Lab7: TCP, UDP and Transport Layer Services - Solutions

COM-208: Computer Networks

Objectives

- Understand the services of the transport layer
- Understand how TCP provides these services
- Get familiar with the ns-2 simulator

Prerequisites

- Textbook chapter 3

The Network Simulator: ns-2

ns-2 is a powerful simulator that provides substantial support for simulating TCP, routing and multicast protocols, among other things. It is capable of simulating the conditions that occur in wired or wireless (local & satellite) networks.

The simulator is written in C++. However, it uses OTcl as its command and configuration interface. In this TP, we will use scripts written in OTcl.

We will also use a network animator tool: Nam.

A Toy Example: 2 nodes communicating directly over UDP

The OTcl script for the first experiment is simpleSim.tcl. It creates two nodes and simulates sending data packets from one node to the other. The full explanation of the script can be found in DetailsSimpleSim.pdf file. You can find both simpleSim.tcl and DetailsSimpleSim.pdf on Moodle under Lab session 7.

Running simulations

You can run the script by typing the following command:

```
$ ns simpleSim.tcl
```

Graphical interface

When you click on the `play` button in the nam window, you will see that after 0.5 seconds of simulated time, node 0 starts sending data packets to node 1. The transmission stops at t = 1.5 seconds and the simulation stops at time t = 2 seconds. You might want to slow nam down by using the 'Step' slider.

Now, we can analyze the trace of the simulation.

Analyzing traces

When you run the script, you get a trace file named 'out.tr'. The format is the following:

event time src dest PktType PktSize Flags Fid SrcSddr DestAddr SeqNum PktId

where:

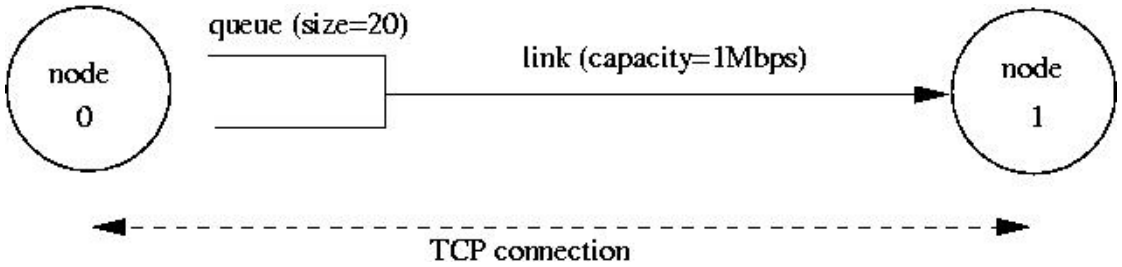
- event:
  - r: receive at dest
  - +: enqueue
  - -: dequeue
  - d: drop
- ScrAddr and DestAddr:
  - node.port (e.g., 0.1 means node 0, port 1)

Example trace

r 0.519 0 1 cbr 500 ------- 0 0.0 1.0 1 1
+ 0.52 0 1 cbr 500 ------- 0 0.0 1.0 4 4
+ 0.52 0 1 cbr 500 ------- 0 0.0 1.0 4 4

Exercises:

1. TCP Congestion Control: the congestion window and the slow start mechanism



1.1. Each TCP sender limits the rate at which it sends traffic as a function of perceived network congestion. In order to do this, TCP uses two mechanisms:

1. A varying congestion window, which determines how many packets can be sent before the acknowledgment for the first packet arrives.
2. A slow-start mechanism, which allows the congestion window to increase exponentially in the initial phase, before it stabilizes when it reaches threshold value. A TCP sender re-enters the slow-start state whenever it detects congestion in the network.

The tpWindow.tcl script implements the setup illustrated in the figure above. You can find it on Moodle under Lab session 7. The script takes two arguments:

1. the maximum value of the congestion window in number of packets (this the threshold value at which slow start stops, and TCP enters the normal operation state).
2. the delay of the link (in milliseconds).

```
$ ns tpWindow.tcl <cong_window> <link_delay>
```

Run tpWindow.tcl with a congestion window of 150 packets and a delay of 100ms:

```
$ ns tpWindow.tcl 150 100ms
```

It generates two traces, Window.tr, which keeps track of the size of the congestion window, and WindowMon.tr, which shows the parameters of the flow. The Window.tr file has two columns:

```
time congestion_window_size
```

The WindowMon.tr file has six columns:

```
time number_of_packets_dropped drop_rate throughput queue_size avg_tput
```

In order to plot the size of the TCP window and the number of queued packets, we use the gnuplot script Window.plot. You can find the script on Moodle under Lab session 7. We use the script as follows:

gnuplot Window.plot

What is the maximum size of the congestion window that TCP reaches in this case? What does TCP do when the congestion window reaches to this value? Why? What happens next?

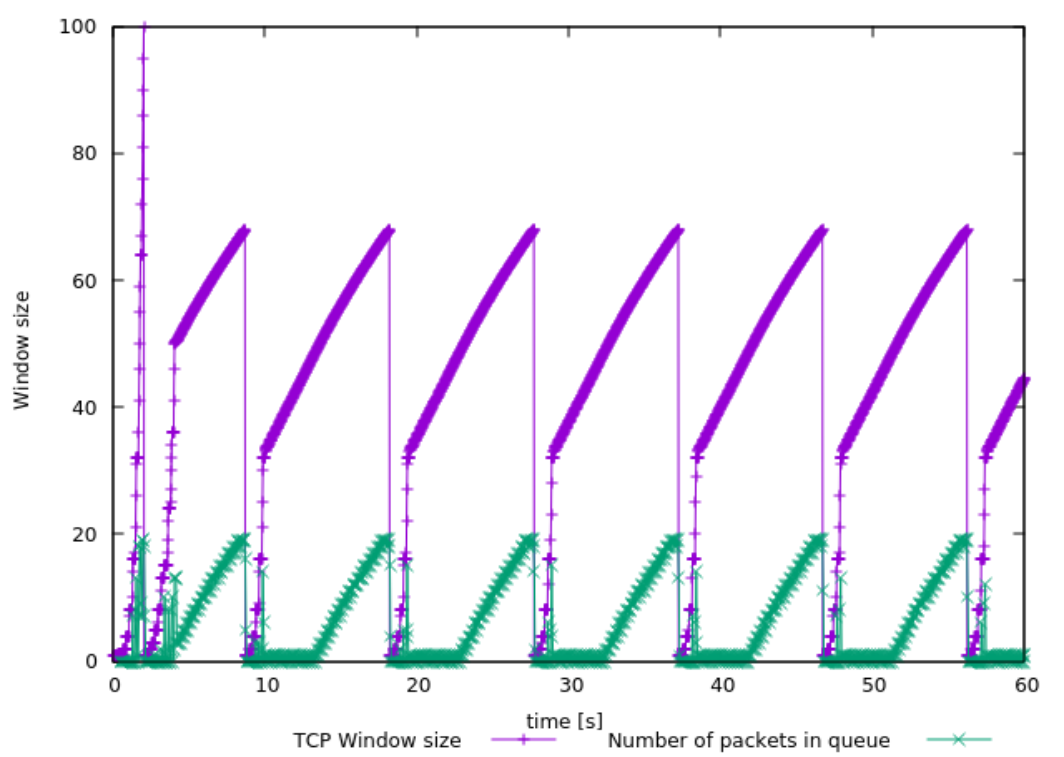


Figure 1: .

The congestion window grows up to 100 packets during the slow start phase, although we've set the threshold to 150 packets.

This is too big for the maximum size of the queue, which is only 20 packets. Thus, some packets get dropped. A retransmission timeout causes TCP to reduce the current congestion window to zero and use the half of the previous value as the new slow start threshold.

TCP, then, enters a new slow start phase and will oscillate between a slow start phase and a congestion avoidance phase; TCP goes back to slow start whenever the queue starts dropping packets again.

1.2 From the simulation script we used, we know that the payload of the packet is 500 Bytes (Keep in mind that the size of the IP and TCP headers is 40 Bytes). Use gnuplot

with the WindowTPut.plot script and find the average throughput of TCP in this case (both in number of packets per second and bps). You can find the script on Moodle under Lab session 7.

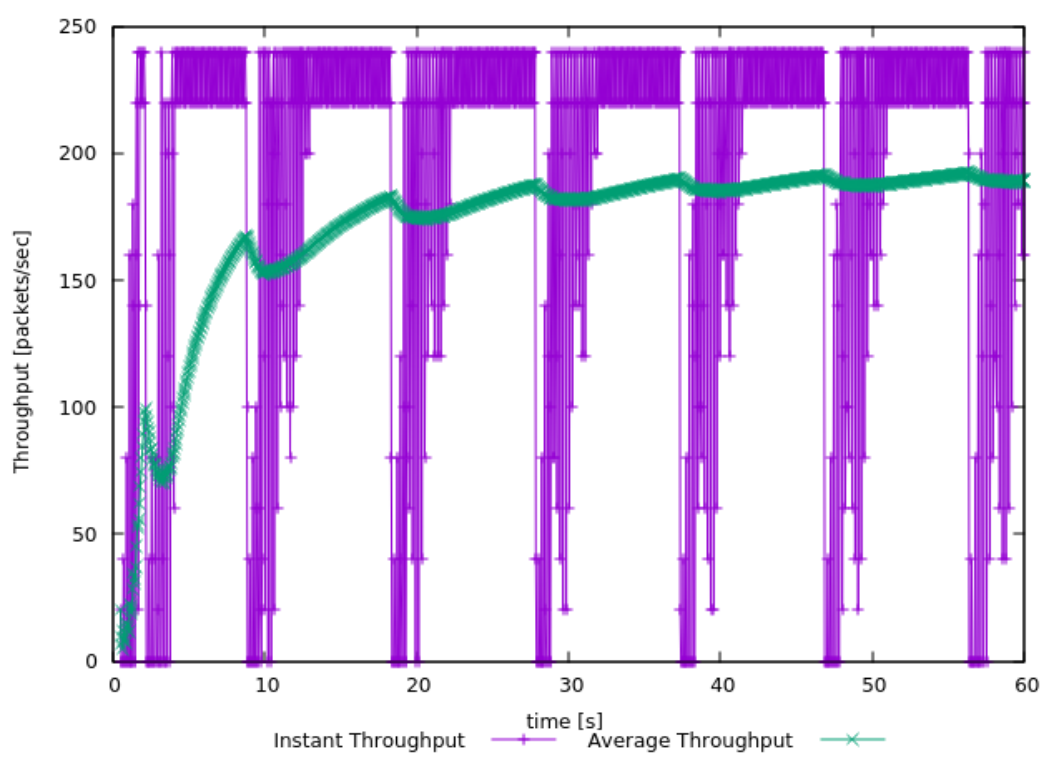


Figure 2: .

The average throughput of TCP in packets per second (pps) is 190 pps.

When it comes to calculating the throughput in bps, we have to make a distinction between:

1. The rate at which TCP transmits **any** kind of data; this includes header and payload data
2. The rate at which TCP transmits **useful** data; this only includes the payload.

Usually, it is the first definition that is more appropriate for throughput. Let's calculate the appropriate rates for both definitions of throughput, however:

1. According to the first definition, the throughput is 802.8 Kbps (190 * (500 + 40) * 8).
2. According to the second definition, the throughput is 760 Kbps (190 * 500 * 8).

1.3 Rerun the tpWindow.tcl script, each time with different values for the maximum congestion window size. Try the following maximum congestion window sizes: 30, 40, 50, 51, 52, 66, 67, 68, 90, and 100. How does TCP's window size respond to the variation of this parameter (as shown by gnuplot Window.plot)? Find the maximum congestion window size at which TCP does not oscillate (i.e., does not move up and down again) to reach a stable behavior.

Here are a few interesting cases for the initial value of the maximum congestion window:

- **Initial maximum congestion window size $<$ 67:** the oscillations stop after the first return to slow start. When we reduce the congestion window to half its size, this is enough to stabilize the number of packets in the sending queue; the queue never gets full, and so packets are not dropped anymore.
- **Initial maximum congestion window size $<$ 51 packets:** TCP stabilizes immediately. At this point, the average packet throughput is close to 225 packets per second.

What is the average throughput (in packets and bps) at this point (as shown by gnuplot WindowTpu.plot)? How does the actual average throughput compare to the link capacity (1Mbps)?

The average throughput 225*540*8=972 Kbps, which is almost equal to the link capacity.

1.4 The version of TCP whose behavior we observed so far is known as TCP Tahoe (which is the default TCP agent in ns-2). The newer version, TCP Reno, acts slightly differently in case it received a triple duplicate ACK (indicating that a packet has been lost, but that at least 3 subsequent packets were received, as it would occur if packets are dropped because a queue is full). In order to understand the difference, slightly modify the tpWindow.tcl OTcl script. Look for the following line:

```
set tcp0 [new Agent/TCP]
```

and replace it with:

```
set tcp0 [new Agent/TCP/Reno]
```

Compare the plot you got in Exercise 1.1 with the one you obtain for TCP Reno and explain the difference between the two implementations (Hint: compare the number of times the congestion window goes back to zero in each case). How does the average throughput differ in both implementations?

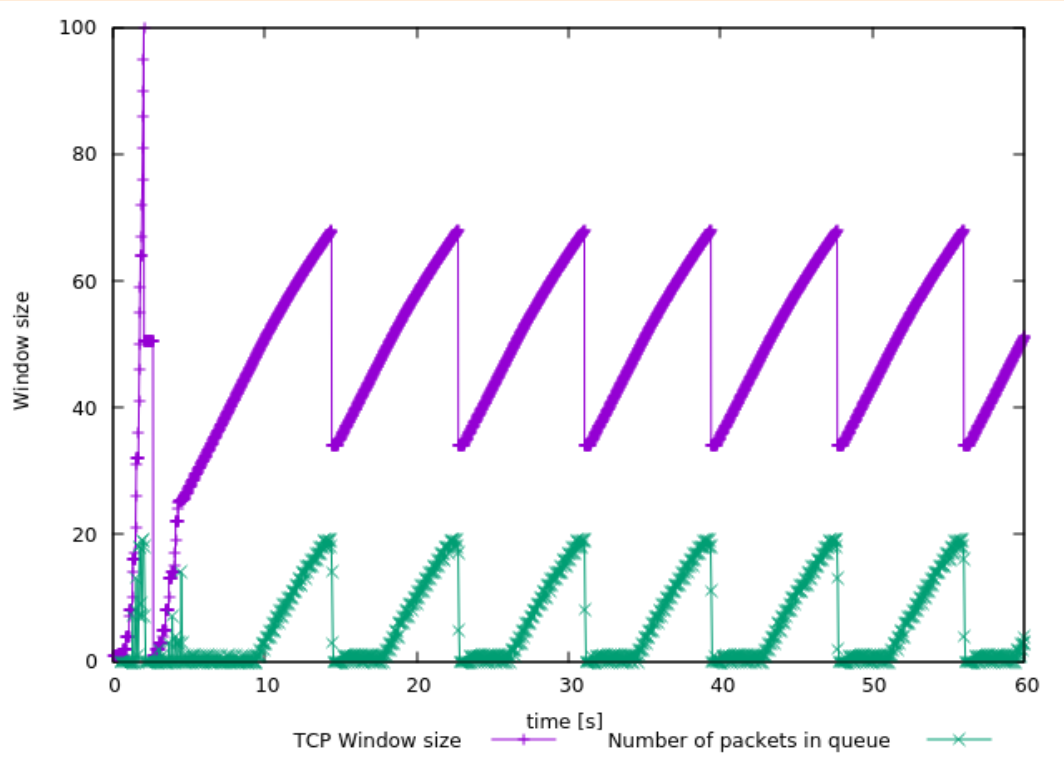


Figure 3: .

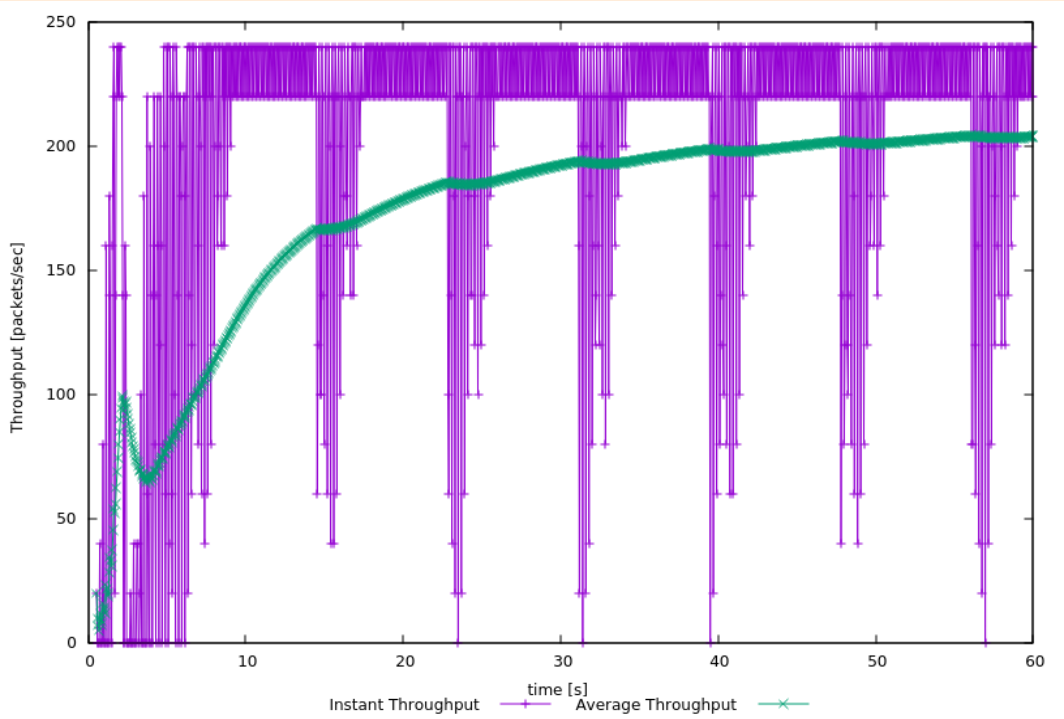


Figure 4: .

TCP Reno does not enter a slow start phase when a loss occurs (i.e., when it receives a triple duplicate ACK). Instead, it halves its current congestion window and increases it linearly, until losses starts taking place again. This process, called “Fast Recover”, is iterative and happens only when losses are detected due to triple duplicate-ACKs (as opposed to timeouts).

You can see that TCP Tahoe re-enters a slow-start phase when a loss occurs as shown by Figure 1, whereas TCP Reno does not as shown by Figure 3.

The throughput of TCP Tahoe in our case is about 190 packets per second as shown by Figure 2. The throughput of TCP Reno is slightly better; 200 packets per second as shown by Figure 4.

1.5 In this section, we are going to experiment with an induced loss rate of 2% (i.e. loss caused by link errors, and not by buffer space exhaustion) on the throughput of TCP. Run the tpWindowLoss.tcl script in ns-2. You can find the script on Moodle under Lab session 7. This script takes two arguments:

1. the maximum value of the congestion window in number of packets and
2. the delay of the link in milliseconds

ns tpWindowLoss.tcl 50 100ms

Use the following command to observe the behavior of the congestion window:

```
gnuplot Window.plot
```

What can we say about the congestion window oscillations, compared to Exercise 1.2?

Now, look at the throughput window:

```
gnuplot WindowTPut.plot
```

How does the throughput of TCP with induced losses compare to throughput of TCP without losses? (see Exercise 1.2)

Tip: If you want to save the figure that gnuplot generates (to a file named figure.eps), add the following two lines to the gunplot script before the plot command and remove the pause command:

```
set term postscript enhanced color eps
set output "figure.eps"
```

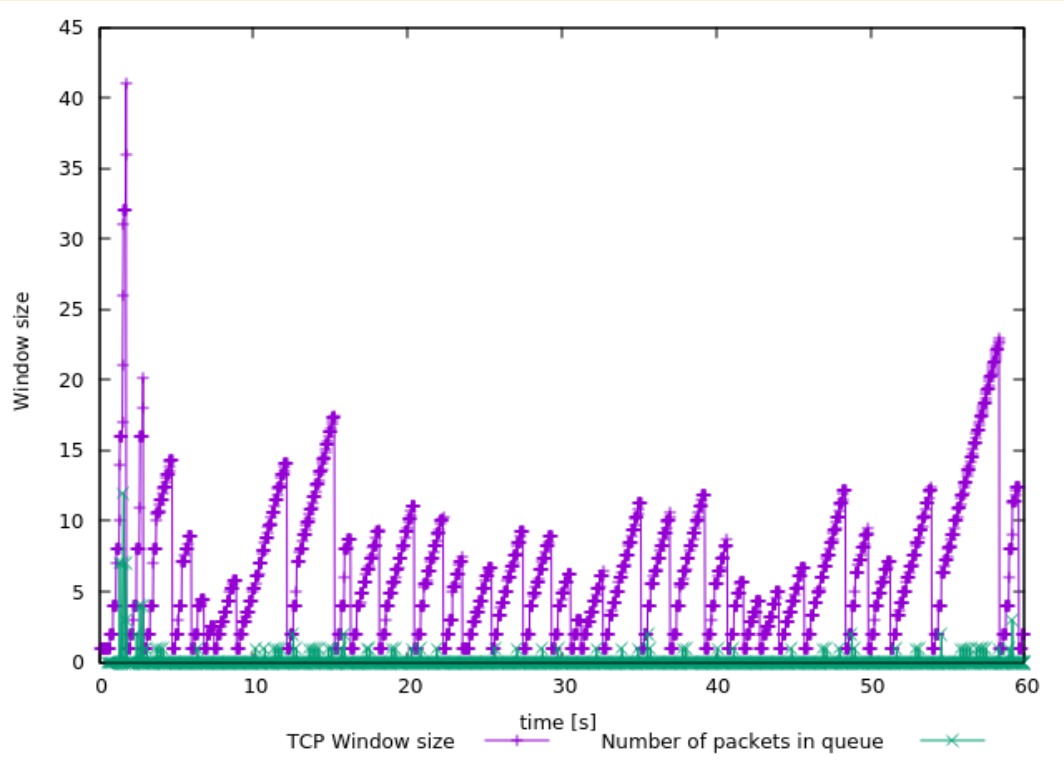


Figure 5: .

With a congestion window of 50 packets, in the case without losses (Exercise 1.3), TCP was not oscillating anymore, whereas here, the packet losses cause TCP to think the link is congested, and it reverts back to the slow start phase whenever that happens.

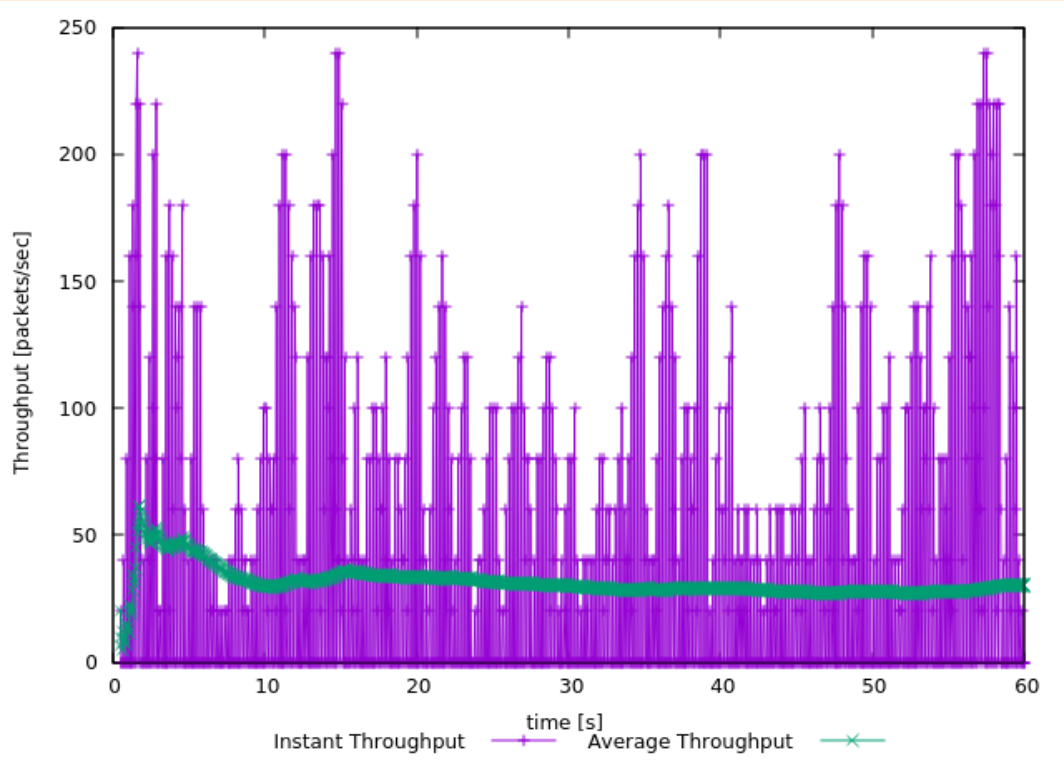


Figure 6: .

In this case, the average throughput is only around 30*540*8= 129.6 Kbps, which is about 1/7.5 of the throughput without losses.