Data networks

DR.Pakravan

electrical engineering department

Ahmadreza Majlesara 400101861

assignment 2

May 23, 2024

Data networks

assignment 2

Ahmadreza Majlesara 400101861

Doing Measurements Using INET Framework

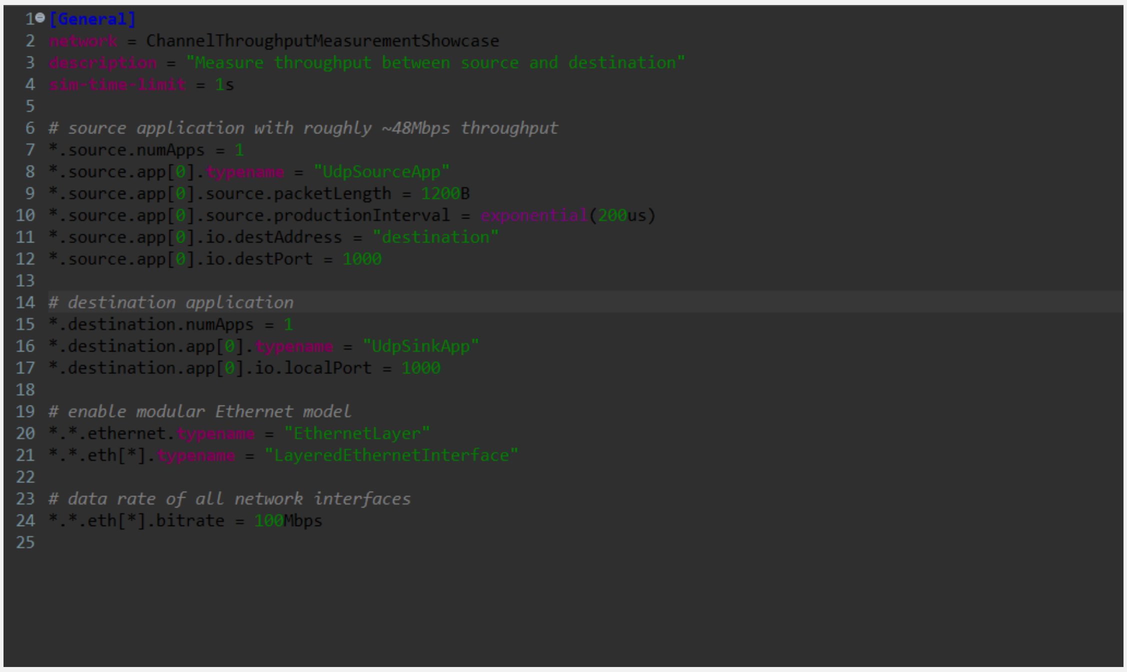
1. *Measuring Channel Throughput*

The channel throughput is measured by observing the packets that are transmitted through the transmission medium over time. For both wired and wireless channels, the throughput is measured for any pair of communicating network interfaces, separately for both directions. Channel throughput is a statistic of transmitter modules, such as the `PacketTransmitter` in `EthernetPhyLayer`. Throughput is measured with a sliding window. By default, the window is 0.1s or 100 packets, whichever comes first. The window parameters, such as the window interval, are configurable from the ini file, as `module.statistic.parameter`.

Now we want to configure a simulation for a wired network that contains two `StandardHosts` as source and destination, connected via 100 Mbps Ethernet. The network is shown in the image below. Configure the packet source in the source hosts' UDP app to generate 1200-byte packets with a period of around 200 us randomly (the `source.productionInterval` follows an exponential distribution). Implement the network and run the simulation for 1 second. Then plot the throughput and compare it with the result of your calculation.

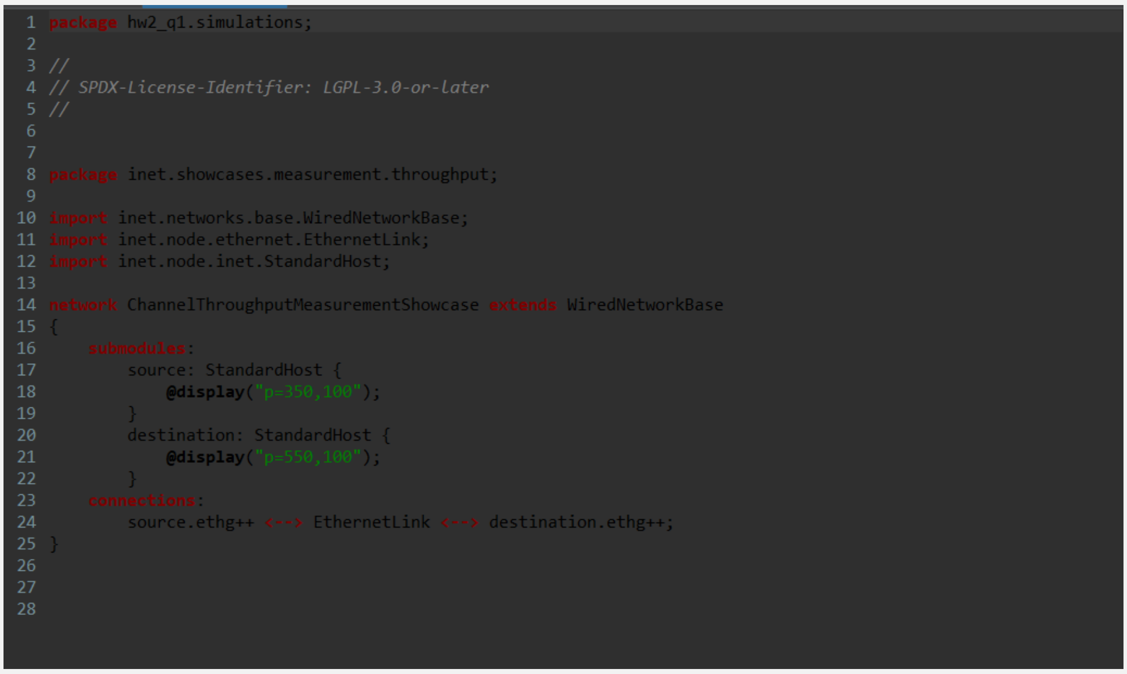
solution

first lets calculate the channel theoughput theorically:
The packet generation rate is the reciprocal of the period $(200\mu s)$ so the packet generation rate is $\frac{1}{200\mu s} = 5000$ packets/sec so in 1 second we have 5000 packets. each packet has a size of 1200 bytes so the total data transmitted in 1 second is $5000 \times 1200 = 6{,}000{,}000$ bytes. so the channel throughput is $6{,}000{,}000 bytes/sec$ or $48{,}000{,}000 bits/sec$ or $48 Mbps$.
now we simulate the network above. the ini file have been uploaded alongside this report.
for a brief explanation this code generates packets with a size of 1200 bytes and a period of 200 us for 1 second then it send these packets via port 1000 of the source host.
the NED file and network topology is shown in figures 2 and 3. as a brief explanation the .NED file sets up a basic network with two hosts connected via an Ethernet link.



```
[General]
network = ChannelThroughputMeasurementShowcase
description = "Measure throughput between source and destination"
sim-time-limit = 1s

# 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(200us)
*.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"

# data rate of all network interfaces
*.*.eth[*].bitrate = 100Mbps
```

Figure 1. ini file for the simulation



```
package hw2_q1.simulations;

//
// SPDX-License-Identifier: LGPL-3.0-or-later
//


package inet.showcases.measurement.throughput;

import inet.networks.base.WiredNetworkBase;
import inet.node.ethernet.EthernetLink;
import inet.node.inet.StandardHost;

network ChannelThroughputMeasurementShowcase extends WiredNetworkBase
{
    submodules:
        source: StandardHost {
            @display("p=350,100");
        }
        destination: StandardHost {
            @display("p=550,100");
        }
    connections:
        source.ethg++ <--> EthernetLink <--> destination.ethg++;
}
```

Figure 2. NED file for the simulation

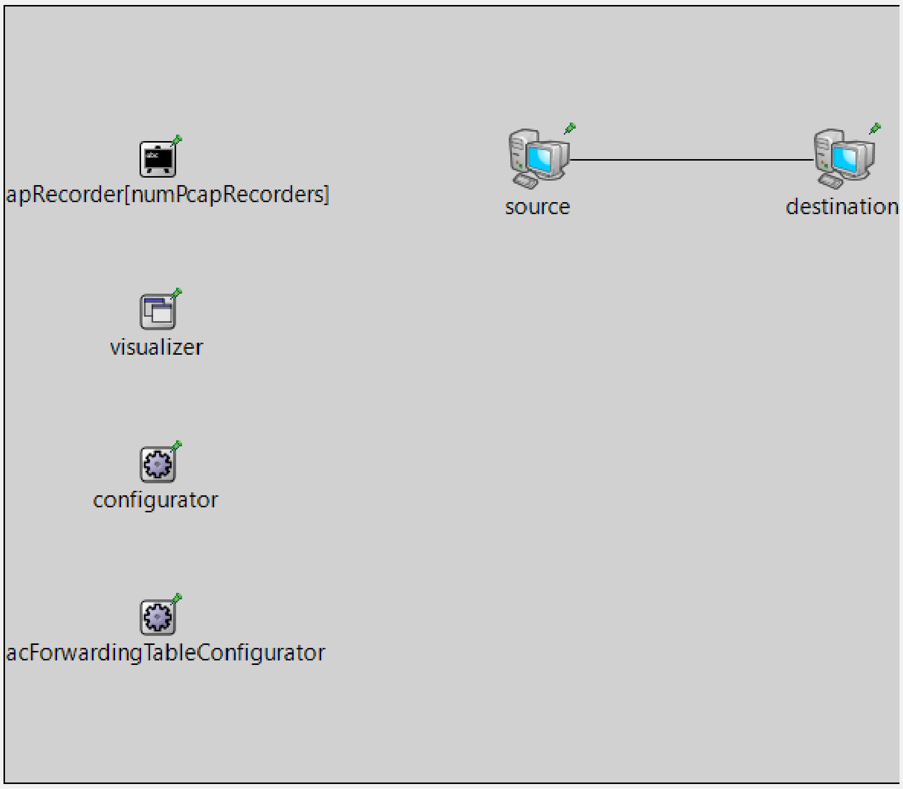


Figure 3. network topology for the simulation

the result is shown in figure 4. as we can see the throughput that is measured by the simulation is $48.0Mbps$ which is the same as the theoretical value.

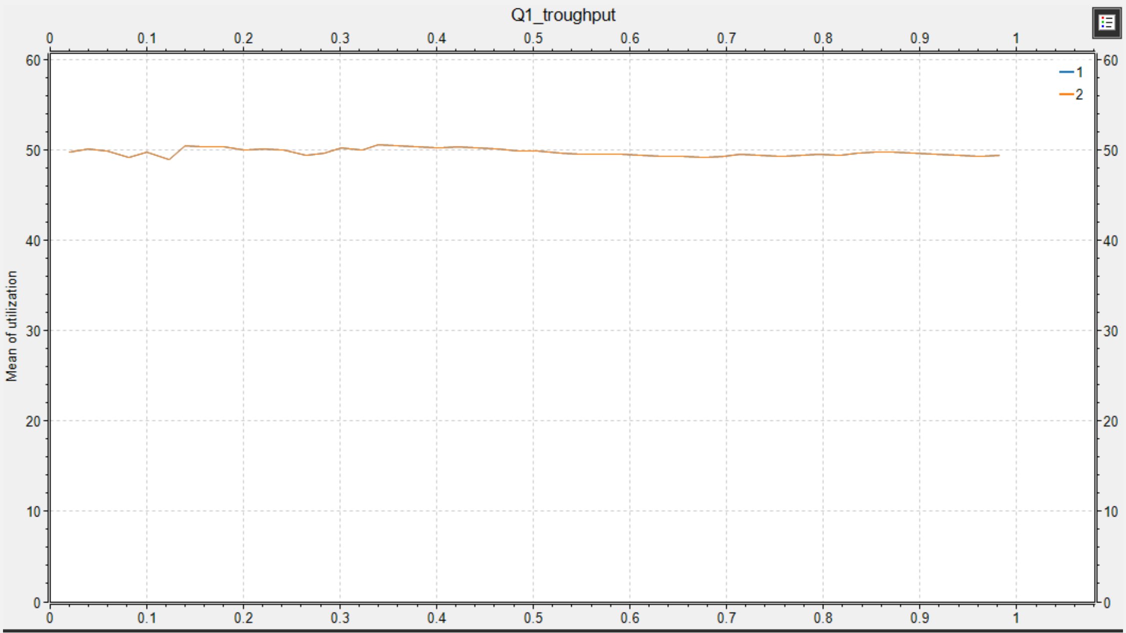


Figure 4. channel throughput

2. Measuring Channel Utilization

The channel utilization statistic is measured by observing the packets that are transmitted through the transmission medium over time. For both wired and wireless channels, the utilization is measured for any pair of communicating network interfaces, separately for both directions. This statistic expresses the relative usage of the channel with a value between 0 and 1, where 0 means the channel is not used at all and 1 means the channel is fully utilized.

Channel utilization is a statistic of transmitter modules, such as the `PacketTransmitter` in `EthernetPhyLayer`. The channel utilization is related to channel throughput in the sense that utilization is the ratio of throughput to channel datarate. By default, channel utilization is calculated for the past 0.1s or the last 100 packets, whichever comes first.

These values are configurable from the ini file with the `interval (|s|)` and `numValueLimit` parameters, as `module.statistic.parameter`.

For the model that you implemented in the last part, plot the throughput, and compare it with the result of your calculation.

solution

to calculate the channel utilization as we calculated the channel throughput in the previous part we simply divide that value by the channel datarate. so the channel utilization is $48Mbps/100Mbps = 0.48$.
the NED file,ini file and network topology is the same as the previous part and are uploaded alongside this report. the result is shown in figure 5. as we can see the channel utilization that is measured by the simulation is 0.48 which is the same as the theoretical value.

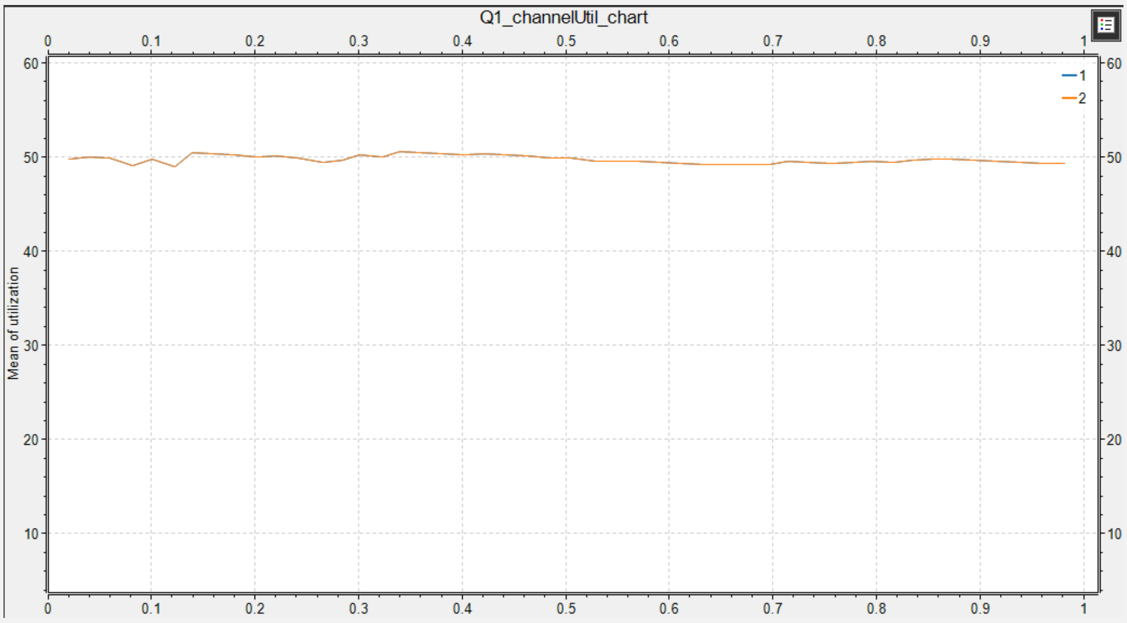


Figure 5. channel utilization

channel utilization histogram is shown in figure 6. as we can see the channel utilization is around 0.48.

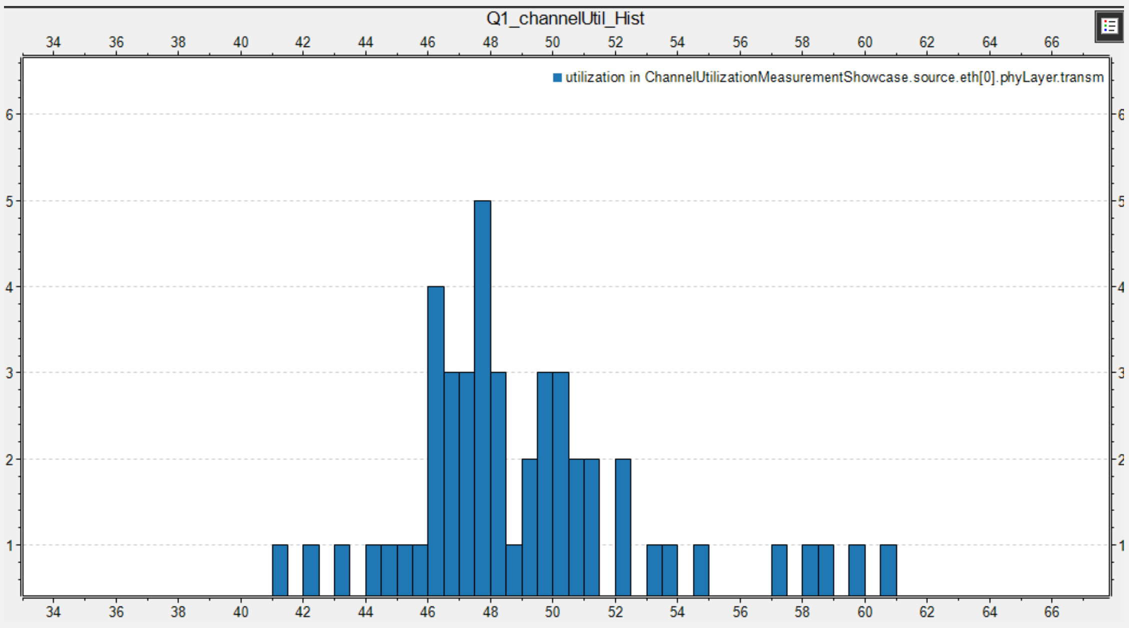


Figure 6. channel utilization histogram

The histogram provides a visual representation of the channel utilization over time. The channel predominantly operates around the expected utilization of 48 Mbps, with some variation due to the random nature of packet generation. The data confirms efficient channel usage, while also highlighting periods of higher and lower utilization that could be investigated further for performance optimization.

3. Measuring End-to-end Delay

The end-to-end delay is measured from the moment the packet leaves the source application to the moment the same packet arrives at the destination application. The end-to-end delay is measured by the `meanBitLifeTimePerPacket` statistic. The statistic measures the lifetime of the packet, i.e. time from creation in the source application to deletion in the destination application.

The `meanBit` part refers to the statistic being defined per bit, and the result is the mean of the per-bit values of all bits in the packet. When there is no packet streaming or fragmentation in the network, the bits of a packet travel together, so they have the same lifetime value.

(a) To visualize the end-to-end delay, run the model that you implemented in the first part, and plot the `meanBitLifeTimePerPacket` statistic in both vector and histogram form.

(b) Now consider that the source generates packets with a period of around 100 us randomly. Again plot the `meanBitLifeTimePerPacket` statistic in both vector and histogram form. Explain the results.

(c) Explain the differences between the results of parts (a) and (b).

solution

the NED file,ini file and network topology is the same as the previous part and are uploaded alongside this report. the result for part a is shown in figure 7 and the histogram for part a is shown in figure 8. these figures are for the packet generation with period of $200\mu s$

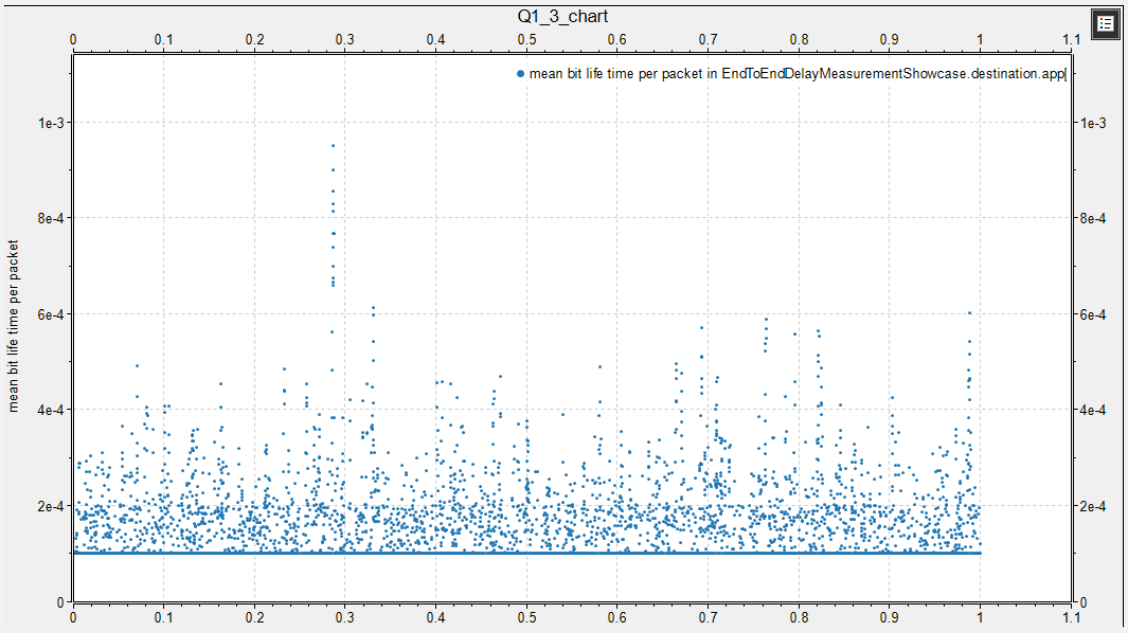


Figure 7. end-to-end delay for part a (period = $200\mu s$)

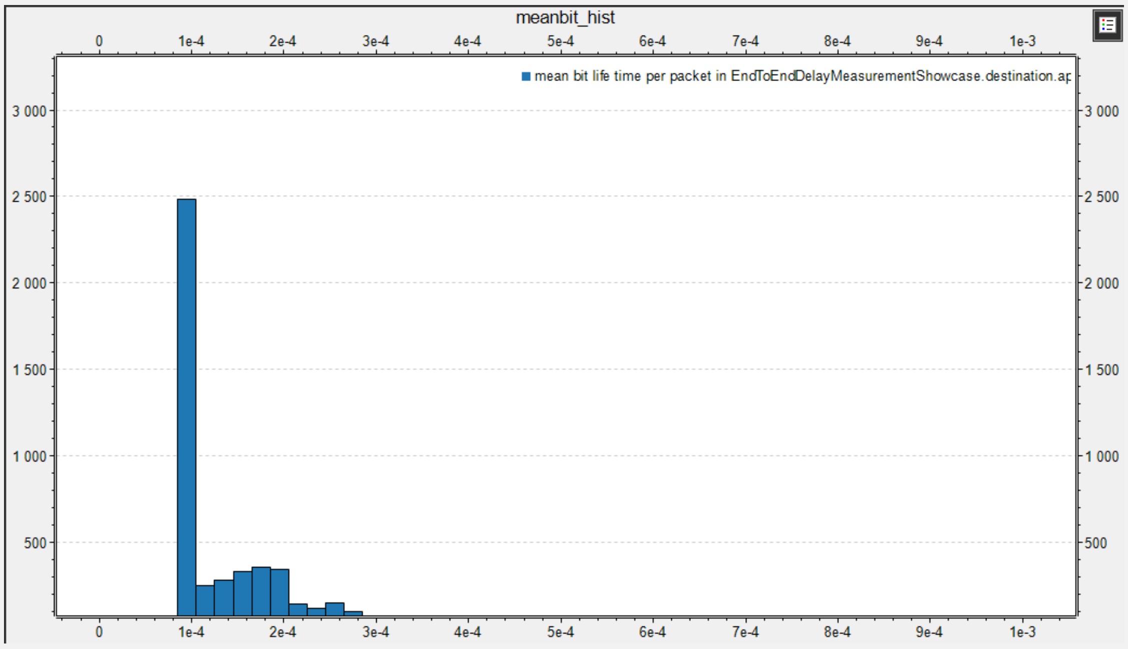
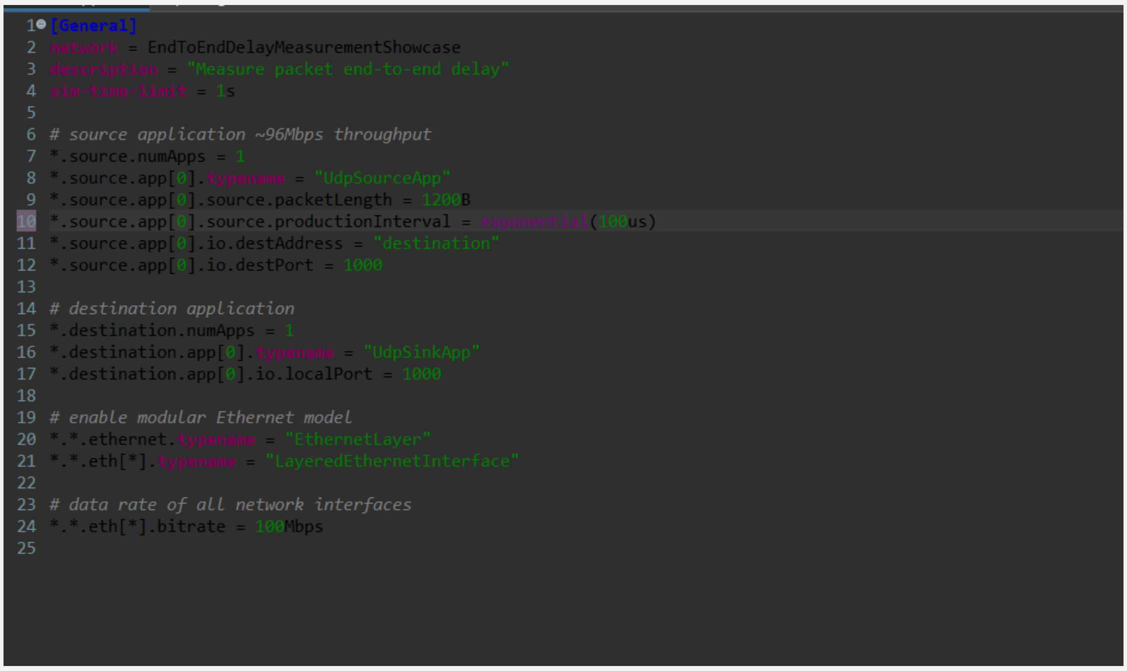


Figure 8. end-to-end delay histogram for part a (period = $200\mu s$)

now for part b we simply set the packet generation period to $100\mu s$ the ini file shown in figure 9. the NED file and network topology is the same as the previous part. the result for part b is shown in figure 10 and the histogram for part b is shown in figure 11.



```
[General]
network = EndToEndDelayMeasurementShowcase
description = "Measure packet end-to-end delay"
sim-time-limit = 1s

# source application ~96Mbps throughput
*.source.numApps = 1
*.source.app[0].typename = "UdpSourceApp"
*.source.app[0].source.packetLength = 1200B
*.source.app[0].source.productionInterval = exponential(100us)
*.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"

# data rate of all network interfaces
*.*.eth[*].bitrate = 100Mbps
```

Figure 9. ini file for part b (period = $100\mu s$)

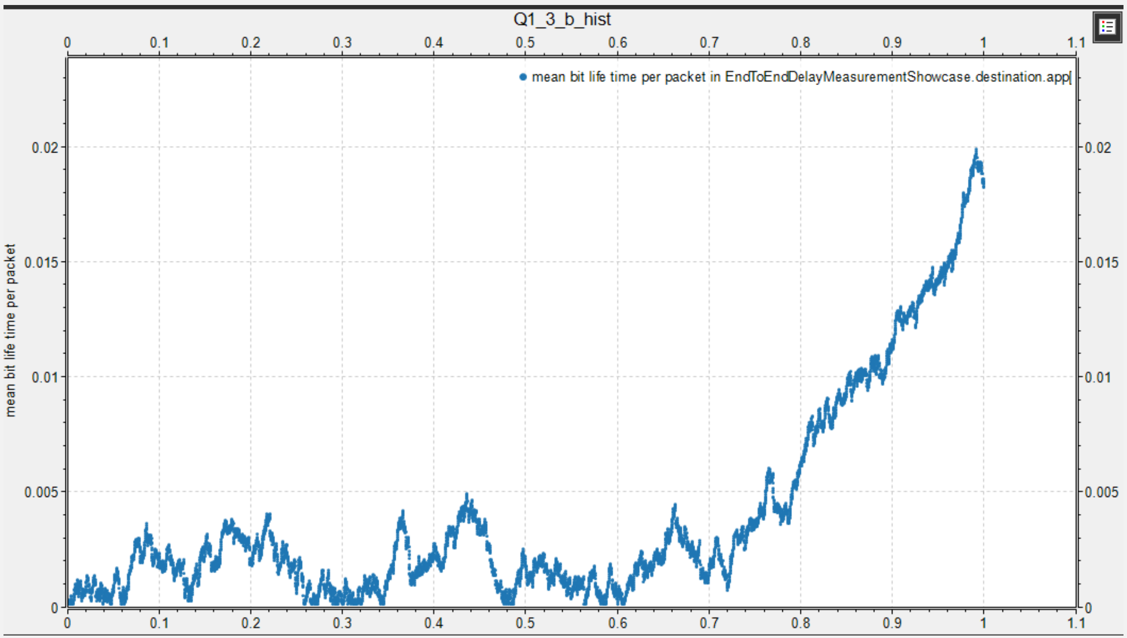


Figure 10. end-to-end delay for part b (period = $100\mu s$)

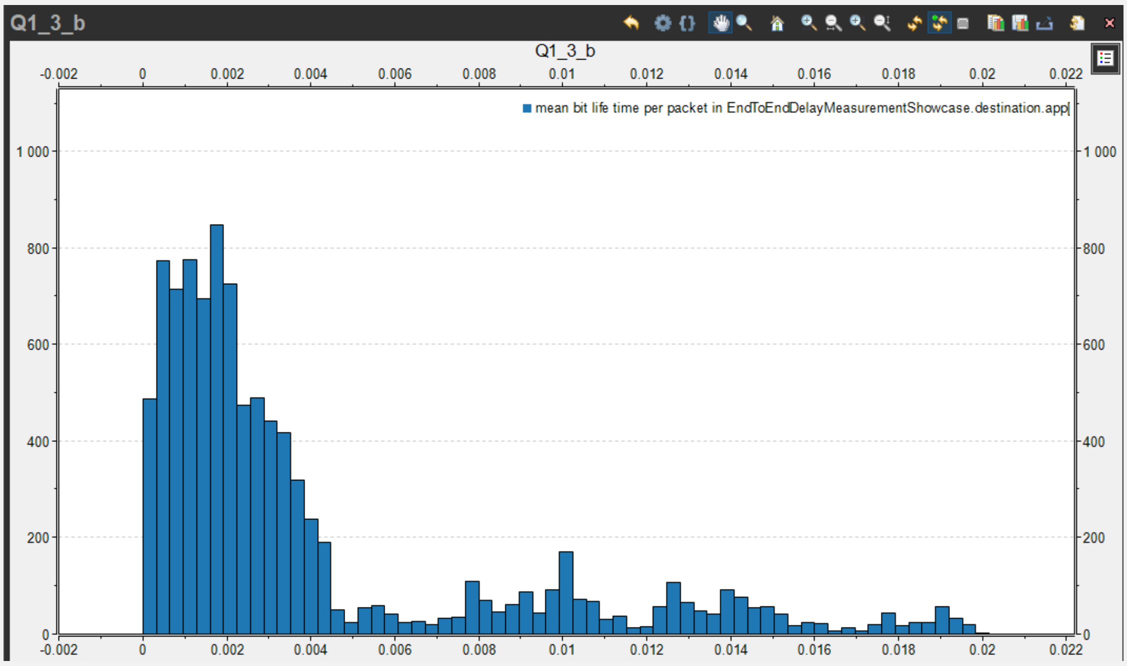


Figure 11. end-to-end delay histogram for part b (period = $100\mu s$)

When packets are generated more frequently (every $100\mu s$), the network experiences higher traffic load.This increased load causes more queuing at intermediate network devices leading to higher end-to-end delays.The increasing trend in the mean bit life time per packet indicates that the network becomes progressively more congested over time, exacerbating delays. so as we see in figure 10 the end-to-end delay is higher than the previous part. and in comparasion with the histogram of part a and part b we can see that the end-to-end delay is higher in part b.

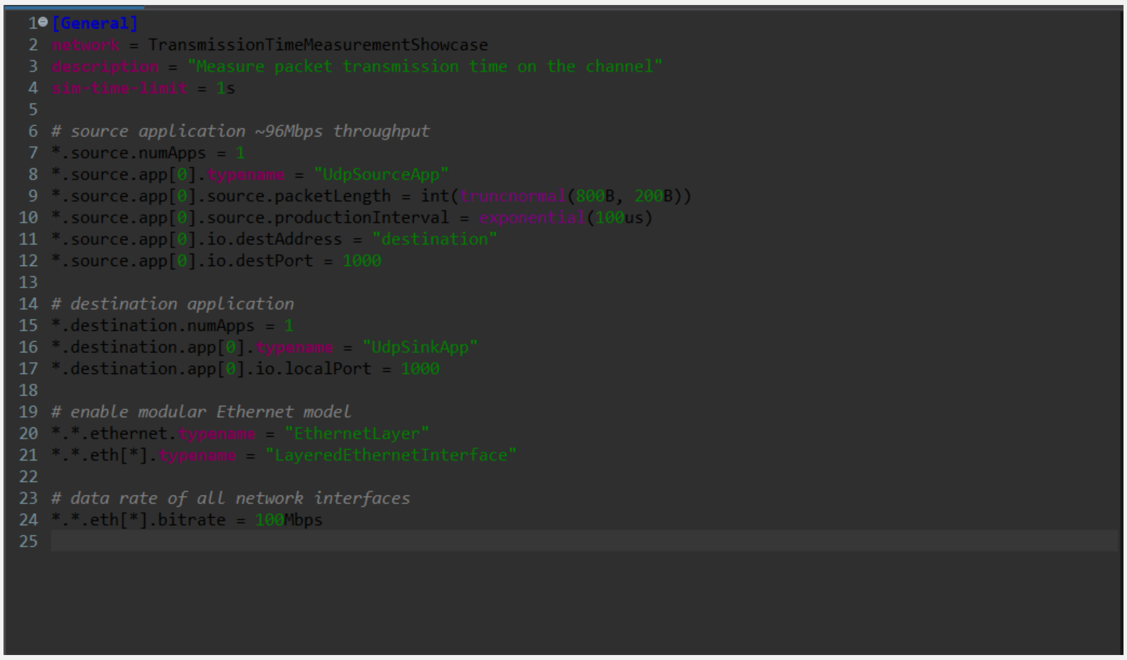
4. Measuring Transmission Time

The packet transmission time is measured from the moment the beginning of the physical signal encoding the packet leaves the network interface up to the moment the end of the same physical signal leaves the same network interface. This time usually equals the packet reception time that is measured at the receiver network interface from the beginning to the end of the physical signal. The exception would be when the receiver is moving relative to the transmitter at a relatively high speed compared to the propagation speed of the physical signal, but this is rarely the case in communication network simulation.

For the model that you implemented, again consider that the source generates packets with a period of around 100 us randomly. In this part, the packet length is variable and follows a truncated normal distribution between 200 bytes and 800 bytes. Plot the source transmission time in both vector and histogram form. Explain the result and compare it with your calculation.

solution

the NED file and network topology is the same as the previous part. the ini file for this part is shown in figure 12. the result for this part is shown in figure 13.the difference is ini file is that the packet length is variable and follows a truncated normal distribution between 200 bytes and 800 bytes.



```
[General]
network = TransmissionTimeMeasurementShowcase
description = "Measure packet transmission time on the channel"
sim-time-limit = 1s

# source application ~96Mbps throughput
*.source.numApps = 1
*.source.app[0].typename = "UdpSourceApp"
*.source.app[0].source.packetLength = int(truncnormal(800B, 200B))
*.source.app[0].source.productionInterval = exponential(100us)
*.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"

# data rate of all network interfaces
*.*.eth[*].bitrate = 100Mbps
```

Figure 12. ini file for part 4

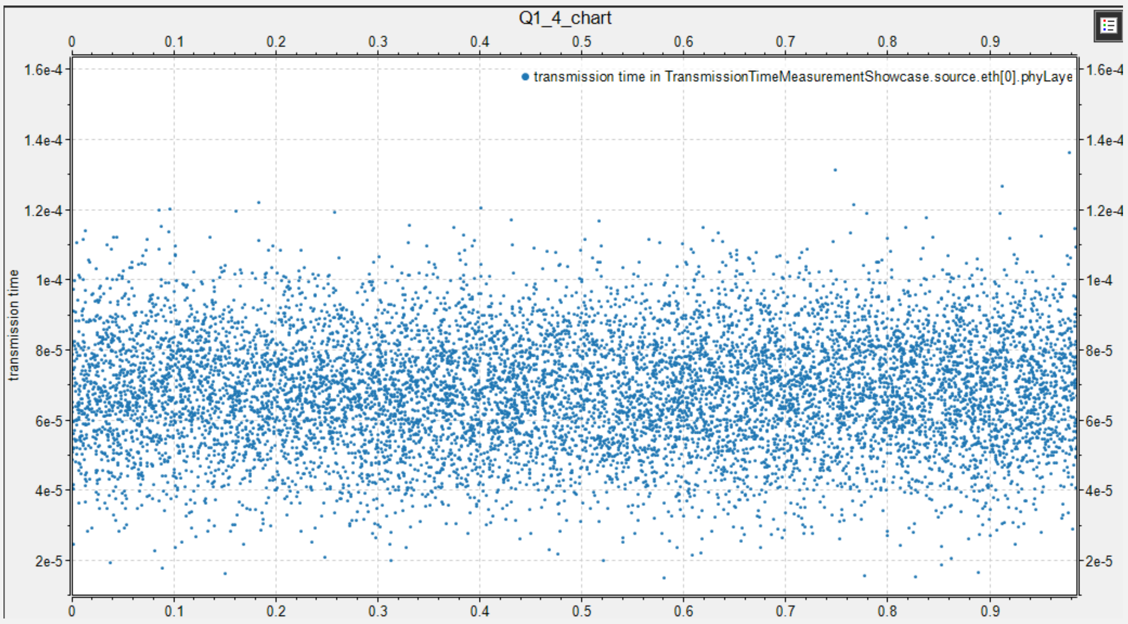


Figure 13. source transmission time for part 4

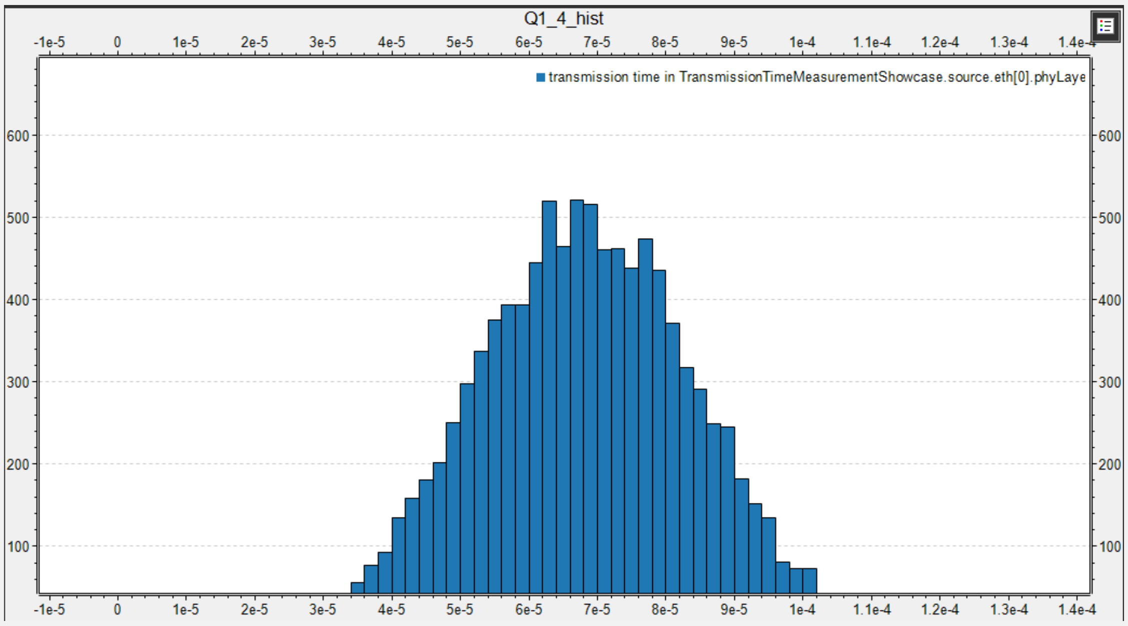


Figure 14. source transmission time histogram for part 4

for the theorical calculation we can calculate the transmission time for a packet with size of 200 to 800 bytes. the transmission time is the time it takes to transmit the packet over the channel. the channel datarate is 100 Mbps so the transmission time is $\frac{packetsize}{channeldatarate}$. so the transmission time for a packet with size of 200 bytes is $\frac{200\times8}{100\times10^6} = 16\mu s$ and for a packet with size of 800 bytes is $\frac{800\times8}{100\times10^6} = 64\mu s$. so the transmission time for a packet with size of 200 to 800 bytes is between 16 to 64 μs. The observed range in the scatter plot ($20\mu s - 160\mu s$) is slightly broader but still within a reasonable range, likely due to additional processing overhead and network conditions.as we can see The histogram shows a peak around $65\mu s$. The histogram's bell-shaped distribution confirms the expected normal distribution of packet sizes and corresponding transmission times.

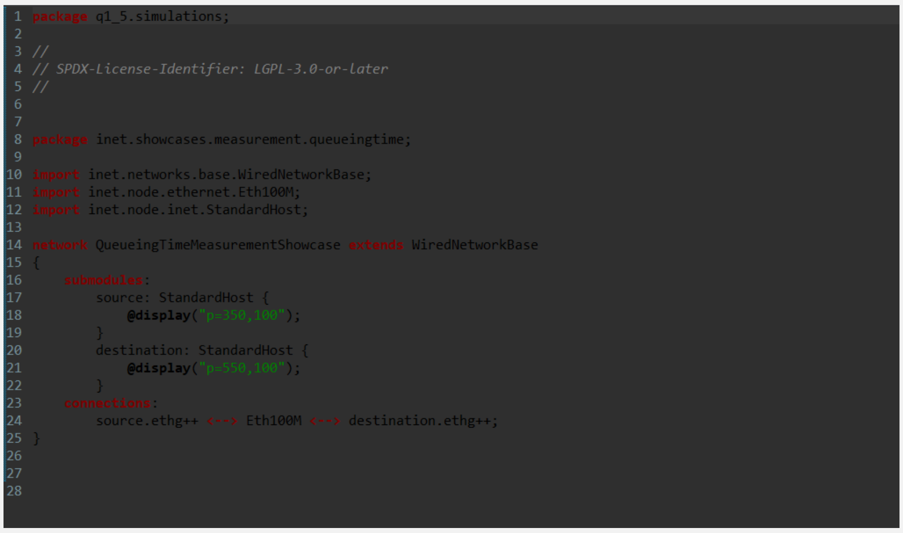
5. *Measuring Queueing Time*

The queueing time is measured from the moment a packet is enqueued up to the moment the same packet is dequeued from the queue. Simple packet queue modules are also often used to build more complicated queues such as a priority queue or even traffic shapers. The queueing time statistics are automatically collected for each of these cases too.

For this part, first add a `recorder[numPcapRecorders]` module to the simulation. Again consider that the source generates packets with a period of around 100 μs randomly. Run the simulation for 1 second, and report the source queueing time in both vector and histogram form. Now run the simulation for 5 seconds to see the trend in plots better. Explain the result.

solution

the NED file and network topology are shoen in figure 15 and 16. the ini file for this part is as same as part 3. the results for this part is shown in figure 17 (1s runtime) and 18 (5s runtime).



```
package q1_5.simulations;

//
// SPDX-License-Identifier: LGPL-3.0-or-later
//


package inet.showcases.measurement.queueingtime;

import inet.networks.base.WiredNetworkBase;
import inet.node.ethernet.Eth100M;
import inet.node.inet.StandardHost;

network QueueingTimeMeasurementShowcase extends WiredNetworkBase
{
    submodules:
        source: StandardHost {
            @display("p=350,100");
        }
        destination: StandardHost {
            @display("p=550,100");
        }
    connections:
        source.ethg++ <--> Eth100M <--> destination.ethg++;
}



```

Figure 15. NED file for part 5

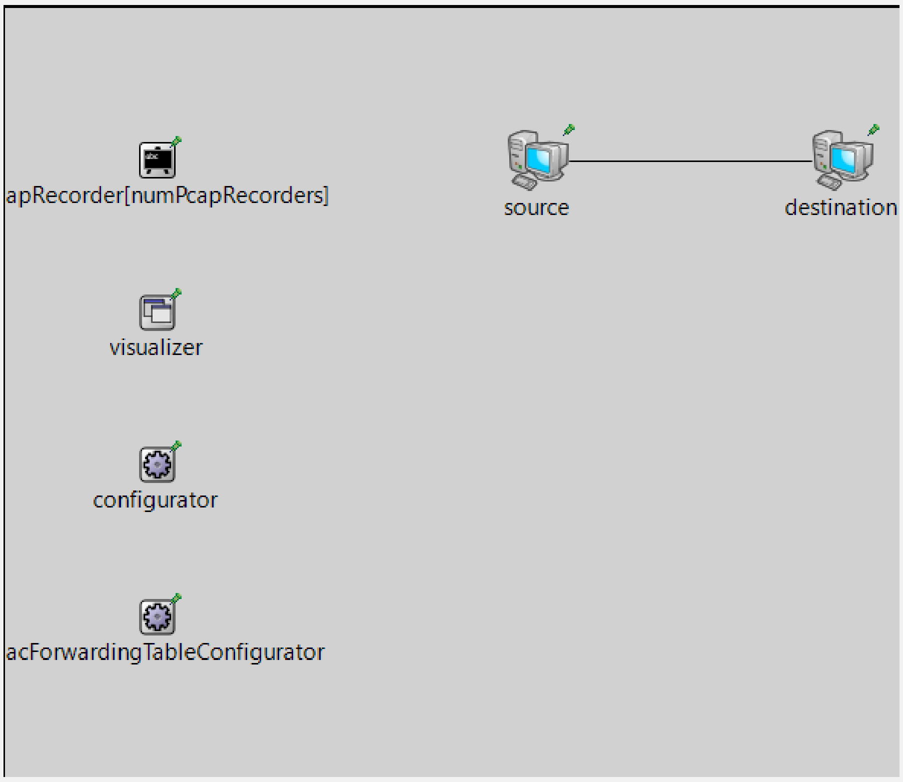


Figure 16. network topology for part 5

as it can be seen the recorder is added to the topology.

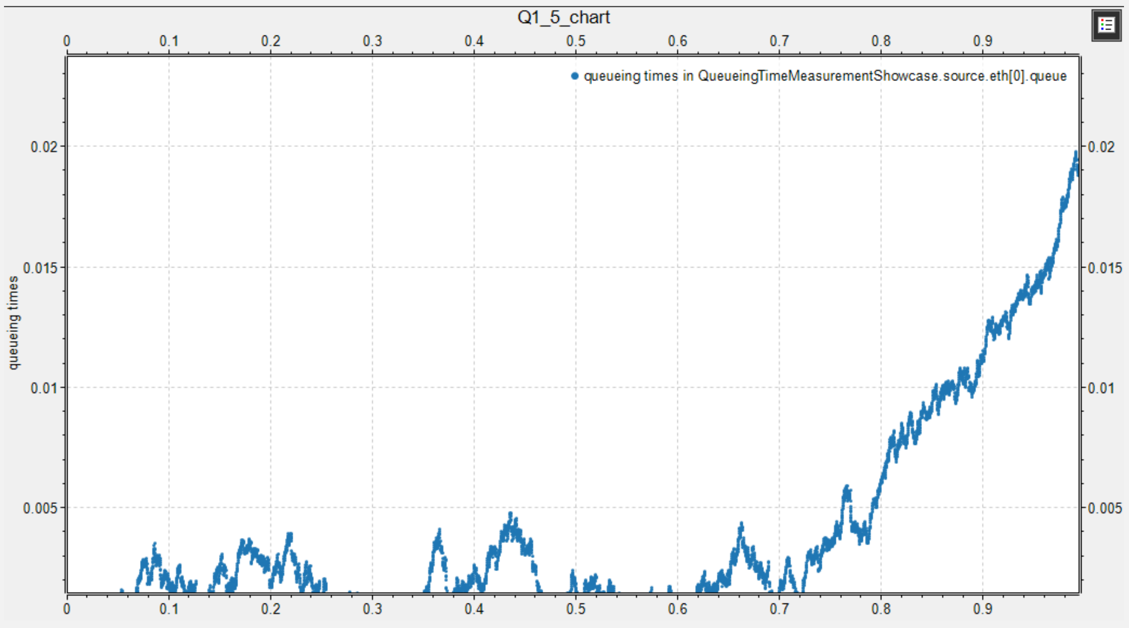


Figure 17. source queueing time for part 5 (1s runtime)

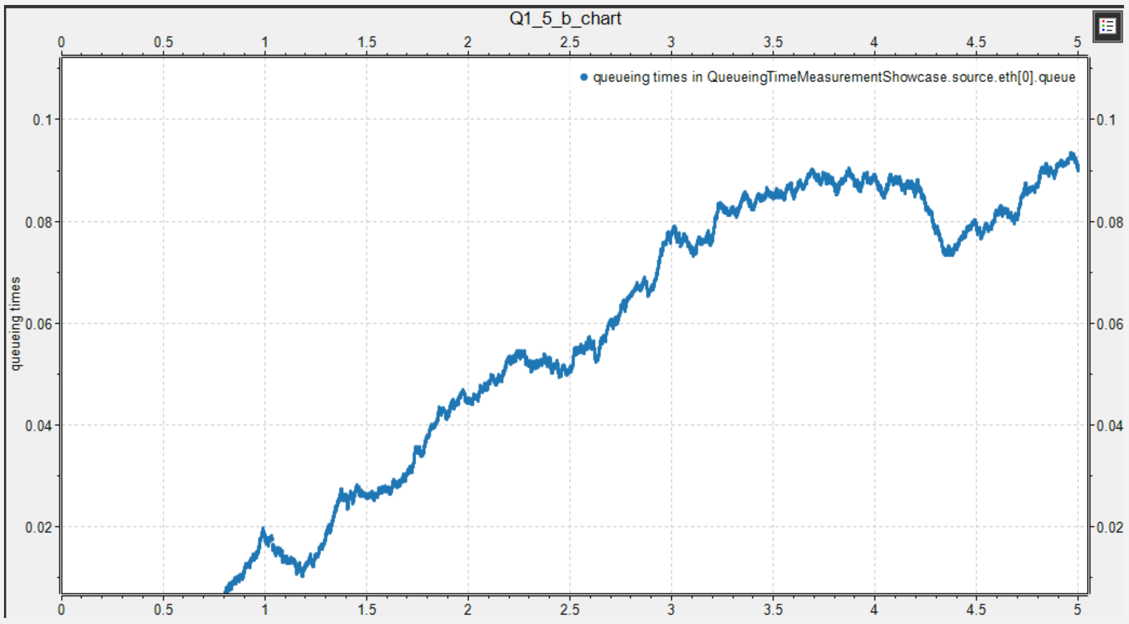


Figure 18. source queueing time for part 5 (5s runtime)

The results from both the line charts suggest that as time progresses, the system experiences increasing strain leading to longer queueing times. This could be due to various factors such as increased packet arrival rates, insufficient processing speed, or prioritization schemes that delay some packets more than others. The sharp rise towards the end of the time frames in both line charts could point to a tipping point where the queue capacity is regularly exceeded, leading to longer delays.

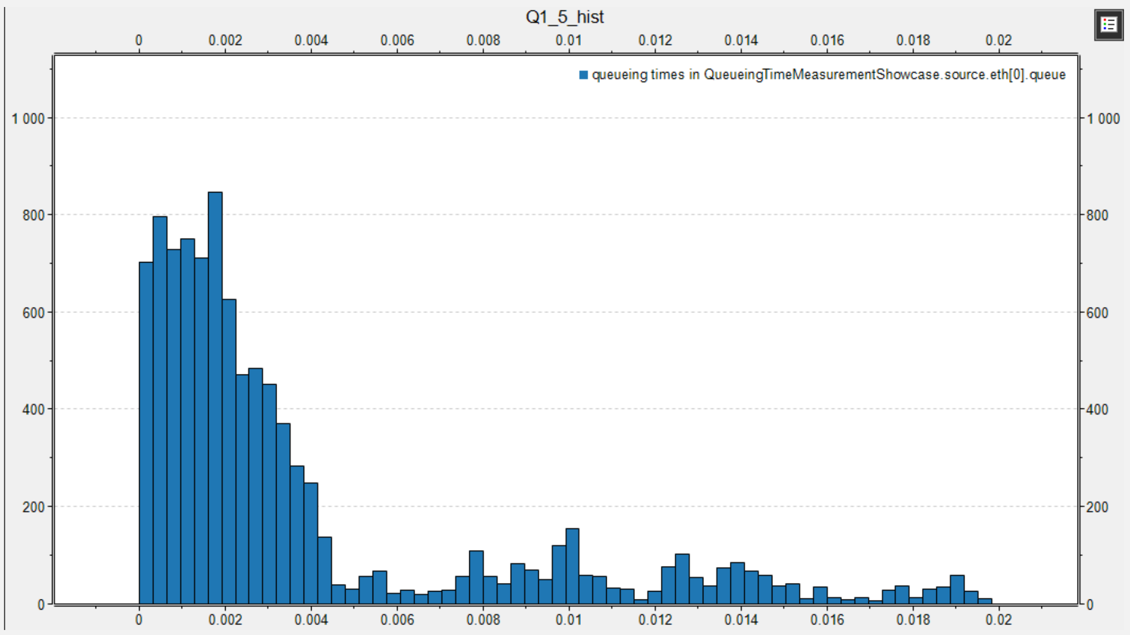


Figure 19. source queueing time histogram for part 5

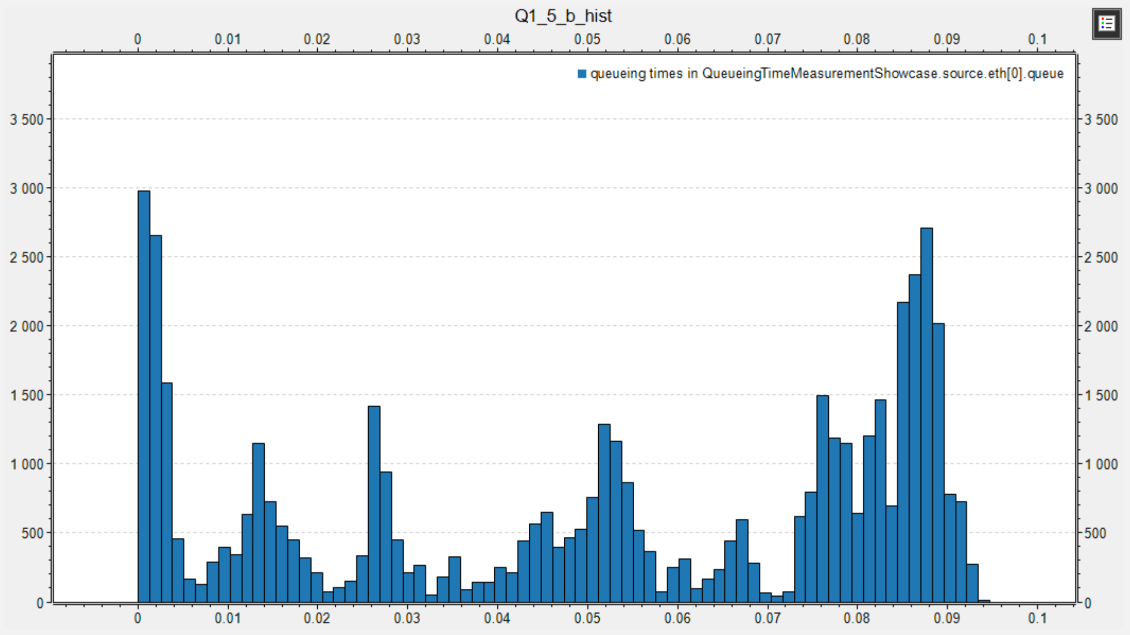


Figure 20. source queueing time histogram for part 5

the histograms for the 1-second and 5-second simulations show that while the queue can handle traffic efficiently most of the time, increased simulation time exposes the system to more complex dynamics and potential congestion, leading to a broader and more variable range of queueing times

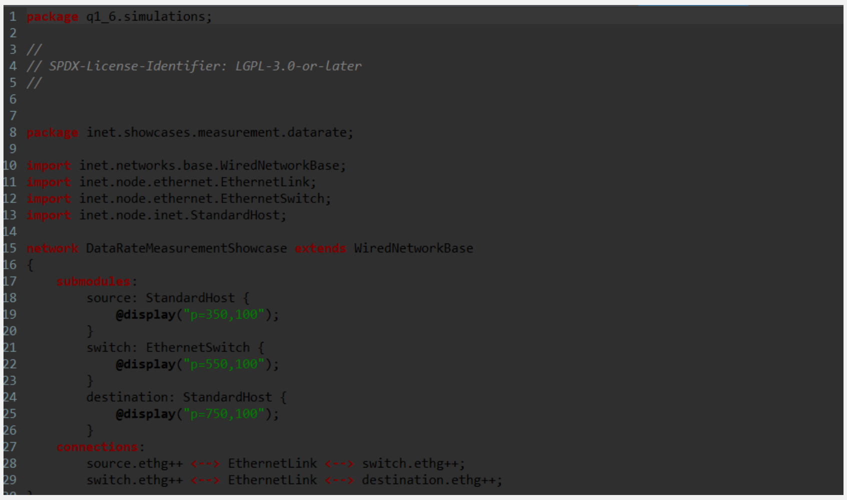
6. Measuring Data Rate

The data rate is measured by observing the packets as they are passing through overtime at a certain point in the node architecture. For example, an application source module produces packets over time and this process has its own data rate. Similarly, a queue module enqueues and dequeues packets over time and both of these processes have their own data rate. These data rates are different, which in turn causes the queue length to increase or decrease over time.

Consider that the source generates packets with a period of around 200 μs randomly. Run the simulation for 1 second, and report the source and destination data rates.

solution

the ini file is the same as part 1. the NED file and network topology is shown in figure 21 and 22. the results for this part are shown in figure 23 and 24.



```
package q1_6.simulations;

//
// SPDX-License-Identifier: LGPL-3.0-or-later
//


package inet.showcases.measurement.datarate;

import inet.networks.base.WiredNetworkBase;
import inet.node.ethernet.EthernetLink;
import inet.node.ethernet.EthernetSwitch;
import inet.node.inet.StandardHost;

network DataRateMeasurementShowcase extends WiredNetworkBase
{
    submodules:
        source: StandardHost {
            @display("p=350,100");
        }
        switch: EthernetSwitch {
            @display("p=550,100");
        }
        destination: StandardHost {
            @display("p=750,100");
        }
    connections:
        source.ethg++ <--> EthernetLink <--> switch.ethg++;
        switch.ethg++ <--> EthernetLink <--> destination.ethg++;
```

Figure 21. NED file for part 6

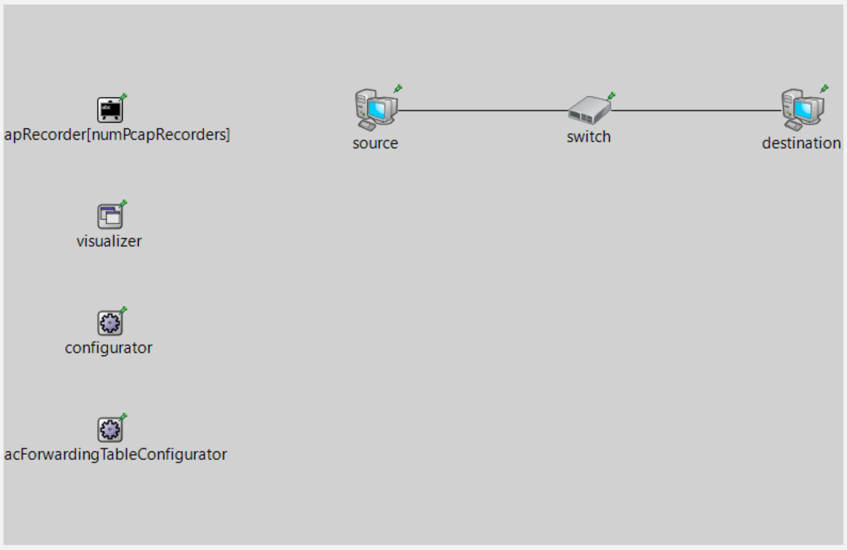


Figure 22. network topology for part 6

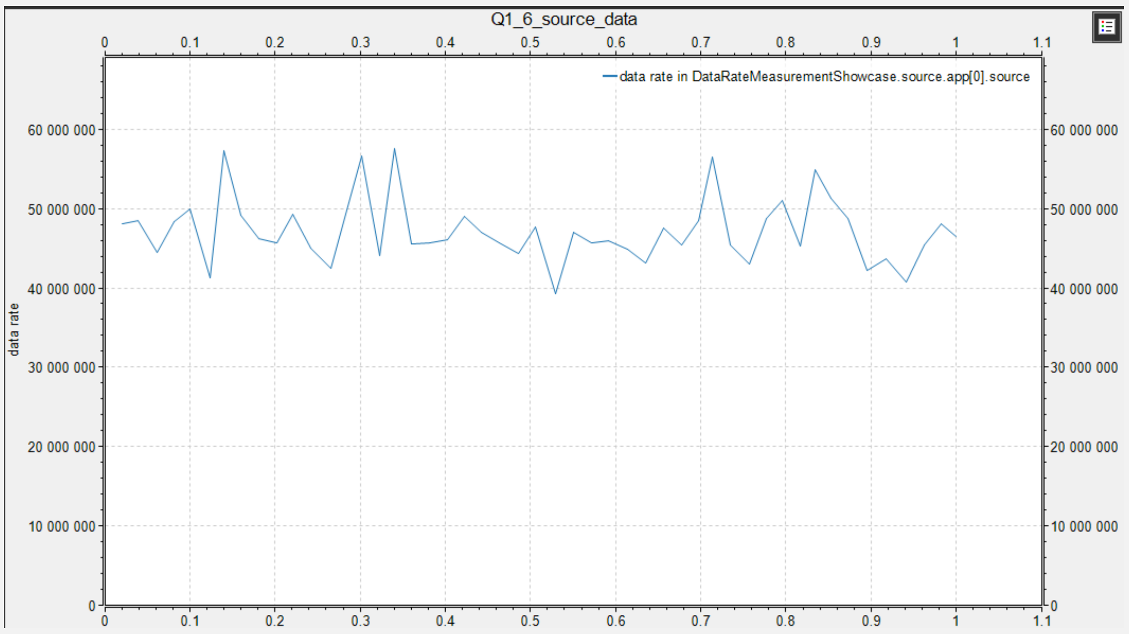


Figure 23. source data rate for part 6

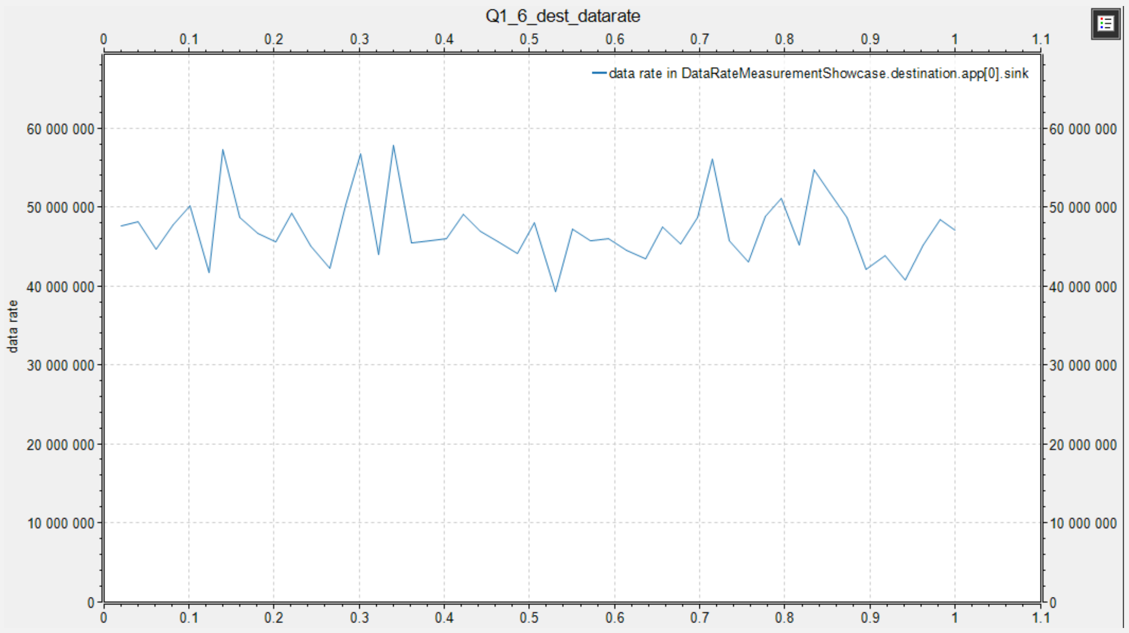


Figure 24. destination data rate for part 6

second practical question

part 1

In this question, you will simulate a communication link in which multiple packets will be sent from transceiver and received in the receiver. First we want to know alittle more about CRC checksum, after studying the way these codes are produced, it should be simulated in python code.First make a random binary array and then use an scrambling method on this binary array(scrambling method is one that evens the number of 1s and 0s in the array for statistical calculations) after that calculate CRC code for this array(you can choose small length for array for convenience). print input array,scrambled array and output array of this code. now imagine this code must be sent to receiver. make 1000 different arrays and for some of them change a number of bits(some arrays must be unchanged though). then in the receiver try to obtain main array from the received array and check if they are the same. show the effect of altering the bits on the error of decryption of CRC code in the receiver.(the protocol of encryption and decryption of CRC must be explained briefly).

solution

The **Cyclic Redundancy Check (CRC)** is an error-detecting technique used to detect accidental changes to data during its transmission over networks. The CRC process involves:

1. **CRC Generator (Sender Side)**: Data is first appended with zeros, the number of which is calculated as $k-1$ where k is the number of bits in the polynomial used for the CRC. The sender then performs modulo binary division of the data using the polynomial, expressed in binary form. The remainder from this division is appended to the data before it is sent.
2. **CRC Checker (Receiver Side)**: The receiver performs the same modulo binary division on the received data. If the remainder is zero, the data is considered correct. If the remainder is not zero, it indicates that errors occurred during transmission.

Example of CRC calculation:

- **Polynomial Used**: x^3+x^2+1, which is converted to binary as 1101.
- **Data Processed**: Starting with data 100100, appending three zeros results in 100100000 (since $k=4$).
- **Modulo Division**: The division at both sender and receiver sides leads to a zero remainder, indicating no errors in transmission.

to implement this in python, i wrote a class for transceiver and a class for receiver. to implement the algorithm in the transceiver, i used the following code in figure 27 and to implement the algorithm in the receiver, i used the following code in figure 28. the algorithm of the reciever is simple. it divides the input stream to data and CRC. then generates the CRC of the data and compares it with the received CRC. if they are the same, it returns the data, otherwise it returns an error message. in the transceiver, the data is first scrambled by the scramble method. then the CRC is calculated by the algorithm mentioned above. the polynomial used in my code is 0xEDB88320. it first initializes the CRC to 0xFFFFFFFF then for each byte in the data it xors the byte with the CRC.

after that it processes each bit of the byte as follows:

1. If the least significant bit (LSB) of CRC is set, the CRC is right-shifted and then XORed with a polynomial (0xEDB88320).
2. If the LSB is not set, it simply right-shifts the CRC.

the final CRC value is XORed with 0xFFFFFFFF to produce the checksum. here is an example CRC in transceiver and receiver:

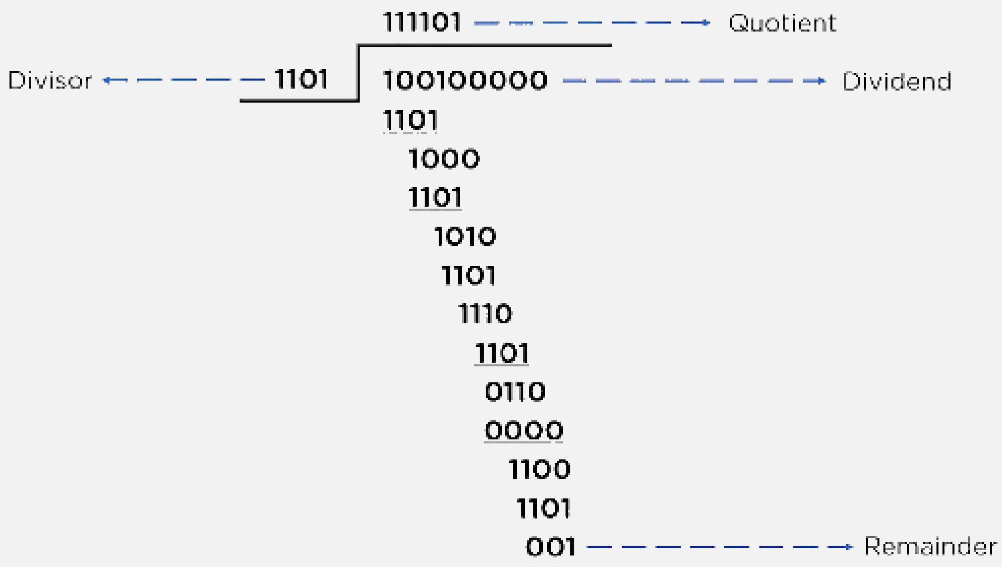


Figure 25. CRC Calculation in transceiver Example

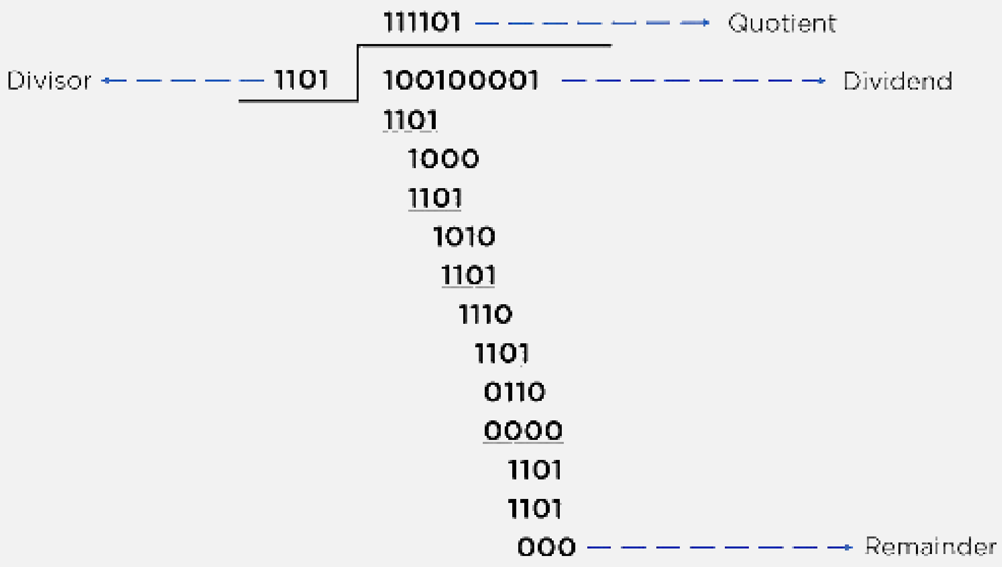
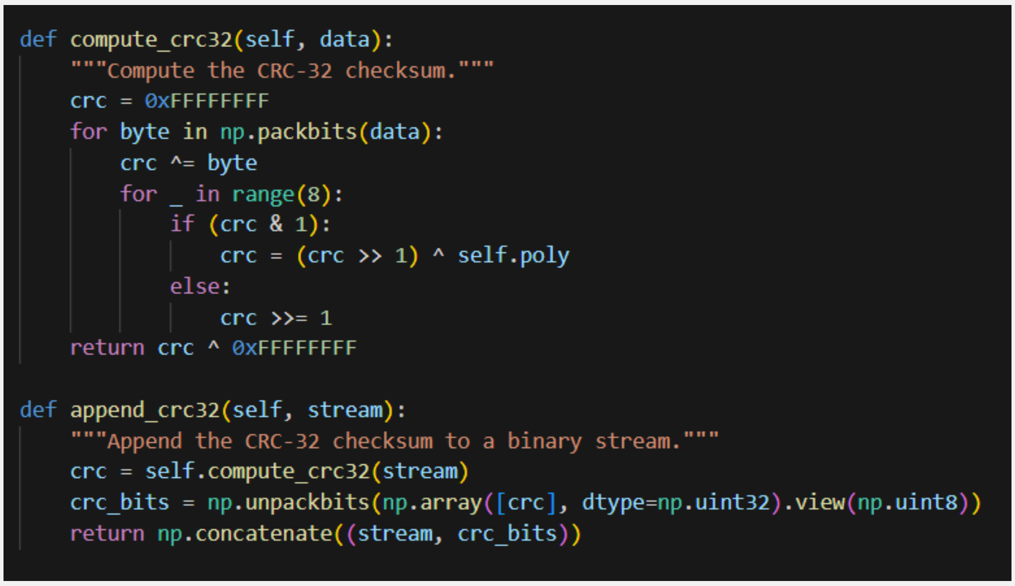
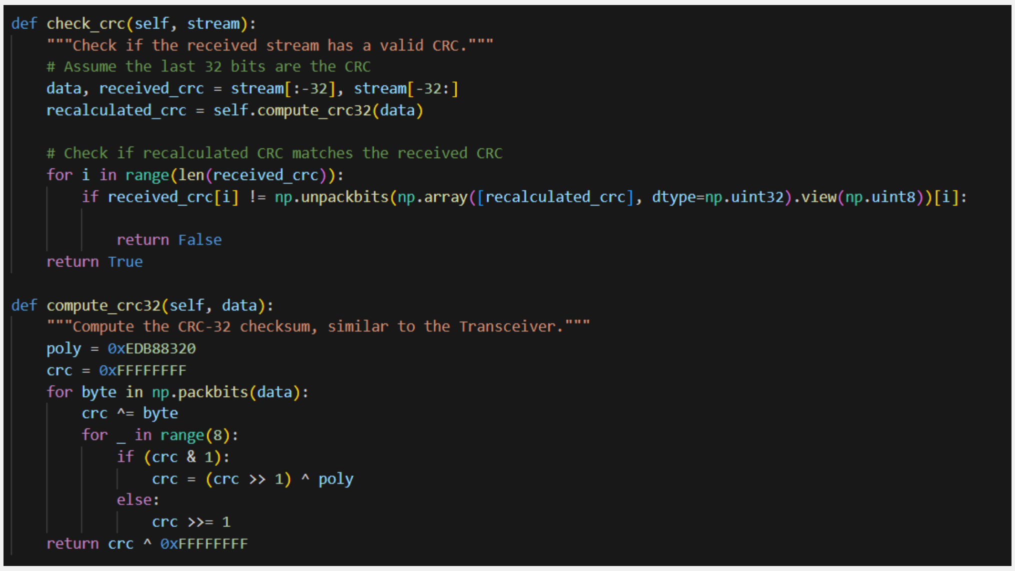


Figure 26. CRC Calculation in reciever Example



```python
def compute_crc32(self, data):
    """Compute the CRC-32 checksum."""
    crc = 0xFFFFFFFF
    for byte in np.packbits(data):
        crc ^= byte
        for _ in range(8):
            if (crc & 1):
                crc = (crc >> 1) ^ self.poly
            else:
                crc >>= 1
    return crc ^ 0xFFFFFFFF

def append_crc32(self, stream):
    """Append the CRC-32 checksum to a binary stream."""
    crc = self.compute_crc32(stream)
    crc_bits = np.unpackbits(np.array([crc], dtype=np.uint32).view(np.uint8))
    return np.concatenate((stream, crc_bits))
```

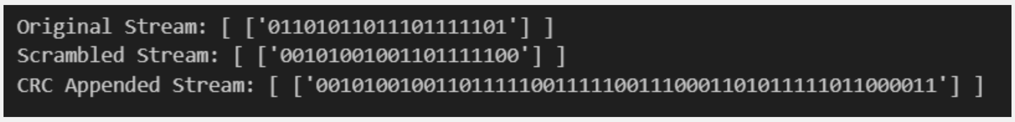
Figure 27. CRC Calculation in transceiver



```python
def check_crc(self, stream):
    """Check if the received stream has a valid CRC."""
    # Assume the last 32 bits are the CRC
    data, received_crc = stream[:-32], stream[-32:]
    recalculated_crc = self.compute_crc32(data)

    # Check if recalculated CRC matches the received CRC
    for i in range(len(received_crc)):
        if received_crc[i] != np.unpackbits(np.array([recalculated_crc], dtype=np.uint32).view(np.uint8))[i]:

            return False
    return True

def compute_crc32(self, data):
    """Compute the CRC-32 checksum, similar to the Transceiver."""
    poly = 0xEDB88320
    crc = 0xFFFFFFFF
    for byte in np.packbits(data):
        crc ^= byte
        for _ in range(8):
            if (crc & 1):
                crc = (crc >> 1) ^ poly
            else:
                crc >>= 1
    return crc ^ 0xFFFFFFFF
```

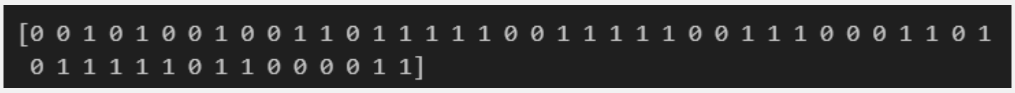
Figure 28. CRC Calculation in transceiver

a method called scramble-stream is in the class of transceiver that scrambles the input stream. it first calculates the number of 1s and 0s in the stream. then if the number of 1s is greater than the number of 0s, it change some of the 1s to 0s and vice versa. the output for the input data, scrambled data and crc appended data is as follows:



```
Original Stream: [ ['01101011011101111101'] ]
Scrambled Stream: [ ['00101001001101111100'] ]
CRC Appended Stream: [ ['0010100100110111110011111001110001101011111011000011'] ]
```

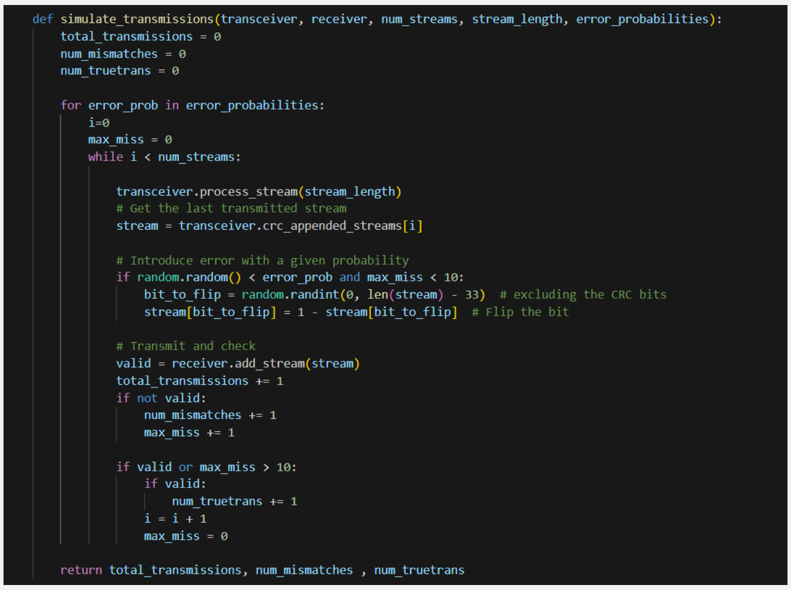
Figure 29. ouput of scrambled data and crc appended data by using built-in method in python the CRC is the same as our function. this output is as follows:



```
[0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0 1
 0 1 1 1 1 0 1 1 0 0 0 0 1 1]
```

Figure 30. ouput of built-in crc appended data

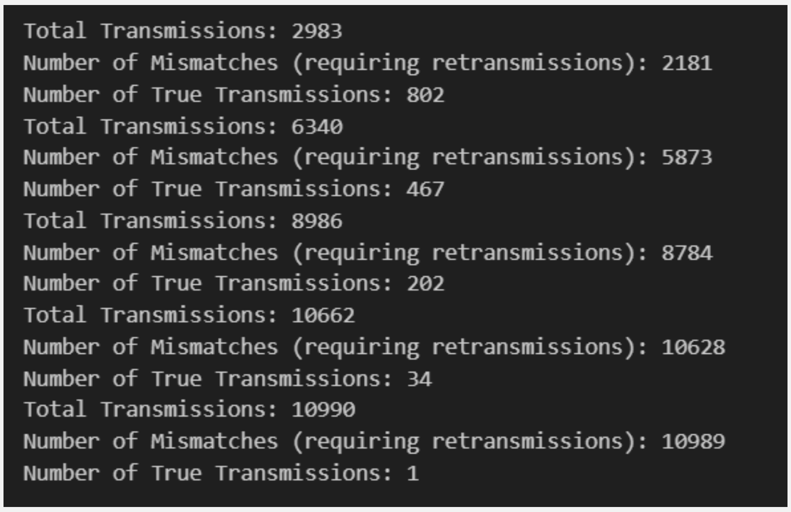
then to see functionality of the receiver, i generated 1000 random arrays and for some of them i changed some bits. this code is as follows:



```python
def simulate_transmissions(transceiver, receiver, num_streams, stream_length, error_probabilities):
    total_transmissions = 0
    num_mismatches = 0
    num_truetrans = 0

    for error_prob in error_probabilities:
        i=0
        max_miss = 0
        while i < num_streams:

            transceiver.process_stream(stream_length)
            # Get the last transmitted stream
            stream = transceiver.crc_appended_streams[i]

            # Introduce error with a given probability
            if random.random() < error_prob and max_miss < 10:
                bit_to_flip = random.randint(0, len(stream) - 33)  # excluding the CRC bits
                stream[bit_to_flip] = 1 - stream[bit_to_flip]  # Flip the bit

            # Transmit and check
            valid = receiver.add_stream(stream)
            total_transmissions += 1
            if not valid:
                num_mismatches += 1
                max_miss += 1

            if valid or max_miss > 10:
                if valid:
                    num_truetrans += 1
                i = i + 1
                max_miss = 0

    return total_transmissions, num_mismatches , num_truetrans
```

Figure 31. code to test the receiver

the output for error probability of 0.2,0.4,0.6,0.8,0.95 is as follows:



```
Total Transmissions: 2983
Number of Mismatches (requiring retransmissions): 2181
Number of True Transmissions: 802
Total Transmissions: 6340
Number of Mismatches (requiring retransmissions): 5873
Number of True Transmissions: 467
Total Transmissions: 8986
Number of Mismatches (requiring retransmissions): 8784
Number of True Transmissions: 202
Total Transmissions: 10662
Number of Mismatches (requiring retransmissions): 10628
Number of True Transmissions: 34
Total Transmissions: 10990
Number of Mismatches (requiring retransmissions): 10989
Number of True Transmissions: 1
```

Figure 32. output of total transmissions for different error probabilities

as we expected number of errors increases with the increase of error probability.

the plot of number of transmissions, errors, and true transmissions by error probability is as follows:

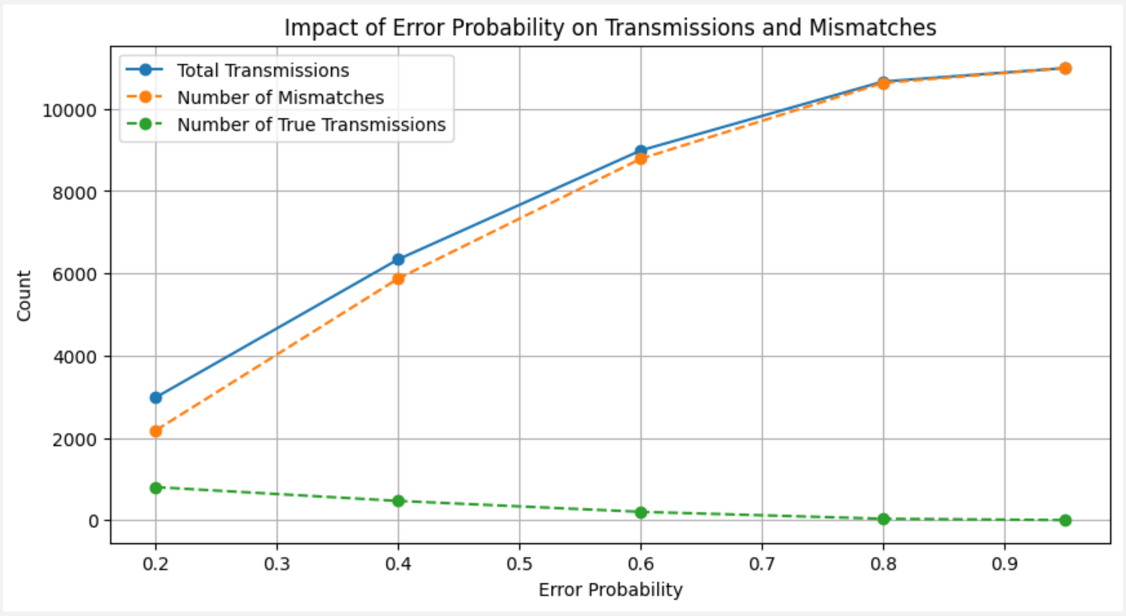


Figure 33. plot of transmission info by error probability

part 2

Now we want to expand this over a selective repeat protocol. make 10 different small random binary arrays as packets now, and calculate CRC for each array and attach it to the array. now those packets are ready to be sent. change some bits randomly in the arrays and simulate selective repeat protocol. Here you should show these items:

solution

for this part i implemented a function to simulate the selective repeat protocol. you can check that in ipynb file named simulate-selective-repeat.the function gets these parameters as input:

1. **transceiver**: an instance of the transceiver class.
2. **receiver**: an instance of the receiver class.
3. **window-size**: the size of the window.
4. **num-streams**: the number of streams to be sent.
5. **stream-length**: the length of each stream.
6. **error-probability**: the probability of error in transmission.
7. **max-attempts**: the maximum number of attempts to send a stream.
8. **number-of-flips**: the number of bits to flip in each stream if error occurs.

and gives these outputs:

1. **total-transmissions**: the total number of transmissions.
2. **num-mismatches**: the number of mismatches in the received streams.
3. **num-truetrans**: the number of true transmissions.
4. **acks**: the acknowledgments received.
5. **timeline**: timeline of acks and nacks.

you can check the logs of the function in the ipynb file. i plotted transmission info by error probability, maximum attempts, and number of flips. the plots are as follows:

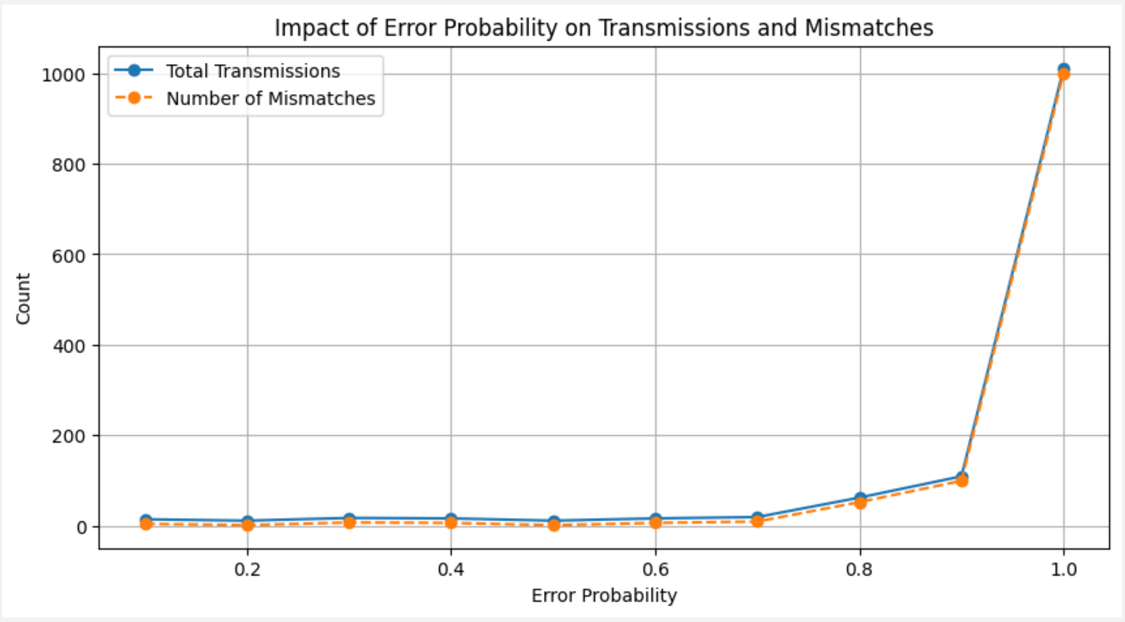


Figure 34. plot of transmission info by error probability

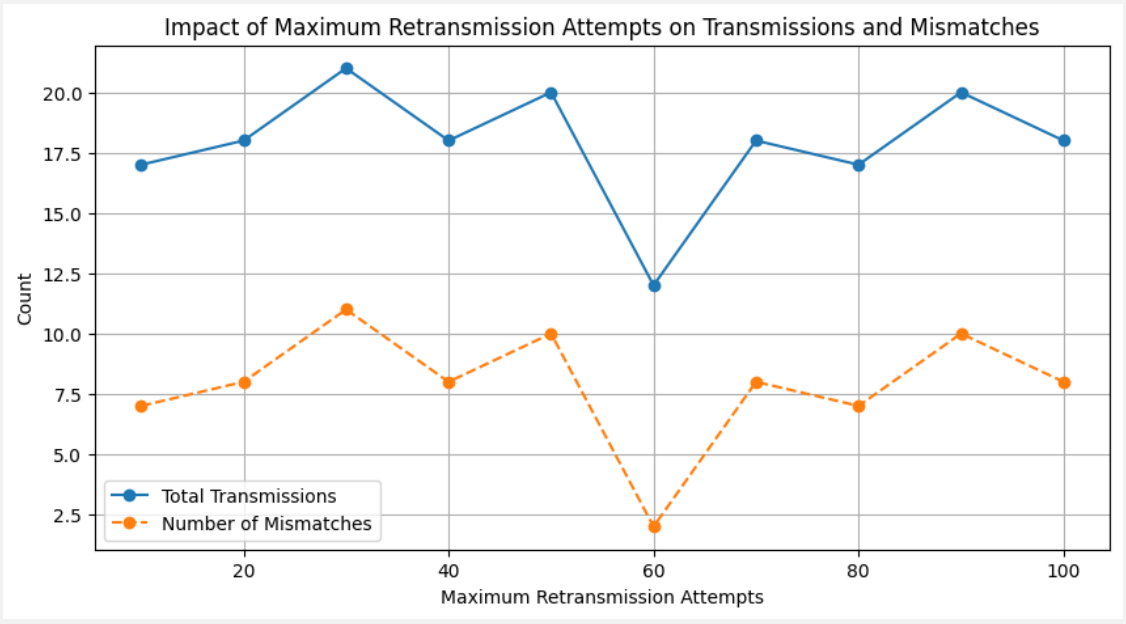


Figure 35. plot of transmission info by maximum attempts

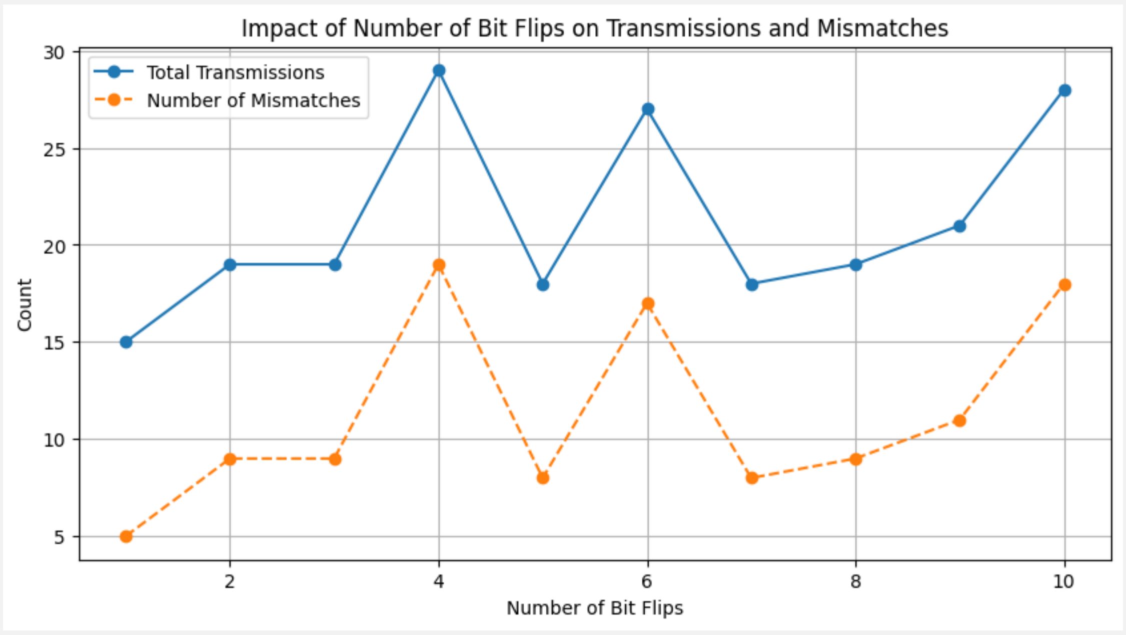


Figure 36. plot of transmission info by number of flips

the plot of transmission info by probability is as we expected. number of transmissions and mismatches increases with the increase of error probability. but the other two plots doesnt have a logic. our CRC code in reciever only checks if the CRC is correct or not. so the number of flips and maximum attempts should not affect the number of transmissions and mismatches. so these two plots only depend on the error probability of channel which is fixed to 0.2 in my simulations and so they have a random behavior.

then i plotted the timeline of acks and nacks for fixed error probability of 0.6. acks are shown as green squares and nacks are shown as red squares. you can rerun the block for that in the ipynb file to see the timeline and its randomness.

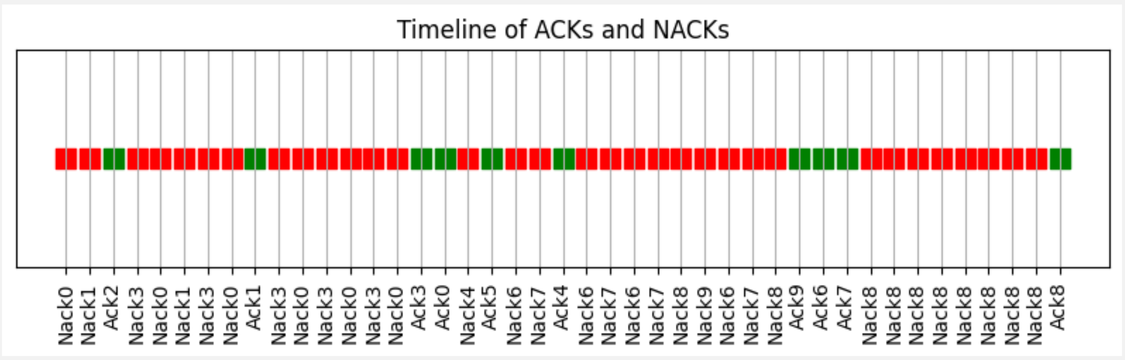


Figure 37. timeline of acks and nacks

part 3

In this part we want to compare stop-and-wait, selective-repeat and Go-Back-n protocols. make random arrays and simulate these three methods by changing a number of bits. plot the number of tries base on the number of changed bits in the simulation(the number of changed bit can also be referred as SNR) show these outputs on the same plot for better comparison(you can use small number of packets with small length and you do not have to implement CRC in this part)

solution

for this part i implemented stop-and-wait and Go-back-N protocols in two different function which similar input and output as selective-repeat function which i explained before. the only difference is that stop-and-wait protocol doesnt have a window-size as input. for the logs check the ipynb file. i plotted timelines of these protocols below for a fixed error probability of 0.6.

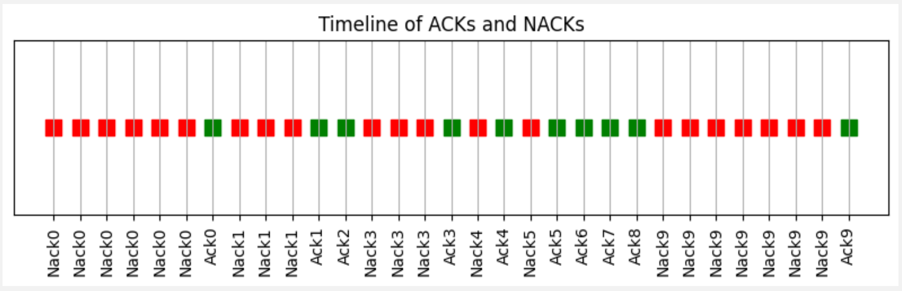


Figure 38. timeline of stop-and-wait protocol

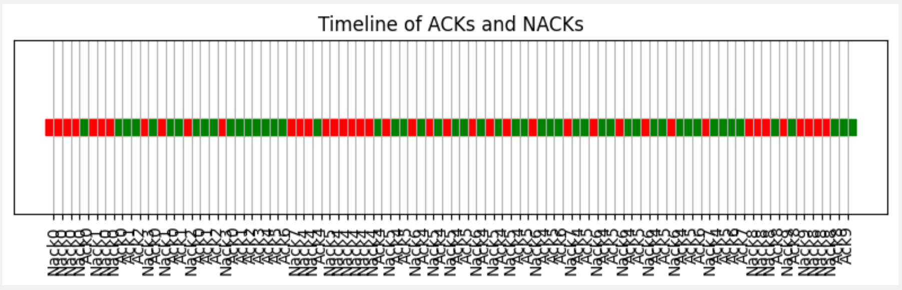


Figure 39. timeline of Go-back-N protocol

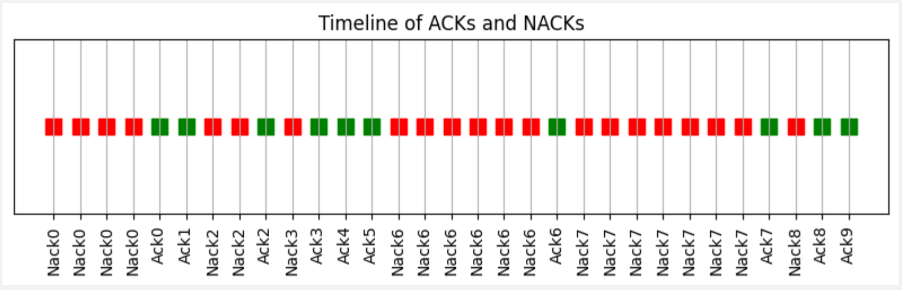


Figure 40. timeline of selective-repeat protocol

you can clearly see and confirm that these protocols works properly by these timelines.for the last part i plotted the number of total tries in figure 41 and number of true transmissions in figure 42 by the error probability. the number of total tries by number of flipped bits also shown in figure 43.

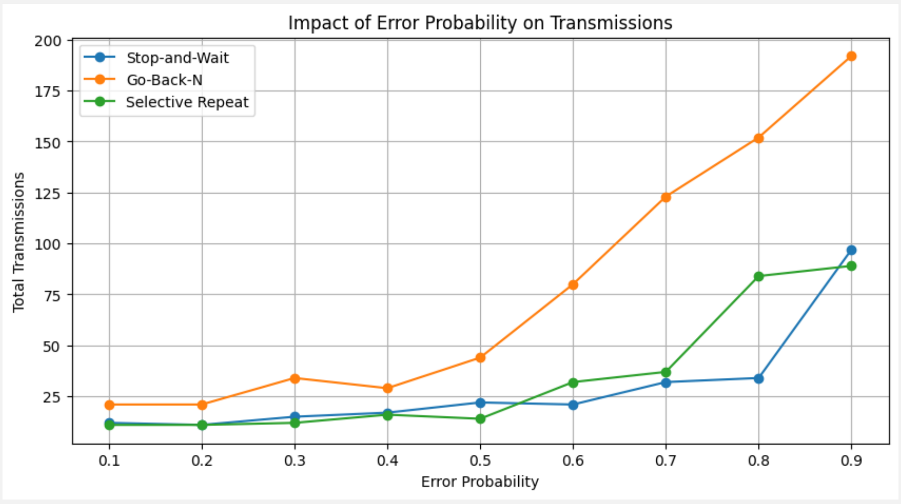


Figure 41. plot of total tries by error probability

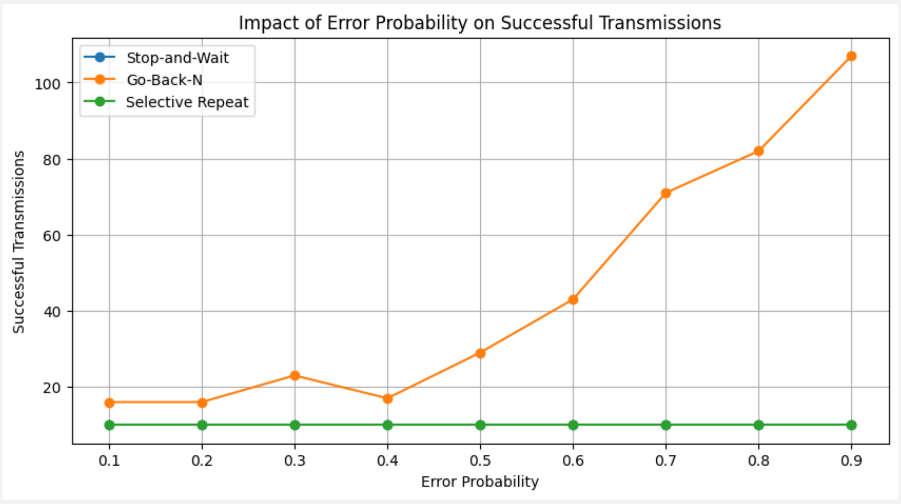


Figure 42. plot of true transmissions by error probability

the trend of the number of total tries is as we expected. it increases with the increase of error probability. hence the number of true transmissions is constant for selective-repeat and stop-and-wait protocols. but for Go-back-N protocol it increases as we expected. the number of total tries by number of flipped bits is random and doesnt have a logic. the reason is the same as the previous part.

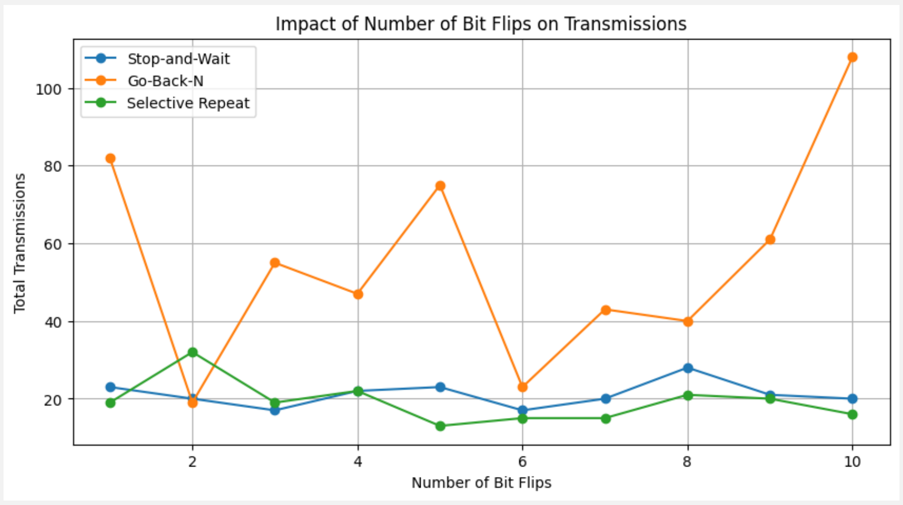


Figure 43. plot of total tries by number of flipped bits