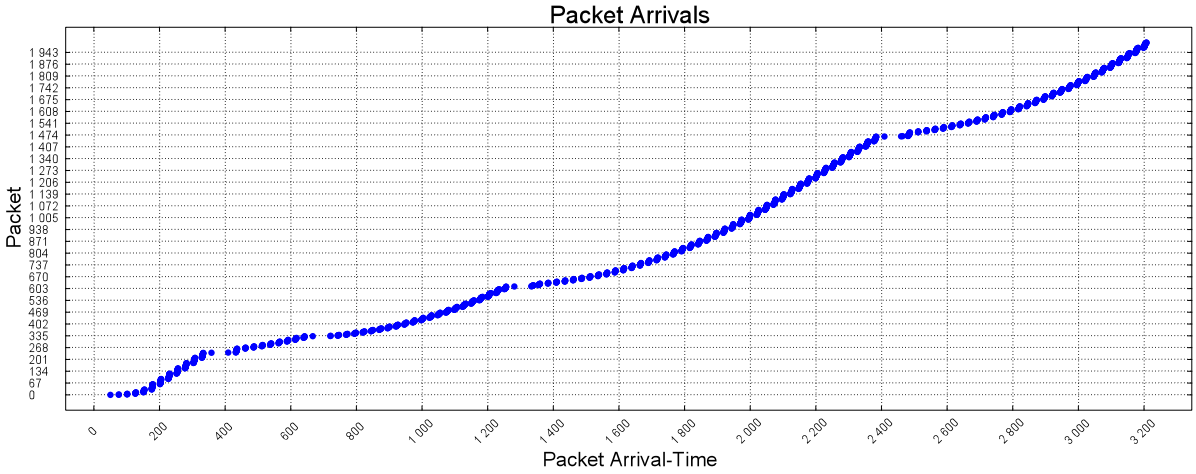


Figure 5.13: TCP Long Path Low Loss Packet Arrivals

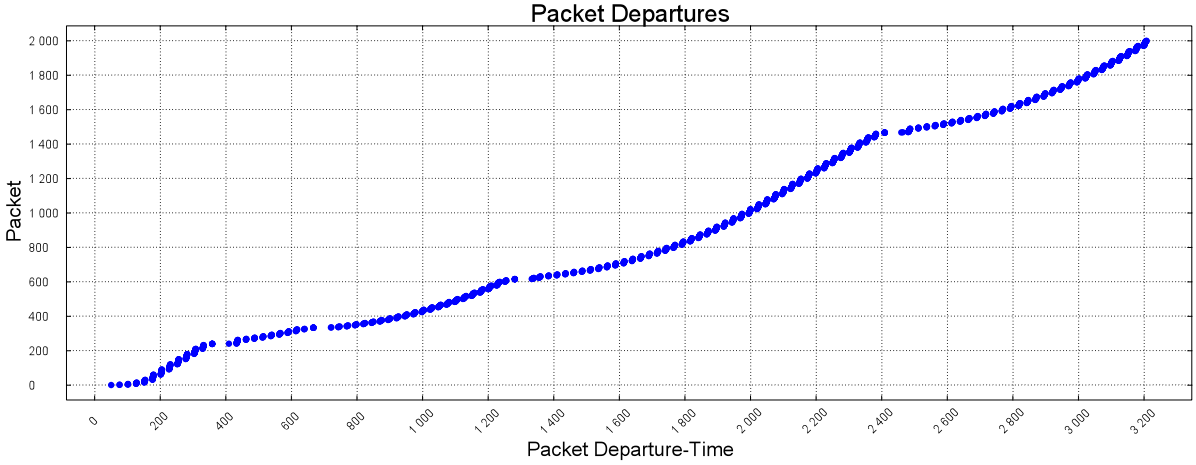


Figure 5.14: TCP Long Path Low Loss Packet Departures

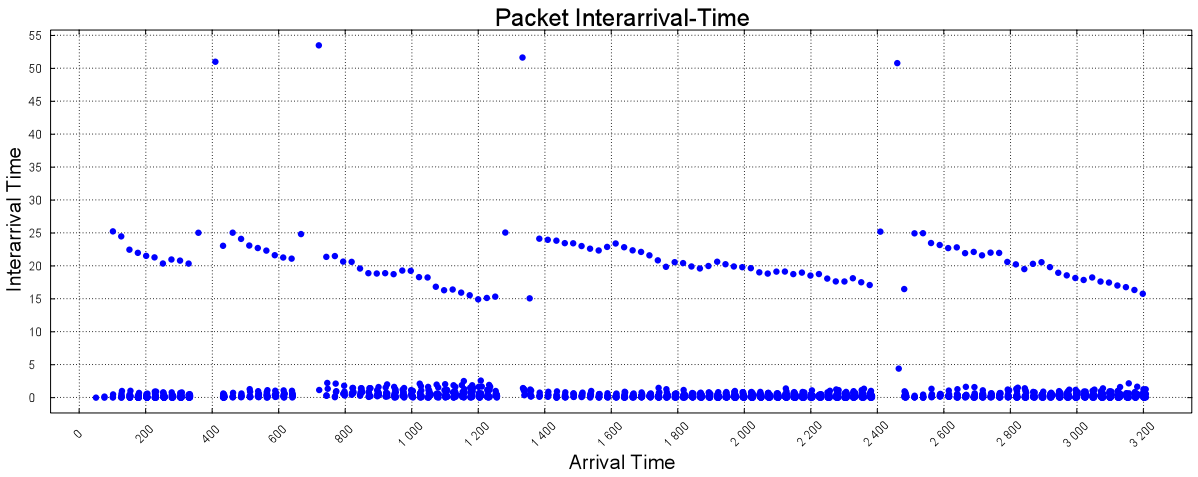


Figure 5.15: TCP Long Path Low Loss Packet Interarrival Time

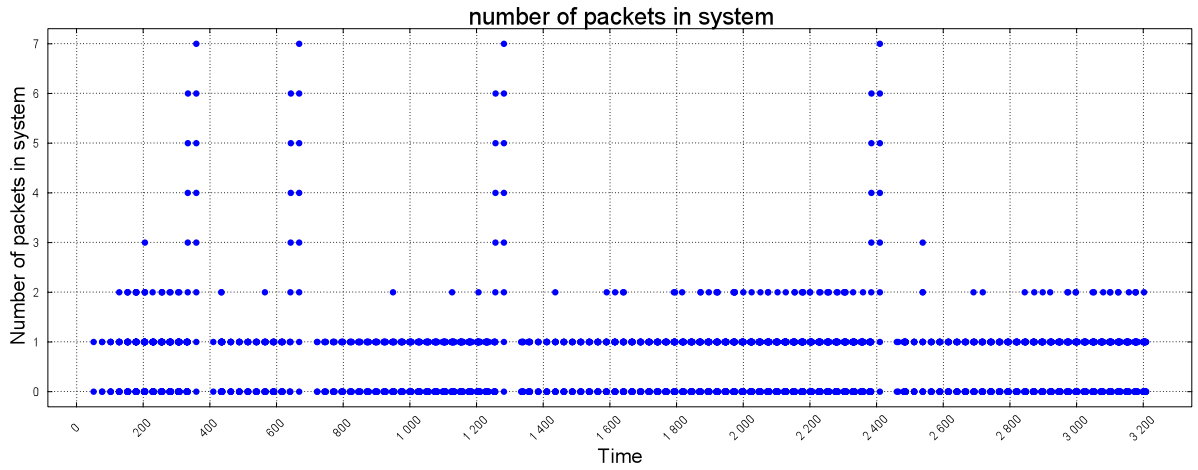


Figure 5.16: TCP Long Path Low Loss Packets in System

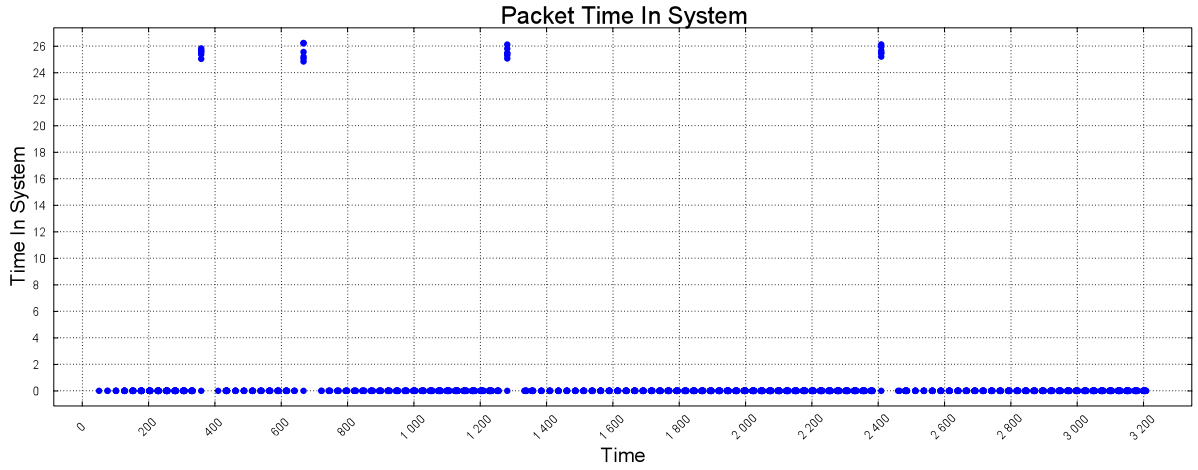


Figure 5.17: TCP Long Path Low Loss Packet Time in System

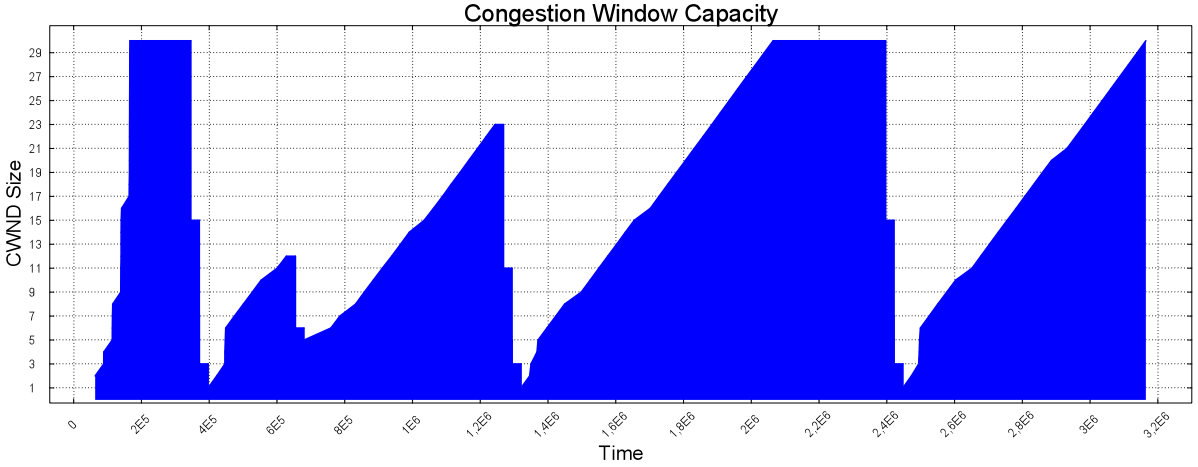


Figure 5.18: TCP Long Path Low Loss Congestion Window

5.2.4 Long Path High Loss

This test is using the long path with high loss section 5.1.1. Comparing this test with the results from the short path with high loss and the long path with low loss, the results are not that surprising. This is the most unforgiving path, with the highest loss and longest path leading to bad goodput. This is also the first test where the number of ACKs is higher than the number of packets sent.

Variable	Result
λ	0.223 t
W	3.523 t
L	0.787 *Packets*
Packets Sent	2000 *Packets*
ACKs Received	2074 *Packets*
RT	132 *Packets*
FRT	64 *Packets*
Goodput	0.224 *Packets*/t
Loss rate	9.8%

Table 5.4: TCP Long Path High Loss Results

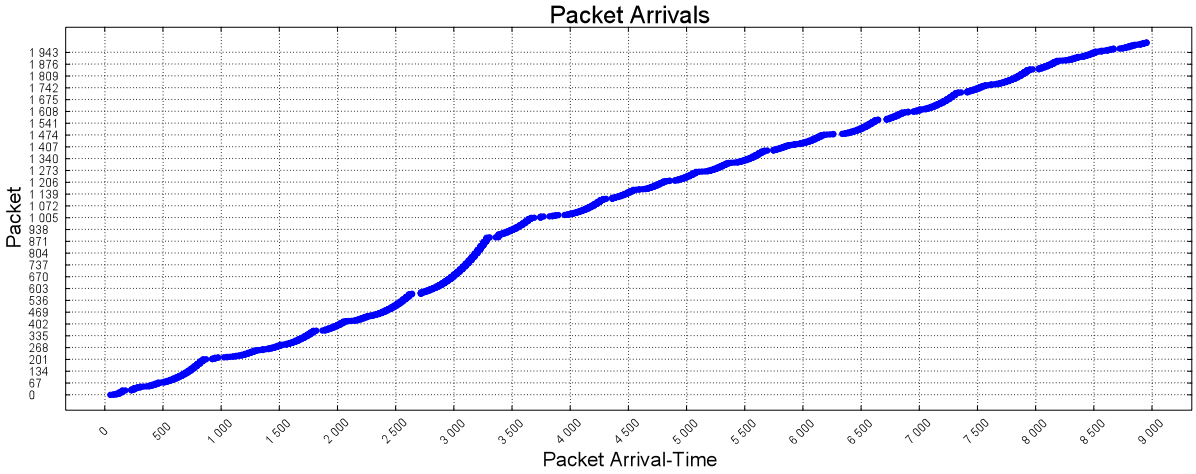


Figure 5.19: TCP Long Path High Loss Packet Arrivals

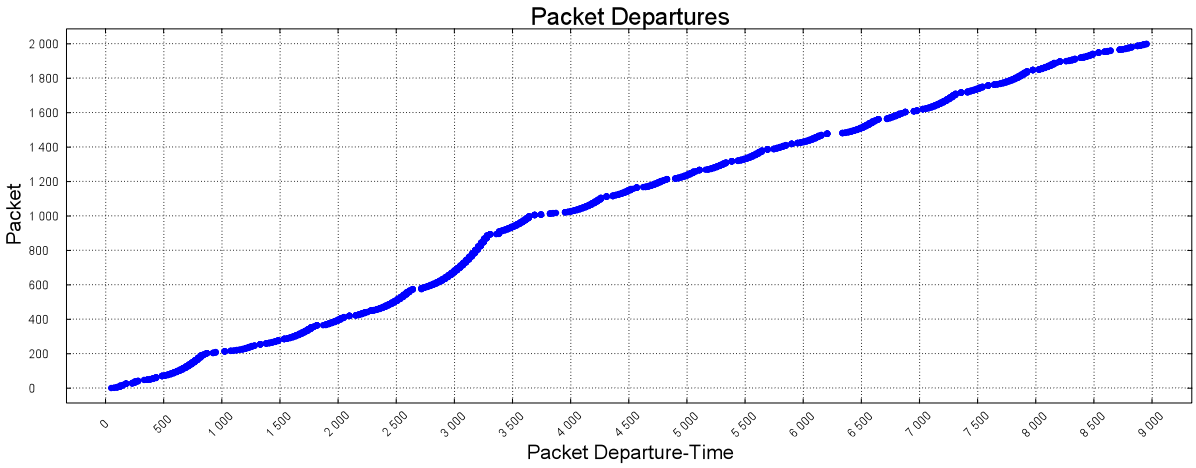


Figure 5.20: TCP Long Path High Loss Packet Departures

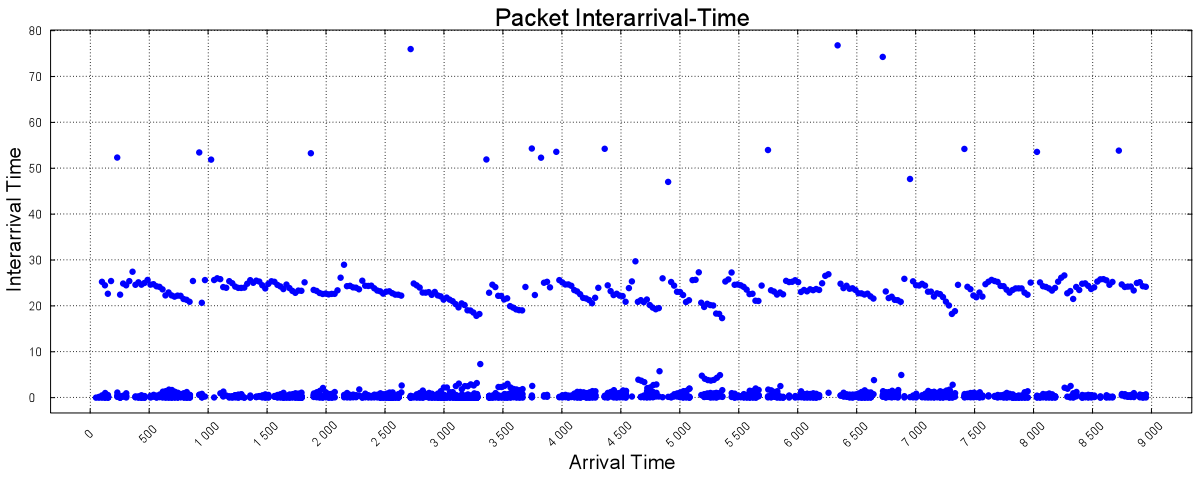


Figure 5.21: TCP Long Path High Loss Packet Interarrival Time

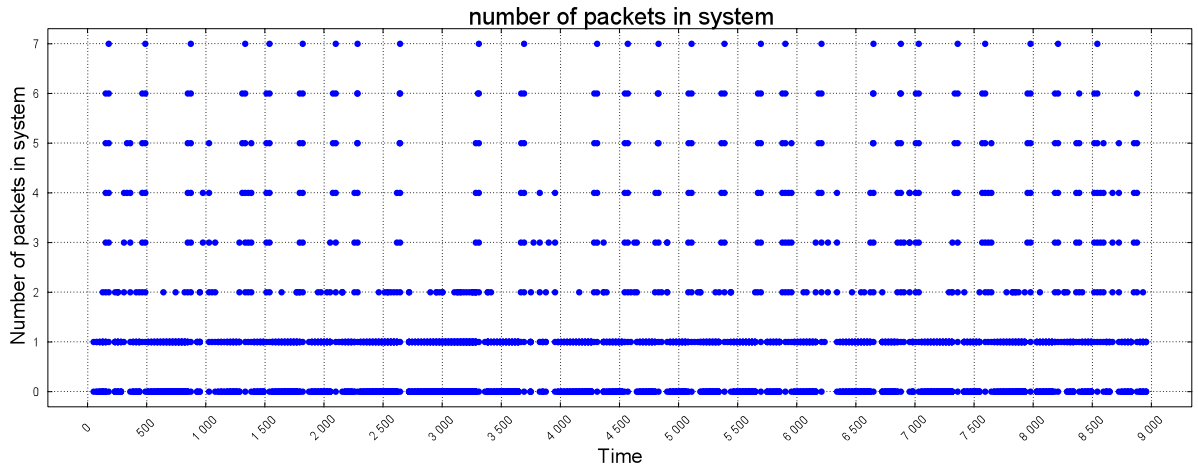


Figure 5.22: TCP Long Path High Loss Packets in System

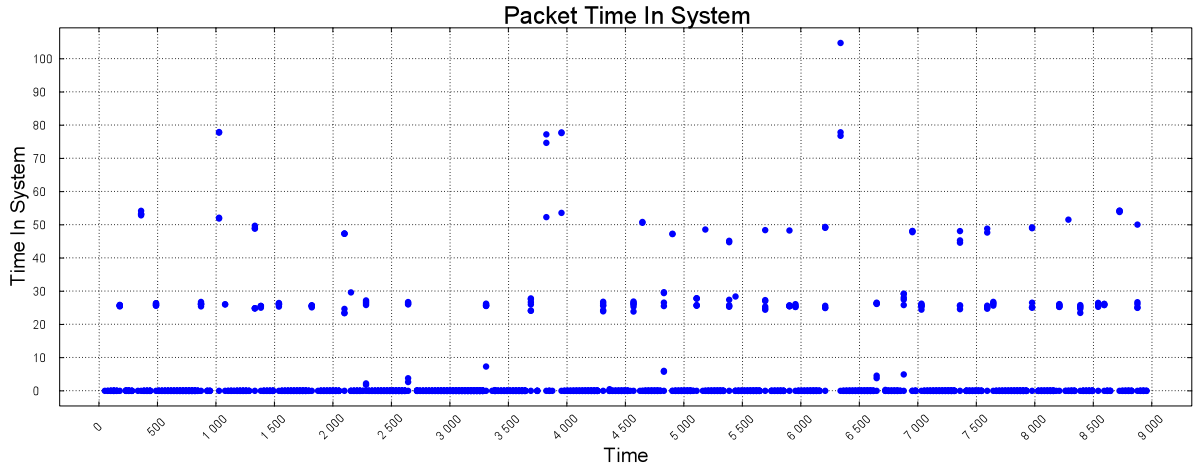


Figure 5.23: TCP Long Path High Loss Packet Time in System

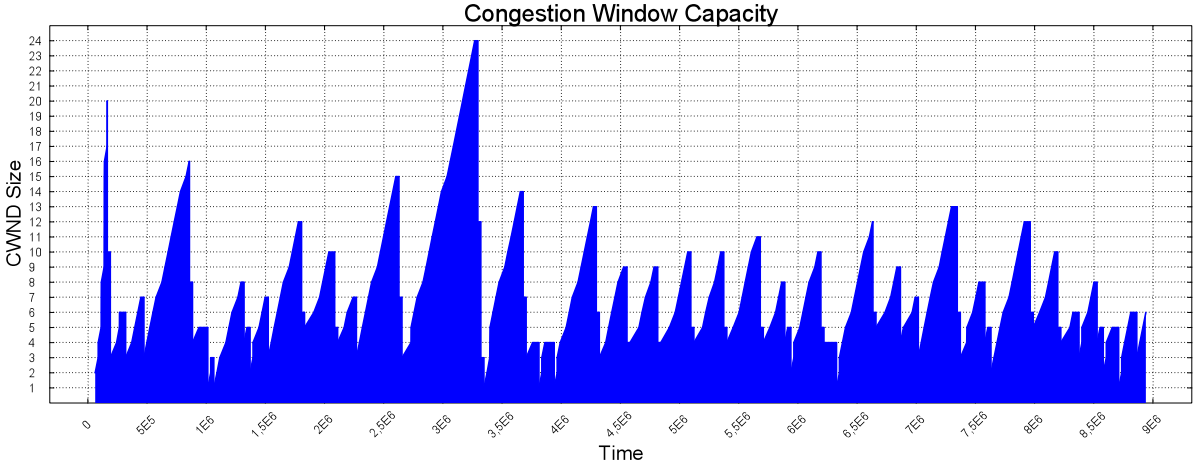


Figure 5.24: TCP Long Path High Loss Congestion Window

5.3 MPTCP

MPTCP is different from TCP in that it is using at least two paths to communicate. This raises issues concerning the behavior differences between homogeneous and heterogeneous paths. Using MPTCP over homogeneous paths, or paths with equal properties, will, in theory, distribute the traffic equally, effectively doubling the throughput compared to using just one of the paths, as is the case for TCP. However, real-life MPTCP situations will more often than not use heterogeneous paths, paths with different properties. All the presented tests have been completed using two subflows as this is the most likely use case in practice. However, the implementation has been tested using several subflows, but this did not dramatically change the overall results. The following sections will test the

implemented MPTCP simulator on multiple environments with high and low loss, long and short path, and homogeneous and heterogeneous paths.

5.3.1 Homogeneous Paths Low Loss

The results achieved for homogeneous paths in the long and short path environments introduced in section 5.2 are essentially the same when adjusting for the added delay that comes from the longer path. For this reason, only the results from the short path are shown. Firstly the two subflows are not distributing the load equally, with subflow 0 sending 1217 out of 2000 packets while subflow 1 is sending 783 packets out of 2000 seen in Table 5.5. This is a problem since the two subflows have equal prerequisites. However, testing with 20000 yields roughly the same packet offset. In addition, the combined goodput of the two subflows is, in fact, lower than the TCP baseline.

The loss of the two subflows is at 2.8% and 4.9%, respectfully. This is not as expected since the standard TCP implementation experienced a 2% loss in the same environment. Further, there are no retransmissions caused by the RTO, but rather an excessive amount of fast retransmissions. This raises an issue that will become clearer as more loss and heterogeneous paths are introduced. When looking at the congestion windows Figure 5.30 it is clear that the subflows are getting unnecessary response and are therefore reducing the congestion window. This response comes from packets received from the other subflow, effectively triggering DupACKs to be sent and consequently interfering with the other subflow. The slow start mechanism at the start of subflow 1's congestion window graph stops much earlier than subflow 0, giving more evidence of interference in the congestion control mechanisms.

The subflows of MPTCP must in some way interfere with each other since packets must arrive in order. Looking at the arrival graph Figure 5.25, departure graph Figure 5.26, interarrival graph Figure 5.27, and especially the number of packets in system graph Figure 5.28 and the packet time in the system graph Figure 5.29 it is clear that the subflows are not operating independently of each other.

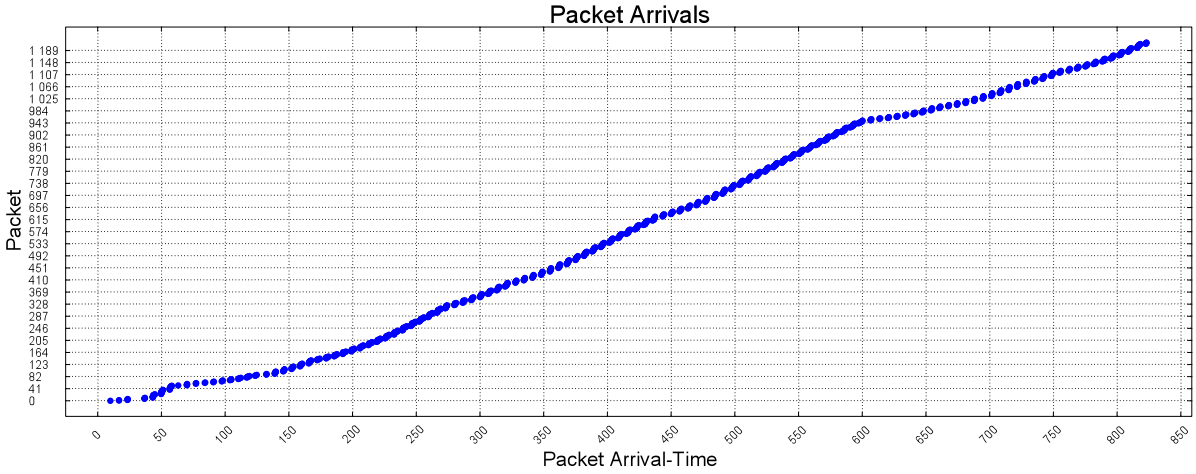
(a) Subflow 0

Variable	Result
λ	1.478 t
W	0.193 t
L	0.285 $Packets$
$Packets\ Sent$	1217 $Packets$
$ACKs\ Received$	1287 $Packets$
RT	0 $Packets$
FRT	34 $Packets$
$Goodput$	1.473 $Packets/t$
$Loss\ rate$	2.8%

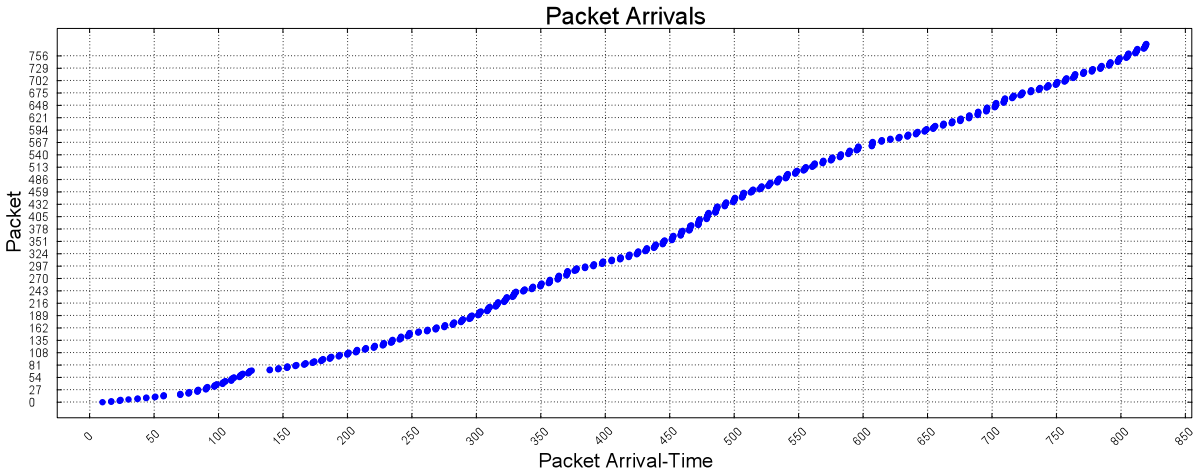
(b) Subflow 1

Variable	Result
λ	0.955 t
W	0.180 t
L	0.172 $Packets$
$Packets\ Sent$	783 $Packets$
$ACKs\ Received$	958 $Packets$
RT	0 $Packets$
FRT	38 $Packets$
$Goodput$	0.953 $Packets/t$
$Loss\ rate$	4.9%

Table 5.5: MPTCP Homogeneous Paths Low Loss Results

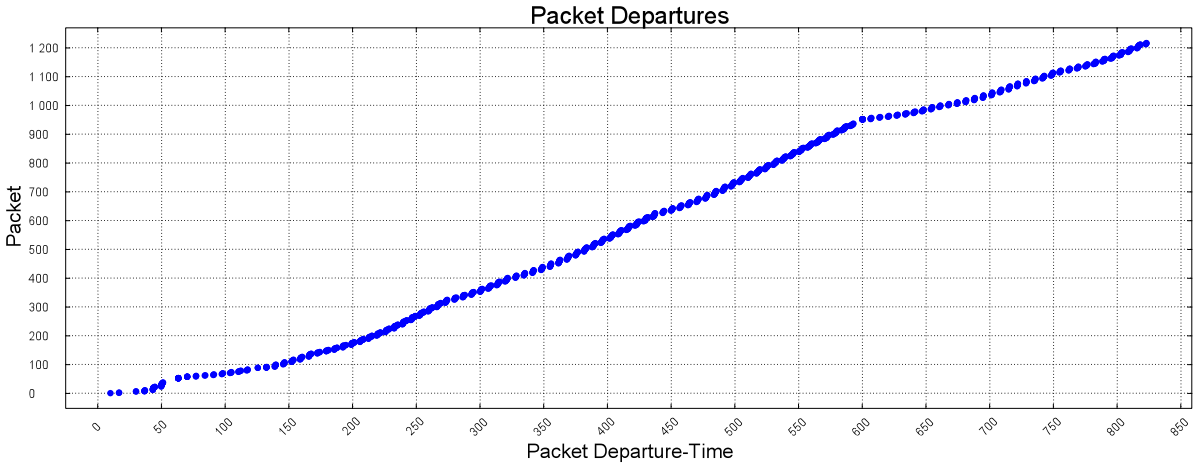


(a) Subflow 0

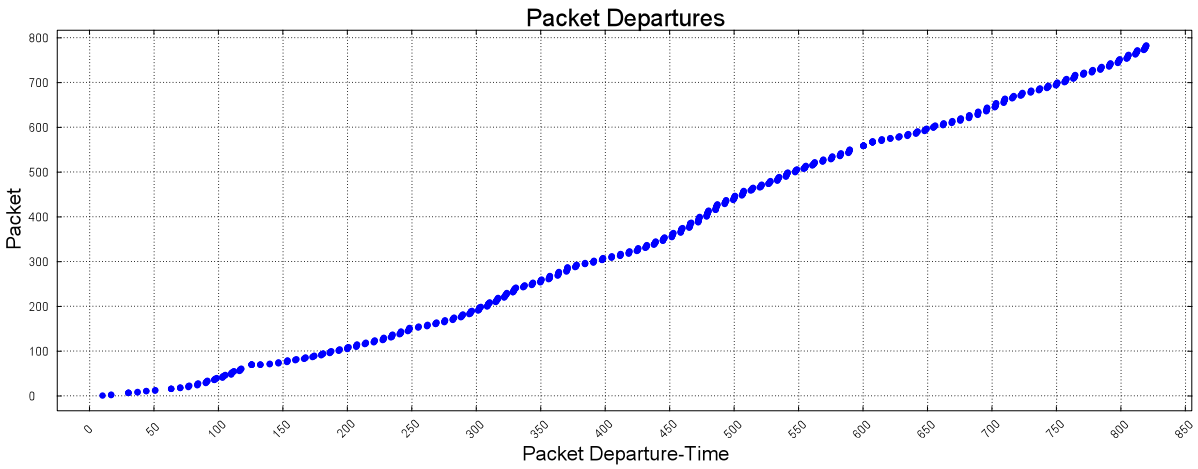


(b) Subflow 1

Figure 5.25: MPTCP Homogeneous Paths Low Loss Packet Arrivals

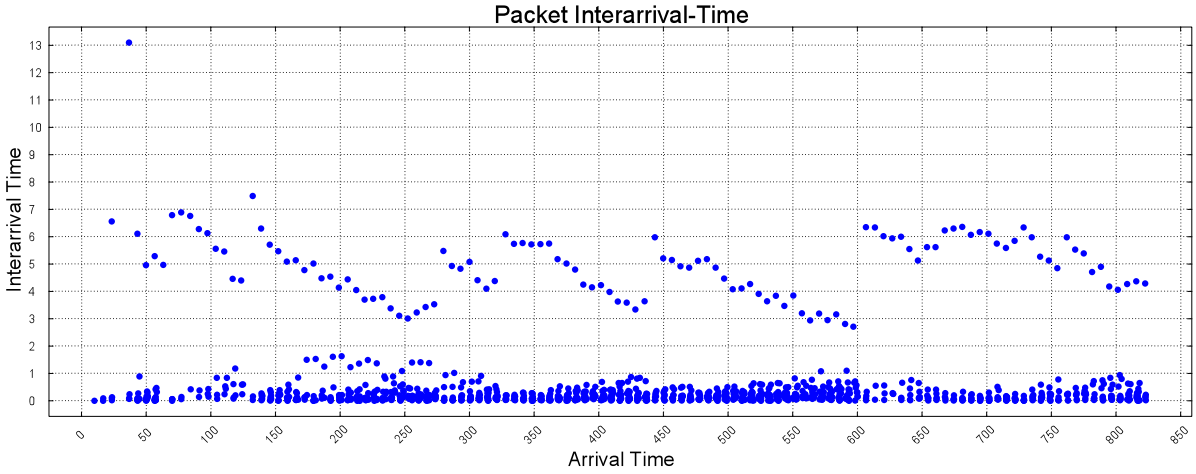


(a) Subflow 0

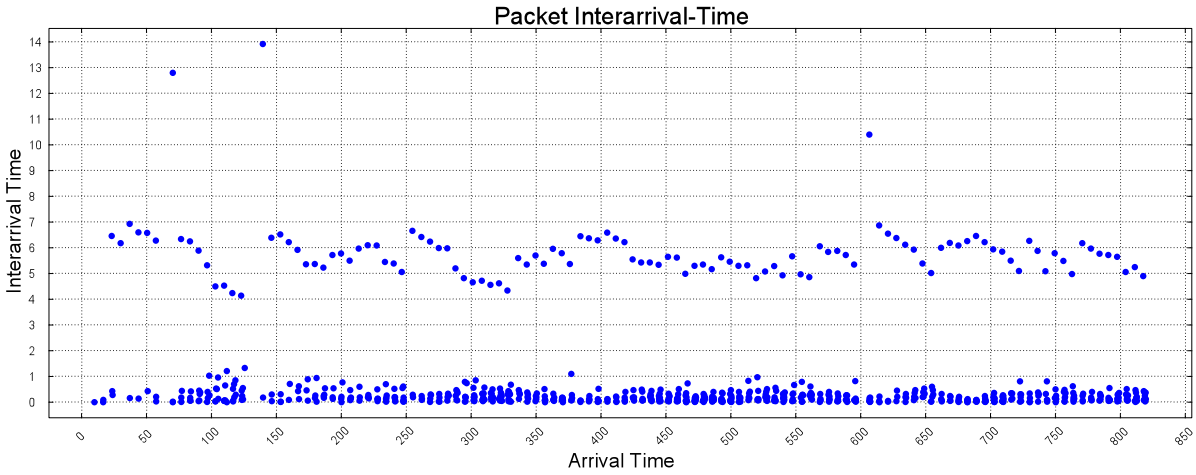


(b) Subflow 1

Figure 5.26: MPTCP Homogeneous Paths Low Loss Packet Departures

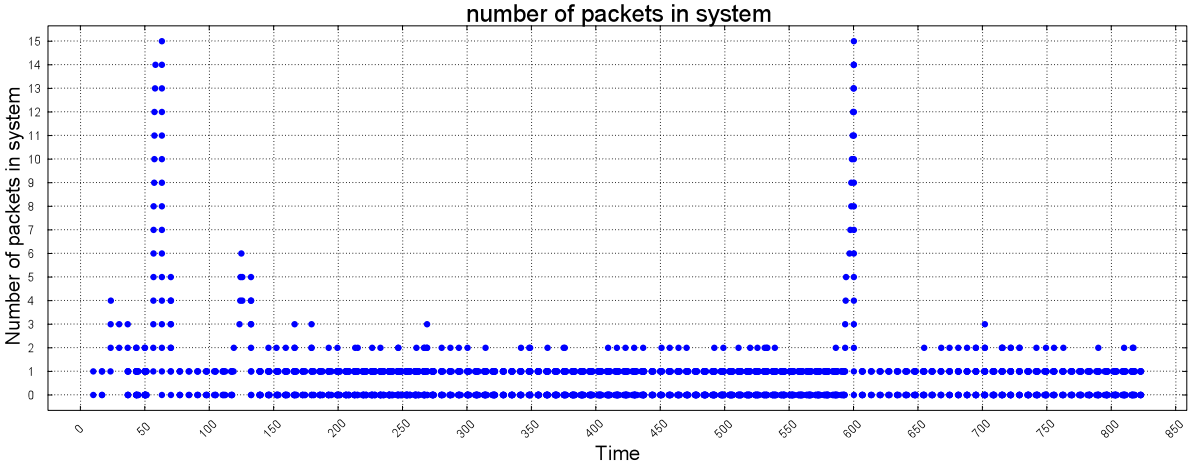


(a) Subflow 0

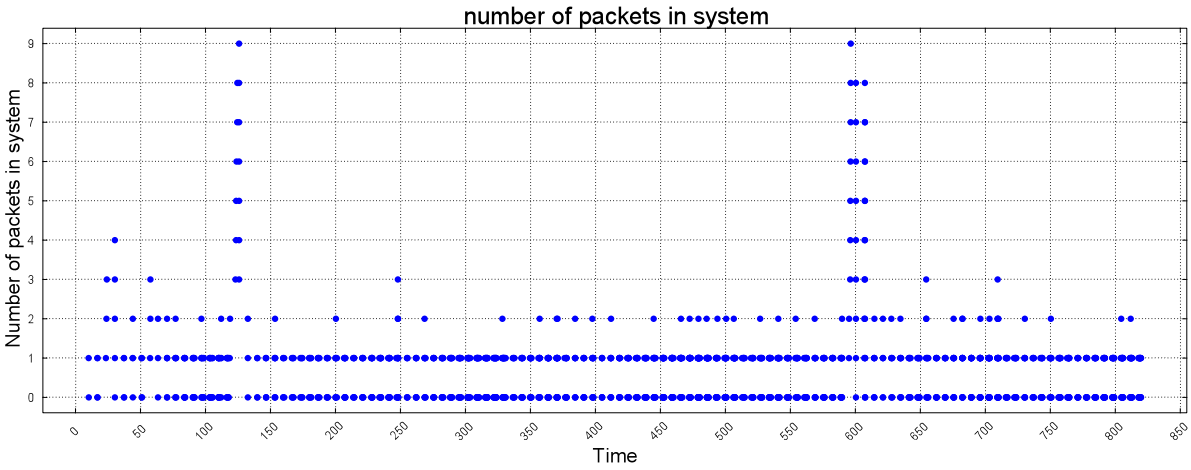


(b) Subflow 1

Figure 5.27: MPTCP Homogeneous Paths Low Loss Packet Interarrival Time

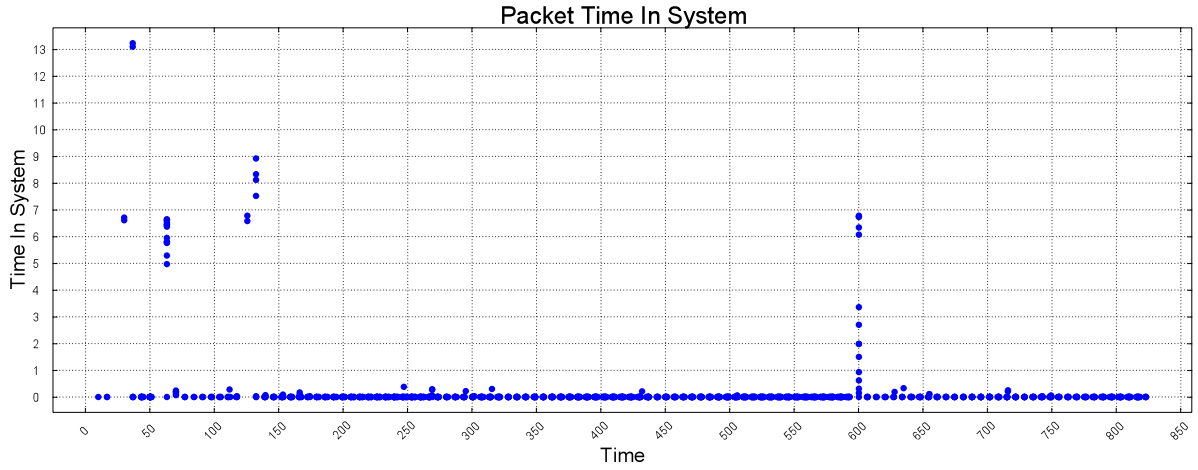


(a) Subflow 0

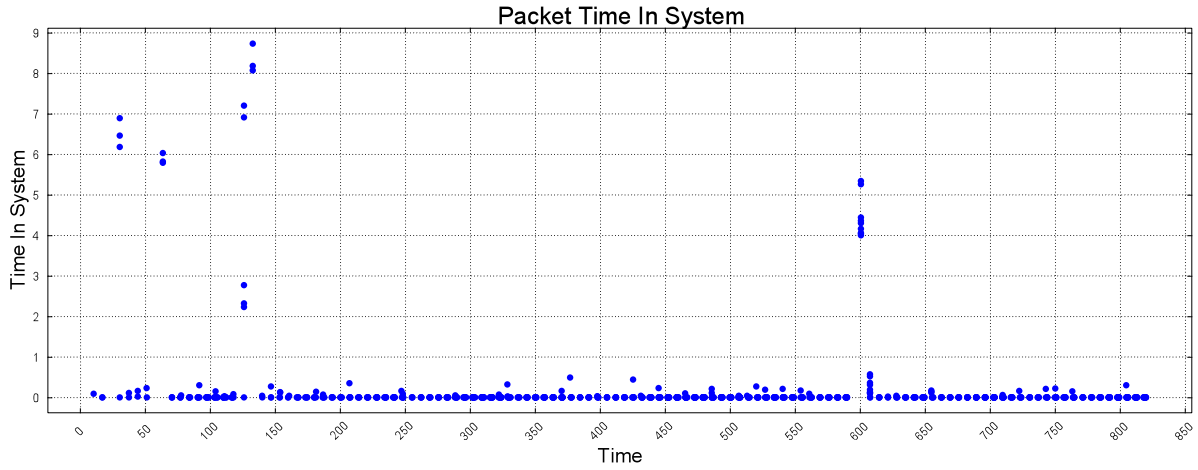


(b) Subflow 1

Figure 5.28: MPTCP Homogeneous Paths Low Loss Packets in System

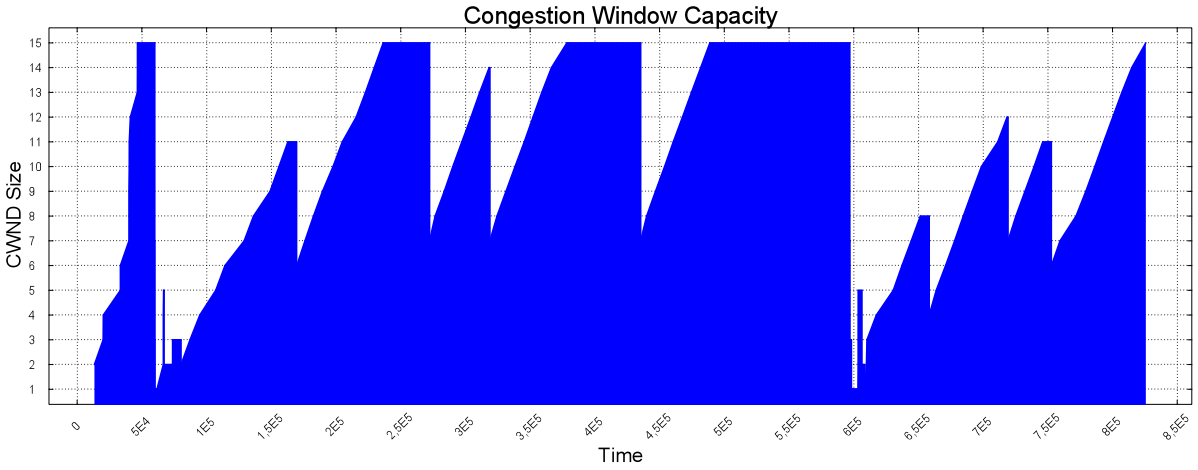


(a) Subflow 0

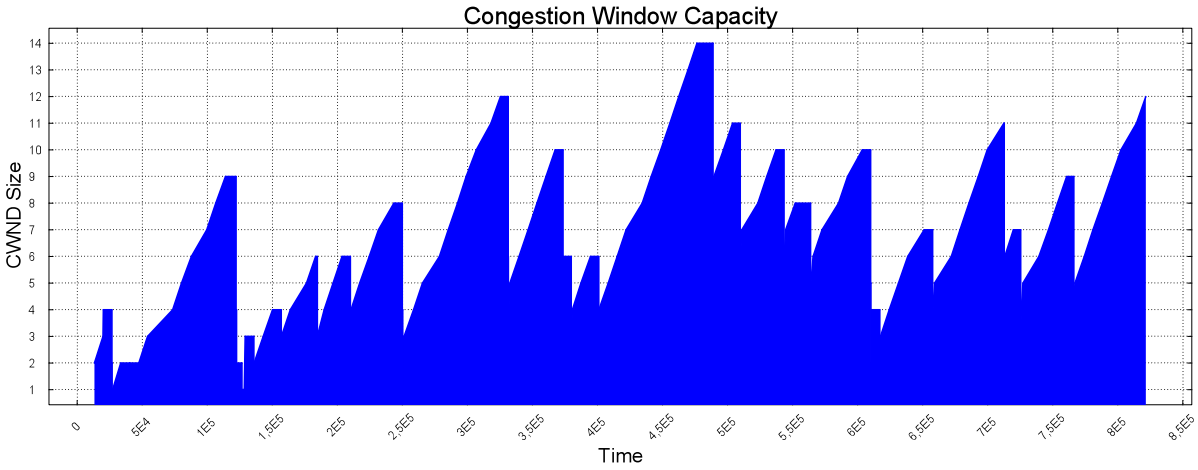


(b) Subflow 1

Figure 5.29: MPTCP Homogeneous Paths Low Loss Packet Time in System



(a) Subflow 0



(b) Subflow 1

Figure 5.30: MPTCP Homogeneous Paths Low Loss Congestion Window

5.3.2 Homogeneous Paths High Loss

For the same reason as discussed in subsection 5.3.1 the results from the short path will be presented rather than give the results for both the long and short path. This test is using the short path high loss path section 5.1.1 for both subflows. The results seen in Table 5.6 are repeating the key take away from the previous test (subsection 5.3.1). Subflow 1 has over double the amount of perceived loss as subflow 0, which increases the difference compared to the previous test with low loss. This can again be connected to the fact that excessive DupACKs are sent, which is evident in the congestion graph and especially during slow start. This tells that increased loss will contribute to more redundant retransmissions.

However, looking at the number of packets sent in Table 5.6, higher loss contributes to

a better load balancing between the two subflows. Perhaps related is the fact that the combined goodput is substantially better compared with the results from normal TCP on the same path (subsection 5.2.2). Individually though, normal TCP outperforms the MPTCP subflows. Nevertheless, this test is promising compared to the test using homogeneous paths with low loss (subsection 5.3.1).

One possible reason that this test has better goodput and balancing compared with the previous test might be that the increased loss limits the dominant subflow and consequently allows the other subflow to send uninterrupted for a while. Another possible explanation might be the effect the increased loss has on the random generation. The additional loss will contribute to even more calls to the random generator, making it unclear if the results are due to lucky randomness or implementation related.

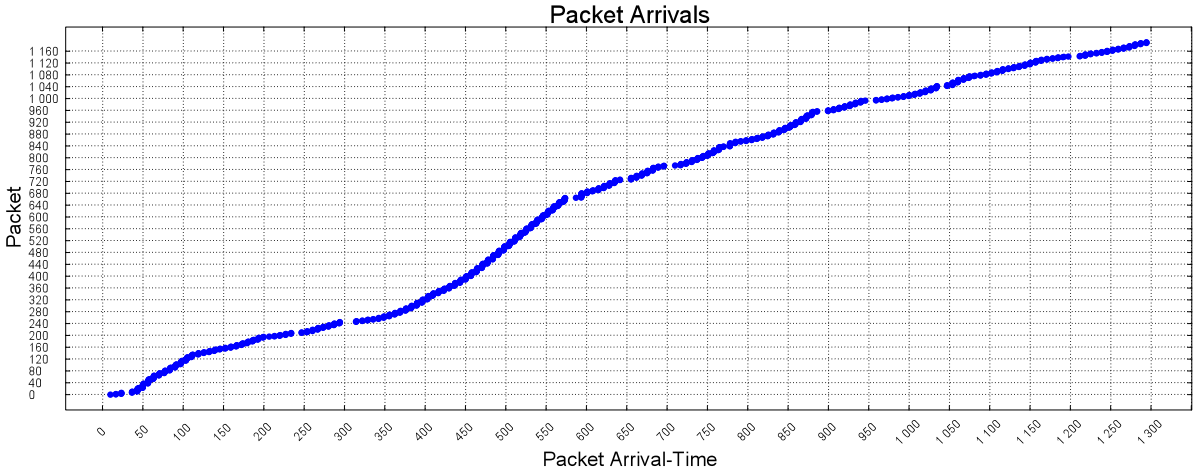
(a) Subflow 0

Variable	Result
λ	0.92 t
W	0.84 t
L	0.773 $Packets$
$Packets\ Sent$	1191 $Packets$
$ACKs\ Received$	1418 $Packets$
RT	25 $Packets$
FRT	73 $Packets$
$Goodput$	0.918 $Packets/t$
$Loss\ rate$	8.2%

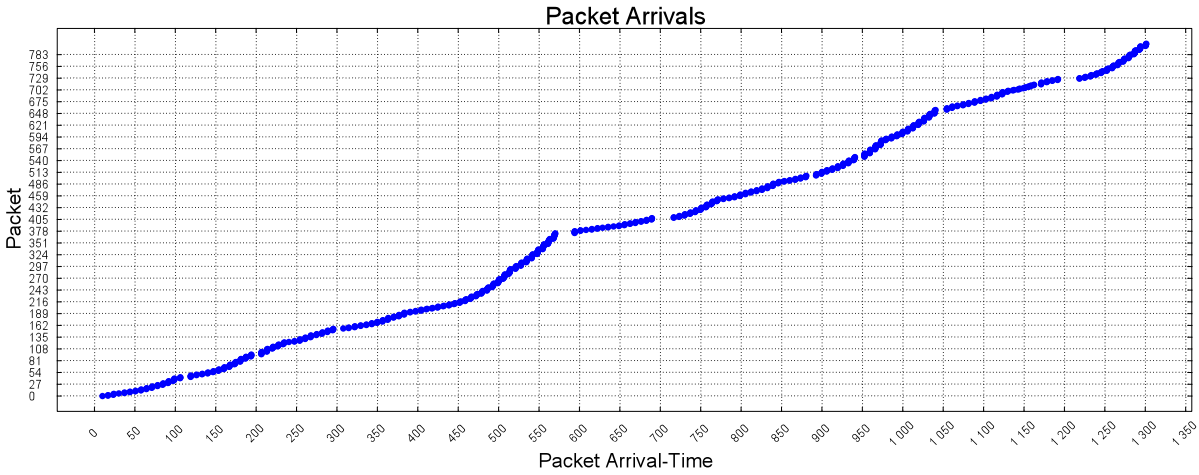
(b) Subflow 1

Variable	Result
λ	0.622 t
W	0.990 t
L	0.615 $Packets$
$Packets\ Sent$	809 $Packets$
$ACKs\ Received$	1391 $Packets$
RT	26 $Packets$
FRT	135 $Packets$
$Goodput$	0.623 $Packets/t$
$Loss\ rate$	19.9%

Table 5.6: MPTCP Homogeneous Paths High Loss Results

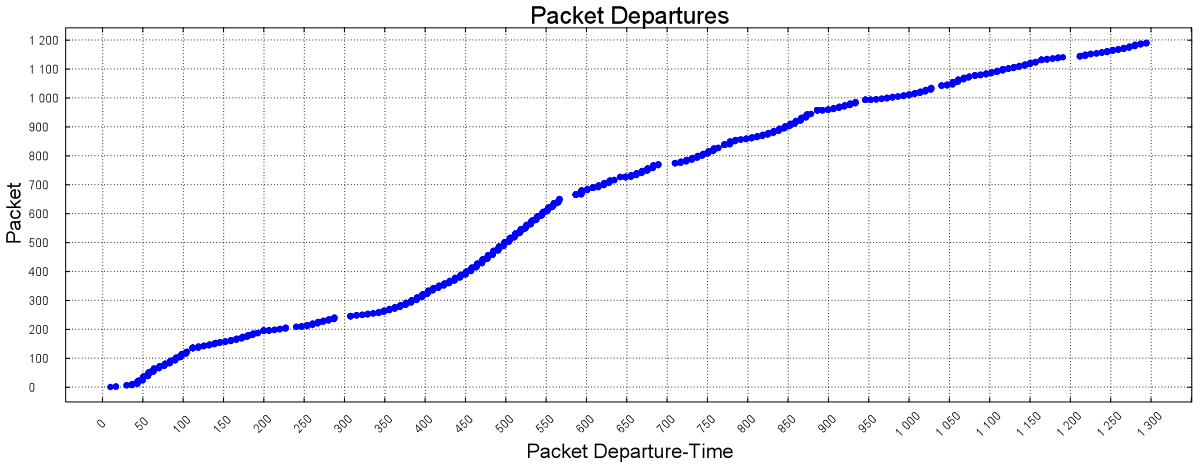


(a) Subflow 0

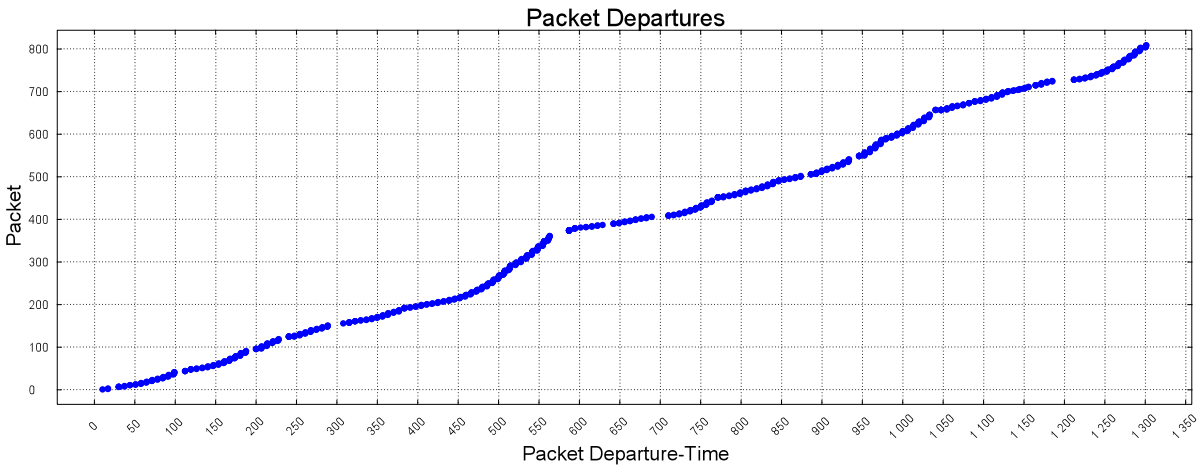


(b) Subflow 1

Figure 5.31: MPTCP Homogeneous Paths High Loss Packet Arrivals

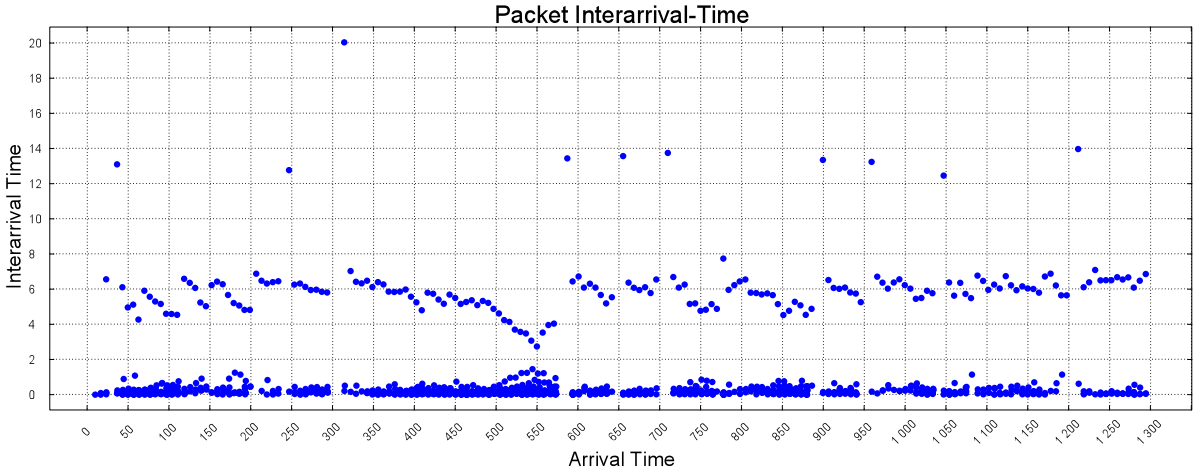


(a) Subflow 0

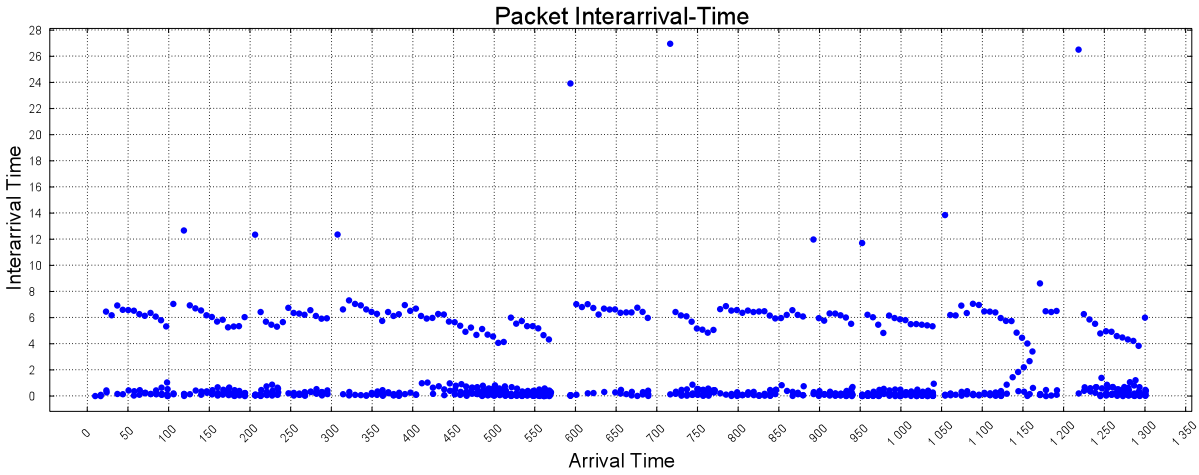


(b) Subflow 1

Figure 5.32: MPTCP Homogeneous Paths High Loss Packet Departures

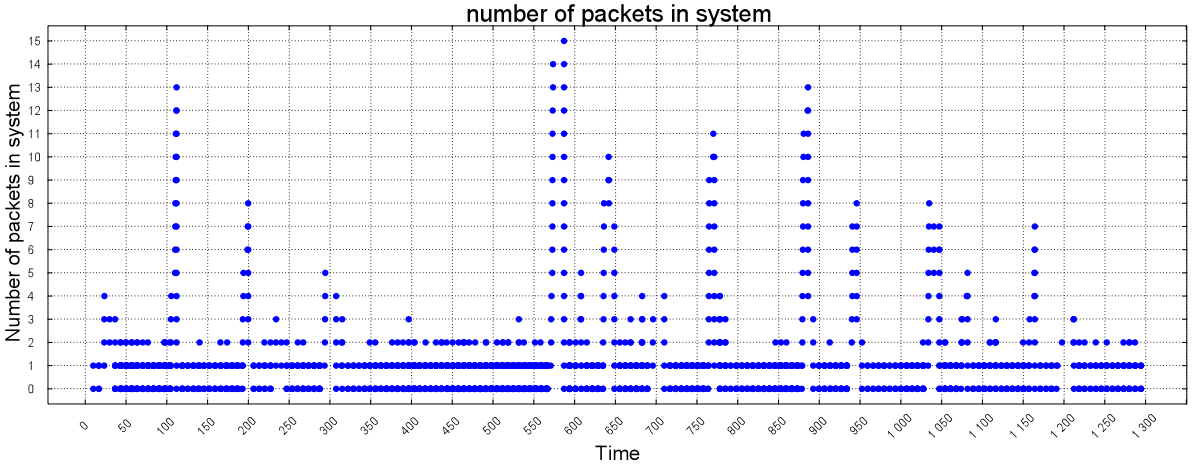


(a) Subflow 0

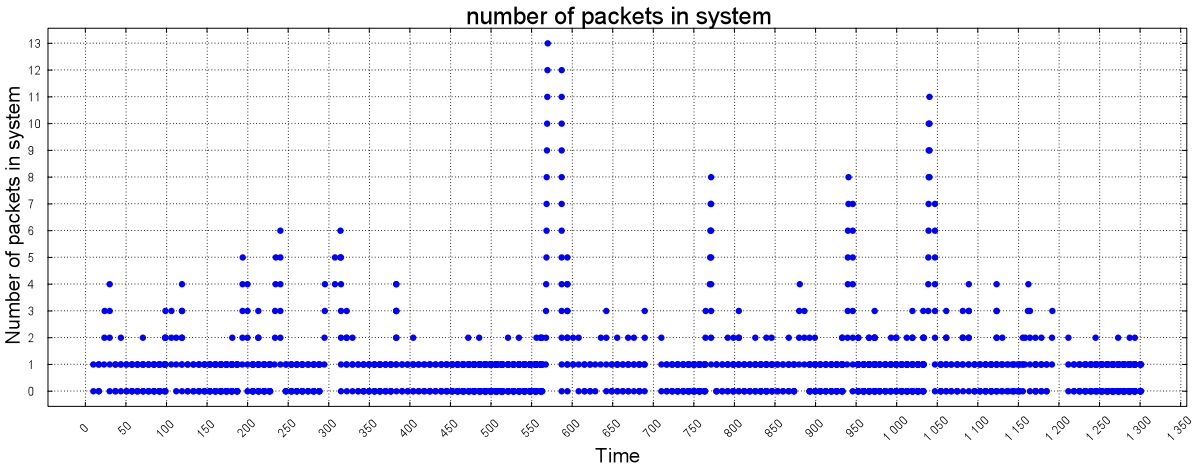


(b) Subflow 1

Figure 5.33: MPTCP Homogeneous Paths High Loss Packet Interarrival Time

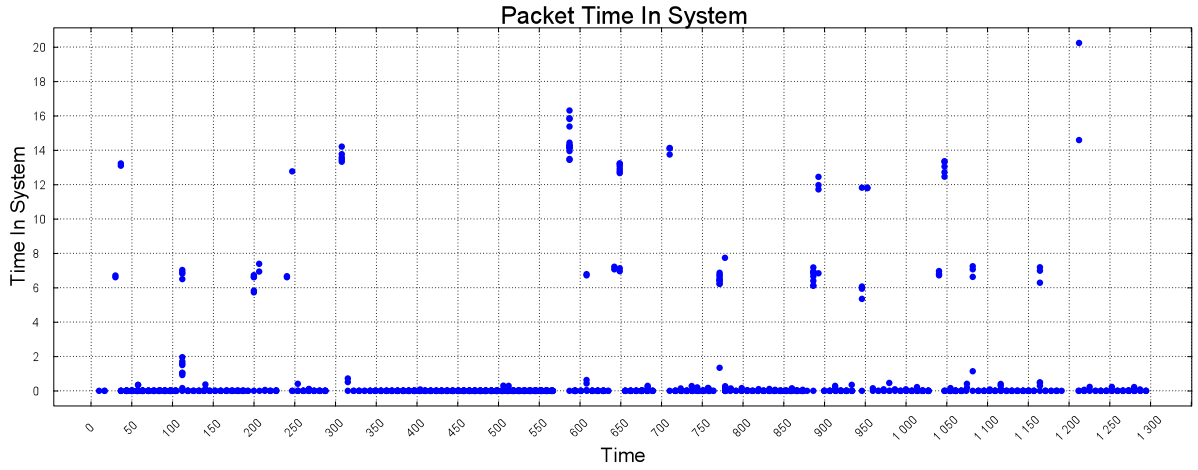


(a) Subflow 0

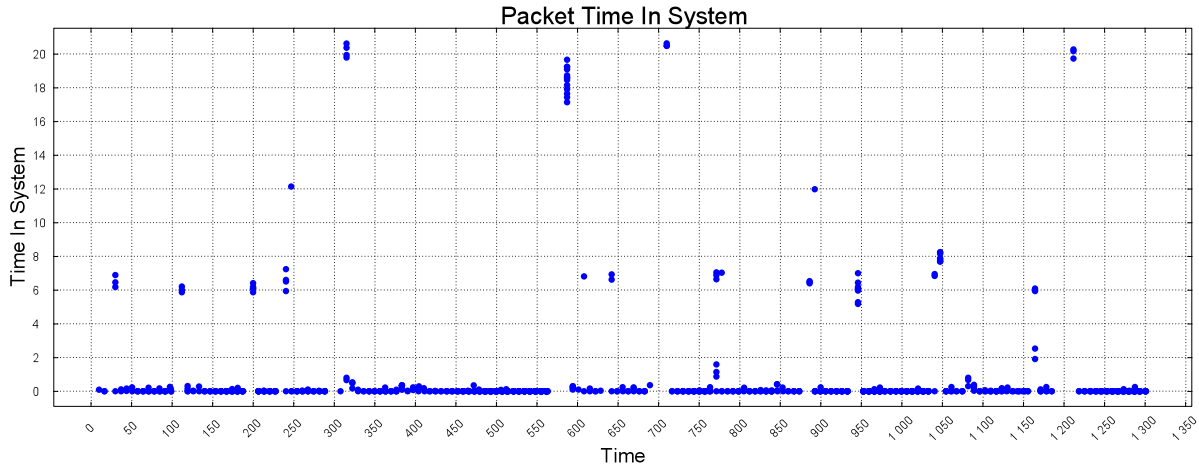


(b) Subflow 1

Figure 5.34: MPTCP Homogeneous Paths High Loss Packets in System

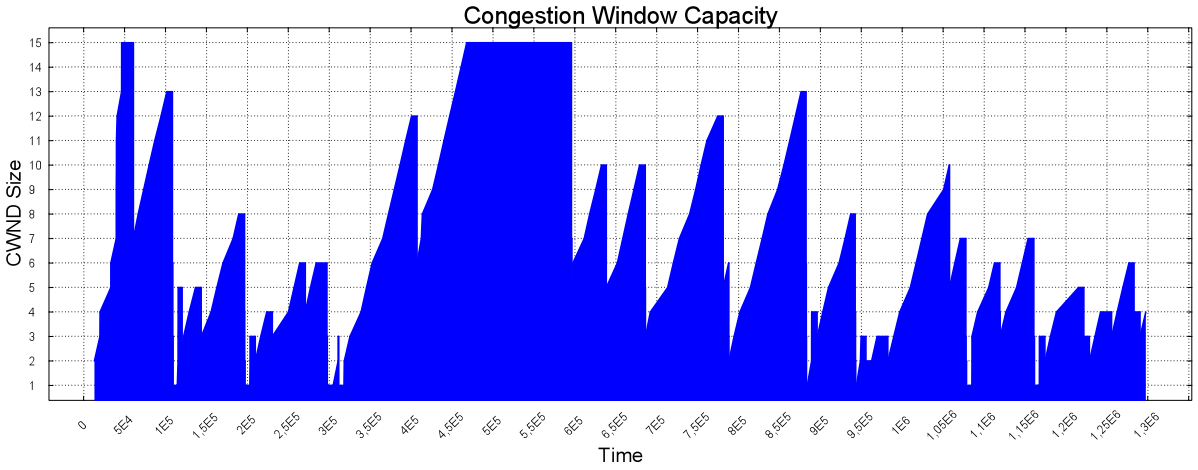


(a) Subflow 0

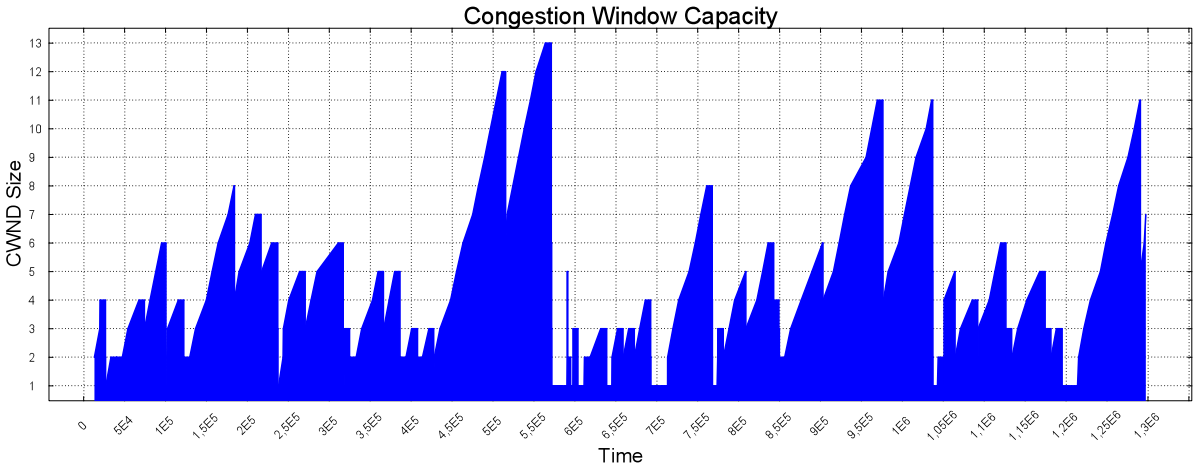


(b) Subflow 1

Figure 5.35: MPTCP Homogeneous Paths High Loss Packet Time in System



(a) Subflow 0



(b) Subflow 1

Figure 5.36: MPTCP Homogeneous Paths High Loss Congestion Window

5.3.3 Heterogeneous Paths Low Loss

This test is using both the short path with low loss (section 5.1.1) and the long path with low loss (section 5.1.1), making it a test of MPTCP on heterogeneous paths with low loss. The general results presented in Table 5.7 show that this simulated implementation of MPTCP performs poorly when the paths are of unequal length. While load balancing between the two paths is somewhat equal with subflow 0 at 1198 packets sent and subflow 1 at 802 packets sent, subflow 1 should have sent most of the packet as this flow sends over the short path. With this in mind, it is clear that subflow 0 is the dominant subflow that severely damages the potential of subflow 1.

The results generated from subflow 0 are somewhat equal to the standard TCP results. Almost all metrics are worse but not unexpected or abnormal, except for no RTO retrans-

missions and a congestion window that seems to stutter. Subflow 1, on the other hand, is producing completely unexpected results. None of the graphs resemble the standard TCP tests, and the performance metrics are flawed. 595 fast retransmissions, a 76.7% loss rate, and 3348 AKCs received (Table 5.7) are results that does not make sense

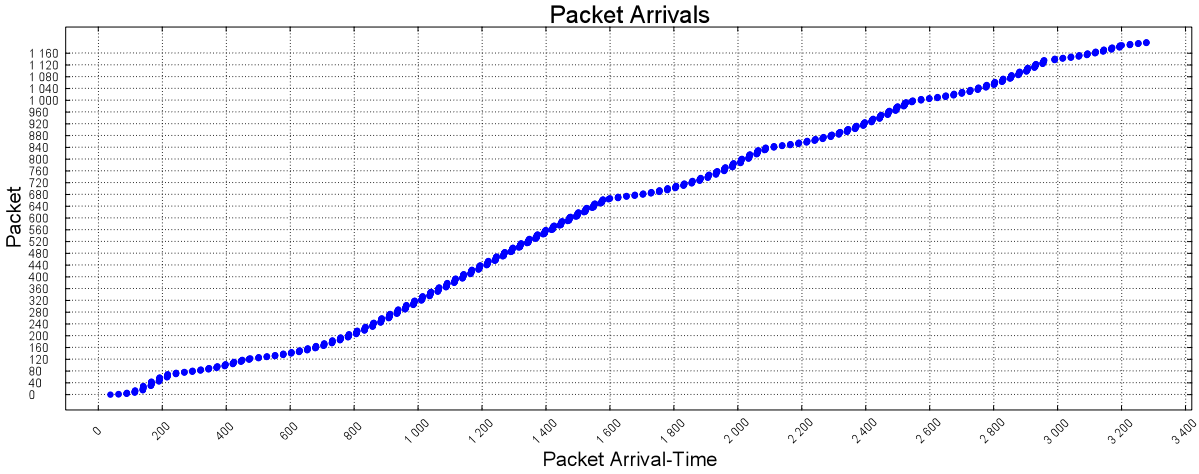
(a) Subflow 0

Variable	Result
λ	0.365 t
W	0.957 t
L	0.35 $Packets$
$Packets\ Sent$	1198 $Packets$
$ACKs\ Received$	1386 $Packets$
RT	0 $Packets$
FRT	68 $Packets$
$Goodput$	0.367 $Packets/t$
$Loss\ rate$	5.7%

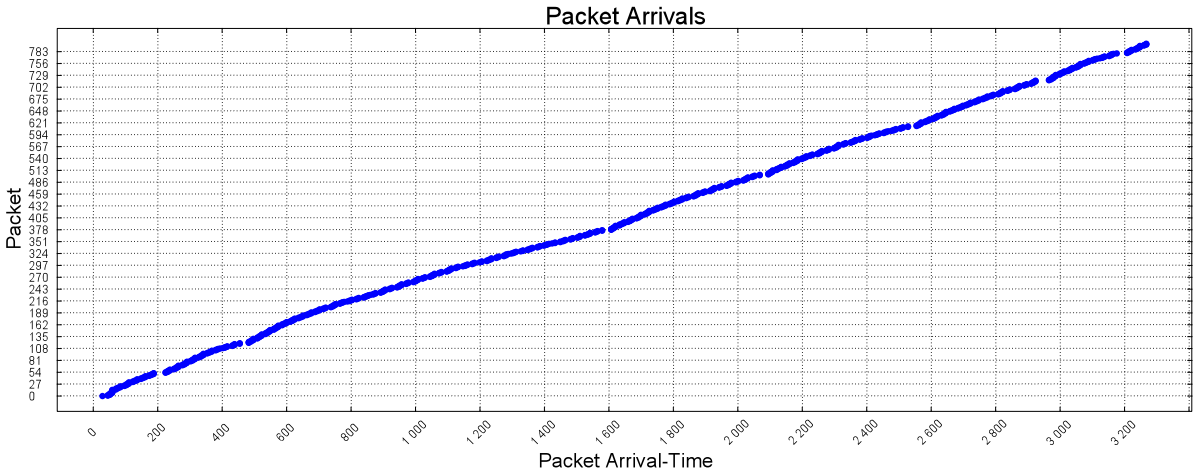
(b) Subflow 1

Variable	Result
λ	0.245 t
W	3.422 t
L	0.84 $Packets$
$Packets\ Sent$	802 $Packets$
$ACKs\ Received$	3348 $Packets$
RT	20 $Packets$
FRT	595 $Packets$
$Goodput$	0.245 $Packets/t$
$Loss\ rate$	76.7%

Table 5.7: MPTCP Heterogeneous Paths Low Loss Results

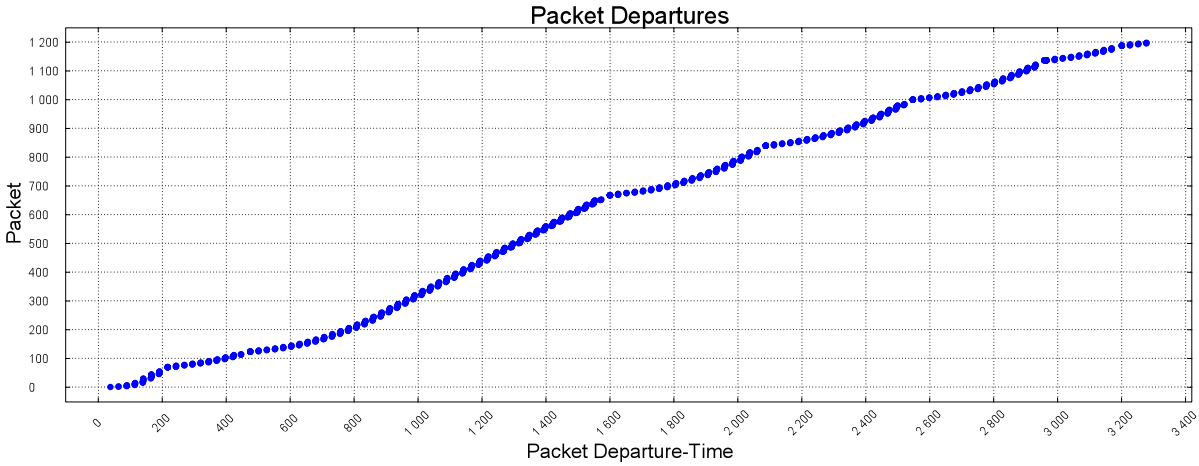


(a) Subflow 0

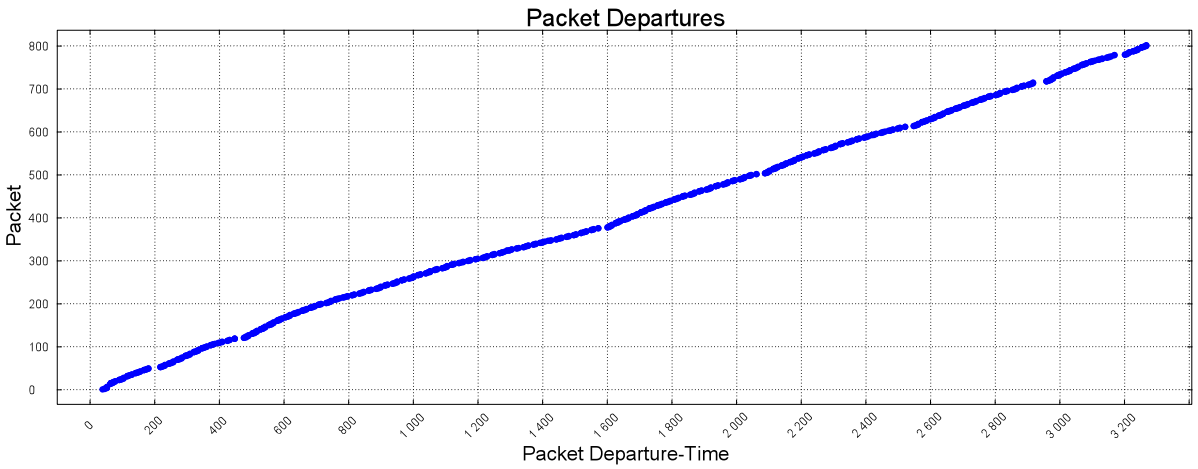


(b) Subflow 1

Figure 5.37: MPTCP Heterogeneous Paths Low Loss Packet Arrivals

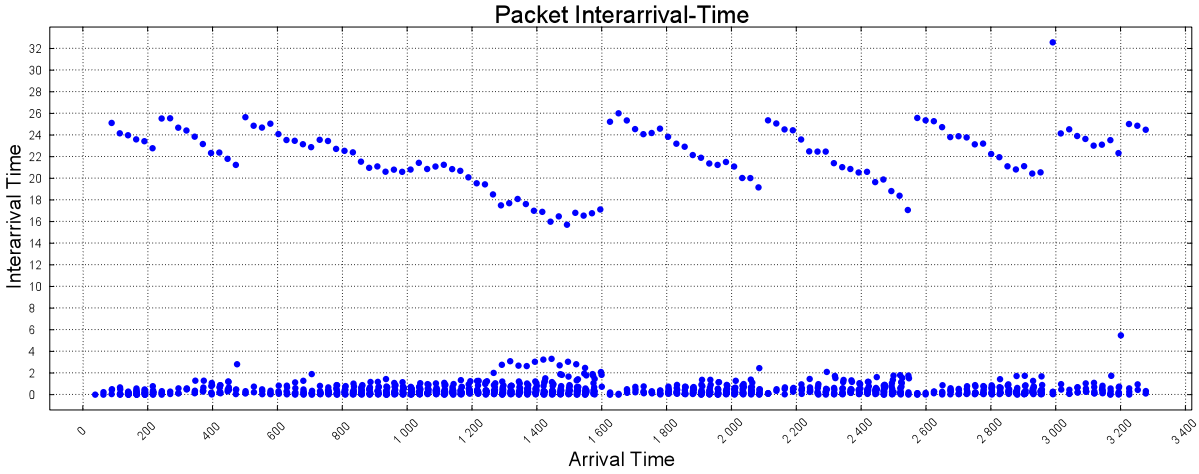


(a) Subflow 0

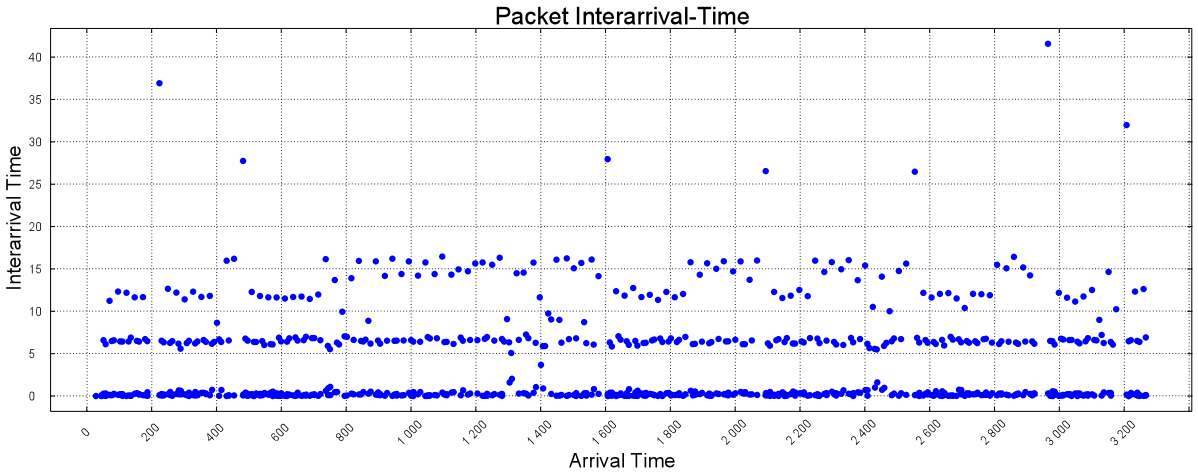


(b) Subflow 1

Figure 5.38: MPTCP Heterogeneous Paths Low Loss Packet Departures

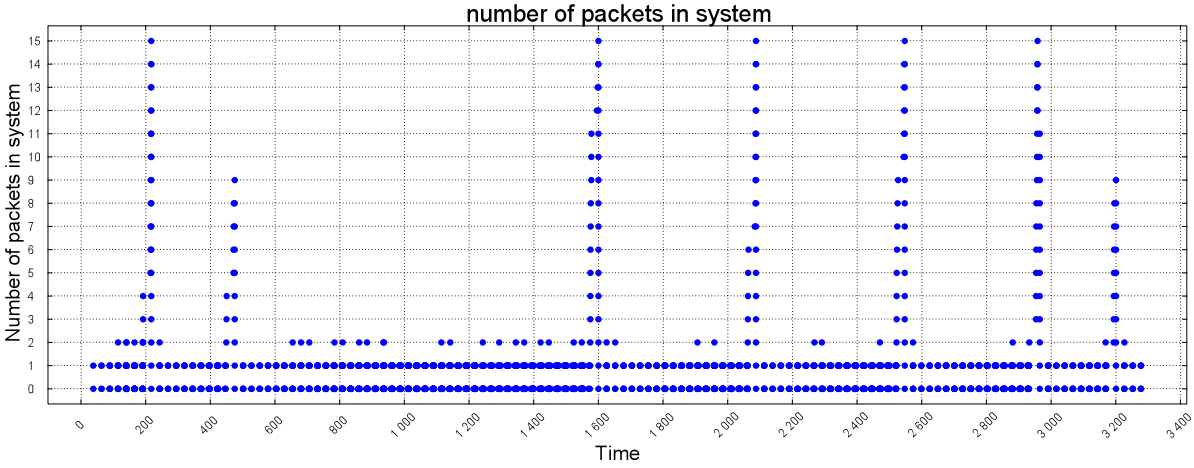


(a) Subflow 0

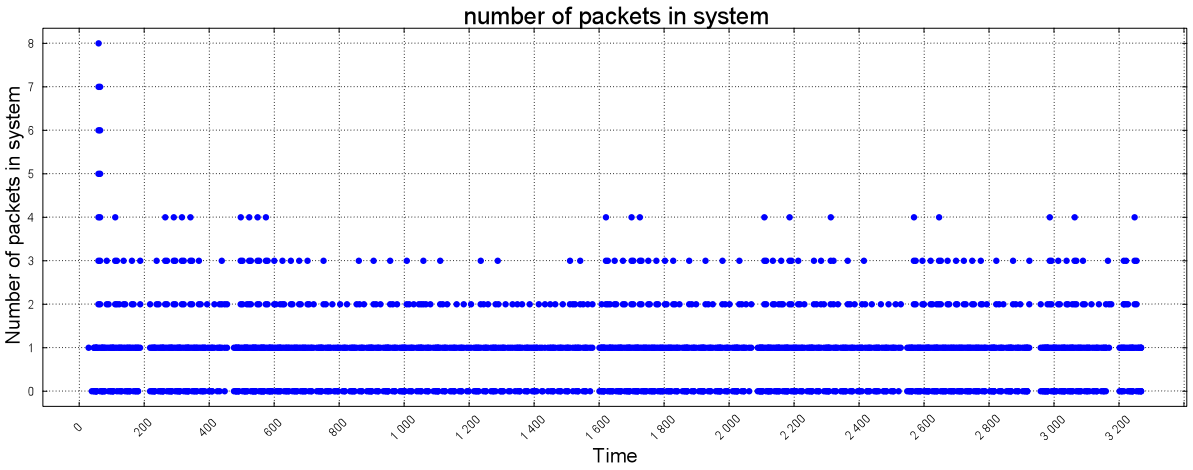


(b) Subflow 1

Figure 5.39: MPTCP Heterogeneous Paths Low Loss Packet Interarrival Time

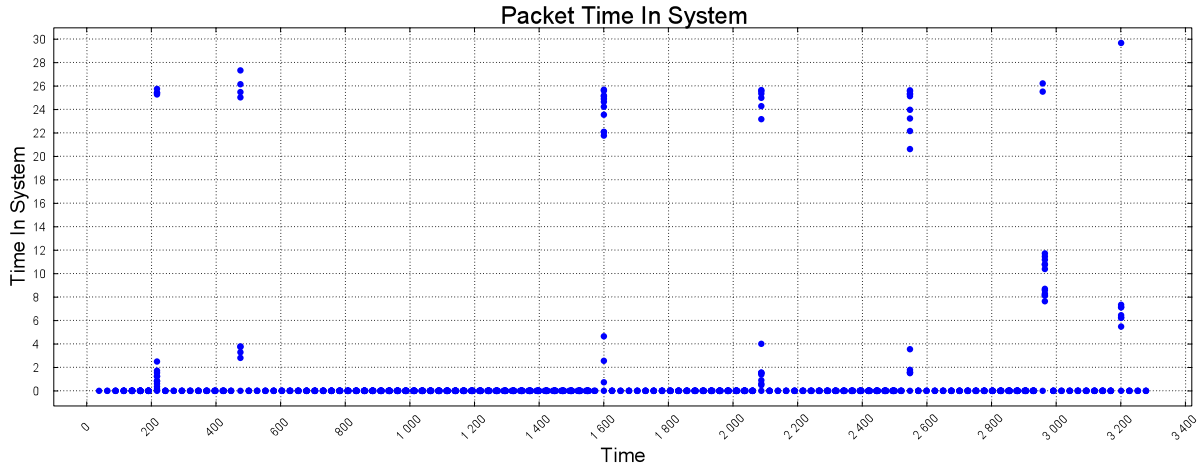


(a) Subflow 0

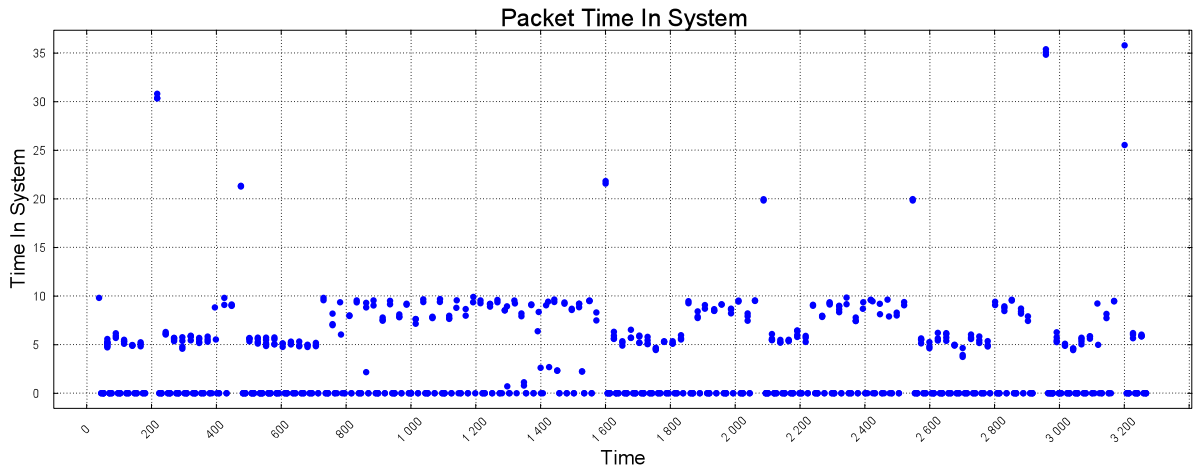


(b) Subflow 1

Figure 5.40: MPTCP Heterogeneous Paths Low Loss Packets in System

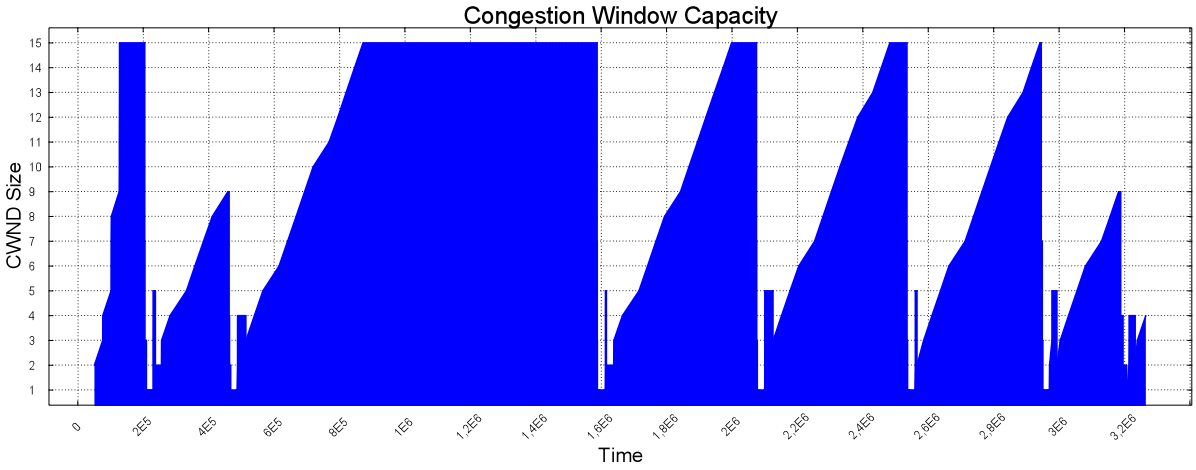


(a) Subflow 0

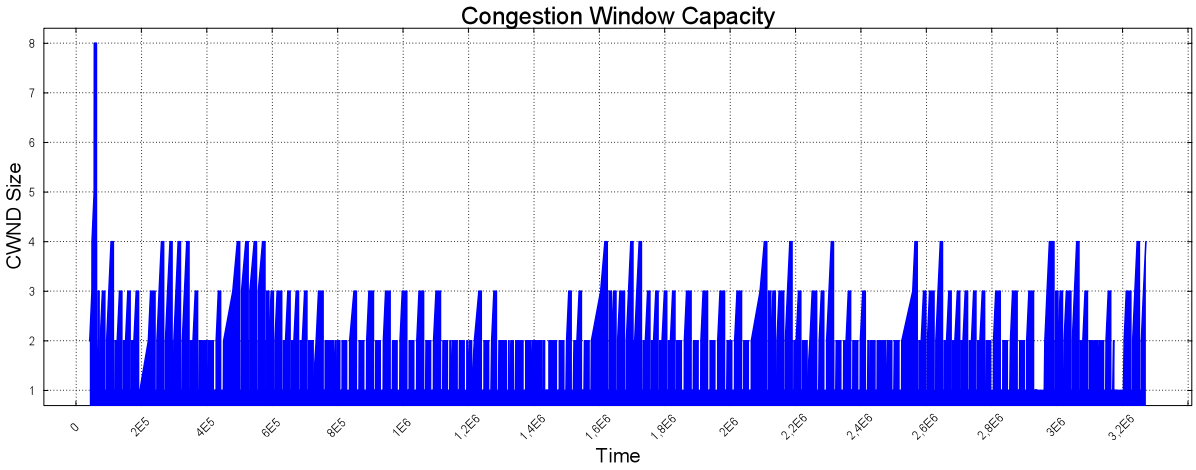


(b) Subflow 1

Figure 5.41: MPTCP Heterogeneous Paths Low Loss Packet Time in System



(a) Subflow 0



(b) Subflow 1

Figure 5.42: MPTCP Heterogeneous Paths Low Loss Congestion Window

5.3.4 Heterogeneous Paths High Loss

This test is using both the short path with high loss and the long path with high loss, making it a test of MPTCP on heterogeneous paths with high loss. It is natural to compare this test to the previous test on heterogeneous paths with low loss. Surprisingly, this test performs better when it comes to balancing the packets equally, with the short path (subflow 1) sending more packets this time. Subflow 1 also has a surprisingly lower loss rate, while subflow 0 has a much higher loss rate, as is expected. All results point to a better performing subflow 1 and a worse performing subflow 0 compared to the previous test. The results are for this reason pointing to better cooperation between the subflows when the loss of the channels are higher.

As discussed earlier in subsection 5.3.1 and subsection 5.3.2 the excessive amount of

retransmissions can cause problems and limit the other subflow. If the dominant subflow limits the other subflow, then loss occurring in the dominant subflow may allow the other subflow to send without as much interference. This may be the reason why subflow 1 is performing better in this test compared to the previous test subsection 5.3.3.

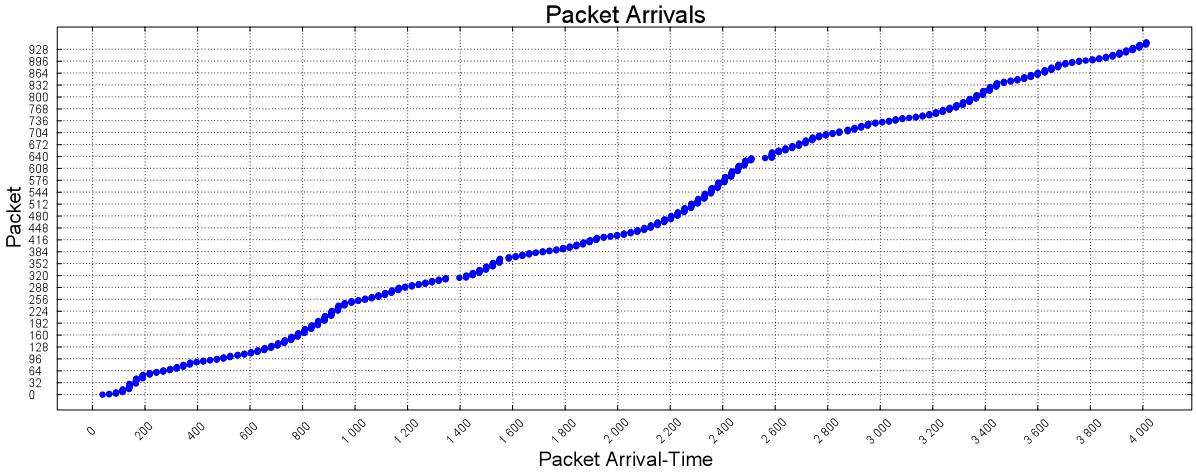
(a) Subflow 0

Variable	Result
λ	0.236 t
W	2.554 t
L	0.603 $Packets$
$Packets\ Sent$	948 $Packets$
$ACKs\ Received$	1249 $Packets$
RT	21 $Packets$
FRT	101 $Packets$
$Goodput$	0.235 $Packets/t$
$Loss\ rate$	12.9%

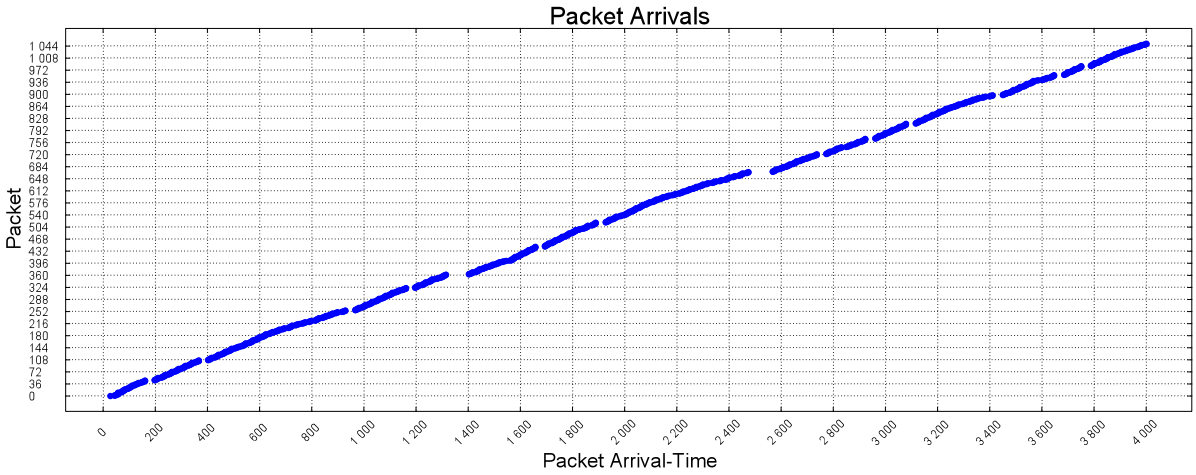
(b) Subflow 1

Variable	Result
λ	0.263 t
W	3.749 t
L	0.985 $Packets$
$Packets\ Sent$	1052 $Packets$
$ACKs\ Received$	3439 $Packets$
RT	68 $Packets$
FRT	559 $Packets$
$Goodput$	0.263 $Packets/t$
$Loss\ rate$	59.6%

Table 5.8: MPTCP Heterogeneous Paths High Loss Results

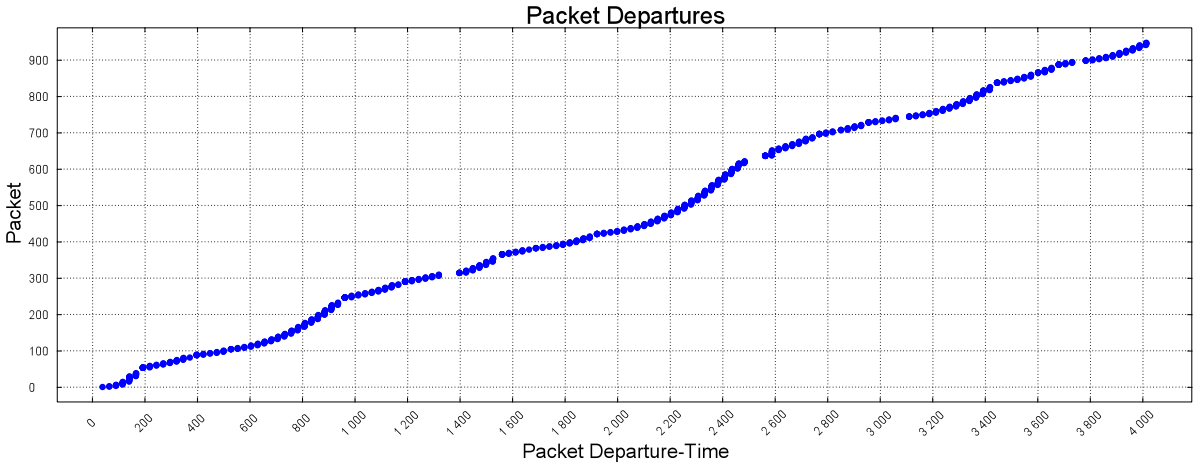


(a) Subflow 0

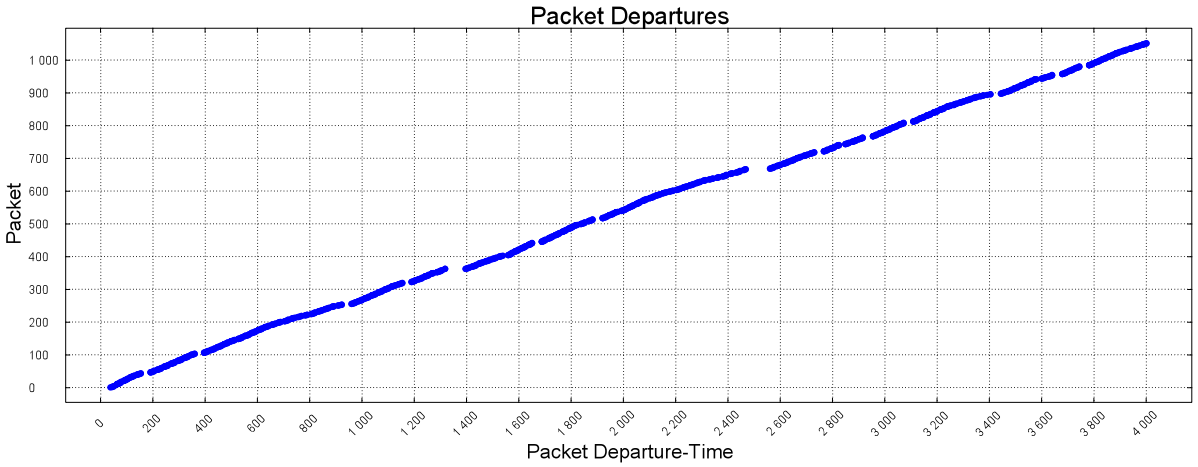


(b) Subflow 1

Figure 5.43: MPTCP Heterogeneous Paths High Loss Packet Arrivals

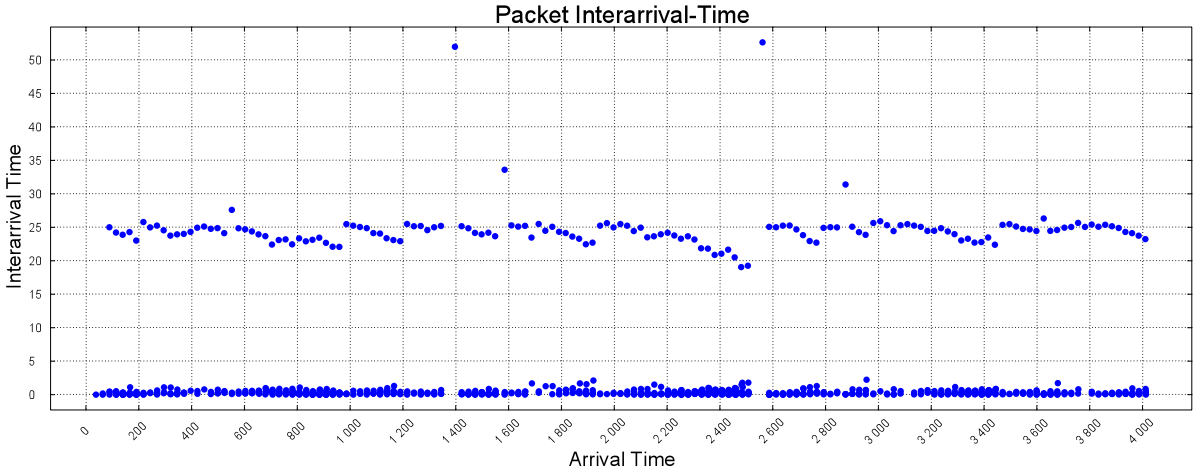


(a) Subflow 0

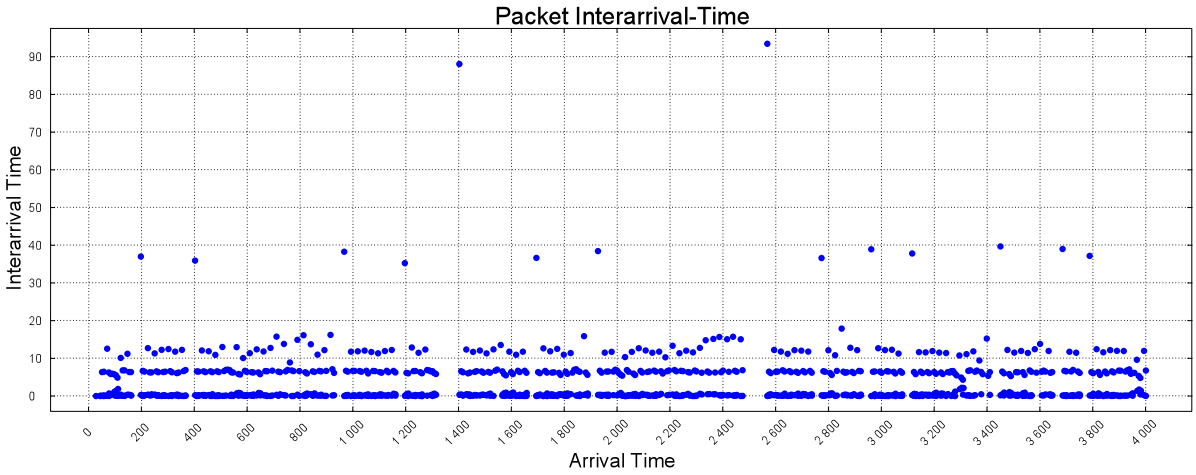


(b) Subflow 1

Figure 5.44: MPTCP Heterogeneous Paths High Loss Packet Departures

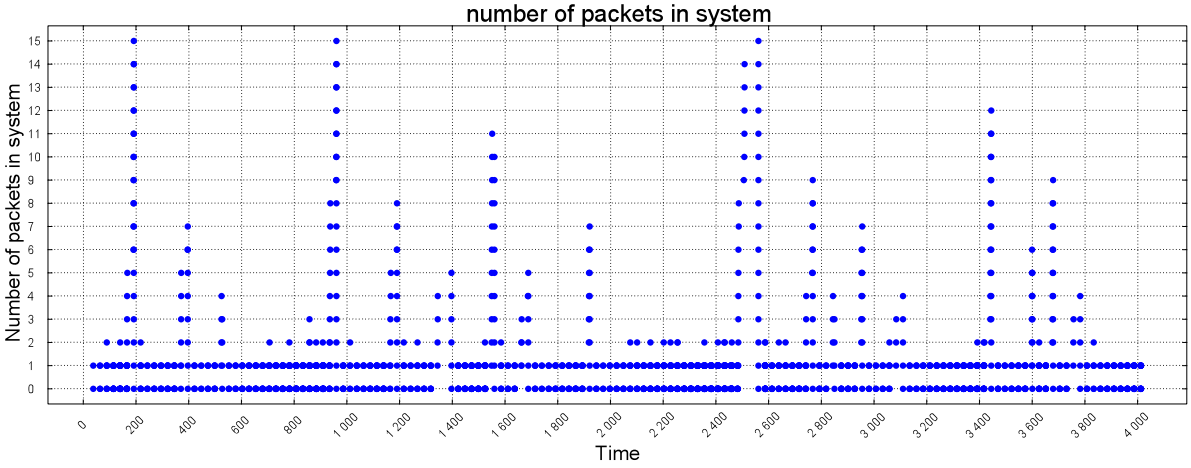


(a) Subflow 0

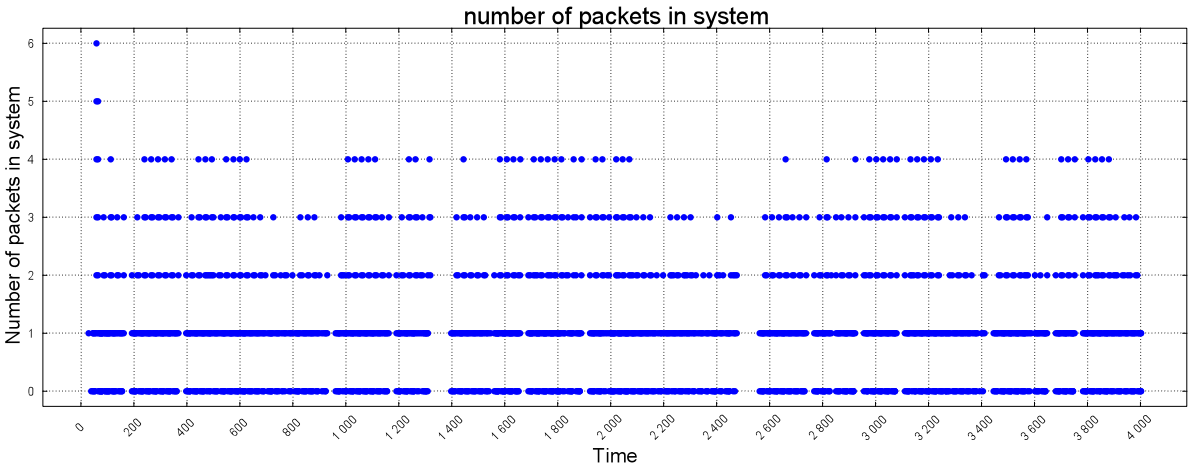


(b) Subflow 1

Figure 5.45: MPTCP Heterogeneous Paths High Loss Packet Interarrival Time

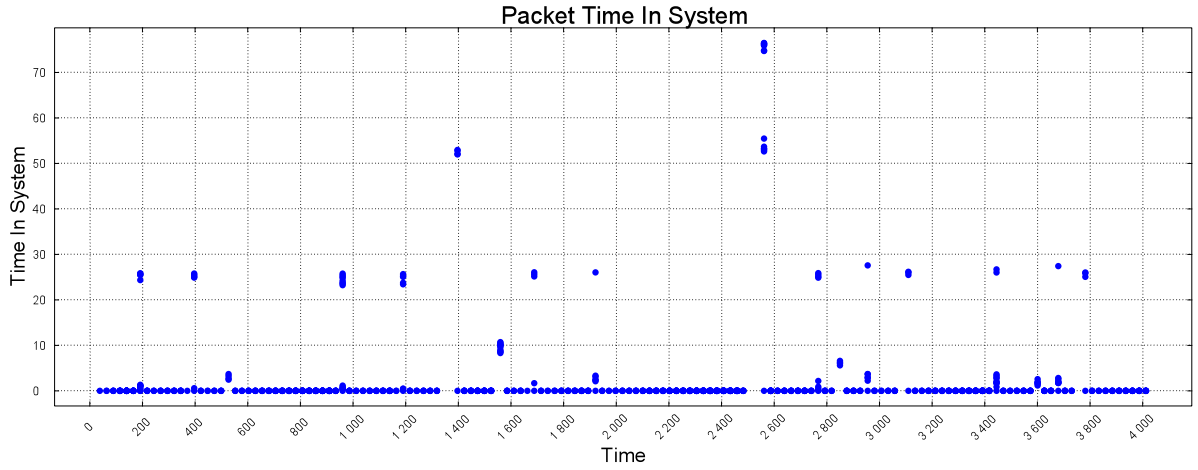


(a) Subflow 0

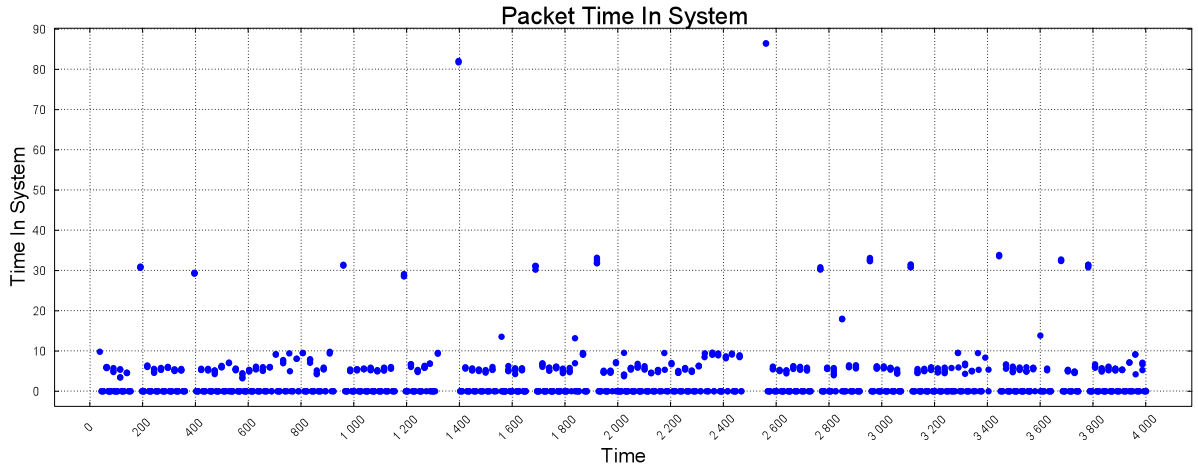


(b) Subflow 1

Figure 5.46: MPTCP Heterogeneous Paths High Loss Packets in System

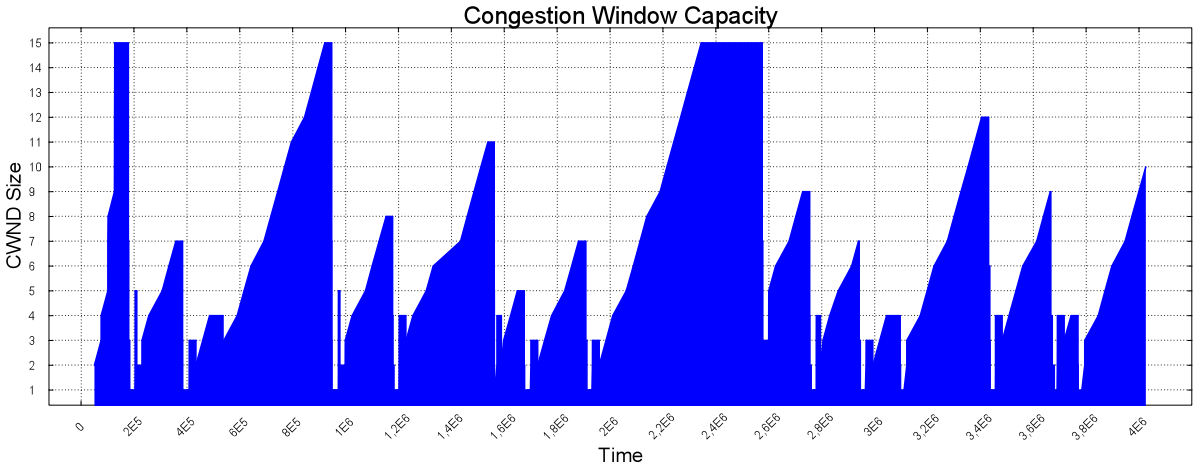


(a) Subflow 0

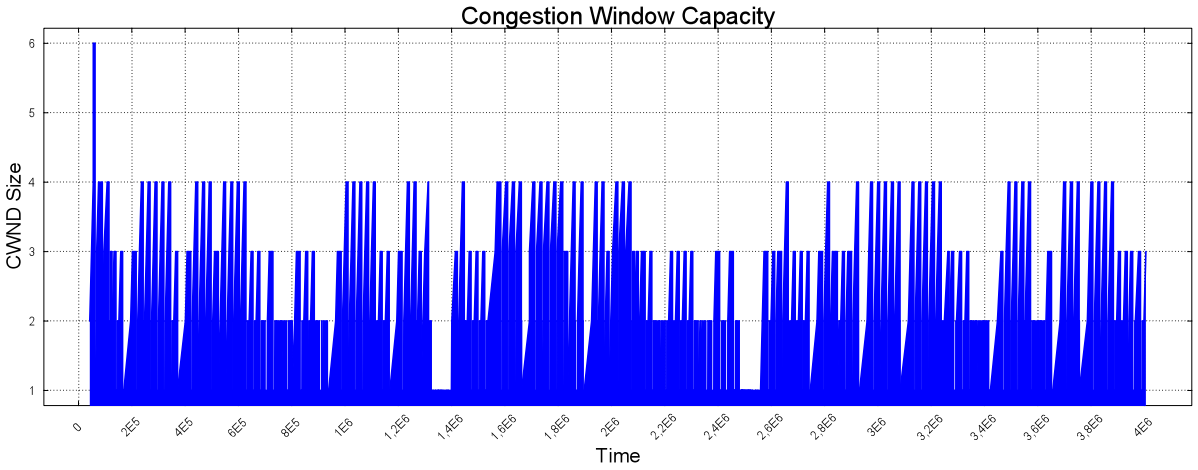


(b) Subflow 1

Figure 5.47: MPTCP Heterogeneous Paths High Loss Packet Time in System



(a) Subflow 0



(b) Subflow 1

Figure 5.48: MPTCP Heterogeneous Paths High Loss Congestion Window

5.4 Model Validity

A simulation has the inherent problem that all factors need to be defined in some way. However, in real life, variables or processes may be too complicated or have too many dependencies to measure correctly. Random generators are often used in simulators to model these variables or processes. It does not mean that the real-life processes are random, but rather that they are too complicated or impossible to model correctly. In the presented discrete event simulator, the loss occurs on a path and the queue size in routers, i.e., the queueing delay is decided by a random process using a random generator. This will never be equal to how a real-life situation is, but it can closely replicate the actual environment. The following sections will go through the modeling that is used and discuss the validity of the model.

5.4.1 Modeling Loss

Loss is modeled using Gilbert-Elliot burst channels implemented in the channels as explained in section 4.3. In the short paths (section 5.1.1 and section 5.1.1), the loss parameter is set in one channel, giving the path one model for loss in contrast to the long paths (section 5.1.1 and section 5.1.1) that have three channels with loss. The literature [31] [31] [32] only describes the use of Gilbert-Elliot channels as modeling the overall loss, however my implementation is using multiple burst models when using the long paths. Since communication over longer paths is prone to more error, more variation in RTT, and less predictability, it can be justified that multiple burst models make sense. Then again, this usage is not described in any literature found.

Further, the loss that occurs in real life is often due to congestion. However, when a wireless link is part of the path, noise and interference are often the cause of loss. The model does not differentiate the cause of loss. As a result, the modeled loss has no relation to the router's congestion at all. This is not realistic since many of the losses that occur in wired connections are from dropped packets. For this reason, it can be argued that there should be a relation between queuing delay and congestion in the network; however, this is not implemented.

Lastly, the Gilbert Elliot burst model is a simplification of how loss occurs. Other loss models could have been used, such as a realistic fading model [32]. That being said, the Gilbert-Elliot channel is simplistic, which in itself is beneficial.

5.4.2 Modeling Router Queue Size

The number of packets situated in the routers is modeled using a Poison process section 4.2. The Poisson process is known not accurately to model queues of routers [25]. However, as discussed earlier, it is chosen for its simplicity and because it is widely used as an arrival process in simulation. The queuing delay is described as the factor that has the most variance, thus contributing the most to the unpredictability of internet communication. For this reason, it is a problem that the accuracy of the Poisson process is inadequate. Also, as discussed in subsection 5.4.1 the size of the queues, and consequently the queuing delay, is not contributing to the loss modeling. As a result, the modeling of queue size and the modeling of queuing delay are not good enough to correctly model internet traffic. Nevertheless, for this simulation, it is sufficient as simplifications are made in every part of the simulator. As seen in section 5.2 the result for TCP resembles something similar to what can be thought of as communication over the internet with random components adding to the results. However, given more time, a better model with a relation between queue size and loss could be implemented.

5.4.3 Packet Size Distribution

It was decided not to model the size of packets to keep the simulator as simple as possible. That is why the packets in the model do not have a packet size associated with them.

However, in actual TCP traffic, the packet sizes are varying a lot causing varying transmission delays and processing delays [34]. In addition, small packets like ACKs should have less probability of error as they can more easily fit the buffers of congested routers and be less exposed to interference. In the simulator, ACKs are handled the same way as packets that contain larger payloads. For the reasons above, the modeled packets size, or rather the lack of modeled packets size, is a simplification of the model and needs to be taken into consideration when evaluating the results

5.4.4 Processing Delay

For the purpose of this simulator, it was decided to neglect the processing delay as it is described to be in the order of microseconds or less [2]. Other sources are, however, describing the processing delay to reach the magnitude of long-distance propagation delay in some cases where complex payload operations, content adaptation, or encryption is needed [26]. Regardless, the fact that the packet processing delay is neglected is a shortcoming of the model and must be considered when evaluating the presented results.

5.4.5 TCP as Subflow

Using TCP directly in a naive way presented some expected issues and unexpected issues that notably reduced the potential performance of the MPTCP implementation.

Scheduling the TCP subflows according to RTT or other performance metrics to receive packets in order is an active research topic and was not implemented [22] [21]. Another solution to this is a pre-division of the receiving buffer based on RTT [23]. The presented model uses the latter but divides the buffer in half regardless of RTT or other metrics. This simplifies the model, limiting the overall performance of MPTCP, especially in heterogeneous paths where delay differences become essential. Consequently, the time in system metric W increases in the heterogeneous test compared to homogeneous tests, being an indicator of packets needing to wait for other packets, i.e., packets out of order.

An unexpected effect caused by using TCP directly in an MPTCP implementation was ARQ combined with congestion control. TCP is implemented to use the acknowledgments or an absence of acknowledgment as indicators of loss, giving ACKs two responsibilities, namely, (1) avoiding congestion and being fair to other users, and (2) providing flow control that guarantees that packets are delivered in the correct order. In an MPTCP use case, this is not working. To illustrate the problem, suppose we have subflow A. A has sent all its packets and is waiting for an ACK. This ACK will signal the A to send more packets. The receiver has received all packets from A, but the packets from another subflow B are missing. Suppose the packets from A are to be received before packets from B. Then only a DupACK can be sent to the sender, effectively telling the sender that loss has occurred even though it's an out-of-order problem. When B finally can send all packets successfully, A must receive an ACK to, in turn, continue to send as normal. This leads to a precarious situation where the subflows must trigger ACKs going to other subflows.

At the same time, ACKs are flow specific and will trigger congestion control mechanisms. One of the goals of MPTCP is to effectively use two or more paths to send data faster and more reliable. This implementation does not achieve this. The main reason is the ambiguity of an ACK and how MPTCP is implemented in the simulator.

5.4.6 Discrete Event Simulation

Discrete event simulators have some properties that make them useful when simulating events that occur in time. This is presented in detail in section 4.1. However, simulation and modeling is, by nature, a simplification of actual processes. For this reason, it should be analyzed with this in mind.

Time is modeled using a logical clock with an imaginary unit simulated as microseconds to scale the delays appropriate to each other. The time unit is not essential, but the proportional delays are. The delays, or time until an event happens, are modeled using approximations from the literature [26] [2], and will inherently have smaller inconsistencies. The high dynamic variance in queuing delays and the variance between channel propagation delays are hard to model. Also, the transmission and processing delays are small but should vary based on packet size and other factors. Together the mentioned inconsistencies can add up to significant disproportions that can affect the validity of the results. Since the delays are what drives time in the model, and the fact that the modeled delays are inconsistent with real life, the simulated time can not be compared with real-life time. Consequently, results that depend on time, such as goodput, can not be compared with results from outside the simulated environment.

5.5 Evaluation of Data

The data gathered from the test runs are based on the implementation of the discrete event simulator and the way MPTCP and TCP were implemented. As discussed in section 5.4 there are several simplifications made to the model as well as the fact that the simulator is only replicating the behavior of real-life communication over the internet and consequently how TCP and MPTCP will behave.

The tests performed on TCP in section 5.2 was in mainly completed to test the implementation of TCP and the underlying network put together by routers and channels. The results and behavior of standard TCP are well known and were a good way to validate how the network behaved. With results that came close to the expedited output, the model showed promise.

The MPTCP implementation lacks several important features to model how a real-life implementation could look like. This taken into consideration, the simulation results are pointing to many of the known problems or obstacles that MPTCP faces. Combined with the promising results from the TCP implementation, this is, to a degree, validating the model. That being said, the results from the simulation must be seen in the light of all simplifications and imperfections that come with simulation and modeling as a whole.

Further, the data should only be used to find broader relations and should not focus on details.

Chapter 6

Conclusion

This chapter present the conclusion of the thesis including several finding in addition to suggested future work.

6.1 Research Conclusion

This thesis presents a discrete event simulator used in modeling networks with different properties to analyze MPTCP. The results are compared with an implementation of TCP over the same paths to validate and support the achieved results. To a large extent, the gathered data are just scratching the surface of bigger and more complicated issues associated with the making of MPTCP. Despite the inherent simplifications in the mode, many of the papers mentioned in chapter 3 are pointing to the same problems as the results derived from the simulator. Papers presenting results from simulation and real-world tests point to the out-of-order problem and the load imbalance problem. The results from the tests conducted in this thesis are showing the same. Both issues have the potential to reduce the usability of MPTCP significantly. In addition, both problems can amplify each other, making it imperative to mitigate the problems. Furthermore, the aforementioned DupACK problem has unfolded a significant characteristic of MPTCP that fundamentally change the way flow control and congestion control should behave. The following sections will describe the issues in detail and explain how the problems can arise, how it's detected, and how to potentially overcome the problems.

6.2 Load Imbalance Problem

The load imbalance problem is the problem of balancing the sending rate so that packets are received as ordered as possible, even if the packets come from different paths with different proprieties. If this is not handled according to the performance of each subflow, packets from one flow may arrive too early, causing the out-of-order problem. This problem becomes evident in the presented simulation when looking at the heterogeneous path

tests in subsection 5.3.4 and subsection 5.3.3. The subflow sending over the shorter path is sending significantly faster but is held back by a slower subflow caused by the lack of scheduling or buffer division. The slower subflow, on the other hand, is trying to catch up with the faster subflow but will always lag behind, causing a degradation of the potential goodput that can be achieved.

6.3 Out-of-Order Problem

The out-of-order problem is caused by receiving packets out of order. Packets must be received in order; consequently, the out-of-order packets must wait in the receiving buffer until the block is resolved. In addition, packets must be reordered at the receiver's side, making the communication more costly. Further, the packet waiting time is increased, which leads to higher latency. A bad load balancing at the sender side, or rather lack of balancing, will result in more packets received out of order, especially in heterogeneous networks. This leads to unnecessary buffer bloat and higher latency caused by the subflow which is lagging behind, ultimately reducing the potential goodput. This problem is particularly evident in the time in system and number of packets in system graphs for the heterogeneous path tests in subsection 5.3.4 and subsection 5.3.3. Comparing the time in the system graph, the number of packets in system graphs, the average time in the system W, and the average number of packets in the system L to the paths with homogeneous paths, it is clear that packets are to a higher degree received out of order. This is due to the better balancing of load when the paths are equal due to the naive 50-50 split of the receive buffer.

6.4 Redundant ACK

The redundant ACK problem explained in subsection 5.4.5 is caused by the ambiguity of ACKs in TCP. This can be seen as small spikes in the congestion window plots caused by the reduction of CWND provoked by the received ACK. Implementing MPTCP should, as a consequence, be able to distinguish the meaning of the received ACK. A concept of seen packets has been proposed by Sundararajan et al.[35] as a solution to the ambiguity of ACKs in coded TCP. This solution can be used in the context of MPTCP as well. The implementation can distinguish between ACKs representing successful sending and ACKs indicating seen packets from the other subflows if this concept is used. ACKs triggered from the other subflows, or rather information from the other subflows, will with this concept never interfere with other subflows congestion control mechanisms, effectively solving the problem. That being said, this concept does not solve the other introduces obstacles alone.

6.5 Possible Solutions

As discussed, the literature and the results gathered from the implemented simulator are pointing to the two presented problems, namely the out-of-order problem and the load imbalance problem. The given redundant ACK problem can be solved using concepts from the coded-TCP research. It turns out that using error-correcting codes can indeed solve, or at least alleviate some of the issues concerning the two problems [19] [20]. Presented in chapter 3 are two papers suggesting network coding as a solution to ordering and load balancing, i.e., the out-of-order problem and load imbalance problem. Additionally, since lost or delayed packets can be reconstructed, the need for retransmissions is reduced. For the same reason, packet reordering is less of an issue.

The Linux kernel implementation has achieved a working and deployable solution to the problems mentioned. However, this implementation which is based on RFC8684 [9], is outperformed in some papers [23] [19] showing that there is still room for improvements. From the results gathered in this thesis and the promise that MPTCP with network code shows, it seems to be a good solution to the presented problems.

6.6 Future Work

Due to lack of time, some adaptations, implementations, and experiments have been left for future work. Several different ideas would have been interesting to analyze; however, completing this was not feasible due to the time constraints. Therefore, the main focus of this thesis was to finish a working discrete event simulator, implement a versatile simulated network with interchanging possibilities, and implement TCP and MPTCP for analyzing purposes. The versatility of the simulated network was important during the early development as modeling different loss and arrival processes to the routers were planned. As the deadline came closer, though, these planes were neglected and thus left as future work. The redundant ACKs discussed in several sections earlier are probably the main priority to fix, as the results would be more precise. Also, a sending scheduler or a better pre-divining of receive-buffer is high on the priority list of improvements that could rather easily be implemented. Given more time, here are some of the ideas that could be interesting to test:

1. The queue utilization of routers on the internet is hard to model as most traffic is TCP traffic. TCP traffic has Long-Range Dependence and is adjusting sending rate according to the sending environment i.e., the path. With minor adaptations, the simulator could facilitate simulated TCP traffic over the routers using multiple TCP implementations, possibly making the model more realistic regarding queueing delay and congestion in routers. In addition, this allows for analyzing the impact MPTCP has on the performance of the router. Using results from Jackson networks and the Pollaczek-Khinchin equations described in queueing theory [12] and section 2.4.5, many essential metrics of the routers queueing systems could be analyzed, giving results about the network as a whole.

2. Another point of interest is the importance of packet sizes regarding transmission and processing delays. For this reason, it could be interesting to use a distribution to model the packet sizes and use this when modeling the transmission and processing delay. By doing this, the added benefit of having processing delay as a factor is added, which in some cases is described to reach the magnitude of long-distance propagation delay [26].

3. The current implementation of MPTCP uses a receive buffer with pre-division-based flow control. However, the implementation is not basing the split on any metric but is splitting equally amongst the available subflows. It would be of interest to test out using different metrics like RTT to separate the buffer according to the performance of the subflows. Alternatively, it could be interesting to implement a sending scheduler and compare the results to the other implementations. Looking at how this will affect the load imbalance problem and inherently the out-of-order problem is of great interest.

4. Finally, the promise of MPTCP with network codes is compelling and would be interesting to analyze and compare against the results achieved in this thesis. This is perhaps the most demanding task of the ones described; however, the results can help in the evolution of the protocol.

Nevertheless, the simulator has produced results that raise the same issues as other researchers have found. Finding reasonable solutions to the problems is, however, a far more difficult task. More research is therefore needed in the field to find the optimal solutions.

Acronyms

4G Fourth generation of broadband cellular network technology. 7, 15

5G Fifth generation of broadband cellular network technology. 1, 7

ACK Acknowledgement. 12, 13, 39–41, 50, 56, 90, 91, 94, 95

AIMD Additive-Increase/Multiplicative-Decrease. 13

ARQ Automatic Repeat Request. 12, 13, 39, 40, 90

CWND Congestion Window. 13–15, 40, 94

DupACK Duplicate Acknowledgement. 12–15, 40, 47, 60, 67, 93

FEC Forward Error Correction. 12

FIFO First In First Out. 16, 24, 25

FMTCP Fountain Code-Based MultiPath Transmission Control Protocol. 31

GB Gigabyte. 44

IP Internet Protocol. 7, 9, 10

LRD Long-Range Dependence. 36, 95

MPTCP MultiPath TCP. 1, 7–9, 15, 16, 31–33, 37, 41, 42, 44–46, 59, 60, 68, 74, 81, 90, 91, 93–96

MTU Maximum Transmission Unit. 10

NAT Network Address Translation. 16

PASTA Poisson Arrivals See Time Averages. 21, 25, 36

RTO Retransmission Timeout. 13–15, 40, 41, 47, 53, 60, 74

RTT Round Trip Time. 14, 40–42, 89, 90, 96

RWND Receiver Window. 12, 13, 50

SYN Synchronize. 12

TCP Transmission Control Protocol. 1, 9–12, 14–16, 31–33, 36, 39–42, 44–47, 59, 60, 68, 74, 75, 89–91, 93–95

UDP User Datagram Protocol. 10

VoIP Voice over Internet Protocol. 7

WiFi A family of wireless network protocols. 7, 15, 31

Bibliography

[1] Robert T. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122. URL: https://rfc-editor.org/rfc/rfc1122.txt.

[2] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. Pearson, 2012. ISBN: 0132856204.

[3] Go Hasegawa, Masayuki Murata, and Hideo Miyahara. "Fairness and stability of congestion control mechanisms of TCP." In: *Telecommunication Systems* 15 (Nov. 2000), pp. 167–184. DOI: 10.1023/A:1019186710820.

[4] Lorenzo Vicisano et al. *The Use of Forward Error Correction (FEC) in Reliable Multicast*. RFC 3453. Dec. 2002. DOI: 10.17487/RFC3453. URL: https://rfc-editor.org/rfc/rfc3453.txt.

[5] Ying-zong Huang, Sanjeev Mehrotra, and Jin li. "A hybrid FEC-ARQ protocol for low-delay lossless sequential data streaming." In: June 2009, pp. 718–725. DOI: 10.1109/ICME.2009.5202596.

[6] *TCP Congestion Control*. 2017. URL: https://yazilimcorbasi.blogspot.com/2016/12/tcp-congestion-control.html (visited on 05/31/2021).

[7] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: https://rfc-editor.org/rfc/rfc5681.txt.

[8] T Nishitha. "A Comparative Analysis of TCP Tahoe, Reno, New- Reno, SACK and Vegas in Homogeneous Networks." In: 2002.

[9] Alan Ford et al. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Mar. 2020. DOI: 10.17487/RFC8684. URL: https://rfc-editor.org/rfc/rfc8684.txt.

[10] Kae Won Choi et al. "Optimal load balancing scheduler for MPTCP-based bandwidth aggregation in heterogeneous wireless environments." In: *Computer Communications* 112 (2017), pp. 116–130. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2017.08.018. URL: https://www.sciencedirect.com/science/article/pii/S0140366417302426.

[11] Costin Raiciu et al. "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP." In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 399–412. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu.

[12] Donald Gross John F. Shortle James M Thompson and Carl M Harris. *Fundamentals Of Queueing Theory*. 5th ed. Wiley Series in Probability and Statistics. Wiley, 2018. ISBN: 9781118943526.

[13] Ronald W. Wolff. "Poisson Arrivals See Time Averages." In: *Operations Research* 30.2 (1982), pp. 223–231. DOI: 10.1287/opre.30.2.223. eprint: https://doi.org/10.1287/opre.30.2.223. URL: https://doi.org/10.1287/opre.30.2.223.

[14] John D. C. Little. "A PROOF FOR THE QUEUING FORMULA: $L = \lambda W$." In: (Nov. 1960). DOI: http://fisherp.scripts.mit.edu/wordpress/wp-content/uploads/2015/11/ContentServer.pdf.

[15] D. V. Lindley. "The theory of queues with a single server." In: *Mathematical Proceedings of the Cambridge Philosophical Society* 48.2 (1952), pp. 277–289. DOI: 10.1017/S0305004100027638.

[16] Jerry Banks et al. *Discrete-Event System Simulation*. July 2009. ISBN: 9780136062127.

[17] Keld Helsgaun. "Discrete Event Simulation in Java." In: (June 2000).

[18] et al. C. Paasch S. Barre. *MultiPath TCP - Linux Kernel implementation*. URL: https://www.multipath-tcp.org/ (visited on 05/24/2021).

[19] Yong Cui et al. "FMTCP: A Fountain Code-Based Multipath Transmission Control Protocol." In: *IEEE/ACM Transactions on Networking* 23.2 (2015), pp. 465–478. DOI: 10.1109/TNET.2014.2300140.

[20] Jason Cloud et al. "Multi-Path TCP with Network Coding for Mobile Devices in Heterogeneous Networks." In: *2013 IEEE 78th Vehicular Technology Conference (VTC Fall)*. 2013, pp. 1–5. DOI: 10.1109/VTCFall.2013.6692295.

[21] Vivek Adarsh, Paul Schmitt, and Elizabeth Belding. "MPTCP Performance over Heterogenous Subpaths." In: July 2019, pp. 1–9. DOI: 10.1109/ICCCN.2019.8847086.

[22] Vu Vu and Brenton Walker. "Redundant Multipath-TCP Scheduling with Desired Packet Latency." In: Oct. 2019, pp. 7–12. ISBN: 978-1-4503-6933-6. DOI: 10.1145/3349625.3355440.

[23] Jiangping Han et al. "Receive Buffer Pre-division Based Flow Control for MPTCP." In: Mar. 2018, pp. 19–31. ISBN: 978-981-10-8889-6. DOI: 10.1007/978-981-10-8890-2_2.

[24] Prateek Sharma. "Discrete-Event Simulation." In: *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH* 4 (Apr. 2015).

[25] L. Muscariello et al. "Markov models of internet traffic and a new hierarchical MMPP model." In: *Computer Communications* 28.16 (2005), pp. 1835–1851. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2005.02.012. URL: https://www.sciencedirect.com/science/article/pii/S0140366405000824.

[26] R. Ramaswamy, Ning Weng, and T. Wolf. "Characterizing network processing delay." In: *IEEE Global Telecommunications Conference, 2004. GLOBECOM '04.* Vol. 3. 2004, 1629–1634 Vol.3. DOI: 10.1109/GLOCOM.2004.1378257.

[27] Nicolas Hohn et al. "Bridging router performance and queuing theory." In: vol. 32. June 2004, pp. 355–366. DOI: 10.1145/1005686.1005728.

[28] Keyur Parikh and Junius Kim. "The Role of Network Packet Loss Modeling in Reliable Transport of Broadcast Audio." In: (). URL: https://www.gatesair.com/documents/papers/Parikh-K130115-Network-Modeling-Revised-02-05-2015.pdf.

[29] E. N. Gilbert. "Capacity of a burst-noise channel." In: *The Bell System Technical Journal* 39.5 (1960), pp. 1253–1265. DOI: 10.1002/j.1538-7305.1960.tb03959.x.

[30] E. O. Elliott. "Estimates of error rates for codes on burst-noise channels." In: *The Bell System Technical Journal* 42.5 (1963), pp. 1977–1997. DOI: 10.1002/j.1538-7305.1963.tb00955.x.

[31] Gerhard Haßlinger and Oliver Hohlfeld. "The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet." In: Jan. 2008, pp. 269–286.

[32] Joseph Kang, Wayne Stark, and Alfred Hero. "Turbo Codes for Fading and Burst Channels." In: (Feb. 1999).

[33] Matt Sargent et al. *Computing TCP's Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298. URL: https://rfc-editor.org/rfc/rfc6298.txt.

[34] Rishi Sinha, Christos Papadopoulos, and John Heidemann. *Internet Packet Size Distributions: Some Observations*. Tech. rep. ISI-TR-2007-643. Orignally released October 2005 as web page http://netweb.usc.edu/%7ersinha/pkt-sizes/. johnh: pafile: USC/Information Sciences Institute, May 2007. URL: http://www.isi.edu/%5C%7ejohnh/PAPERS/Sinha07a.html.

[35] Jay Sundararajan et al. "Network Coding Meets TCP: Theory and Implementation." In: *Proceedings of the IEEE* 99 (Apr. 2011), pp. 490–512. DOI: 10.1109/JPROC.2010.2093850.

Appendix

```java
package org.example.data;


public enum Flag {
    ACK,
    SYN,
    FIN;
}
```

```java
package org.example.data;

public class Message implements Payload {

    private String msg;

    public Message(String msg) {
        this.msg = msg;
    }

    @Override
    public int size() {
        return this.msg.length();
    }

    @Override
    public String toString() {
        return this.msg;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Payload) return this.toString().equals(obj.toString());
        return false;
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

```java
package org.example.data;

import org.example.network.interfaces.Endpoint;

import java.util.List;

public class Packet {
    private final Endpoint destination;
    private final Endpoint origin;
    private final List<Flag> flags;
    private final Payload payload;
    private final int index;

    private final int sequenceNumber;
    private final int acknowledgmentNumber;


    protected Packet(Endpoint destination, Endpoint origin, List<Flag> flags,
        Payload payload, int sequenceNumber, int acknowledgmentNumber, int index) {
        this.destination = destination;
        this.origin = origin;
        this.flags = flags;
        this.payload = payload;
        this.index = index;

        this.sequenceNumber = sequenceNumber;
        this.acknowledgmentNumber = acknowledgmentNumber;
    }

    public boolean hasAllFlags(Flag... flags) {
        if (flags == null || this.flags == null) {
            throw new NullPointerException("flags can't be null");
        }

        if (this.flags.isEmpty()) {
            // flags given are not in the list
            return false;
        }

        if (flags.length == 0) {
            // no flags given implies that all flags given are in list
            return true;
        }

        boolean hasFlag = this.flags.contains(flags[0]);
        for (var i = 1; i < flags.length; i++) {
            hasFlag &= this.flags.contains(flags[i]);
        }
        return hasFlag;
    }

    public Payload getPayload() {
        return payload;
    }

    public Endpoint getDestination() {
        return this.destination;
    }

    public Endpoint getOrigin() {
        return this.origin;
    }
```

```
62
63
64      public int getSequenceNumber() {
65          return this.sequenceNumber;
66      }
67
68      public int getIndex() {
69          return this.index;
70      }
71
72      public int getAcknowledgmentNumber() {
73          return acknowledgmentNumber;
74      }
75
76      public int size() {
77          if (this.payload == null) return 0;
78          return payload.size();
79      }
80
81      protected List<Flag> getFlags() {
82          return flags;
83      }
84
85      @Override
86      public String toString() {
87          String returnString;
88          if (this.payload == null) {
89              returnString = this.flags.toString();
90          } else {
91              returnString = "[" + this.payload.toString() + "]";
92          }
93          returnString += "[seq: " + this.getSequenceNumber() + "]" + "[index: " +
                this.getIndex() + "]";
94
95          return returnString;
96      }
97
98      @Override
99      public int hashCode() {
100         return super.hashCode();
101     }
102
103     @Override
104     public boolean equals(Object o) {
105         if (this == o) return true;
106         if (o == null || getClass() != o.getClass()) return false;
107         var packet = (Packet) o;
108         return sequenceNumber == packet.sequenceNumber
109                 && acknowledgmentNumber == packet.acknowledgmentNumber
110                 && destination.equals(packet.destination)
111                 && origin.equals(packet.origin)
112                 && packet.hasAllFlags(this.flags.toArray(Flag[]::new));
113     }
114 }
```

```java
package org.example.data;

import org.example.network.interfaces.Endpoint;
import org.example.protocol.Connection;

import java.util.ArrayList;
import java.util.List;


public class PacketBuilder {
    private Endpoint destination = null;
    private Endpoint origin = null;
    private List<Flag> flags = new ArrayList<>();
    private Payload payload = null;
    private int index = 0;

    private int sequenceNumber = -1;
    private int acknowledgmentNumber = -1;

    public Packet build() {
        if (!this.hasFlag(Flag.ACK)) {
            this.acknowledgmentNumber = -1;
        }
        return new Packet(this.destination, this.origin, this.flags, this.payload,
            this.sequenceNumber, this.acknowledgmentNumber, this.index);
    }

    public Packet ackBuild(Packet packetToAck) {
        if (packetToAck.getOrigin() == null) throw new IllegalArgumentException("no
            origin");
        if (packetToAck.getDestination() == null) throw new IllegalArgumentException
            ("no destination");

        this.withDestination(packetToAck.getOrigin());
        this.withOrigin(packetToAck.getDestination());
        this.withFlags(Flag.ACK);
        this.withPayload(null);
        this.withAcknowledgmentNumber(packetToAck.getSequenceNumber() + 1);
        this.withSequenceNumber(packetToAck.getSequenceNumber());
        this.withIndex(packetToAck.getIndex());
        return new Packet(this.destination, this.origin, this.flags, this.payload,
            this.sequenceNumber, this.acknowledgmentNumber, this.index);
    }

    public PacketBuilder withFlags(Flag... flags) {
        for (Flag flag : flags) {
            if (this.flags.contains(flag)) continue;
            this.flags.add(flag);
        }
        return this;
    }

    public PacketBuilder withPayload(Payload payload) {
        this.payload = payload;
        return this;
    }

    public PacketBuilder withOrigin(Endpoint self) {
        this.origin = self;
        return this;
    }

```

```
59      public PacketBuilder withDestination(Endpoint destination) {
60          this.destination = destination;
61          return this;
62      }
63
64      public PacketBuilder withSequenceNumber(int sequenceNumber) {
65          this.sequenceNumber = sequenceNumber;
66          return this;
67      }
68
69      public PacketBuilder withAcknowledgmentNumber(int acknowledgmentNumber) {
70          this.acknowledgmentNumber = acknowledgmentNumber;
71          return this;
72      }
73
74      public PacketBuilder withConnection(Connection connection) {
75          this.withOrigin(connection.getConnectionSource());
76          this.withDestination(connection.getConnectedNode());
77          return this;
78      }
79
80      public PacketBuilder withIndex(int index) {
81          this.index = index;
82          return this;
83      }
84
85
86      public boolean hasFlag(Flag... flags) {
87          var hasFlag = true;
88          for (Flag flag : flags) {
89              hasFlag &= this.flags.contains(flag);
90          }
91          return hasFlag;
92      }
93
94  }
```

```java
package org.example.data;

public interface Payload {

    /**
     * A method that returns the size of the packet.
     * The size of an object is abstract and must be implemented according to what
        type
     * Payload is created.
     *
     * @return the size of the packet
     */
    int size();

    /**
     * A method that returns a String representation of the Payload
     *
     * @return String representation.
     */
    String toString();

    /**
     * A method to check if two object are equal or not
     *
     * @param obj to be checked against
     * @return true if this is considered equal to given object
     */
    boolean equals(Object obj);

    /**
     * A method that creates a hash number of this object
     *
     * @return Integer hash of the object
     */
    int hashCode();

}
```

```java
package org.example.network;

import org.example.data.Packet;
import org.example.network.interfaces.NetworkNode;
import org.example.protocol.MPTCP;
import org.example.util.Util;

import java.util.Queue;
import java.util.concurrent.ArrayBlockingQueue;

public class Channel implements Comparable<Channel> {

    private static final double BAD_TO_GOOD_PROB = 0.2;
    private static final int CAPACITY = 1000;
    private final Queue<Packet> line;
    private final NetworkNode source;
    private final NetworkNode destination;
    private final int cost;
    private final double loss;
    private boolean goodState;


    public Channel(NetworkNode source, NetworkNode destination, double loss, int
        cost) {
        this.line = new ArrayBlockingQueue<>(CAPACITY);
        this.source = source;
        this.destination = destination;
        this.loss = loss;
        this.cost = cost;
        this.goodState = true;
    }

    public Channel(NetworkNode source, NetworkNode destination, double loss) {
        this(source, destination, loss, Util.getNextRandomInt(100));
    }

    //loopback
    public Channel(NetworkNode source) {
        this(source, source, 0, 0);
    }

    public long propagationDelay() {
        return 10 * (this.cost + 1L);
    }

    private boolean lossy() {
        if (this.source.equals(this.destination)) return false;
        if (this.goodState) {
            this.goodState = Util.getNextRandomDouble() >= this.loss;
            return Util.getNextRandomDouble() < this.loss;
        }
        this.goodState = Util.getNextRandomDouble() >= BAD_TO_GOOD_PROB;
        return Util.getNextRandomDouble() < BAD_TO_GOOD_PROB;
    }

    public void channelPackage(Packet packet) {
        if (!this.line.offer(packet)) {
            throw new IllegalStateException("packet dropped in channel");
        }
    }

    public NetworkNode getSource() {
```

```
62          return source;
63      }
64
65      public NetworkNode getDestination() {
66          return destination;
67      }
68
69      public int getCost() {
70          return cost;
71      }
72
73      public boolean channel() {
74          if (this.line.isEmpty()) {
75              return false;
76          }
77
78          var packet = this.line.poll();
79          if (lossy()) {
80              return false;
81          }
82          return this.destination.enqueueInputBuffer(packet);
83
84      }
85
86      @Override
87      public String toString() {
88          return this.source.toString() + " -> [" + this.cost + "] -> " + this.
                destination.toString();
89      }
90
91      @Override
92      public int compareTo(Channel channel) {
93          if (this.cost > channel.getCost()) return 1;
94          if (this.cost < channel.getCost()) return -1;
95          return 0;
96      }
97
98      @Override
99      public int hashCode() {
100         return super.hashCode();
101     }
102
103     @Override
104     public boolean equals(Object obj) {
105         return super.equals(obj);
106     }
107
108     public static class ChannelBuilder {
109
110         double loss = 0;
111         int cost = Util.getNextRandomInt(100);
112
113         public ChannelBuilder withCost(int cost) {
114             if (cost < 0 || cost > 100) throw new IllegalStateException("cost is not
                    valid");
115             this.cost = cost;
116             return this;
117         }
118
119         public ChannelBuilder withLoss(double loss) {
120             this.loss = loss;
121             return this;
122         }
```

```
123
124         public void build(NetworkNode node1, NetworkNode node2) {
125             if (node1 instanceof MPTCP) node1 = ((MPTCP) node1).
                    getEndpointToAddChannelTo();
126             if (node2 instanceof MPTCP) node2 = ((MPTCP) node2).
                    getEndpointToAddChannelTo();
127             node1.addChannel(node2, this.loss, this.cost);
128             node2.addChannel(node1, this.loss, this.cost);
129         }
130
131     }
132
133 }
```

```java
package org.example.network;

import org.example.data.Packet;
import org.example.network.address.Address;
import org.example.network.interfaces.NetworkNode;
import org.example.simulator.statistics.Stats;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;

public abstract class Routable implements NetworkNode {

    protected final BlockingQueue<Packet> inputBuffer;
    private final RoutingTable routingTable;
    private final List<Channel> channels;
    private final Address address;
    private List<Channel> channelsUsed;

    protected Routable(BlockingQueue<Packet> inputBuffer, Address address) {
        this.inputBuffer = inputBuffer;
        this.channels = new ArrayList<>();
        this.address = address;
        this.routingTable = new RoutingTable();

        this.channelsUsed = new ArrayList<>(1);
    }

    public abstract Stats getStats();

    @Override
    public void updateRoutingTable() {
        this.routingTable.update(this);
    }

    @Override
    public void route(Packet packet) {
        if (packet == null) throw new IllegalStateException("Null packet can't be
            routed");
        NetworkNode destination = packet.getDestination();
        var nextChannelOnPath = this.routingTable.getPath(this, destination);
        nextChannelOnPath.channelPackage(packet);

        this.channelsUsed.add(nextChannelOnPath);
    }

    @Override
    public List<Channel> getChannelsUsed() {
        List<Channel> used = this.channelsUsed;
        this.channelsUsed = new ArrayList<>(1);
        return used;
    }

    @Override
    public long delay() {
        return ((long) this.inputBufferSize()) * 10;
    }

    @Override
    public List<Channel> getChannels() {
        return this.channels;
    }
```

```
62
63      @Override
64      public void addChannel(NetworkNode node, double noiseTolerance, int cost) {
65          var channel = new Channel(this, node, noiseTolerance, cost);
66          this.channels.add(channel);
67      }
68
69      @Override
70      public Address getAddress() {
71          return this.address;
72      }
73
74      @Override
75      public boolean enqueueInputBuffer(Packet packet) {
76          return this.inputBuffer.offer(packet);
77      }
78
79      @Override
80      public Packet peekInputBuffer() {
81          return this.inputBuffer.peek();
82      }
83
84      @Override
85      public Packet dequeueInputBuffer() {
86          return this.inputBuffer.poll();
87      }
88
89      @Override
90      public boolean inputBufferIsEmpty() {
91          return this.inputBuffer.isEmpty();
92      }
93
94      @Override
95      public int inputBufferSize() {
96          return this.inputBuffer.size();
97      }
98
99      @Override
100     public boolean equals(Object obj) {
101         if (obj instanceof NetworkNode) {
102             NetworkNode node = (NetworkNode) obj;
103             return this.getAddress().equals(node.getAddress());
104         }
105         return false;
106     }
107
108     @Override
109     public int hashCode() {
110         return this.getAddress().hashCode();
111     }
112
113     @Override
114     public String toString() {
115         return this.address.toString();
116     }
117
118
119 }
```

```java
package org.example.network;

import org.example.data.Packet;
import org.example.network.address.UUIDAddress;
import org.example.network.interfaces.Endpoint;
import org.example.simulator.statistics.Stats;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class RoutableEndpoint extends Routable implements Endpoint {

    private final BlockingQueue<Packet> receivedPackets;
    protected BlockingQueue<Packet> outputBuffer;

    public RoutableEndpoint(BlockingQueue<Packet> inputBuffer, BlockingQueue<Packet>
         outputBuffer) {
        super(inputBuffer, new UUIDAddress());
        this.outputBuffer = outputBuffer;
        this.receivedPackets = new ArrayBlockingQueue<>(10000);
    }

    @Override
    public Stats getStats() {
        return null;
    }

    public Packet dequeueOutputBuffer() {
        return this.outputBuffer.poll();
    }

    public boolean enqueueOutputBuffer(Packet packet) {
        return this.outputBuffer.offer(packet);
    }

    @Override
    public boolean outputBufferIsEmpty() {
        return this.outputBuffer.isEmpty();
    }

    public int outputBufferSize() {
        return this.outputBuffer.size();
    }

    @Override
    public boolean isConnected() {
        return false;
    }

    public Packet getReceivedPacket() {
        return this.receivedPackets.poll();
    }

    @Override
    public void run() {
        if (this.inputBufferIsEmpty()) return;
        Packet received = this.dequeueInputBuffer();
        this.receivedPackets.add(received);
    }


    @Override
```

```
62      public String toString() {
63          return "Endpoint: " + super.toString();
64      }
65
66      @Override
67      public boolean equals(Object o) {
68          return super.equals(o);
69      }
70
71      @Override
72      public int hashCode() {
73          return super.hashCode();
74      }
75
76  }
```

```java
package org.example.network;

import org.apache.commons.math3.distribution.PoissonDistribution;
import org.example.network.address.Address;
import org.example.network.address.UUIDAddress;
import org.example.simulator.statistics.Stats;
import org.example.util.Util;

import java.util.concurrent.ArrayBlockingQueue;

public class Router extends Routable {

    private final double queueSizeMean;
    private final int bufferSize;
    private final PoissonDistribution poissonDistribution;
    private int artificialQueueSize;

    private Router(int bufferSize, Address address, double averageQueueUtilization)
        {
        super(new ArrayBlockingQueue<>(bufferSize), address);
        this.bufferSize = bufferSize;
        this.queueSizeMean = averageQueueUtilization * this.bufferSize;
        this.poissonDistribution = Util.getPoissonDistribution(this.queueSizeMean);
        this.setArtificialQueueSize();
    }

    @Override
    public Stats getStats() {
        return null;
    }

    private void setArtificialQueueSize() {
        int queueSize = this.poissonDistribution.sample();
        this.artificialQueueSize = queueSize;
    }

    @Override
    public long delay() {
        this.setArtificialQueueSize();
        long transmissionDelay = 10;
        return transmissionDelay + this.artificialQueueSize * transmissionDelay;
    }

    @Override
    public void run() {
        if (!this.inputBufferIsEmpty()) {
            var packet = this.dequeueInputBuffer();
            this.route(packet);
        }
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    public static class RouterBuilder {
```

```
62
63          private int bufferSize = 100;
64          private double averageQueueUtilization = 0.85;
65          private Address address = new UUIDAddress();
66
67          public RouterBuilder withAverageQueueUtilization(double
                averageQueueUtilization) {
68              this.averageQueueUtilization = averageQueueUtilization;
69              return this;
70          }
71
72          public RouterBuilder withAddress(Address address) {
73              this.address = address;
74              return this;
75          }
76
77          public Router build() {
78              return new Router(this.bufferSize, this.address, this.
                    averageQueueUtilization);
79          }
80
81      }
82
83
84  }
```

```java
package org.example.network;

import org.example.network.interfaces.NetworkNode;

import java.util.*;

public class RoutingTable {

    private Map<NetworkNode, Map.Entry<Channel, Integer>> table;


    public RoutingTable() {
        this.table = new HashMap<>();
    }


    private void updateTable(NetworkNode node, Channel channel) {
        if (this.table.containsKey(node)) {
            this.updateEntry(node, channel);
            return;
        }
        this.addEntry(node, channel);
    }

    private void addEntry(NetworkNode node, Channel channel) {
        if (channel == null) {
            this.table.put(node, Map.entry(new Channel(node), 0));
            return;
        }

        boolean prevNodeInTable = table.containsKey(channel.getSource());
        int newCost = prevNodeInTable ? table.get(channel.getSource()).getValue() +
            channel.getCost() : channel.getCost();

        table.put(node, Map.entry(channel, newCost));
    }

    private void updateEntry(NetworkNode node, Channel channel) {
        boolean prevNodeInTable = table.containsKey(channel.getSource());
        int newCost = prevNodeInTable ? table.get(channel.getSource()).getValue() +
            channel.getCost() : 0;
        if (newCost < this.table.get(node).getValue()) {
            table.replace(node, Map.entry(channel, newCost));
        }
    }


    public void update(NetworkNode startingNode) {
        this.addEntry(startingNode, null);
        update(startingNode, new ArrayList<>(), new PriorityQueue<>());
    }

    private void update(NetworkNode curNode, List<NetworkNode> visited, Queue<
        Channel> priorityQueue) {
        visited.add(curNode);

        for (Channel channel : curNode.getChannels()) {
            priorityQueue.offer(channel);
            this.updateTable(channel.getDestination(), channel);
        }

        while (!priorityQueue.isEmpty()) {
```

```
60              var bestChannel = priorityQueue.poll();
61              if (visited.contains(bestChannel.getDestination())) continue;
62              update(bestChannel.getDestination(), visited, priorityQueue);
63          }
64      }
65
66
67      public Channel getPath(NetworkNode source, NetworkNode destination) {
68          if (this.table.isEmpty()) throw new IllegalStateException("The routing table
                is empty");
69          if (destination == null) throw new IllegalArgumentException("The destination
                can't be null");
70          if (!this.table.containsKey(destination))
71              throw new IllegalArgumentException("Destination " + destination + " does
                    not exist in this routing table");
72
73          var curChannel = this.table.get(destination).getKey();
74          NetworkNode prevNode = curChannel.getSource();
75          if (source.equals(prevNode)) return curChannel;
76          return getPath(source, prevNode);
77      }
78
79      @Override
80      public String toString() {
81          var builder = new StringBuilder();
82          for (Map.Entry<NetworkNode, Map.Entry<Channel, Integer>> entry : this.table.
                entrySet()) {
83              var networkNode = entry.getKey();
84              Map.Entry<Channel, Integer> value = entry.getValue();
85              var channel = value.getKey();
86              int cost = value.getValue();
87
88              builder.append("Node: ");
89              builder.append(networkNode);
90              builder.append(" | Channel: ");
91              builder.append(channel);
92              builder.append(" | Cost: ");
93              builder.append(cost);
94              builder.append("\n");
95          }
96          return builder.toString();
97      }
98
99
100 }
```

```java
package org.example.network.address;

import java.util.Objects;

public abstract class Address {

    public abstract String getId();

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        var address = (Address) o;
        return Objects.equals(this.getId(), address.getId());
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.getId());
    }

    @Override
    public String toString() {
        return this.getId();
    }

}
```

```java
package org.example.network.address;

public class SimpleAddress extends Address {

    private final String id;

    public SimpleAddress(String id) {
        this.id = id;
    }

    @Override
    public String getId() {
        return this.id;
    }

    @Override
    public boolean equals(Object o) {
        return super.equals(o);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

```java
package org.example.network.address;

import java.util.UUID;

public class UUIDAddress extends Address {

    private final UUID id;

    public UUIDAddress() {
        this.id = UUID.randomUUID();
    }

    @Override
    public String getId() {
        return this.id.toString();
    }

    @Override
    public boolean equals(Object o) {
        return super.equals(o);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

}
```

```java
package org.example.network.interfaces;


public interface Endpoint extends NetworkNode {
    /**
     * A method that checks if the outputBuffer has any Packets
     *
     * @return True if the outputBuffer is empty
     */
    boolean outputBufferIsEmpty();


    /**
     * A method that checks is the endpoint is connected
     *
     * @return True if connected
     */
    boolean isConnected();


}
```

```java
package org.example.network.interfaces;

import org.example.data.Packet;
import org.example.network.Channel;
import org.example.network.address.Address;

import java.util.List;

public interface NetworkNode {


    /**
     * A method that updates the routing table according to the cost
     */
    void updateRoutingTable();

    /**
     * A method that routes the packet to the next router on the path to it's
        destination
     *
     * @param packet to route
     */
    void route(Packet packet);


    /**
     * A method that returns the NetworkNode's associated delay
     *
     * @return delay
     */
    long delay();


    /**
     * A method that returns a List of the NetworkNode's outgoing Channels
     *
     * @return List of outgoing Channels from this NetworkNode
     */
    List<Channel> getChannels();


    /**
     * * A method that creates and adds a Channel to the list of channels.
     *
     * @param node           NetworkNode to create channel to
     * @param noiseTolerance
     * @param cost           the cost of the channel used in routing
     */
    void addChannel(NetworkNode node, double noiseTolerance, int cost);


    /**
     * A method that returns the unique Address associated with this NetworkNode
     *
     * @return unique Address
     */
    Address getAddress();


    /**
     * A method that returns the first Packet form the inputBuffer without removing
        it from the buffer
```

```
61          *
62          * @return the dequeued Packet
63          */
64         Packet peekInputBuffer();
65
66
67         /**
68          * A method that returns and dequeues a Packet form the inputBuffer
69          *
70          * @return the dequeued Packet
71          */
72         Packet dequeueInputBuffer();
73
74         /**
75          * A method that enqueues the given Packet to the inputBuffer
76          *
77          * @param packet to be enqueued
78          * @return True if successful
79          */
80         boolean enqueueInputBuffer(Packet packet);
81
82         /**
83          * A method that checks if the inputBuffer has any Packets
84          *
85          * @return True if the inputBuffer is empty
86          */
87         boolean inputBufferIsEmpty();
88
89         /**
90          * A method that returns the size of the inputBuffer
91          *
92          * @return size of the inputBuffer
93          */
94         int inputBufferSize();
95
96         /**
97          * A method that returns a list of all channels used during an event
98          *
99          * @return A List with Channel objects
100         */
101        List<Channel> getChannelsUsed();
102
103        /**
104         * A method to run the NetworkNode
105         */
106        void run();
107
108
109 }
```

```java
package org.example.protocol;

import org.example.data.*;
import org.example.network.Channel;
import org.example.network.Routable;
import org.example.network.address.Address;
import org.example.network.address.UUIDAddress;
import org.example.network.interfaces.Endpoint;
import org.example.protocol.window.receiving.ReceivingWindow;
import org.example.protocol.window.receiving.SelectiveRepeat;
import org.example.protocol.window.sending.SendingWindow;
import org.example.protocol.window.sending.SlidingWindow;
import org.example.simulator.statistics.TCPStats;
import org.example.util.Util;
import org.javatuples.Pair;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClassicTCP extends Routable implements TCP {

    private static final Comparator<Packet> SENDING_WINDOW_PACKET_COMPARATOR =
        Comparator.comparingInt(Packet::getSequenceNumber);
    private final TCPStats tcpStats;
    private final Logger logger = Logger.getLogger(ClassicTCP.class.getSimpleName())
        ;
    private final List<Packet> receivedPackets;
    private final List<Pair<Integer, Payload>> payloadsToSend;
    private final int thisReceivingWindowCapacity;
    private final boolean isReno;
    private final TCP mainFlow;
    private int otherReceivingWindowCapacity;
    private int initialSequenceNumber;
    private long rtt;
    private long rttStartTimer = 0;
    private boolean timerStarted = false;
    private boolean rttSet = false;
    private SendingWindow sendingWindow;

    private ClassicTCP(int thisReceivingWindowCapacity,
                       List<Packet> receivedPackets,
                       List<Pair<Integer, Payload>> payloadsToSend,
                       boolean isReno,
                       Address address,
                       TCP mainFlow,
                       ReceivingWindow receivingWindow) {
        super(receivingWindow, address);
        this.receivedPackets = receivedPackets;
        this.payloadsToSend = payloadsToSend;
        this.thisReceivingWindowCapacity = thisReceivingWindowCapacity;
        this.isReno = isReno;

        if (mainFlow == null) {
            this.mainFlow = this;
        } else {
            this.mainFlow = mainFlow;
        }

        this.tcpStats = new TCPStats(address);
```

```
61      }
62
63
64      @Override
65      public void connect(TCP host) {
66          if (this.isConnected()) return;
67          this.initialSequenceNumber = Util.getNextRandomInt(10000);
68          Packet syn = new PacketBuilder()
69                  .withDestination(host)
70                  .withOrigin(this)
71                  .withFlags(Flag.SYN)
72                  .withSequenceNumber(this.initialSequenceNumber)
73                  .withPayload(new Message(this.thisReceivingWindowCapacity + ""))
74                  .build();
75          this.route(syn);
76          if (!this.timerStarted) {
77              this.rttStartTimer = Util.seeTime();
78              this.timerStarted = true;
79          }
80      }
81
82      public void continueConnect(Packet synAck) {
83          Endpoint host = synAck.getOrigin();
84          if (synAck.hasAllFlags(Flag.SYN, Flag.ACK)) {
85              if (synAck.getAcknowledgmentNumber() != this.initialSequenceNumber + 1)
                    {
86                  return;
87              }
88              this.otherReceivingWindowCapacity = Integer.parseInt(synAck.getPayload()
                    .toString());
89              int finalSeqNum = synAck.getAcknowledgmentNumber();
90              int ackNum = synAck.getSequenceNumber() + 1;
91              Packet ack = new PacketBuilder()
92                      .withDestination(host)
93                      .withOrigin(this)
94                      .withFlags(Flag.ACK)
95                      .withSequenceNumber(finalSeqNum)
96                      .withAcknowledgmentNumber(ackNum)
97                      .build();
98              this.route(ack);
99
100             this.setRTT();
101             this.setConnection(new Connection(this, host, finalSeqNum, ackNum));
102
103             this.logger.log(Level.INFO, () -> "connection established with host: " +
                    this.getConnection());
104         }
105
106     }
107
108     @Override
109     public void connect(Packet syn) {
110         if (this.isConnected()) return;
111         Endpoint node = syn.getOrigin();
112         var seqNum = Util.getNextRandomInt(10000);
113         var ackNum = syn.getSequenceNumber() + 1;
114         this.otherReceivingWindowCapacity = Integer.parseInt(syn.getPayload().
                toString());
115         Packet synAck = new PacketBuilder()
116                 .withDestination(node)
117                 .withOrigin(this)
118                 .withFlags(Flag.SYN, Flag.ACK)
119                 .withSequenceNumber(seqNum)
```

```
                .withAcknowledgmentNumber(ackNum)
                .withPayload(new Message(this.thisReceivingWindowCapacity + ""))
                .build();
        this.route(synAck);

        if (!this.timerStarted) {
            this.rttStartTimer = Util.seeTime();
            this.timerStarted = true;
        }

        this.setConnection(new Connection(this, node, seqNum, ackNum));

        this.logger.log(Level.INFO, () -> "connection established with client: " +
            this.getConnection());

    }

    private void setRTT() {
        if (rttSet) return;
        this.rtt = Util.seeTime() - this.rttStartTimer;
        this.rttSet = true;
    }

    @Override
    public void send(Payload payload) {
        if (this.payloadsToSend.isEmpty()) {
            this.payloadsToSend.add(Pair.with(0, payload));
            return;
        }
        int indexOfLastAdded = this.payloadsToSend.get(this.payloadsToSend.size() -
            1).getValue0();
        this.payloadsToSend.add(Pair.with(indexOfLastAdded + 1, payload));

    }

    @Override
    public Packet receive() {
        if (this.receivedPackets.isEmpty()) return null;
        return this.receivedPackets.remove(0);
    }

    @Override
    public Channel getChannel() {
        return this.getChannels().get(0);
    }

    public int getThisReceivingWindowCapacity() {
        return this.thisReceivingWindowCapacity;
    }

    public int getOtherReceivingWindowCapacity() {
        return this.otherReceivingWindowCapacity;
    }

    @Override
    public boolean isConnected() {
        try {
            return this.getSendingWindow().getConnection() != null;
        } catch (IllegalAccessException e) {
            //no SendingWindow means no connection
            return false;
        }
    }
```

```
181
182     public Connection getConnection() {
183         try {
184             return this.getSendingWindow().getConnection();
185         } catch (IllegalAccessException e) {
186             //No SendingWindow means no connection is established
187             return null;
188         }
189     }
190
191     private void setConnection(Connection connection) {
192         this.sendingWindow = new SlidingWindow(this.otherReceivingWindowCapacity,
                this.isReno, connection, SENDING_WINDOW_PACKET_COMPARATOR, this.
                payloadsToSend, this.tcpStats);
193     }
194
195     @Override
196     public long getRTO() {
197         return 3 * this.rtt;
198     }
199
200
201     @Override
202     public Packet fastRetransmit() {
203         try {
204             return this.getSendingWindow().fastRetransmit();
205         } catch (IllegalAccessException e) {
206             //no Sending Window results in no retransmit
207             return null;
208         }
209     }
210
211     @Override
212     public long delay() {
213         return 10;
214     }
215
216     @Override
217     public TCP getMainFlow() {
218         return this.mainFlow;
219     }
220
221     @Override
222     public int getNumberOfFlows() {
223         return 1;
224     }
225
226     public SendingWindow getSendingWindow() throws IllegalAccessException {
227         if (this.sendingWindow != null) return this.sendingWindow;
228         throw new IllegalAccessException("SendingWindow is null");
229     }
230
231     public ReceivingWindow getReceivingWindow() throws IllegalAccessException {
232         if (this.inputBuffer instanceof ReceivingWindow) return (ReceivingWindow)
                this.inputBuffer;
233         throw new IllegalAccessException("The outputBuffer is not of type
                ReceivingWindow");
234     }
235
236     private boolean packetIsFromValidConnection(Packet packet) {
237         if (packet.hasAllFlags(Flag.SYN) && !this.isConnected()) return true;
238         if (!this.isConnected()) return false;
239
```

```
240         var conn = this.getConnection();
241         if (packet.hasAllFlags(Flag.ACK)) return true;
242 
243 
244         try {
245             boolean inWindow = this.getReceivingWindow().inReceivingWindow(packet,
                    conn);
246             if (!inWindow) {
247                 this.ack(this.getReceivingWindow().ackThis(this.getSendingWindow().
                        getConnection().getConnectedNode()));
248                 return false;
249             }
250 
251             return packet.getOrigin().equals(conn.getConnectedNode())
252                     && packet.getDestination().equals(conn.getConnectionSource());
253         } catch (IllegalAccessException e) {
254             return false;
255         }
256     }
257 
258     @Override
259     public boolean outputBufferIsEmpty() {
260         return this.sendingWindow.isEmpty();
261     }
262 
263     @Override
264     public boolean enqueueInputBuffer(Packet packet) {
265         if (packetIsFromValidConnection(packet)) {
266             if (super.enqueueInputBuffer(packet)) {
267                 if (!packet.hasAllFlags(Flag.ACK) && !packet.hasAllFlags(Flag.SYN))
                        this.tcpStats.packetArrival(packet);
268                 return true;
269             }
270             return true;
271         }
272         // return true because the packet has arrived the endpoint
273         // the packet is not added to the input buffer, but it is checked
274         return false;
275     }
276 
277     private boolean unconnectedInputHandler() {
278         if (this.inputBuffer.isEmpty()) return false;
279         if (!this.peekInputBuffer().getDestination().equals(this)) return false;
280 
281         var packet = this.dequeueInputBuffer();
282 
283         if (packet.hasAllFlags(Flag.SYN, Flag.ACK)) {
284             this.continueConnect(packet);
285             return true;
286         }
287 
288         if (packet.hasAllFlags(Flag.SYN)) {
289             this.connect(packet);
290             return true;
291         }
292         return false;
293     }
294 
295 
296     public boolean canRetransmit(Packet packet) {
297         try {
298             return this.getSendingWindow().canRetransmit(packet);
299         } catch (IllegalAccessException e) {
```

```
300             //should not retransmit without SendingWindow
301             return false;
302         }
303     }
304
305
306     private void ack(Packet packet) {
307         assert packet != null : "Packet is null";
308
309         if (packet.getSequenceNumber() == -1) return;
310
311         if (packet.getOrigin() == null) {
312             //can't call ack on packet with no origin
313             Endpoint connectedNode;
314             try {
315                 connectedNode = this.getSendingWindow().getConnection().
                        getConnectedNode();
316             } catch (IllegalAccessException e) {
317                 //should not be able to ack without a SendingWindow
318                 return;
319             }
320             packet = new PacketBuilder()
321                     .withDestination(connectedNode)
322                     .withOrigin(this)
323                     .withSequenceNumber(packet.getSequenceNumber())
324                     .withAcknowledgmentNumber(packet.getAcknowledgmentNumber())
325                     .withPayload(packet.getPayload())
326                     .build();
327
328         }
329         Packet ack = new PacketBuilder().ackBuild(packet);
330         this.route(ack);
331     }
332
333
334     public boolean handleIncoming() {
335         if (!isConnected()) return unconnectedInputHandler();
336
337         try {
338             var receivingWindow = this.getReceivingWindow();
339             var packetReceived = receivingWindow.receive(this.getSendingWindow());
340             if (packetReceived && receivingWindow.shouldAck()) {
341                 try {
342                     this.ack(receivingWindow.ackThis(this.getSendingWindow().
                            getConnection().getConnectedNode()));
343                 } catch (IllegalArgumentException e) {
344                     return false;
345                 }
346             }
347
348             this.setRTT();
349
350             var packetToFastRetransmit = this.fastRetransmit();
351             if (packetToFastRetransmit != null) {
352                 this.route(packetToFastRetransmit);
353                 this.tcpStats.packetFastRetransmit();
354             }
355             return true;
356         } catch (IllegalAccessException e) {
357             throw new IllegalStateException("TCP is connected but no ReceivingWindow
                    or SendingWindow is established");
358         }
359     }
```

```
360
361
362     @Override
363     public List<Packet> trySend() {
364         return this.trySend(new ArrayList<>());
365     }
366
367     private List<Packet> trySend(List<Packet> packetsSent) {
368         if (this.sendingWindow == null) return packetsSent;
369         if (this.sendingWindow.isQueueEmpty()) return packetsSent;
370         if (this.sendingWindow.isWaitingForAck()) return packetsSent;
371
372         var packetToSend = this.sendingWindow.send();
373         assert packetToSend != null;
374
375         this.route(packetToSend);
376         this.tcpStats.packetSend();
377
378         packetsSent.add(packetToSend);
379         return trySend(packetsSent);
380     }
381
382     @Override
383     public TCPStats getStats() {
384         return tcpStats;
385     }
386
387     @Override
388     public void run() {
389         this.handleIncoming();
390         this.trySend();
391     }
392
393     @Override
394     public boolean equals(Object o) {
395         return super.equals(o);
396     }
397
398     @Override
399     public int hashCode() {
400         return super.hashCode();
401     }
402
403
404     public static class ClassicTCPBuilder {
405
406         private int receivingWindowCapacity = 7;
407         private List<Packet> receivedPacketsContainer = new ArrayList<>();
408         private List<Pair<Integer, Payload>> payloadsToSend = new ArrayList<>();
409         private boolean isReno = true;
410         private Address address = new UUIDAddress();
411         private TCP mainflow = null;
412         private ReceivingWindow receivingWindow = new SelectiveRepeat(this.
                receivingWindowCapacity, Comparator.comparingInt(Packet::
                getSequenceNumber), this.receivedPacketsContainer);
413
414         public ClassicTCPBuilder withReceivingWindowCapacity(int
                receivingWindowCapacity) {
415             this.receivingWindowCapacity = receivingWindowCapacity;
416             return this;
417         }
418
419         public ClassicTCPBuilder withReceivedPacketsContainer(List<Packet>
```

```
            receivedPacketsContainer) {
420             this.receivedPacketsContainer = receivedPacketsContainer;
421             return this;
422         }
423 
424         public ClassicTCPBuilder withPayloadsToSend(List<Pair<Integer, Payload>>
                payloadsToSend) {
425             this.payloadsToSend = payloadsToSend;
426             return this;
427         }
428 
429         public ClassicTCPBuilder setTahoe() {
430             this.isReno = false;
431             return this;
432         }
433 
434         public ClassicTCPBuilder setReno() {
435             this.isReno = true;
436             return this;
437         }
438 
439         public ClassicTCPBuilder withAddress(Address address) {
440             this.address = address;
441             return this;
442         }
443 
444         public ClassicTCPBuilder withMainFlow(TCP tcp) {
445             this.mainflow = tcp;
446             return this;
447         }
448 
449         public ClassicTCPBuilder withReceivingWindow(ReceivingWindow receivingWindow
                ) {
450             this.receivingWindow = receivingWindow;
451             return this;
452         }
453 
454         public ClassicTCP build() {
455             return new ClassicTCP(this.receivingWindowCapacity,
456                     this.receivedPacketsContainer,
457                     this.payloadsToSend,
458                     this.isReno,
459                     this.address,
460                     this.mainflow,
461                     this.receivingWindow);
462         }
463     }
464 
465 }
```

```java
package org.example.protocol;

import org.example.data.Flag;
import org.example.data.Packet;
import org.example.network.interfaces.Endpoint;

import java.util.Objects;

public class Connection {

    private Endpoint self;
    private Endpoint other;
    private int sequenceNumber;
    private int acknowledgementNumber;


    public Connection(Endpoint self, Endpoint other, int sequenceNumber, int
        acknowledgementNumber) {
        this.self = self;
        this.other = other;
        this.sequenceNumber = sequenceNumber;
        this.acknowledgementNumber = acknowledgementNumber;
    }

    public void update(Packet packet) {
        if (packet.hasAllFlags(Flag.ACK)) {
            this.sequenceNumber = packet.getAcknowledgmentNumber();
            return;
        }
        this.acknowledgementNumber = packet.getSequenceNumber() + 1;
    }


    public int getNextSequenceNumber() {
        return sequenceNumber;
    }

    public int getNextAcknowledgementNumber() {
        return acknowledgementNumber;
    }

    public Endpoint getConnectedNode() {
        return other;
    }

    public Endpoint getConnectionSource() {
        return self;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Connection that = (Connection) o;
        return Objects.equals(self, that.self) &&
                Objects.equals(other, that.other);
    }

    @Override
    public int hashCode() {
        return Objects.hash(self, other);
    }
```

```
62
63      @Override
64      public String toString() {
65          return "Connection[" + other + "]";
66      }
67  }
```

```java
package org.example.protocol;

import org.example.data.Packet;
import org.example.data.Payload;
import org.example.network.Channel;
import org.example.network.address.Address;
import org.example.network.address.SimpleAddress;
import org.example.network.address.UUIDAddress;
import org.example.network.interfaces.Endpoint;
import org.example.network.interfaces.NetworkNode;
import org.example.protocol.window.receiving.ReceivingWindow;
import org.example.protocol.window.receiving.SelectiveRepeat;
import org.example.simulator.statistics.TCPStats;
import org.javatuples.Pair;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MPTCP implements TCP {

    private static final String DEPRECATED_STRING = "DEPRECATED";
    private static final Comparator<Packet> PACKET_INDEX_COMPARATOR = Comparator.
        comparingInt(Packet::getIndex);
    private final List<Packet> receivedPackets;
    private final List<Pair<Integer, Payload>> payloadsToSend;
    private final TCP[] subflows;
    private final Address address;
    private final ReceivingWindow receivingWindow;

    private MPTCP(int numberOfSubflows, int receivingWindowCapacity, Address address
        ) {
        this.receivedPackets = new ArrayList<>();
        this.payloadsToSend = new ArrayList<>();
        this.subflows = new TCP[numberOfSubflows];
        this.address = address;
        this.receivingWindow = new SelectiveRepeat(receivingWindowCapacity,
            PACKET_INDEX_COMPARATOR, this.receivedPackets);

        for (var i = 0; i < numberOfSubflows; i++) {
            TCP tcp = new ClassicTCP.ClassicTCPBuilder()
                    .withReceivingWindowCapacity(receivingWindowCapacity /
                        numberOfSubflows)
                    .withAddress(new SimpleAddress("Subflow " + i + " " + this.
                        address))
                    .withReceivedPacketsContainer(this.receivedPackets)
                    .withPayloadsToSend(this.payloadsToSend)
                    .withReceivingWindow(this.receivingWindow)
                    .withMainFlow(this)
                    .setReno()
                    .build();
            this.subflows[i] = tcp;
        }
    }

    @Override
    public TCP getMainFlow() {
        return this;
    }

```

```
58      @Override
59      public int getNumberOfFlows() {
60          return this.subflows.length;
61      }
62
63      @Override
64      public boolean outputBufferIsEmpty() {
65          return this.payloadsToSend.isEmpty();
66      }
67
68      @Override
69      public void updateRoutingTable() {
70          for (TCP subflow : this.subflows) {
71              subflow.updateRoutingTable();
72          }
73      }
74
75      @Override
76      public void route(Packet packet) {
77          throw new IllegalStateException(DEPRECATED_STRING);
78      }
79
80      @Override
81      public long delay() {
82          return 10;
83      }
84
85      @Override
86      public List<Channel> getChannels() {
87          List<Channel> channels = new ArrayList<>();
88          for (TCP subflow : this.subflows) {
89              for (Channel c : subflow.getChannels()) {
90                  channels.add(c);
91              }
92          }
93          return channels;
94      }
95
96      public Endpoint getEndpointToAddChannelTo() {
97          for (Endpoint endpoint : this.subflows) {
98              if (endpoint.getChannels().isEmpty()) return endpoint;
99          }
100         throw new IllegalStateException("no endpoints available for channel adding")
                ;
101     }
102
103
104     @Override
105     public void addChannel(NetworkNode node, double noiseTolerance, int cost) {
106         for (TCP subflow : this.subflows) {
107             if (subflow.getChannels().isEmpty()) { // only allowing one channel per
                    subflow
108                 subflow.addChannel(node, noiseTolerance, cost);
109                 return;
110             }
111         }
112     }
113
114     @Override
115     public Address getAddress() {
116         throw new IllegalStateException(DEPRECATED_STRING);
117     }
118
```

```
119     @Override
120     public Packet peekInputBuffer() {
121         throw new IllegalStateException(DEPRECATED_STRING);
122     }
123
124     @Override
125     public Packet dequeueInputBuffer() {
126         throw new IllegalStateException(DEPRECATED_STRING);
127     }
128
129     @Override
130     public boolean enqueueInputBuffer(Packet packet) {
131         throw new IllegalStateException(DEPRECATED_STRING);
132     }
133
134     @Override
135     public boolean inputBufferIsEmpty() {
136         return this.receivingWindow.isEmpty();
137     }
138
139     @Override
140     public int inputBufferSize() {
141         return this.receivingWindow.size();
142     }
143
144
145     @Override
146     public void run() {
147         throw new IllegalStateException(DEPRECATED_STRING);
148     }
149
150     public TCP[] getSubflows() {
151         return this.subflows;
152     }
153
154     @Override
155     public void connect(TCP host) {
156         if (!(host instanceof MPTCP)) {
157             this.subflows[0].connect(host);
158             return;
159         }
160         var mptcpHost = (MPTCP) host;
161         TCP[] hostSubflows = mptcpHost.getSubflows();
162
163         for (var i = 0; i < this.subflows.length; i++) {
164             TCP cFlow = this.subflows[i];
165             if (cFlow.isConnected()) continue;
166             for (var j = i; j < hostSubflows.length; j++) {
167                 TCP hFlow = hostSubflows[i];
168                 if (hFlow.isConnected()) continue;
169                 try {
170                     cFlow.connect(hFlow);
171                     break;
172                 } catch (IllegalArgumentException e) {
173                     Logger.getLogger(this.getClass().getSimpleName()).log(Level.INFO
                            , "connection failed");
174                 }
175             }
176         }
177     }
178
179     @Override
180     public void connect(Packet syn) {
```

```
181         throw new IllegalStateException(DEPRECATED_STRING);
182     }
183 
184     @Override
185     public void send(Payload payload) {
186         // avoiding duplicate code
187         // adds to the payloadsToSend list that is shared among all subflows
188         // therefore, it does not matter which subflow does this
189         this.subflows[0].send(payload);
190     }
191 
192     @Override
193     public Packet receive() {
194         if (this.receivedPackets.isEmpty()) return null;
195         return this.receivedPackets.remove(0);
196     }
197 
198     @Override
199     public boolean isConnected() {
200         for (TCP subflow : this.subflows) {
201             if (!subflow.isConnected()) return false;
202         }
203         return true;
204 
205     }
206 
207     @Override
208     public Channel getChannel() {
209         throw new IllegalStateException(DEPRECATED_STRING);
210     }
211 
212     @Override
213     public List<Channel> getChannelsUsed() {
214         List<Channel> usedChannels = new ArrayList<>(this.subflows.length);
215         for (TCP subflow : this.subflows) {
216             for (Channel channel : subflow.getChannelsUsed()) {
217                 usedChannels.add(channel);
218             }
219         }
220         return usedChannels;
221     }
222 
223     @Override
224     public long getRTO() {
225         throw new IllegalStateException(DEPRECATED_STRING);
226     }
227 
228 
229     @Override
230     public boolean handleIncoming() {
231         for (var i = 0; i < this.subflows.length; i++) {
232             for (TCP subflow : this.subflows) {
233                 try {
234                     subflow.handleIncoming();
235                 } catch (IllegalArgumentException e) {
236                     //do nothing, continue loop
237                 }
238             }
239         }
240         return false;
241     }
242 
243     @Override
```

```
244     public List<Packet> trySend() {
245         List<Packet> packetsSent = new ArrayList<>();
246         for (TCP subflow : this.subflows) {
247             for (Packet packet : subflow.trySend()) {
248                 packetsSent.add(packet);
249             }
250         }
251         return packetsSent;
252     }
253 
254     @Override
255     public boolean canRetransmit(Packet packet) {
256         for (TCP subflow : this.subflows) {
257             if (subflow.canRetransmit(packet)) return true;
258         }
259         return false;
260     }
261 
262     public TCPStats[] getTcpStats() {
263         var tcpStats = new TCPStats[this.subflows.length];
264         for (var i = 0; i < this.subflows.length; i++) {
265             tcpStats[i] = ((ClassicTCP) this.subflows[i]).getStats();
266         }
267         return tcpStats;
268     }
269 
270     @Override
271     public Packet fastRetransmit() {
272         throw new IllegalStateException(DEPRECATED_STRING);
273     }
274 
275     public static class MPTCPBuilder {
276 
277         private int numberOfSubflows = 2;
278         private int receivingWindowCapacity = 20;
279         private Address address = new UUIDAddress();
280 
281         public MPTCPBuilder withNumberOfSubflows(int numberOfSubflows) {
282             this.numberOfSubflows = numberOfSubflows;
283             return this;
284         }
285 
286         public MPTCPBuilder withReceivingWindowCapacity(int capacity) {
287             this.receivingWindowCapacity = capacity;
288             return this;
289         }
290 
291         public MPTCPBuilder withAddress(Address address) {
292             this.address = address;
293             return this;
294         }
295 
296         public MPTCP build() {
297             return new MPTCP(this.numberOfSubflows, this.receivingWindowCapacity,
                    this.address);
298         }
299 
300     }
301 }
```

```java
package org.example.protocol;

import org.example.data.Packet;
import org.example.data.Payload;
import org.example.network.Channel;
import org.example.network.interfaces.Endpoint;

import java.util.List;

public interface TCP extends Endpoint {

    /**
     * A method that initiates connection with a host
     * <p>
     * 1. Send SYN with random number A
     * 2. Receive SYN-ACK with A+1 and sequence number random B (the server chooses B)
     * 3. Send ACK back to server. The ACK is now A+2 and sequence number is B+1
     *
     * @param host to connect to
     */
    void connect(TCP host);

    /**
     * A method that handles incoming connections
     * <p>
     * 1. Receive SYN Packet with random number A that indicates that a client wants to connect
     * 2. Send SYN-ACK with A+1 and sequence number random B
     * 3. Receive ACK with ACK-number A+2 and sequence number is B+1
     *
     * @param syn Packet to start incoming connection from a client
     */
    void connect(Packet syn);

    /**
     * Creates a packet with Payload and enqueues the new Packet to the output-buffer
     *
     * @param payload
     */
    void send(Payload payload);

    /**
     * Dequeues the Packet from the received queue
     * All packets in the received queue are acknowledged and in correct order
     *
     * @return next Packet
     */
    Packet receive();


    /**
     * A method that to determine if TCP has an open connection
     *
     * @return true if TCP has an active connection
     */
    boolean isConnected();

    /**
     * A method that returns the available channel to route packets through
     *
```

```
60          * @return an available Channel
61          */
62         Channel getChannel();
63
64         /**
65          * A method that returns the retransmission timeout value
66          *
67          * @return the RTO for this connection
68          */
69         long getRTO();
70
71         /**
72          * A method to handle the incoming packets
73          *
74          * @return returns true if a packet should be ACKed
75          */
76         boolean handleIncoming();
77
78         /**
79          * A method to handle the sending process of TCP
80          *
81          * @return a List of the sent packets
82          */
83         List<Packet> trySend();
84
85         /**
86          * A method that checks if a packet should be retransmitted or not
87          *
88          * @param packet that should be checked for possible retransmission
89          * @return true if the packet should be retransmitted
90          */
91         boolean canRetransmit(Packet packet);
92
93         /**
94          * A method that returns a packet that should be fast retransmitted
95          *
96          * @return Packet if there is a packet to fast-retransmit, else null
97          */
98         Packet fastRetransmit();
99
100        /**
101         * A method that returns the main flow or controller of this TCP
102         *
103         * @return the main TCP flow
104         */
105        TCP getMainFlow();
106
107        /**
108         * A method that retuns the number of subflows available
109         *
110         * @return the number of subflows
111         */
112        int getNumberOfFlows();
113
114
115    }
```

```java
package org.example.protocol.window;


public interface IWindow {

    /**
     * A method that uses the given packet-index to determine
     * if the packet is inside the window
     *
     * @param packetIndex
     * @return true if the packet is in the window
     */
    boolean inWindow(int packetIndex);

    /**
     * A method that returns the window capacity
     *
     * @return the window capacity
     */
    int getWindowCapacity();

}
```

```java
package org.example.protocol.window;

import org.example.data.Packet;
import org.example.util.BoundedPriorityBlockingQueue;

import java.util.Comparator;

public abstract class Window extends BoundedPriorityBlockingQueue<Packet> implements
     IWindow {

    protected Window(int windowCapacity, Comparator<Packet> comparator) {
        super(windowCapacity, comparator);
    }

    @Override
    public boolean inWindow(int packetIndex) {
        return packetIndex < this.getWindowCapacity() && packetIndex >= 0;
    }

    @Override
    public int getWindowCapacity() {
        return this.bound();
    }

    @Override
    public String toString() {
        var stringBuilder = new StringBuilder();
        for (Packet packet : this) {
            stringBuilder.append("[");
            stringBuilder.append(packet);
            stringBuilder.append("]");
            stringBuilder.append("\n");
        }
        return stringBuilder.toString();
    }
}
```

```java
package org.example.protocol.window.receiving;

import org.example.data.Packet;
import org.example.network.interfaces.Endpoint;
import org.example.protocol.Connection;
import org.example.protocol.window.IWindow;
import org.example.protocol.window.sending.SendingWindow;
import org.example.util.BoundedQueue;

public interface ReceivingWindow extends IWindow, BoundedQueue<Packet> {

    /**
     * A method that receives incoming packets and returns true
     * if a packet was added to the received packets queue
     *
     * @param sendingWindow
     * @return true if a non ACK packet was received
     */
    boolean receive(SendingWindow sendingWindow);

    /**
     * A method that returns the packet to ACK
     *
     * @param endpointToReceiveAck the endpoint to receive the ACK
     * @return the packet to ACK
     */
    Packet ackThis(Endpoint endpointToReceiveAck);

    /**
     * A method that checks if a ACK should be sent or not
     *
     * @return
     */
    boolean shouldAck();

    /**
     * A method that calculates the receiving window packet index
     * of the given packet
     *
     * @param packet     to calculate packet index of
     * @param connection to use in the calculation
     * @return the packet index of the given packet
     */
    int receivingPacketIndex(Packet packet, Connection connection);

    /**
     * A method that checks if a packet is inside the receiving window
     *
     * @param packet     packet to check
     * @param connection to use in calculation
     * @return true if the packet is inside the receiving window
     */
    boolean inReceivingWindow(Packet packet, Connection connection);

}
```

```java
package org.example.protocol.window.receiving;

import org.example.data.Flag;
import org.example.data.Packet;
import org.example.data.PacketBuilder;
import org.example.network.interfaces.Endpoint;
import org.example.protocol.Connection;
import org.example.protocol.window.Window;
import org.example.protocol.window.sending.SendingWindow;

import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SelectiveRepeat extends Window implements ReceivingWindow {

    private final List<Packet> received;
    private int packetCount;
    private Map<Endpoint, Packet> ackThisMap;

    public SelectiveRepeat(int windowSize, Comparator<Packet> comparator, List<Packet> receivedContainer) {
        super(windowSize, comparator);
        this.received = receivedContainer;
        this.packetCount = 0;
        this.ackThisMap = new HashMap<>();
    }

    private void receive(Packet packet) {
        if (this.received.contains(packet)) return;
        boolean added = this.received.add(packet);
        if (!added) throw new IllegalStateException("Packet was not added to the received queue");
    }

    @Override
    public boolean receive(SendingWindow sendingWindow) {
        if (this.isEmpty()) return false;
        if (this.peek().hasAllFlags(Flag.SYN)) return false;

        var connection = sendingWindow.getConnection();

        if (this.peek().hasAllFlags(Flag.ACK)) {
            if (this.peek().getOrigin().equals(connection.getConnectedNode())) {
                sendingWindow.ackReceived(this.peek());
                this.poll();
            }
            return false;
        }

        var packetToUpdateWith = this.ackThisMap.get(connection.getConnectedNode());
        if (packetToUpdateWith != null) connection.update(packetToUpdateWith);

        if (this.inReceivingWindow(this.peek(), connection)) {
            while (receivingPacketIndex(this.peek(), connection) == 0 && this.peek().getIndex() == this.packetCount) {
                connection.update(this.peek());
                this.ackThisMap.put(this.peek().getOrigin(), this.peek());
                this.receive(this.peek());
                var packetRemoved = this.remove();
                if (packetRemoved == null) throw new IllegalStateException("removing
```

```
                    null packet");
60                  sendingWindow.getStats().packetDeparture(packetRemoved);
61                  this.packetCount++;
62                  if (this.isEmpty()) return true;
63              }
64              return true; // true so that duplicate AKCs are sent
65          }
66          return false;
67      }
68
69      @Override
70      public boolean shouldAck() {
71          return true;
72      }
73
74      @Override
75      public Packet ackThis(Endpoint endpointToReceiveAck) {
76          return this.ackThisMap.getOrDefault(endpointToReceiveAck, new PacketBuilder
                ().build());
77      }
78
79      @Override
80      public int receivingPacketIndex(Packet packet, Connection connection) {
81          int seqNum = packet.getSequenceNumber();
82          int ackNum = connection.getNextAcknowledgementNumber();
83          return seqNum - ackNum;
84      }
85
86      @Override
87      public boolean inReceivingWindow(Packet packet, Connection connection) {
88          int packetIndex = receivingPacketIndex(packet, connection);
89          return inWindow(packetIndex);
90      }
91
92      @Override
93      public boolean offer(Packet packet) {
94          if (this.isFull()) {
95              return false;
96          }
97          if (this.contains(packet) && !packet.hasAllFlags(Flag.ACK)) {
98              return false;
99          }
100         return super.offer(packet);
101     }
102
103 }
```

```java
package org.example.protocol.window.sending;

import org.example.data.Packet;
import org.example.protocol.Connection;
import org.example.protocol.window.IWindow;
import org.example.simulator.statistics.TCPStats;
import org.example.util.BoundedQueue;

public interface SendingWindow extends IWindow, BoundedQueue<Packet> {

    /**
     * A method that checks if the sendigwindow should wait for ACK or not
     *
     * @return true if the sendingwindow should wait
     */
    boolean isWaitingForAck();

    /**
     * A method that handles received ACKs
     *
     * @param ack
     */
    void ackReceived(Packet ack);

    /**
     * A method that returns the packet that should be sent
     *
     * @return packet to be sent
     */
    Packet send();

    /**
     * A method that checks if a packet should be retransmitted or not
     *
     * @param packet that should be checked for possible retransmission
     * @return true if the packet should be retransmitted
     */
    boolean canRetransmit(Packet packet);

    /**
     * A method that returns a packet that should be fast retransmitted
     *
     * @return Packet if there is a packet to fast-retransmit, else null
     */
    Packet fastRetransmit();

    /**
     * A method that is called to increase the size of the sendigwindow.
     */
    void increase();

    /**
     * A method that is called to decrease the size of the sendigwindow.
     */
    void decrease();

    /**
     * A method that calculates the sending window packet index
     * of the given packet
     *
     * @param packet to calculate packet index of
     * @return the packet index of the given packet
```

```
63          */
64         int sendingPacketIndex(Packet packet);
65
66         /**
67          * A method that returns the connection
68          *
69          * @return connection
70          */
71         Connection getConnection();
72
73         /**
74          * A method that checks if the sending window is empty or not
75          *
76          * @return true if the sending window is empty
77          */
78         boolean isQueueEmpty();
79
80         /**
81          * A method that returns the stats of the sending window
82          *
83          * @return the TCPStats
84          */
85         TCPStats getStats();
86
87 }
```

```java
package org.example.protocol.window.sending;

import org.example.data.Packet;
import org.example.data.PacketBuilder;
import org.example.data.Payload;
import org.example.protocol.Connection;
import org.example.protocol.window.Window;
import org.example.simulator.statistics.TCPStats;
import org.example.util.BoundedQueue;
import org.javatuples.Pair;

import java.util.Comparator;
import java.util.List;

public class SlidingWindow extends Window implements SendingWindow, BoundedQueue<Packet> {

    private static final int DEFAULT_CONGESTION_WINDOW_CAPACITY = 1;
    private final int numDupAckFastRetransmitTrigger;
    private final List<Pair<Integer, Payload>> payloadsToSend;
    private final int receiverWindowSize;
    private final boolean isReno;
    private final Connection connection;
    private final TCPStats stats;
    private boolean seriousLossDetected = false;
    private int numPacketsReceivedWithoutIncreasingWindow;
    private int ssthresh;
    private int dupAckCount;
    private boolean fastRetransmitted;

    public SlidingWindow(int receiverWindowCapacity, boolean isReno, Connection connection, Comparator<Packet> comparator, List<Pair<Integer, Payload>> payloadsToSend, TCPStats stats) {
        super(DEFAULT_CONGESTION_WINDOW_CAPACITY, comparator);
        this.payloadsToSend = payloadsToSend;
        this.receiverWindowSize = receiverWindowCapacity;
        this.numPacketsReceivedWithoutIncreasingWindow = 0;
        this.ssthresh = receiverWindowCapacity; //initial value is essentially no ssthresh
        this.isReno = isReno;
        this.dupAckCount = 0;
        this.fastRetransmitted = false;
        this.connection = connection;
        this.numDupAckFastRetransmitTrigger = 3;
        this.stats = stats;

    }

    @Override
    public boolean isWaitingForAck() {
        return this.isFull();
    }

    @Override
    public void ackReceived(Packet ack) {
        this.stats.ackReceived();
        int ackIndex = this.sendingPacketIndex(ack);

        boolean dupAck = ackIndex == -1;
        if (dupAck) {
            this.dupAckCount++;
            return;
```

```
59          }
60          this.dupAckCount = 0;
61
62          for (var i = 0; i <= ackIndex; i++) {
63              if (this.isEmpty()) break;
64              this.poll();
65              this.increase();
66              this.seriousLossDetected = false;
67          }
68          this.connection.update(ack);
69      }
70
71      @Override
72      public Packet send() {
73          int nextPacketSeqNum = this.connection.getNextSequenceNumber() + this.size()
                ;
74          Pair<Integer, Payload> indexAndPayload = this.payloadsToSend.remove(0);
75
76          var packet = new PacketBuilder()
77                  .withConnection(this.connection)
78                  .withPayload(indexAndPayload.getValue1())
79                  .withSequenceNumber(nextPacketSeqNum)
80                  .withIndex(indexAndPayload.getValue0())
81                  .build();
82
83          if (this.contains(packet)) throw new IllegalStateException("can't add same
                packet twice");
84          if (super.offer(packet)) {
85              return packet;
86          }
87          return null;
88      }
89
90
91      @Override
92      public boolean canRetransmit(Packet packet) {
93          if (this.contains(packet)) {
94              if (this.sendingPacketIndex(packet) == 0) {
95                  if (this.fastRetransmitted) this.fastRetransmitted = false;
96                  else this.decrease();
97              }
98              if (this.sendingPacketIndex(packet) >= this.getWindowCapacity() - 1)
                    this.seriousLossDetected = true;
99              this.dupAckCount = 0;
100             return true;
101         }
102         return false;
103     }
104
105     @Override
106     public Packet fastRetransmit() {
107         if (this.dupAckCount > this.numDupAckFastRetransmitTrigger) {
108             this.dupAckCount = 0;
109             this.decrease(false);
110             this.fastRetransmitted = true;
111             return this.peek();
112         }
113         return null;
114     }
115
116     @Override
117     public void increase() {
118         if (this.getWindowCapacity() >= this.receiverWindowSize) return;
```

```
119         if (this.ssthresh > this.getWindowCapacity()) slowStart();
120         else congestionAvoidance();
121     }
122
123     private void slowStart() {
124         this.setBound(this.getWindowCapacity() + 1);
125         this.stats.trackCwnd(this.getWindowCapacity());
126     }
127
128     private void congestionAvoidance() {
129         boolean allPacketsInOneWindowReceived =
130                 this.numPacketsReceivedWithoutIncreasingWindow % this.
                        getWindowCapacity() == 0
131                         && this.numPacketsReceivedWithoutIncreasingWindow != 0;
132         if (allPacketsInOneWindowReceived) {
133             this.numPacketsReceivedWithoutIncreasingWindow = 0;
134             this.setBound(this.getWindowCapacity() + 1);
135             this.stats.trackCwnd(this.getWindowCapacity());
136             return;
137         }
138         this.numPacketsReceivedWithoutIncreasingWindow++;
139     }
140
141     @Override
142     public void decrease() {
143         this.decrease(true);
144     }
145
146     public void decrease(boolean timeout) {
147         this.stats.trackCwnd(this.getWindowCapacity());
148         this.ssthresh = this.findNewSSThresh();
149         int newWindowSize = this.findNewWindowSize(timeout);
150         this.setBound(newWindowSize);
151         this.stats.trackCwnd(this.getWindowCapacity());
152     }
153
154
155     private int findNewWindowSize(boolean timeout) {
156         if (this.isReno && !timeout) return this.ssthresh;
157         return DEFAULT_CONGESTION_WINDOW_CAPACITY;
158     }
159
160     private int findNewSSThresh() {
161         return Math.max((int) (this.getWindowCapacity() / 2.0),
                DEFAULT_CONGESTION_WINDOW_CAPACITY);
162     }
163
164     @Override
165     public int sendingPacketIndex(Packet packet) {
166         int packetSeqNum = packet.getSequenceNumber();
167         int connSeqNum = this.connection.getNextSequenceNumber();
168         return packetSeqNum - connSeqNum;
169     }
170
171     @Override
172     public Connection getConnection() {
173         return connection;
174     }
175
176
177     @Override
178     public boolean isQueueEmpty() {
179         return this.payloadsToSend.isEmpty();
```

```
180     }
181
182     @Override
183     public TCPStats getStats() {
184         return this.stats;
185     }
186 }
```

```java
package org.example.simulator;

import org.example.simulator.events.Event;
import org.example.util.Util;

import java.util.PriorityQueue;
import java.util.Queue;

public class EventHandler {

    private final Queue<Event> events;

    public EventHandler() {
        this.events = new PriorityQueue<>();
    }

    public void addEvent(Event event) {
        this.events.add(event);
    }

    public Event peekEvent() {
        return this.events.peek();
    }

    public Queue<Event> getEvents() {
        return events;
    }

    public int getNumberOfEvents() {
        return this.events.size();
    }

    public boolean singleRun() {
        var event = this.events.poll();
        if (event == null) {
            if (!this.events.isEmpty())
                throw new IllegalStateException("get null event when events queue
                    are nonempty!");
            return false;
        }
        Util.tickTime(event);
        event.run();
        event.generateNextEvent(this.events);
        return true;
    }

    public void run() {
        while (singleRun()) ;
    }
}
```

```java
package org.example.simulator.events;

import org.example.network.Channel;
import org.example.network.interfaces.Endpoint;
import org.example.network.interfaces.NetworkNode;
import org.example.protocol.TCP;
import org.example.simulator.events.run.RunEndpointEvent;
import org.example.simulator.events.run.RunNetworkNodeEvent;
import org.example.simulator.events.tcp.RunTCPEvent;

import java.util.Queue;

public class ChannelEvent extends Event {

    private final Channel channel;
    private boolean channelSuccess;

    public ChannelEvent(Channel channel) {
        super(channel);
        this.channel = channel;
        this.channelSuccess = false;
    }

    @Override
    public void run() {
        this.channelSuccess = this.channel.channel();
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        if (this.channelSuccess) {
            NetworkNode nextNode = this.channel.getDestination();
            if (nextNode instanceof TCP) {
                var tcp = ((TCP) nextNode).getMainFlow();
                events.add(new RunTCPEvent(tcp));
                return;
            }
            assert !nextNode.inputBufferIsEmpty() : "The next NetworkNode has no
                packet in the input buffer";
            if (nextNode instanceof Endpoint) {
                events.add(new RunEndpointEvent((Endpoint) nextNode));
                return;
            }
            events.add(new RunNetworkNodeEvent(nextNode));
            return;
        }

        NetworkNode nextNode = this.channel.getDestination();
        if (nextNode instanceof TCP) {
            var tcp = ((TCP) nextNode).getMainFlow();
            if (!tcp.inputBufferIsEmpty()) return;
            events.add(new RunTCPEvent(tcp));
        }
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
```

```
62            return super.hashCode();
63        }
64    }
```

```java
package org.example.simulator.events;

import org.example.network.Channel;
import org.example.network.interfaces.NetworkNode;
import org.example.util.Util;

import java.util.Queue;

public abstract class Event implements Comparable<Event> {

    private final long instant;

    protected Event(long delay) {
        this.instant = this.findInstant(delay);
    }

    protected Event() {
        this.instant = findInstant(0);
    }

    protected Event(NetworkNode node) {
        if (node == null) throw new IllegalArgumentException("node is null");
        this.instant = this.findInstant(node.delay());
    }

    protected Event(Channel channel) {
        if (channel == null) throw new IllegalArgumentException("channel is null");
        this.instant = this.findInstant(channel.propagationDelay());
    }

    private long findInstant(long delay) {
        return Util.getTime() + delay;
    }

    public abstract void run();

    public abstract void generateNextEvent(Queue<Event> events);

    public long getInstant() {
        return this.instant;
    }

    @Override
    public int compareTo(Event o) {
        return Long.compare(this.getInstant(), o.getInstant());
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }
}
```

```java
package org.example.simulator.events;

public abstract class EventGenerator extends Event {

    protected EventGenerator(long delay) {
        super(delay);
    }

    @Override
    public void run() {
        //EventGenerators should only generate new events and not run
    }
}
```

```java
package org.example.simulator.events;

import org.example.data.Packet;
import org.example.network.Channel;
import org.example.network.interfaces.Endpoint;
import org.example.protocol.MPTCP;

import java.util.List;
import java.util.Queue;

public class RouteEvent extends Event {

    private final Packet packet;
    private final Endpoint endpoint;

    public RouteEvent(Endpoint endpoint, Packet packet) {
        super();
        if (endpoint instanceof MPTCP) throw new IllegalArgumentException("Should only handle Regular tcp or subflows");
        this.endpoint = endpoint;
        this.packet = packet;
    }

    @Override
    public void run() {
        this.endpoint.route(this.packet);
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        List<Channel> channelsUsed = this.endpoint.getChannelsUsed();
        var channel = channelsUsed.get(0);
        if (channel == null) return;
        events.add(new ChannelEvent(channel));
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

}
```

```java
package org.example.simulator.events.run;

import org.example.network.interfaces.Endpoint;
import org.example.simulator.events.Event;

import java.util.Queue;


public class RunEndpointEvent extends Event {

    private final Endpoint endpoint;

    public RunEndpointEvent(Endpoint endpoint) {
        super(endpoint);
        this.endpoint = endpoint;
    }

    @Override
    public void run() {
        this.endpoint.run();
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        // do nothing because and an endpoint without tcp functionality is not doing
            anything
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

}
```

```java
package org.example.simulator.events.run;

import org.example.network.Channel;
import org.example.network.interfaces.NetworkNode;
import org.example.simulator.events.ChannelEvent;
import org.example.simulator.events.Event;

import java.util.List;
import java.util.Queue;

public class RunNetworkNodeEvent extends Event {

    private final NetworkNode node;

    public RunNetworkNodeEvent(NetworkNode node) {
        super(node);
        this.node = node;
    }

    @Override
    public void run() {
        assert !this.node.inputBufferIsEmpty() : "RunNetworkNodeEvent added, but no
            packet to be sent";
        this.node.run();
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        List<Channel> channelsUsed = this.node.getChannelsUsed();
        assert channelsUsed.size() == 1 : "Multiple channels used in one
            NetworkNodeEvent";
        var channel = channelsUsed.get(0);
        events.add(new ChannelEvent(channel));
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

}
```

```java
package org.example.simulator.events.tcp;

import org.example.data.Packet;
import org.example.network.Channel;
import org.example.protocol.MPTCP;
import org.example.protocol.TCP;
import org.example.simulator.events.ChannelEvent;
import org.example.simulator.events.Event;

import java.util.List;
import java.util.Queue;

public class RunTCPEvent extends Event {

    private final TCP tcp;
    private List<Packet> packetsSent;
    private boolean scheduleFirstSend;

    public RunTCPEvent(TCP tcp) {
        super(tcp);
        this.tcp = tcp;
        this.scheduleFirstSend = false;
    }

    public RunTCPEvent(TCP tcp, long delay) {
        super(delay);
        this.tcp = tcp;
        this.scheduleFirstSend = false;
    }

    @Override
    public void run() {
        if (this.tcp instanceof MPTCP) {
            for (TCP subflow : ((MPTCP) this.tcp).getSubflows()) {
                if (subflow.isConnected()) {
                    this.tcp.handleIncoming();
                    this.packetsSent = this.tcp.trySend();
                    return;
                }
            }
        } else if (this.tcp.isConnected()) {
            this.tcp.handleIncoming();
            this.packetsSent = this.tcp.trySend();
            return;
        }
        this.tcp.handleIncoming();
        this.scheduleFirstSend = true;
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        for (Channel channel : this.tcp.getChannelsUsed()) {
            events.add(new ChannelEvent(channel));
        }

        if (this.scheduleFirstSend) {
            //add delay before sending first packet
            //this is important if we have a "downloading situation"
            events.add(new RunTCPEvent(this.tcp, 1000000));
            return;
        }

```

```
63
64          if (packetsSent == null) return;
65          for (Packet packet : this.packetsSent) {
66              events.add(new TCPRetransmitEventGenerator(packet, 0));
67          }
68      }
69
70      @Override
71      public boolean equals(Object obj) {
72          return super.equals(obj);
73      }
74
75      @Override
76      public int hashCode() {
77          return super.hashCode();
78      }
79  }
```

```java
package org.example.simulator.events.tcp;

import org.example.network.Channel;
import org.example.protocol.TCP;
import org.example.simulator.events.ChannelEvent;
import org.example.simulator.events.Event;

import java.util.Queue;

public class TCPConnectEvent extends Event {

    private final TCP client;
    private final TCP host;
    private final int numAttempts;

    private TCPConnectEvent(int delay, TCP client, TCP host, int numAttempts) {
        super(delay);
        this.client = client;
        this.host = host;
        this.numAttempts = numAttempts;
    }

    public TCPConnectEvent(TCP client, TCP host) {
        super();
        this.client = client;
        this.host = host;
        this.numAttempts = 0;
    }

    @Override
    public void run() {
        if (this.client.isConnected()) return;
        this.client.connect(this.host);
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        if (this.numAttempts > this.client.getNumberOfFlows() * 3) return;
        if (this.client.isConnected()) return;

        for (Channel channel : this.client.getChannelsUsed()) {
            events.add(new ChannelEvent(channel));
        }
        events.add(new TCPConnectEvent(10000000, this.client, this.host, this.
            numAttempts + 1));
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

```java
package org.example.simulator.events.tcp;

import org.example.data.Packet;
import org.example.protocol.ClassicTCP;
import org.example.protocol.TCP;
import org.example.simulator.events.Event;
import org.example.simulator.events.EventGenerator;
import org.example.simulator.events.RouteEvent;

import java.util.Queue;

public class TCPRetransmitEventGenerator extends EventGenerator {

    private final TCP tcp;
    private final Packet packet;
    private final int numAttempts;

    public TCPRetransmitEventGenerator(Packet packet, int numAttempts) {
        super(((TCP) packet.getOrigin()).getRTO());
        this.tcp = (TCP) packet.getOrigin();
        this.packet = packet;
        this.numAttempts = numAttempts;
    }

    @Override
    public void generateNextEvent(Queue<Event> events) {
        if (this.numAttempts > 3) return;
        if (!this.tcp.isConnected()) return;
        if (this.tcp.canRetransmit(this.packet)) {
            ((ClassicTCP) this.tcp).getStats().packetRetransmit();
            events.add(new RouteEvent(this.tcp, this.packet));
            events.add(new TCPRetransmitEventGenerator(this.packet, this.numAttempts
                + 1));
        }
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }


}
```

```java
package org.example.simulator.statistics;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.example.data.Packet;
import org.example.util.Util;
import org.knowm.xchart.VectorGraphicsEncoder;
import org.knowm.xchart.XYChart;
import org.knowm.xchart.XYChartBuilder;
import org.knowm.xchart.XYSeries;
import org.knowm.xchart.style.Styler;

import java.awt.*;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;


public abstract class Stats {

    protected static final String DIR = "./charts/";
    protected static final Styler.ChartTheme theme = Styler.ChartTheme.Matlab;
    protected static final int CHART_WIDTH = 1500;
    protected static final int CHART_HEIGHT = 600;
    protected static final Font TITLE_FONT = new Font("titleFont", 0, 30);
    protected static final Font AXIS_FONT = new Font("axisFont", 0, 25);
    protected static final Font TICK_FONT = new Font("tickFont", 0, 15);

    protected static final int TIMESCALE = 1000;

    // Arrival
    protected final ArrayList<Packet> arrivalPacket = new ArrayList<>();
    protected final ArrayList<Double> arrivalTime = new ArrayList<>();
    protected final ArrayList<Integer> arrivalNum = new ArrayList<>();

    // Departure
    protected final ArrayList<Packet> departurePacket = new ArrayList<>();
    protected final ArrayList<Double> departureTime = new ArrayList<>();
    protected final ArrayList<Integer> departureNum = new ArrayList<>();

    // Inter arrival times
    protected final ArrayList<Double> interArrivalTimes = new ArrayList<>();

    // Time in system
    protected final ArrayList<Double> timeInSystem = new ArrayList<>();

    // number of packets in system
    protected final ArrayList<Integer> numPacketsInSystem = new ArrayList<>();

    private double meanArrivalRate;
    private double meanTimeInSystem;
    private double meanNumPacketsInSystem;

    protected abstract String fileName();

    protected abstract void additionalCalculations();

    private void doCalculations() {
        if (this.interArrivalTimes.isEmpty()) return;
        this.meanArrivalRate = this.arrivalNum.size() / (this.arrivalTime.get(this.
            arrivalTime.size() - 1));
```

```
62
63          this.meanTimeInSystem = this.timeInSystem.stream().mapToDouble(Double::
                doubleValue).average().getAsDouble();
64
65          //Little's law
66          // L = 1/E[A] * W
67          this.meanNumPacketsInSystem = this.meanArrivalRate * this.meanTimeInSystem;
68      }
69
70      public void packetArrival(Packet packet) {
71          this.arrivalPacket.add(packet);
72          this.arrivalNum.add(this.arrivalNum.size());
73          this.arrivalTime.add((double) Util.seeTime() / TIMESCALE);
74
75          // number of packets in system
76          this.addPacketsInSystem();
77
78          // inter arrival time calculation
79          this.addInterArrivalTime();
80      }
81
82      public void packetDeparture(Packet packet) {
83          this.departurePacket.add(packet);
84          this.departureNum.add(this.arrivalNum.size());
85          this.departureTime.add((double) Util.seeTime() / TIMESCALE);
86
87          // number of packets in system
88          this.removePacketsInSystem();
89
90          // Time in system calculation
91          this.addTimeInSystem(packet);
92      }
93
94      private void addTimeInSystem(Packet packet) {
95          int packetArrivalIndex = this.arrivalIndexOf(packet);
96          int packetDepartureIndex = this.departureIndexOf(packet);
97
98          if (packetArrivalIndex == -1 || packetDepartureIndex == -1) {
99              return;
100         }
101
102         double arrival = this.arrivalTime.get(packetArrivalIndex);
103         double departure = this.departureTime.get(packetDepartureIndex);
104         double inSystem = departure - arrival;
105
106         double minTime = (double) 10 / (double) TIMESCALE;
107         if (inSystem < minTime) inSystem = minTime;
108         if (inSystem < 0) throw new IllegalStateException("packet cannot be in
                system a negative amount of time");
109         this.timeInSystem.add(inSystem);
110     }
111
112     private void addInterArrivalTime() {
113         int n = this.arrivalNum.size() - 1;
114         if (n > 1) {
115             double interArrivalTime = this.arrivalTime.get(n) - this.arrivalTime.get
                    (n - 1);
116             if (interArrivalTime < 0) throw new IllegalStateException("interarrival
                    time is less then 0");
117             this.interArrivalTimes.add(interArrivalTime);
118         } else {
119             this.interArrivalTimes.add(0.0);
120         }
```

```
121     }
122
123     private void addPacketsInSystem() {
124         if (this.numPacketsInSystem.isEmpty()) {
125             this.numPacketsInSystem.add(1);
126         } else {
127             this.numPacketsInSystem.add(this.numPacketsInSystem.get(this.
                    numPacketsInSystem.size() - 1) + 1);
128         }
129     }
130
131     private void removePacketsInSystem() {
132         if (!this.numPacketsInSystem.isEmpty()) {
133             this.numPacketsInSystem.add(this.numPacketsInSystem.get(this.
                    numPacketsInSystem.size() - 1) - 1);
134         }
135     }
136
137     private int arrivalIndexOf(Packet packet) {
138         return this.arrivalPacket.indexOf(packet);
139     }
140
141     private int departureIndexOf(Packet packet) {
142         return this.departurePacket.indexOf(packet);
143     }
144
145
146     public void createArrivalChart() {
147         if (this.arrivalTime.isEmpty() || this.arrivalNum.isEmpty()) return;
148         if (this.arrivalTime.size() != this.arrivalNum.size())
149             throw new IllegalStateException("the arrays must be of equal length");
150
151         XYChart chart = new XYChartBuilder()
152                 .width(CHART_WIDTH)
153                 .height(CHART_HEIGHT)
154                 .xAxisTitle("Packet Arrival-Time")
155                 .yAxisTitle("Packet")
156                 .title("Packet Arrivals")
157                 .theme(theme)
158                 .build();
159         chart.getStyler().setChartTitleFont(TITLE_FONT);
160         chart.getStyler().setAxisTitleFont(AXIS_FONT);
161         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
162         chart.getStyler().setLegendVisible(false);
163         chart.getStyler().setXAxisLabelRotation(45);
164         chart.addSeries("Packet Arrivals", this.arrivalTime, this.arrivalNum);
165         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Scatter);
166         saveChart(chart, "ArrivalChart_");
167     }
168
169     public void createDepartureChart() {
170         if (this.departureTime.isEmpty() || this.departureNum.isEmpty()) return;
171         if (this.departureTime.size() != this.departureNum.size())
172             throw new IllegalStateException("the arrays must be of equal length");
173
174         XYChart chart = new XYChartBuilder()
175                 .width(CHART_WIDTH)
176                 .height(CHART_HEIGHT)
177                 .xAxisTitle("Packet Departure-Time")
178                 .yAxisTitle("Packet")
179                 .title("Packet Departures")
180                 .theme(theme)
```

```
181                 .build();
182         chart.getStyler().setChartTitleFont(TITLE_FONT);
183         chart.getStyler().setAxisTitleFont(AXIS_FONT);
184         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
185         chart.getStyler().setLegendVisible(false);
186         chart.getStyler().setXAxisLabelRotation(45);
187         chart.addSeries("Packet Departures", this.departureTime, this.departureNum);
188         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Scatter);
189         saveChart(chart, "DepartureChart_");
190     }
191
192     public void createInterArrivalChart() {
193         XYChart chart = new XYChartBuilder()
194                 .width(CHART_WIDTH)
195                 .height(CHART_HEIGHT)
196                 .xAxisTitle("Arrival Time")
197                 .yAxisTitle("Interarrival Time")
198                 .title("Packet Interarrival-Time")
199                 .theme(theme)
200                 .build();
201         chart.getStyler().setChartTitleFont(TITLE_FONT);
202         chart.getStyler().setAxisTitleFont(AXIS_FONT);
203         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
204         chart.getStyler().setLegendVisible(false);
205         chart.getStyler().setXAxisLabelRotation(45);
206         chart.addSeries("Interarrival Times", this.arrivalTime, this.
                interArrivalTimes);
207         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Scatter);
208         saveChart(chart, "InterarrivalTime_");
209     }
210
211     public void createTimeInSystemChart() {
212         XYChart chart = new XYChartBuilder()
213                 .width(CHART_WIDTH)
214                 .height(CHART_HEIGHT)
215                 .xAxisTitle("Time")
216                 .yAxisTitle("Time In System")
217                 .title("Packet Time In System")
218                 .theme(theme)
219                 .build();
220         chart.getStyler().setChartTitleFont(TITLE_FONT);
221         chart.getStyler().setAxisTitleFont(AXIS_FONT);
222         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
223         chart.getStyler().setLegendVisible(false);
224         chart.getStyler().setXAxisLabelRotation(45);
225         chart.addSeries("Time in system", this.departureTime, this.timeInSystem);
226         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Scatter);
227         saveChart(chart, "TimeInSystem_");
228     }
229
230     public void createNumberOfPacketsInSystemChart() {
231         XYChart chart = new XYChartBuilder()
232                 .width(CHART_WIDTH)
233                 .height(CHART_HEIGHT)
234                 .xAxisTitle("Time")
235                 .yAxisTitle("Number of packets in system")
236                 .title("number of packets in system")
237                 .theme(theme)
238                 .build();
239         chart.getStyler().setChartTitleFont(TITLE_FONT);
```

```
240         chart.getStyler().setAxisTitleFont(AXIS_FONT);
241         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
242         chart.getStyler().setLegendVisible(false);
243         chart.getStyler().setXAxisLabelRotation(45);
244         chart.addSeries("Packets in system", this.combine(this.arrivalTime, this.
                departureTime), this.numPacketsInSystem);
245         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Scatter);
246         saveChart(chart, "PacketsInSystem_");
247     }
248 
249     private List<Double> combine(List<Double> l1, List<Double> l2) {
250         List<Double> result = new ArrayList<>();
251         result.addAll(l1);
252         result.addAll(l2);
253         Collections.sort(result);
254         return result;
255     }
256 
257     public double getMeanArrivalRate() {
258         return meanArrivalRate;
259     }
260 
261     public double getMeanTimeInSystem() {
262         return meanTimeInSystem;
263     }
264 
265     public double getMeanNumPacketsInSystem() {
266         return meanNumPacketsInSystem;
267     }
268 
269 
270     protected void saveChart(XYChart chart, String chartName) {
271         try {
272             VectorGraphicsEncoder.saveVectorGraphic(chart, DIR + chartName +
                    fileName(), VectorGraphicsEncoder.VectorGraphicsFormat.SVG);
273         } catch (IOException e) {
274         }
275     }
276 
277     @Override
278     public String toString() {
279         this.doCalculations();
280         this.additionalCalculations();
281         var mapper = new ObjectMapper();
282         try {
283             var formattedString = mapper.writeValueAsString(this)
284                     .replace("{", "{\n      ")
285                     .replace("}", "\n}")
286                     .replace(",", ",\n      ");
287             return this.fileName() + " " + formattedString;
288         } catch (JsonProcessingException e) {
289             return "fail";
290         }
291     }
292 }
```

```java
package org.example.simulator.statistics;

import org.example.data.Packet;
import org.example.network.address.Address;
import org.example.util.Util;
import org.knowm.xchart.XYChart;
import org.knowm.xchart.XYChartBuilder;
import org.knowm.xchart.XYSeries;
import org.knowm.xchart.style.markers.SeriesMarkers;

import java.util.ArrayList;

public class TCPStats extends Stats {

    private final String filename;
    // CWND
    private final ArrayList<Integer> congestionWindowCapacities = new ArrayList<>();
    private final ArrayList<Double> congestionWindowTime = new ArrayList<>();
    private int numberOfPacketsSent; //total number of packets sent (both normal and
        retransmissions)
    private int numberOfPacketsRetransmitted; //total number of packets
       retransmitted
    private int numberOfPacketsFastRetransmitted; // total number of packets dropped
    private int numberOfPacketsArrived; //total number of packets that are enqueued
    private int numberOfAcksReceived; //total number of ACKs received
    private int numberOfPacketsReceived; //total number of packets received

    private double goodput;
    private double lossRate;

    public TCPStats(Address address) {
        this.filename = address.toString();
    }

    @Override
    protected void additionalCalculations() {
        this.setGoodput();
        this.setLossRate();
    }

    @Override
    protected String fileName() {
        return this.filename;
    }

    public void packetSend() {
        this.numberOfPacketsSent++;
    }

    public void packetRetransmit() {
        this.numberOfPacketsRetransmitted++;
    }

    public void packetFastRetransmit() {
        this.numberOfPacketsFastRetransmitted++;
    }

    @Override
    public void packetArrival(Packet packet) {
        this.numberOfPacketsArrived++;
        super.packetArrival(packet);
    }
```

```
61
62      @Override
63      public void packetDeparture(Packet packet) {
64          this.numberOfPacketsReceived++;
65          super.packetDeparture(packet);
66      }
67
68      public void ackReceived() {
69          this.numberOfAcksReceived++;
70      }
71
72      public int getNumberOfPacketsSent() {
73          return this.numberOfPacketsSent;
74      }
75
76      public int getNumberOfPacketsRetransmitted() {
77          return this.numberOfPacketsRetransmitted;
78      }
79
80      public int getNumberOfPacketsFastRetransmitted() {
81          return this.numberOfPacketsFastRetransmitted;
82      }
83
84      public int getNumberOfAcksReceived() {
85          return this.numberOfAcksReceived;
86      }
87
88      public int getNumberOfPacketsArrived() {
89          return numberOfPacketsArrived;
90      }
91
92      public int getNumberOfPacketsReceived() {
93          return numberOfPacketsReceived;
94      }
95
96      public double getGoodput() {
97          return goodput;
98      }
99
100     public double getLossRate() {
101         return lossRate;
102     }
103
104     private void setGoodput() {
105         if (congestionWindowTime.isEmpty()) return;
106         this.goodput = (double) this.numberOfPacketsSent / (this.
                congestionWindowTime.get(this.congestionWindowTime.size() - 1) / (double
                ) TIMESCALE);
107     }
108
109     private void setLossRate() {
110         if (this.numberOfPacketsSent == 0) {
111             this.lossRate = 0;
112             return;
113         }
114
115         this.lossRate = (this.numberOfPacketsFastRetransmitted + this.
                numberOfPacketsRetransmitted) / (double) this.numberOfPacketsSent;
116     }
117
118     public void trackCwnd(int cwnd) {
119         congestionWindowCapacities.add(cwnd);
120         congestionWindowTime.add((double) Util.seeTime());
```

```
121     }
122
123     public void createCWNDChart() {
124         if (this.congestionWindowTime.isEmpty() || this.congestionWindowCapacities.
                isEmpty()) return;
125         if (this.congestionWindowTime.size() != this.congestionWindowCapacities.size
                ())
126             throw new IllegalStateException("the arrays must be of equal length");
127
128         XYChart chart = new XYChartBuilder()
129                 .width(CHART_WIDTH)
130                 .height(CHART_HEIGHT)
131                 .xAxisTitle("Time")
132                 .yAxisTitle("CWND Size")
133                 .title("Congestion Window Capacity")
134                 .theme(theme)
135                 .build();
136         chart.getStyler().setChartTitleFont(TITLE_FONT);
137         chart.getStyler().setAxisTitleFont(AXIS_FONT);
138         chart.getStyler().setAxisTickLabelsFont(TICK_FONT);
139         chart.getStyler().setLegendVisible(false);
140         chart.getStyler().setXAxisLabelRotation(45);
141         chart.addSeries("CWND", congestionWindowTime, congestionWindowCapacities).
                setMarker(SeriesMarkers.NONE);
142         chart.getStyler().setDefaultSeriesRenderStyle(XYSeries.XYSeriesRenderStyle.
                Area);
143         this.saveChart(chart, "CWNDChart_");
144     }
145 }
```

```java
package org.example.util;

import java.util.Collection;
import java.util.Comparator;
import java.util.Iterator;
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

public class BoundedPriorityBlockingQueue<T> implements BoundedQueue<T> {


    private PriorityBlockingQueue<T> pbq;
    private int bound;

    public BoundedPriorityBlockingQueue(int bound, Comparator<T> comparator) {
        this.pbq = new PriorityBlockingQueue<>(bound, comparator);
        this.bound = bound;
    }

    public BoundedPriorityBlockingQueue(int bound) {
        this.pbq = new PriorityBlockingQueue<>(bound);
        this.bound = bound;
    }

    @Override
    public boolean isFull() {
        return this.pbq.size() >= this.bound;
    }

    @Override
    public boolean offer(T t) {
        if (isFull()) return false;
        return this.pbq.offer(t);
    }

    @Override
    public void put(T t) throws InterruptedException {
        if (isEmpty()) return;
        this.pbq.put(t);
    }

    @Override
    public boolean offer(T t, long l, TimeUnit timeUnit) throws InterruptedException
         {
        if (isFull()) return false;
        return this.pbq.offer(t);
    }

    @Override
    public T take() throws InterruptedException {
        return this.pbq.take();
    }

    @Override
    public T poll(long l, TimeUnit timeUnit) throws InterruptedException {
        return this.pbq.poll();
    }

    @Override
    public int remainingCapacity() {
        return this.bound - this.pbq.size();
    }
```

```
62
63      @Override
64      public int bound() {
65          return this.bound;
66      }
67
68      @Override
69      public void setBound(int bound) {
70          this.bound = bound;
71      }
72
73      /**
74       * @deprecated because it may violate the bound constraint of this class.
75       * New implementation that follows the capacity constraint is needed
76       */
77      @Override
78      @Deprecated(since = "4.5", forRemoval = true)
79      public int drainTo(Collection<? super T> collection) {
80          return this.pbq.drainTo(collection);
81      }
82
83      /**
84       * @deprecated because it may violate the bound constraint of this class.
85       * New implementation that follows the capacity constraint is needed
86       */
87      @Override
88      @Deprecated(since = "4.5", forRemoval = true)
89      public int drainTo(Collection<? super T> collection, int i) {
90          return this.pbq.drainTo(collection, i);
91      }
92
93      @Override
94      public T remove() {
95          return this.pbq.remove();
96      }
97
98      @Override
99      public T poll() {
100         return this.pbq.poll();
101     }
102
103     @Override
104     public T element() {
105         return this.pbq.element();
106     }
107
108     @Override
109     public T peek() {
110         return this.pbq.peek();
111     }
112
113     @Override
114     public int size() {
115         return this.pbq.size();
116     }
117
118     @Override
119     public boolean isEmpty() {
120         return this.pbq.isEmpty();
121     }
122
123     @Override
124
```

```
125     public boolean contains(Object o) {
126         return this.pbq.contains(o);
127     }
128 
129     @Override
130     public Object[] toArray() {
131         return this.pbq.toArray();
132     }
133 
134     @Override
135     public <E> E[] toArray(E[] ts) {
136         return this.pbq.toArray(ts);
137     }
138 
139     @Override
140     public boolean add(T t) {
141         if (isFull()) return false;
142         return this.pbq.add(t);
143     }
144 
145     @Override
146     public boolean remove(Object o) {
147         return this.pbq.remove(o);
148     }
149 
150     @Override
151     public boolean containsAll(Collection<?> collection) {
152         return this.pbq.containsAll(collection);
153     }
154 
155     /**
156      * @deprecated because it may violate the bound constraint of this class.
157      * New implementation that follows the capacity constraint is needed
158      */
159     @Override
160     @Deprecated(since = "4.5", forRemoval = true)
161     public boolean addAll(Collection<? extends T> collection) {
162         return this.pbq.addAll(collection);
163     }
164 
165     @Override
166     public boolean removeAll(Collection<?> collection) {
167         return this.pbq.removeAll(collection);
168     }
169 
170     /**
171      * @deprecated because it may violate the bound constraint of this class.
172      * New implementation that follows the capacity constraint is needed
173      */
174     @Override
175     @Deprecated(since = "4.5", forRemoval = true)
176     public boolean retainAll(Collection<?> collection) {
177         return this.pbq.retainAll(collection);
178     }
179 
180     @Override
181     public void clear() {
182         this.pbq.clear();
183     }
184 
185     @Override
186     public Iterator<T> iterator() {
187         return this.pbq.iterator();
```

```
188     }
189 }
```

```
1  package org.example.util;
2
3  import java.util.Collection;
4  import java.util.Queue;
5  import java.util.concurrent.BlockingQueue;
6
7  public interface BoundedQueue<T> extends Iterable<T>, Collection<T>, BlockingQueue<T
       >, Queue<T> {
8
9      /**
10      * A method to check is the queue is full
11      *
12      * @return True if the queue is full.
13      */
14     boolean isFull();
15
16     /**
17      * A method that returns the bound of the BoundedQueue
18      *
19      * @return the bound
20      */
21     int bound();
22
23     /**
24      * A method to set the bound of the BoundedQueue
25      *
26      * @param bound
27      */
28     void setBound(int bound);
29 }
```

```java
package org.example.util;

import org.apache.commons.math3.distribution.PoissonDistribution;
import org.apache.commons.math3.random.RandomGeneratorFactory;
import org.example.simulator.events.Event;

import java.util.Random;

public class Util {

    public static final Random random = new Random(1337);
    private static long time = 0;

    private Util() {
        throw new IllegalStateException("Utility class");
    }

    public static void setSeed(int seed) {
        random.setSeed(seed);
    }

    public static double getNextRandomDouble() {
        return random.nextDouble();
    }

    public static int getNextRandomInt() {
        return random.nextInt();
    }

    public static int getNextRandomInt(int bound) {
        return random.nextInt(bound);
    }

    public static double getNextGaussian() {
        return random.nextGaussian();
    }

    public static void tickTime(Event event) {
        time = event.getInstant();
    }

    public static long getTime() {
        return time;
    }

    public static long seeTime() {
        return time;
    }

    public static void resetTime() {
        time = 0;
    }

    public static PoissonDistribution getPoissonDistribution(double mean) {
        return new PoissonDistribution(RandomGeneratorFactory.createRandomGenerator(
            random), mean, PoissonDistribution.DEFAULT_EPSILON, PoissonDistribution.
            DEFAULT_MAX_ITERATIONS);
    }


}
```

```
import org.example.data.Message;
import org.example.data.Packet;
import org.example.network.Channel;
import org.example.network.Routable;
import org.example.network.Router;
import org.example.network.address.SimpleAddress;
import org.example.protocol.ClassicTCP;
import org.example.protocol.MPTCP;
import org.example.protocol.TCP;
import org.example.simulator.EventHandler;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.simulator.statistics.TCPStats;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class Simulator {

    private int numPacketsToSend = 2000;

    @Before
    public void setUp() {
        Util.resetTime();
        Util.setSeed(1);
    }

    private void numPacketsToSend(TCP tcpToSend, int numPacketsToSend) {
        for (int i = 1; i <= numPacketsToSend; i++) {
            Message msg = new Message("test " + i);
            tcpToSend.send(msg);
        }
    }

    private EventHandler connectAndRun(TCP client, TCP server) {
        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();
        return eventHandler;
    }

    private void allReceived(TCP receiver, int numPacketsToSend) {
        for (int i = 1; i <= numPacketsToSend; i++) {
            Message msg = new Message("test " + i);
            Packet received = receiver.receive();
            Assert.assertNotNull(received);
            Assert.assertEquals(msg, received.getPayload());
        }
    }

    @Test
    public void shortPathLowLoss() {
        ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
            "Client-ShortPathLowLoss")).build();
        Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
            withAddress(new SimpleAddress("Router 1")).build();
        Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
            withAddress(new SimpleAddress("Router 2")).build();
        Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
            withAddress(new SimpleAddress("Router 3")).build();
        Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
```

```
            withAddress(new SimpleAddress("Router 4")).build();
58          ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Server-ShortPathLowLoss")).build();
59
60          new Channel.ChannelBuilder().withCost(1).build(client, r1);
61          new Channel.ChannelBuilder().withCost(10).build(r1, r2);
62          new Channel.ChannelBuilder().withCost(10).withLoss(0.001).build(r2, r3);
63          new Channel.ChannelBuilder().withCost(10).build(r3, r4);
64          new Channel.ChannelBuilder().withCost(1).build(r4, server);
65
66          client.updateRoutingTable();
67          r1.updateRoutingTable();
68          r2.updateRoutingTable();
69          r3.updateRoutingTable();
70          r4.updateRoutingTable();
71          server.updateRoutingTable();
72
73          this.numPacketsToSend(server, this.numPacketsToSend);
74          EventHandler eventHandler = this.connectAndRun(client, server);
75
76          Assert.assertTrue(client.isConnected());
77          Assert.assertTrue(server.isConnected());
78          Assert.assertTrue(client.inputBufferIsEmpty());
79          Assert.assertTrue(server.inputBufferIsEmpty());
80          Assert.assertTrue(client.outputBufferIsEmpty());
81          Assert.assertTrue(server.outputBufferIsEmpty());
82
83          this.allReceived(client, this.numPacketsToSend);
84
85          TCPStats stat = client.getStats();
86          System.out.println(stat);
87          stat.createArrivalChart();
88          stat.createDepartureChart();
89          stat.createInterArrivalChart();
90          stat.createTimeInSystemChart();
91          stat.createNumberOfPacketsInSystemChart();
92
93          System.out.println(server.getStats().toString());
94          server.getStats().createCWNDChart();
95          server.getStats().createDepartureChart();
96      }
97
98      @Test
99      public void shortPathHighLoss() {
100         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Client-ShortPathHighLoss")).build();
101         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
102         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
103         Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
104         Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
105         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Server-ShortPathHighLoss")).build();
106
107
108         new Channel.ChannelBuilder().withCost(1).build(client, r1);
109         new Channel.ChannelBuilder().withCost(10).build(r1, r2);
```

```
110         new Channel.ChannelBuilder().withCost(10).withLoss(0.01).build(r2, r3);
111         new Channel.ChannelBuilder().withCost(10).build(r3, r4);
112         new Channel.ChannelBuilder().withCost(1).build(r4, server);
113
114         client.updateRoutingTable();
115         r1.updateRoutingTable();
116         r2.updateRoutingTable();
117         r3.updateRoutingTable();
118         r4.updateRoutingTable();
119         server.updateRoutingTable();
120
121         this.numPacketsToSend(server, this.numPacketsToSend);
122         EventHandler eventHandler = this.connectAndRun(client, server);
123
124         Assert.assertTrue(client.isConnected());
125         Assert.assertTrue(server.isConnected());
126         Assert.assertTrue(client.inputBufferIsEmpty());
127         Assert.assertTrue(server.inputBufferIsEmpty());
128         Assert.assertTrue(client.outputBufferIsEmpty());
129         Assert.assertTrue(server.outputBufferIsEmpty());
130
131         this.allReceived(client, this.numPacketsToSend);
132
133         TCPStats stat = client.getStats();
134         System.out.println(stat);
135         stat.createArrivalChart();
136         stat.createDepartureChart();
137         stat.createInterArrivalChart();
138         stat.createTimeInSystemChart();
139         stat.createNumberOfPacketsInSystemChart();
140
141         System.out.println(server.getStats().toString());
142         server.getStats().createCWNDChart();
143         server.getStats().createDepartureChart();
144     }
145
146     @Test
147     public void longPathLowLoss() {
148         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Client-LongPathLowLoss")).build();
149         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
150         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
151         Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
152         Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 4")).build();
153         Routable r5 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 5")).build();
154         Routable r6 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 6")).build();
155         Routable r7 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 7")).build();
156         Routable r8 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 8")).build();
157         Routable r9 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 9")).build();
158         Routable r10 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 10")).build();
159         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
```

```
            "Server-LongPathLowLoss")).build();
160
161         new Channel.ChannelBuilder().withCost(1).build(client, r1);
162         new Channel.ChannelBuilder().withCost(10).build(r1, r2);
163         new Channel.ChannelBuilder().withCost(50).build(r2, r3);
164         new Channel.ChannelBuilder().withCost(50).build(r3, r4);
165         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r4, r5);
166         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r5, r6);
167         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r6, r7);
168         new Channel.ChannelBuilder().withCost(50).build(r7, r8);
169         new Channel.ChannelBuilder().withCost(50).build(r8, r9);
170         new Channel.ChannelBuilder().withCost(10).build(r9, r10);
171         new Channel.ChannelBuilder().withCost(1).build(r10, server);
172
173         client.updateRoutingTable();
174         r1.updateRoutingTable();
175         r2.updateRoutingTable();
176         r3.updateRoutingTable();
177         r4.updateRoutingTable();
178         r5.updateRoutingTable();
179         r6.updateRoutingTable();
180         r7.updateRoutingTable();
181         r8.updateRoutingTable();
182         r9.updateRoutingTable();
183         r10.updateRoutingTable();
184         server.updateRoutingTable();
185
186         this.numPacketsToSend(server, this.numPacketsToSend);
187         EventHandler eventHandler = this.connectAndRun(client, server);
188
189         Assert.assertTrue(client.isConnected());
190         Assert.assertTrue(server.isConnected());
191         Assert.assertTrue(client.inputBufferIsEmpty());
192         Assert.assertTrue(server.inputBufferIsEmpty());
193         Assert.assertTrue(client.outputBufferIsEmpty());
194         Assert.assertTrue(server.outputBufferIsEmpty());
195
196         this.allReceived(client, this.numPacketsToSend);
197
198         TCPStats stat = client.getStats();
199         System.out.println(stat);
200         stat.createArrivalChart();
201         stat.createDepartureChart();
202         stat.createInterArrivalChart();
203         stat.createTimeInSystemChart();
204         stat.createNumberOfPacketsInSystemChart();
205
206         System.out.println(server.getStats().toString());
207         server.getStats().createCWNDChart();
208     }
209
210     @Test
211     public void longPathHighLoss() {
212         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Client-LongPathHighLoss")).build();
213         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
214         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
215         Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
216         Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
```

```
            withAddress(new SimpleAddress("Router 4")).build();
217         Routable r5 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 5")).build();
218         Routable r6 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 6")).build();
219         Routable r7 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 7")).build();
220         Routable r8 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 8")).build();
221         Routable r9 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 9")).build();
222         Routable r10 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 10")).build();
223         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
                "Server-LongPathHighLoss")).build();
224
225         new Channel.ChannelBuilder().withCost(1).build(client, r1);
226         new Channel.ChannelBuilder().withCost(10).build(r1, r2);
227         new Channel.ChannelBuilder().withCost(50).build(r2, r3);
228         new Channel.ChannelBuilder().withCost(50).build(r3, r4);
229         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r4, r5);
230         new Channel.ChannelBuilder().withCost(50).withLoss(0.01).build(r5, r6);
231         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r6, r7);
232         new Channel.ChannelBuilder().withCost(50).build(r7, r8);
233         new Channel.ChannelBuilder().withCost(50).build(r8, r9);
234         new Channel.ChannelBuilder().withCost(10).build(r9, r10);
235         new Channel.ChannelBuilder().withCost(1).build(r10, server);
236
237         client.updateRoutingTable();
238         r1.updateRoutingTable();
239         r2.updateRoutingTable();
240         r3.updateRoutingTable();
241         r4.updateRoutingTable();
242         r5.updateRoutingTable();
243         r6.updateRoutingTable();
244         r7.updateRoutingTable();
245         r8.updateRoutingTable();
246         r9.updateRoutingTable();
247         r10.updateRoutingTable();
248         server.updateRoutingTable();
249
250         this.numPacketsToSend(server, this.numPacketsToSend);
251         EventHandler eventHandler = this.connectAndRun(client, server);
252
253         Assert.assertTrue(client.isConnected());
254         Assert.assertTrue(server.isConnected());
255         Assert.assertTrue(client.inputBufferIsEmpty());
256         Assert.assertTrue(server.inputBufferIsEmpty());
257         Assert.assertTrue(client.outputBufferIsEmpty());
258         Assert.assertTrue(server.outputBufferIsEmpty());
259
260         this.allReceived(client, this.numPacketsToSend);
261
262         TCPStats stat = client.getStats();
263         System.out.println(stat);
264         stat.createArrivalChart();
265         stat.createDepartureChart();
266         stat.createInterArrivalChart();
267         stat.createTimeInSystemChart();
268         stat.createNumberOfPacketsInSystemChart();
269
270         System.out.println(server.getStats().toString());
```

```
271         server.getStats().createCWNDChart();
272     }
273
274
275     private void getStats(MPTCP sender, MPTCP receiver) {
276         //receiver
277         for (TCPStats stat : receiver.getTcpStats()) {
278             System.out.println(stat.toString());
279             stat.createArrivalChart();
280             stat.createDepartureChart();
281             stat.createInterArrivalChart();
282             stat.createTimeInSystemChart();
283             stat.createNumberOfPacketsInSystemChart();
284         }
285
286         //sender
287         for (TCPStats stat : sender.getTcpStats()) {
288             System.out.println(stat.toString());
289             stat.createCWNDChart();
290         }
291     }
292
293     @Test
294     public void MPTCP_HomoShortPathLowLoss() {
295         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-ShortPathLowLoss")).build();
296
297         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
298         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
299         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
300         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
301
302         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
303         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
304         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
305         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
306
307         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-ShortPathLowLoss")).build();
308
309         //path one
310         new Channel.ChannelBuilder().withCost(1).build(client, r11);
311         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
312         new Channel.ChannelBuilder().withCost(10).withLoss(0.001).build(r12, r13);
313         new Channel.ChannelBuilder().withCost(10).build(r13, r14);
314         new Channel.ChannelBuilder().withCost(1).build(r14, server);
315
316         //path two
317         new Channel.ChannelBuilder().withCost(1).build(client, r21);
318         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
319         new Channel.ChannelBuilder().withCost(10).withLoss(0.001).build(r22, r23);
320         new Channel.ChannelBuilder().withCost(10).build(r23, r24);
321         new Channel.ChannelBuilder().withCost(1).build(r24, server);
```

```
322
323         client.updateRoutingTable();
324         r11.updateRoutingTable();
325         r12.updateRoutingTable();
326         r13.updateRoutingTable();
327         r14.updateRoutingTable();
328         r21.updateRoutingTable();
329         r22.updateRoutingTable();
330         r23.updateRoutingTable();
331         r24.updateRoutingTable();
332         server.updateRoutingTable();
333
334         this.numPacketsToSend(server, this.numPacketsToSend);
335         EventHandler eventHandler = this.connectAndRun(server, client);
336
337         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
338         Assert.assertTrue(server.outputBufferIsEmpty());
339         Assert.assertTrue(client.inputBufferIsEmpty());
340         Assert.assertTrue(server.inputBufferIsEmpty());
341         Assert.assertTrue(r11.inputBufferIsEmpty());
342         Assert.assertTrue(r12.inputBufferIsEmpty());
343         Assert.assertTrue(r13.inputBufferIsEmpty());
344         Assert.assertTrue(r14.inputBufferIsEmpty());
345         Assert.assertTrue(r21.inputBufferIsEmpty());
346         Assert.assertTrue(r22.inputBufferIsEmpty());
347         Assert.assertTrue(r23.inputBufferIsEmpty());
348         Assert.assertTrue(r24.inputBufferIsEmpty());
349
350         this.allReceived(client, this.numPacketsToSend);
351
352         Assert.assertNull(server.receive());
353         this.getStats(server, client);
354
355     }
356
357     @Test
358     public void MPTCP_HomoShortPathHighLoss() {
359         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-ShortPathHighLoss")).build();
360
361         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
362         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
363         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
364         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
365
366         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
367         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
368         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
369         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
370
371         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-ShortPathHighLoss")).build();
```

```
372
373         //path one
374         new Channel.ChannelBuilder().withCost(1).build(client, r11);
375         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
376         new Channel.ChannelBuilder().withCost(10).withLoss(0.01).build(r12, r13);
377         new Channel.ChannelBuilder().withCost(10).build(r13, r14);
378         new Channel.ChannelBuilder().withCost(1).build(r14, server);
379
380         //path two
381         new Channel.ChannelBuilder().withCost(1).build(client, r21);
382         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
383         new Channel.ChannelBuilder().withCost(10).withLoss(0.01).build(r22, r23);
384         new Channel.ChannelBuilder().withCost(10).build(r23, r24);
385         new Channel.ChannelBuilder().withCost(1).build(r24, server);
386
387         client.updateRoutingTable();
388         r11.updateRoutingTable();
389         r12.updateRoutingTable();
390         r13.updateRoutingTable();
391         r14.updateRoutingTable();
392         r21.updateRoutingTable();
393         r22.updateRoutingTable();
394         r23.updateRoutingTable();
395         r24.updateRoutingTable();
396         server.updateRoutingTable();
397
398         this.numPacketsToSend(server, this.numPacketsToSend);
399         EventHandler eventHandler = this.connectAndRun(server, client);
400
401         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
402         Assert.assertTrue(server.outputBufferIsEmpty());
403         Assert.assertTrue(client.inputBufferIsEmpty());
404         Assert.assertTrue(server.inputBufferIsEmpty());
405         Assert.assertTrue(r11.inputBufferIsEmpty());
406         Assert.assertTrue(r12.inputBufferIsEmpty());
407         Assert.assertTrue(r13.inputBufferIsEmpty());
408         Assert.assertTrue(r14.inputBufferIsEmpty());
409         Assert.assertTrue(r21.inputBufferIsEmpty());
410         Assert.assertTrue(r22.inputBufferIsEmpty());
411         Assert.assertTrue(r23.inputBufferIsEmpty());
412         Assert.assertTrue(r24.inputBufferIsEmpty());
413
414         this.allReceived(client, this.numPacketsToSend);
415
416         Assert.assertNull(server.receive());
417         this.getStats(server, client);
418
419     }
420
421     @Test
422     public void MPTCP_HomoLongPathLowLoss() {
423         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-LongPathLowLoss")).build();
424
425         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 11")).build();
426         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 12")).build();
427         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 13")).build();
428         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
```

```
            withAddress(new SimpleAddress("Router 14")).build();
429         Routable r15 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 15")).build();
430         Routable r16 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 16")).build();
431         Routable r17 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 17")).build();
432         Routable r18 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 18")).build();
433         Routable r19 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 19")).build();
434         Routable r110 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 110")).build();
435
436         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 21")).build();
437         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 22")).build();
438         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 23")).build();
439         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 24")).build();
440         Routable r25 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 25")).build();
441         Routable r26 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 26")).build();
442         Routable r27 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 27")).build();
443         Routable r28 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 28")).build();
444         Routable r29 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 29")).build();
445         Routable r210 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 210")).build();
446
447         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-LongPathLowLoss")).build();
448
449         //path one
450         new Channel.ChannelBuilder().withCost(1).build(client, r11);
451         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
452         new Channel.ChannelBuilder().withCost(50).build(r12, r13);
453         new Channel.ChannelBuilder().withCost(50).build(r13, r14);
454         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r14, r15);
455         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r15, r16);
456         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r16, r17);
457         new Channel.ChannelBuilder().withCost(50).build(r17, r18);
458         new Channel.ChannelBuilder().withCost(50).build(r18, r19);
459         new Channel.ChannelBuilder().withCost(10).build(r19, r110);
460         new Channel.ChannelBuilder().withCost(1).build(r110, server);
461
462         //path two
463         new Channel.ChannelBuilder().withCost(1).build(client, r21);
464         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
465         new Channel.ChannelBuilder().withCost(50).build(r22, r23);
466         new Channel.ChannelBuilder().withCost(50).build(r23, r24);
467         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r24, r25);
468         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r25, r26);
469         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r26, r27);
470         new Channel.ChannelBuilder().withCost(50).build(r27, r28);
471         new Channel.ChannelBuilder().withCost(50).build(r28, r29);
472         new Channel.ChannelBuilder().withCost(10).build(r29, r210);
```

```
473         new Channel.ChannelBuilder().withCost(1).build(r210, server);
474
475         client.updateRoutingTable();
476
477         r11.updateRoutingTable();
478         r12.updateRoutingTable();
479         r13.updateRoutingTable();
480         r14.updateRoutingTable();
481         r15.updateRoutingTable();
482         r16.updateRoutingTable();
483         r17.updateRoutingTable();
484         r18.updateRoutingTable();
485         r19.updateRoutingTable();
486         r110.updateRoutingTable();
487
488         r21.updateRoutingTable();
489         r22.updateRoutingTable();
490         r23.updateRoutingTable();
491         r24.updateRoutingTable();
492         r25.updateRoutingTable();
493         r26.updateRoutingTable();
494         r27.updateRoutingTable();
495         r28.updateRoutingTable();
496         r29.updateRoutingTable();
497         r210.updateRoutingTable();
498
499         server.updateRoutingTable();
500
501         this.numPacketsToSend(server, this.numPacketsToSend);
502         EventHandler eventHandler = this.connectAndRun(server, client);
503
504         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
505         Assert.assertTrue(server.outputBufferIsEmpty());
506         Assert.assertTrue(client.inputBufferIsEmpty());
507         Assert.assertTrue(server.inputBufferIsEmpty());
508         Assert.assertTrue(r11.inputBufferIsEmpty());
509         Assert.assertTrue(r12.inputBufferIsEmpty());
510         Assert.assertTrue(r13.inputBufferIsEmpty());
511         Assert.assertTrue(r14.inputBufferIsEmpty());
512         Assert.assertTrue(r15.inputBufferIsEmpty());
513         Assert.assertTrue(r16.inputBufferIsEmpty());
514         Assert.assertTrue(r17.inputBufferIsEmpty());
515         Assert.assertTrue(r18.inputBufferIsEmpty());
516         Assert.assertTrue(r19.inputBufferIsEmpty());
517         Assert.assertTrue(r110.inputBufferIsEmpty());
518
519         Assert.assertTrue(r21.inputBufferIsEmpty());
520         Assert.assertTrue(r22.inputBufferIsEmpty());
521         Assert.assertTrue(r23.inputBufferIsEmpty());
522         Assert.assertTrue(r24.inputBufferIsEmpty());
523         Assert.assertTrue(r25.inputBufferIsEmpty());
524         Assert.assertTrue(r26.inputBufferIsEmpty());
525         Assert.assertTrue(r27.inputBufferIsEmpty());
526         Assert.assertTrue(r28.inputBufferIsEmpty());
527         Assert.assertTrue(r29.inputBufferIsEmpty());
528         Assert.assertTrue(r210.inputBufferIsEmpty());
529
530         this.allReceived(client, this.numPacketsToSend);
531
532         Assert.assertNull(server.receive());
533         this.getStats(server, client);
534
```

```
535     }
536
537     @Test
538     public void MPTCP_HomoLongPathHighLoss() {
539         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-LongPathHighLoss")).build();
540
541         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 11")).build();
542         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 12")).build();
543         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 13")).build();
544         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 14")).build();
545         Routable r15 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 15")).build();
546         Routable r16 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 16")).build();
547         Routable r17 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 17")).build();
548         Routable r18 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 18")).build();
549         Routable r19 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 19")).build();
550         Routable r110 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 110")).build();
551
552         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 21")).build();
553         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 22")).build();
554         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 23")).build();
555         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 24")).build();
556         Routable r25 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 25")).build();
557         Routable r26 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 26")).build();
558         Routable r27 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 27")).build();
559         Routable r28 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 28")).build();
560         Routable r29 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 29")).build();
561         Routable r210 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 210")).build();
562
563         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-LongPathHighLoss")).build();
564
565         //path one
566         new Channel.ChannelBuilder().withCost(1).build(client, r11);
567         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
568         new Channel.ChannelBuilder().withCost(50).build(r12, r13);
569         new Channel.ChannelBuilder().withCost(50).build(r13, r14);
570         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r14, r15);
571         new Channel.ChannelBuilder().withCost(50).withLoss(0.01).build(r15, r16);
572         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r16, r17);
573         new Channel.ChannelBuilder().withCost(50).build(r17, r18);
```

```
574         new Channel.ChannelBuilder().withCost(50).build(r18, r19);
575         new Channel.ChannelBuilder().withCost(10).build(r19, r110);
576         new Channel.ChannelBuilder().withCost(1).build(r110, server);
577 
578         //path two
579         new Channel.ChannelBuilder().withCost(1).build(client, r21);
580         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
581         new Channel.ChannelBuilder().withCost(50).build(r22, r23);
582         new Channel.ChannelBuilder().withCost(50).build(r23, r24);
583         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r24, r25);
584         new Channel.ChannelBuilder().withCost(50).withLoss(0.01).build(r25, r26);
585         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r26, r27);
586         new Channel.ChannelBuilder().withCost(50).build(r27, r28);
587         new Channel.ChannelBuilder().withCost(50).build(r28, r29);
588         new Channel.ChannelBuilder().withCost(10).build(r29, r210);
589         new Channel.ChannelBuilder().withCost(1).build(r210, server);
590 
591         client.updateRoutingTable();
592 
593         r11.updateRoutingTable();
594         r12.updateRoutingTable();
595         r13.updateRoutingTable();
596         r14.updateRoutingTable();
597         r15.updateRoutingTable();
598         r16.updateRoutingTable();
599         r17.updateRoutingTable();
600         r18.updateRoutingTable();
601         r19.updateRoutingTable();
602         r110.updateRoutingTable();
603 
604         r21.updateRoutingTable();
605         r22.updateRoutingTable();
606         r23.updateRoutingTable();
607         r24.updateRoutingTable();
608         r25.updateRoutingTable();
609         r26.updateRoutingTable();
610         r27.updateRoutingTable();
611         r28.updateRoutingTable();
612         r29.updateRoutingTable();
613         r210.updateRoutingTable();
614 
615         server.updateRoutingTable();
616 
617         this.numPacketsToSend(server, this.numPacketsToSend);
618         EventHandler eventHandler = this.connectAndRun(server, client);
619 
620         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
621         Assert.assertTrue(server.outputBufferIsEmpty());
622         Assert.assertTrue(client.inputBufferIsEmpty());
623         Assert.assertTrue(server.inputBufferIsEmpty());
624         Assert.assertTrue(r11.inputBufferIsEmpty());
625         Assert.assertTrue(r12.inputBufferIsEmpty());
626         Assert.assertTrue(r13.inputBufferIsEmpty());
627         Assert.assertTrue(r14.inputBufferIsEmpty());
628         Assert.assertTrue(r15.inputBufferIsEmpty());
629         Assert.assertTrue(r16.inputBufferIsEmpty());
630         Assert.assertTrue(r17.inputBufferIsEmpty());
631         Assert.assertTrue(r18.inputBufferIsEmpty());
632         Assert.assertTrue(r19.inputBufferIsEmpty());
633         Assert.assertTrue(r110.inputBufferIsEmpty());
634 
635         Assert.assertTrue(r21.inputBufferIsEmpty());
```

```
636         Assert.assertTrue(r22.inputBufferIsEmpty());
637         Assert.assertTrue(r23.inputBufferIsEmpty());
638         Assert.assertTrue(r24.inputBufferIsEmpty());
639         Assert.assertTrue(r25.inputBufferIsEmpty());
640         Assert.assertTrue(r26.inputBufferIsEmpty());
641         Assert.assertTrue(r27.inputBufferIsEmpty());
642         Assert.assertTrue(r28.inputBufferIsEmpty());
643         Assert.assertTrue(r29.inputBufferIsEmpty());
644         Assert.assertTrue(r210.inputBufferIsEmpty());
645 
646         this.allReceived(client, this.numPacketsToSend);
647 
648         Assert.assertNull(server.receive());
649         this.getStats(server, client);
650 
651     }
652 
653     @Test
654     public void MPTCP_HeteroLowLoss() {
655         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-HeteroLowLoss")).build();
656 
657         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 11")).build();
658         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 12")).build();
659         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 13")).build();
660         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 14")).build();
661         Routable r15 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 15")).build();
662         Routable r16 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 16")).build();
663         Routable r17 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 17")).build();
664         Routable r18 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 18")).build();
665         Routable r19 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 19")).build();
666         Routable r110 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 110")).build();
667 
668         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
669         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
670         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
671         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
672 
673         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-HeteroLowLoss")).build();
674 
675         // long path
676         new Channel.ChannelBuilder().withCost(1).build(client, r11);
677         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
678         new Channel.ChannelBuilder().withCost(50).build(r12, r13);
679         new Channel.ChannelBuilder().withCost(50).build(r13, r14);
680         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r14, r15);
```

```
681         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r15, r16);
682         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r16, r17);
683         new Channel.ChannelBuilder().withCost(50).build(r17, r18);
684         new Channel.ChannelBuilder().withCost(50).build(r18, r19);
685         new Channel.ChannelBuilder().withCost(10).build(r19, r110);
686         new Channel.ChannelBuilder().withCost(1).build(r110, server);
687 
688         // short path
689         new Channel.ChannelBuilder().withCost(1).build(client, r21);
690         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
691         new Channel.ChannelBuilder().withCost(10).withLoss(0.001).build(r22, r23);
692         new Channel.ChannelBuilder().withCost(10).build(r23, r24);
693         new Channel.ChannelBuilder().withCost(1).build(r24, server);
694 
695         client.updateRoutingTable();
696 
697         r11.updateRoutingTable();
698         r12.updateRoutingTable();
699         r13.updateRoutingTable();
700         r14.updateRoutingTable();
701         r15.updateRoutingTable();
702         r16.updateRoutingTable();
703         r17.updateRoutingTable();
704         r18.updateRoutingTable();
705         r19.updateRoutingTable();
706         r110.updateRoutingTable();
707 
708         r21.updateRoutingTable();
709         r22.updateRoutingTable();
710         r23.updateRoutingTable();
711         r24.updateRoutingTable();
712 
713         server.updateRoutingTable();
714 
715         this.numPacketsToSend(server, this.numPacketsToSend);
716         EventHandler eventHandler = this.connectAndRun(server, client);
717 
718 
719         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
720         Assert.assertTrue(server.outputBufferIsEmpty());
721         Assert.assertTrue(client.inputBufferIsEmpty());
722         Assert.assertTrue(server.inputBufferIsEmpty());
723         Assert.assertTrue(r11.inputBufferIsEmpty());
724         Assert.assertTrue(r12.inputBufferIsEmpty());
725         Assert.assertTrue(r13.inputBufferIsEmpty());
726         Assert.assertTrue(r14.inputBufferIsEmpty());
727         Assert.assertTrue(r15.inputBufferIsEmpty());
728         Assert.assertTrue(r16.inputBufferIsEmpty());
729         Assert.assertTrue(r17.inputBufferIsEmpty());
730         Assert.assertTrue(r18.inputBufferIsEmpty());
731         Assert.assertTrue(r19.inputBufferIsEmpty());
732         Assert.assertTrue(r110.inputBufferIsEmpty());
733 
734         Assert.assertTrue(r21.inputBufferIsEmpty());
735         Assert.assertTrue(r22.inputBufferIsEmpty());
736         Assert.assertTrue(r23.inputBufferIsEmpty());
737         Assert.assertTrue(r24.inputBufferIsEmpty());
738 
739         this.allReceived(client, this.numPacketsToSend);
740 
741         Assert.assertNull(server.receive());
742         this.getStats(server, client);
```

```
743
744     }
745
746     @Test
747     public void MPTCP_HeteroHighLoss() {
748         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Client-HeteroHighLoss")).build();
749
750         Routable r11 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 11")).build();
751         Routable r12 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 12")).build();
752         Routable r13 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 13")).build();
753         Routable r14 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 14")).build();
754         Routable r15 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 15")).build();
755         Routable r16 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 16")).build();
756         Routable r17 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 17")).build();
757         Routable r18 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 18")).build();
758         Routable r19 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 19")).build();
759         Routable r110 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 110")).build();
760
761         Routable r21 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 1")).build();
762         Routable r22 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 2")).build();
763         Routable r23 = new Router.RouterBuilder().withAverageQueueUtilization(0.98).
                withAddress(new SimpleAddress("Router 3")).build();
764         Routable r24 = new Router.RouterBuilder().withAverageQueueUtilization(0.5).
                withAddress(new SimpleAddress("Router 4")).build();
765
766         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(30).withAddress(new SimpleAddress("MPTCP-
                Server-HeteroHighLoss")).build();
767
768         // long path
769         new Channel.ChannelBuilder().withCost(1).build(client, r11);
770         new Channel.ChannelBuilder().withCost(10).build(r11, r12);
771         new Channel.ChannelBuilder().withCost(50).build(r12, r13);
772         new Channel.ChannelBuilder().withCost(50).build(r13, r14);
773         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r14, r15);
774         new Channel.ChannelBuilder().withCost(50).withLoss(0.01).build(r15, r16);
775         new Channel.ChannelBuilder().withCost(50).withLoss(0.001).build(r16, r17);
776         new Channel.ChannelBuilder().withCost(50).build(r17, r18);
777         new Channel.ChannelBuilder().withCost(50).build(r18, r19);
778         new Channel.ChannelBuilder().withCost(10).build(r19, r110);
779         new Channel.ChannelBuilder().withCost(1).build(r110, server);
780
781         // short path
782         new Channel.ChannelBuilder().withCost(1).build(client, r21);
783         new Channel.ChannelBuilder().withCost(10).build(r21, r22);
784         new Channel.ChannelBuilder().withCost(10).withLoss(0.01).build(r22, r23);
785         new Channel.ChannelBuilder().withCost(10).build(r23, r24);
786         new Channel.ChannelBuilder().withCost(1).build(r24, server);
787
```

```
788         client.updateRoutingTable();
789
790         r11.updateRoutingTable();
791         r12.updateRoutingTable();
792         r13.updateRoutingTable();
793         r14.updateRoutingTable();
794         r15.updateRoutingTable();
795         r16.updateRoutingTable();
796         r17.updateRoutingTable();
797         r18.updateRoutingTable();
798         r19.updateRoutingTable();
799         r110.updateRoutingTable();
800
801         r21.updateRoutingTable();
802         r22.updateRoutingTable();
803         r23.updateRoutingTable();
804         r24.updateRoutingTable();
805
806         server.updateRoutingTable();
807
808         this.numPacketsToSend(server, this.numPacketsToSend);
809         EventHandler eventHandler = this.connectAndRun(server, client);
810
811
812         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
813         Assert.assertTrue(server.outputBufferIsEmpty());
814         Assert.assertTrue(client.inputBufferIsEmpty());
815         Assert.assertTrue(server.inputBufferIsEmpty());
816         Assert.assertTrue(r11.inputBufferIsEmpty());
817         Assert.assertTrue(r12.inputBufferIsEmpty());
818         Assert.assertTrue(r13.inputBufferIsEmpty());
819         Assert.assertTrue(r14.inputBufferIsEmpty());
820         Assert.assertTrue(r15.inputBufferIsEmpty());
821         Assert.assertTrue(r16.inputBufferIsEmpty());
822         Assert.assertTrue(r17.inputBufferIsEmpty());
823         Assert.assertTrue(r18.inputBufferIsEmpty());
824         Assert.assertTrue(r19.inputBufferIsEmpty());
825         Assert.assertTrue(r110.inputBufferIsEmpty());
826
827         Assert.assertTrue(r21.inputBufferIsEmpty());
828         Assert.assertTrue(r22.inputBufferIsEmpty());
829         Assert.assertTrue(r23.inputBufferIsEmpty());
830         Assert.assertTrue(r24.inputBufferIsEmpty());
831
832         this.allReceived(client, this.numPacketsToSend);
833
834         Assert.assertNull(server.receive());
835         this.getStats(server, client);
836
837     }
838
839 }
```

```java
package org.example.data;

import org.junit.Assert;
import org.junit.Test;

public class MessageTest {

    @Test
    public void createMessageWorksTest() {
        Message message = new Message("test");
        Assert.assertTrue(message instanceof Message);
    }

    @Test
    public void stringInMessageIsCorrectTest() {
        String s1 = "test";
        Message message1 = new Message(s1);
        Assert.assertEquals(s1, message1.toString());

        String s2 = "test12345";
        Message message2 = new Message(s2);
        Assert.assertEquals(s2, message2.toString());
    }

    @Test
    public void sizeOfMessageIsCorrectTest() {
        String s1 = "test";
        Message message1 = new Message(s1);
        Assert.assertEquals(s1.length(), message1.size());

        String s2 = "test12345";
        Message message2 = new Message(s2);
        Assert.assertEquals(s2.length(), message2.size());
    }


    @Test
    public void messagesWithEqualStringsAreEqualTest() {
        String s = "test";
        Message message1 = new Message(s);
        Message message2 = new Message(s);
        Assert.assertEquals(message1, message2);
    }
}
```

```java
package org.example.data;

import org.example.network.interfaces.Endpoint;
import org.example.protocol.ClassicTCP;
import org.junit.Assert;
import org.junit.Test;

import java.util.List;

public class PacketTest {

    @Test
    public void buildPacketBuildsPacketTest() {
        Assert.assertTrue(new PacketBuilder().build() instanceof Packet);
    }

    @Test
    public void buildPacketWithDestinationBuildsPacketWithDestinationTest() {
        Endpoint endpoint = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();
        Packet packet = new PacketBuilder().withDestination(endpoint).build();
        Assert.assertEquals(endpoint, packet.getDestination());
    }

    @Test
    public void buildPacketWithOriginBuildsPacketWithOriginTest() {
        Endpoint endpoint = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();
        Packet packet = new PacketBuilder().withOrigin(endpoint).build();
        Assert.assertEquals(endpoint, packet.getOrigin());
    }

    @Test
    public void buildPacketWithMsgBuildsPacketMsgOriginTest() {
        Message msg = new Message("Test");
        Packet packet = new PacketBuilder().withPayload(msg).build();
        Assert.assertEquals(msg, packet.getPayload());
    }

    @Test
    public void buildPacketWithSeqNumBuildsPacketSeqNumOriginTest() {
        int seqNum = 123;
        Packet packet = new PacketBuilder().withSequenceNumber(seqNum).build();
        Assert.assertEquals(seqNum, packet.getSequenceNumber());
    }

    @Test
    public void buildPacketWithAckNumBuildsPacketAckNumOriginTest() {
        int ackNum = 321;
        Packet packet = new PacketBuilder().withAcknowledgmentNumber(ackNum).
            withFlags(Flag.ACK).build();
        Assert.assertEquals(ackNum, packet.getAcknowledgmentNumber());
    }

    @Test
    public void buildPacketWithFlagBuildsPacketThatHasFlagTest() {
        Flag flag = Flag.ACK;
        Packet packet = new PacketBuilder().withFlags(flag).build();
        Assert.assertTrue(packet.hasAllFlags(flag));
    }

    @Test
```

```
60      public void buildPacketWithFlagsBuildsPacketThatHasFlagsTest() {
61          Flag ack = Flag.ACK;
62          Flag syn = Flag.SYN;
63          Flag fin = Flag.FIN;
64          Packet packet = new PacketBuilder().withFlags(ack, syn, fin).build();
65          Assert.assertTrue(packet.hasAllFlags(syn, fin));
66          Assert.assertTrue(packet.hasAllFlags(ack));
67      }
68
69      @Test
70      public void buildPacketWithoutFlagBuildsPacketThatDoesNotHaveThatFlagTest() {
71          Flag ack = Flag.ACK;
72          Flag syn = Flag.SYN;
73          Flag fin = Flag.FIN;
74          Packet packet = new PacketBuilder().withFlags(syn, fin).build();
75          Assert.assertFalse(packet.hasAllFlags(syn, fin, ack));
76          Assert.assertFalse(packet.hasAllFlags(ack));
77          Assert.assertFalse(packet.hasAllFlags(syn, ack));
78          Assert.assertTrue(packet.hasAllFlags(syn, fin));
79      }
80
81      @Test
82      public void buildPacketWithDuplicateFlagsBuildsPacketThatHasOneOfThatFlagTest()
            {
83          Flag ack = Flag.ACK;
84          Flag syn = Flag.SYN;
85          Flag fin = Flag.FIN;
86          Packet packet = new PacketBuilder().withFlags(ack, syn, syn, fin, fin).build
                ();
87          List<Flag> flags = packet.getFlags();
88          Assert.assertEquals(3, flags.size());
89          Assert.assertTrue(packet.hasAllFlags(syn, fin, ack));
90      }
91
92      @Test
93      public void buildPacketWithPayloadHasCorrectSizeTest() {
94          Message message = new Message("test");
95          Packet packet = new PacketBuilder()
96                  .withPayload(message)
97                  .build();
98          Assert.assertEquals(message.size(), packet.size());
99      }
100
101     @Test
102     public void buildPacketWithoutPayloadHasZeroSizeTest() {
103         Packet packet = new PacketBuilder()
104                 .build();
105         Assert.assertEquals(0, packet.size());
106     }
107
108 }
```

```java
package org.example.network;

import org.example.network.address.Address;
import org.example.network.address.UUIDAddress;
import org.junit.Assert;
import org.junit.Test;

public class AddressTest {

    @Test
    public void createAddressWorksTest() {
        Address address = new UUIDAddress();
        Assert.assertTrue(address instanceof Address);
    }

    @Test
    public void twoUUIDAddressesAreNotEqual() {
        Address address1 = new UUIDAddress();
        Address address2 = new UUIDAddress();
        Assert.assertNotEquals(address1, address2);
    }

    @Test
    public void nullAddressesAreNotEqualToAddress() {
        Address address1 = new UUIDAddress();
        Address address2 = null;
        Assert.assertNotEquals(address1, address2);
    }

}
```

```java
package org.example.network;

import org.example.data.*;
import org.example.network.interfaces.Endpoint;
import org.example.protocol.ClassicTCP;
import org.junit.Assert;
import org.junit.Test;

import java.util.PriorityQueue;

public class ChannelTest {

    private Endpoint source = new ClassicTCP.ClassicTCPBuilder().
        withReceivingWindowCapacity(7).build();
    private Endpoint destination = new ClassicTCP.ClassicTCPBuilder().
        withReceivingWindowCapacity(7).build();

    @Test
    public void channelConstructor1Test() {
        Channel channel = new Channel(source, destination, 100);
        Assert.assertTrue(channel instanceof Channel);
        Assert.assertEquals(source, channel.getSource());
        Assert.assertEquals(destination, channel.getDestination());
    }

    @Test
    public void channelConstructor2Test() {
        Channel channel = new Channel(source);
        Assert.assertTrue(channel instanceof Channel);
        Assert.assertEquals(source, channel.getSource());
        Assert.assertEquals(source, channel.getDestination());
        Assert.assertEquals(0, channel.getCost());
    }

    @Test
    public void channelWithConstructor1CostIsPositiveTest() {
        for (int i = 0; i < 10000; i++) {
            Channel channel = new Channel(source, destination, 100);
            Assert.assertTrue(channel.getCost() >= 0);
        }
    }

    @Test
    public void channelWithConstructor2CostIsZeroTest() {
        for (int i = 0; i < 10000; i++) {
            Channel channel = new Channel(source);
            Assert.assertEquals(0, channel.getCost());
        }
    }

    @Test
    public void channelPacketDeliversPacketToDestinationNodeTest() {
        Channel channel = new Channel(source, destination, 0);
        Payload payload = new Message("Test");
        Packet packet = new PacketBuilder()
                .withOrigin(source)
                .withDestination(destination)
                .withPayload(payload)
                .withFlags(Flag.SYN) // hack to overcome connection check in the
                    endpoints
                .build();
        channel.channelPackage(packet);
```

```
60          Assert.assertTrue(channel.channel());
61          Packet receivedPacket = destination.dequeueInputBuffer();
62          Assert.assertEquals(packet, receivedPacket);
63      }
64
65
66      @Test
67      public void lossyChannelDropPacketTest() {
68          Channel channel = new Channel(source, destination, 0);
69          Payload payload = new Message("Test");
70          Packet packet = new PacketBuilder()
71                  .withOrigin(source)
72                  .withDestination(destination)
73                  .withPayload(payload)
74                  .withFlags(Flag.SYN) // hack to overcome connection check in the
                        endpoints
75                  .build();
76          channel.channelPackage(packet);
77          Packet receivedPacket = destination.dequeueInputBuffer();
78          Assert.assertEquals(null, receivedPacket);
79      }
80
81
82      @Test
83      public void aLittleLossyChannelWillDropPacketAfterAWhileTest() {
84          for (int i = 0; i < 10000; i++) {
85              Channel channel = new Channel(source, destination, 3);
86              Payload payload = new Message("Test");
87              Packet packet = new PacketBuilder()
88                      .withOrigin(source)
89                      .withDestination(destination)
90                      .withPayload(payload)
91                      .withFlags(Flag.SYN) // hack to overcome connection check in the
                            endpoints
92                      .build();
93              channel.channelPackage(packet);
94              Packet receivedPacket = destination.dequeueInputBuffer();
95              if (receivedPacket == null) {
96                  Assert.assertTrue(true);
97                  return;
98              }
99          }
100         Assert.assertFalse(true);
101     }
102
103     @Test
104     public void channelCompareToWorksInPriorityQueueTest() {
105         int size = 1000;
106         PriorityQueue<Channel> pq = new PriorityQueue<>(size);
107         for (int i = 0; i < size; i++) {
108             Channel c = new Channel(source, destination, 0);
109             pq.add(c);
110         }
111         while (pq.size() > 1) {
112             Assert.assertTrue(pq.poll().getCost() <= pq.peek().getCost());
113         }
114     }
115
116     @Test
117     public void differentChannelsAreNotEqualTest() {
118         Endpoint source1 = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
119         Endpoint destination1 = new ClassicTCP.ClassicTCPBuilder().
```

```
            withReceivingWindowCapacity(7).build();

        Endpoint source2 = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();
        Endpoint destination2 = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();

        Channel channel1 = new Channel(source1, destination1, 100);
        Channel channel2 = new Channel(source2, destination2, 100);

        Assert.assertNotEquals(channel1, channel2);
    }


}
```

```java
package org.example.network;

import org.example.data.*;
import org.example.network.interfaces.Endpoint;
import org.example.network.interfaces.NetworkNode;
import org.example.simulator.EventHandler;
import org.example.simulator.events.RouteEvent;
import org.junit.Assert;
import org.junit.Test;

import java.util.concurrent.ArrayBlockingQueue;

public class RoutableTest {

    @Test
    public void routableWithRandomAddressAreNotEqualTest() {
        NetworkNode node1 = new Router.RouterBuilder().build();
        NetworkNode node2 = new Router.RouterBuilder().build();
        Assert.assertNotEquals(node1, node2);
    }


    @Test
    public void routingPacketRoutsItToItsDestinationStraitLine() {
        Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
            ArrayBlockingQueue<>(100));
        Router r1 = new Router.RouterBuilder().build();
        Router r2 = new Router.RouterBuilder().build();
        Router r3 = new Router.RouterBuilder().build();
        Router r4 = new Router.RouterBuilder().build();
        Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
            ArrayBlockingQueue<>(100));

        new Channel.ChannelBuilder().build(client, r1);
        new Channel.ChannelBuilder().build(r1, r2);
        new Channel.ChannelBuilder().build(r2, r3);
        new Channel.ChannelBuilder().build(r3, r4);
        new Channel.ChannelBuilder().build(r4, server);

        client.updateRoutingTable();
        r1.updateRoutingTable();
        r2.updateRoutingTable();
        r3.updateRoutingTable();
        r4.updateRoutingTable();
        server.updateRoutingTable();

        Message msg = new Message("Test");
        Packet packet = new PacketBuilder()
                .withPayload(msg)
                .withDestination(server)
                .build();

        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new RouteEvent(client, packet));
        eventHandler.run();

        Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
        Payload receivedPayload = receivedPacket.getPayload();

        Assert.assertEquals(receivedPayload, msg);
    }

```

```
61
62      @Test
63      public void routingPacketRoutsItToItsDestinationCircleGraph() {
64          Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
65          Router r1 = new Router.RouterBuilder().build();
66          Router r2 = new Router.RouterBuilder().build();
67          Router r3 = new Router.RouterBuilder().build();
68          Router r4 = new Router.RouterBuilder().build();
69          Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
70
71          new Channel.ChannelBuilder().build(client, r1);
72          new Channel.ChannelBuilder().build(client, r2);
73          new Channel.ChannelBuilder().build(r1, r3);
74          new Channel.ChannelBuilder().build(r2, r4);
75          new Channel.ChannelBuilder().build(server, r3);
76          new Channel.ChannelBuilder().build(server, r4);
77
78          client.updateRoutingTable();
79          r1.updateRoutingTable();
80          r2.updateRoutingTable();
81          r3.updateRoutingTable();
82          r4.updateRoutingTable();
83          server.updateRoutingTable();
84
85
86          Message msg = new Message("Test");
87          Packet packet = new PacketBuilder()
88                  .withPayload(msg)
89                  .withDestination(server)
90                  .withFlags(Flag.SYN) // hack to overcome connection check in the
                        endpoints
91                  .build();
92
93          EventHandler eventHandler = new EventHandler();
94          eventHandler.addEvent(new RouteEvent(client, packet));
95          eventHandler.run();
96
97          Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
98          Payload receivedPayload = receivedPacket.getPayload();
99
100         Assert.assertEquals(receivedPayload, msg);
101     }
102
103
104     @Test
105     public void routingPacketRoutsItToItsDestinationWithCycle() {
106         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
107         Router r1 = new Router.RouterBuilder().build();
108         Router r2 = new Router.RouterBuilder().build();
109         Router r3 = new Router.RouterBuilder().build();
110         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
111
112         new Channel.ChannelBuilder().build(client, r1);
113         new Channel.ChannelBuilder().build(r1, r2);
114         new Channel.ChannelBuilder().build(r1, r3);
115         new Channel.ChannelBuilder().build(r2, r3);
116         new Channel.ChannelBuilder().build(server, r3);
117
118         client.updateRoutingTable();
```

```
119         server.updateRoutingTable();
120         r1.updateRoutingTable();
121         r2.updateRoutingTable();
122         r3.updateRoutingTable();
123
124
125         Message msg = new Message("Test");
126         Packet packet = new PacketBuilder()
127                 .withPayload(msg)
128                 .withDestination(server)
129                 .build();
130
131         EventHandler eventHandler = new EventHandler();
132         eventHandler.addEvent(new RouteEvent(client, packet));
133         eventHandler.run();
134
135         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
136         Payload receivedPayload = receivedPacket.getPayload();
137
138         Assert.assertEquals(receivedPayload, msg);
139     }
140
141     @Test
142     public void routingPacketRoutsItToItsDestinationWithDeadEnd() {
143         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
144         Router r1 = new Router.RouterBuilder().build();
145         Router r2 = new Router.RouterBuilder().build();
146         Router r3 = new Router.RouterBuilder().build();
147         Router r4 = new Router.RouterBuilder().build();
148         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
149
150         new Channel.ChannelBuilder().build(client, r1);
151         new Channel.ChannelBuilder().build(r1, r2);
152         new Channel.ChannelBuilder().build(r1, r4);
153         new Channel.ChannelBuilder().build(r1, r3);
154         new Channel.ChannelBuilder().build(server, r4);
155
156         client.updateRoutingTable();
157         r1.updateRoutingTable();
158         r2.updateRoutingTable();
159         r3.updateRoutingTable();
160         r4.updateRoutingTable();
161         server.updateRoutingTable();
162
163         Message msg = new Message("Test");
164         Packet packet = new PacketBuilder()
165                 .withPayload(msg)
166                 .withDestination(server)
167                 .withFlags(Flag.SYN) // hack to overcome connection check in the
                        endpoints
168                 .build();
169
170         EventHandler eventHandler = new EventHandler();
171         eventHandler.addEvent(new RouteEvent(client, packet));
172         eventHandler.run();
173
174         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
175         Payload receivedPayload = receivedPacket.getPayload();
176
177         Assert.assertEquals(msg, receivedPayload);
178     }
```

```
179
180
181     @Test
182     public void routingPacketRoutsItToItsDestinationForrest() {
183         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
184         Router r1 = new Router.RouterBuilder().build();
185         Router r2 = new Router.RouterBuilder().build();
186         Router r3 = new Router.RouterBuilder().build();
187         Router r4 = new Router.RouterBuilder().build();
188         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
189
190         //tree one
191         new Channel.ChannelBuilder().build(client, r1);
192         new Channel.ChannelBuilder().build(r1, server);
193
194         //tree two
195         new Channel.ChannelBuilder().build(r2, r3);
196         new Channel.ChannelBuilder().build(r3, r4);
197
198         client.updateRoutingTable();
199         r1.updateRoutingTable();
200         r2.updateRoutingTable();
201         r3.updateRoutingTable();
202         r4.updateRoutingTable();
203         server.updateRoutingTable();
204
205
206         Message msg = new Message("Test");
207         Packet packet = new PacketBuilder()
208                 .withPayload(msg)
209                 .withDestination(server)
210                 .build();
211
212         EventHandler eventHandler = new EventHandler();
213         eventHandler.addEvent(new RouteEvent(client, packet));
214         eventHandler.run();
215
216         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
217         Payload receivedPayload = receivedPacket.getPayload();
218
219         Assert.assertEquals(msg, receivedPayload);
220     }
221
222
223     @Test
224     public void routingPacketRoutsItToItsDestinationWithUnconnectedNode() {
225         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
226         Router r1 = new Router.RouterBuilder().build();
227         Router r2 = new Router.RouterBuilder().build();
228         Router r3 = new Router.RouterBuilder().build();
229         Router r4 = new Router.RouterBuilder().build();
230         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
231
232         new Channel.ChannelBuilder().build(client, r1);
233         new Channel.ChannelBuilder().build(r1, r2);
234         new Channel.ChannelBuilder().build(r2, r3);
235         new Channel.ChannelBuilder().build(server, r3);
236
237         client.updateRoutingTable();
```

```
238         r1.updateRoutingTable();
239         r2.updateRoutingTable();
240         r3.updateRoutingTable();
241         r4.updateRoutingTable();
242         server.updateRoutingTable();
243 
244         Message msg = new Message("Test");
245         Packet packet = new PacketBuilder()
246                 .withPayload(msg)
247                 .withDestination(server)
248                 .build();
249 
250         EventHandler eventHandler = new EventHandler();
251         eventHandler.addEvent(new RouteEvent(client, packet));
252         eventHandler.run();
253 
254         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
255         Payload receivedPayload = receivedPacket.getPayload();
256 
257         Assert.assertEquals(msg, receivedPayload);
258     }
259 
260 
261     @Test
262     public void routingPacketRoutsItToItsDestinationCrazyGraph() {
263         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
264         Router r1 = new Router.RouterBuilder().build();
265         Router r2 = new Router.RouterBuilder().build();
266         Router r3 = new Router.RouterBuilder().build();
267         Router r4 = new Router.RouterBuilder().build();
268         Router r5 = new Router.RouterBuilder().build();
269         Router r6 = new Router.RouterBuilder().build();
270         Router r7 = new Router.RouterBuilder().build();
271         Router r8 = new Router.RouterBuilder().build();
272         Router r9 = new Router.RouterBuilder().build();
273         Router r10 = new Router.RouterBuilder().build();
274         Router r11 = new Router.RouterBuilder().build();
275         Router r12 = new Router.RouterBuilder().build();
276         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
277 
278         new Channel.ChannelBuilder().build(client, r1);
279         new Channel.ChannelBuilder().build(client, r2);
280         new Channel.ChannelBuilder().build(r1, r3);
281         new Channel.ChannelBuilder().build(r2, r3);
282         new Channel.ChannelBuilder().build(r3, r4);
283         new Channel.ChannelBuilder().build(r3, r9);
284         new Channel.ChannelBuilder().build(r4, r5);
285         new Channel.ChannelBuilder().build(r4, r6);
286         new Channel.ChannelBuilder().build(r5, r6);
287         new Channel.ChannelBuilder().build(r6, r9);
288         new Channel.ChannelBuilder().build(r6, r7);
289         new Channel.ChannelBuilder().build(r7, r8);
290         new Channel.ChannelBuilder().build(r9, r10);
291         new Channel.ChannelBuilder().build(r10, r11);
292         new Channel.ChannelBuilder().build(r11, r12);
293         new Channel.ChannelBuilder().build(r12, server);
294 
295         client.updateRoutingTable();
296         r1.updateRoutingTable();
297         r2.updateRoutingTable();
298         r3.updateRoutingTable();
```

```
299         r4.updateRoutingTable();
300         r5.updateRoutingTable();
301         r6.updateRoutingTable();
302         r7.updateRoutingTable();
303         r8.updateRoutingTable();
304         r9.updateRoutingTable();
305         r10.updateRoutingTable();
306         r11.updateRoutingTable();
307         r12.updateRoutingTable();
308         server.updateRoutingTable();
309 
310         Message msg = new Message("Test");
311         Packet packet = new PacketBuilder()
312                 .withPayload(msg)
313                 .withDestination(server)
314                 .build();
315 
316         EventHandler eventHandler = new EventHandler();
317         eventHandler.addEvent(new RouteEvent(client, packet));
318         eventHandler.run();
319 
320         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
321         Payload receivedPayload = receivedPacket.getPayload();
322 
323         Assert.assertEquals(msg, receivedPayload);
324     }
325 
326 
327     @Test(expected = IllegalArgumentException.class)
328     public void unconnectedClientCantRoutPacketToDestination() {
329         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
330         Router r1 = new Router.RouterBuilder().build();
331         Router r2 = new Router.RouterBuilder().build();
332         Router r3 = new Router.RouterBuilder().build();
333         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
334 
335         new Channel.ChannelBuilder().build(r1, r2);
336         new Channel.ChannelBuilder().build(r2, r3);
337         new Channel.ChannelBuilder().build(server, r3);
338 
339         client.updateRoutingTable();
340         r1.updateRoutingTable();
341         r2.updateRoutingTable();
342         r3.updateRoutingTable();
343         server.updateRoutingTable();
344 
345         Message msg = new Message("Test");
346         client.route(new PacketBuilder()
347                 .withPayload(msg)
348                 .withDestination(server)
349                 .build());
350     }
351 
352 
353     @Test(expected = IllegalArgumentException.class)
354     public void unconnectedTreesCantRoutPacket() {
355         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
356         Router r1 = new Router.RouterBuilder().build();
357         Router r2 = new Router.RouterBuilder().build();
358         Router r3 = new Router.RouterBuilder().build();
```

```
359         Router r4 = new Router.RouterBuilder().build();
360         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
361
362         new Channel.ChannelBuilder().build(client, r1);
363         new Channel.ChannelBuilder().build(r1, r2);
364
365         new Channel.ChannelBuilder().build(r3, r4);
366         new Channel.ChannelBuilder().build(server, r4);
367
368         client.updateRoutingTable();
369         r1.updateRoutingTable();
370         r2.updateRoutingTable();
371         r3.updateRoutingTable();
372         server.updateRoutingTable();
373
374         Message msg = new Message("Test");
375         client.route(new PacketBuilder()
376                 .withPayload(msg)
377                 .withDestination(server)
378                 .build());
379     }
380
381
382     @Test
383     public void faultyChannelsDropPacket() {
384         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
385         Router r1 = new Router.RouterBuilder().build();
386         Router r2 = new Router.RouterBuilder().build();
387         Router r3 = new Router.RouterBuilder().build();
388         Router r4 = new Router.RouterBuilder().build();
389         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(100), new
                ArrayBlockingQueue<>(100));
390
391         new Channel.ChannelBuilder().build(client, r1);
392         new Channel.ChannelBuilder().withLoss(100).build(r1, r2);
393         new Channel.ChannelBuilder().build(r2, r3);
394         new Channel.ChannelBuilder().build(r3, r4);
395         new Channel.ChannelBuilder().build(server, r4);
396
397         client.updateRoutingTable();
398         r1.updateRoutingTable();
399         r2.updateRoutingTable();
400         r3.updateRoutingTable();
401         r4.updateRoutingTable();
402         server.updateRoutingTable();
403
404
405         Message msg = new Message("Test");
406         Packet packet = new PacketBuilder()
407                 .withPayload(msg)
408                 .withDestination(server)
409                 .build();
410
411         EventHandler eventHandler = new EventHandler();
412         eventHandler.addEvent(new RouteEvent(client, packet));
413         eventHandler.run();
414
415         Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
416
417         Assert.assertNull(receivedPacket);
418     }
```

```
419
420
421     @Test
422     public void not100PercentLossyRoutersAreLoosingPacketIfEnoughPacketsAreSent() {
423         double noiseTolerance = 2.5;
424         Endpoint client = new RoutableEndpoint(new ArrayBlockingQueue<>(1000), new
                ArrayBlockingQueue<>(1000));
425         Router r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.1).
                build();
426         Router r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.1).
                build();
427         Router r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.1).
                build();
428         Router r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.1).
                build();
429         Endpoint server = new RoutableEndpoint(new ArrayBlockingQueue<>(1000), new
                ArrayBlockingQueue<>(1000));
430
431         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(client, r1);
432         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(r1, r2);
433         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(r2, r3);
434         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(r3, r4);
435         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(server, r4);
436
437         client.updateRoutingTable();
438         r1.updateRoutingTable();
439         r2.updateRoutingTable();
440         r3.updateRoutingTable();
441         r4.updateRoutingTable();
442         server.updateRoutingTable();
443
444
445         EventHandler eventHandler = new EventHandler();
446
447         for (int i = 0; i < 1000; i++) {
448             Message msg = new Message(i + "");
449             Packet packet = new PacketBuilder()
450                     .withPayload(msg)
451                     .withDestination(server)
452                     .build();
453             eventHandler.addEvent(new RouteEvent(client, packet));
454         }
455
456         eventHandler.run();
457
458         for (int i = 0; i < 1000; i++) {
459             Packet receivedPacket = ((RoutableEndpoint) server).getReceivedPacket();
460             if (receivedPacket == null) {
461                 return;
462             }
463         }
464         Assert.fail();
465
466     }
467
468
469 }
```

```java
package org.example.network;

import org.example.data.PacketBuilder;
import org.example.network.interfaces.Endpoint;
import org.example.network.interfaces.NetworkNode;
import org.example.protocol.ClassicTCP;
import org.junit.Assert;
import org.junit.Test;

public class RoutingTableTest {

    @Test(expected = IllegalStateException.class)
    public void routingTableTrowsIllegalStateExceptionIfNotUpdated() {
        Endpoint r1 = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();
        NetworkNode r2 = new Router.RouterBuilder().build();
        NetworkNode r3 = new Router.RouterBuilder().build();
        Endpoint r4 = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();

        new Channel.ChannelBuilder().build(r1, r2);
        new Channel.ChannelBuilder().build(r2, r3);
        new Channel.ChannelBuilder().build(r3, r4);

        r1.route(new PacketBuilder().withDestination(r4).build());
    }


    @Test(expected = IllegalArgumentException.class)
    public void routingTableTrowsIllegalArgumentExceptionIfDestinationIsNull() {
        NetworkNode r1 = new Router.RouterBuilder().build();
        NetworkNode r2 = new Router.RouterBuilder().build();
        NetworkNode r3 = new Router.RouterBuilder().build();
        NetworkNode r4 = new Router.RouterBuilder().build();

        new Channel.ChannelBuilder().build(r1, r2);
        new Channel.ChannelBuilder().build(r2, r3);
        new Channel.ChannelBuilder().build(r3, r4);

        r1.updateRoutingTable();
        r2.updateRoutingTable();
        r3.updateRoutingTable();
        r4.updateRoutingTable();

        r1.route(new PacketBuilder().withDestination(null).build());
    }

    @Test(expected = IllegalArgumentException.class)
    public void routingTableTrowsIllegalArgumentExceptionIfDestinationIsUnconnected
        () {
        Endpoint r1 = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();
        NetworkNode r2 = new Router.RouterBuilder().build();
        NetworkNode r3 = new Router.RouterBuilder().build();
        Endpoint r4 = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();

        new Channel.ChannelBuilder().build(r1, r2);
        new Channel.ChannelBuilder().build(r2, r3);

        r1.updateRoutingTable();
        r2.updateRoutingTable();
```

```
58          r3.updateRoutingTable();
59          r4.updateRoutingTable();
60
61          r1.route(new PacketBuilder().withDestination(r4).build());
62      }
63
64
65      private Channel getChannelFromTo(NetworkNode from, NetworkNode destination) {
66          for (Channel c : from.getChannels()) {
67              if (c.getDestination().equals(destination)) return c;
68          }
69          Assert.assertFalse(true);
70          return null;
71      }
72
73
74      @Test
75      public void getPathChoosesShortestPathTest() {
76          for (int i = 0; i < 100; i++) {
77              Endpoint r1 = new ClassicTCP.ClassicTCPBuilder().
                    withReceivingWindowCapacity(7).build();
78              NetworkNode r2 = new Router.RouterBuilder().build();
79              NetworkNode r3 = new Router.RouterBuilder().build();
80              Endpoint r4 = new ClassicTCP.ClassicTCPBuilder().
                    withReceivingWindowCapacity(7).build();
81
82              new Channel.ChannelBuilder().build(r1, r2);
83              new Channel.ChannelBuilder().build(r1, r3);
84              new Channel.ChannelBuilder().build(r2, r4);
85              new Channel.ChannelBuilder().build(r3, r4);
86
87              r1.updateRoutingTable();
88              r2.updateRoutingTable();
89              r3.updateRoutingTable();
90              r4.updateRoutingTable();
91
92              Channel channelOnePathOne = getChannelFromTo(r1, r2);
93              Channel channelTwoPathOne = getChannelFromTo(r2, r4);
94
95              Channel channelOnePathTwo = getChannelFromTo(r1, r3);
96              Channel channelTwoPathTwo = getChannelFromTo(r3, r4);
97
98              int pathOne = channelOnePathOne.getCost() + channelTwoPathOne.getCost();
99              int pathTwo = channelOnePathTwo.getCost() + channelTwoPathTwo.getCost();
100             if (pathOne == pathTwo) continue;
101
102             Channel usedChannel = pathOne < pathTwo ? channelOnePathOne :
                    channelOnePathTwo;
103             Channel notUsedChannel = pathOne > pathTwo ? channelOnePathOne :
                    channelOnePathTwo;
104
105             r1.route(new PacketBuilder().withOrigin(r1).withDestination(r4).build())
                    ;
106
107             Assert.assertTrue(usedChannel.channel());
108             Assert.assertFalse(notUsedChannel.channel());
109
110             Assert.assertTrue(!usedChannel.getDestination().inputBufferIsEmpty());
111             Assert.assertNull(notUsedChannel.getDestination().dequeueInputBuffer());
112         }
113     }
114
115
```

```
116     @Test
117     public void toStringTest() {
118         RoutingTable routingTable = new RoutingTable();
119         routingTable.update(new Router.RouterBuilder().build());
120         routingTable.toString();
121         Assert.assertTrue(routingTable.toString() instanceof String);
122     }
123
124     @Test(expected = IllegalArgumentException.class)
125     public void getPathWhenNoPathIsThereTest() {
126         RoutingTable routingTable = new RoutingTable();
127         routingTable.update(new Router.RouterBuilder().build());
128         routingTable.getPath(new Router.RouterBuilder().build(), new Router.
                RouterBuilder().build());
129     }
130
131 }
```

```java
package org.example.protocol;

import org.example.data.Message;
import org.example.data.Packet;
import org.example.data.PacketBuilder;
import org.example.network.Channel;
import org.example.network.Routable;
import org.example.network.Router;
import org.example.protocol.window.receiving.ReceivingWindow;
import org.example.simulator.EventHandler;
import org.example.simulator.events.RouteEvent;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import java.util.concurrent.TimeUnit;

public class ClassicTCPTest {

    @Rule
    public Timeout globalTimeout = new Timeout(60, TimeUnit.SECONDS);


    @Before
    public void setup() {
        Util.setSeed(1996);
        Util.resetTime();
    }

    @Test
    public void connectToEndpointTest() {
        ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().build();
        Routable router = new Router.RouterBuilder().build();
        ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().build();

        new Channel.ChannelBuilder().build(client, router);
        new Channel.ChannelBuilder().build(router, server);

        client.updateRoutingTable();
        router.updateRoutingTable();
        server.updateRoutingTable();

        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();

        Assert.assertEquals(server, client.getConnection().getConnectedNode());
        Assert.assertEquals(client, server.getConnection().getConnectedNode());

        Assert.assertEquals(server.getConnection().getNextSequenceNumber(), client.
            getConnection().getNextAcknowledgementNumber());
    }

    @Test
    public void
        connectToEndpointShouldResultInCorrectReceivingAndSendingWindowCapacityTest
        () {
```

```
60          int clientReceivingWindow = 3;
61          int serverReceivingWindow = 7;
62          ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(clientReceivingWindow).build();
63          Routable router = new Router.RouterBuilder().build();
64          ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(serverReceivingWindow).build();
65
66          new Channel.ChannelBuilder().build(client, router);
67          new Channel.ChannelBuilder().build(router, server);
68
69          client.updateRoutingTable();
70          router.updateRoutingTable();
71          server.updateRoutingTable();
72
73          EventHandler eventHandler = new EventHandler();
74          eventHandler.addEvent(new TCPConnectEvent(client, server));
75          eventHandler.run();
76
77          Assert.assertEquals(server, client.getConnection().getConnectedNode());
78          Assert.assertEquals(client, server.getConnection().getConnectedNode());
79
80          Assert.assertEquals(server.getConnection().getNextSequenceNumber(), client.
                getConnection().getNextAcknowledgementNumber());
81
82          Assert.assertEquals(clientReceivingWindow, client.
                getThisReceivingWindowCapacity());
83          Assert.assertEquals(serverReceivingWindow, server.
                getThisReceivingWindowCapacity());
84
85          Assert.assertEquals(serverReceivingWindow, client.
                getOtherReceivingWindowCapacity());
86          Assert.assertEquals(clientReceivingWindow, server.
                getOtherReceivingWindowCapacity());
87      }
88
89      @Test
90      public void connectThenSendMsgWorksTest() {
91          ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
92          Routable router = new Router.RouterBuilder().build();
93          ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
94
95          new Channel.ChannelBuilder().build(client, router);
96          new Channel.ChannelBuilder().build(router, server);
97
98          client.updateRoutingTable();
99          router.updateRoutingTable();
100         server.updateRoutingTable();
101
102         EventHandler eventHandler = new EventHandler();
103         eventHandler.addEvent(new TCPConnectEvent(client, server));
104         eventHandler.run();
105
106         Message msg = new Message("Test");
107
108         client.send(msg);
109         eventHandler.addEvent(new RunTCPEvent(client));
110         eventHandler.run();
111
112         Assert.assertEquals(msg, server.receive().getPayload());
113
```

```
114     }
115
116
117     @Test
118     public void connectThenSendMsgOverMultipleNodesLineWorksTest() {
119         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
120         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
121         Router r1 = new Router.RouterBuilder().build();
122         Router r2 = new Router.RouterBuilder().build();
123         Router r3 = new Router.RouterBuilder().build();
124         Router r4 = new Router.RouterBuilder().build();
125
126         new Channel.ChannelBuilder().build(client, r1);
127         new Channel.ChannelBuilder().build(r1, r2);
128         new Channel.ChannelBuilder().build(r2, r3);
129         new Channel.ChannelBuilder().build(r3, r4);
130         new Channel.ChannelBuilder().build(r4, server);
131
132         client.updateRoutingTable();
133         r1.updateRoutingTable();
134         r2.updateRoutingTable();
135         r3.updateRoutingTable();
136         r4.updateRoutingTable();
137         server.updateRoutingTable();
138
139         EventHandler eventHandler = new EventHandler();
140         eventHandler.addEvent(new TCPConnectEvent(client, server));
141         eventHandler.run();
142
143         Message msg = new Message("Test");
144
145         client.send(msg);
146         eventHandler.addEvent(new RunTCPEvent(client));
147         eventHandler.run();
148
149         Assert.assertEquals(msg, server.receive().getPayload());
150
151     }
152
153     @Test
154     public void unorderedPacketsAreNotReceivedTest() {
155         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
156         Routable router = new Router.RouterBuilder().build();
157         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
158
159         new Channel.ChannelBuilder().build(client, router);
160         new Channel.ChannelBuilder().build(router, server);
161
162         client.updateRoutingTable();
163         router.updateRoutingTable();
164         server.updateRoutingTable();
165
166         EventHandler eventHandler = new EventHandler();
167         eventHandler.addEvent(new TCPConnectEvent(client, server));
168         eventHandler.run();
169
170         System.out.println("connected");
171
172         Message msg = new Message("test1");
```

```
173         client.send(msg);
174         eventHandler.addEvent(new RunTCPEvent(client));
175
176         Message msg2 = new Message("test2");
177         Packet packet = new PacketBuilder()
178                 .withPayload(msg2)
179                 .withOrigin(client)
180                 .withDestination(server)
181                 .withSequenceNumber(client.getConnection().getNextSequenceNumber() +
                        100)
182                 .build();
183
184         eventHandler.addEvent(new RouteEvent(client, packet));
185
186         eventHandler.run();
187
188         Assert.assertEquals(msg, server.receive().getPayload());
189         Assert.assertNull(server.receive());
190     }
191
192
193     @Test
194     public void
            unorderedPacketsAreDroppedAndOrderedPacketsAreReceivedWithoutBlockTest() {
195         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
196         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
197         Router r1 = new Router.RouterBuilder().build();
198         Router r2 = new Router.RouterBuilder().build();
199         Router r3 = new Router.RouterBuilder().build();
200         Router r4 = new Router.RouterBuilder().build();
201
202         new Channel.ChannelBuilder().build(client, r1);
203         new Channel.ChannelBuilder().build(r1, r2);
204         new Channel.ChannelBuilder().build(r2, r3);
205         new Channel.ChannelBuilder().build(r3, r4);
206         new Channel.ChannelBuilder().build(r4, server);
207
208         client.updateRoutingTable();
209         r1.updateRoutingTable();
210         r2.updateRoutingTable();
211         r3.updateRoutingTable();
212         r4.updateRoutingTable();
213         server.updateRoutingTable();
214
215         EventHandler eventHandler = new EventHandler();
216         eventHandler.addEvent(new TCPConnectEvent(client, server));
217         eventHandler.run();
218
219
220         for (int i = 0; i < server.getThisReceivingWindowCapacity(); i++) {
221             Packet packet1 = new PacketBuilder()
222                     .withSequenceNumber(client.getConnection().getNextSequenceNumber
                            () + 5000 + i)
223                     .withAcknowledgmentNumber(client.getConnection().
                            getNextAcknowledgementNumber() + 20000 + i)
224                     .withOrigin(client)
225                     .withDestination(server)
226                     .withPayload(new Message("should not be received 1"))
227                     .withIndex(i + 50000)
228                     .build();
229             eventHandler.addEvent(new RouteEvent(client, packet1));
```

```
230
231             Message msg = new Message("test " + i);
232             client.send(msg);
233             eventHandler.addEvent(new RunTCPEvent(client));
234
235             Packet packet2 = new PacketBuilder()
236                     .withSequenceNumber(client.getConnection().getNextSequenceNumber
                            () + 100 + i)
237                     .withAcknowledgmentNumber(client.getConnection().
                            getNextAcknowledgementNumber() + 20 + i)
238                     .withOrigin(client)
239                     .withDestination(server)
240                     .withPayload(new Message("should not be received 2"))
241                     .withIndex(i + 100)
242                     .build();
243             eventHandler.addEvent(new RouteEvent(client, packet2));
244         }
245         eventHandler.run();
246
247         for (int i = 0; i < server.getThisReceivingWindowCapacity(); i++) {
248             Packet received = server.receive();
249             Assert.assertNotNull(received);
250             Assert.assertEquals("test " + i, received.getPayload().toString());
251         }
252
253         Packet received = server.receive();
254         Assert.assertNull(received);
255     }
256
257
258     @Test
259     public void routedMessagesUnorderedReceiveOrderedTest() throws
            IllegalAccessException {
260         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
261         Routable router = new Router.RouterBuilder().build();
262         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
263
264         new Channel.ChannelBuilder().build(client, router);
265         new Channel.ChannelBuilder().build(router, server);
266
267         client.updateRoutingTable();
268         router.updateRoutingTable();
269         server.updateRoutingTable();
270
271         EventHandler eventHandler = new EventHandler();
272         eventHandler.addEvent(new TCPConnectEvent(client, server));
273         eventHandler.run();
274
275         System.out.println("connected");
276
277         int seqNum = client.getConnection().getNextSequenceNumber();
278         int ackNum = client.getConnection().getNextAcknowledgementNumber();
279
280         //increase window capacity to max
281         for (int i = 0; i < client.getThisReceivingWindowCapacity(); i++) {
282             client.getSendingWindow().increase();
283         }
284
285         for (int i = client.getThisReceivingWindowCapacity() - 1; i >= 0; i--) {
286             Packet packet = new PacketBuilder()
287                     .withOrigin(client)
```

```
                    .withDestination(server)
                    .withSequenceNumber(seqNum + i)
                    .withAcknowledgmentNumber(ackNum + i)
                    .withPayload(new Message(i + ""))
                    .withIndex(i)
                    .build();
            eventHandler.addEvent(new RouteEvent(client, packet));
        }
        eventHandler.run();

        for (int i = 0; i < client.getThisReceivingWindowCapacity(); i++) {
            Packet received = server.receive();
            Assert.assertNotNull("iteration: " + i, received);
            Assert.assertEquals(i + "", received.getPayload().toString());
        }
        Packet received = server.receive();
        Assert.assertNull(received);
    }


    @Test
    public void routeToManyMessagesUnorderedReceiveOrderedAndDropCorrectTest() {
        ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();
        Routable router = new Router.RouterBuilder().build();
        ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(7).build();

        new Channel.ChannelBuilder().build(client, router);
        new Channel.ChannelBuilder().build(router, server);

        client.updateRoutingTable();
        router.updateRoutingTable();
        server.updateRoutingTable();

        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();

        System.out.println("connected");

        int seqNum = client.getConnection().getNextSequenceNumber();
        int ackNum = client.getConnection().getNextAcknowledgementNumber();

        for (int i = client.getThisReceivingWindowCapacity() * 2; i >= 0; i--) {
            eventHandler.addEvent(new RouteEvent(client, new PacketBuilder()
                    .withOrigin(client)
                    .withDestination(server)
                    .withSequenceNumber(seqNum + i)
                    .withAcknowledgmentNumber(ackNum + i)
                    .withPayload(new Message(i + ""))
                    .withIndex(i)
                    .build()
            ));
            eventHandler.run();
        }

        for (int i = 0; i < client.getThisReceivingWindowCapacity(); i++) {
            Packet received = server.receive();
            Assert.assertNotNull(received);
            Assert.assertEquals(i + "", received.getPayload().toString());
        }
        Packet received = server.receive();
```

```
349         Assert.assertNull(received);
350     }
351
352     @Test
353     public void packetIndexShouldUpdateAfterReceivingPacketInOrderTest() throws
            IllegalAccessException {
354         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
355         Routable router = new Router.RouterBuilder().build();
356         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
357
358         new Channel.ChannelBuilder().build(client, router);
359         new Channel.ChannelBuilder().build(router, server);
360
361         client.updateRoutingTable();
362         router.updateRoutingTable();
363         server.updateRoutingTable();
364
365         EventHandler eventHandler = new EventHandler();
366         eventHandler.addEvent(new TCPConnectEvent(client, server));
367         eventHandler.run();
368
369         System.out.println("connected");
370
371         Packet packet = new PacketBuilder()
372                 .withConnection(client.getConnection())
373                 .withSequenceNumber(client.getConnection().getNextSequenceNumber())
374                 .build();
375
376
377         int indexBeforeSending = server.getReceivingWindow().receivingPacketIndex(
                packet, server.getConnection());
378         Assert.assertEquals(0, indexBeforeSending);
379
380         client.send(packet.getPayload());
381         eventHandler.addEvent(new RunTCPEvent(client));
382         eventHandler.run();
383         Assert.assertNotNull(server.receive());
384
385         int indexAfterReceived = server.getReceivingWindow().receivingPacketIndex(
                packet, server.getConnection());
386         Assert.assertEquals(-1, indexAfterReceived);
387
388     }
389
390     @Test
391     public void
            packetIndexShouldNotUpdateAfterReceivingPacketOutOfOrderButInWindowTest()
            throws IllegalAccessException {
392         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
393         Routable router = new Router.RouterBuilder().build();
394         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
395
396         new Channel.ChannelBuilder().build(client, router);
397         new Channel.ChannelBuilder().build(router, server);
398
399         client.updateRoutingTable();
400         router.updateRoutingTable();
401         server.updateRoutingTable();
402
```

```
403
404         EventHandler eventHandler = new EventHandler();
405         eventHandler.addEvent(new TCPConnectEvent(client, server));
406         eventHandler.run();
407
408         System.out.println("connected");
409
410         //this test does not make sens for window size = 1
411         if (client.getThisReceivingWindowCapacity() <= 1) return;
412
413         int seqNum = client.getConnection().getNextSequenceNumber();
414         int ackNum = client.getConnection().getNextAcknowledgementNumber();
415
416
417         Packet packet = new PacketBuilder()
418                 .withConnection(client.getConnection())
419                 .withSequenceNumber(seqNum + client.getThisReceivingWindowCapacity()
                        - 1)
420                 .withAcknowledgmentNumber(ackNum + client.
                        getThisReceivingWindowCapacity() - 1)
421                 .build();
422
423         int indexBeforeSending = server.getReceivingWindow().receivingPacketIndex(
                packet, server.getConnection());
424         Assert.assertEquals(client.getThisReceivingWindowCapacity() - 1,
                indexBeforeSending);
425
426
427         eventHandler.addEvent(new RouteEvent(client, packet));
428         eventHandler.run();
429
430         int indexAfterReceived = server.getReceivingWindow().receivingPacketIndex(
                packet, server.getConnection());
431         Assert.assertEquals(client.getThisReceivingWindowCapacity() - 1,
                indexAfterReceived);
432     }
433
434     @Test
435     public void inWindowShouldWorkOnPacketsThatShouldBeInWindowTest() {
436         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
437         Routable router = new Router.RouterBuilder().build();
438         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
439
440         new Channel.ChannelBuilder().build(client, router);
441         new Channel.ChannelBuilder().build(router, server);
442
443
444         client.updateRoutingTable();
445         router.updateRoutingTable();
446         server.updateRoutingTable();
447
448         EventHandler eventHandler = new EventHandler();
449         eventHandler.addEvent(new TCPConnectEvent(client, server));
450         eventHandler.run();
451
452         System.out.println("connected");
453
454         Assert.assertNotNull(client.getConnection());
455         Assert.assertNotNull(server.getConnection());
456
457         int seqNum = client.getConnection().getNextSequenceNumber();
```

```
458         int ackNum = client.getConnection().getNextAcknowledgementNumber();
459
460         for (int i = 0; i < client.getThisReceivingWindowCapacity(); i++) {
461             Packet packet = new PacketBuilder()
462                     .withOrigin(client)
463                     .withDestination(server)
464                     .withSequenceNumber(seqNum + i)
465                     .withAcknowledgmentNumber(ackNum + i)
466                     .build();
467
468
469             try {
470                 ReceivingWindow receivingWindow = server.getReceivingWindow();
471                 Assert.assertTrue(receivingWindow.inReceivingWindow(packet, server.
                        getConnection()));
472             } catch (IllegalAccessException e) {
473                 Assert.fail();
474             }
475         }
476     }
477
478     @Test
479     public void inWindowShouldNotWorkOnPacketsThatShouldNotBeInWindowTest() {
480         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
481         Routable router = new Router.RouterBuilder().build();
482         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
483
484         new Channel.ChannelBuilder().build(client, router);
485         new Channel.ChannelBuilder().build(router, server);
486
487         client.updateRoutingTable();
488         router.updateRoutingTable();
489         server.updateRoutingTable();
490
491         EventHandler eventHandler = new EventHandler();
492         eventHandler.addEvent(new TCPConnectEvent(client, server));
493         eventHandler.run();
494
495         System.out.println("connected");
496
497         int seqNum = client.getConnection().getNextSequenceNumber();
498         int ackNum = client.getConnection().getNextAcknowledgementNumber();
499
500         for (int i = 0; i < client.getThisReceivingWindowCapacity(); i++) {
501             Packet packet = new PacketBuilder()
502                     .withOrigin(client)
503                     .withDestination(server)
504                     .withSequenceNumber(seqNum + i + 1000)
505                     .withAcknowledgmentNumber(ackNum + i + 1000)
506                     .build();
507
508             try {
509                 ReceivingWindow receivingWindow = server.getReceivingWindow();
510                 Assert.assertFalse(receivingWindow.inReceivingWindow(packet, server.
                        getConnection()));
511             } catch (IllegalAccessException e) {
512                 Assert.fail();
513             }
514
515         }
516     }
```

```
517
518     @Test
519     public void floodWithPacketsInOrderShouldWorkTest() {
520         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().setReno().
                withReceivingWindowCapacity(7).build();
521         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.86).
                build();
522         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.86).
                build();
523         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().setReno().
                withReceivingWindowCapacity(20).build();
524
525         new Channel.ChannelBuilder().build(client, r1);
526         new Channel.ChannelBuilder().build(r1, r2);
527         new Channel.ChannelBuilder().build(r2, server);
528
529         client.updateRoutingTable();
530         r1.updateRoutingTable();
531         r2.updateRoutingTable();
532         server.updateRoutingTable();
533
534         EventHandler eventHandler = new EventHandler();
535         eventHandler.addEvent(new TCPConnectEvent(client, server));
536         eventHandler.run();
537
538         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 100;
539
540         for (int i = 1; i <= numPacketsToSend; i++) {
541             Message msg = new Message("test " + i);
542             client.send(msg);
543         }
544         eventHandler.addEvent(new RunTCPEvent(client));
545         eventHandler.run();
546
547         Assert.assertTrue(client.inputBufferIsEmpty());
548         Assert.assertTrue(server.inputBufferIsEmpty());
549         Assert.assertTrue(client.outputBufferIsEmpty());
550         Assert.assertTrue(server.outputBufferIsEmpty());
551         Assert.assertTrue(r1.inputBufferIsEmpty());
552
553         for (int i = 1; i <= numPacketsToSend; i++) {
554             Message msg = new Message("test " + i);
555             Packet received = server.receive();
556             Assert.assertNotNull(received);
557             Assert.assertEquals(received.getPayload(), msg);
558         }
559
560     }
561
562     @Test
563     public void floodWithPacketsInBigCongestedNetworkShouldWorkTest() {
564         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(10).build();
565         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                build();
566         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                build();
567         Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                build();
568         Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                build();
569         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).build();
```

```
570
571         new Channel.ChannelBuilder().build(client, r1);
572         new Channel.ChannelBuilder().build(r1, r2);
573         new Channel.ChannelBuilder().build(r2, r3);
574         new Channel.ChannelBuilder().build(r3, r4);
575         new Channel.ChannelBuilder().build(r4, server);
576
577         client.updateRoutingTable();
578         r1.updateRoutingTable();
579         r2.updateRoutingTable();
580         r3.updateRoutingTable();
581         r4.updateRoutingTable();
582         server.updateRoutingTable();
583
584         EventHandler eventHandler = new EventHandler();
585         eventHandler.addEvent(new TCPConnectEvent(client, server));
586         eventHandler.run();
587
588         Assert.assertTrue(client.isConnected());
589         Assert.assertTrue(server.isConnected());
590
591         System.out.println("connected");
592
593         Assert.assertTrue(client.inputBufferIsEmpty());
594         Assert.assertTrue(server.inputBufferIsEmpty());
595         Assert.assertTrue(client.outputBufferIsEmpty());
596         Assert.assertTrue(server.outputBufferIsEmpty());
597
598         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 1000;
599         for (int i = 1; i <= numPacketsToSend; i++) {
600             Message msg = new Message("test " + i);
601             client.send(msg);
602         }
603
604         eventHandler.addEvent(new RunTCPEvent(client));
605         eventHandler.run();
606
607         Assert.assertTrue(client.inputBufferIsEmpty());
608         Assert.assertTrue(server.inputBufferIsEmpty());
609         Assert.assertTrue(client.outputBufferIsEmpty());
610         Assert.assertTrue(server.outputBufferIsEmpty());
611
612         for (int i = 1; i <= numPacketsToSend; i++) {
613             Message msg = new Message("test " + i);
614             Packet received = server.receive();
615             Assert.assertNotNull(received);
616             Assert.assertEquals(msg, received.getPayload());
617         }
618     }
619
620
621     @Test
622     public void floodWithPacketsInOrderButInLossyChannelShouldWorkTest() {
623         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).build();
624         Routable router = new Router.RouterBuilder().build();
625         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(30).build();
626
627         new Channel.ChannelBuilder().withLoss(0.001).build(client, router);
628         new Channel.ChannelBuilder().build(router, server);
629
630         client.updateRoutingTable();
```

```
631         router.updateRoutingTable();
632         server.updateRoutingTable();
633 
634         EventHandler eventHandler = new EventHandler();
635         eventHandler.addEvent(new TCPConnectEvent(client, server));
636         eventHandler.run();
637 
638         Assert.assertTrue(client.isConnected());
639         Assert.assertTrue(server.isConnected());
640 
641         System.out.println("connected");
642 
643         Assert.assertTrue(client.inputBufferIsEmpty());
644         Assert.assertTrue(server.inputBufferIsEmpty());
645         Assert.assertTrue(client.outputBufferIsEmpty());
646         Assert.assertTrue(server.outputBufferIsEmpty());
647         Assert.assertTrue(router.inputBufferIsEmpty());
648 
649         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 100;
650         for (int i = 1; i <= numPacketsToSend; i++) {
651             Message msg = new Message("test " + i);
652             client.send(msg);
653         }
654 
655         eventHandler.addEvent(new RunTCPEvent(client));
656         eventHandler.run();
657 
658         Assert.assertTrue(client.inputBufferIsEmpty());
659         Assert.assertTrue(server.inputBufferIsEmpty());
660         Assert.assertTrue(client.outputBufferIsEmpty());
661         Assert.assertTrue(server.outputBufferIsEmpty());
662         Assert.assertTrue(router.inputBufferIsEmpty());
663 
664         for (int i = 1; i <= numPacketsToSend; i++) {
665             Message msg = new Message("test " + i);
666             Packet received = server.receive();
667             Assert.assertNotNull("iteration: " + i, received);
668             Assert.assertEquals(msg, received.getPayload());
669         }
670     }
671 
672     @Test
673     public void increasingWindowCapacityWithFloodWithPacketsInOrderButShouldWorkTest
        () {
674         for (int windowCapacity = 1; windowCapacity < 100; windowCapacity++) {
675             Util.setSeed(1337);
676             Util.resetTime();
677             ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(windowCapacity).build();
678             Routable router = new Router.RouterBuilder().build();
679             ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(windowCapacity).build();
680 
681             new Channel.ChannelBuilder().build(client, router);
682             new Channel.ChannelBuilder().build(router, server);
683 
684             client.updateRoutingTable();
685             router.updateRoutingTable();
686             server.updateRoutingTable();
687 
688             EventHandler eventHandler = new EventHandler();
689             eventHandler.addEvent(new TCPConnectEvent(client, server));
690             eventHandler.run();
```

```
691
692             Assert.assertTrue(client.isConnected());
693             Assert.assertTrue(server.isConnected());
694
695             System.out.println("connected");
696
697             Assert.assertTrue(client.inputBufferIsEmpty());
698             Assert.assertTrue(server.inputBufferIsEmpty());
699             Assert.assertTrue(client.outputBufferIsEmpty());
700             Assert.assertTrue(server.outputBufferIsEmpty());
701             Assert.assertTrue(router.inputBufferIsEmpty());
702
703             int numPacketsToSend = server.getThisReceivingWindowCapacity() * 10;
704             for (int i = 1; i <= numPacketsToSend; i++) {
705                 Message msg = new Message("test " + i);
706                 client.send(msg);
707             }
708
709             eventHandler.addEvent(new RunTCPEvent(client));
710             eventHandler.run();
711
712             Assert.assertTrue(client.inputBufferIsEmpty());
713             Assert.assertTrue(server.inputBufferIsEmpty());
714             Assert.assertTrue(client.outputBufferIsEmpty());
715             Assert.assertTrue(server.outputBufferIsEmpty());
716             Assert.assertTrue(router.inputBufferIsEmpty());
717
718             for (int i = 1; i <= numPacketsToSend; i++) {
719                 Message msg = new Message("test " + i);
720                 Packet received = server.receive();
721                 Assert.assertNotNull(received);
722                 Assert.assertEquals(msg, received.getPayload());
723             }
724         }
725     }
726
727     @Test
728     public void
        increasingWindowCapacityWithFloodWithPacketsInOrderButInLossyChannelShouldWorkTest
        () {
729         for (int windowCapacity = 1; windowCapacity < 50; windowCapacity++) {
730             Util.setSeed(1337);
731             Util.resetTime();
732             ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                    withReceivingWindowCapacity(windowCapacity).build();
733             Routable router = new Router.RouterBuilder().build();
734             ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                    withReceivingWindowCapacity(windowCapacity).build();
735
736             new Channel.ChannelBuilder().withLoss(0.001).build(client, router);
737             new Channel.ChannelBuilder().build(router, server);
738
739             client.updateRoutingTable();
740             router.updateRoutingTable();
741             server.updateRoutingTable();
742
743             EventHandler eventHandler = new EventHandler();
744             eventHandler.addEvent(new TCPConnectEvent(client, server));
745             eventHandler.run();
746
747             Assert.assertTrue(client.isConnected());
748             Assert.assertTrue(server.isConnected());
749
```

```
750             System.out.println("connected");
751
752             Assert.assertTrue(client.inputBufferIsEmpty());
753             Assert.assertTrue(server.inputBufferIsEmpty());
754             Assert.assertTrue(client.outputBufferIsEmpty());
755             Assert.assertTrue(server.outputBufferIsEmpty());
756             Assert.assertTrue(router.inputBufferIsEmpty());
757
758             int numPacketsToSend = server.getThisReceivingWindowCapacity() * 10;
759             for (int i = 1; i <= numPacketsToSend; i++) {
760                 Message msg = new Message("test " + i);
761                 client.send(msg);
762             }
763
764             eventHandler.addEvent(new RunTCPEvent(client));
765             eventHandler.run();
766
767             Assert.assertTrue(client.inputBufferIsEmpty());
768             Assert.assertTrue(server.inputBufferIsEmpty());
769             Assert.assertTrue(client.outputBufferIsEmpty());
770             Assert.assertTrue(server.outputBufferIsEmpty());
771             Assert.assertTrue(router.inputBufferIsEmpty());
772
773             for (int i = 1; i <= numPacketsToSend; i++) {
774                 Message msg = new Message("test " + i);
775                 Packet received = server.receive();
776                 Assert.assertNotNull(received);
777                 Assert.assertEquals(msg, received.getPayload());
778             }
779         }
780     }
781
782
783     @Test
784     public void floodWithPacketsBeforeConnectingShouldWorkTest() {
785         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
786         Routable router = new Router.RouterBuilder().withAverageQueueUtilization
                (0.7).build();
787         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
788
789         new Channel.ChannelBuilder().withLoss(0.0001).build(client, router);
790         new Channel.ChannelBuilder().build(router, server);
791
792         client.updateRoutingTable();
793         router.updateRoutingTable();
794         server.updateRoutingTable();
795
796         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 1000;
797         for (int i = 1; i <= numPacketsToSend; i++) {
798             Message msg = new Message("test " + i);
799             client.send(msg);
800         }
801
802         Assert.assertFalse(client.isConnected());
803         Assert.assertFalse(server.isConnected());
804
805         EventHandler eventHandler = new EventHandler();
806         eventHandler.addEvent(new TCPConnectEvent(client, server));
807         eventHandler.run();
808
809         Assert.assertTrue(client.isConnected());
```

```
810         Assert.assertTrue(server.isConnected());
811
812         System.out.println("connected");
813
814         Assert.assertTrue(client.inputBufferIsEmpty());
815         Assert.assertTrue(server.inputBufferIsEmpty());
816         Assert.assertTrue(client.outputBufferIsEmpty());
817         Assert.assertTrue(server.outputBufferIsEmpty());
818         Assert.assertTrue(router.inputBufferIsEmpty());
819
820         for (int i = 1; i <= numPacketsToSend; i++) {
821             Message msg = new Message("test " + i);
822             Packet received = server.receive();
823             Assert.assertNotNull(received);
824             Assert.assertEquals(msg, received.getPayload());
825         }
826     }
827
828
829     @Test
830     public void serverFloodWithPacketsInOrderShouldWorkTest() {
831         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
832         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.86).
                build();
833         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.86).
                build();
834         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(20).build();
835
836         new Channel.ChannelBuilder().build(client, r1);
837         new Channel.ChannelBuilder().build(r1, r2);
838         new Channel.ChannelBuilder().build(r2, server);
839
840         client.updateRoutingTable();
841         r1.updateRoutingTable();
842         r2.updateRoutingTable();
843         server.updateRoutingTable();
844
845         EventHandler eventHandler = new EventHandler();
846         eventHandler.addEvent(new TCPConnectEvent(client, server));
847         eventHandler.run();
848
849         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 100;
850
851         for (int i = 1; i <= numPacketsToSend; i++) {
852             Message msg = new Message("test " + i);
853             server.send(msg);
854         }
855         eventHandler.addEvent(new RunTCPEvent(server));
856         eventHandler.run();
857
858         Assert.assertTrue(client.inputBufferIsEmpty());
859         Assert.assertTrue(server.inputBufferIsEmpty());
860         Assert.assertTrue(client.outputBufferIsEmpty());
861         Assert.assertTrue(server.outputBufferIsEmpty());
862         Assert.assertTrue(r1.inputBufferIsEmpty());
863
864         for (int i = 1; i <= numPacketsToSend; i++) {
865             Message msg = new Message("test " + i);
866             Packet received = client.receive();
867             Assert.assertNotNull(received);
868             Assert.assertEquals(received.getPayload(), msg);
```

```
869         }
870     }
871
872
873     @Test
874     public void serverFloodWithPacketsBeforeConnectingShouldWorkTest() {
875         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
876         Routable router = new Router.RouterBuilder().build();
877         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
878
879         new Channel.ChannelBuilder().build(client, router);
880         new Channel.ChannelBuilder().build(router, server);
881
882         client.updateRoutingTable();
883         router.updateRoutingTable();
884         server.updateRoutingTable();
885
886         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 1000;
887         for (int i = 1; i <= numPacketsToSend; i++) {
888             Message msg = new Message("test " + i);
889             server.send(msg);
890         }
891
892         Assert.assertFalse(client.isConnected());
893         Assert.assertFalse(server.isConnected());
894
895         EventHandler eventHandler = new EventHandler();
896         eventHandler.addEvent(new TCPConnectEvent(client, server));
897         eventHandler.run();
898
899         Assert.assertTrue(client.isConnected());
900         Assert.assertTrue(server.isConnected());
901
902         System.out.println("connected");
903
904         Assert.assertTrue(client.inputBufferIsEmpty());
905         Assert.assertTrue(server.inputBufferIsEmpty());
906         Assert.assertTrue(client.outputBufferIsEmpty());
907         Assert.assertTrue(server.outputBufferIsEmpty());
908         Assert.assertTrue(router.inputBufferIsEmpty());
909
910         for (int i = 1; i <= numPacketsToSend; i++) {
911             Message msg = new Message("test " + i);
912             Packet received = client.receive();
913             Assert.assertNotNull(received);
914             Assert.assertEquals(msg, received.getPayload());
915         }
916     }
917
918 }
```

```java
package org.example.protocol;

import org.example.data.Message;
import org.example.data.Packet;
import org.example.data.Payload;
import org.example.network.Channel;
import org.example.network.Routable;
import org.example.network.Router;
import org.example.network.address.SimpleAddress;
import org.example.simulator.EventHandler;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import java.util.concurrent.TimeUnit;

public class MPTCPTest {

    @Rule
    public Timeout globalTimeout = new Timeout(120, TimeUnit.SECONDS);

    @Before
    public void setup() {
        Util.setSeed(1337);
        Util.resetTime();
    }

    @Test
    public void MPTCPWithTwoSubFlowsAndNonDistinctPathConnectAndSendCorrectTest() {
        MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
            withReceivingWindowCapacity(14).withAddress(new SimpleAddress("MPTCP-
            Client")).build();

        Router r11 = new Router.RouterBuilder().build();
        Router r12 = new Router.RouterBuilder().build();

        Router r3 = new Router.RouterBuilder().build();
        Router r4 = new Router.RouterBuilder().build();

        MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
            withReceivingWindowCapacity(14).withAddress(new SimpleAddress("MPTCP-
            Server")).build();

        new Channel.ChannelBuilder().build(client, r11);
        new Channel.ChannelBuilder().build(client, r12);
        new Channel.ChannelBuilder().build(r11, r3);
        new Channel.ChannelBuilder().build(r12, r3);
        new Channel.ChannelBuilder().build(r3, r4);
        new Channel.ChannelBuilder().build(r4, server);
        new Channel.ChannelBuilder().build(r4, server);

        client.updateRoutingTable();
        r11.updateRoutingTable();
        r12.updateRoutingTable();
        r3.updateRoutingTable();
        r4.updateRoutingTable();
        server.updateRoutingTable();
```

```
59
60          EventHandler eventHandler = new EventHandler();
61          eventHandler.addEvent(new TCPConnectEvent(client, server));
62          eventHandler.run();
63          System.out.println("connected");
64
65
66          Message msg = new Message("Test");
67          client.send(msg);
68          eventHandler.addEvent(new RunTCPEvent(client));
69          eventHandler.run();
70
71          Packet receivedPacket = server.receive();
72          Assert.assertNotNull(receivedPacket);
73
74          Payload receivedPayload = receivedPacket.getPayload();
75          Assert.assertEquals(receivedPayload, msg);
76
77          Assert.assertNull(server.receive());
78      }
79
80      @Test
81      public void MPTCPConnectToTCPTest() {
82          MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(14).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
83          Routable router = new Router.RouterBuilder().withAddress(new SimpleAddress("
                router")).build();
84          ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().withAddress(new
                SimpleAddress("server")).withReceivingWindowCapacity(7).build();
85
86          new Channel.ChannelBuilder().build(client, router);
87          new Channel.ChannelBuilder().build(client, router);
88          new Channel.ChannelBuilder().build(router, server);
89          new Channel.ChannelBuilder().build(router, server);
90
91
92          client.updateRoutingTable();
93          router.updateRoutingTable();
94          server.updateRoutingTable();
95
96          System.out.println(client.getChannels());
97          System.out.println(router.getChannels());
98          System.out.println(server.getChannels());
99
100         EventHandler eventHandler = new EventHandler();
101         eventHandler.addEvent(new TCPConnectEvent(client, server));
102         eventHandler.run();
103
104         Assert.assertFalse(client.isConnected());
105         Assert.assertTrue(client.getSubflows()[0].isConnected());
106         Assert.assertFalse(client.getSubflows()[1].isConnected());
107     }
108
109
110     @Test
111     public void MPTCPConnectThenSendMsgOverTwoSubflowsTest() {
112         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(14).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
113         Routable r1 = new Router.RouterBuilder().build();
114         Routable r2 = new Router.RouterBuilder().build();
115         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
```

```
            withReceivingWindowCapacity(7).build();
116
117         //path one
118         new Channel.ChannelBuilder().build(client, r1);
119         new Channel.ChannelBuilder().build(r1, server);
120
121         //path two
122         new Channel.ChannelBuilder().build(client, r2);
123         new Channel.ChannelBuilder().build(r2, server);
124
125
126         client.updateRoutingTable();
127         r1.updateRoutingTable();
128         r2.updateRoutingTable();
129         server.updateRoutingTable();
130
131         EventHandler eventHandler = new EventHandler();
132         eventHandler.addEvent(new TCPConnectEvent(client, server));
133         eventHandler.run();
134
135         Assert.assertTrue(client.getSubflows()[0].isConnected());
136         Assert.assertFalse(client.getSubflows()[1].isConnected());
137
138         Message msg1 = new Message("Test 1!");
139         Message msg2 = new Message("Test 2!");
140
141         client.send(msg1);
142         client.send(msg2);
143         eventHandler.addEvent(new RunTCPEvent(client));
144         eventHandler.run();
145
146         Packet received1 = server.receive();
147         Assert.assertNotNull(received1);
148         Assert.assertEquals(msg1, received1.getPayload());
149
150         Packet received2 = server.receive();
151         Assert.assertNotNull(received2);
152         Assert.assertEquals(msg2, received2.getPayload());
153     }
154
155
156     @Test
157     public void MPTCPConnectToMPTCPNonDistinctPathTest() {
158         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(20).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
159         Routable r1 = new Router.RouterBuilder().withAddress(new SimpleAddress("A"))
                .build();
160         Routable r2 = new Router.RouterBuilder().withAddress(new SimpleAddress("B"))
                .build();
161         Routable r3 = new Router.RouterBuilder().withAddress(new SimpleAddress("C"))
                .build();
162         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(2).
                withReceivingWindowCapacity(20).withAddress(new SimpleAddress("MPTCP-
                Server")).build();
163
164         new Channel.ChannelBuilder().build(client, r1);
165         new Channel.ChannelBuilder().build(client, r2);
166
167         new Channel.ChannelBuilder().build(r3, r1);
168         new Channel.ChannelBuilder().build(r3, r2);
169
170         new Channel.ChannelBuilder().build(server, r3);
```

```
171         new Channel.ChannelBuilder().build(server, r3);
172
173
174         client.updateRoutingTable();
175         r1.updateRoutingTable();
176         r2.updateRoutingTable();
177         r3.updateRoutingTable();
178         server.updateRoutingTable();
179
180         EventHandler eventHandler = new EventHandler();
181         eventHandler.addEvent(new TCPConnectEvent(client, server));
182         eventHandler.run();
183         System.out.println("connected");
184
185         Assert.assertTrue(client.isConnected());
186
187         Message msg1 = new Message("Test 1!");
188         Message msg2 = new Message("Test 2!");
189         Message msg3 = new Message("Test 3!");
190
191         client.send(msg1);
192         client.send(msg2);
193         client.send(msg3);
194         eventHandler.addEvent(new RunTCPEvent(client));
195         eventHandler.run();
196
197         Packet received1 = server.receive();
198         Assert.assertNotNull(received1);
199         Assert.assertEquals(msg1, received1.getPayload());
200
201         Packet received2 = server.receive();
202         Assert.assertNotNull(received2);
203         Assert.assertEquals(msg2, received2.getPayload());
204
205         Packet received3 = server.receive();
206         Assert.assertNotNull(received3);
207         Assert.assertEquals(msg3, received3.getPayload());
208
209         Assert.assertNull(server.receive());
210     }
211
212     @Test
213     public void MPTCPConnectToMPTCPThenSendMsgOverThreeSubflowsTest() {
214         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
215         Routable r1 = new Router.RouterBuilder().withAddress(new SimpleAddress("A"))
                .build();
216         Routable r2 = new Router.RouterBuilder().withAddress(new SimpleAddress("B"))
                .build();
217         Routable r3 = new Router.RouterBuilder().withAddress(new SimpleAddress("C"))
                .build();
218         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Server")).build();
219
220         //path one
221         new Channel.ChannelBuilder().build(client, r1);
222         new Channel.ChannelBuilder().build(r1, server);
223
224         //path two
225         new Channel.ChannelBuilder().build(client, r2);
226         new Channel.ChannelBuilder().build(r2, server);
```

```
227
228         //path three
229         new Channel.ChannelBuilder().build(client, r3);
230         new Channel.ChannelBuilder().build(r3, server);
231
232
233         client.updateRoutingTable();
234         r1.updateRoutingTable();
235         r2.updateRoutingTable();
236         r3.updateRoutingTable();
237         server.updateRoutingTable();
238
239         EventHandler eventHandler = new EventHandler();
240         eventHandler.addEvent(new TCPConnectEvent(client, server));
241         eventHandler.run();
242         System.out.println("connected");
243
244         Assert.assertTrue(client.getSubflows()[0].isConnected());
245         Assert.assertTrue(client.getSubflows()[1].isConnected());
246         Assert.assertTrue(client.getSubflows()[2].isConnected());
247
248         Message msg1 = new Message("Test 1!");
249         Message msg2 = new Message("Test 2!");
250         Message msg3 = new Message("Test 3!");
251
252         client.send(msg1);
253         client.send(msg2);
254         client.send(msg3);
255         eventHandler.addEvent(new RunTCPEvent(client));
256         eventHandler.run();
257
258         Packet received1 = server.receive();
259         Assert.assertNotNull(received1);
260         Assert.assertEquals(msg1, received1.getPayload());
261
262         Packet received2 = server.receive();
263         Assert.assertNotNull(received2);
264         Assert.assertEquals(msg2, received2.getPayload());
265
266         Packet received3 = server.receive();
267         Assert.assertNotNull(received3);
268         Assert.assertEquals(msg3, received3.getPayload());
269
270         Assert.assertNull(server.receive());
271     }
272
273
274     @Test
275     public void MPTCPFloodWithPacketsInOrderShouldWorkTest() {
276         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
277         Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                withAddress(new SimpleAddress("A")).build();
278         Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                withAddress(new SimpleAddress("B")).build();
279         Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.8).
                withAddress(new SimpleAddress("C")).build();
280         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Server")).build();
281
282         //path one
```

```
283         new Channel.ChannelBuilder().build(client, r1);
284         new Channel.ChannelBuilder().build(r1, server);
285 
286         //path two
287         new Channel.ChannelBuilder().build(client, r2);
288         new Channel.ChannelBuilder().build(r2, server);
289 
290         //path three
291         new Channel.ChannelBuilder().build(client, r3);
292         new Channel.ChannelBuilder().build(r3, server);
293 
294 
295         client.updateRoutingTable();
296         r1.updateRoutingTable();
297         r2.updateRoutingTable();
298         r3.updateRoutingTable();
299         server.updateRoutingTable();
300 
301         EventHandler eventHandler = new EventHandler();
302         eventHandler.addEvent(new TCPConnectEvent(client, server));
303         eventHandler.run();
304         System.out.println("connected");
305 
306         Assert.assertTrue(client.getSubflows()[0].isConnected());
307         Assert.assertTrue(client.getSubflows()[1].isConnected());
308         Assert.assertTrue(client.getSubflows()[2].isConnected());
309 
310         int numPacketsToSend = 10000;
311 
312         for (int i = 1; i <= numPacketsToSend; i++) {
313             Message msg = new Message("test " + i);
314             client.send(msg);
315         }
316         eventHandler.addEvent(new RunTCPEvent(client));
317         eventHandler.run();
318 
319         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
320         Assert.assertTrue(server.outputBufferIsEmpty());
321         Assert.assertTrue(client.inputBufferIsEmpty());
322         Assert.assertTrue(server.inputBufferIsEmpty());
323         Assert.assertTrue(r1.inputBufferIsEmpty());
324         Assert.assertTrue(r2.inputBufferIsEmpty());
325         Assert.assertTrue(r3.inputBufferIsEmpty());
326 
327 
328         for (int i = 1; i <= numPacketsToSend; i++) {
329             Message msg = new Message("test " + i);
330             Packet received = server.receive();
331             Assert.assertNotNull(received);
332             Assert.assertEquals("iteration " + i, received.getPayload(), msg);
333         }
334         Assert.assertNull(server.receive());
335 
336     }
337 
338     @Test
339     public void MPTCPFloodWithPacketsInLossyChannelsShouldWorkTest() {
340         MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
341         Routable r1 = new Router.RouterBuilder().withAddress(new SimpleAddress("A"))
                .build();
```

```
342         Routable r2 = new Router.RouterBuilder().withAddress(new SimpleAddress("B"))
                .build();
343         Routable r3 = new Router.RouterBuilder().withAddress(new SimpleAddress("C"))
                .build();
344         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(21).withAddress(new SimpleAddress("MPTCP-
                Server")).build();
345
346         //path one
347         new Channel.ChannelBuilder().withLoss(0.001).build(client, r1);
348         new Channel.ChannelBuilder().withLoss(0.01).build(r1, server);
349
350         //path two
351         new Channel.ChannelBuilder().withLoss(0.001).build(client, r2);
352         new Channel.ChannelBuilder().withLoss(0.001).build(r2, server);
353
354         //path three
355         new Channel.ChannelBuilder().withLoss(0.01).build(client, r3);
356         new Channel.ChannelBuilder().withLoss(0.001).build(r3, server);
357
358         client.updateRoutingTable();
359         r1.updateRoutingTable();
360         r2.updateRoutingTable();
361         r3.updateRoutingTable();
362         server.updateRoutingTable();
363
364         EventHandler eventHandler = new EventHandler();
365         eventHandler.addEvent(new TCPConnectEvent(client, server));
366         eventHandler.run();
367         System.out.println("connected");
368
369         Assert.assertTrue(client.getSubflows()[0].isConnected());
370         Assert.assertTrue(client.getSubflows()[1].isConnected());
371         Assert.assertTrue(client.getSubflows()[2].isConnected());
372
373         int numPacketsToSend = 10000;
374
375         for (int i = 1; i <= numPacketsToSend; i++) {
376             Message msg = new Message("test " + i);
377             client.send(msg);
378         }
379         eventHandler.addEvent(new RunTCPEvent(client));
380         eventHandler.run();
381
382         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
383         Assert.assertTrue(server.outputBufferIsEmpty());
384
385         Assert.assertTrue(client.inputBufferIsEmpty());
386         Assert.assertTrue(server.inputBufferIsEmpty());
387         Assert.assertTrue(r1.inputBufferIsEmpty());
388         Assert.assertTrue(r2.inputBufferIsEmpty());
389         Assert.assertTrue(r3.inputBufferIsEmpty());
390
391         for (int i = 1; i <= numPacketsToSend; i++) {
392             Message msg = new Message("test " + i);
393             Packet received = server.receive();
394             Assert.assertNotNull(received);
395             Assert.assertEquals("iteration " + i, received.getPayload(), msg);
396         }
397         Assert.assertNull(server.receive());
398
399     }
```

```
400
401
402     @Test
403     public void
            MPTCPFloodWithPacketsInOrderWithVariableNumberOfSubflowsShouldWorkTest() {
404         int maxSubflows = 5;
405         for (int numSubflows = 1; numSubflows <= maxSubflows; numSubflows++) {
406             Util.setSeed(1337);
407             Util.resetTime();
408             MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(numSubflows
                    ).withReceivingWindowCapacity(21).withAddress(new SimpleAddress("
                    MPTCP-Client")).build();
409             MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(numSubflows
                    ).withReceivingWindowCapacity(21).withAddress(new SimpleAddress("
                    MPTCP-Server")).build();
410
411             for (int i = 1; i <= numSubflows; i++) {
412                 Routable router = new Router.RouterBuilder().withAddress(new
                        SimpleAddress("Router " + i)).build();
413
414                 new Channel.ChannelBuilder().build(client, router);
415                 new Channel.ChannelBuilder().build(router, server);
416                 router.updateRoutingTable();
417             }
418             client.updateRoutingTable();
419             server.updateRoutingTable();
420
421             EventHandler eventHandler = new EventHandler();
422             eventHandler.addEvent(new TCPConnectEvent(client, server));
423             eventHandler.run();
424             System.out.println("connected");
425
426             for (TCP subflow : client.getSubflows()) {
427                 Assert.assertTrue(subflow.isConnected());
428             }
429
430             int numPacketsToSend = 10000;
431             for (int i = 1; i <= numPacketsToSend; i++) {
432                 Message msg = new Message("test " + i);
433                 client.send(msg);
434             }
435             eventHandler.addEvent(new RunTCPEvent(client));
436             eventHandler.run();
437
438             Assert.assertTrue("client still has packets to send", client.
                    outputBufferIsEmpty());
439             Assert.assertTrue(server.outputBufferIsEmpty());
440             Assert.assertTrue(client.inputBufferIsEmpty());
441             Assert.assertTrue(server.inputBufferIsEmpty());
442             for (TCP subflow : client.getSubflows()) {
443                 Assert.assertTrue(subflow.inputBufferIsEmpty());
444             }
445
446             for (int i = 1; i <= numPacketsToSend; i++) {
447                 Message msg = new Message("test " + i);
448                 Packet received = server.receive();
449                 Assert.assertNotNull(received);
450                 Assert.assertEquals("iteration " + i, received.getPayload(), msg);
451             }
452             Assert.assertNull(server.receive());
453         }
454     }
455
```

```
456
457 }
```

```java
package org.example.protocol.window;

import org.example.data.*;
import org.example.network.Channel;
import org.example.protocol.ClassicTCP;
import org.example.protocol.window.receiving.ReceivingWindow;
import org.example.protocol.window.receiving.SelectiveRepeat;
import org.example.protocol.window.sending.SendingWindow;
import org.example.protocol.window.sending.SlidingWindow;
import org.example.simulator.EventHandler;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.simulator.statistics.TCPStats;
import org.javatuples.Pair;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.concurrent.TimeUnit;


public class ReceivingWindowTest {

    private static final Comparator<Packet> PACKET_COMPARATOR = Comparator.
        comparingInt(Packet::getSequenceNumber);
    @Rule
    public Timeout globalTimeout = new Timeout(30, TimeUnit.SECONDS);
    private ClassicTCP client;
    private ClassicTCP server;
    private ReceivingWindow receivingWindow;
    private List<Packet> receivedPackets;
    private List<Pair<Integer, Payload>> payloadsToSend;

    @Before
    public void setup() {
        this.receivedPackets = new ArrayList<>();
        this.payloadsToSend = new ArrayList<>();
        this.client = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();
        this.server = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (7).build();
        this.connect(client, server);

        this.receivingWindow = new SelectiveRepeat(10, PACKET_COMPARATOR, this.
            receivedPackets);
    }

    private void connect(ClassicTCP client, ClassicTCP server) {
        new Channel.ChannelBuilder().build(client, server);
        client.updateRoutingTable();
        server.updateRoutingTable();

        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();
    }


```

```
59      @Test
60      public void ackThisReturnsPacketTest() {
61          Packet toBeAcked = this.receivingWindow.ackThis(this.client);
62          Assert.assertNotNull(toBeAcked);
63      }
64
65
66      @Test
67      public void shouldAckReturnsTrueTest() {
68          Assert.assertTrue(this.receivingWindow.shouldAck());
69      }
70
71      @Test
72      public void getReceivedPacketsShouldReturnEmptyQueueWhenNoPacketsAreReceived() {
73          Assert.assertTrue(this.receivedPackets.isEmpty());
74      }
75
76      @Test
77      public void receiveShouldReturnFalseIfNoPacketsAreReceived() {
78          Assert.assertFalse(this.receivingWindow.receive(null));
79      }
80
81
82      @Test(expected = NullPointerException.class)
83      public void receiveAckWithNullSendingWindowArgumentCausesNullPointerException()
            {
84          Packet ackFromServer = new PacketBuilder()
85                  .withSequenceNumber(this.client.getConnection().
                        getNextAcknowledgementNumber()) //hack to find clients expected
                        next packet sequence number
86                  .withFlags(Flag.ACK)
87                  .withConnection(this.server.getConnection())
88                  .build();
89          Assert.assertTrue(this.receivingWindow.inReceivingWindow(ackFromServer,
                client.getConnection()));
90          this.receivingWindow.offer(ackFromServer);
91          Assert.assertFalse(this.receivingWindow.receive(null));
92      }
93
94      @Test
95      public void receiveReturnsFalseIfAckIsReceived() {
96          Packet ackFromServer = new PacketBuilder()
97                  .withSequenceNumber(this.client.getConnection().
                        getNextAcknowledgementNumber())
98                  .withFlags(Flag.ACK)
99                  .withConnection(this.server.getConnection())
100                 .build();
101
102         Assert.assertTrue(this.receivingWindow.inReceivingWindow(ackFromServer,
                client.getConnection()));
103         this.receivingWindow.offer(ackFromServer);
104
105         SendingWindow sendingWindow = new SlidingWindow(10, true, client.
                getConnection(), PACKET_COMPARATOR, this.payloadsToSend, new TCPStats(
                this.client.getAddress()));
106         Assert.assertFalse(this.receivingWindow.receive(sendingWindow));
107     }
108
109     @Test
110     public void receiveReturnsTrueIfPacketIsReceived() {
111         Packet packetFromServer = new PacketBuilder()
112                 .withSequenceNumber(this.client.getConnection().
                        getNextAcknowledgementNumber())
```

```
113                     .withPayload(new Message("test"))
114                     .withConnection(this.server.getConnection())
115                     .build();
116
117             Assert.assertTrue(this.receivingWindow.inReceivingWindow(packetFromServer,
                    client.getConnection()));
118             this.receivingWindow.offer(packetFromServer);
119
120             SendingWindow sendingWindow = new SlidingWindow(10, true, client.
                    getConnection(), PACKET_COMPARATOR, this.payloadsToSend, new TCPStats(
                    this.client.getAddress()));
121             Assert.assertTrue(this.receivingWindow.receive(sendingWindow));
122
123             Assert.assertEquals(packetFromServer, this.receivedPackets.remove(0));
124         }
125
126
127 }
```

```java
package org.example.protocol.window;

import org.example.data.Message;
import org.example.data.Packet;
import org.example.data.PacketBuilder;
import org.example.network.Channel;
import org.example.protocol.ClassicTCP;
import org.example.protocol.window.sending.SendingWindow;
import org.example.simulator.EventHandler;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import java.util.concurrent.TimeUnit;


public class SendingWindowTest {

    @Rule
    public Timeout globalTimeout = new Timeout(30, TimeUnit.SECONDS);
    private ClassicTCP client;
    private ClassicTCP server;
    private SendingWindow sendingWindow;
    private EventHandler eventHandler;

    @Before
    public void setup() throws IllegalAccessException {
        this.client = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (20).setTahoe().build();
        this.server = new ClassicTCP.ClassicTCPBuilder().withReceivingWindowCapacity
            (20).setTahoe().build();
        this.connect(client, server);

        this.sendingWindow = this.client.getSendingWindow();

    }

    private void connect(ClassicTCP client, ClassicTCP server) {
        new Channel.ChannelBuilder().build(client, server);
        client.updateRoutingTable();
        server.updateRoutingTable();

        this.eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();
    }

    @Test
    public void initCorrectWindowSizeTest() {
        Assert.assertEquals(1, this.sendingWindow.getWindowCapacity());
    }

    @Test
    public void initIsWaitingForAckIsFalseIfNoPacketsAreSentTest() {
        Assert.assertFalse(this.sendingWindow.isWaitingForAck());
    }

    @Test
```

```
61      public void initIsWaitingForAckIsTrueIfSendingWindowIsFullTest() {
62          for (int i = 0; i < this.client.getThisReceivingWindowCapacity(); i++) {
63              this.client.send(new Message("test " + i));
64          }
65          for (int i = 0; i < this.client.getThisReceivingWindowCapacity(); i++) {
66              this.client.trySend();
67          }
68          Assert.assertTrue(this.sendingWindow.isWaitingForAck());
69      }
70
71      @Test
72      public void isWaitingForAckIsFalseIfSendingWindowIsAlmostFullTest() throws
            IllegalAccessException {
73          for (int i = 0; i < Math.pow(this.server.getThisReceivingWindowCapacity(),
                2); i++) {
74              this.sendingWindow.increase();
75          }
76          for (int i = 0; i < this.client.getOtherReceivingWindowCapacity() - 1; i++)
                {
77              this.client.getSendingWindow().add(new PacketBuilder().build());
78          }
79          Assert.assertEquals(this.client.getOtherReceivingWindowCapacity(), this.
                sendingWindow.getWindowCapacity());
80          Assert.assertFalse(this.sendingWindow.isWaitingForAck());
81      }
82
83      @Test
84      public void windowCantIncreaseToMoreThanServersReceivingWindow() {
85          for (int i = 0; i < Math.pow(this.server.getThisReceivingWindowCapacity(),
                2) * 10; i++) {
86              this.sendingWindow.increase();
87          }
88          Assert.assertEquals(this.server.getThisReceivingWindowCapacity(), this.
                sendingWindow.getWindowCapacity());
89      }
90
91      @Test
92      public void windowCanDecreaseWhenInMaxCapacity() {
93          for (int i = 0; i < Math.pow(this.server.getThisReceivingWindowCapacity(),
                2); i++) {
94              this.sendingWindow.increase();
95          }
96          this.sendingWindow.decrease();
97          Assert.assertEquals(1, this.sendingWindow.getWindowCapacity());
98      }
99
100     @Test
101     public void initWindowCanNotDecrease() {
102         this.sendingWindow.decrease();
103         Assert.assertEquals(1, this.sendingWindow.getWindowCapacity());
104     }
105
106     @Test
107     public void windowWillDecreaseToDefaultValueIfDecreasedEnough() {
108         for (int i = 0; i < this.server.getThisReceivingWindowCapacity(); i++) {
109             this.sendingWindow.increase();
110         }
111
112         for (int i = 0; i < this.server.getThisReceivingWindowCapacity(); i++) {
113             this.sendingWindow.decrease();
114         }
115         Assert.assertEquals(1, this.sendingWindow.getWindowCapacity());
116     }
```

```
117
118     @Test
119     public void floodWithPacketsInLossyChannelShouldResultInVariableWindowCapacity()
            throws IllegalAccessException {
120         Assert.assertTrue(server.isConnected());
121         Assert.assertTrue(client.isConnected());
122
123         System.out.println("connected");
124
125         Assert.assertTrue(client.inputBufferIsEmpty());
126         Assert.assertTrue(server.inputBufferIsEmpty());
127         Assert.assertTrue(client.outputBufferIsEmpty());
128         Assert.assertTrue(server.outputBufferIsEmpty());
129
130         int numPacketsToSend = server.getThisReceivingWindowCapacity() * 100;
131         for (int i = 1; i <= numPacketsToSend; i++) {
132             Message msg = new Message("test " + i);
133             client.send(msg);
134         }
135
136         int prevWindowCapacity = client.getSendingWindow().getWindowCapacity();
137         eventHandler.addEvent(new RunTCPEvent(client));
138         while (eventHandler.singleRun()) {
139             int curWindowCapacity = client.getSendingWindow().getWindowCapacity();
140
141             boolean loss = curWindowCapacity < prevWindowCapacity;
142             boolean packetAcked = curWindowCapacity > prevWindowCapacity;
143
144             if (loss) {
145                 Assert.assertEquals(1, curWindowCapacity);
146             } else if (packetAcked) {
147                 Assert.assertTrue(this.client.getOtherReceivingWindowCapacity() >=
                        curWindowCapacity);
148             } else {
149                 Assert.assertEquals(prevWindowCapacity, curWindowCapacity);
150             }
151             prevWindowCapacity = curWindowCapacity;
152         }
153
154         Assert.assertTrue(client.inputBufferIsEmpty());
155         Assert.assertTrue(server.inputBufferIsEmpty());
156         Assert.assertTrue(client.outputBufferIsEmpty());
157         Assert.assertTrue(server.outputBufferIsEmpty());
158
159         for (int i = 1; i <= numPacketsToSend; i++) {
160             Message msg = new Message("test " + i);
161             Packet received = server.receive();
162             Assert.assertNotNull(received);
163             Assert.assertEquals(msg, received.getPayload());
164         }
165
166     }
167
168
169 }
```

```java
package org.example.simulator;

import org.example.data.Message;
import org.example.data.Packet;
import org.example.network.Channel;
import org.example.network.Router;
import org.example.protocol.ClassicTCP;
import org.example.simulator.events.Event;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import java.util.ArrayList;
import java.util.Queue;
import java.util.concurrent.TimeUnit;


public class EventHandlerTest {

    @Rule
    public Timeout globalTimeout = new Timeout(30, TimeUnit.SECONDS);

    @Before
    public void setup() {
        Util.setSeed(1337);
        Util.resetTime();
    }

    @Test
    public void runRunsWithoutErrorTest() {
        EventHandler eventHandler = new EventHandler();
        Event eventOne = new Event(Util.getTime()) {
            @Override
            public void run() {
                System.out.println(this.getInstant());
            }

            @Override
            public void generateNextEvent(Queue<Event> events) {
                events.add(new Event(Util.getTime()) {
                    @Override
                    public void run() {
                        System.out.println(this.getInstant());
                    }

                    @Override
                    public void generateNextEvent(Queue<Event> events) {

                    }
                });
            }
        };

        Event eventTwo = new Event(Util.getTime()) {
            @Override
            public void run() {
                System.out.println(this.getInstant());
```

```
63              }
64
65              @Override
66              public void generateNextEvent(Queue<Event> events) {
67                  events.add(new Event(Util.getTime()) {
68                      @Override
69                      public void run() {
70                          System.out.println(this.getInstant());
71                      }
72
73                      @Override
74                      public void generateNextEvent(Queue<Event> events) {
75
76                      }
77                  });
78              }
79          };
80
81          //start condition
82          eventHandler.addEvent(eventOne);
83          eventHandler.addEvent(eventTwo);
84
85          eventHandler.run();
86
87          Assert.assertEquals(0, eventHandler.getNumberOfEvents());
88      }
89
90
91      @Test
92      public void runTest() {
93          EventHandler eventHandler = new EventHandler();
94
95          ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
96          ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
97          Router r1 = new Router.RouterBuilder().build();
98
99          new Channel.ChannelBuilder().build(client, r1);
100         new Channel.ChannelBuilder().build(r1, server);
101
102         client.updateRoutingTable();
103         r1.updateRoutingTable();
104         server.updateRoutingTable();
105
106         eventHandler.addEvent(new TCPConnectEvent(client, server));
107         eventHandler.run();
108
109
110         client.send(new Message("test"));
111         eventHandler.addEvent(new RunTCPEvent(client));
112         eventHandler.run();
113
114         Assert.assertEquals(0, eventHandler.getNumberOfEvents());
115     }
116
117     @Test
118     public void runFloodWithPacketsInOrderButInLossyChannelShouldWorkTest() {
119         EventHandler eventHandler = new EventHandler();
120
121         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
122         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
```

```
            withReceivingWindowCapacity(7).build();
123         Router r1 = new Router.RouterBuilder().build();
124
125         new Channel.ChannelBuilder().build(client, r1);
126         new Channel.ChannelBuilder().build(r1, server);
127
128         client.updateRoutingTable();
129         r1.updateRoutingTable();
130         server.updateRoutingTable();
131
132         eventHandler.addEvent(new TCPConnectEvent(client, server));
133         eventHandler.run();
134
135         int multiplier = 100;
136         int numPacketsToSend = server.getThisReceivingWindowCapacity() * multiplier;
137
138         for (int i = 1; i <= numPacketsToSend; i++) {
139             Message msg = new Message("test " + i);
140             client.send(msg);
141         }
142         eventHandler.addEvent(new RunTCPEvent(client));
143         eventHandler.run();
144
145         for (int i = 1; i <= numPacketsToSend; i++) {
146             Message msg = new Message("test " + i);
147             Packet received = server.receive();
148             Assert.assertNotNull(received);
149             Assert.assertEquals(received.getPayload(), msg);
150         }
151     }
152
153     private ArrayList<Event> allEventsList(int numPacketsToSend, double
            noiseTolerance) {
154         Util.setSeed(1337);
155         Util.resetTime();
156         EventHandler eventHandler = new EventHandler();
157
158         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
159         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
160         Router r1 = new Router.RouterBuilder().build();
161
162         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(client, r1);
163         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(r1, server);
164
165         client.updateRoutingTable();
166         r1.updateRoutingTable();
167         server.updateRoutingTable();
168
169         eventHandler.addEvent(new TCPConnectEvent(client, server));
170         eventHandler.run();
171
172         Assert.assertNull(eventHandler.peekEvent());
173
174         for (int i = 1; i <= numPacketsToSend; i++) {
175             Message msg = new Message("test " + i);
176             client.send(msg);
177         }
178         eventHandler.addEvent(new RunTCPEvent(client));
179
180         ArrayList<Event> eventList = new ArrayList<>();
181
```

```
182         while (eventHandler.peekEvent() != null) {
183             eventList.add(eventHandler.peekEvent());
184             eventHandler.singleRun();
185         }
186 
187         Assert.assertNull(server.dequeueInputBuffer());
188         Assert.assertNull(client.dequeueInputBuffer());
189         Assert.assertNull(r1.dequeueInputBuffer());
190         Assert.assertNull(eventHandler.peekEvent());
191 
192         return eventList;
193     }
194 
195     @Test
196     public void eventArrangementsAreConsistent() {
197         double noiseTolerance = 0.01;
198         int numPacketsToSend = 1001;
199         ArrayList<Event> eventList1 = this.allEventsList(numPacketsToSend,
                noiseTolerance);
200         ArrayList<Event> eventList2 = this.allEventsList(numPacketsToSend,
                noiseTolerance);
201 
202         for (Event event : eventList1) {
203             Assert.assertEquals(event.getClass(), eventList2.remove(0).getClass());
204         }
205     }
206 
207 
208     @Test
209     public void eventAreRunningInCorrectOrderWithRespectToTime() {
210         double noiseTolerance = 0.01;
211         EventHandler eventHandler = new EventHandler();
212 
213         ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
214         ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
215         Router r1 = new Router.RouterBuilder().build();
216 
217         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(client, r1);
218         new Channel.ChannelBuilder().withLoss(noiseTolerance).build(r1, server);
219 
220         client.updateRoutingTable();
221         r1.updateRoutingTable();
222         server.updateRoutingTable();
223 
224         eventHandler.addEvent(new TCPConnectEvent(client, server));
225         eventHandler.run();
226 
227         Assert.assertNull(eventHandler.peekEvent());
228 
229         Util.setSeed(1337);
230 
231         int numPacketsToSend = 1000;
232 
233         for (int i = 1; i <= numPacketsToSend; i++) {
234             Message msg = new Message("test " + i);
235             client.send(msg);
236         }
237         eventHandler.addEvent(new RunTCPEvent(client));
238 
239         Event prevEvent;
240         while (true) {
```

```
241             prevEvent = eventHandler.peekEvent();
242             eventHandler.singleRun();
243
244             if (eventHandler.peekEvent() == null) break;
245
246             Event curEvent = eventHandler.peekEvent();
247             Assert.assertTrue(prevEvent.getInstant() <= curEvent.getInstant());
248         }
249         Assert.assertNull(server.dequeueInputBuffer());
250         Assert.assertNull(client.dequeueInputBuffer());
251         Assert.assertNull(r1.dequeueInputBuffer());
252         Assert.assertNull(eventHandler.peekEvent());
253     }
254 }
```

```java
package org.example.simulator;

import org.example.data.Message;
import org.example.data.PacketBuilder;
import org.example.network.Channel;
import org.example.protocol.ClassicTCP;
import org.example.protocol.TCP;
import org.example.simulator.events.ChannelEvent;
import org.example.simulator.events.Event;
import org.example.simulator.events.RouteEvent;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.simulator.events.tcp.TCPRetransmitEventGenerator;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Test;

import java.util.PriorityQueue;
import java.util.Queue;

public class EventTest {

    private Queue<Event> events;
    private ClassicTCP tcp;
    private ClassicTCP host;

    public void connect(TCP linkedClient, TCP linkedServer) {
        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(linkedClient, linkedServer));
        eventHandler.run();
        Assert.assertTrue(linkedClient.isConnected());
        Assert.assertTrue(linkedServer.isConnected());
    }

    @Test(expected = IllegalArgumentException.class)
    public void expectIllegalArgumentExceptionIfNullChannelGiven() {
        new ChannelEvent(null);
    }

    @Test(expected = IllegalArgumentException.class)
    public void expectIllegalArgumentExceptionIfNullNodeGiven() {
        new RunTCPEvent(null);
    }

    @Test
    public void routeEventEqualsTest() {
        Event event1 = new RouteEvent(new ClassicTCP.ClassicTCPBuilder().build(),
            new PacketBuilder().build());
        Event event2 = new RouteEvent(new ClassicTCP.ClassicTCPBuilder().build(),
            new PacketBuilder().build());
        Assert.assertNotEquals(event1, event2);
    }


    @Test
    public void EventsOccurInAPreMatchedSequence() {
        int numberOfRuns = 100;
        for (int i = 0; i < numberOfRuns; i++) {
            Util.resetTime();
            Util.setSeed(1337);
            this.events = new PriorityQueue<>();
            this.tcp = new ClassicTCP.ClassicTCPBuilder().
```

```
                withReceivingWindowCapacity(7).build();
61          this.host = new ClassicTCP.ClassicTCPBuilder().
                withReceivingWindowCapacity(7).build();
62
63          new Channel.ChannelBuilder().build(this.tcp, this.host);
64
65          this.tcp.updateRoutingTable();
66          this.host.updateRoutingTable();
67          this.connect(this.tcp, this.host);
68
69          this.tcp.send(new Message("hello"));
70          this.events.add(new RunTCPEvent(this.tcp));
71
72          String debugString = "Iteration: " + (i + 1);
73
74          Event curEvent = this.events.poll();
75          Assert.assertNotNull(curEvent);
76          Assert.assertEquals(debugString, RunTCPEvent.class, curEvent.getClass())
                ;
77          curEvent.run();
78          curEvent.generateNextEvent(this.events);
79
80          curEvent = this.events.poll();
81          Assert.assertNotNull(curEvent);
82          Assert.assertEquals(debugString, ChannelEvent.class, curEvent.getClass()
                );
83          curEvent.run();
84          curEvent.generateNextEvent(this.events);
85
86          curEvent = this.events.poll();
87          Assert.assertNotNull(curEvent);
88          Assert.assertEquals(debugString, RunTCPEvent.class, curEvent.getClass())
                ;
89          curEvent.run();
90          curEvent.generateNextEvent(this.events);
91
92          curEvent = this.events.poll();
93          Assert.assertNotNull(curEvent);
94          Assert.assertEquals(debugString, ChannelEvent.class, curEvent.getClass()
                );
95          curEvent.run();
96          curEvent.generateNextEvent(this.events);
97
98          curEvent = this.events.poll();
99          Assert.assertNotNull(curEvent);
100         Assert.assertEquals(debugString, RunTCPEvent.class, curEvent.getClass())
                ;
101         curEvent.run();
102         curEvent.generateNextEvent(this.events);
103
104         curEvent = this.events.poll();
105         Assert.assertNotNull(curEvent);
106         Assert.assertEquals(debugString, TCPRetransmitEventGenerator.class,
                curEvent.getClass());
107         curEvent.run();
108         curEvent.generateNextEvent(this.events);
109
110         Assert.assertTrue(debugString, this.tcp.outputBufferIsEmpty());
111         Assert.assertTrue(debugString, this.host.outputBufferIsEmpty());
112         Assert.assertTrue(debugString, this.tcp.inputBufferIsEmpty());
113         Assert.assertTrue(debugString, this.host.inputBufferIsEmpty());
114         Assert.assertEquals(debugString, 0, this.events.size());
115     }
```

```
116         }
117
118
119 }
```

```java
package org.example.simulator.statistics;

import org.example.data.Message;
import org.example.data.Packet;
import org.example.network.Channel;
import org.example.network.Routable;
import org.example.network.Router;
import org.example.network.address.SimpleAddress;
import org.example.protocol.ClassicTCP;
import org.example.protocol.MPTCP;
import org.example.simulator.EventHandler;
import org.example.simulator.events.tcp.RunTCPEvent;
import org.example.simulator.events.tcp.TCPConnectEvent;
import org.example.util.Util;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class TCPStatsTest {

    @Before
    public void setup() {
        Util.resetTime();
        Util.setSeed(1);
    }

    @Test
    public void floodWithPacketsInBigCongestedNetworkShouldWorkTest() {
        ClassicTCP client = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(10).setReno().withAddress(new SimpleAddress(
            "Client")).build();
        Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.91).
            withAddress(new SimpleAddress("Router 1")).build();
        Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.91).
            withAddress(new SimpleAddress("Router 2")).build();
        Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.91).
            withAddress(new SimpleAddress("Router 3")).build();
        Routable r4 = new Router.RouterBuilder().withAverageQueueUtilization(0.91).
            withAddress(new SimpleAddress("Router 4")).build();
        ClassicTCP server = new ClassicTCP.ClassicTCPBuilder().
            withReceivingWindowCapacity(30).setReno().withAddress(new SimpleAddress(
            "Server")).build();

        new Channel.ChannelBuilder().build(client, r1);
        new Channel.ChannelBuilder().build(r1, r2);
        new Channel.ChannelBuilder().withLoss(0.01).build(r2, r3);
        new Channel.ChannelBuilder().build(r3, r4);
        new Channel.ChannelBuilder().build(r4, server);


        client.updateRoutingTable();
        r1.updateRoutingTable();
        r2.updateRoutingTable();
        r3.updateRoutingTable();
        r4.updateRoutingTable();
        server.updateRoutingTable();

        EventHandler eventHandler = new EventHandler();
        eventHandler.addEvent(new TCPConnectEvent(client, server));
        eventHandler.run();

        Assert.assertTrue(client.isConnected());
```

```
55          Assert.assertTrue(server.isConnected());
56
57          System.out.println("connected");
58
59          Assert.assertTrue(client.inputBufferIsEmpty());
60          Assert.assertTrue(server.inputBufferIsEmpty());
61          Assert.assertTrue(client.outputBufferIsEmpty());
62          Assert.assertTrue(server.outputBufferIsEmpty());
63
64          int numPacketsToSend = 10000;
65          for (int i = 1; i <= numPacketsToSend; i++) {
66              Message msg = new Message("test " + i);
67              client.send(msg);
68          }
69
70          eventHandler.addEvent(new RunTCPEvent(client));
71          eventHandler.run();
72
73          Assert.assertTrue(client.inputBufferIsEmpty());
74          Assert.assertTrue(server.inputBufferIsEmpty());
75          Assert.assertTrue(client.outputBufferIsEmpty());
76          Assert.assertTrue(server.outputBufferIsEmpty());
77
78          for (int i = 1; i <= numPacketsToSend; i++) {
79              Message msg = new Message("test " + i);
80              Packet received = server.receive();
81              Assert.assertNotNull(received);
82              Assert.assertEquals(msg, received.getPayload());
83          }
84          TCPStats stat = server.getStats();
85          System.out.println(stat);
86          stat.createArrivalChart();
87          stat.createDepartureChart();
88          stat.createInterArrivalChart();
89          stat.createTimeInSystemChart();
90          stat.createNumberOfPacketsInSystemChart();
91
92      }
93
94      @Test
95      public void MPTCPFloodWithPacketsInOrderShouldWorkTest() {
96          MPTCP client = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(60).withAddress(new SimpleAddress("MPTCP-
                Client")).build();
97          Routable r1 = new Router.RouterBuilder().withAverageQueueUtilization(0.94).
                withAddress(new SimpleAddress("A")).build();
98          Routable r2 = new Router.RouterBuilder().withAverageQueueUtilization(0.94).
                withAddress(new SimpleAddress("B")).build();
99          Routable r3 = new Router.RouterBuilder().withAverageQueueUtilization(0.9).
                withAddress(new SimpleAddress("C")).build();
100         MPTCP server = new MPTCP.MPTCPBuilder().withNumberOfSubflows(3).
                withReceivingWindowCapacity(60).withAddress(new SimpleAddress("MPTCP-
                Server")).build();
101
102         //path one
103         new Channel.ChannelBuilder().build(client, r1);
104         new Channel.ChannelBuilder().withLoss(0.015).build(r1, server);
105
106         //path two
107         new Channel.ChannelBuilder().build(client, r2);
108         new Channel.ChannelBuilder().withLoss(0.015).build(r2, server);
109
110         //path three
```

```
111         new Channel.ChannelBuilder().build(client, r3);
112         new Channel.ChannelBuilder().withLoss(0.005).build(r3, server);
113
114         client.updateRoutingTable();
115         r1.updateRoutingTable();
116         r2.updateRoutingTable();
117         r3.updateRoutingTable();
118         server.updateRoutingTable();
119
120         EventHandler eventHandler = new EventHandler();
121         eventHandler.addEvent(new TCPConnectEvent(client, server));
122         eventHandler.run();
123         System.out.println("connected");
124
125         Assert.assertTrue(client.getSubflows()[0].isConnected());
126         Assert.assertTrue(client.getSubflows()[1].isConnected());
127         Assert.assertTrue(client.getSubflows()[2].isConnected());
128
129         int numPacketsToSend = 10000;
130
131         for (int i = 1; i <= numPacketsToSend; i++) {
132             Message msg = new Message("test " + i);
133             client.send(msg);
134         }
135         eventHandler.addEvent(new RunTCPEvent(client));
136         eventHandler.run();
137
138         Assert.assertTrue("client still has packets to send", client.
                outputBufferIsEmpty());
139         Assert.assertTrue(server.outputBufferIsEmpty());
140         Assert.assertTrue(client.inputBufferIsEmpty());
141         Assert.assertTrue(server.inputBufferIsEmpty());
142         Assert.assertTrue(r1.inputBufferIsEmpty());
143         Assert.assertTrue(r2.inputBufferIsEmpty());
144         Assert.assertTrue(r3.inputBufferIsEmpty());
145
146
147         for (int i = 1; i <= numPacketsToSend; i++) {
148             Message msg = new Message("test " + i);
149             Packet received = server.receive();
150             Assert.assertNotNull(received);
151             Assert.assertEquals("iteration " + i, received.getPayload(), msg);
152         }
153         Assert.assertNull(server.receive());
154
155         //receiver
156         for (TCPStats stat : server.getTcpStats()) {
157             System.out.println(stat.toString());
158             stat.createArrivalChart();
159             stat.createDepartureChart();
160             stat.createInterArrivalChart();
161             stat.createTimeInSystemChart();
162             stat.createNumberOfPacketsInSystemChart();
163         }
164
165         //sender
166         for (TCPStats stat : client.getTcpStats()) {
167             System.out.println(stat.toString());
168             stat.createCWNDChart();
169         }
170
171     }
172
```

```
173 }
```

```java
package org.example.util;

import org.example.data.Packet;
import org.example.data.PacketBuilder;
import org.junit.Assert;
import org.junit.Test;

public class BoundedPriorityBlockingQueueTest {


    @Test
    public void bothConstructorsWorkTest() {
        BoundedQueue bq1 = new BoundedPriorityBlockingQueue(10);
        BoundedQueue<Packet> bq2 = new BoundedPriorityBlockingQueue<>(10, (packet,
            t1) -> packet.getSequenceNumber() - t1.getSequenceNumber());
        Assert.assertTrue(bq1 instanceof BoundedPriorityBlockingQueue);
        Assert.assertTrue(bq2 instanceof BoundedPriorityBlockingQueue);
    }


    @Test
    public void offerInsertsOrdered() {
        BoundedPriorityBlockingQueue<Packet> boundedPriorityBlockingQueue =
                new BoundedPriorityBlockingQueue<>(5, (packet, t1) -> packet.
                    getSequenceNumber() - t1.getSequenceNumber());

        Packet one = new PacketBuilder()
                .withSequenceNumber(1)
                .build();
        Packet two = new PacketBuilder()
                .withSequenceNumber(2)
                .build();
        Packet three = new PacketBuilder()
                .withSequenceNumber(3)
                .build();
        Packet four = new PacketBuilder()
                .withSequenceNumber(4)
                .build();
        Packet five = new PacketBuilder()
                .withSequenceNumber(5)
                .build();

        boundedPriorityBlockingQueue.offer(three);
        boundedPriorityBlockingQueue.offer(four);
        boundedPriorityBlockingQueue.offer(one);
        boundedPriorityBlockingQueue.offer(five);
        boundedPriorityBlockingQueue.offer(two);

        Assert.assertEquals(one, boundedPriorityBlockingQueue.poll());
        Assert.assertEquals(two, boundedPriorityBlockingQueue.poll());
        Assert.assertEquals(three, boundedPriorityBlockingQueue.poll());
        Assert.assertEquals(four, boundedPriorityBlockingQueue.poll());
        Assert.assertEquals(five, boundedPriorityBlockingQueue.poll());
        Assert.assertEquals(null, boundedPriorityBlockingQueue.poll());
    }

    @Test
    public void isFullWorksReturnsTrueIfFullTest() {
        BoundedPriorityBlockingQueue<Packet> boundedPriorityBlockingQueue =
                new BoundedPriorityBlockingQueue<>(5, (packet, t1) -> packet.
                    getSequenceNumber() - t1.getSequenceNumber());

```

```
60          Packet one = new PacketBuilder()
61                  .withSequenceNumber(1)
62                  .build();
63          Packet two = new PacketBuilder()
64                  .withSequenceNumber(2)
65                  .build();
66          Packet three = new PacketBuilder()
67                  .withSequenceNumber(3)
68                  .build();
69          Packet four = new PacketBuilder()
70                  .withSequenceNumber(4)
71                  .build();
72          Packet five = new PacketBuilder()
73                  .withSequenceNumber(5)
74                  .build();
75
76          boundedPriorityBlockingQueue.offer(one);
77          boundedPriorityBlockingQueue.offer(two);
78          boundedPriorityBlockingQueue.offer(three);
79          boundedPriorityBlockingQueue.offer(four);
80          boundedPriorityBlockingQueue.offer(five);
81
82          Assert.assertTrue(boundedPriorityBlockingQueue.isFull());
83      }
84
85
86      @Test
87      public void isFullWorksReturnsFalseIfNotFullTest() {
88          BoundedPriorityBlockingQueue<Packet> boundedPriorityBlockingQueue =
89                  new BoundedPriorityBlockingQueue<>(5, (packet, t1) -> packet.
                        getSequenceNumber() - t1.getSequenceNumber());
90
91          Packet one = new PacketBuilder()
92                  .withSequenceNumber(1)
93                  .build();
94          Packet two = new PacketBuilder()
95                  .withSequenceNumber(2)
96                  .build();
97          Packet three = new PacketBuilder()
98                  .withSequenceNumber(3)
99                  .build();
100         Packet four = new PacketBuilder()
101                 .withSequenceNumber(4)
102                 .build();
103
104         boundedPriorityBlockingQueue.offer(one);
105         boundedPriorityBlockingQueue.offer(two);
106         boundedPriorityBlockingQueue.offer(three);
107         boundedPriorityBlockingQueue.offer(four);
108
109         Assert.assertFalse(boundedPriorityBlockingQueue.isFull());
110     }
111
112
113     @Test
114     public void resizeWorksTest() {
115         BoundedPriorityBlockingQueue<Packet> boundedPriorityBlockingQueue =
116                 new BoundedPriorityBlockingQueue<>(5, (packet, t1) -> packet.
                        getSequenceNumber() - t1.getSequenceNumber());
117
118         boundedPriorityBlockingQueue.setBound(10);
119         Assert.assertEquals(10, boundedPriorityBlockingQueue.bound());
120     }
```

```
121
122     @Test
123     public void resizeAllowsMoreElementsInQueue() {
124         BoundedPriorityBlockingQueue<Packet> boundedPriorityBlockingQueue =
125                 new BoundedPriorityBlockingQueue<>(4, (packet, t1) -> packet.
                        getSequenceNumber() - t1.getSequenceNumber());
126         Packet one = new PacketBuilder()
127                 .withSequenceNumber(1)
128                 .build();
129         Packet two = new PacketBuilder()
130                 .withSequenceNumber(2)
131                 .build();
132         Packet three = new PacketBuilder()
133                 .withSequenceNumber(3)
134                 .build();
135         Packet four = new PacketBuilder()
136                 .withSequenceNumber(4)
137                 .build();
138
139         boundedPriorityBlockingQueue.offer(one);
140         boundedPriorityBlockingQueue.offer(two);
141         boundedPriorityBlockingQueue.offer(three);
142         boundedPriorityBlockingQueue.offer(four);
143
144         Assert.assertTrue(boundedPriorityBlockingQueue.isFull());
145
146         boundedPriorityBlockingQueue.setBound(8);
147         Assert.assertEquals(8, boundedPriorityBlockingQueue.bound());
148
149
150         Packet five = new PacketBuilder()
151                 .withSequenceNumber(5)
152                 .build();
153         Packet six = new PacketBuilder()
154                 .withSequenceNumber(6)
155                 .build();
156         Packet seven = new PacketBuilder()
157                 .withSequenceNumber(7)
158                 .build();
159         Packet eight = new PacketBuilder()
160                 .withSequenceNumber(8)
161                 .build();
162
163         boundedPriorityBlockingQueue.offer(five);
164         boundedPriorityBlockingQueue.offer(six);
165         boundedPriorityBlockingQueue.offer(seven);
166         boundedPriorityBlockingQueue.offer(eight);
167
168         Assert.assertTrue(boundedPriorityBlockingQueue.isFull());
169
170
171         Packet nine = new PacketBuilder()
172                 .withSequenceNumber(9)
173                 .build();
174         boundedPriorityBlockingQueue.offer(nine);
175
176         Assert.assertTrue(boundedPriorityBlockingQueue.isFull());
177         Assert.assertEquals(8, boundedPriorityBlockingQueue.size());
178     }
179
180 }
```

```java
package org.example.util;

import org.junit.Assert;
import org.junit.Test;

import java.util.ArrayList;

public class UtilTest {


    @Test
    public void getGaussianIsStable() {
        ArrayList<Double> doubles = new ArrayList<>();
        Util.setSeed(1337);

        int arraySize = 10000;
        for (int i = 0; i < arraySize; i++) {
            doubles.add(Util.getNextGaussian());
        }

        Assert.assertEquals(doubles.size(), arraySize);

        Util.setSeed(1337);
        for (double gaussian : doubles) {
            Assert.assertEquals(gaussian, Util.getNextGaussian(), 0);
        }
    }


    @Test
    public void getNextRandomIsStable() {
        ArrayList<Integer> ints = new ArrayList<>();
        Util.setSeed(1337);

        int arraySize = 10000;
        for (int i = 0; i < arraySize; i++) {
            ints.add(Util.getNextRandomInt());
        }

        Assert.assertEquals(ints.size(), arraySize);

        Util.setSeed(1337);
        for (double rand : ints) {
            Assert.assertEquals(rand, Util.getNextRandomInt(), 0);
        }
    }


    @Test
    public void getNextRandomWithBoundIsStable() {
        ArrayList<Integer> ints = new ArrayList<>();
        int bound = 100;
        Util.setSeed(1337);

        int arraySize = 10000;
        for (int i = 0; i < arraySize; i++) {
            ints.add(Util.getNextRandomInt(bound));
        }

        Assert.assertEquals(ints.size(), arraySize);

        Util.setSeed(1337);
```

```
63          for (double rand : ints) {
64              Assert.assertEquals(rand, Util.getNextRandomInt(bound), 0);
65          }
66      }
67
68
69      @Test
70      public void allIsStable() {
71          ArrayList<Integer> ints = new ArrayList<>();
72          ArrayList<Integer> boundedInts = new ArrayList<>();
73          ArrayList<Double> doubles = new ArrayList<>();
74          final int bound = 100;
75          Util.setSeed(1337);
76
77          int arraySize = 10000;
78          for (int i = 0; i < arraySize; i++) {
79              ints.add(Util.getNextRandomInt());
80              boundedInts.add(Util.getNextRandomInt(bound));
81              doubles.add(Util.getNextGaussian());
82          }
83
84          Assert.assertEquals(ints.size(), arraySize);
85          Assert.assertEquals(boundedInts.size(), arraySize);
86          Assert.assertEquals(doubles.size(), arraySize);
87
88          Util.setSeed(1337);
89          for (int i = 0; i < arraySize; i++) {
90              Assert.assertEquals(ints.get(i), Util.getNextRandomInt(), 0);
91              Assert.assertEquals(boundedInts.get(i), Util.getNextRandomInt(bound), 0)
                    ;
92              Assert.assertEquals(doubles.get(i), Util.getNextGaussian(), 0);
93          }
94      }
95
96
97  }
```