# Introduction to Computer Networks

# Homework #2

# Submission by:

* Alexander Shender 328626114
* Netanel Rotschild 204937841

## Question 1.

### 

The calculation of the throughput is similar to the exercise in the Tutorial lecture, where *n* packets were sent every time. In our case, n=2. The equations are still fully developed here:

Given that we send 2 frames together:

Thus:

Calculating the throughput:

### 

As we can see, the throughput is dependent on 2 parameters – and . By taking the throughput from the original protocol, which is:

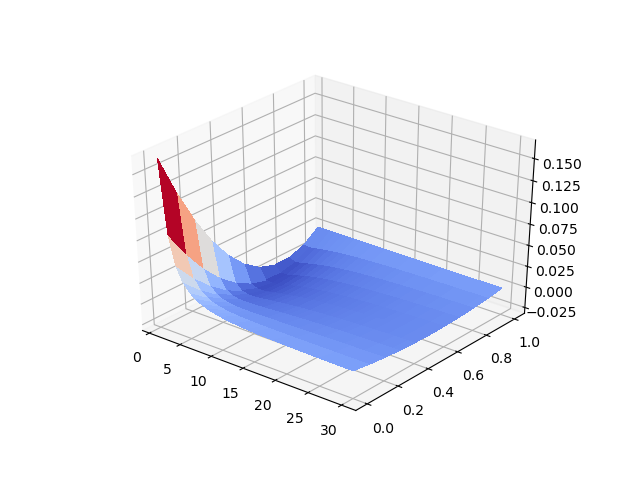
We are required to find the for which the equations holds:

For different values, the which results in the inequality is different. We use the graph to visualize. Using python, the throughput values were calculated for various values. 2 graphs are shown:

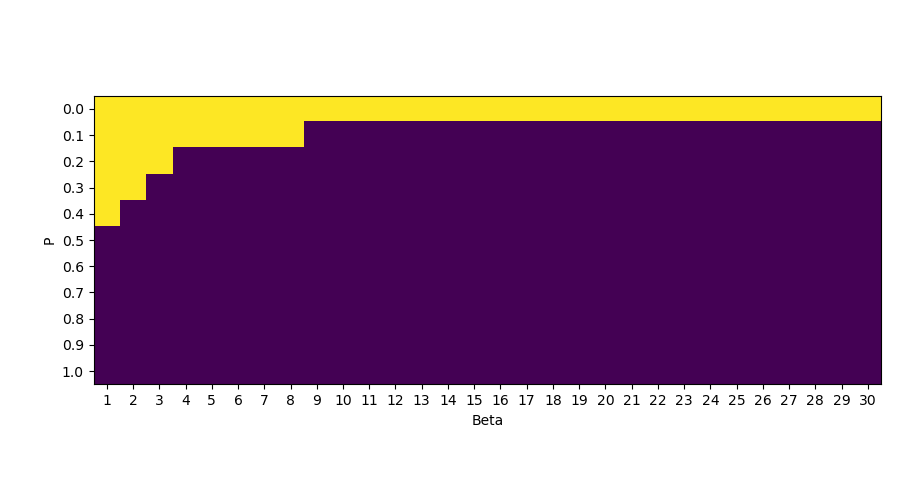
The first graph shows the result of the difference of the throughput values:

Where this values is positive, meaning using the original protocol is beneficial. Ranges used:

Result:



As we can see, where the probability of packet not being delivered is low, and Beta is low (propagation time is small), the original algorithm is more beneficial. To be sure, another binary chart shows yellow bricks where : (and purple if other)



We can observe that:

* With high error probability, the new protocol is beneficial
* If the error value is low, for certain Beta values we prefer the original algorithm
* If the error probability is 0, original protocol is better

All of those conclusions are also intuitive.

### 

Now, in the equation for the throughput calculation, , what changes is the value. Recalculating:

To validate our result, we calculate the average different between the average message in the original protocol, and in the protocol in this case:

We indeed see that on average the length of the successful transmission is less by :

* If error probability is 0, we always save time!
* As error probability increases, we save less time.

Which are intuitive conclusions.

## Question 2.

### 

First, calculating :

The optimal :

This is the optimal time. If after this time the ‘ack’ is not received, it is the sign that the message (or ack) didn’t reach the destination.

### 

The throughput is given by:

Where is the total time for 1 message for being processed

is a random variable showing the number of times the message is sent. Distributed geometrically with the probability of error . Thus, :

### 

The optimal window size would be the one, which can send maximal amount of messages before the for the first message has to arrive. This way, if there was no ack received, the first message will be resent. Calculating :

We use the ceil() function to round it up:

Thus, the window size is given by:

### 

Given:

The throughput is given again by:

As in tutorial, we define as number of times the message is resent. Then, for a given , the time until a message is transmitted:

The average time for 1 message to finish:

As in tutorial,

Thus,

Calculating the :

As we can see:

## Question 3.

### 

The scenario is similar to the one explained in the lecture, where it is promised that second package will reach the receiver with 100% certainty. To calculate the optimal window size, we take into account , which is not equal to .

Any increase in this window size will not benefit the throughput.

### 

The receiver optimal size is then:

### 

Calculating the :

is the probability of error on the first packet.

Summing up the probabilities of getting each transmission time for the packet to get the average transmission time for 1 packet.

### 

Finding the throughput through the equation:

## Question 4.

### 

In S&W, the change is done to , total time to transmit 1 message, and to probability of error .

The throughput calculation is then:

We can see that we cannot say for sure whether the throughput increases or decreases:

* Decreasing will increase
* Increasing will decrease

We can elaborate a condition which will ensure :

When the last condition is satisfied, the new throughput is bigger than the original.

### 

In GBN, if window size is less than , the throughput **decreases** for sure. There will be an ‘idle’ time, when the first package still hasn’t returned ack, but small window size doesn’t allow to send another packet, and the system will simply wait for ack from first package and in the meanwhile will not transmit

### 

The window increase beyond will not affect the throughput in this case. The packets beyond the size will be ready to be sent, but not be sent until the correct ACK was received for the first packet from sender (, which will happen after RTT. Thus, it will **not be different** from a case where the window size is . The situation is presented below:

A picture containing star, light

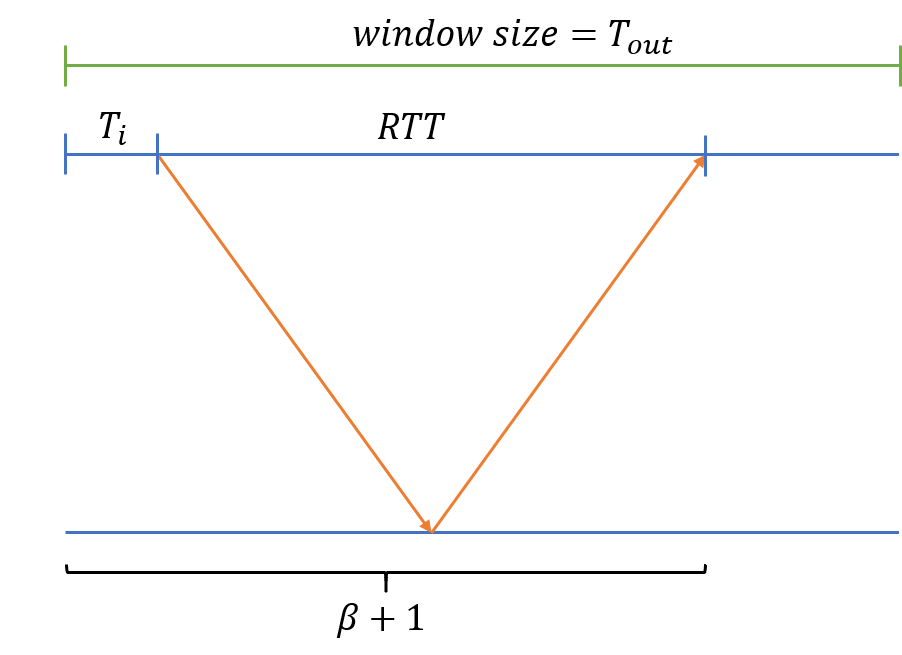
Description automatically generated

Summing up:

### 

In this case, the sender will keep on sending packets until the , and only then discover that there was no ACK for message . Which will make him resend the whole window of size , instead of window of size , or . This protocol **does NOT gain improvement** in performance (the sender will always transmit also in the original GBN), but loses when the packet is not received correctly ( ).

The situation is depicted below:



Summing up: , and **throughput decreases**

### 

Each packet is sent twice in GBN. From intuition, this can be beneficial is is large, and can be redundant when is small. Thus, it is impossible to know without more information.

Infinite window size on both ends means that the sender will send packets all the time. Thus, the throughput **will not change** if , since the packet will be resent anyway, if is a finite time. In SR, the throughput is not affected by other packets, since only the failed packet will be resent if case it failed to arrive (or the ACK was lost).

### 

We need a calculation here. Let be the number of resends we have to make. Then, the mean amount is:

The first part of the equation is here because the sender will send the failed packet until the indication for it was received, this means at least this amount of times.

Calculating the average time for 1 packet is then:

We multiply by since we already summed the Timeout to the E(k).

This throughput is of course less than the throughput of the original SR protocol . Intuitively, the protocol does a redundant job by sending over and over a package until the first ACK. While the good package may be on its way to receiver, or its ACK back to sender, the sender may have already sent same packages which will now be thrown. So the throughput will decrease

## APPENDIX – python code

## Question 1

import numpy as np

from matplotlib import pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

from matplotlib import cm

from matplotlib.ticker import LinearLocator, FormatStrFormatter

import numpy as np

def main():

beta\_list = np.linspace(start = 1, stop = 30, num = 30, dtype = int)

fail\_prob\_list = np.linspace(start = 0, stop = 1.0, num = 11, dtype=float)

calc\_s\_new = lambda B, p : (1-(p\*\*2)) / (2 + B)

calc\_s\_original = lambda B, p: (1-p) / (1 + B)

s\_new\_arr = np.empty((len(beta\_list),len(fail\_prob\_list)))

s\_original\_arr = np.empty((len(beta\_list),len(fail\_prob\_list)))

s\_difference = np.empty((len(beta\_list),len(fail\_prob\_list)))

s\_difference\_binary = np.empty((len(beta\_list),len(fail\_prob\_list)))

for B\_idx, B in enumerate(beta\_list):

for p\_idx, p in enumerate(fail\_prob\_list):

s\_new\_arr[B\_idx][p\_idx] = calc\_s\_new(B,p)

s\_original\_arr[B\_idx][p\_idx] = calc\_s\_original(B,p)

s\_difference[B\_idx][p\_idx] = s\_original\_arr[B\_idx][p\_idx] - s\_new\_arr[B\_idx][p\_idx]

if s\_difference[B\_idx][p\_idx] > 0 :

s\_difference\_binary[B\_idx][p\_idx] = 1

else:

s\_difference\_binary[B\_idx][p\_idx] = 0

# for the plot

s\_new\_rot = np.fliplr(np.rot90(s\_new\_arr, axes=(1,0)))

s\_original\_rot = np.fliplr(np.rot90(s\_original\_arr, axes=(1,0)))

s\_diff\_rot = np.fliplr(np.rot90(s\_difference, axes=(1,0)))

s\_binary\_rot = np.fliplr(np.rot90(s\_difference\_binary, axes=(1,0)))

beta\_grid, p\_grid = np.meshgrid(beta\_list, fail\_prob\_list)

fig = plt.figure()

ax = fig.gca(projection='3d')

# Plot the surface.

surf = ax.plot\_surface(beta\_grid, p\_grid, s\_diff\_rot, cmap=cm.coolwarm,

linewidth=0, antialiased=False)

plt.show()

fig, ax = plt.subplots()

ax.imshow(s\_binary\_rot)

ax.set\_xticks(np.arange(len(beta\_list)))

ax.set\_yticks(np.arange(len(fail\_prob\_list)))

ax.set\_xticklabels(beta\_list)

ax.set\_yticklabels(np.round(fail\_prob\_list, 3))

ax.set\_xlabel("Beta")

ax.set\_ylabel("P")

fig.tight\_layout()

plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

main()

## Question 2

import math

def print\_var(var\_name : str):

print(f"\t{var\_name} = {eval(var\_name)}")

dist = 2 \* 10\*\*6

R = 5 \* 10\*\*6

msg\_len = 512\*8

ack\_len = 16\*8

speed = 2\*(10\*\*8)

print("\nA:")

# SNW

T\_i = msg\_len/R

T\_ack = ack\_len/R

T\_p = dist / speed

print\_var("T\_i")

print\_var("T\_ack")

print\_var("T\_p")

RTT = T\_p\*2

T\_out = RTT + T\_ack

print\_var("T\_out")

print("\nB:")

# calculating throughput

T\_t = T\_i + T\_out

print\_var("T\_t")

M1\_avg = lambda p : 1/(1-p)

T\_i\_over\_T\_t = T\_i / T\_t

print\_var("T\_i\_over\_T\_t")

beta\_exact = RTT / T\_i

throughput\_SNW = lambda p : T\_i\_over\_T\_t / M1\_avg(p)

throughput\_SNW\_2 = lambda p : (1-p) / (1 + beta\_exact)

print("\nC:")

# GBN

beta\_round = math.ceil(beta\_exact)

print\_var("beta\_exact")

print\_var("beta\_round")

n = beta\_round + 1

T\_window = n \* T\_i

print\_var("T\_window")

print("\nD:")

p = 0.3

S\_SWN = throughput\_SNW(p)

S\_SWN\_2 = throughput\_SNW(p)

print\_var("S\_SWN")

print\_var("S\_SWN\_2")

E\_k = lambda p : p / (1-p)

T\_v = T\_i + E\_k(p) \* (T\_window)

print\_var("T\_v")

S\_GBN = T\_i / T\_v

throughput\_GBN\_2 = lambda p : 1 / (1 + (p/(1-p)\*(beta\_round + 1)))

S\_GBN\_2 = throughput\_GBN\_2(p)

print\_var("S\_GBN")

print\_var("S\_GBN\_2")