Academic Year 2023/2024

Internet Performance and Troubleshooting Lab

Politecnico di Torino

Report II

Analysis of Chargen

Group 3

Brendon Mendicino (s317639)

Alessandro Ciullo (s310023)

Davide Colaiacomo (s313372)

1 Network Configuration

The network configuration used during the experiments is the following:

Figure 1.1: Enter Caption

Host name	**IP address**	Role
H1	10.0.3.1/25	Server
H3	10.0.3.3/25	Client

Table 1.1: Net hosts

2 Chargen experiment: A Macroscopic view

2.1 How chargen works

Chargen is a character generator protocol standardized by RFC 864. The popular pattern works by sending lines composed of 72 characters taken from a sliding window over a circular buffer of 95 printable characters and terminated with `\r\n` for a total of 74 bytes per line.

2.2 The experiment steps

The following experiment aims to analyze how the fields of the **TCP** header change over time during a connection between a `Chargen server` and a `client`.

The first experiment was performed by executing a series of actions on the client-side, each separated by a 5 seconds delay, in the following order:

1. create a connection to the Chargen server, using the `Telnet` protocol.
2. try to decrease the size of the terminal window.
3. try to increase the size of the terminal window.
4. press the key combination `^C` on the client terminal.
5. press the key combination `^]` on the client to access the `Telnet` console.
6. press the `Enter` key on the client to resume the connection.
7. press the key combination `^]` on the client.
8. close the connection through the command `telnet> quit`.

As it noticeable in Figure 5.1a, the first 10 seconds (step 2 and 3) do not show any noticeable change in the slope of the graph; this means that resizing the window is not enough to cause a slowdown of the throughput as we expected and, actually, we'll find out a completely different result in the corresponding microscopic view of the experience.

When the client presses `^C`, Telnet will send some bytes to the server, which will represent the command ASCII notation of `^C +\r\n` (5 bytes corresponding to the only grow in the client sequence number), on top of that telnet will stop printing characters on screen, this will cause (probably) telnet to dump the bytes got from the socket instantly, thus causing the TCP buffer to get emptied faster that what the server is able to send to the client. For this reason we can see that the angle of the curve in the interval $[10s, 15s]$ is much greater than before, this means that the data being sent by the server to the client is much greater than before. This is further confirmed by the **Throughput Graph** shown in Wireshark Figure 5.1e which reaches $0.8Gb/s$. After this phase we press the `^]` combination and pass to the telnet terminal: this gradually interrupts the communication and will mark the beginning of a waiting procedure that we addressed as the **Keep-Alive** phase and analyzed it later in the report. Than we resumed the communication by pressing `Enter` and observed the same behavior of the `^C` phase till we returned in the telnet terminal mode and by writing `quit` we closed the connection.

From analysing the Figure 5.1d it's also noticeable that the server segment size is always changing, in the first 10 seconds this is explained by the fact that the server reaches the full size of the Client window, which implies that the segment will be truncated earlier than the MSS. This is fluctuation in the segment size is also present in the `^C` phase where the client receiving window is never saturated; the reason behind this is in the implementation of Chargen: the application makes use of the `PUSH` flag to delimit and push sets of completed lines. During the first part of the experiment in fact the sum of data from a push to the next one was always a multiple of 74 (justifying the truncation) but later even if this synchrony for what we thinks are implementation reasons this behavior continue.

One last thing to be noted is that, when the connection is closed at the end of the experiment (step 8), the Chargen server sends a **FIN/ACK** fragment to request the termination of the connection, then immediately sends a **RST/ACK** fragment without actually waiting for any acknowledgement for the former, leading to a connection shutdown without further availability from the server to receive any client's packets related to the current connection.

3 Chargen experiment: A Microscopic view

3.1 The handshake phase

As the connection between the Chargen server and the client relies on TCP, it is appropriate to dive deeper into the establishment of the connection between them. Taking for granted that the following packets all carry a 14 bytes `Ethernet header`, a 20 bytes `IP header` and a 20 bytes `TCP header` plus the `TCP options`, figure 3.1 shows, as expected according to the `three-way handshake` procedure, that:

1. the client starts sending a 74 bytes **SYN** fragment to the Chargen server.
2. the Chargen server answers back to the client with a 74 bytes **SYN/ACK** fragment.
3. the client finally answers back to the Chargen server with a 66 bytes **ACK** fragment.

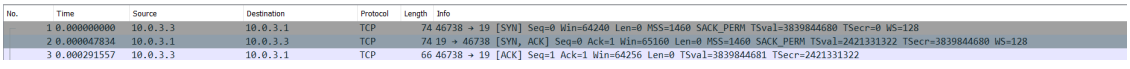


Figure 3.1: Wireshark - Three-way Handshake

3.1.1 SYN fragment (client → server)

Figure 5.2 shows that the `source port` is a random one picked above the well-known ports (0-1023), while the `destination port` is 19, which is the default port on which the Chargen service listens; the initial

`sequence number` is set (and considered relatively as 0), while the `acknowledgement number` is in this context to be ignored, since this is not an ACK fragment and so does not have any useful meaning; the flag **SYN** is the only one activated and the `Window` field specifies the current client's window size in bytes. Taking a closer look at the 20 bytes `options` section, the client includes:

- a 4 bytes `Maximum Segment Size` option as part of the MSS negotiation procedure.
- a 2 bytes `SACK permitted` option to allow the receiver to adopt the selective acknowledgement mechanism in the process of exchanging fragments.
- a 10 bytes `timestamps` option to keep track of the current RTT of the network between itself and the server.
- a 1 byte `No-operation` option for padding reasons, as TCP segments need to be of length that is multiple of 4.
- a 3 bytes `Window Scale` option to specify the `shifting count` of the window, in order to scale future its future sizes (this means that, given the `Window` field in the TCP header in future client's fragments, the real dimension of the client's window will be equal to `Window`$*2^{Shiftingcount}$ bytes)

3.1.2 SYN/ACK fragment (client ← server)

Analyzing the 5.3, it can be noticed that the `sequence number` of the Chargen server is set similarly to what happened for the client; the flags **SYN** and **ACK** are activated and the `acknowledgement number` is relatively set to one more with respect to the previous fragment's `sequence number`; this is due to the fact that, even if no payload has been sent, the Chargen server considers the byte containing the **SYN** flag as useful for the counting, avoiding ambiguities of interpretation when the client gets the response. It is to be noted that the 20 bytes `options` section is very similar to the one sent by the client, as it aims at completing the configuration of the connection; this means that the Chargen server is also specifying its own `MSS` and `Window Scale`, while also allowing the client to adopt the `Selective Acknowledgement` mechanism; consequentially, the relative 3 options will not be necessary in future fragments. The `timestamps` option and the `No-operation` option have the same purpose as before.

3.1.3 ACK fragment (client → server)

In conclusion, from the 5.4, it can be seen that the client increases its `sequence number` and `acknowledgement number` by 1, again to take into account the byte of the **SYN** flag received by the Chargen server; the `Window` field is now scaled as a consequence of the previous `Window Scale` option and the flag **ACK** is the only one activated. From this segment onward, the `options` section only contains the **timestamps** option (10 bytes) and 2 **No-Operation** options (1 byte each), resulting in 12 bytes of options for all the fragments that will follow. The connection between the client and the Chargen server is now established.

3.2 Resizing the terminal window

One of the ways we have to interact with the application is to resize the window in which the Telnet process resides; by doing so, we change the number of characters showed on the screen simultaneously. During the lectures, we observed that the slope of the sequence number graph grows inversely proportional to the size of the terminal window: when the window is bigger the application needs to manage more characters and the load on the system is bigger, causing a slowdown that is reflected in a slower emptying of the `RWND` and a decrease in the throughput.

What we observe in our captures is different: as shown in the 5.5 graph, the throughput increases slightly during the second part of the experiment. What differs between the two experience is the environment. In the first case we have a slower and virtualized OS, and so the burden of the bigger needs of the window terminal is the bottle-neck, while in the second case the hardware is not only able to keep up with the increasing needs but now there are also more characters that can be removed from the socket buffer and printed on screen, generating a less congested `RWND`.

During the first two phases something hardly unnoticed are the fluctuations of the client `RWND` that start at 64240B and grows till 240000B, than the congestion windows of the server goes to regime. From that point the `RWND` often goes to 0 bytes and than spikes back to around 60000B with an average of 25000B as shown in the figure 3.2. By analyzing the Wireshark capture we discover how, periodically, the server sends packet until the last advertised client `RWND` is full: this is highlighted by a TCP segments marked as **TCP Window Full** by Wireshark. To this situation the client can give three possible responses:

- Can acknowledge the received segments and put the new, non zero, window size in the corresponding field.
- Can acknowledge the received segments and than send a replicated TCP with an updated receiving window size (this kind of packets are marked as **TCP Window Update** by Wireshark.
- Can send an acknowledge of the last received segments with a zero window size (marked as **TCP Zero Window** by Wireshark) so as to provide confirmation to the server about the lack of space in the socket buffer and keep him waiting till the upcoming TCP Window Update.

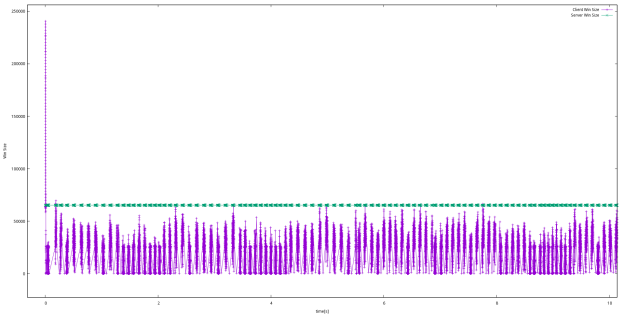


Figure 3.2: Client and Server RWND during phase 1-2

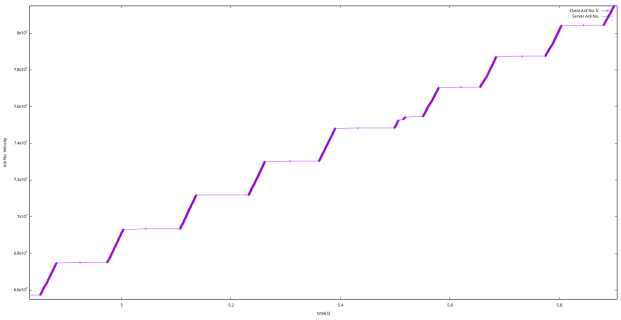


Figure 3.3: A microscopic view of the first two phases

A last scenario can be observed by zooming a lot on our server seq number graph as shown in graph 3.3, periodically the server after sending a variable quantity of data based on implementation parameters and current connection status interrupts is communication before saturating the last advertised `RWND`. If the corresponding `ACK` from the client doesn't arrive in the next 10-15ms the server re-transmits the last packet (marked as **TCP Retransmission** by Wireshark and highlighted as the isolated segment in nearly each plateau of the graph 3.3) in order to force the client to answer , something similar to what we'll see in the Keep-Alive section: by observing the `ctrl+c` phase of the experiment we can assume that the re-transmission timer is parameterized also around the `RWND` of the other host because in that experience, in which the client receiving window is a lot bigger, we don't have retransmission even if we have waiting time before some `ACK` longer than 30ms and that's the probable reason because some plateau has no TCP retransmission.

3.3 System Overloading

This experiment aims to provide evidence of a **decreased throughput** when the system is begin overloaded.

In order to do this we opened two terminals, in one of them a openssl benchmark was run, while on the other a telnet session was opened with the Chargen server. The reason why we used openssl is because by launching the benchmark mode, openssl starts performing as many cryptographic operations as possible, because they are very demanding in terms of CPU power required, the whole system is affected by this operations. At the 15th second we stopped the openssl program, the Figure 5.6 shows the pace of the graph, shortly after we see an increase in the slope of the Client Ack No., and further supported by the time derivative of the Ack, which shows greater peaks after the 15 seconds threshold.

We can clearly state after this experiment the fact that we did not see any noticeable change in the original experiment while performing a terminal window resizing in Subsection 3.2, is totally justified by the fact that the PC we test were able to handle a simple window resizing.

3.4 Opening the Telnet terminal

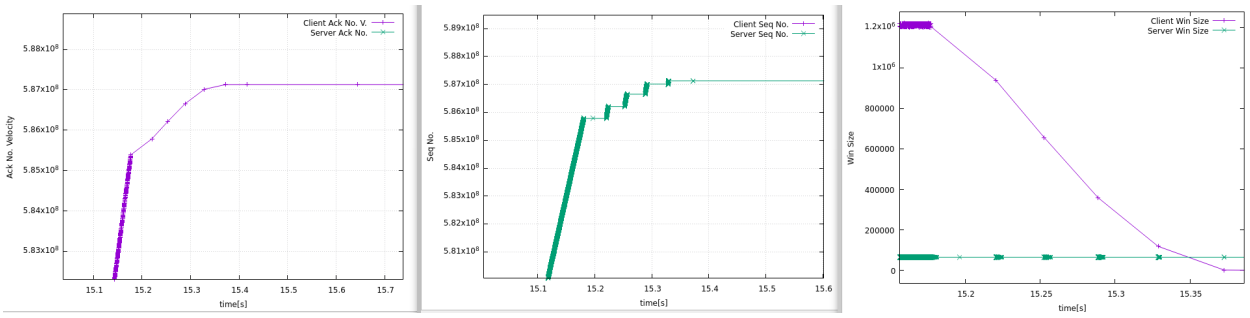


Figure 3.4: Closeup during the terminal opening

When the key combination `^]` gets pressed, the client enters into the `command mode` of Telnet; at this point, the client starts sending the server `ACK` segments with always decreasing window size, clearly illustrated in the third graph, in the end the client sends an `ACK` with `win=0`, which means that the client has put the connection on hold and doesn't want to receive any more data from the server. This could be due to the fact that when the telnet terminal is opened all the data to the client cannot be displayed to the screen so TCP is not able to pass this information from its cache to the telnet application and the window slowly fills up, up to the point where it is completely full. Only when the client will press Enter from the telnet terminal, the application will start printing again character on the screen thus start consuming the TCP buffer, and again increase its window size. Now the problem is: *What happens in the time interval between when the telnet terminal is opened and it gets closed?*

3.5 The Keep-Alive phase

```
473428 15.372825034 10.0.3.1 10.0.3.3  TCP   154 [TCP Window Full] 19 → 46738 [PSH, ACK] Seq=587131193 Ack=6 Win=65280 Len=88 TSval=2421346695 TSecr=3839860053
473429 *REF*        10.0.3.3 10.0.3.1  TCP    66 [TCP ZeroWindow] 46738 → 19 [ACK] Seq=6 Ack=587131281 Win=0 Len=0 TSval=3839860097 TSecr=2421346695
473430 0.227632909  10.0.3.1 10.0.3.3  TCP    66 [TCP Keep-Alive] 19 → 46738 [ACK] Seq=587131280 Ack=6 Win=65280 Len=0 TSval=2421346967 TSecr=3839860097
473431 0.227974504  10.0.3.3 10.0.3.1  TCP    66 [TCP ZeroWindow] 46738 → 19 [ACK] Seq=6 Ack=587131281 Win=0 Len=0 TSval=3839860325 TSecr=2421346695
473432 0.679613379  10.0.3.1 10.0.3.3  TCP    66 [TCP Keep-Alive] 19 → 46738 [ACK] Seq=587131280 Ack=6 Win=65280 Len=0 TSval=2421347419 TSecr=3839860325
473433 1.607819492  10.0.3.1 10.0.3.3  TCP    66 [TCP Keep-Alive] 19 → 46738 [ACK] Seq=587131280 Ack=6 Win=65280 Len=0 TSval=2421348347 TSecr=3839860325
473434 1.608155722  10.0.3.3 10.0.3.1  TCP    66 [TCP ZeroWindow] 46738 → 19 [ACK] Seq=6 Ack=587131281 Win=0 Len=0 TSval=3839861705 TSecr=2421346695
473435 3.431838993  10.0.3.1 10.0.3.3  TCP    66 [TCP Keep-Alive] 19 → 46738 [ACK] Seq=587131280 Ack=6 Win=65280 Len=0 TSval=2421350171 TSecr=3839861705
473436 3.432188048  10.0.3.3 10.0.3.1  TCP    66 [TCP ZeroWindow] 46738 → 19 [ACK] Seq=6 Ack=587131281 Win=0 Len=0 TSval=3839863529 TSecr=2421346695
473437 4.578258343  10.0.3.3 10.0.3.1  TCP    66 [TCP Window Update] 46738 → 19 [ACK] Seq=6 Ack=587131281 Win=91008 Len=0 TSval=3839864675 TSecr=2421346695
```

Figure 3.5: Server tries to keep alive the connection

This figure has the time of the segments referencing the `TCP ZeroWindow` segment. By doing that it is possible to see how the server is able to keep alive the connection.

As we can see from the Figure 3.5 when the client send the `ZeroWindow` segment the server goes into **keep-alive mode**, this is done by sending a special kind of segments, that are able to force an answer from the client without sending any new data, that are called **TCP Keep-Alive** by Wireshark. This segments don't carry any data but their sequence number is peculiar, it corresponds to the next Sequence Number minus 1 (e.g. in this case, after the first `ZeroWindow`, the next sequence no. would be the sequence number of the last segment plus its length: $587131193 + 88 = 587131281$). This is able to force an answer from the client which will acknowledge the fact that the server already sent this bytes along its Window size which may be updated or not.

This probe segments are not sent with the same time interval, but it **increases** over time, from the trace is not very clear what is the multiplying factor, it start at around 3x and it does look approaching 2x.

Reading the TCP man pages it is clear also the keep-alive mode has a timeout which by default is 7200 seconds (2 hours), and all the TCP parameters can be located at `/proc/sys/net/`. The file which gives out the keep-alive timeout is `tcp_keepalive_time`.

4 Cable-Cut

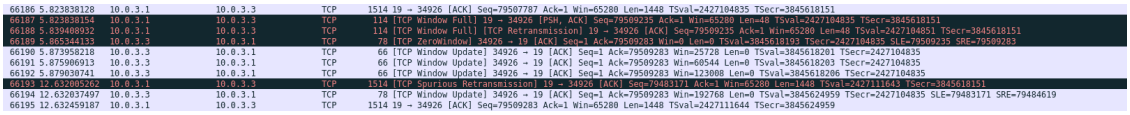


Figure 4.1: Interface of the client during cable cut

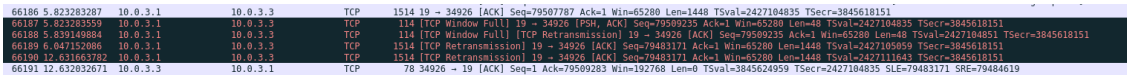


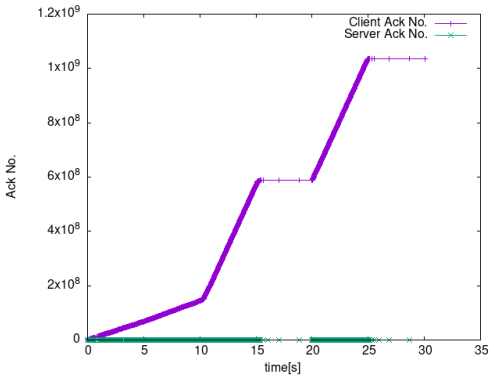
Figure 4.2: Interface of the server during cable cut

Another experiment we performed was to try and cut the cable while the connection between the client and the server was already opened.

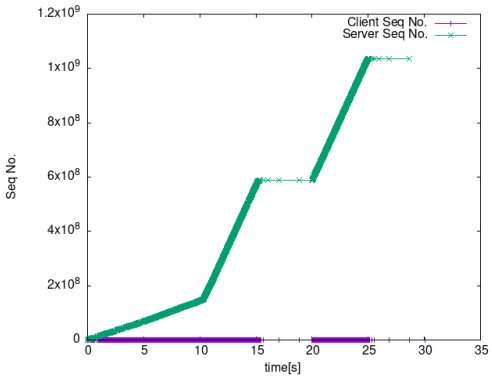
What happened was that unlike the Subsection 3.5, where the server tried to keep-alive the connection by sending the client special segments, the server only tried to to do one retransmission after the cable was cut, because the first retransmission was received by the client, also the client tried to send three Ack, but none of them reached the server, during this period neither the server nor the client sent any other segments. Only after the cable was put in place the server sent another retransmission to the client and the connection continued just as normal as before.

5 Appendix

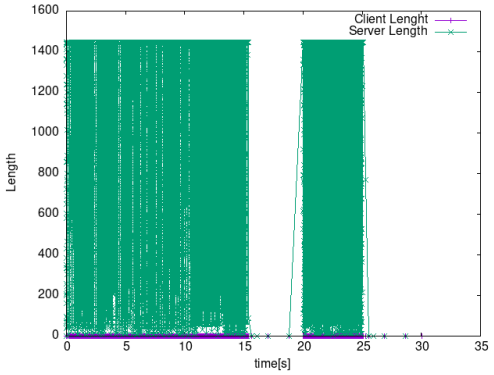
5.1 Chargen captures



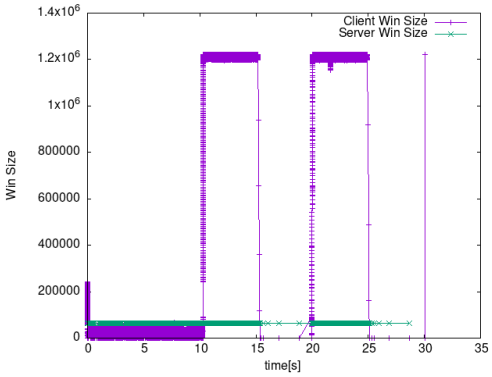
(a) Client/Server Ack No.



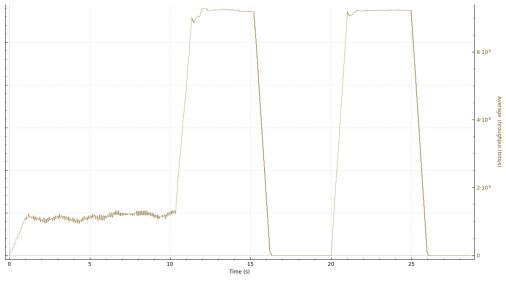
(b) Client/Server Seq No.



(c) Client/Server Len



(d) Client/Server Win



(e) Server Throughput

Figure 5.1: Chargen Capture

5.2 Handshake Segments

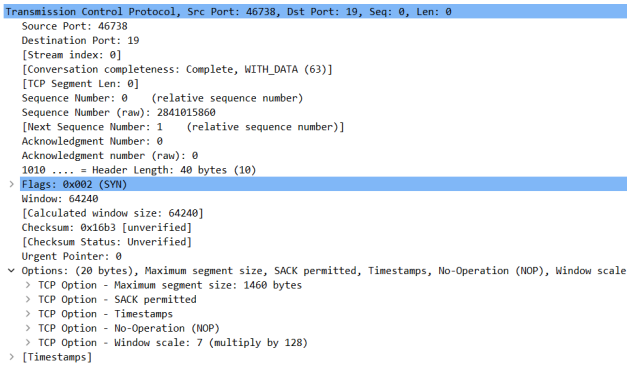


Figure 5.2: SYN fragment

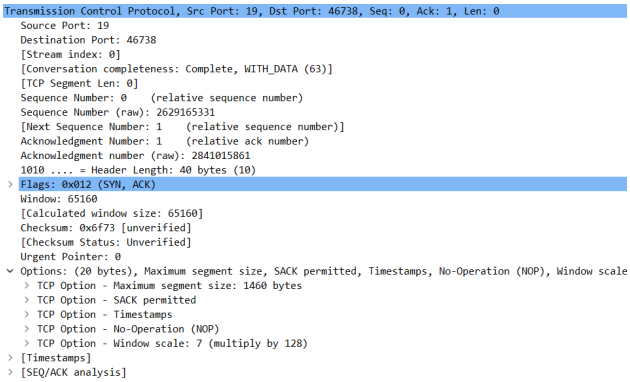


Figure 5.3: SYN/ACK fragment

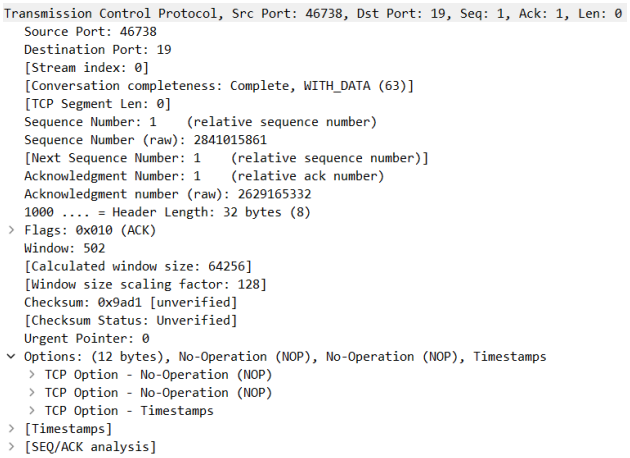


Figure 5.4: ACK fragment

5.3 Throughput

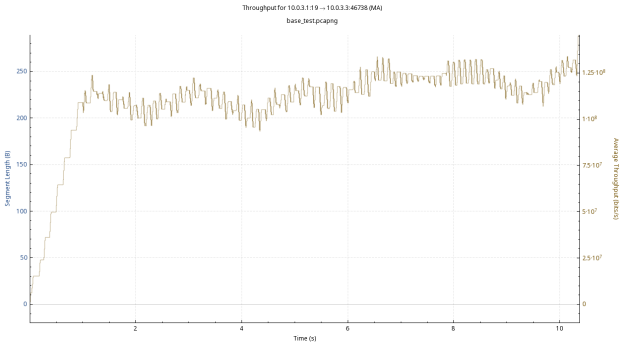
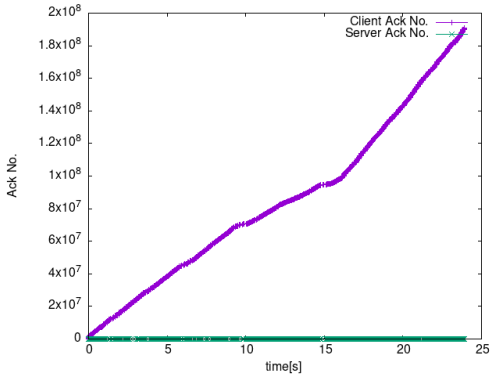
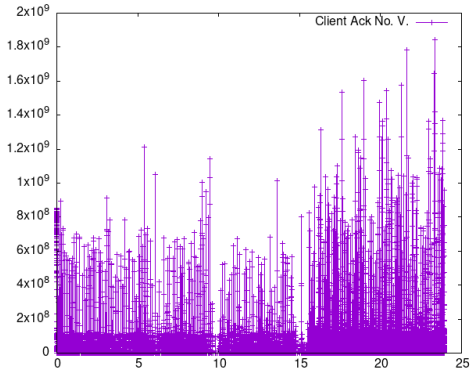


Figure 5.5: Throughput during phase 1-2

5.4 System Overloading



(a) Client Acknowledge



(b) Client Acknowledge Velocity

Figure 5.6: System overloading client ACK graphs

5.5 Script for image generation

```
#!/bin/sh

for arg in $@; do
  case $arg in
    -d|--dynamic)
      DYNAMIC=1
      shift
      ;;
    -*|--*)
      echo "Invalid option!"
      exit 1
      ;;
    *)
      ;;
  esac
done

if [ ! -f $1 ] || [ -z $1 ]; then
  echo "$0: Wrong params!"
  exit 1
```

```
fi

echo "generating client values..."
# Generate client time and values, remove first ACK packet
tail -n +3 $1 | grep -P "> *19" | tr -s "\"" "," | tr "," " " | tr -s " " | cut -d " " -f 3 > client_time.txt
tail -n +3 $1 | grep -P "> *19" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Ack=)([^ ]*)" > client_ack.txt
tail -n +3 $1 | grep -P "> *19" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Seq=)([^ ]*)" > client_seq.txt
tail -n +3 $1 | grep -P "> *19" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Len=)([^ ]*)" > client_len.txt
tail -n +3 $1 | grep -P "> *19" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Win=)([^ ]*)" > client_win.txt

paste client_time.txt client_ack.txt > client_ta.txt
paste client_time.txt client_seq.txt > client_ts.txt
paste client_time.txt client_win.txt > client_tw.txt
paste client_time.txt client_len.txt > client_tl.txt

echo "generating server values..."
# Generate server time and values
tail -n +2 $1 | grep -P "19 *>" | tr -s "\"" "," | tr "," " " | tr -s " " | cut -d " " -f 3 > server_time.txt
tail -n +2 $1 | grep -P "19 *>" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Ack=)([^ ]*)" > server_ack.txt
tail -n +2 $1 | grep -P "19 *>" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Seq=)([^ ]*)" > server_seq.txt
tail -n +2 $1 | grep -P "19 *>" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Len=)([^ ]*)" > server_len.txt
tail -n +2 $1 | grep -P "19 *>" | tr -s "\"" "," | tr "," " " | tr -s " " | grep -oP "(?<=Win=)([^ ]*)" > server_win.txt

paste server_time.txt server_ack.txt > server_ta.txt
paste server_time.txt server_seq.txt > server_ts.txt
paste server_time.txt server_win.txt > server_tw.txt
paste server_time.txt server_len.txt > server_tl.txt

echo "plotting..."

if [ $DYNAMIC ]; then
  echo "Ack..."
  gnuplot <<EOF &
set xlabel "time[s]"
set ylabel "Ack No. Velocity"
plot "client_ta.txt" using 1:2 title "Client Ack No." with linespoint, "server_ta.txt" using 1:2 title "Server Ack No." with linespoint
pause 9999
EOF

  echo "Ack Velocity..."
  gnuplot <<EOF &
set xlabel "time[s]"
set ylabel "Ack No."
dx(x) = (x0=x1, x1=column(x), (x0 + x1) / 2.)
dy(y) = (y0=y1, y1=column(y), (y1 - y0)/(x1 - x0))
plot x1=y1=NaN "client_ta.txt" using (dx(1)):(dy(2))  title "Client Ack No. V." with linespoint
pause 9999
EOF

  echo "Seq..."
  gnuplot <<EOF &
set xlabel "time[s]"
set ylabel "Seq No."
plot "client_ts.txt" using 1:2 title "Client Seq No." with linespoint, "server_ts.txt" using
```

```
     1:2 title "Server Seq No." with linespoint
pause 9999
EOF

  echo "Win..."
  gnuplot <<EOF &
set xlabel "time[s]"
set ylabel "Win Size"
plot "client_tw.txt" using 1:2 title "Client Win Size" with linespoint, "server_tw.txt"
    using 1:2 title "Server Win Size" with linespoint
pause 9999
EOF

  echo "Len..."
  gnuplot <<EOF &
set xlabel "time[s]"
set ylabel "Length"
plot "client_tl.txt" using 1:2 title "Client Length" with linespoint, "server_tl.txt" using
    1:2 title "Server Length" with linespoint
pause 9999
EOF

else
  echo "Ack..."
  gnuplot -persist <<EOF
set term png
set output "ack.png"
set xlabel "time[s]"
set ylabel "Ack No."
plot "client_ta.txt" using 1:2 title "Client Ack No." with linespoint, "server_ta.txt" using
     1:2 title "Server Ack No." with linespoint
EOF

  echo "Ack Velocity..."
  gnuplot -persist <<EOF
set term png
set output "ack-vel.png"
dx(x) = (x0=x1, x1=column(x), (x0 + x1) / 2.)
dy(y) = (y0=y1, y1=column(y), (y1 - y0)/(x1 - x0))
plot x1=y1=NaN "client_ta.txt" using (dx(1)):(dy(2))  title "Client Ack No. V." with
    linespoint
EOF

  echo "Seq..."
  gnuplot -persist <<EOF
set term png
set output "seq.png"
set xlabel "time[s]"
set ylabel "Seq No."
plot "client_ts.txt" using 1:2 title "Client Seq No." with linespoint, "server_ts.txt" using
     1:2 title "Server Seq No." with linespoint
EOF

  echo "Win..."
  gnuplot -persist <<EOF
set term png
set output "win.png"
set xlabel "time[s]"
set ylabel "Win Size"
plot "client_tw.txt" using 1:2 title "Client Win Size" with linespoint, "server_tw.txt"
    using 1:2 title "Server Win Size" with linespoint
EOF

  echo "Len..."
  gnuplot -persist <<EOF
set term png
```

```
set output "len.png"
set xlabel "time[s]"
set ylabel "Length"
plot "client_tl.txt" using 1:2 title "Client Lenght" with linespoint, "server_tl.txt" using
    1:2 title "Server Length" with linespoint
EOF
fi

echo "Finished"
```