# Computer Communications and Networks (COMN)
# 2020/21, Semester 2

## Assignment 2 Results Sheet

| | |
|---|---|
| **Forename and Surname:** | Mihai Ciobanu |
| **Matriculation Number:** | s1519734 |

**Question 1 –** Number of retransmissions and throughput with different retransmission timeout values with stop-and-wait protocol. For each value of retransmission timeout, run the experiments for **5 times** and write down **average number of retransmissions** and **average throughput**.

| Retransmission timeout (ms) | Average number of re-transmissions | Average throughput (Kilobytes per second) |
|---|---|---|
| 5 | 1128 | 61 |
| 10 | 750 | 71 |
| 15 | 107 | 71 |
| 20 | 92 | 70 |
| 25 | 104 | 62 |
| 30 | 95 | 64 |
| 40 | 95 | 59 |
| 50 | 107 | 53 |
| 75 | 98 | 45 |
| 100 | 101 | 35 |

**Question 2 –** Discuss the impact of retransmission timeout value on the number of retransmissions and throughput. Indicate the optimal timeout value from a communication efficiency viewpoint (i.e., the timeout that minimizes the number of retransmissions while ensuring a high throughput).

Considering that we have a 5ms one-way propagation delay, an ACK for a packet will come back after at least 10ms from sending it. This explains why the average number of retransmissions is so high for timeouts <= 10ms: the client waits too little for an ACK and blasts another packet (or even two for timeout=5ms) before an ACK even has a chance to arrive.

The ideal timeout value would be higher than 10ms, but not too high: just enough to account for the other types of delay (realistically, we are sending and receiving a packet at a time, so dqueue can't be high;

similarly, dproc is also low for the same reason and dprop can be safely ignored - we are sending packets from localhost to localhost) and the variability of the RTT generated by the loopback interface (which can get quite high at times for no discernible reason - see Question 4). This way, after waiting this long, we can be quite sure that either the packet or the ACK has been lost and retransmit the packet. We can see that this **optimal timeout value is 20ms,** as it minimises the average number of retransmissions and maximises the throughput.

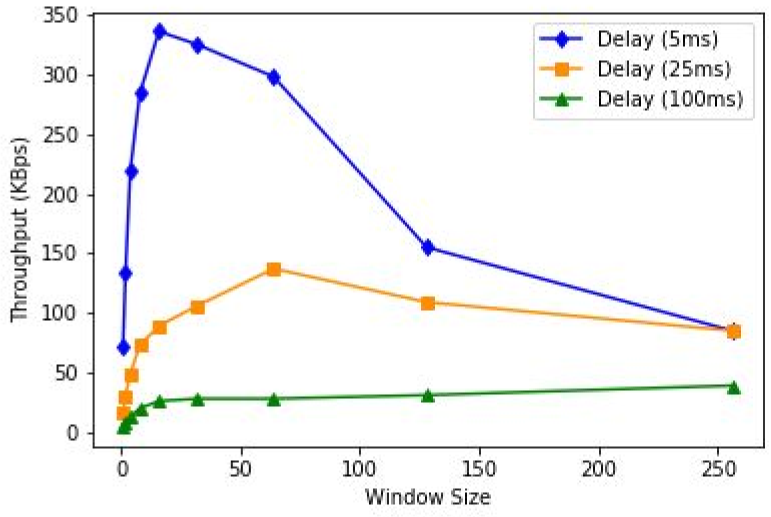
If we increase the timeout value further, we can see that the average number of retransmissions stays largely the same and that the throughput goes down. This makes sense: the packet loss rate stays constant and we are transferring the same file; it's only natural that the number of retransmissions stays the same if the timeout is optimal (e.g: we're sure that the packet has been lost). It's also natural that the throughput goes down, as the client spends more time waiting for roughly the same number of lost packets (we must remember that the packet loss remains constant, and that we're sending the same file in all the experiments).

In conclusion, the ideal timeout value should be higher than 1 RTT (=2 * one-way delay time) in order to minimise the number of retransmissions and lower than 2-3 * RTTs in order to maximise the throughput.

**Question 3 –** Experimentation with Go-Back-N. For each value of window size, run the experiments for **5 times** and write down **average throughput**.

| | **Average throughput (Kilobytes per second)** | | |
|---|---|---|---|
| **Window Size** | **Delay = 5ms** | **Delay = 25ms** | **Delay = 100ms** |
| 1 | 72 | 17 | 4.5 |
| 2 | 133 | 30 | 8 |
| 4 | 219 | 49 | 13 |
| 8 | 284 | 73 | 20 |
| 16 | 336 | 89 | 26 |
| 32 | 325 | 106 | 28 |
| 64 | 298 | 137 | 28 |
| 128 | 155 | 109 | 31 |
| 256 | 85 | 85 | 39 |

Create a graph as shown below using the results from the above table:



**Question 4 –** Discuss your results from Question 3.

**Optimal retransmission value**
By optimal, I assume that we use the same definition of optimality as in Question 2: "the timeout that minimizes the number of retransmissions while ensuring a high throughput". The number of retransmissions and the throughput are correlated. In general, the lower the number of retransmissions are, the higher the throughput will be. As we will see in Question 8 however, when our application is the only one using the link, a higher number of retransmissions might lead to higher throughput.

Timeout values used:
Delay 5ms - timeout value 20ms.
Delay 25ms - timeout value 66ms.
Delay 100ms - timeout value: 219ms.

**A) Initial formula for optimal retransmission value**
The formula for choosing an appropriate timeout value I started with was: 1*mean(RTT) + 2*std(RTT) + epsilon.

Explanation:
1. The optimal timeout value must ensure that for any window size, we avoid resending a window unnecessarily. For this, we need to allow sufficient time to pass between sending a packet and deciding that either itself or its ACK has been lost. If neither the packet nor its ACK are lost in transit, let the time between sending the packet and receiving its ACK be a random variable RTT. By setting the delay to D, I assumed that RTT is normally distributed, with mean(RTT) approximately equal to 2*D + a tiny constant (3-4ms). Both mean(RTT) and std(RTT) will be estimated for each delay D by pinging localhost 100 times, with packets of size 1027 (thus emulating our application' payload). Setting a timeout value greater than or equal to 1*mean(RTT) + 2*std(RTT) will guarantee that 97.8% of the time the RTT will be smaller than the timeout value, thus giving the ACKs a fair chance to be received before a timeout is triggered.

2. Epsilon accounts for the time it takes the Receiver to get the packet number from a received packet, save the data (or discard it) and send the ACK to the Sender.

3. As the timeout increases above the optimal value, the average throughput will go down. This happens because the Sender has to wait more for every packet loss before resending the window (since the time between the last ACK received for the missing packet and the resending of the window is approximately equal to the timeout value).

4. As the window size grows larger, the cost of re-sending a whole window increases proportionally. Therefore, we could adapt our timeout values based on the window size: we can have lower timeout values at smaller window sizes (where we - and the network - can tolerate some unnecessary resends and higher timeout values at larger window sizes, where resends become more expensive for the network and for us.
e.g: for a window size of 1, we could use a value x for which P(RTT<=x) = 0.7, while for a window of size 256 we might be more stringent and use a value of x for which P(RTT<=x) = 0.95.

**B) Simplified model for estimating the effect of the delay on throughput**
Let us assume a simplistic approximation to GBN, in order to model our expectations. Every lost packet is always the first packet in a window and ACKs cannot be lost. Also, assume that dqueue and dproc are 0.
L1 = Sender -> Receiver packet length. Equal to 1069 bytes (data + UDP header + IPV4 header)
L2 = Receiver -> Sender ACK length. Equal to 10 bytes.
dprop = one-way propagation delay, equal to one of {5ms, 25ms, 100ms}
R = link bandwidth = 10Mbit/s = 1.25MB/s
w = window size
N = number of packets to transmit.
Probability that a window has to be re-transmitted (under this model): p=0.05.

Then, a window of packets reaches the Receiver in w*L1/R + dprop sec. Assuming that no packet is dropped in this window, the ACK for the last packet arrives after w*L1/R + 2*dprop + L2/R seconds after the Sender starts sending the packets in the window.
There are N/w windows. We would expect to have to retransmit p*N/w windows, therefore we need to send N/w * 1.05 windows.
The throughput can be estimated to be: NL1/[(1.05*N/w) * (w*L1/R + L2/R + 2*dprop)].
Let dprop1 = 5ms, dprop2 = 25ms, dprop3=100ms.. We would expect throughput1/throughput2 to be (w*(L1/R) + L2/R + 2*dprop2)/(w*L1/R + L2/R + 2*dprop1)

Below, I have included a table containing estimated throughput ratios in relation with the chosen window size.

So, according to this approximation, we would expect for the throughputs at large window sizes to be affected less by the delay than the throughputs at low window sizes, due to the fact that the w*L1/R terms become non-negligible relative to the delay as w grows.

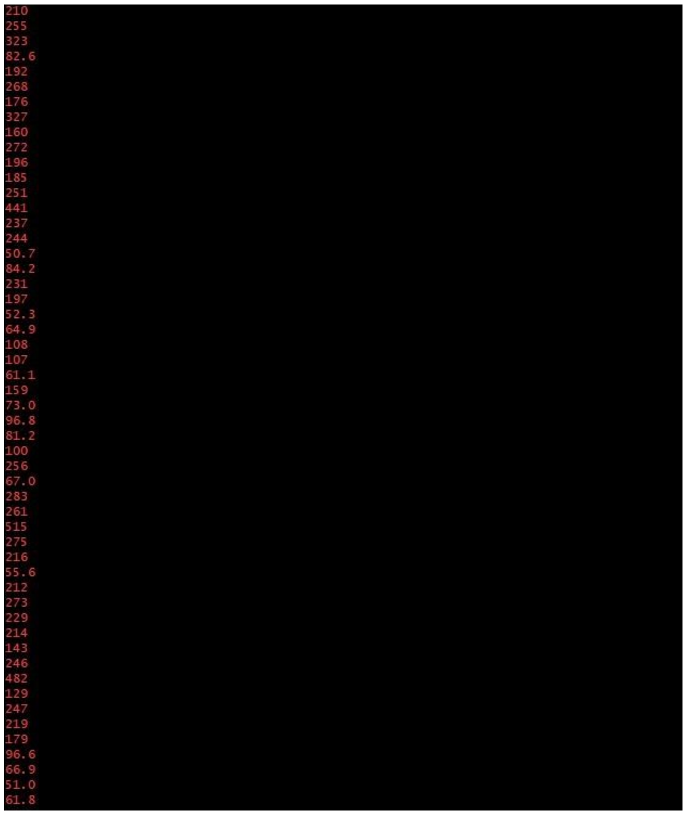
Naturally, as the delay increases, we expect the throughput to go down for any window size.

| Window size | Throughput (5ms)/Throughput(25ms) | Throughput(5ms)/Throughput(100ms) |
|---|---|---|
| 1 | 4.67 | 18.45 |
| 2 | 4.40 | 17.18 |
| 4 | 3.97 | 15.12 |
| 8 | 3.37 | 12.26 |
| 16 | 2.68 | 9.01 |
| 32 | 2.06 | 6.08 |
| 64 | 1.62 | 3.93 |
| 128 | 1.33 | 2.59 |
| 256 | 1.17 | 1.83 |

**C) Estimating the mean and std of the RTT**

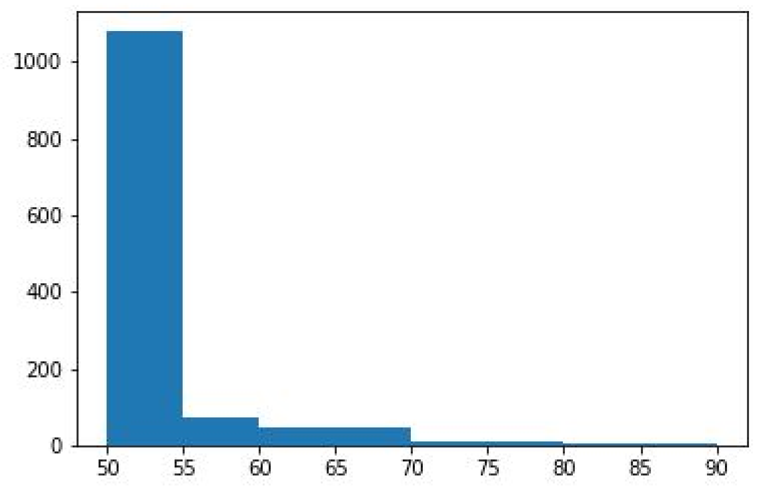
Estimating these values proved to be more difficult than expected. My intention was to estimate them from the delays we get from pinging localhost. Most of the time the RTT has a value close to 50ms, as expected. Sometimes though, as can be seen in Piazza posts @267 and @269 and in the screenshot below, the delays take very high values for a time. After many VM restarts and a lot of pings, I can conclude that the RTT never dips below 2x the delay we set using TC.

Unusual ping delays: (netem delay = 25ms).

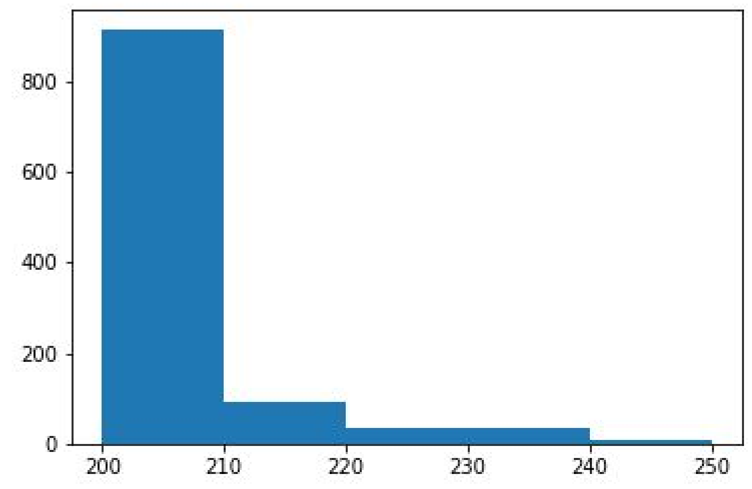


```
210
255
323
82.6
192
268
176
327
160
272
196
185
251
441
237
244
50.7
84.2
231
197
52.3
64.9
108
107
61.1
159
73.0
96.8
81.2
100
256
67.0
283
261
515
275
216
55.6
212
273
229
214
143
246
482
129
247
219
179
96.6
66.9
51.0
61.8
```

For a delay of 25ms, the distribution of RTTs can be seen below (x-axis = RTT vals, y-axis=counts).



And for a delay of 100ms:



This is clearly not a normal distribution. It resembles a Poisson distribution, but scipy.stats does not have fit procedures for discrete distributions. Therefore, as an estimation we will just choose the RTT value from the samples we've collected which is higher than 95% of the samples (an approximation of the CDF of the distribution of RTTs). This is how the initial timeout values were chosen:

Delay = 25ms -> optimal timeout: 66ms
Delay = 100ms -> optimal timeout: 219ms.

**D) Discussion on results**

1. In the table below, I have included the actual throughput ratios based on the delays and window sizes. A red cell indicates that the actual ratio value is lower than the predicted value and a green cell indicates the opposite.

| Window size | Throughput (5ms)/Throughput(25ms) | Throughput (5ms)/Throughput(100ms) |
|---|---|---|
| 1 | 4.23 | 16 |
| 2 | 4.43 | 16.62 |
| 4 | 4.47 | 16.85 |
| 8 | 3.89 | 14.2 |
| 16 | 3.77 | 11.6 |
| 32 | 3.07 | 6.08 |
| 64 | 2.18 | 10.64 |
| 128 | 1.42 | 5 |
| 256 | 1 | 2.18 |

We can see that our predictive model in B) predicts the throughput ratios for window sizes < 64 pretty well.
However, we see that for window sizes of 64, 128 and 256, and a delay of 100ms the actual ratios are twice the predicted ratios, which means that the throughputs at 100ms delay are much lower than expected. This is due to the fact that our assumption that dqueue = 0 is provably false for large windows.

We will examine the case of a window size = 64 and a delay of 100ms.
Our link's maximum bandwidth is 10Mbit/s = 1.25MB/s = 1.25KB/ms.
When we send or resend a full window of packets, we send 64 * 1069 = 68.416 KB of data per window.
Experiments reveal that those packets are all sent to the socket in roughly 10ms.
This means that we are sending in data at a rate of 6.8416KB/ms.
Which means that we basically saturate the link; the ACKs need to use the same link however, which means that they will be arriving much slower than expected.
An easy experiment revealed that after sending a full window, the ACK for the first packet in the window (lowest seq number) arrives in 80ms, even though the packet reaches the Receiver in 26ms (slightly higher than dprop) and the ACK for it is queued immediately. This of course triggers a timeout and a subsequent resend of the whole window. The waiting for ACKs period before the two resends allows the ACKs from the first window to reach the sender (unfortunately after it resends a whole window unnecessarily).

This phenomenon happens for the large window sizes at every delay value, which means that it is one of the factors why the throughput goes down at those window sizes (as we can see in the graph). The other factor is E) 2.

In short, large window sizes saturate the link (creating a packet queue at the sender) which causes a lot of unnecessary resends, which further drives down the throughput.

**E) Other observations**
1. Even at higher delays, we can see that GBN outperforms stop-and-wait, which is to be expected as it uses the available bandwidth more efficiently (by sending more packets at a time rather than a single packet).
2. We see that for any delay, having very large window sizes is not a good idea. If for instance the first packet in a large window is lost (and the other packets are transferred correctly) then the sender will have to retransmit the

whole window, leading to a significant loss in throughput. The receiver doesn't help either in this case: it simply discards all the packets in that window, making their successful transmission completely worthless.
3. If we expect the network to be congested, we should increase the timeout value to get a higher throughput.

**Question 5 –** Experimentation with Selective Repeat. For each value of window size, run the experiments for **5 times** and write down **average throughput**.

| | **Average throughput (Kilobytes per second)** |
|---|---|
| **Window Size** | **Delay = 25ms** |
| 1 | 16.70 |
| 2 | 31.81 |
| 4 | 56.36 |
| 8 | 100.74 |
| 16 | 170.38 |
| 32 | 283.42 |
| 64 | 440.30 |

**Question 6** - Compare the throughput obtained when using "Selective Repeat" with the corresponding results you got from the "Go Back N" experiment and explain the reasons behind any differences.

We used the same optimal timeout value (66ms) as the one used for the GBN protocol for a delay of 25ms. Both protocols have to send exactly the same file (split into the same number of chunks) over the same network conditions, which makes the comparison valid.

As the GBN and SR protocols are identical for a window size of 1, it's no surprise that in this case they yield extremely similar average throughputs (17 KBps vs 16.7KBps).

We notice that the difference between the average throughput of the SR protocol and the average throughput of the GBN protocol is proportional to the chosen window size: the bigger the window size, the better is SR's performance relative to GBN. This is completely normal. As we have established in the previous section, GBN's largest issue (in regards with the throughput) is that it discards out of order packets that were sent correctly, which is extremely inefficient.

In a large window, let's say of size 64, we expect 64 * 0.05=3 packets to be lost in transit on average.
Depending on where the first loss in the window occurred, GBN would have to re-transmit up to 64 packets from the window (if the first packet of the window was lost) and at least 3 packets (if only the last 3 packets were lost, which is rather improbable).

In the same situation, SR has to retransmit 3 packets + however many ACKs were lost in transit. Since we have the same loss, we would expect (64-3)*0.05=3 ACKs to be lost in transit. Therefore, SR would only need to re-transmit 6 packets on average (per full window sent). However, since SR does not have cumulative ACKs, how much the window moves is dependent on where the first ACK loss occurred. But how much GBN's window moves is dependent on where the first packet loss occurred. All things considered, on average SR will have to retransmit fewer packets than GBN, which naturally leads to a higher throughput.

There's also the issue of saturating the link. As we have seen in the previous part, sending a large number of packets in a short amount of time (aka sending a full window of packets, when the window size is big) is not a good idea, as for extreme cases it can increase the time the ACKs take to reach the Sender (in our experimental set-up at the very least) and substantially decrease the throughput. This is something that can happen in both protocols, but SR is more robust to it, as it has to send a full window much more rarely than GBN (same explanation as above: every time a packet is lost - assuming it's the first lost packet in a window, GBN sends a full window starting from that packet, while SR advances either to it or to a packet before it whose ACK is missing and sends only the missing packets in the new window).

Also, timeouts work quite differently in the two protocols. For GBN, once a packet is lost, the Sender will keep receiving ACKs for it until it times out. Instead of noticing duplicate ACKs (like in TCP for instance, but it wouldn't work here since the duplicate ACKs might be from a previous re-transmission), it keeps resetting the timer until it empties the buffer and a timeout occurs. If a duplicate ACK is late, it can arrive at the GBN sender just before it is about to timeout, thus re-starting the timer and delaying the re-send unnecessarily. This is not an issue for SR, since each packet has its own timer (which can only be reset when it expires) and its own ACK. Due to this reason, GBN may respond slower than SRto packet losses, which could also contribute to the difference in throughputs.

**Question 7** – Experimentation with *iperf.* For each value of window size, run the experiments for **5 times** and write down **average throughput**.

| | **Average throughput (Kilobytes per second)** |
|---|---|
| **Window Size (KB)** | **Delay = 25ms** |
| 1 | 10.38 |
| 2 | 19 |
| 4 | 28.4 |
| 8 | 51 |
| 16 | 65.84 |
| 32 | 68.25 |

**Question 8** - Compare the throughput obtained when using "Selective Repeat" and "Go Back N" with the corresponding results you got from the *iperf* experiment and explain the reasons behind any differences.

Observations:
1. We can observe that for any window size, TCP has a lower average throughput than both GBN and SR. This is to be expected due to TCP's congestion control feature.
2. We also notice that the average throughput of TCP goes up as we increase the window size.

Explanation:
1. In all three protocols, the window size represents the maximum possible in-flight data at that time.
In GBN and SR the window size remains constant throughout the application's lifetime: it maintains whatever window size we initialise the transfer with.
In TCP however, the window size changes dynamically during the transfer (a feature that guarantees congestion control). At any given time, the window size is lower than or equal to the minimum between the sender's congestion window size (cwnd) and the receiver's window size (rwnd) (which represents the amount of data that the receiver can handle at that time). rwnd's value ensures that the sender does not overwhelm the receiver (leading to more packet loss) and cwnd's value ensures that the sender does not overwhelm the network. These values are of course also updated dynamically: at every loss event (3 duplicate ACKs received by the sender), the sender cuts cwnd by half and the Receiver's rwnd value is updated at every ACK on the sender side. Naturally, this leads to a loss of throughput comparative to the two UDP-based transfer protocols.

As a result of this congestion control mechanism, TCP's maximum throughput is upper bounded by 1.22 * (MSS/RTT) * 1/sqrt(L) (formula given in lecture L18), where MSS = maximum segment size = 1KB in our case, RTT = 50ms and L = packet loss rate = 5%. Thus, in this experimental setup, TCP's maximum possible throughput is lower than or equal to 109.12 KBps (regardless of the chosen window size), a value which as we have seen can be achieved (and overtaken) by both SR (for window sizes=16, 32 and 64) and GBN (for a window size of 64).

2. Comment:
If I understand iperf correctly, its "-w" option sets the maximum window size for cwnd and rwnd (assumption which was validated empirically).

The maximum window size assuming that the 1 Mbit/s link is utilized fully (+ ignoring slow start) is RTT*(1 MBit/s)/MSS = 62.5 segments. This suggests that even if we set the window size to a value higher than 62KB, the actual window sizes used throughout the transfer will be smaller.

Also, from our experiments for a window size of 32KB cwnd is higher than 10KB only 4% of the time and is never higher than 14KB. The same situation applies for the 16KB-window experiments: most (>95%) of the values for cwnd throughout the run are below 10KB. The max window size we set with iperf seems to be restricting (too small) only for the other window sizes (1, 2, 4, 8KB), since in these cases cwnd frequently reaches this upper limit. This is the reason why the throughput goes up as we increase the max window size limit then seems to hit a plateau for the highest two window sizes.

(The last two paragraphs also serve as further proof for observation 1).