Politecnico di Torino

Internet Performance and Troubleshooting Lab

Evaluating TCP Congestion Control Algorithms

Group 9

Report 4

Giuseppe Bruno s319892
Michela Salvadori s302544
Salvatore Giarracca s314701

January 21, 2024

1 Introduction

1.1 Environment setup

We set up two pc having linux **Ubunutu 23.10** as operating system: the client with **10.0.9.2** and the server with **10.0.9.3** as ip address. Then we started to collect data with **iperf3** tool, aiming to spot the principal behaviours of some TCP congestion control algorithms. The tests have been done only once, sequentially (see bash script), over different condition in terms of **probability loss** and **delay** of a TCP data exchange. We set up the link between the two hosts at speed of **10 Mbits/s**, leaving unchanged the default offloading capabilities of the client. Furthermore, we notice that some of the tested congestion control algorithms appear to be unsupported by the server, so we loaded all the necessary kernel modules on it.

1.2 TCP Congestion Control

The general policy for handling congestion in TCP are based on three main phases: **slow start**, **congestion avoidance** and **congestion detection**. In the slow start phase (**SS**), the sender starts to send packets at slow rate, increasing the congestion window (**CWND**) in an exponential way, in order to reach the slow start threshold as fast as possible. In the congestion avoidance phase (**CA**), the rate is reduced to avoid congestion and this appens if ssthresh is reached or if congestion is detected. The sender now increase the CWND slower (**Additive Increase**). When congestion is detected, the sender reduces the CWND (**Multiplicative Decrease**) and go back to SS or CA based on the comparison of CWND with SSTHRESH. The schema follows this general rule:

$$\text{CWND}(t+1) = \begin{cases} \text{CWND}(t) + a & \text{if congestion is not detected,} \\ \text{CWND}(t) \cdot b & \text{if congestion is detected,} \end{cases}$$

where $a \geq 0$ and $b \in (0, 1)$.

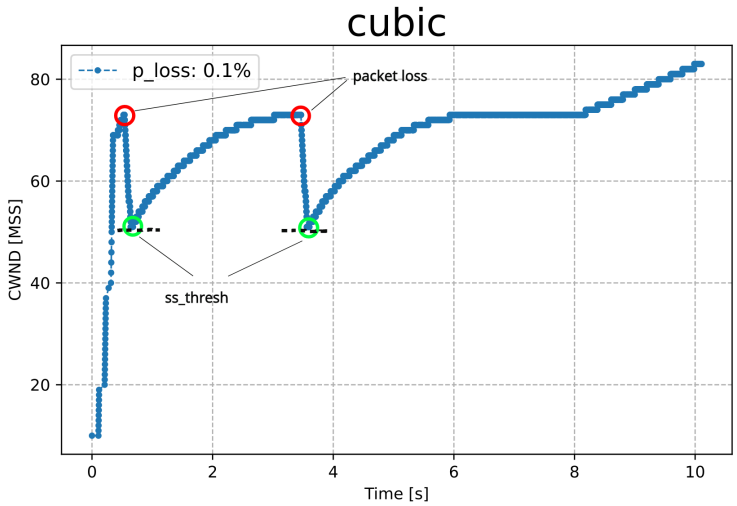


Figure 1: Packet loss example

As an general example, in Figure 1 it is shown the CWND trend of TCP Cubic CCA. In particular we can see that the SS phase ends after a packet loss, then the CWND shrinks itself to the 70% of original value, reaching the ssthresh.

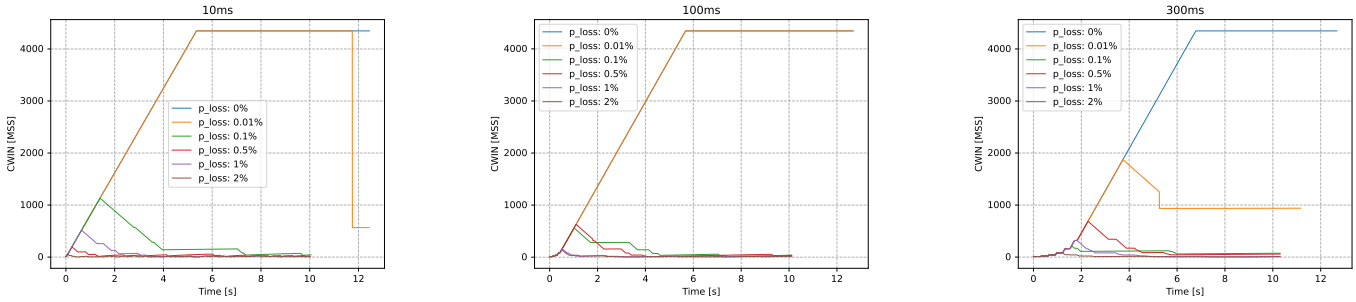
2 Evaluating Congestion Control Algorithms

2.1 Congestion Detection

Control congestion algorithms could use different transmission feedback, such as a loss of packet (**loss-based**) and the increase of the RTT(**delay-based**). In the first case the CWND decreases if the sender notices that a packet has been probably lost due to the receipt of **three duplicate ACKs** or due to expiration of the timeout (**RTO**). In the second case, the CWND decrease when the **RTT** increase. There are two (optional) techniques that helps the recovery of lost packet in TCP: **fast retransmission** and **fast recovery**. When a stand-alone duplicate ACK arrives, the sender interprets it as a possible lost packet and immediately retransmit it without waiting for the timer to expire. This behaviour improve recovery from packet loss and speed up the transmission.

2.2 TCP Reno

Reno is a **loss-based** CCA. At the beginning the **CWND** is set to 1 (MSS) and increase of 1 for each acknowledged segment, effectively doubling it at every RTT (**slow start**). If the CWND reaches the **ssthresh**, the **congestion avoidance** proceeds. In case of the **RTO** expires (congestion) the **new** ssthresh is equal to CWND / 2 and CWND is initialized again to 1, otherwise if 3 dublicated ACK are received the **new** ssthresh is equal to CWND / 2 and the CWND to ssthresh + 3 * **MSS**.



(a) (b) (c)

Figure 2: TCP Reno

In the plots above, it is shown the trend of CWND over time, for different network conditions. Due to its nature (loss-driven), TCP Reno behaves different under light delayed network, in respect of heavily delayed one. In particular, wen the delay is 10ms (Figure 2a), the **SS** phase ends immediately after connection's start, thus the **CA** phase is predominant.

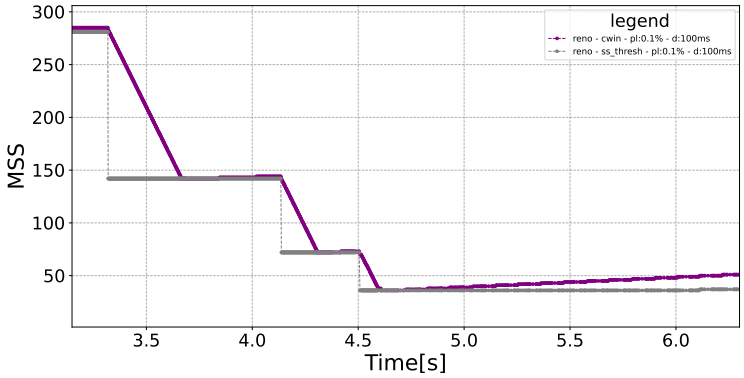


Figure 3: TCP Reno CWND and ssthreshold

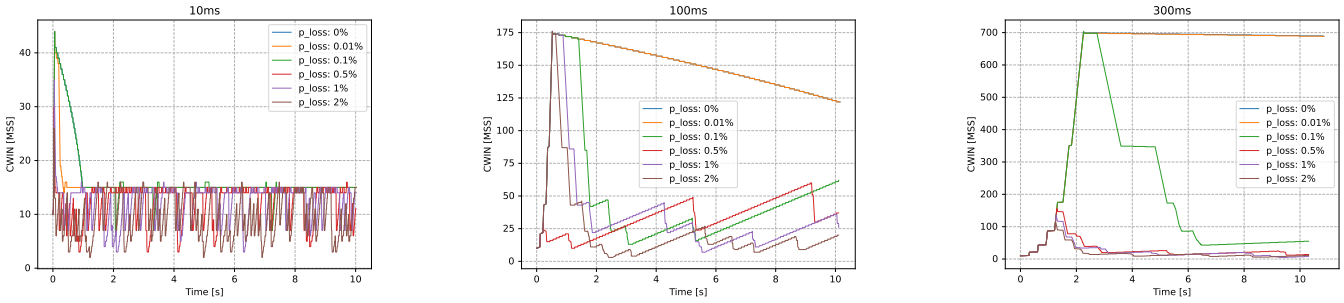
In Figure 3 are represented the congestion window with its corresponding **ssthreshold**. From ours experiments, the size of the **new** window is not instant equal to the new one, but tends towards that value following a slope.
From our hypotheses, in case of losses, the congestion window tends to its half plus 3 MSS, but by observing the graphs, it is noticeable that this 3 MSS gap is never present (it is always equal to 0).

2.3 Vegas

TCP Vegas is a **delay-based** CCA. The algorithm compute the **actual throughput** and the **expected throughput**. The first one is equal to $CWND/RTT$ and the second one to $CWND/RTT_0$ where RTT_0 is the minimum RTT obtained. The slow start phase is similar to TCP Reno, with the difference that it stops this phase also when $ExpectedThroughput - ActualThroughput < \delta$ (threshold). After that, the congestion avoidance phase begins. Here, two thresholds are used to update the CWND, α and β, as described in the relation below:

$$\text{CWND}(t+1) = \begin{cases} \text{CWND}(t) + \frac{1}{RTT} & \text{if diff} < \alpha \ , \\ \text{CWND}(t) - \frac{1}{RTT} & \text{if diff} > \beta \ , \\ \text{CWND}(t), \end{cases}$$

with $\alpha = 1$ and $\beta = 3$, usually. These values may vary, depending on kernel implementation. As described in the formula above, the algorithm rely solely to the accurate computation of RTT. If it is too small then throughput of the connection will be less than the bandwidth available while if the value is too large then it will overrun the connection. However it tends to give up bandwidth when it coexists with other algorithms like Reno.



(a) (b) (c)

Figure 4: TCP Vegas

In the plots above, it is shown the trend of CWND over time, for different network conditions. TCP Vegas CWND does not start from 1 MSS but always = 10. It grows up to a certain value based on the network delay ($MAXCWND \approx 40$ (10ms), ≈ 175 (100ms), ≈ 700 (300ms)) and, when the max value is reached, it decreases less rapidly for increasing delays. One of the main problem of TCP Vegas is the packet losses over a link: due to the linear increase of the CWND, it could be very slow to saturate the available bandwidth, resulting in very low throughput under network with high delay and high packet loss probability. This could be clearly seen comparing Figure 4a and Figure 4b.

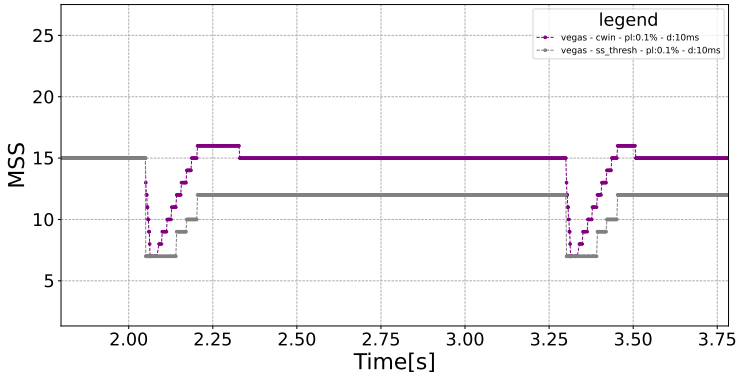


Figure 5: TCP Vegas CWND and ssthreshold

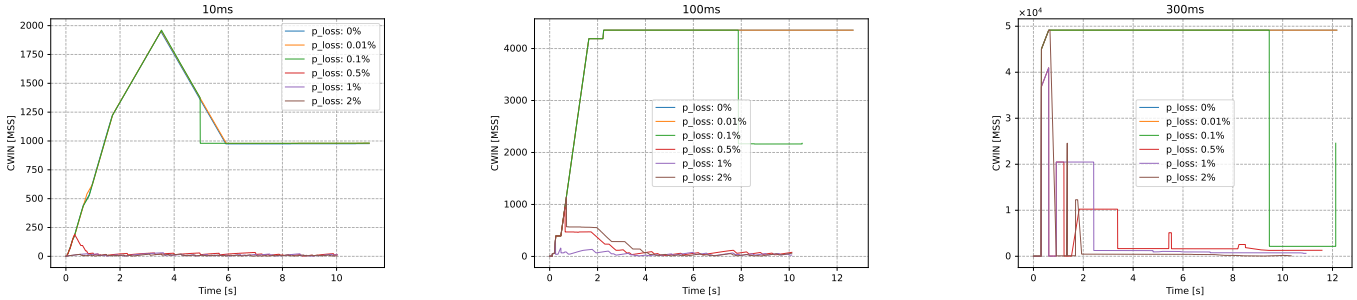
From the Figure5, it is possible to observe that the threshold varies only in case of congestion. The variation in samples from 15 to 16 and vice versa is due to the Vegas congestion avoidance algorithm itself.

2.4 Hybla

Hybla is a new **loss-based** algorithm and has been invented to have high performances on long wireless connections, like satellite's communications. In order to make the throughput of the connection independent of RTT the **congestion avoidance** is a function of RTT itself. At the beginning is defined $\rho = \frac{RTT}{RTT_0}$ where RTT_0 is pre-determined (e.g., 25 ms). The slow start and congestion avoidance algotihms are equal to:

$$\text{CWND}(t+1) = \begin{cases} \rho & t = 0\ , \\ \text{CWND}(t) + 2^{\rho} - 1 & \text{SS}\ , \\ \text{CWND}(t) + \frac{\rho^2}{CWND(t)} & \text{CA} \end{cases}$$

If a **timeout** is detected $ssthreshold = \rho \cdot CWND_{current}/2$ and the $CWND_{new} = \rho$, otherwise (in case of duplicated ACKs) $ssthreshold = CWND/2$ and $CWND_{new} = CWND_{current}/2$.



(a) (b) (c)

Figure 6: TCP Hybla

The most interesting thing of TCP Hybla is to have a throughput unrelated to the delay of the connection, as described previously. As a consequence, we noticed that it exit always at the same time from the slow start phase, in different condition of delay. Compared to TCP Reno, it could reach higher throughput in high delayed connections, as we can see from Figure 6c, in which the CWND is very high right from the beginning of transmission. However, it is not very resilient to packet loss, even if it manages to reach higher speed compared to TCP Vegas.

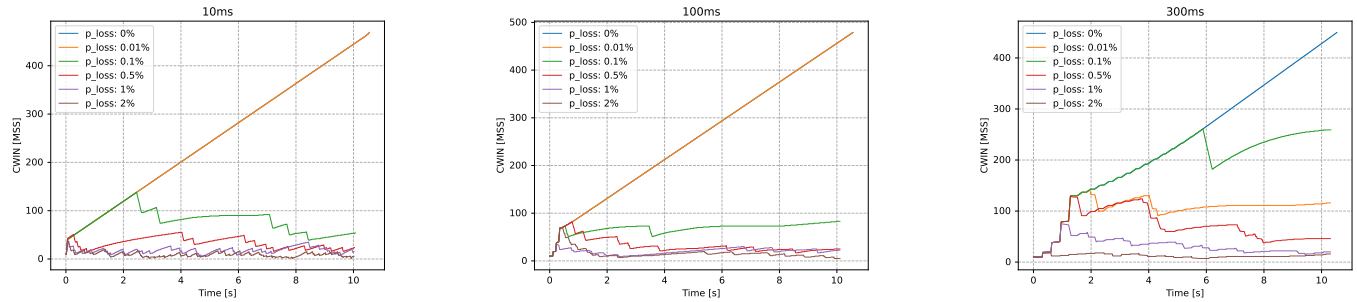
2.5 Cubic

Cubic is a **loss-based** algorithm. The **slow start** phase is similar to Reno, in same cases could be implemented the **HyStart** algorithm. Its purpose is to verify that the congestion avoidance phase has not started prematurely, allowing the receiver to increase the congestion window exponentially (with a smaller coefficient).
The **AIMD** is based on a **cubic** function.

$$CWND = C \cdot (T - K)^3 + W_{\max} \tag{1}$$

$$\text{where} \quad K = \sqrt[3]{\frac{W_{\max} \cdot (1 - B)}{C}}$$

In the above formula **C** is usually equal to 0.4, **B** is the **multiplication decrease factor** equal to 0.7, $W_{\mathbf{max}}$ is the window size just before the last reduction and **T** is the time elapsed since the last window reduction. In case of congestion the new **ssthresh** is CWND / 2 and the **new CWND** is equal to ssthresh.



(a) (b) (c)

Figure 7: TCP Cubic

TCP Cubic is one of the most adopted CCA today and it is the default option in Linux kernel. It is throughput is mostly independent from the network delay, thus it rely only the last congestion event. Thus TCP Cubic probes gently the network with the convexity of the curve that it uses, then it stands most of the time in the plateau between the convex and concave growth region, allowing to stabilize the network before it begins looking for more bandwidth. TCP Cubic manages to be less aggressive, compared to TCP Reno, as we can see from the plots above, over a congested channel.

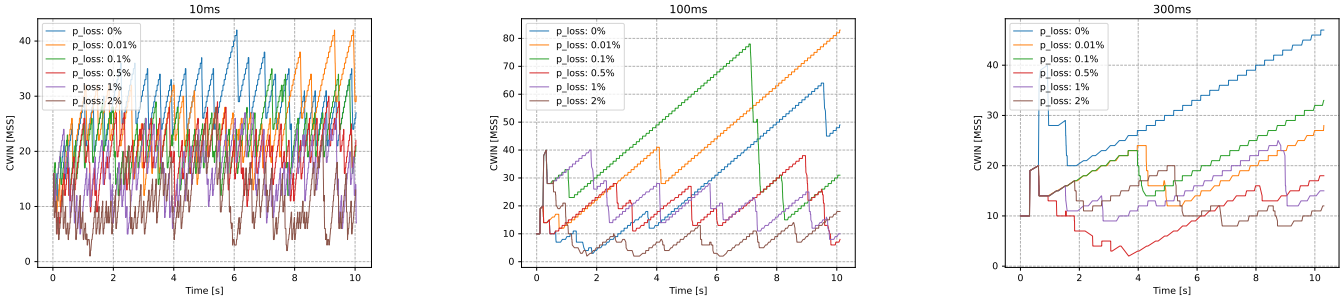
2.6 CAIA Delay Gradients (cdg)

CAIA is a **delay-based** algorithm. The CDG approach is to look at the minimum and maximum RTTs observed for a given connection over a period of time. The minimum is called RTT_{min}, while the maximum is RTT_{max}. From subsequent observations of these two values, the algorithm calculates the rate of change of each (δ_{min} and δ_{max}). The algorithm uses also random variables in the **AIMD** system, as following:

$$\text{CWND}(n+1) = \begin{cases} \text{CWND}(n) \cdot \beta & X < P[backoff] \wedge \delta_n > 0 \\ \text{CWND}(n) + 1 & \text{Otherwise} \end{cases},$$

$$\text{where} \quad P[backoff] = 1 - e^{\frac{\delta_n}{G}}$$

In this system, β is the multiplicative decrease factor equal to $\approx$ **0.7**, $X \in [0, 1]$ and is a uniformly distributed random number, **n** is the nth RTT, δ_n will either be δ_{min} or δ_{max} (depending what the background algorithm decides). **G** is a scaling parameter major than 0, implemented on kernel.



(a) (b) (c)

Figure 8: TCP CDG

TCP CDG, due to its nature of being a delay driven algorithm, aims to avoid depending on packet loss as a signal of congestion. It also aims to allow a machine to make full use of available bandwidth while sharing it with other systems without a central control mechanism. Additionally, CDG is designed to behave friendly with conventional TCP Reno flows. From the plots above we can observe that, as TCP Vegas, the congestion window start point is always 10 MSS.
This algorithm has the highest slop than the others but the CWND remains on relative small values.

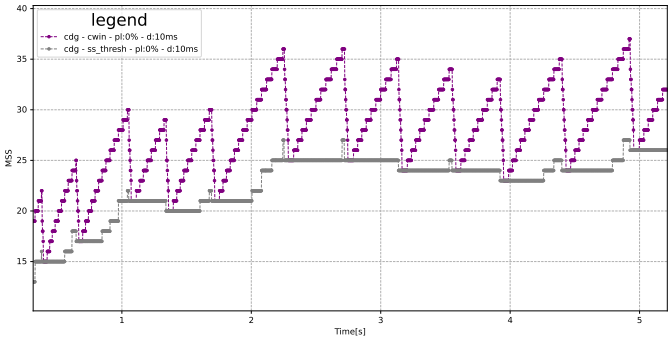


Figure 9: TCP CDG loss 0% CWND and ssthreshold

The Figure9 precisely illustrates how the congestion window varies in a delay-based algorithm, even though there is no chance of packet loss. The reduction is random and seems to have (on small RTTs) constant behavior.

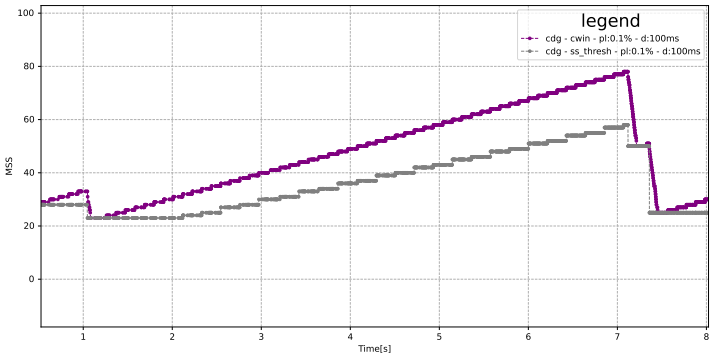


Figure 10: TCP CDG CWND and ssthreshold

One of the features we observed in CDG (on high RTTs) is the variation of the threshold. In case of congestion, for the initial moments it has a constant trend and after it follows the same trend as CWND but with a slope of approximately 50%

2.7 Conclusions

We have observed that, when there is a low probability of packet loss, the CWND of a loss driven algorithm tends to grow to very high values. In particular, in the slow start phase, they should have an exponential growth when try to reach the SS threshold, but we observed that the slope of the curve is linear. We believe that this happens due to the limited capacity of the link in which the connection take place (10 MBits/s). It also important to note that there is no a "better" Congestion Control Algorithm, but each one of them can be specifically better under specific network conditions.
In addition, from what regards the delay driven algorithm, for example Vegas (Figure 4b), with low loss probability, the CWIN and also the ssthreshold grows to very high values at the beginning of the connection and then it starts to decrease slowly, also with no congestion signal detected. This could be also due to the limitation of the link.

Appendix

Data collection script

```
#!/bin/bash

SRC_PORT=11111
SERVER="10.0.9.3"
IFACE=$(ip addr | awk '/state UP/ {print $2}' | sed 's/.$//')
ethtool -s ${IFACE} speed 10 duplex full autoneg on
modprobe tcp_diag
cd /sys/kernel/debug/tracing
echo 1 > events/tcp/tcp_probe/enable
cd -
sysctl net.ipv4.tcp_no_metrics_save=0
sysctl net.ipv4.tcp_sack=1
tc qdisc add dev ${IFACE} root netem

delays=("10ms" "50ms" "100ms" "300ms" "500ms")
algos=("reno" "cubic" "vegas" "hybla" "cdg")
losses=("0%" "0.01%" "0.1%" "0.5%" "1%" "2%")

ping -c 3 $SERVER

for delay in ${delays[@]}; do
  mkdir ${delay}; cd ${delay}
  for algo in ${algos[@]}; do
    modprobe tcp_$algo
    sysctl net.ipv4.tcp_allowed_congestion_control="${algo}"
    mkdir ${algo} ; cd ${algo}
    for loss in ${losses[@]}; do
      mkdir ${loss} ; cd ${loss}

        tc qdisc change dev ${IFACE} root netem loss ${loss} delay ${delay}
        echo tc qdisc change dev ${IFACE} root netem loss ${loss} delay ${delay}
        echo > /sys/kernel/debug/tracing/trace
        TRACE_FILE=trace
        iperf3 -c ${SERVER} -C ${algo} --cport ${SRC_PORT}
        cat /sys/kernel/debug/tracing/trace > $TRACE_FILE

        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | tr -s ' ' | cut -d ' ' -f 5 | cut -d ':' -f 1 > time
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 6 | cut -d ' ' -f 1 > len
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 7 | cut -d ' ' -f 1 > seqno
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 8 | cut -d ' ' -f 1 > una
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 9 | cut -d ' ' -f 1 > cwnd
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 10 | cut -d ' ' -f 1 > ssthresh
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 11 | cut -d ' ' -f 1 > snd_wnd
        cat $TRACE_FILE | grep cwnd | grep $SRC_PORT | cut -d '=' -f 12 | cut -d ' ' -f 1 > rtt

        paste time len seqno una cwnd ssthresh snd_wnd rtt > dirt_data
      cat dirt_data | grep -v "s" | grep -v -E "^.*\[.*\].*$" > data
      rm -f dirt_data
        cd ..
    done
    cd ..
  done
  chown -R cybermonkey: .
  cd ..
done
```

Data visualization script

```
import os
import pandas as pd
import matplotlib.pyplot as plt

algos = ["cdg", "cubic", "hybla", "reno", "vegas"]
losses = ["0%", "0.01%", "0.1%", "0.5%", "1%", "2%"]
variables = ["len", "seqno", "una", "cwnd", "ss_thresh", "snd_wnd", "rtt"]
delays = ["10ms", "50ms", "100ms", "300ms", "500ms"]

for delay in delays:
    os.chdir(delay)
    for algo in algos:
        os.chdir(algo)
        for i, variable in enumerate(variables):
            plt.figure()
            for loss in losses:
                os.chdir(loss)

                df = pd.read_csv("data", sep="\t", header=None)

                times_abs = df[0].tolist()
                times = [x - times_abs[0] for x in times_abs]

                if variable == "seqno" or variable == "una":
                    data_abs = (
                        df[i + 1]
                        .apply(lambda x: int(x, 16) if isinstance(x, str) else x)
                        .tolist()
                    )
                    data = []

                    # sequence reset handled partially :(
                    def get_relative(data_abs):
                        last_no = data_abs[0]
                        for no in data_abs:
                            if no < last_no:
                                last_no = no
                            data.append(no - last_no)

                    get_relative(data_abs)

                elif variable == "rtt":
                    data = df[i + 1].apply(
                        lambda x: x * 10 ** (-3)
                    )  # from microseconds to milliseconds
                else:
                    data = df[i + 1].tolist()

                plt.figure(1000)
                plt.plot(
                    times,
                    data,
                    label=f"p_loss: {loss}",
                    linewidth=1,
                    marker=".",
                    linestyle="--",
                )
                plt.title(variable.upper())
                plt.xlabel("Time [s]")
                unit = (
                    "[ms]"
                    if variable == "rtt"
                    else "[bytes]"
                    if variable == "len"
                    else "[MSS]"
                    if variable == "ss_thresh"
                    or variable == "cwin"
                    or variable == "snd_wnd"
                    else ""
                )
                plt.ylabel(variable.upper() + " " + unit)
```

```
                plt.ticklabel_format(
                    style="sci", axis="y", scilimits=(0, 4), useMathText=True
                )
                plt.grid(which="both", linestyle="--")
                plt.savefig(f"{algo}_{variable}.pdf")
                plt.close()

                plt.plot(
                    times,
                    data,
                    label=f"p_loss: {loss}",
                    linewidth=1,
                    marker=".",
                    linestyle="--",
                )

                os.chdir("..")

            plt.title(f"{delay}")
            plt.xlabel("Time [s]")
            unit = (
                "[ms]"
                if variable == "rtt"
                else "[bytes]"
                if variable == "len"
                else "[MSS]"
                if variable == "ss_thresh"
                or variable == "cwnd"
                or variable == "snd_wnd"
                else ""
            )
            plt.ylabel(variable.upper() + " " + unit)
            plt.ticklabel_format(
                style="sci", axis="y", scilimits=(0, 4), useMathText=True
            )
            plt.grid(which="both", linestyle="--")
            plt.legend()
            os.makedirs("plots_pdf", exist_ok=True)
            os.chdir("plots_pdf")
            plt.savefig(f"{algo}-{variable}.pdf")
            plt.close()
            os.chdir("..")
        os.chdir("..")
    os.chdir("..")
```