# **Submit to srp@es.aau.dk with subject [NetPerf24][MiniProject][Emil L. H Jens-Ulrik L-M]**



Contents

[How to run the project: 3](#_Toc184113952)

[Task 1 - Source coding 4](#_Toc184113953)

[Shannon and Huffman -Algorithm core functionalities 4](#_Toc184113954)

[The Entropy calculation for both algorithms 4](#_Toc184113955)

[Shannon core functionality 5](#_Toc184113956)

[Huffman core functionalities 5](#_Toc184113957)

[Results for Huffman and Shannon 6](#_Toc184113958)

[LZW 7](#_Toc184113959)

[Results of LZW 7](#_Toc184113960)

[Task 2 - Buffer bloat lab exercise 9](#_Toc184113961)

[Part 2: Web page download - Sketch the TCP CWND 9](#_Toc184113962)

[Part 3: “Streaming video” - Sketch the TCP CWND and Buffer Occupancy. 9](#_Toc184113963)

[Part 4: Measuring the real cwnd and buffer occupancy values. 11](#_Toc184113964)

[Part 5: Make the router buffer smaller. Reduce it from 100 packets to 20 packets. 12](#_Toc184113965)

[Part 6: Different queues 13](#_Toc184113966)

[Task 3 - Dumbbell and types of TCP (lossless) 14](#_Toc184113967)

[Task 4 - Dumbbell and types of TCP (loss) 16](#_Toc184113968)

[Task 5 (Bonus) - BBR simulation (loss/lossless) 18](#_Toc184113969)

## How to run the project:

The code in this project has been written in python, all source coding exercises are meant for execution on any device with the correct packages installed such as numpy, heapq and reg-ex. For the mininet part the code is expected to run on an ubuntu 16.04 mininet VM. The code has been split into folders, thus LZW, Shannon, Huffmann, Bufferbloat, cs144Bufferbloat and Dumbell code are in these folders. The code should be in an executable state, but important parts are included in this report.

**T1 part:**

The code expects the following working directory [C:\<path-to-dir>\MINI-PROJECT NETPERF 2024]

The code can be found in:

[Mini-Project NetPerf 2024\Shannon Encoder\Shannon.py]

[Mini-Project NetPerf 2024\Huffmand Encoder\Huffmand.py]

[GIT\Mini-Project NetPerf 2024\LZW Encoder\LZW.py]

**T2 part:**

The must be executed on the mininet VM

The code can be found in:

[Mini-Project NetPerf 2024\cs144\_bufferbloat\]

[Mini-Project NetPerf 2024\Dumbbell\MiniPerfEx.py] Simulations

[Mini-Project NetPerf 2024\Dumbbell\Plot dumbbell.py] Plotting

All the code can also be downloaded from the following git repository:

<https://github.com/IIHermandII/Mini-Project-NetPerf-2024.git>

# Task 1 - Source coding

# Shannon and Huffman -Algorithm core functionalities

## The Entropy calculation for both algorithms

Entropy is a measure for information and will be used to gauge the performance of both Shannon and Huffman algorithms. The entropy tells theminimum number of needed bits on average for transmitting a value.

The entropy is calculated using:

We have constructed a python function that takes the novel as an argument, formatted as an array of either characters or words. The function computes the entropy ‘Hx’. It also produces an Entropy list, being a list containing all unique characters/words and their probabilities. The code is illustrated below with comments.

def Entropy(*Novel*):

    UniqueCharacters = set(*Novel*) 🡨 first we find the all the unique characters in this Novel

    NumberOfUniqueCharacters = len(UniqueCharacters) 🡨 Taking the length we how many there are

    MobyDickNrOfCharracters = len(*Novel*) 🡨 we find out how many characters there are in the novel

    Hx = 0 🡨 Hx will hold the value for H(x)

    EntropyList = [] 🡨 will hold prob of character and probability of character

    for character in UniqueCharacters: 🡨 walks through all the unique characters

        NumberOfCertainCharInNovel = *Novel*.count(character) 🡨 the amount of in the Novel

        ProbOfCharacter = NumberOfCertainCharInNovel/MobyDickNrOfCharacter 🡨

        Product = ProbOfCharacter \* numpy.log(1/ProbOfCharacter) 🡨

        Hx += Product 🡨

        EntropyList.append([character,ProbOfCharacter])

    EntropyList = sorted(EntropyList,*key*=lambda *x*: *x*[1])

    return Hx, EntropyList

## Shannon core functionality

The idea behind Shannon is to take every unique character used in Moby Dick, with their probability sort them highest to lowest. After the list has been sorted it can then be split in two. The split should be performed such that the entropy is equally distributed in the two new lists. You denote every symbol in the right list with ‘1’ and left list ‘0’. Now repeat this process until all lists have reached length 1. The code for each symbol will then accumulate through this process and all symbols will be coded. This way common symbols will have short codes and uncommon ones longer codes. We have done this in python and the code that do so is explained below:

def SHANNON(*EntropyList*, *prefix*=""):

    if len(*EntropyList*) == 1: 🡨 when we encounter a branch that is only ‘1’ (character) big we return it

        symbol = *EntropyList*[0][0]

        return {symbol: *prefix*}

All below is the case when we have a branch that is not ‘1’ (character) but bigger

    split\_idx = HalfPoint(*EntropyList*) 🡨 we split the list so we get the half point closest to 50%

    left = *EntropyList*[:split\_idx + 1] 🡨 we denote the one side left

    right = *EntropyList*[split\_idx + 1:] 🡨 and the other one right

    codes = {}

    codes.update(SHANNON(left, *prefix* + "0"))  *🡨 we update code with a “1” and call this function for the left part*

    codes.update(SHANNON(right, *prefix* + "1"))  *🡨 same for the right*

    return codes

## Huffman core functionalities

The idea behind Huffman is kind of the opposite of Shannon. You start again with the sorted list, however this time you take the elements with lowest probabilities and denote them ‘1’ and ‘0’. Now take right and left and add them together. You sort the list and repeat until all elements have been coded and the list has reached length 1.

def EvaluateHuffmann(*EntropyList*, *node*, *val*=''):

    total = 0 🡨 This will hold the value for the Entropy

    newVal = *val* + str(*node*.huff) 🡨 Updates the three that gets smaller and smaller

    if(*node*.left):

        total += EvalueateHuffmand(*EntropyList*, *node*.left, newVal) 🡨 we adds 0 to the branches below

    if(*node*.right):

        total += EvalueateHuffmand(*EntropyList*, *node*.right, newVal) 🡨 we adds 1 to the branches below

    if(not *node*.left and not *node*.right): 🡨 when final list

*#print(f"{node.symbol} -> {newVal}")*

        for i in *EntropyList*:

            if i[0] == *node*.symbol: 🡨 calculate the value for all Characters

*val* = i[1] \* len(newVal)

                total += *val*

    return total 🡨 The E[L] value

## Results for Huffman and Shannon

The results of the two encoding schemes can be seen below:

Table 1 Shannon Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Shannon Encoder/Shannon.py"  Shannon  Character Chosen  Working On Entropy Calculations ...  Entropy: 3.1169  Working on SHANNON Algorithm ...  Shannon Codes: can be shown in [def EvalueateShannon]  Algorithm Entropy : 4.5382 Entropy: 3.1169  Total number of bits: 5619288.0000 |

Table 2 Shannon Words selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Shannon Encoder/Shannon.py"  Shannon  Words Chosen  Working On Entropy Calculations ...  Entropy: 4.8073  Working on SHANNON Algorithm ...  Shannon Codes: can be shown in [def EvalueateShannon]  Algorithm Entropy : 7.0652 Entropy: 4.8073  Total number of bits: 3171226.0000 |

Table 3 huffman Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Huffmand Encoder/Huffmand.py"  Huffman  Character Chosen  Working On Entropy Calculations ...  Entropy: 3.1169  Working on HUFFMAND Algorithm ...  Huffman Codes: can be shown in [def EvalueateHuffmand]  Algorithm Entropy : 4.5261 Entropy: 3.1169  Total number of bits: 5604276.0000 |

Table 4 huffman Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Huffmand Encoder/Huffmand.py"  Huffman  Words Chosen  Working On Entropy Calculations ...  Entropy: 4.8073  Working on HUFFMAND Algorithm ...  Huffman Codes: can be shown in [def EvalueateHuffmand]  Algorithm Entropy : 6.9999 Entropy: 4.8073  Total number of bits: 3141926.0000 |

We see what we expected that Huffman is a little better than Shannon in general. And that character is better than words when it comes to Entropy. The reason for this could be that in normal use cases like this there are a lot fewer unique characters than words. However, there in general the number of characters is much larger than words. We see that Huffman is better for words if one must pick then Huffmann would be the best choice for encoding Moby Dick.

## LZW

The idea behind LZW is to have a dictionary and replicate the source with only the dictionary. However as one encodes the dictionary is expanded, thus say you encounter ‘SA’ but you already know ‘S’ you would then encode ‘S’ and append ‘SA’, such that future occurrences of ‘SA’ would be encoded more efficiently. This process keeps going until eventually common words or phrases are encoded in the dictionary. Thus, the code would consist of a number representing a place in the dictionary.

The code is described below:

def LZW\_Homebrew(*Alphabet*, *Data*):

    list = *Alphabet*

    Novel = *Data*

    i = 0 🡨 calculate we stand at

    j = 0 🡨 holds old values of i

    code = [] 🡨 Where the code will be sored

    while 1: 🡨 while we not done

        j = i 🡨 we update j to match j

        while 1: 🡨 while we haven’t fount a match in the alphabet

            lzw = ''.join(Novel[j:i+1]) 🡨 the character string we trying to macth

            if lzw not in list: 🡨 is it not in alpabet ad it

                list.append(lzw)

                code.append(list.index(lzw[:-1])) 🡨 we append the old code ((names) append (name))

                j = i 🡨 updates the steps we have taken

                break

            i += 1 🡨 if it was or wasn’t in alphabet we move one to the next character

            if lzw == ''.join(Novel[j:]): 🡨 if this is the end alpabet append to code

                code.append(list.index(lzw))

                break

*# print("----------")*

*# print(list)*

*# print(i)*

*# print("----------")*

        print(round((i/(len(Novel)-1))\*100, 3), "\t %\r", *end*="") 🡨 progress bar

        if i >= len(Novel)-1: 🡨 are we done with the novel break

            break

    code.append(0) 🡨 end character

*#print(code)*

    f = open("Homebrew LZW.txt","w") 🡨 write to text file.

    f.write(str(code))

    f.close()

### Results of LZW

|  |
| --- |
| The string to compress “TOBEORNOTTOBEORTOBEORNOT “  Aalborg Universitet/GIT/Mini-Project NetPerf 2024/LZW Encoder/LZW.py"  working on OPtimized LZW: ...  working on HOMEBREW LZW: ...  Homebrew LZW.txt  gives [20, 15, 2, 5, 15, 18, 14, 15, 20, 27, 29, 31, 36, 30, 32, 34, 0]  Optimized LZW.tzt  gives [21, 16, 3, 6, 16, 19, 15, 16, 21, 28, 30, 32, 37, 31, 33, 35, 0]  This is because it adds the “” none Caracter, to the alphabet this will come in handy later.  This it to prove that the models work (at least for this string) |

We believe the algorithm works but that the novel is not big enough or that there are too many special cases where it doesn’t repeat enough caused by special characters.

The novel is 1,247 KB

The LZW encoded (Characters) is 1,609 KB

The LZW encoded (Words) is 1,277 KB

The LZW encoder works for the test string “TOBEORNOTTOBEORTOBEORNOT“. However, the result for Moby Dick is not found to be compressed as wanted.

# Task 2 - Buffer bloat lab exercise

Description:

Complete the Mininet bufferbloat exercise available at the link  
[https://github.com/mininet/mininet/wiki/Bufferbloat](https://github.com/mininet/mininet/wiki/Bufferbloat%20) and include the results in your report.

Lots of problems with VM, pip, matplotlib, tcpprobe, linux kernel … however we think it is working now.

## Part 2: Web page download - Sketch the TCP CWND

The first objective in the exercise is to measure how long it takes to download a web page from H1 and get a transfer time of 1 second.

The behavior of the CWND at H1 would look like since it is following TCP reno policy, see sketch.

Et billede, der indeholder tekst, diagram, linje/række, Font/skrifttype

Automatisk genereret beskrivelse

Reno features slow start (SS) which uses exponential growth to grow until SSTHRES (8 in plot) is reached then uses additive increase and multiplicative decrease until the transfer is complete. In this case the buffer can almost hold the entire webpage, meaning once the slow start is over, H1 never has to hold back sending. In this short traffic the TCP pretty much stays in the SS-phase.

## Part 3: “Streaming video” - Sketch the TCP CWND and Buffer Occupancy.

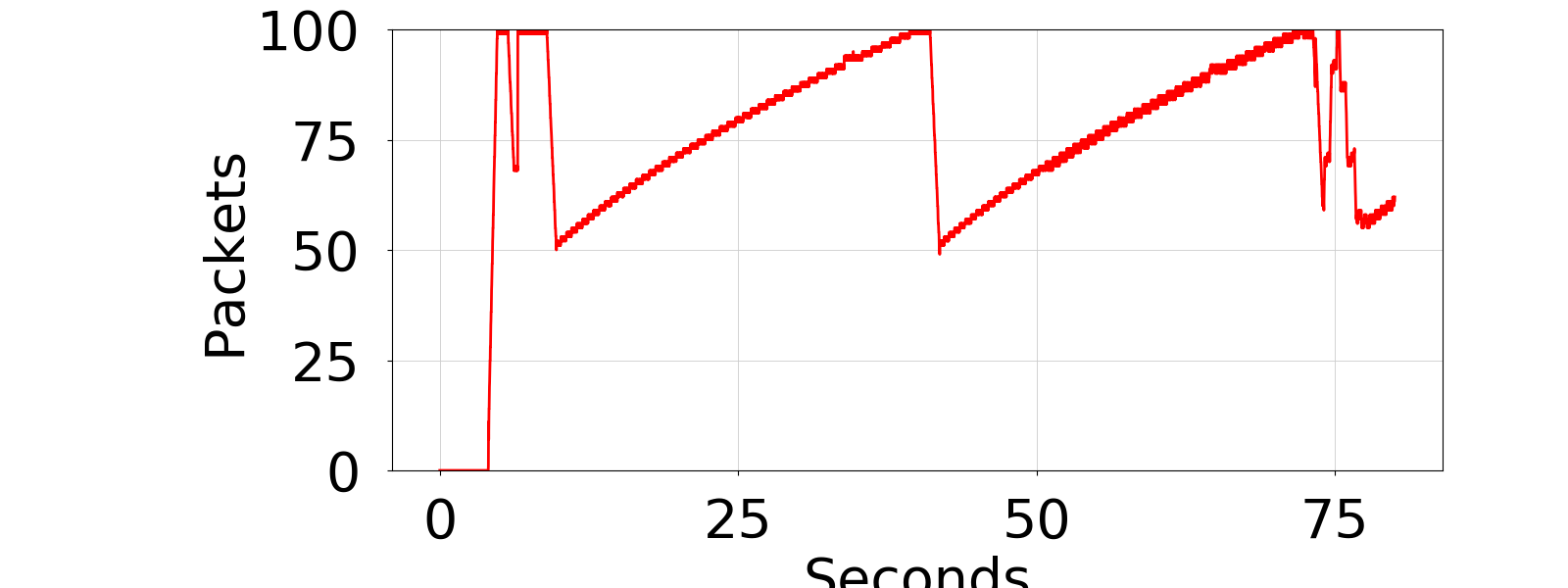
In this part we now set up some traffic between H1 and H2 using iperf. This added traffic will cause congestion on the line and thus the same transfer cannot be achieved anymore. Using the ping command, we can see that the ping has increased and that it goes between 400ms and 800ms in a cycle. This results from the TCP policy and materializes as the TCP sawtooth pattern, see sketch.

Et billede, der indeholder tekst, diagram, linje/række, Font/skrifttype

Automatisk genereret beskrivelseWhen rerunning the webpage download it now takes 7.2 seconds instead of the original 1 second. This is because the buffer in the router of the network gets congested by all the additional traffic caused by the iperf command causing H1 to halt/slow down the transmission of data to H2 for the packet to arrive. Thus, the TCP will enter congestion avoidance and go into additive increase mode. In this mode the CWND will gros slowly and if dup ACK’s is received the CWND will be set to CWND/2 and from where the fast recovery will set in. This dance will continue until the transfer is complete.

## Part 4: Measuring the real cwnd and buffer occupancy values.

In this part we do actual measurements of the cwnd and buffer during the iperf traffic and the webpage download. We use restart mininet, setup the iperf traffic and wait 70 seconds (says so in exercise) and do the webpage download. While this happens a monitor script measures the cwnd and buffer, the result is shown in the three figures below. We see initially just the iperf running, then the iperf and wget running and finally the queue throughout the experiment.

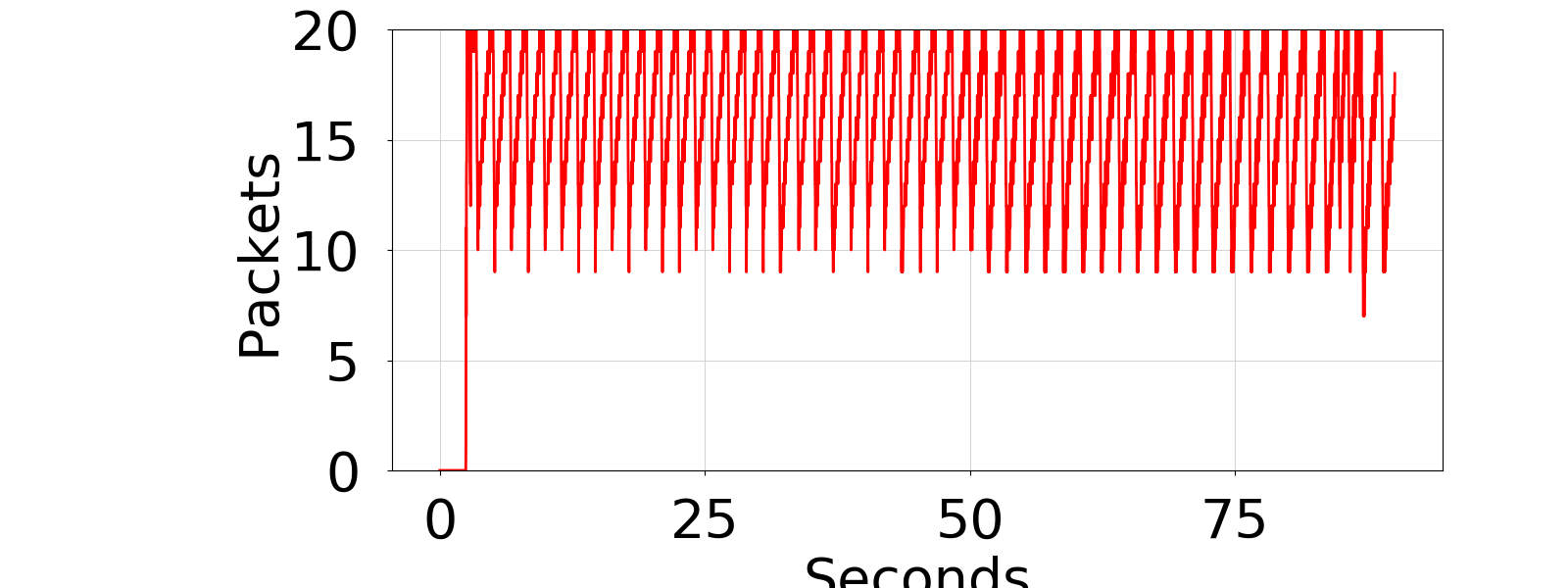
Here we see that there is an initial spike in packets, due to slow start quickly ramping to a high transfer rate. After that we see congestion avoidance kick in and the queue growth is controlled. When the sudden spike happens, it’s caused by the wget traffic trying to get through the buffer.Et billede, der indeholder linje/række, Kurve, diagram, skibakke

Automatisk genereret beskrivelse

This causes the congestion avoidance dance of TCP seen earlier, the wget is not as noticeable here however a small increase in slope of the cwnd can be seen at the end.

## Part 5: Make the router buffer smaller. Reduce it from 100 packets to 20 packets.

We repeat the experiment with downloading the webpage, this time with a smaller buffer and both with and without iperf running. This time running the wget produces two times, 1.2 seconds without iperf and 2.8 seconds with iperf. This is a slight decrease without and a large increase with iperf running. The reason being there are less ”junk” packets blocking the way of the webpage packets, leading to an overall reduction in transfer time when iperf is running. The experiment was logged, and the graphs can be seen below:



Here we again see that the buffer quickly fills up and then starts the TCP dance due to congestion avoidacne. At the end where the queue gets a bit more hectiv is when the wget is called and the webpage downloaded.

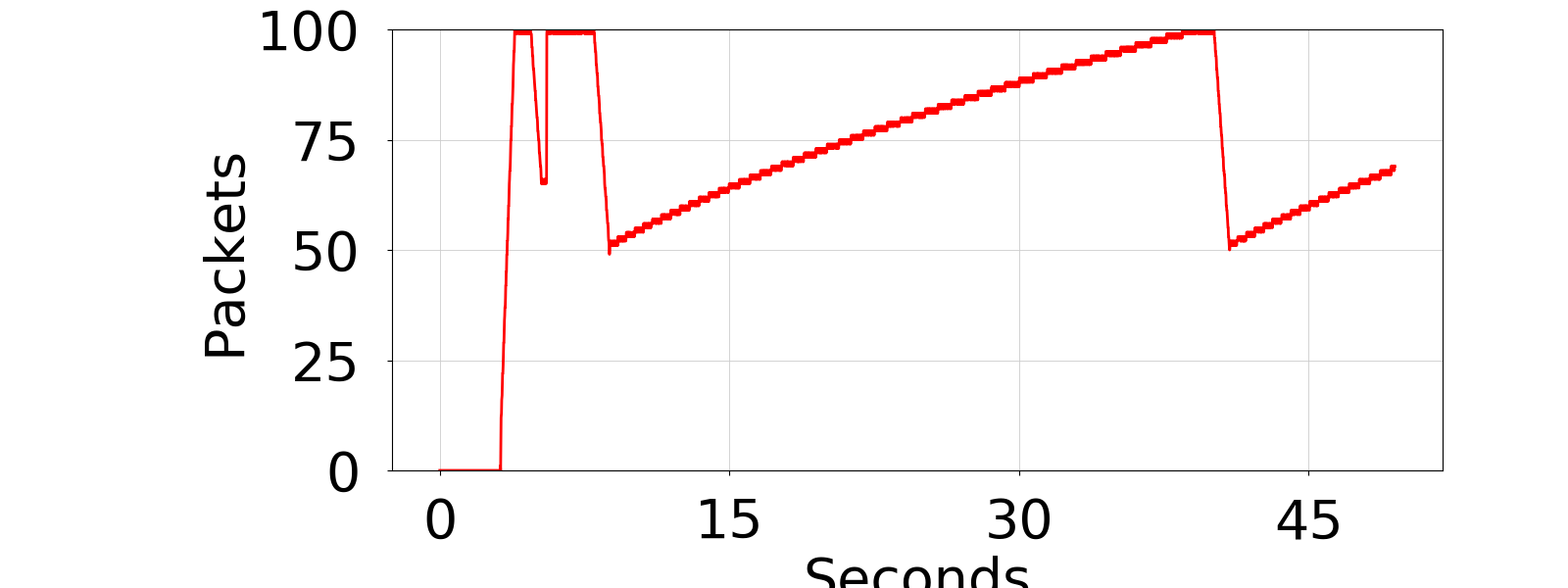
Et billede, der indeholder tekst, linje/række, Kurve, Font/skrifttype

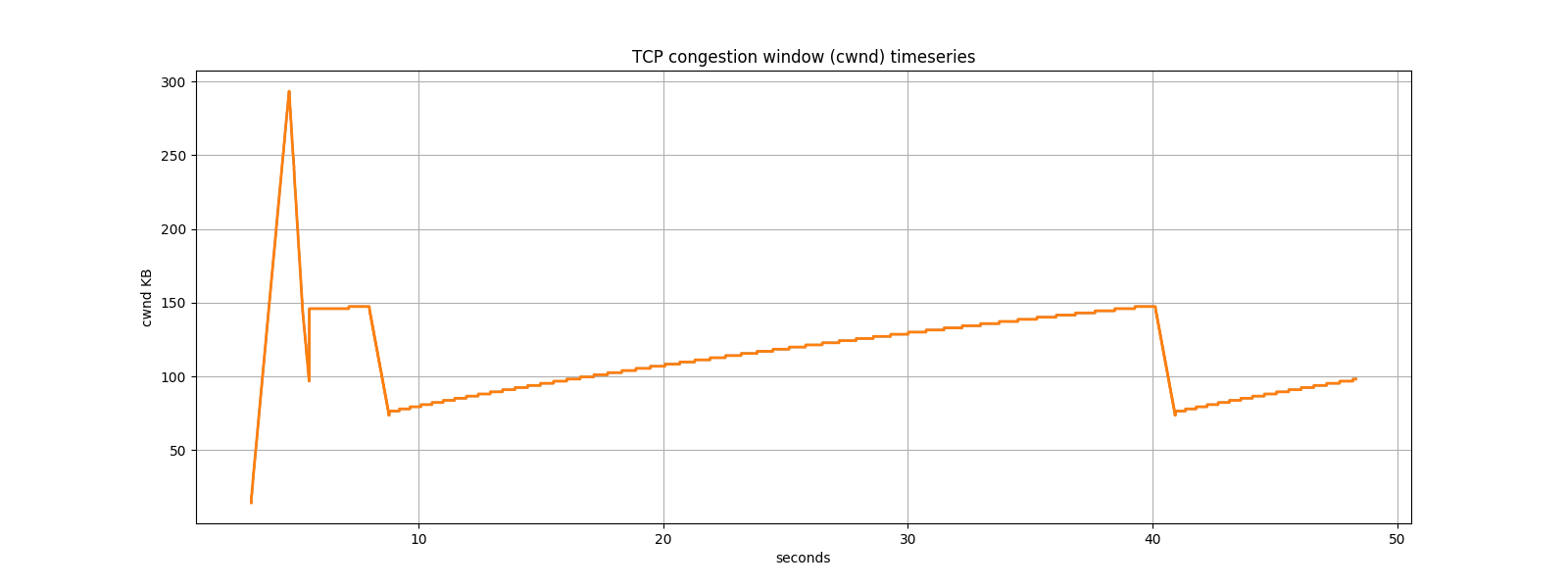
Automatisk genereret beskrivelse

Here we again see the slow start followed by congestion avoidance. We here see a smaller swnd since the buffer is smaller and here it’s really difficult to see the effect of wget since the duration is much shorter due to the decreased queue length.

## Part 6: Different queues

In this final part we split the iperf and wget traffic into two separate queues. The split will allow the slow/long lasting traffic from blocking the quick/short flow of downloading the webpage. Thus, the ping and download speed will be much less affected by the iperf. Due to this we get a wget time of 1.1 seconds without iperf and a wget time of 1.2 seconds with iperf running. This experiment was also logged:



In this experiment the queue is large again, however due to the split this is only the iperf queue.

And this performs similarly to earlier however with the benefit that the quick/short traffic gets a way better opportunity to pass unhindered.

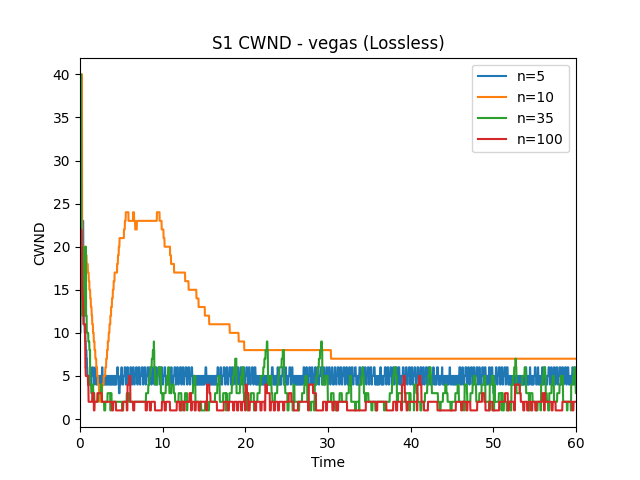
# Task 3 - Dumbbell and types of TCP (lossless)

In this task we are to construct the shown dumbbell network structure and simulate the CWND with different configurations, loss and TCP congestion control algorithms. For this purpose, Mininet will again be used, we repurpose the code in the bufferbloat exercise to now construct the dumbbell structure. With this we can use the cwnd monitor from the bufferbloat exercise to measure the performance for different source/destination counts and with/without loss. In this task we test TCP reno/vegas/cubic for the counts: 5, 10, 35, 100. The results can be seen below:

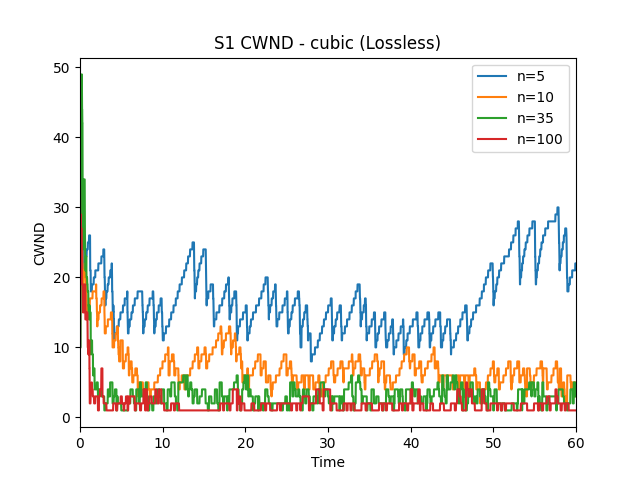
Et billede, der indeholder tekst, skærmbillede, Kurve, diagram

Automatisk genereret beskrivelse

In the case of TCP Reno, we clearly see the initial slow start scale exponentially followed by the cycle of AI and MD leading stairway structure. We see that the CWND is lowered for more users of the network since they must share the bottleneck transfer rate between the routers.



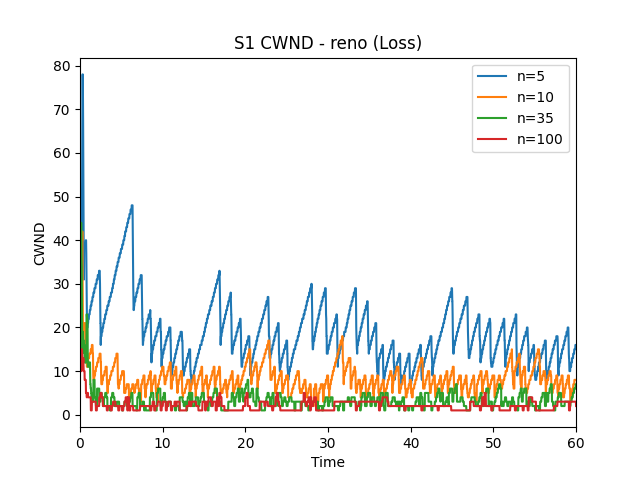
Here we see TCP Vegas which does not change between AI and MD but uses the measured RTT to adjust the CWND. In this way TCP Vegas tries to keep the CWND constant. This can be seen by the lack of stair structure like in Reno.



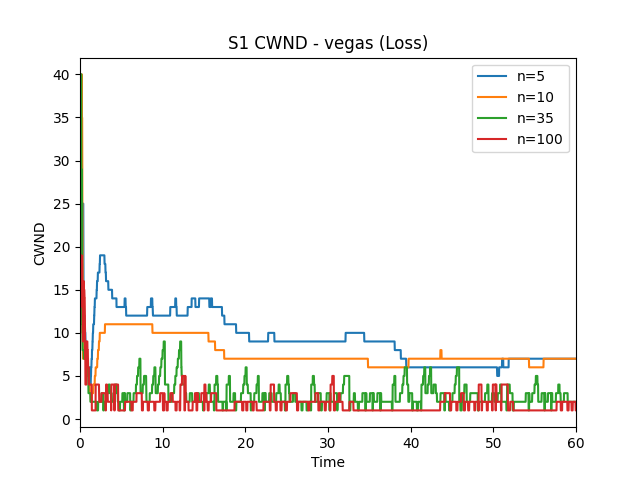
Finally, we have TCP Cubic, which tries to be more aggressive compared to Reno. The cubic growth is faster than the AI when increasing the CWND. In Cubic the growth slows as the previous CWND limit is reached, thus spending more time near maximum throughput. Cubic still uses MD, and thus the result becomes a Reno like structure, but Cubic the CWND is generally larger equaling more throughput.

# Task 4 - Dumbbell and types of TCP (loss)

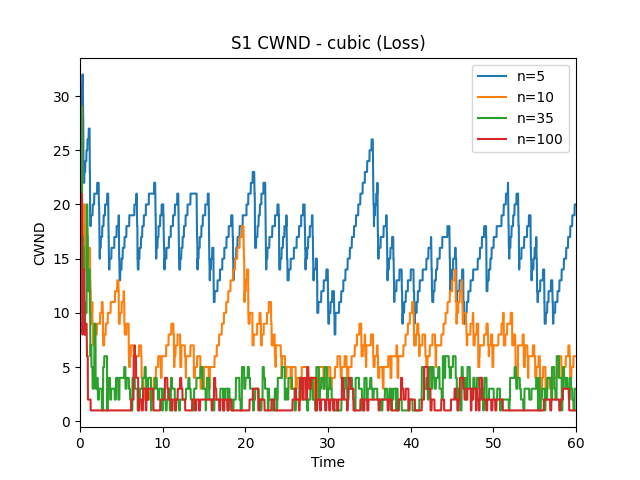
In this task we are to repeat the simulations now introducing a 1% packet loss rate, the new results can be seen below:



In the case of loss, it can be seen the CWND is lowered a bit since more MD occurs lowering the CWND more times during the 60 sec simulation.



Here it can be seen that the packet losses cause the RTT to suffer leading the TCP Vegas to lower the CWND.



Here it can be seen that the effect of the packet loss is less since the aggressive nature of Cubic recovers more quickly than Reno and Vegas.

# Task 5 (Bonus) - BBR simulation (loss/lossless)

In this task we try to use the more modern TCP congestion control algorithm BBR (by Google). This however produces an error:

Setting TCP congestion control to bbr

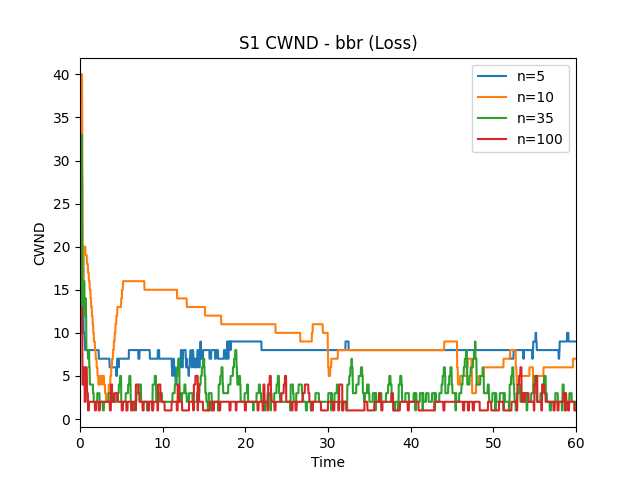
sysctl: setting key "net.ipv4.tcp\_congestion\_control": No such file or directory

net.ipv4.tcp\_congestion\_control = bbr

However, the code still executes and produces a result:

Et billede, der indeholder tekst, skærmbillede, diagram, linje/række

Automatisk genereret beskrivelse



The performance looks quite like Vegas. BBR tries to keep the latency low, thus in the low source/destination count cases the CWND is smaller than compared with TCP Reno and Cubic which try to maximize throughput and not latency.