Middle East Technical University

Department of Computer Engineering

CENG 435

Data Communications and Networking

Assignment 1 - Submission

Yarkın Özcan
ozcan.yarkin@metu.edu.tr

Ahmet Mürtezaoğlu
ahmet.murtezaoglu@metu.edu.tr

November 3, 2025

1 Introduction

This report is authored by Yarkın Özcan and Ahmet Mürtezaoğlu. Each section explains the answers to the questions using OmNeT++ graphs and the codes used to generate them. The following subsections describe the detailed process of creating this report.

1.1 Experiment Setup

The experiments were conducted using the **OMNeT++ 6.0.1** simulation environment together with the **INET 4.5.4** framework. The network models and configurations were implemented using `.ned` topology files and `.ini` configuration files, while the simulation results were collected in scalar and vector CSV outputs for further analysis with Python. We have simulated our assigned tasks first in OMNeT, then in order to see the simulation results in .sca and .vec format, we have used the .anf files. After going into .anf files, we clicked at browse_data section and filter the result that we want.

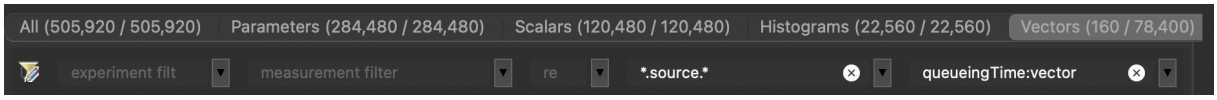

Figure 1: Result Visualization

After that, we selected all related .vec or .sca (depending on the question) and exported it as a .csv file in order to feed it easily to our python script to plot our data.

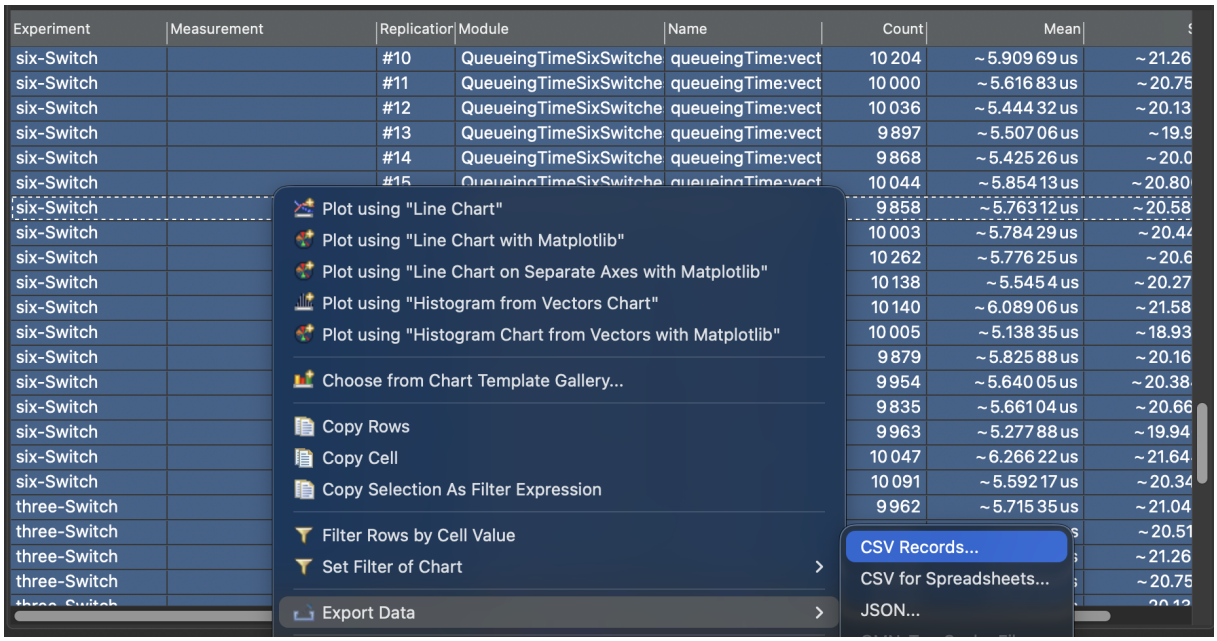

Figure 2: Export Data

As the last step, we have written python scripts for each different question to plot our data and then feed the corresponding .csv file to that python code.

1.2 Author Contribution

Both *Yarkın* and *Ahmet* installed and configured OmNeT++ on their own computers and performed the measurements. Each person wrote the section of the report corresponding to the graphs they generated. *Ahmet Mürtezaoğlu* measured and reported the first three question: Channel Throughput, Channel Utilization, and End-to-End Delay. *Yarkın Özcan* measured and reported Packet Delay Variation, Transmission Time, Propagation Time, and Queuing Time.

2 Channel Throughput

2.1 Question 1

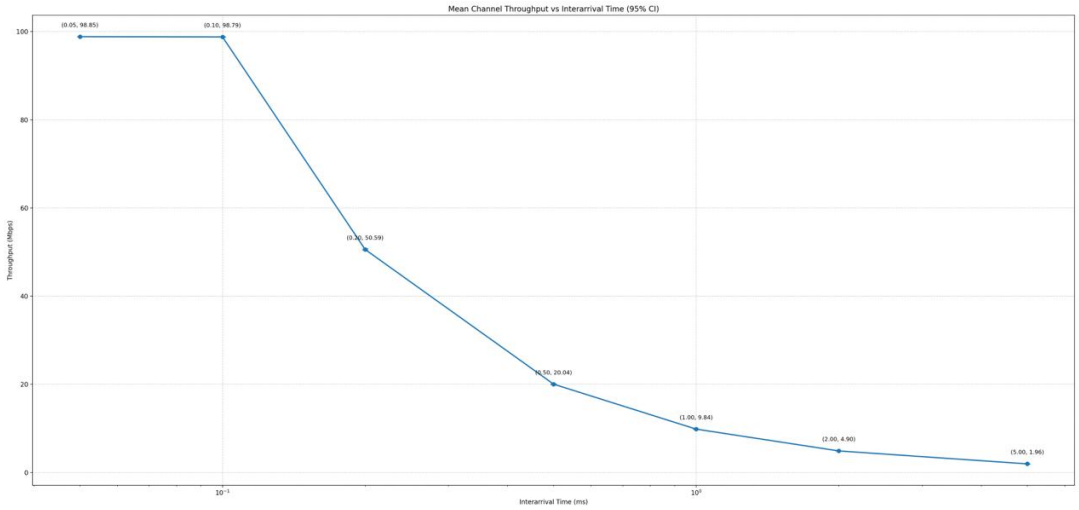

Figure 3: Mean Channel Throughput Depending On Interarrival Time

Before analyzing the graph, let's first explain how it was constructed by examining the code and use this code as a reference for the rest of the report.

```
[General]
network = ChannelThroughputMeasurementShowcase
description = "Measure throughput between source and destination"
repeat = 20
sim-time-limit = 5s
# source application with roughly ~48Mbps throughput
*.source.numApps = 1
*.source.app[0].typename = "UdpSourceApp"
*.source.app[0].source.packetLength = 1200B
*.source.app[0].source.productionInterval = exponential(${interval =
↪  0.05ms, 0.1ms, 0.2ms, 0.5ms, 1ms, 2ms})
*.source.app[0].io.destAddress = "destination"
*.source.app[0].io.destPort = 1000
# destination application
*.destination.numApps = 1
*.destination.app[0].typename = "UdpSinkApp"
*.destination.app[0].io.localPort = 1000
# enable modular Ethernet model
*.*.ethernet.typename = "EthernetLayer"
*.*.eth[*].typename = "LayeredEthernetInterface"
*.*.eth[*].bitrate = 100Mbps
```

Listing 1: Simulation configuration for throughput measurement

To obtain accurate results, we repeated the simulation 20 times and used a reasonable simulation time limit of 5 seconds. For the production interval values, we selected ranges that would produce a clear and observable graph for analyzing saturation or changes in throughput. Now let's answer the question.

Based on the Figure 3, there is a obvious inverse relationship between the interarrival time and the mean channel throughput, up to a certain point:

1. When you **increase** the interarrival time: The channel throughput **decreases**. This is because packets are being sent less frequently, leading to lower overall data transfer. For example, as the iat increases from 0.20 ms to 1.00 ms, the throughput drops significantly from 50.59 Mbps to 9.84 Mbps.

2. When you **decrease** the interarrival time: The channel throughput **increases**. This is because packets are being sent more frequently, filling up the channel's capacity.For example, we can consider the inverse of the previous case: decreasing the interarrival time from 1.00 ms to 0.20 ms causes the throughput to increase from 9.84 Mbps to 50.59 Mbps.

2.2 Question 2

For this question, we can refer to the same graph constructed for the first question. As shown in Figure 3, the channel capacity saturates for interarrival time values at or below **0.10 ms**. Here's the evidence from the graph:

As the interarrival time decreases from 0.20 ms (56.59 Mbps) to 0.10 ms, the throughput nearly doubles to 98.79 Mbps. However, when the interarrival time is decreased further from 0.10 ms to 0.05 ms, the throughput remains almost unchanged, increasing only slightly from 98.79 Mbps to 98.85 Mbps.

The graph saturates because for any interarrival time of 0.10 ms or less, the offered load (96 Mbps or more) meets or exceeds the channel's 100 Mbps capacity. The channel becomes the **bottleneck**, and the throughput is capped at the channel's maximum speed. The measured throughput of $\approx$ 98.8 Mbps represents the 100 Mbps link capacity.

2.3 Question 3

Based on the Figure 4 , the relationship between Ethernet bitrate and channel throughput has **two distinct** behaviors:

1. When you **increase** the Ethernet bitrate in **1 Mbps to 10 Mbps** range, the channel throughput increases at an almost **1-to-1 rate**. For example, at a 5 Mbps bitrate, the throughput is 4.85 Mbps. When you increase the bitrate to 8 Mbps, the throughput follows, rising to 7.76 Mbps. In this phase, Channel's throughput is limited directly by its capacity.

2. **Saturation Phase**, When you increase the Ethernet bitrate beyond **10 Mbps**, the channel throughput does not increase. The graph shows that at 10 Mbps, the throughput is 9.66 Mbps. When the bitrate is doubled to 20 Mbps, the throughput

only nudges up to 9.86 Mbps. Further increases to 50 Mbps (9.91 Mbps) and 100 Mbps (9.88 Mbps) show no meaningful change.

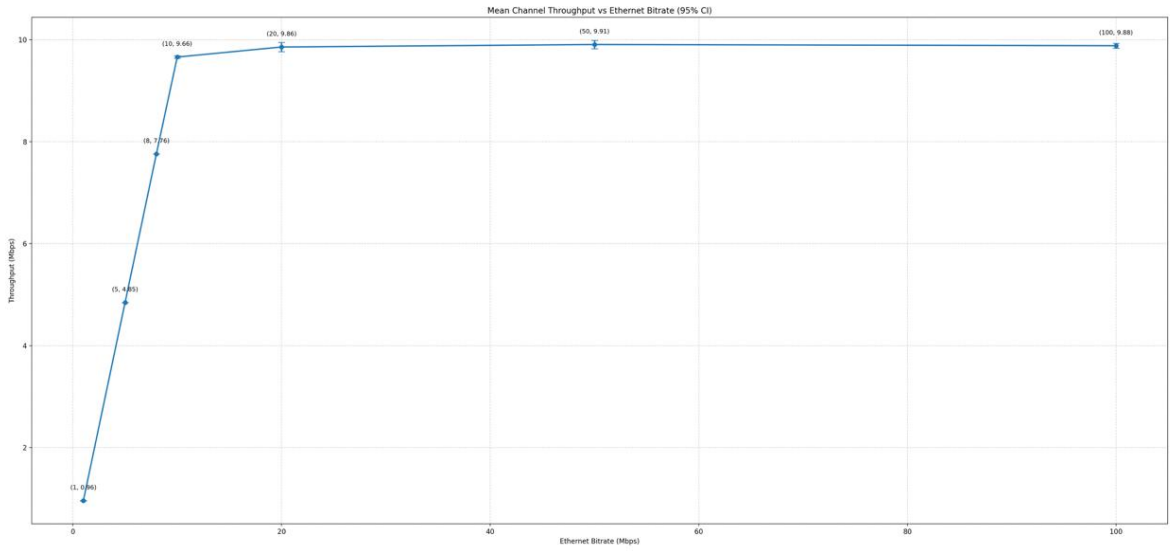

Figure 4: Mean Channel Throughput Depending On Ethernet Bitrate

At this point, we can examine the code to understand how the graph was constructed. We have already generated a throughput graph using the code in listing 1. Now, it is sufficient to make only the following changes, which keep the interarrival time constant and modify the Ethernet bitrate according to the parameter set values shown below.

```
[General]
.
.
.
*.source.app[0].source.productionInterval = exponential(1ms)
.
.
.
# data rate of all network interfaces
*.*.eth[*].bitrate = ${bitrate_mbps = 1, 5, 8, 10, 20, 50, 100}Mbps
*.destination.app[0].throughput.result-recording-modes = +scalar
```

3 Channel Utilization

3.1 Question 1

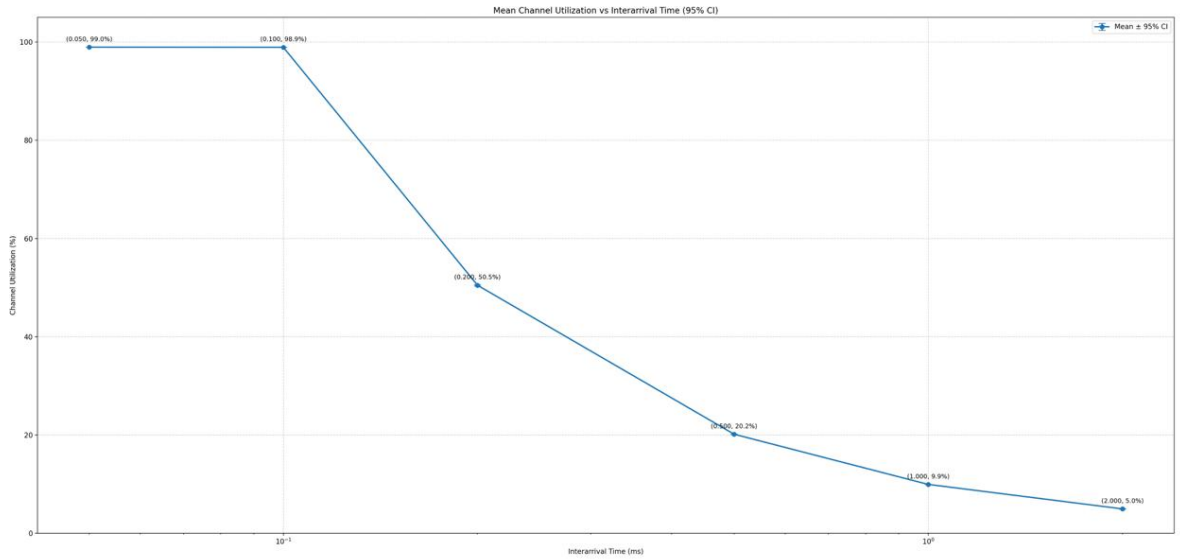

Figure 5: Mean Channel Utilization Depending On Ethernet Bitrate

Based on the Figure 6 , the channel utilization has a clear **inverse relationship** with the interarrival time of the packets:

1. When you **increase** the interarrival time: The channel utilization **decreases**. This is because packets are being sent less frequently, leaving the channel idle for longer periods. For example, the graph shows that increasing the iat from 0.200 ms to 1.000 ms causes the utilization to drop sharply from $50.5\% to 9.9\%$.

2. When you **decrease** the interarrival time: The channel utilization **increases**. This is because packets are being sent more frequently, keeping the channel busy. For example, decreasing the iat from 0.500 ms to 0.200 ms causes the utilization to jump from 20.2% to 50.5%.

The graph also shows that the channel reaches its maximum utilization (saturation) at 99% when the iat is 0.100 ms. Decreasing the iat even further to 0.050 ms does not increase the utilization, as the channel is already as busy as it can be.

3.2 Question 2

To be able to answer this question we need to look at our configuration code and our code is almost same with listing 1. Only changing network variable to corresponding NED network is enough.

```
[General]
```

```
network = ChannelUtilizationMeasurementShowcase
.
.
sim-time-limit = 10s
.
.
*.source.app[0].source.packetLength = 1200B
.
.
*.*.eth[*].bitrate = 100Mbps
```

The link saturates for any **offered load of 96 Mbps or greater**.
Here is the breakdown of the calculation:

1. **Saturation Point:** As Figure 6 shows, and as we found in Section 2.2, the link becomes saturated at an interarrival time of **0.10 ms**. Decreasing the `iat` further does not increase throughput or utilization.

2. **Packet Size:** In the configuration file packet length is: `*.source.app[0].source.packetLength = 1200B`. To use this in our calculation, we convert Bytes to bits:

$$1200 \text{ Bytes} \times 8 \text{ bits} = 9600 \text{ bits}$$

3. **Calculate Offered Load:** The offered load is the packet size divided by the interarrival time. Using the saturation `iat` of 0.10 ms (or 0.0001 s):

$$\text{Offered Load} = \frac{\text{Packet Size}}{\text{Interarrival Time}} = \frac{9600 \text{ bits}}{0.0001 \text{ s}} = 96,000,000 \text{ bps}$$

   This is equal to **96 Mbps**.

Therefore, any offered load of 96 Mbps or more will saturate the link.

3.3 Question 3

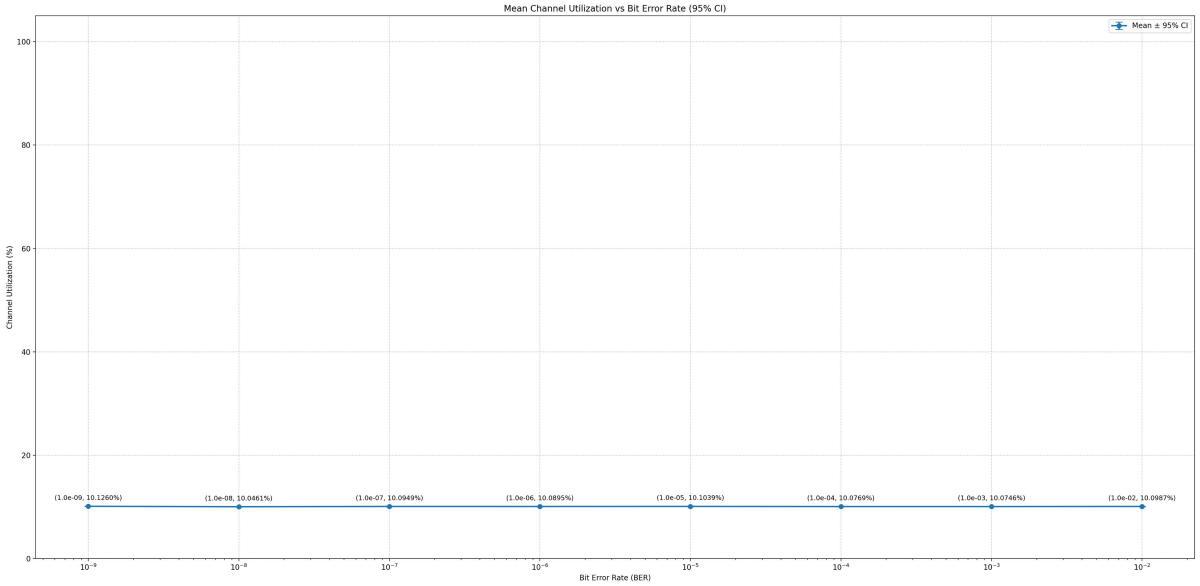

Figure 6: Mean Channel Utilization Depending On BER

```
[General]
network = ChannelUtilizationMeasurementShowcase
repeat = 20
sim-time-limit = 10s
.
.
*.source.app[0].typename = "UdpSourceApp"
*.source.app[0].source.packetLength = 1200B # larger frames -> more
↪  likely to be corrupted
*.source.app[0].source.productionInterval = exponential(0.1ms) # higher
↪  offered load
.
.
*.destination.app[0].typename = "UdpSinkApp"
.
.
# Sweep over BER values
**.channel.typename = "MyEthernetLink"
**.channel.ber = ${ber_rate = 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3,
↪  1e-2}
```

Based on Figure 6, the Mean Channel Utilization remains constant at approximately **10.1%** across the entire range of Bit Error Rate (BER) values, from a highly reliable 10^{-9} to a unreliable 10^{-2}.

This flat line is the expected behavior and is a direct consequence of using the **UDP protocol**, which has no error recovery or flow control mechanisms. The behavior can be explained by a few key factors:

- **Constant Offered Load:** The `UdpSourceApp` is configured to send packets at a fixed rate (an `exponential(0.1ms)` interval, as seen in the configuration). This is a "fire-and-forget" protocol; it does not wait for acknowledgments before sending the next packet.
- **No Error Recovery:** When a high BER (e.g., 10^{-3}) causes bit errors in a packet, the receiving Ethernet layer detects this corruption and simply discards the corrupt frame or warns the receiver.
- **No Feedback to Sender:** Because UDP is a connectionless protocol, the receiver does not send any message (like an ACK or NACK) back to the sender to report that packets are being lost. The sender is completely unaware of the deteriorating channel conditions.

Since the sender receives no feedback, its behavior never changes. It continues to transmit packets at its fixed, pre-configured rate, which corresponds to the $\approx 10.1\%$ utilization seen in the graph. sender's activity. The sender is 10.1% busy transmitting, even if 100% of that data is corrupt and being discarded.

4 End-to-end Delay

4.1 Question 1

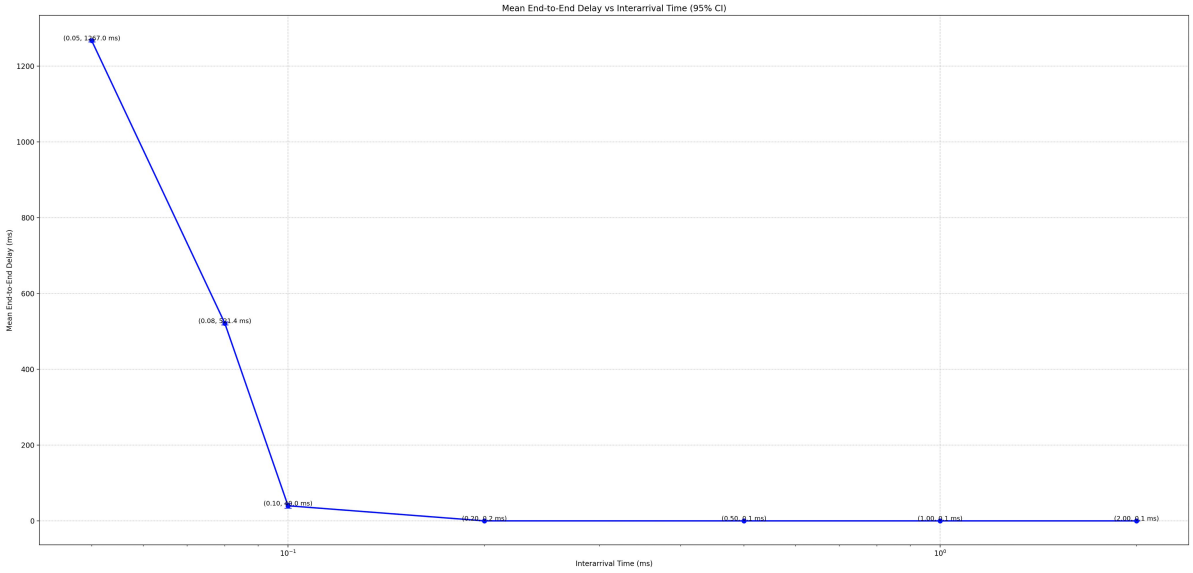

Figure 7: Mean End-to-End Delay Depending On Interarrival Time

Before answering the question, we can see that the configuration is similar to Listing 1, except for the NED network and the set of exponential interarrival time (`iat`) parameters.

```
[General]
network = EndToEndDelayMeasurementShowcase
description = "Measure packet end-to-end delay"
.
.

*.source.app[0].source.productionInterval = exponential(${iat_ms = 2, 1,
↪  0.5, 0.2, 0.1, 0.08, 0.05}ms)
.
.
.
```

Based on Figure 7, the relationship between interarrival time and mean end-to-end delay has two distinct behaviors: a stable low-delay region and an exponential high-delay region.

1. **Stable Region (iat $\geq$ 0.2 ms):** In this region, the offered load is low.
    - **Increasing** the `iat` (e.g., from 0.2 ms to 2.0 ms) has **almost no effect**. The delay remains at a stable, negligible minimum (around 0.1-0.2 ms).
    - **Decreasing** the `iat` (e.g., from 2.0 ms to 0.2 ms) also has **almost no effect**. The delay stays minimal because the offered load is still far below the channel's capacity, so no queue forms.
2. **Exponential Region (iat $\leq$ 0.1 ms):** In this region, the offered load approaches or exceeds the channel's capacity.
    - **Decreasing** the `iat` (e.g., from 0.08 ms to 0.05 ms) causes the mean delay to **increase exponentially**, from 521.4 ms to 1267 ms. This is because the channel is saturated, and the router's buffer is filling rapidly, leading to extreme **queuing delay**.
    - **Increasing** the `iat` (e.g., from 0.08 ms to 0.1 ms) causes the delay to **collapse dramatically** from 521.4 ms to 40.0 ms, as the offered load drops just below the saturation point and the queue is able to clear.

4.2 Question 2

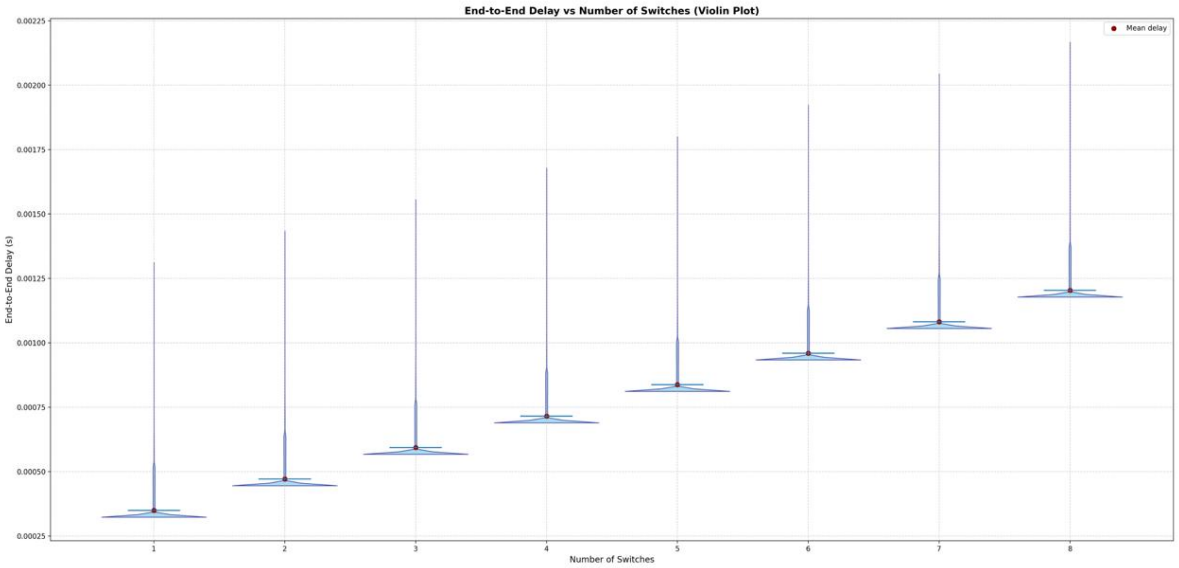

Figure 8: Mean End-to-End Delay Depending On Number of Switches

Based on Figure 8, the mean end-to-end delay **increases linearly** as the number of switches in the path increases from 1 to 8.

The violin plot provides several key insights into this relationship:

1. **Linear Increase in Mean Delay:** The mean delay, indicated by the red dot, steps up by a consistent amount with each added switch. This shows that each switch adds a predictable, fixed amount of processing, transmission, and queuing delay to the packet's total journey.

2. **Right Skewed Graph:** The violin plots are all right-skewed with a wide base at the bottom and a long, thin tail pointing upwards. This is the expected behavior and it is **caused by queuing delay**.

    - The **wide base** represents the minimum possible delay. Most packets arrive at a switch when it's idle and they are forwarded immediately.
    - The **long tail** represents the "unlucky" packets that arrive at a switch and find it busy processing another packet. These packets must wait in a queue, which adds a random, variable amount of queuing delay.

To generate the data for Figure 8, we couldn't use a single parameter set because the network itself had to change for each data point. Therefore, we manually ran the experiment eight separate times.

For each run, we defined a specific case in both the .ned and .ini files. As an example, let's look at the .ned file for the three-switch configuration.

```
network EndtoEndDelayThreeSwitches extends WiredNetworkBase
```

```
{
    submodules:
        source: StandardHost { @display("p=300,100"); }
        sw1: EthernetSwitch { @display("p=450,100"); }
        sw2: EthernetSwitch { @display("p=600,100"); }
        sw3: EthernetSwitch { @display("p=750,100"); }
        destination: StandardHost { @display("p=900,100"); }

    connections:
        source.ethg++ <--> Eth100M <--> sw1.ethg++;
        sw1.ethg++    <--> Eth1G <--> sw2.ethg++;
        sw2.ethg++    <--> Eth1G <--> sw3.ethg++;
        sw3.ethg++    <--> Eth100M <--> destination.ethg++;
}
```

Similarly, in our .ini file, we manually adjusted the link bitrates for each case and explicitly selected them before each run. As shown below, this is the configuration for the three-switch case.

```
[Config three-Switch]
extends = General
network = inet.showcases.measurement.endtoenddelay.EndToEndDelaySwitches

# Increased packet length slightly
*.source.app[0].source.packetLength = 2400B

# 1. Set the default link speed
..eth[*].bitrate = 100Mbps

# 2. Overrode the backbone links to 1Gbps
*.sw1.eth[1].bitrate = 1Gbps
*.sw2.eth[0].bitrate = 1Gbps
*.sw2.eth[1].bitrate = 1Gbps
*.sw3.eth[0].bitrate = 1Gbps

# 3. 100Mbps edge links
*.source.eth[0].bitrate = 100Mbps
*.sw1.eth[0].bitrate = 100Mbps
*.sw3.eth[1].bitrate = 100Mbps
*.destination.eth[0].bitrate = 100Mbps
```

4.3 Question 3

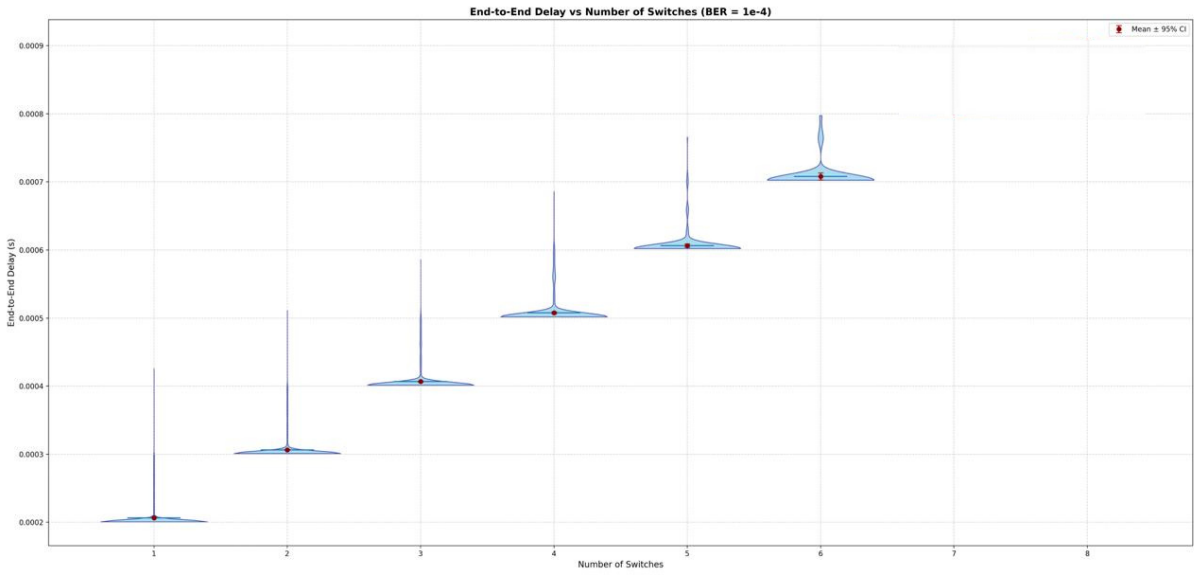

Figure 9: Mean End-to-End Delay vs Number of Switches (BER = 1e-4)

Based on Figure 9, the mean end-to-end delay **increases linearly** as the number of switches in the path increases from 1 to 8, even in the presence of BER. This can be observed from the red dots in the graph, which rise by a consistent amount with each additional switch. The mean delay is approximately 0.0002 s for 1 switch, increasing linearly to about 0.0008 s for 7 switches and 0.0009 s for 8 switches. This indicates that each switch adds a predictable and nearly constant amount of delay.

The violin plot provides similar key insights as in section 4.2 like **linear increase in mean delay and right skewed graph**. So let's focus on difference that BER causes.

The difference is "disappearance" of the violin plots for higher switch counts. This is not an error; it indicates that **almost no packets successfully reached the destination**. This packet loss is a direct result of the high Bit Error Rate (10^{-4}) combined with the increasing number of links. A BER of 10^{-4} means 1 in every 10,000 bits is corrupted. With a 2400-byte packet, it is statistically almost certain that every packet will have at least one-bit error on any given link and when a switch receives a packet it checks for the bit error. If it detects a bit error, it considers the packet corrupt and **drops it**. The packet is not forwarded.

To generate the data for Figure 9, we again had to run the experiment 8 separate times, as the network itself changed for each data point. For each run, we used a specific case in the `.ned` file. The 3-switch case, for example, defines a topology where all links are 100 Mbps.

```
network EndtoEndDelayThreeSwitches extends WiredNetworkBase
{
    submodules:
        source: StandardHost { @display("p=300,100"); }
```

```
        sw1: EthernetSwitch { @display("p=450,100"); }
        sw2: EthernetSwitch { @display("p=600,100"); }
        sw3: EthernetSwitch { @display("p=750,100"); }
        destination: StandardHost { @display("p=900,100"); }

    connections:
        source.ethg++ <--> Eth100M <--> sw1.ethg++;
        sw1.ethg++    <--> Eth100M <--> sw2.ethg++;
        sw2.ethg++    <--> Eth100M <--> sw3.ethg++;
        sw3.ethg++    <--> Eth100M <--> destination.ethg++;
}
```

In our `omnetpp.ini` file, we first set the global parameters for this experiment, applying a **100 Mbps bitrate** and a **BER of 1e-4** to all channels.

```
# data rate of all network interfaces
*.*.eth[*].bitrate = 100Mbps
**.channel.errorModelType = "ber"
**.channel.ber = 1e-4
```

Then, we created 8 separate configuration blocks, one for each `.ned` file, to run the simulations. The configuration for the 3-switch case simply extends the general config and points to the correct network.

```
[Config three-Switch]
extends = General
network =
↪  inet.showcases.measurement.endtoenddelay.EndToEndDelayThreeSwitches
*.*.eth[*].bitrate = 100Mbps
```

5 Packet Delay Variation

For the questions related to Packet Delay Variation, we have observed that the OMNeT actually applies the interarrival jitter calculations as outlined in RFC 1889. So in order to observe the packet delay variation, we have used packetJitter:vector that is generated out of box by OMNeT.

5.1 Question 1

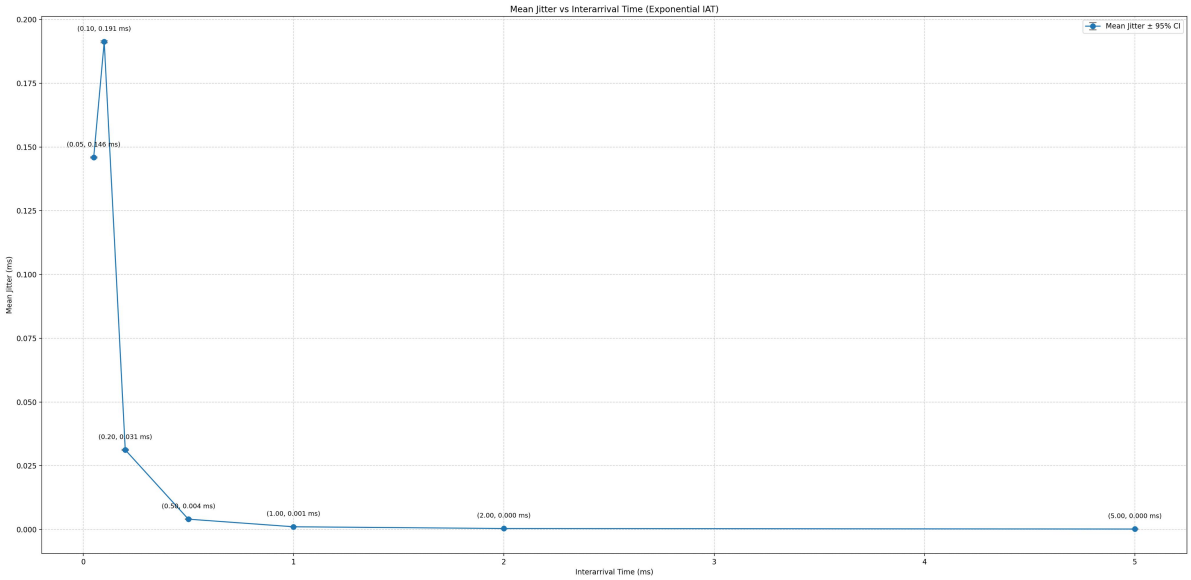

Figure 10: Mean Jitter Depending On Interarrival Time

Based on the Figure 10, the relationship between mean jitter and interarrival time is complex and can be broken into three distinct regions:

- **iat $\geq$ 0.5 ms Region :**When the `iat` is high (e.g., 5 ms, 2 ms, 1 ms, 0.5 ms), the mean jitter is **almost zero** (0.000 ms to 0.004 ms). In this region, increasing or decreasing the `iat` has a negligible effect. This is because the offered load is so low that packets never have to queue, and thus they all experience the same minimal jitter.

- **0.5 ms $\leq$ iat $\leq$ 0.1 ms Region:** As you **decrease** the `iat` in this range, the mean jitter **increases exponentially**. It jumps from 0.031 ms (at 0.2 ms `iat`) to a **peak of 0.191 ms** (at 0.1 ms `iat`). This is because the offered load is approaching the channel's capacity. The queue becomes unstable and sometimes packets arrive to an empty queue, other times to a growing one. This high variance in queuing time results in the maximum observed jitter.

- **iat $<$ 0.1 ms Region:** As you **decrease** the `iat` even further, from 0.1 ms to 0.05 ms, the mean jitter **decreases** from its peak of 0.191 ms down to 0.146 ms. This is because the offered load is now far greater than the channel's capacity. Packets arrive so rapidly that the queue is **almost always full**. Because every packet is forced to wait in a long, stable queue, *the variation* in their delay starts to decrease, even though the *mean delay* itself section 4.1 is extremely high.

In order to achieve this graph, we have configured the Bitrate in .ini file and adjusted the .ned file to reflect this change. Here are our configuration files.

```
[General]
```

```
network = JitterMeasurementShowcase
.
.
**.datarate = ${bitrate=10Mbps, 100Mbps, 1Gbps, 10Gbps}
```

```
network JitterMeasurementShowcase extends WiredNetworkBase
{
    .
    .
    connections:
        source.ethg++ <--> EthernetLink {
            parameters:
                datarate = parent.datarate;
                delay = parent.delay;
        } <--> destination.ethg++;
}
```

5.2 Question 2

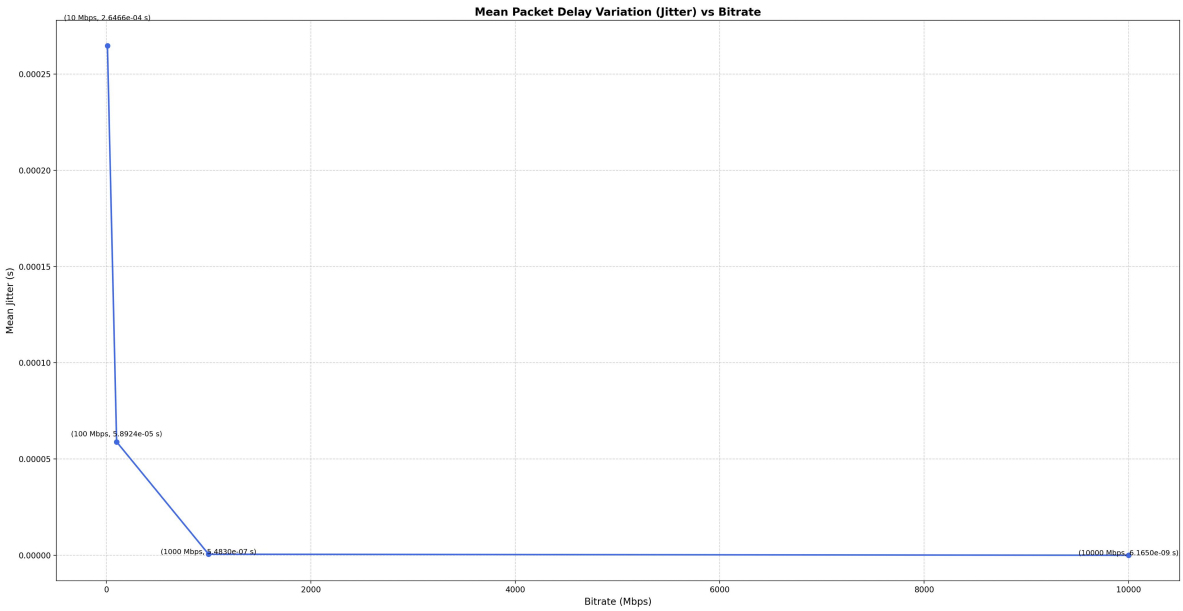

Figure 11: Mean Jitter Depending On Bitrate

Based on Figure 11, there is a strong, non-linear inverse relationship between the link's bitrate and the mean packet delay variation (jitter). The jitter is highest when the link capacity is low and rapidly approaches zero as the capacity increases.

The behavior can be broken down as follows:

- **Peak Jitter at 10 Mbps:** The graph shows the mean jitter is at its **maximum peak** of 2.646×10^{-4} s (0.265 ms) when the bitrate is 10 Mbps. This indicates that

the link's capacity is likely very close to, or slightly below, the offered load. This creates an unstable "saturation point" where the router's queue is constantly and untidily forming and clearing, leading to a high variation in packet delays.

- **Dramatic Drop at 100 Mbps:** When the bitrate is increased to 100 Mbps, the jitter **collapses dramatically** by over 77%, falling to 5.892×10^{-5} s (0.059 ms). At this speed, the link capacity is now comfortably above the offered load. A queue rarely forms, so most packets experience a similar, minimal delay, thus reducing the variation.
- **Negligible Jitter (Bitrate $\geq$ 1000 Mbps):** As the bitrate is increased further to 1000 Mbps and 10000 Mbps, the mean jitter becomes **effectively zero**, dropping to 5.483×10^{-7} s and 6.1650×10^{-9} s, respectively. At these high speeds, the channel is significantly overprovisioned. Packets are transmitted almost instantaneously, the queuing time is always zero, and therefore the *variation* in delay (jitter) is also zero.

6 Transmission Time

6.1 Question 1

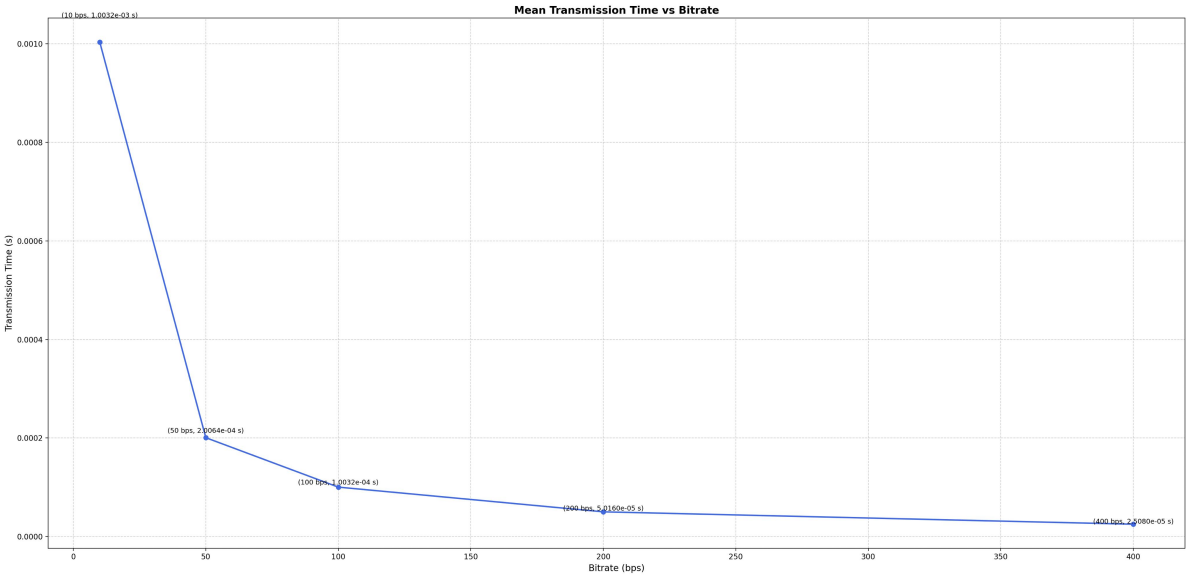

Figure 12: Transmission Time Depending On Ethernet Bitrate

Based on Figure 12, there is a clear and strong **inverse relationship** between the Ethernet bitrate and the mean transmission time.

1. **When you increase the bitrate:** The mean transmission time **decreases** proportionally. This is because a faster link can place the bits of a packet onto the wire in less time.
    - For example, increasing the bitrate from 10 Mbps to 50 Mbps causes the transmission time to drop sharply from 1.003×10^{-3} s to 2.0064×10^{-4} s.

- This trend continues as the bitrate gets higher. Doubling the bitrate from 200 Mbps to 400 Mbps almost perfectly halves the transmission time, from 5.016×10^{-5} s down to 2.508×10^{-5} s.

2. **When you decrease the bitrate:** The mean transmission time **increases** proportionally.
    - For example, decreasing the bitrate from 50 Mbps to 10 Mbps causes the transmission time to increase fivefold, from 2.004×10^{-4} s to 1.003×10^{-3} s.

This behavior is expected and directly demonstrates the formula for transmission time:

$$T = \frac{\text{Packet Length (L)}}{\text{Bitrate (R)}}$$

Since the packet size (1200 Bytes) is constant, the transmission time is directly and inversely proportional to the link's bitrate.

In order to achieve this graph, we have configured only the Bitrate. Here is our configuration file.

```
[General]
network = TransmissionTimeMeasurementShowcase
description = "Measure packet transmission time on the channel"
.
.

*.*.eth[*].bitrate = ${bitrate=10, 50, 100, 200, 400}Mbps
.
.
```

6.2 Question 2

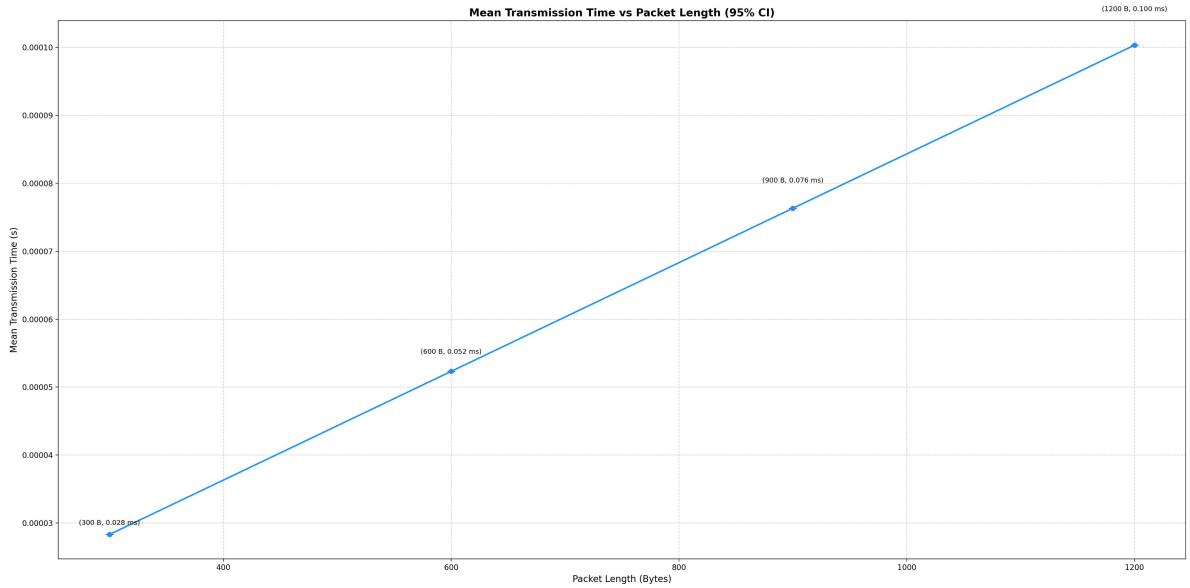

Figure 13: Transmission Time Depending On Packet Length

Based on Figure 13, the transmission time increases linearly with the packet length. This is because the time required to transmit a packet over a link is directly proportional to the packet size and inversely proportional to the link's data rate. The small confidence intervals confirm that the simulation results are highly consistent across repetitions, implying that random effects such as queueing or stochastic interarrival times had negligible influence on the transmission time.

In quantitative terms, the transmission time rises from approximately 0.028 ms for 300 B packets to about 0.100 ms for 1200 B packets. This linear trend validates the theoretical relationship:

$$T_{\text{tx}} = \frac{L}{R}$$

where T_{tx} is the transmission time, L is the packet length (in bits), and R is the channel bitrate (in bits per second). The experimental data therefore align closely with the analytical expectation for a constant 100 Mbps link.

In order to achieve this graph, we have configured only the packetLength. Here is our configuration file.

```
[General]
network = TransmissionTimeMeasurementShowcase
description = "Measure packet transmission time on the channel"
.
.

*.source.app[0].source.packetLength = ${pkt=300B, 600B, 900B, 1200B}
.
```

6.3 Question 3

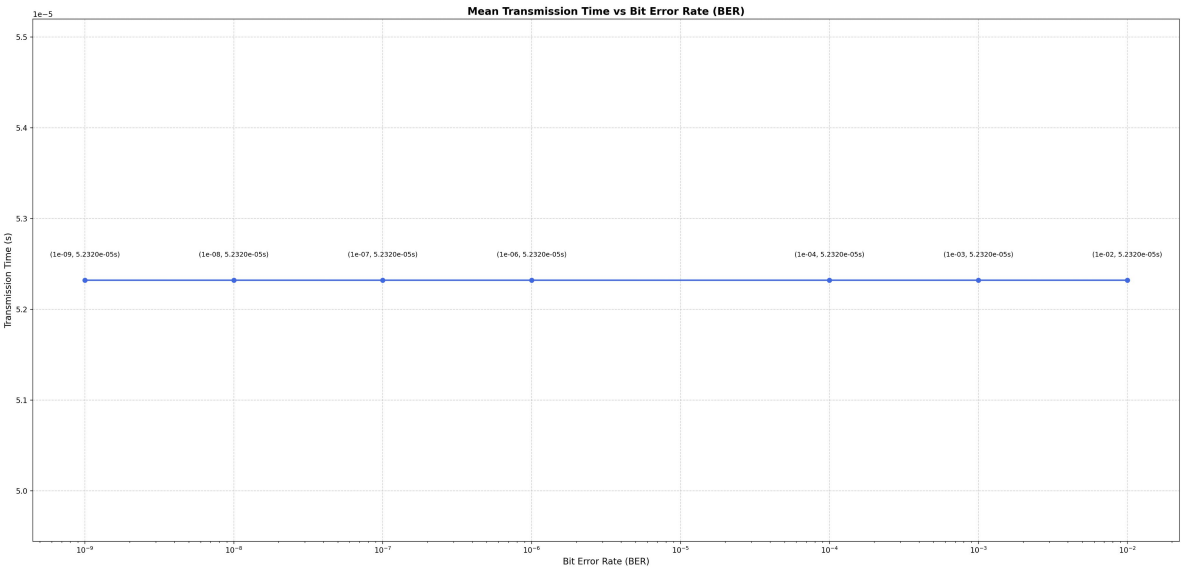

Figure 14: Transmission Time Against BER

Based on Figure 14, the Mean Transmission Time is **completely unaffected** by the Bit Error Rate (BER). The graph shows a perfectly flat horizontal line, with the transmission time remaining constant at 5.232×10^{-5} s (or 0.05232 ms) across the entire range of BER values, from 10^{-9} to 10^{-2}.

This result is expected and demonstrates the fundamental definition of transmission time.

What Transmission Time Is: Transmission time is a function of only two variables: the packet size (L) and the link's bitrate (R). The formula is:

$$T_{\text{tx}} = \frac{\text{Packet Size (L)}}{\text{Bitrate (R)}}$$

This calculation represents the physical time required for the sender's network interface to place all the bits of the packet onto the transmission medium.

BER is a measure of the channel's *quality* or *noise level*. It determines the probability that a bit will be corrupted *after* it has been transmitted.

The sender's hardware transmits at a fixed bitrate (R) regardless of how noisy the channel is. It does not slow down or speed up based on channel quality. Therefore, since the packet size (L) and the bitrate (R) were both constant in this experiment, the transmission time (T_{tx}) must also be constant. The BER only affects whether the packet *arrives correctly* at the destination, not how long it takes to *send* it.

In order to achieve this graph, we have configured only the BER rate. Here is our configuration file.

```
[General]
network = TransmissionTimeMeasurementShowcase
description = "Measure packet transmission time on the channel"
.
.

*.*.eth[*].ber = ${ber_rate = 1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3,
↪  1e-2}
.
.
```

7 Propagation Time

7.1 Question 1

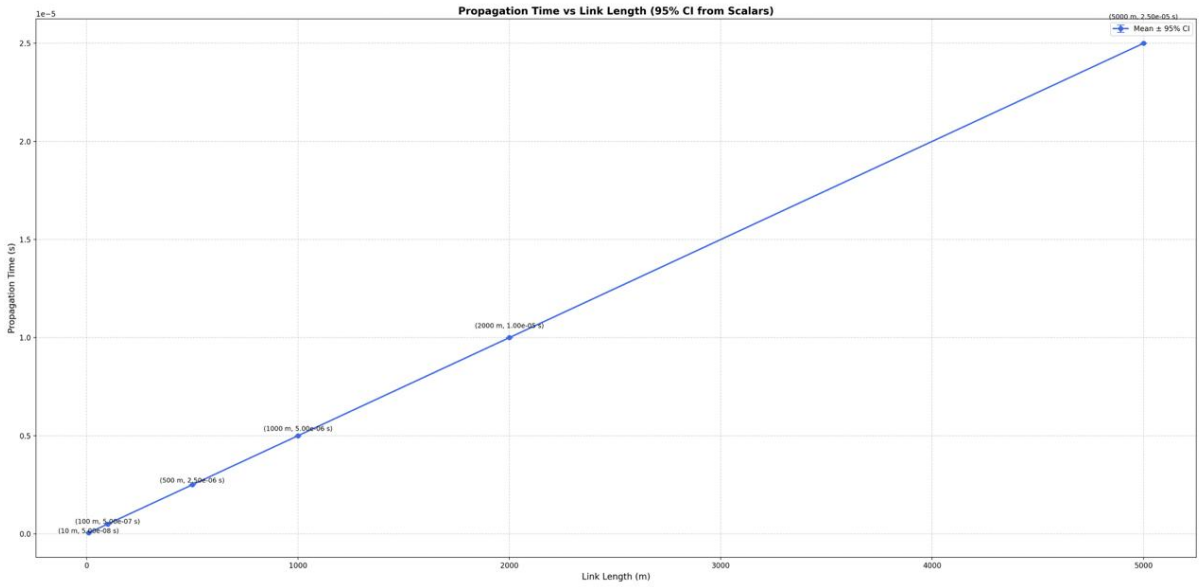

Figure 15: Propagation Time Depending on Channel Length

Based on Figure 15, there is a **direct and perfectly linear relationship** between the link's length and the mean propagation time. As the length of the channel increases, the propagation time increases by the exact same factor. This behavior is demonstrated clearly in the graph's data points

This result is expected and correctly illustrates the physical formula for propagation time:

$$T_p = \frac{\text{Distance}}{\text{Propagation Speed}}$$

The following code, which parameterizes the length value, is sufficient to construct this graph.

```
[General]
network =PropagationTimeMeasurementShowcase
description = "Measure packet propagation time on channel"
.
.
**.channel.length = ${length = 10m, 100m, 500m, 1000m, 2000m, 5000m}
.
.
```

7.2 Question 2

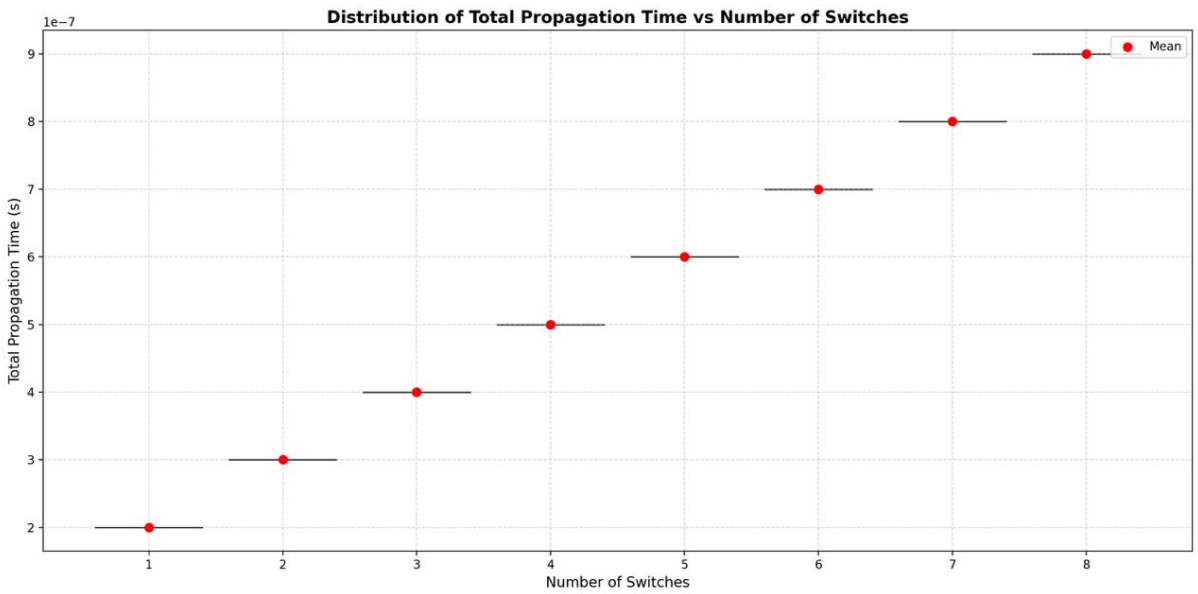

Figure 16: Propagation Time Depending on Number of Switches

Based on Figure 16, there is a **perfectly linear relationship** between the number of switches and the total propagation time. The graph shows that the total propagation time increases by a **constant step of** 1.0×10^{-7} **s** (or 100 ns) for each switch that is added.

This is clearly visible in the data points:

- **1 Switch (2 links):** The total propagation time is 2.0×10^{-7} s.
- **2 Switches (3 links):** The total propagation time is 3.0×10^{-7} s.
- **3 Switches (4 links):** The total propagation time is 4.0×10^{-7} s.
- ...
- **8 Switches (9 links):** The total propagation time is 9.0×10^{-7} s.

This result is expected and demonstrates the additive nature of propagation delay:

1. **Total Delay is a Sum:** The total end-to-end propagation time is simply the sum of the individual propagation times of every link in the path.

2. **Constant Link Delay:** The graph proves that each link in the path has a fixed propagation delay of 1.0×10^{-7} s. Since this experiment used the same `.ned` and `.ini` file configurations as described in Section 4.2, this shows that the `Eth100M` and `Eth1G` channel types both share the same default propagation delay (100 ns), which is independent of their bitrate.

8 Queuing Time

8.1 Question 1

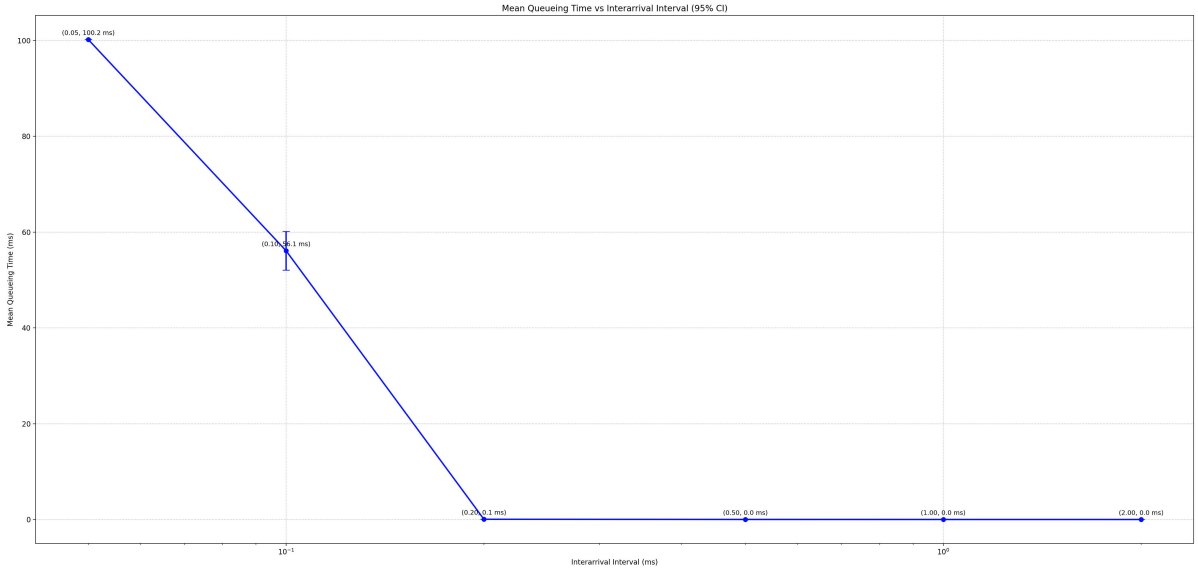

Figure 17: Mean Queueing Time Depending On Interarrival Time

Based on Figure 17, the relationship between interarrival time (iat) and mean queuing time is highly non-linear and shows a clear saturation point.

1. **When you increase the interarrival time:** The mean queuing time **decreases dramatically**. For instance, increasing the `iat` from 0.05 ms to 0.1 ms causes the queuing time to collapse from 100.2 ms down to 56.1 ms. Increasing it further to 0.2 ms causes the queuing time to drop to nearly zero (0.1 ms). For any `iat` above 0.2 ms (e.g., 0.5 ms, 1.0 ms, 2.0 ms), the queuing time is effectively 0.0 ms. This is because packets are arriving less frequently than the channel can process them, so no queue forms.

2. **When you decrease the interarrival time:** The effect depends on the region.

    - **In the low-load region (iat $>$ 0.2 ms):** Decreasing the `iat` (e.g., from 2.0 ms to 0.5 ms) has **no effect**. The queuing time remains at 0.0 ms because the offered load is still well below the channel's capacity.

- **At the saturation point (iat <0.2 ms):** Decreasing the `iat` causes the mean queuing time to **increase exponentially**. As the `iat` drops from 0.2 ms to 0.1 ms, the offered load exceeds the channel capacity, and the queue begins to build rapidly, pushing the delay from 0.1 ms up to 56.1 ms. Decreasing the `iat` even further, from 0.1 ms to 0.05 ms, causes the queue to become even more congested, with the mean delay rising to 100.2 ms.

To construct this graph, it was only necessary to change the network setting in the configuration file. As shown below, the only difference from the configuration in listing 1 is the network parameter; the parameter set for the iat values remains the same.

```
[General]
network = QueueingTimeMeasurementShowcase
description = "Measure queueing time change with respect to interarrival
↪  time"
.
.
*.source.app[0].source.productionInterval = exponential(${interval =
↪  0.05ms, 0.1ms, 0.2ms, 0.5ms, 1ms, 2ms})
.
.
```

8.2 Question 2

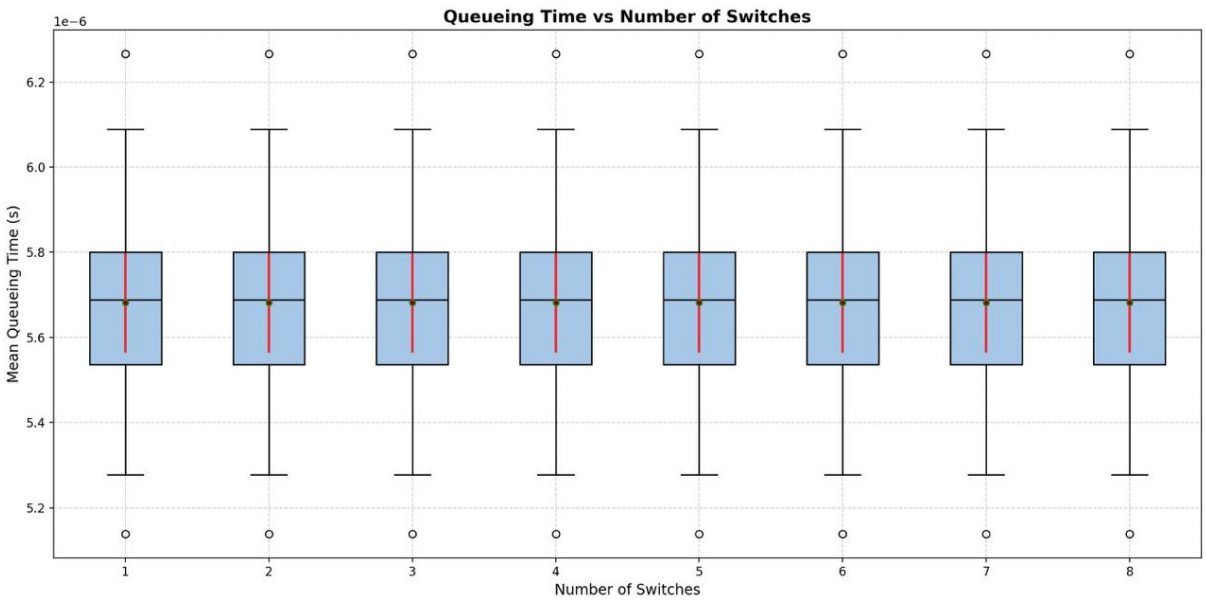

Figure 18: Mean Queueing Time Depending On Number of Switches

Based on Figure 18, the mean queuing time **remains constant** and is completely unaffected by the number of switches in the path.

This is immediately clear from the box plots, which are identical for all switch counts from 1 to 8:

- The mean (green dot) and median (orange line) form a perfectly horizontal line at approximately 5.7×10^{-6} s
- The interquartile range and the outlier points are also identical in every case.

This result is expected and demonstrates that our network is **uncongested**.

1. **No Bottlenecks:** The network configuration (as described in Section 4.2) uses 100 Mbps edge links and a 1 Gbps backbone. As long as the offered load from the source application is less than 100 Mbps, no link in the entire path will be saturated.
2. **No Queues:** Because there is no congestion, packets are not forced to wait in a buffer at any switch. They are processed and forwarded immediately. This means no significant queuing delay is generated.

This experiment successfully isolates the queuing component of delay. It proves that in our uncongested setup, the queuing delay is a minimal and constant value. This also confirms that the linear increase in total end-to-end delay (which we saw in Figure 8) is almost entirely due to the sum of the *propagation delays* and *transmission delays* added by each link.