Network Architecture: Lab 2 — Transport Layer TCP/UDP

Students:

- Median Mehi Edine
- Mahmoud El Sayed
- Felipe Alvarez Suarez

Date: 23/10/2025

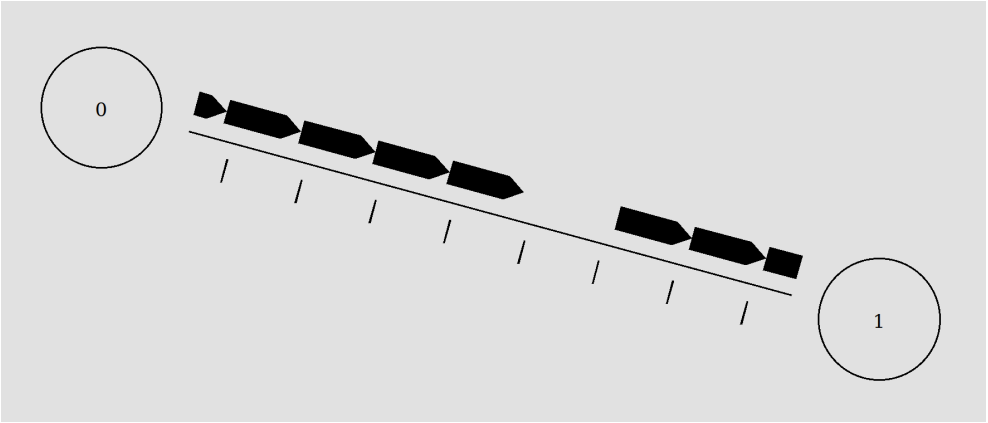
Outline

Example Analysis

```
#Create simulator instance
set ns [new Simulator]

# Creating a trace files that will hold information about the size
of the congestion window at each time t of the simulation
set ftrace [open trace.tr w]

# Creating a nam file that will hold data relevant to the nam
program to visuals the sending of packets and receiving of
acknowledgements by both nodes
set nf [open out.nam w]
$ns namtrace-all $nf

# Used at the end of the program, it uses the info stored in the
trace file to construct a nam file to visualize
proc finish {} {
    global ns ftrace nf
    $ns flush-trace         ;# flush buffers to files
    close $nf               ;# close NAM file
    close $ftrace           ;# close text trace
    exit 0                  ;# terminate the simulation
}

#Record TCP cwnd to the trace file
proc tracewindow {} {
    global tcp0 ftrace
    set ns [Simulator instance]
    if {![info exists tcp0]} { return }   ;# if TCP not set, stop
    set time 0.001                         ;
    set now [$ns now]                      ;# current sim time
    set now [format "%.3f" $now]           ;# format time (ms
precision)
    set cwnd [$tcp0 set cwnd_]             ;# read TCP congestion
window
    puts $ftrace "$now $cwnd"              ;# write "time cwnd" to
trace
    $ns at [expr $now+$time] {tracewindow} ;# reschedule next sample
}
```

```
#Set TCP defaults: packet size and max cwnd (pkts)
Agent/TCP set packetSize_ 1500
Agent/TCP set maxcwnd_ 30

#Create sender node and attach a TCP agent
set n0 [$ns node]
set tcp0 [new Agent/TCP]
$ns attach-agent $n0 $tcp0

#Attach an FTP application over the TCP agent
set ftp [new Application/FTP]
$ftp attach-agent $tcp0

#Create receiver node and attach a TCP sink
set n1 [$ns node]
set tcp1 [new Agent/TCPSink]
$ns attach-agent $n1 $tcp1

#Connect the two nodes with a duplex point-to-point link
#Link: 10 Mb bandwidth, 10 ms delay, DropTail queue
$ns duplex-link $n0 $n1 10Mb 10ms DropTail

#Connect TCP source to TCP sink
$ns connect $tcp0 $tcp1

#Schedule: start FTP at 0.0s, stop at 0.7s
$ns at 0.0 {$ftp start}
$ns at 0.7 {$ftp stop}

#Start periodic cwnd tracing at 0.0s
$ns at 0.0 {tracewindow}

#Finish simulation at 1.0s (flush+exit)
$ns at 1.0 {finish}

#Run the simulation
$ns run
```

NAM Visualization:



This screenshot shows the Network Animator (NAM) visualization for the example.tcl script.

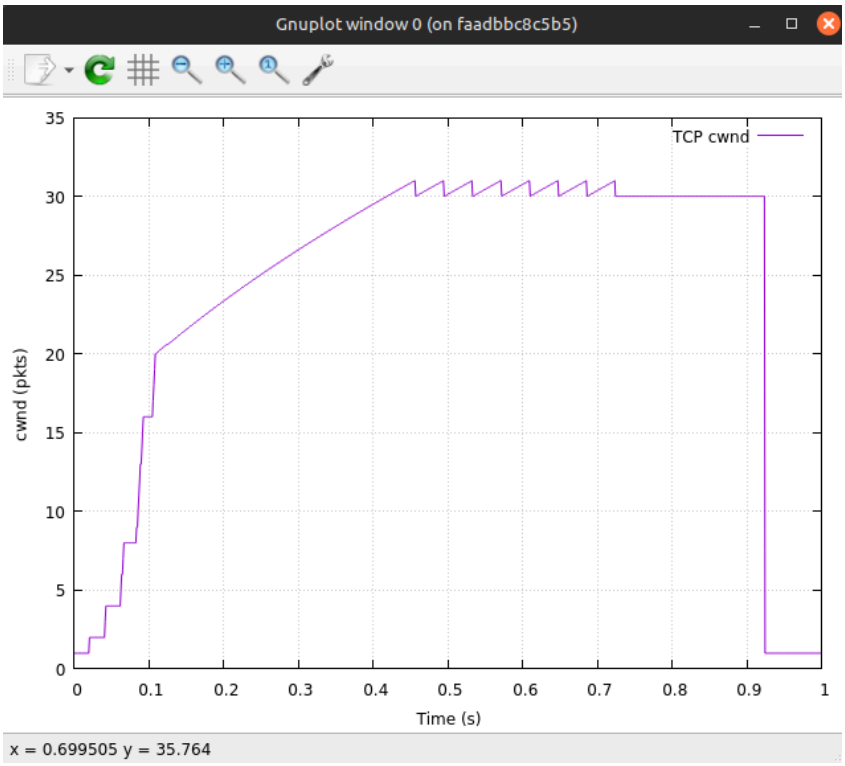
Topology: It displays the simple two-node topology defined in the script: Node 0 on the left and Node 1 on the right.

Link: The nodes are connected by a single duplex link, which allows data to flow in both directions simultaneously.

Traffic Flow: The script configures an FTP application (a TCP sender) on Node 0 and a TCP "sink" (a receiver) on Node 1 . The small dashes moving along the link from 0 to 1 represent the TCP data segments from the FTP application. In response, TCP ACKs (acknowledgments) are sent from Node 1 back to Node 0. This visualizes the logical end-to-end transport of packets.

Sender’s congestion window size with function of time

We have used gnuplot library to plot the results using this command: **gnuplot -persist -e "set datafile sep ' '; set xlabel 'Time (s)'; set ylabel 'cwnd (pkts)'; set grid; plot 'trace.tr' using 1:2 with lines title 'TCP cwnd'"**



Analysis of the results:

This graph plots the sender's (Node 0's) congestion window (cwnd) size over time, which is the data captured in trace.tr. The behavior shows classic TCP phases:

TCP Slow Start (Approx. 0.0s - 0.15s): The plot begins with the cwnd at 1 MSS (Maximum Segment Size). It then increases exponentially, doubling roughly every Round-Trip Time (RTT). This is visible as the steep, stair-stepped curve. The "steps" occur because the sender increases its cwnd for every ACK it receives.

Congestion Avoidance (Approx. 0.15s - 0.45s): At ~0.15s, the growth rate changes from exponential to linear. This indicates the cwnd has reached the ssthresh (Slow Start Threshold) and the sender has switched to the Congestion Avoidance phase. In this phase, the sender increases its cwnd additively (by about 1 MSS per RTT), "probing" the network for more bandwidth more cautiously.

Window Capping (Approx. 0.45s - 0.7s): At ~0.45s, the cwnd hits 30 and stops growing. This is not due to network congestion (packet loss), but because the script explicitly sets the maximum window size to 30 (Agent/TCP set maxcwnd_ 30). The small "sawtooth" pattern in this phase shows the sender filling this capped window.

Application Stop (0.7s - 1.0s): At 0.7s, the FTP application is scheduled to stop. The tracewindow procedure keeps polling the cwnd, which remains at its last value (30) until the simulation itself prepares to terminate at 1.0s, causing the final drop.

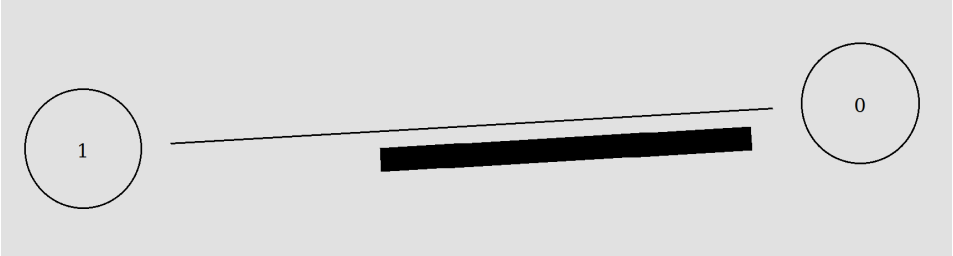
Exercise 1

Changing values and visualizing changes

After changing the delay, length of the simulation, and the time interval in the function **tracewindow**:

```
$ns duplex-link $n0 $n1 10Mb 40ms DropTail

# Connect TCP source to TCP sink (end-to-end flow)
$ns connect $tcp0 $tcp1

# Schedule: start FTP at 0.0s, stop at 0.7s
$ns at 0.0 {$ftp start}
$ns at 3.0 {$ftp stop}

# Start periodic cwnd tracing at 0.0s
$ns at 0.0 {tracewindow}

# Finish simulation at 1.0s (flush+exit)
$ns at 3.1 {finish}
```

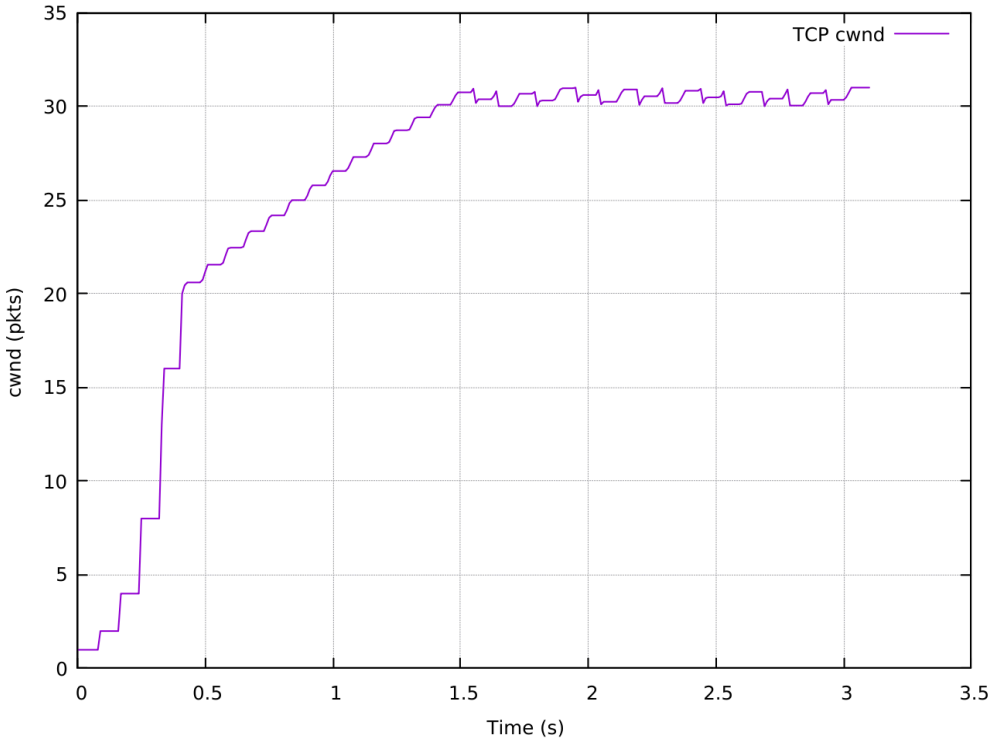
We get the following, **NAM Visualization:**



Topology: We see the same two-node setup from before. Based on the packet flow, Node 0 (right) is the TCP sender (FTP source), and Node 1 (left) is the TCP sink (receiver).

Packet Flow: The thick, dark bar represents the stream of TCP data segments heading from Node 0 to Node 1. The thinner, single line represents the returning TCP ACKs from Node 1 to Node 0. The density of the data packets indicates that the sender has a large window of data "in-flight," filling the network link.

Congestion Window Graph



This graph shows the same fundamental TCP behavior as the baseline example, but the 40ms delay modification (versus 10ms) makes the difference. **This effectively quadruples the RTT to ~80ms, which "stretches out" all phases of the window growth**:

Slower Slow Start (0.0s - 0.4s): The initial exponential growth phase is elongated. Because ACKs take 4x longer to return, the ramp-up (which happens per-RTT) is 4x slower against the wall clock.

Flatter Congestion Avoidance (0.4s - 1.5s): The subsequent linear growth phase has a much shallower slope. The window grows additively per RTT, so the 4x longer RTT results in a 4x slower linear climb.

Identical Capping (1.5s - 3.0s): The window still flattens at the script's maxcwnd_ limit of 30, confirming the sender is limited by this parameter, not by network congestion (packet loss).

Creating a CSV output file

A small note, we wish to keep old trace file alongside a new CSV file. So for that case we open a new CSV file:

```
5 set ftrace [open trace.tr w]
6 set csv [open trace.csv w]
```

Add it as a global variable to the **tracewindow** function, and output the columns in csv format to the csv file, and close it in the **finish** function:

```
20 proc finish {} {
21     global ns ftrace nf
22     $ns flush-trace         ;# flush buffers to files
23     close $nf               ;# close NAM file
24     close $ftrace           ;# close text trace
25     close $csv
26     exit 0                  ;# terminate the simulation
27 }
28
29 # Record TCP cwnd to the trace file
30 proc tracewindow {} {
31     global tcp0 ftrace csv
32     set ns [Simulator instance]
33     if {![info exists tcp0]} { return }   ;# if TCP not set, stop
34     set time 0.001                         ;
35     set now [$ns now]                      ;# current sim time
36     set now [format "%.3f" $now]           ;# format time (ms precision)
37     set cwnd [$tcp0 set cwnd_]             ;# read TCP congestion window
38     set seqno [$tcp0 set seqno_]
39     set ackno [$tcp0 set ack_]
40     puts $ftrace "$now $cwnd"              ;# write "time cwnd" to trace
41     puts $csv "$now;$cwnd;$seqno;$ackno"             ;# write "time cwnd" to trace
42     $ns at [expr $now+$time] {tracewindow} ;# reschedule next sample
43 }
```

Exercise 2

Parameterizing the CSV file

Similarly to bash and the C language, we read from the **$argv** array the new params:

```
if {[llength $argv] < 3} {
    puts stderr "Usage: ns ex2.tcl <cwnd_pkts> <bandwidth_Mb> <delay_ms>"
    exit 1
}

set cwnd_pkts [lindex $argv 0]
set bandwidth_Mb [lindex $argv 1]
set delay_ms [lindex $argv 2]
```

And replace the corresponding values where needed:

```
Agent/TCP set maxcwnd_ $cwnd_pkts
```

```
$ns duplex-link $n0 $n1 ${bandwidth_Mb}Mb ${delay_ms}ms DropTail
```

Creating a bash script

Our bash script is mainly this simple function, calculating the throughput is done from an awk file since it is easier to read a csv file using awk.

```
run_and_analyze() {
    local CWND=$1
    local BW=$2
    local DELAY=$3
    local OUTPUT_FILE=$4

    echo -n "  CWND=$CWND, BW=${BW}Mb, Delay=${DELAY}ms... "

    # Running the simulation
    ns $TCL_SCRIPT $CWND $BW $DELAY

    # Reading the output of the csv the trace file using the external AWK script
    # The AWK script reads trace.csv and prints the throughput to stdout
    # The -f flag specifies the AWK script file.
    THROUGHPUT_Mbps=$(awk -f analyze_throughput.awk trace.csv)

    # Just some  error handeling
    if [[ -z "$THROUGHPUT_Mbps" || "$THROUGHPUT_Mbps" == "0.0" ]]; then
        echo "FAILED (Analysis error or 0.0 Mbps)"
        THROUGHPUT_Mbps="ERROR"
    else
        echo "done (throughput=${THROUGHPUT_Mbps} Mbps)"
    fi

    # Appending results to our output CSV file to plot later
    echo "$CWND;$BW;$DELAY;$THROUGHPUT_Mbps" >> "$OUTPUT_FILE"
}
```

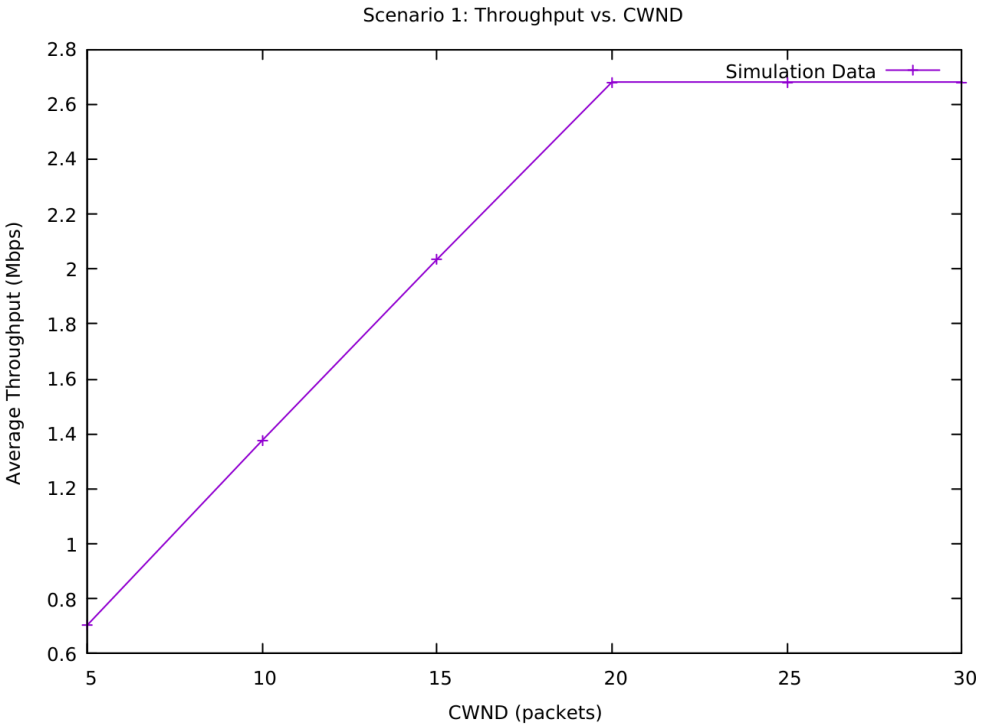
This AWK script reads the CSV output from our TCL file which will calculate the throughput:

```
# Process the body of the file (skip the header if one exists)
{
    # Check if the first field is a number (to skip a text header if present)
    if ($1 ~ /^[0-9]/) {
        # The 'ackno' column is the 4th field
        CURRENT_ACK = $4
    }
}

END {
    # If the file was empty or no data was processed
    if (NR <= 1) {
        exit
    }

    TOTAL_PKTS_ACKED = CURRENT_ACK
    TOTAL_BITS = TOTAL_PKTS_ACKED * PACKET_SIZE * 8
    THROUGHPUT_bps = TOTAL_BITS / SIM_DURATION
    THROUGHPUT_Mbps = THROUGHPUT_bps / 1000000.0

    # Print the final result to standard output (where the Bash script will read it)
    printf "%.4f", THROUGHPUT_Mbps
}
```

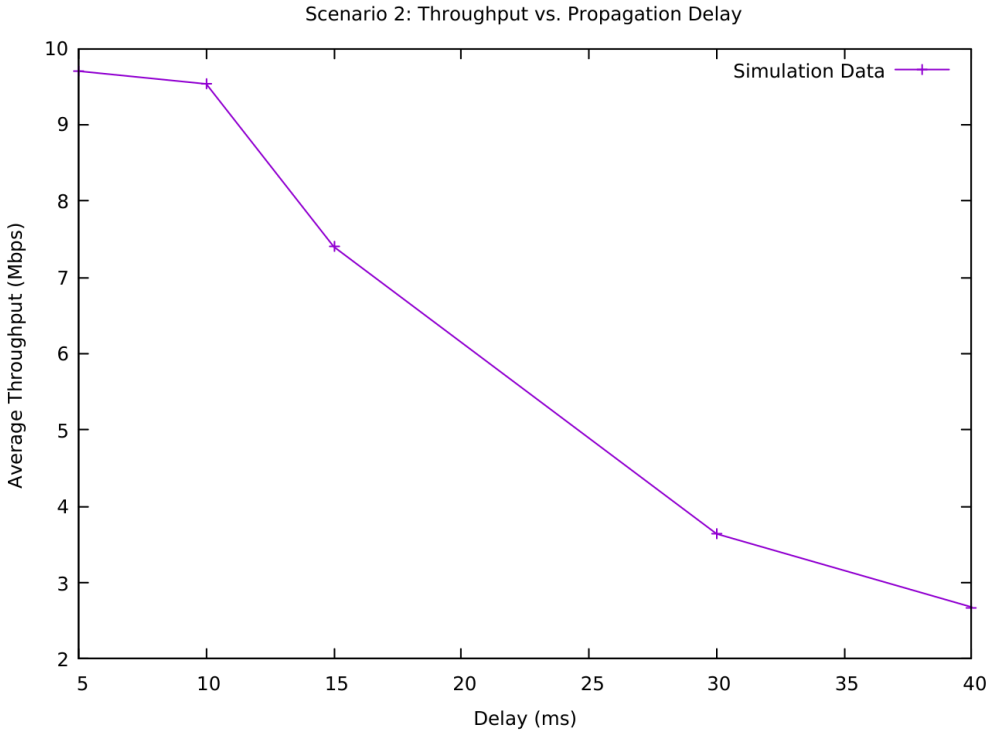
Then our bash file will output 3 CSV files for each scenario, which we plot next.

Plotting the results:

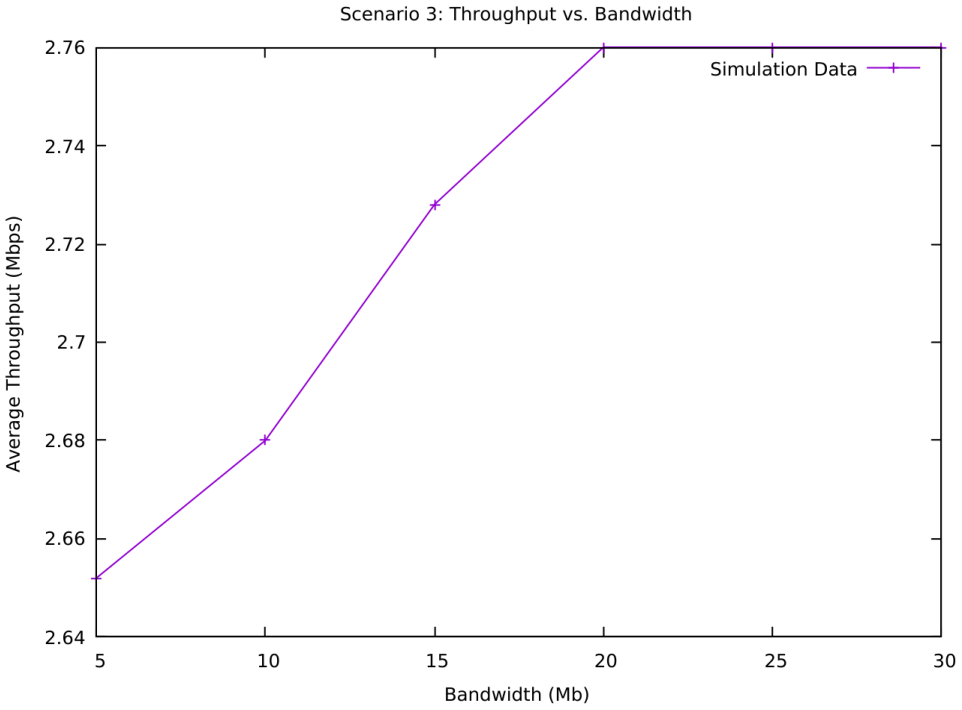
Scenario 1



Scenario 2



Scenario 3



Analysis and conclusion

For **Scenario 1** in the following regions**:**

- **Linear Region:** As the maximum congestion window increases from 5 to 20 packets, the TCP sender is able to inject more data into the pipe during each RTT. Since the link capacity is greater than the rate at which the data is being injected, the throughput is limited solely by the max cwnd.
- **Plateau Region:** The throughput flattens out at ≈2.67 Mbps when cwnd max reaches 20 packets. This means that 20 packets is the minimum window size required to completely fill the link's capacity given the delay. Setting the window larger than this threshold provides no further benefit, as the link simply cannot transport data faster than its physical limit.

For **Scenario 2** throughput is inversely proportional to the RTT. As the one-way delay increases from 5 ms to 40 ms, the RTT increases, giving the TCP sender less time to send data before waiting for an acknowledgement.

For **Scenario 3** in the following regions At the plateau**:** the throughput remains nearly constant at ≈2.76 Mbps regardless of the bandwidth. This occurs because the fixed cwnd max and RTT are too small to ever fill a link operating at 5 Mb or higher.

From this we can know that it is possible to forecast the max avg throughput behavior. The formula is concluded by determining the bottleneck limit of the system, which is the minimum of two constraints:

1. the physical link capacity (C)
2. and the rate at which the protocol can send data, constrained by the cwnd max and the RTT.

Throughput is defined as the amount of data transmitted per unit of time:

$$\text{Throughput} = \frac{\text{Data Transferred}}{\text{Time Taken}}$$

In this ideal scenario the sender attempts to keep the data pipe full, transmitting an entire window of data every RTT:

$$\text{Throughput}_{\text{Window}} = \frac{\text{Window Size (bits)}}{\text{RTT (seconds)}}$$

The actual throughput achieved will be the minimum of the physical rate and the protocol's rate:

Throughput avg=min(Link Capacity,Window-Limited Rate)

Where the Link Capacity=C (bits/second)

And the window size is limited by the maximum congestion window set in the system and the packet size (S, in bytes). The RTT is primarily determined by the propagation delay.

Giving us:

$$\text{Window Size (bits)} = \text{cwnd}_{\text{max}} \times S \times 8$$

$$\text{RTT} \approx 2 \times \text{Delay}$$

$$\text{Window-Limited Rate} = \frac{\text{cwnd}_{\text{max}} \times S \times 8}{2 \times \text{Delay}_{\text{prop}}}$$

Combining these two constraints gives the general formula:

$$\text{Thr}_{\text{avg}} = \min\left(\mathbf{C}, \quad \frac{\text{cwnd}_{\text{max}} \times \mathbf{S} \times \mathbf{8}}{\mathbf{2} \times \text{Delay}_{\text{prop}}}\right)$$

Exercise 3

Given Parameters:

- Link span: 2000 km
- Link capacity : 155.5 Mb/s
- Propagation speed: 200,000 km/s
- CWND : 65kb
- Transmission delays are negligible

1. To calculate the time it takes for a signal to travel across the link = link span/propagation speed = 2000km/200,000km/s = 0.01s => 10ms
2. To calculate the round trip time = one way propagation delay x 2 = 10ms x2 = 20 ms
3. To calculate the bandwidth delay product = link capacity x rtt = 155.5Mb/s x 0.02s = (155.5 x 10^6 bits/s) x 0.02s = 3,110,000 bits => /8 => 388,750 bytes
   When comparing the bandwidth delay product(388.7kB) and the max CWND(65kB) we see that the bandwidth delay product is much larger that the CWND

- The 65kB limit can only have a max of 65kB of data at a time. To fill the entire network (bandwidth delay product of ~380kB), we would need at least a CWND of 380kB.
- The maximum achievable throughput is limited by the maxCWND.
- Throughputmax = CWNDmax/RTT => 65kB/0.02s = 3250kB/s ~ 26Mb/s which is way less than the link's physical capacity of 155.5 Mb/s.

Conclusion: the 65kB limit imposes a significant constraint on this high speed network, preventing the connection from achieving its potential and leading to underutilization of the bandwidth..

Measures implemented in modern TCP stacks to overcome this limitation:

TCP Window Scaling Option (RFC 1323):

- This is the standard solution that increases the maximum window size beyond the 2^16 = 65,535 bytes limit.
- It introduces a **scaling factor** to the 16-bit window field in the TCP header, effectively allowing the window size to be up to 2^30~1GB. This allows the CWND to easily accommodate the required BDP of 380 kB.

Large Initial Window (LIW):

- While not directly related to the *maximum* size, it helps achieve the large required window size faster by increasing the initial window size from the historical 1 or 2 segments to 10 or more segments, improving the connection's slow start phase.

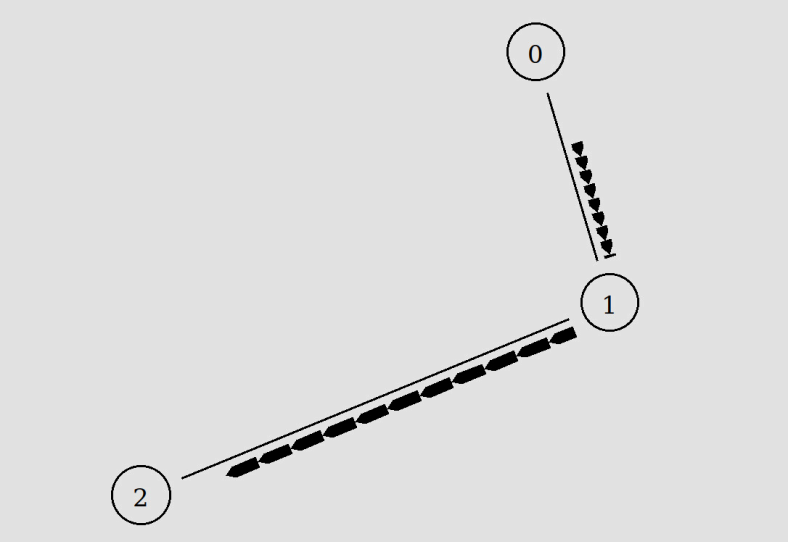
TCP Extensions for High Performance (Alternative TCP Variants):

- For extremely high BDP networks, specialized TCP congestion control algorithms like TCP Hybla or Compound TCP are used. These algorithms are specifically designed

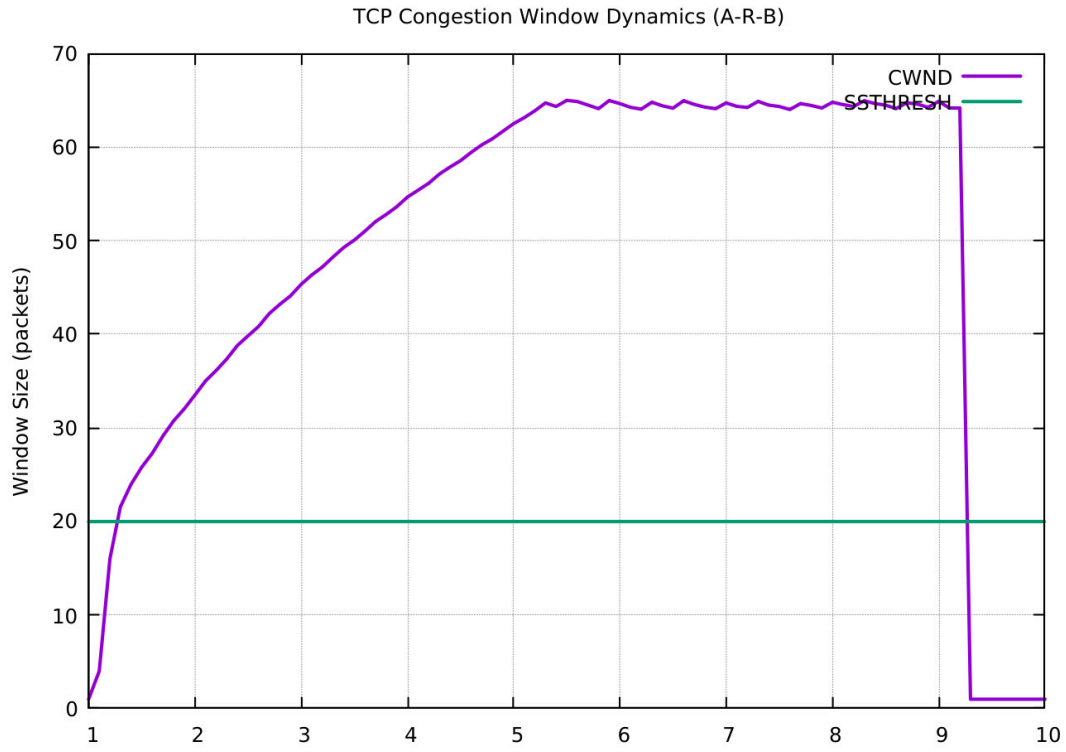
to more quickly and effectively open up the congestion window to match the large BDP of "long-fat networks."

Exercise 4

After building the TCL file and running it, we get the following animation of the nodes 0 and 2 communicating through the router:



And with the following plot:



The plot shows the classic behavior of TCP congestion control, driven by the 20-packet queue limit at the bottleneck router. In it we see:

1. **Initial Congestion Detection:** The TCP flow correctly detects congestion caused by the limited router buffer (20 packets) and resets the **Slow Start Threshold (SSTHRESH)** to a value near that bottleneck.

2. **Conservative Probing:** The protocol then switches from exponential **Slow Start** to linear **Congestion Avoidance** to prevent further packet loss.
3. **Maximum Window Limit:** The flow is capped by the user-defined **maxcwnd=64** setting, which is the final limiting factor on the amount of data the sender is allowed to transmit before waiting for an ACK. This demonstrates that network throughput can be limited by **system configuration** (maxcwnd) as much as by physical constraints (C and RTT).