

Session 8

TCP under UDP interference: congestion window tracing

This session reproduces the classic experiment: start a TCP flow, then add UDP traffic that takes half the bottleneck, and trace how the TCP congestion window reacts.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 19 to 22 of the manual: tcp under udp interference: congestion window tracing
- Prepare the deliverable before the lab and finish it during the session
- Be ready to explain every step in the viva

Questions Covered

✕ Do not copy. Read for understanding and the viva

Question	Requirement	Status
Q19	Use the setup made in session 2 and monitor the traffic flow, plot the packets received	Complete
Q20	Start the TCP application at Time 1 second	Complete
Q21	After 20 seconds, start the UDP application at Rate1 which clogs the half of the...	Complete
Q22	Using ns-3 tracing mechanism, plot the changes in the TCP window size over the time	Complete

Preparation

✗ Do not copy. Read for understanding and the viva

- Reuse the Session 2 dumbbell. Choose the bridge (n1 to n2) capacity, for example 1 Mbit/s, so Rate1 is 500 kbit/s.
- Start applications with `app.Start(Seconds(1.0))` for TCP and `Seconds(20.0)` for UDP.
- Trace cwnd by connecting to `/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow` after the socket exists (schedule the connect at 1.001 s) and write time and cwnd to a file.

Question 19

Problem Statement

■ Write in lab record

Use the setup made in session 2 and monitor the traffic flow, plot the packets received.

Solution

■ Write in lab record

One script, `dumbbell-cwnd.cc`, answers questions 19 to 22. It rebuilds the Session 2 dumbbell (same node numbers and subnets) with the bridge lowered from 2 Mbit/s to 1 Mbit/s so that “half the bridge” is a round 500 kbit/s. The manual calls the routers n1 and n2; as in Session 2 they are n2 and n3 here so the clients keep the numbers n0 and n1.

Steps

1. Copy the listing into `scratch/dumbbell-cwnd.cc` and run `./ns3 run scratch/dumbbell-cwnd`.
2. Nodes are created in three containers, `left` (n0, n1), `routers` (n2, n3) and `right` (n4, n5). Creating `left` first matters: the cwnd trace path in question 22 names the TCP source as `/NodeList/0`.
3. Two `PointToPointHelper` objects: `access` at 10 Mbps and 1 ms for the four leaf links, `bridge` at 1 Mbps and 10 ms for n2 to n3. Five subnets 10.1.1.0/24 to 10.1.5.0/24, then `Ipv4GlobalRoutingHelper::PopulateRoutingTables()` so the routers know every subnet.

4. Monitoring: each `PacketSink` has an `Rx` trace source that fires per delivered packet.
`TraceConnectWithoutContext("Rx", MakeBoundCallback(&RxPacket, &g_tcpPackets))` binds the callback to a counter; the UDP sink gets its own counter.
5. `SamplePackets` runs once per second from 1 s, writes `time tcpPackets udpPackets` for the packets received during the last second to `packets.txt`, and re-schedules itself.
6. `gnuplot plot_packets.gp` draws both series to `packets.png` with a dashed line at 20 s.

Program

- Lab record: every tab is one file of the answer. Write all of them.

dumbbell-cwnd.cc

dumbbell-cwnd.cc

```

1  /*
2  * dumbbell-cwnd.cc  --  MCSL-223 Session 8, questions 19 to 22
3  *
4  * Purpose of the program:
5  *   Rebuilds the Session 2 dumbbell with point-to-point links.  A TCP flow
6  *   starts at 1 s (Q20).  At 20 s a UDP OnOff flow starts at Rate1, half of
7  *   the bridge capacity (Q21).  Packets received per second at both sinks
8  *   are written to packets.txt (Q19) and every change of the TCP congestion
9  *   window is written to cwnd.txt through the ns-3 tracing mechanism (Q22).
10 *
11 * Topology (Session 2 dumbbell, all links point-to-point):
12 *   n0 (TCP src) --10Mbps,1ms--\          /--10Mbps,1ms-- n4 (TC
13 *                               n2 --1Mbps,10ms-- n3
14 *   n1 (UDP src) --10Mbps,1ms--/  (the bridge)  \--10Mbps,1ms-- n5 (UDP
15 *
16 * Build and run (ns-3.36 or later):
17 *   cp dumbbell-cwnd.cc scratch/
18 *   ./ns3 run scratch/dumbbell-cwnd
19 *   gnuplot plot_packets.gp          (packets.png)
20 */
21
22 #include "ns3/applications-module.h"
23 #include "ns3/core-module.h"
24 #include "ns3/internet-module.h"
25 #include "ns3/network-module.h"
26 #include "ns3/point-to-point-module.h"
27
28 using namespace ns3;
29
30 NS_LOG_COMPONENT_DEFINE("DumbbellCwnd");
31
32 static uint32_t g_tcpPackets = 0;  // packets seen by the TCP sink so far
33 static uint32_t g_udpPackets = 0;  // packets seen by the UDP sink so far
34
35 /*
36 * RxPacket: PacketSink "Rx" trace sink.  Bound to one of the two counters
37 * above; every delivered packet adds one.
38 */
39 static void
40 RxPacket(uint32_t* counter, Ptr<const Packet> packet, const Address& from)
41 {
42     (void)packet;
43     (void)from;

```

```

44     ++(*counter);
45 }
46
47 /*
48  * SamplePackets: once per second writes "time tcpPackets udpPackets" for
49  * the packets received during the last second, then re-schedules itself.
50  */
51 static void
52 SamplePackets(Ptr<OutputStreamWrapper> stream)
53 {
54     static uint32_t lastTcp = 0;
55     static uint32_t lastUdp = 0;
56     *stream->GetStream() << Simulator::Now().GetSeconds() << " " << (g_tcpPa
57                               << " " << (g_udpPackets - lastUdp) << std::endl;
58     lastTcp = g_tcpPackets;
59     lastUdp = g_udpPackets;
60     Simulator::Schedule(Seconds(1.0), &SamplePackets, stream);
61 }
62
63 /*
64  * CwndChange: trace sink for the CongestionWindow attribute of the TCP
65  * socket. Writes "time newCwnd" (bytes) on every change.
66  */
67 static void
68 CwndChange(Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCw
69 {
70     (void)oldCwnd;
71     *stream->GetStream() << Simulator::Now().GetSeconds() << " " << newCwnd
72 }
73
74 /*
75  * TraceCwnd: connects CwndChange to the first TCP socket of node 0. The
76  * socket only exists after BulkSend starts at 1 s, so this is scheduled
77  * at 1.001 s; connecting earlier would find no socket.
78  */
79 static void
80 TraceCwnd(Ptr<OutputStreamWrapper> stream)
81 {
82     Config::ConnectWithoutContext("/NodeList/0/$ns3::TcpL4Protocol/SocketLis
83                                   MakeBoundCallback(&CwndChange, stream));
84 }
85
86 /*
87  * main: builds the dumbbell, installs TCP (1 s) and UDP (20 s) flows,

```

```
88  * starts the two tracers and prints the totals at the end.
89  */
90  int
91  main(int argc, char* argv[])
92  {
93      std::string bridgeRate = "1Mbps";
94      std::string rate1 = "500kb/s"; // half of the bridge
95      double udpStart = 20.0;
96      double simTime = 40.0;
97
98      CommandLine cmd(__FILE__);
99      cmd.AddValue("bridgeRate", "Data rate of the n2-n3 bridge", bridgeRate);
100
101      cmd.AddValue("rate1", "UDP rate from udpStart onwards (Rate1)", rate1);
102
103      cmd.AddValue("udpStart", "Time in seconds at which UDP starts", udpStart);
104
105      cmd.AddValue("simTime", "Simulation time in seconds", simTime);
106
107      cmd.Parse(argc, argv);
108
109
110      Config::SetDefault("ns3::TcpSocket::SegmentSize", UIntegerValue(1000));
111
112      // NewReno halves cwnd on loss, as in the formula sheet (ns-3.35+ default)
113
114      Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(TcpNewR
115
116
117      // ---- nodes: create left first so the TCP source is NodeList/0 -----
118
119      NodeContainer left;
120
121      left.Create(2); // n0, n1
122
123      NodeContainer routers;
124
125      routers.Create(2); // n2, n3
126
127      NodeContainer right;
128
129      right.Create(2); // n4, n5
```

```
11
6
11
7    PointToPointHelper access;
11
8    access.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
11
9    access.SetChannelAttribute("Delay", StringValue("1ms"));
12
0
12
1    PointToPointHelper bridge;
12
2    bridge.SetDeviceAttribute("DataRate", StringValue(bridgeRate));
12
3    bridge.SetChannelAttribute("Delay", StringValue("10ms"));
12
4
12
5    NetDeviceContainer d02 = access.Install(left.Get(0), routers.Get(0));
12
6    NetDeviceContainer d12 = access.Install(left.Get(1), routers.Get(0));
12
7    NetDeviceContainer d23 = bridge.Install(routers.Get(0), routers.Get(1));
12
8    NetDeviceContainer d34 = access.Install(routers.Get(1), right.Get(0));
12
9    NetDeviceContainer d35 = access.Install(routers.Get(1), right.Get(1));
13
0
13
1    InternetStackHelper stack;
13
2    stack.Install(left);
13
3    stack.Install(routers);
13
4    stack.Install(right);
13
5
13
6    Ipv4AddressHelper address;
13
7    address.SetBase("10.1.1.0", "255.255.255.0");
```

```
13
8    address.Assign(d02);
13
9    address.SetBase("10.1.2.0", "255.255.255.0");
14
0    address.Assign(d12);
14
1    address.SetBase("10.1.3.0", "255.255.255.0");
14
2    address.Assign(d23);
14
3    address.SetBase("10.1.4.0", "255.255.255.0");
14
4    Ipv4InterfaceContainer i34 = address.Assign(d34);
14
5    address.SetBase("10.1.5.0", "255.255.255.0");
14
6    Ipv4InterfaceContainer i35 = address.Assign(d35);
14
7
14
8    Ipv4GlobalRoutingHelper::PopulateRoutingTables();
14
9
15
0    // ---- Q20: TCP n0 → n4, starts at 1 s -----
15
1    uint16_t tcpPort = 8080;
15
2    PacketSinkHelper tcpSinkHelper("ns3::TcpSocketFactory",
15
3                                     InetSocketAddress(Ipv4Address::GetAny(),
15
4    ApplicationContainer tcpSink = tcpSinkHelper.Install(right.Get(0));
15
5    tcpSink.Start(Seconds(0.0));
15
6    tcpSink.Stop(Seconds(simTime));
15
7
15
8    BulkSendHelper bulk("ns3::TcpSocketFactory", InetSocketAddress(i34.GetAd
15
9    bulk.SetAttribute("MaxBytes", UintegerValue(0));
```

```

16
0   ApplicationContainer tcpSrc = bulk.Install(left.Get(0));
16
1   tcpSrc.Start(Seconds(1.0));
16
2   tcpSrc.Stop(Seconds(simTime));
16
3
16
4   // ---- Q21: UDP n1 → n5 at Rate1, starts at 20 s -----
16
5   uint16_t udpPort = 9000;
16
6   PacketSinkHelper udpSinkHelper("ns3::UdpSocketFactory",
16
7                                   InetSocketAddress(Ipv4Address::GetAny(),
16
8   ApplicationContainer udpSink = udpSinkHelper.Install(right.Get(1));
16
9   udpSink.Start(Seconds(0.0));
17
0   udpSink.Stop(Seconds(simTime));
17
1
17
2   OnOffHelper onoff("ns3::UdpSocketFactory", InetSocketAddress(i35.GetAddr
17
3   onoff.SetConstantRate(DataRate(rate1), 1000);
17
4   ApplicationContainer udpSrc = onoff.Install(left.Get(1));
17
5   udpSrc.Start(Seconds(udpStart));
17
6   udpSrc.Stop(Seconds(simTime));
17
7
17
8   // ---- Q19: packets received per second at both sinks -----
17
9   tcpSink.Get(0)→TraceConnectWithoutContext("Rx", MakeBoundCallback(&RxPa
18
0   udpSink.Get(0)→TraceConnectWithoutContext("Rx", MakeBoundCallback(&RxPa
18
1

```

```

18
2   AsciiTraceHelper ascii;
18
3   Simulator::Schedule(Seconds(1.0), &SamplePackets, ascii.CreateFileStream
18
4
18
5   // ---- Q22: cwnd trace, connected just after the socket exists -----
18
6   Simulator::Schedule(Seconds(1.001), &TraceCwnd, ascii.CreateFileStream("
18
7
18
8   Simulator::Stop(Seconds(simTime));
18
9   Simulator::Run();
19
0
19
1   // ---- report -----
19
2   Ptr<PacketSink> ts = DynamicCast<PacketSink>(tcpSink.Get(0));
19
3   Ptr<PacketSink> us = DynamicCast<PacketSink>(udpSink.Get(0));
19
4   std::cout << "TCP sink n4: " << ts->GetTotalRx() << " bytes, " << g_tcpP
19
5       << " packets over " << simTime - 1.0 << " s = "
19
6       << ts->GetTotalRx() * 8.0 / (simTime - 1.0) / 1000.0 << " kbit
19
7   std::cout << "UDP sink n5: " << us->GetTotalRx() << " bytes, " << g_udpP
19
8       << " packets over " << simTime - udpStart << " s = "
19
9       << us->GetTotalRx() * 8.0 / (simTime - udpStart) / 1000.0 << "
20
0
20
1   Simulator::Destroy();
20
2   return 0;
20
3 }

```

Diagram

```

1      n0 10.1.1.1 (TCP BulkSend, 1 s)  --10 Mbps, 1 ms--\
2                                          n2 ---- 1 Mbps, 10 ms
3      n1 10.1.2.1 (UDP OnOff, 20 s)    --10 Mbps, 1 ms--/      DropTail 100 p

```

Output

Expected output (ns-3 is not installed here; the listing was checked by reading against the ns-3.36 helper API). Packets per second at the two sinks, with the transition at 20 s:

```

1 $ ./ns3 run scratch/dumbbell-cwnd
2 TCP sink n4: 3430000 bytes, 3430 packets over 39 s = 703.59 kbit/s
3 UDP sink n5: 1238000 bytes, 1238 packets over 20 s = 495.2 kbit/s
4 $ cat packets.txt
5 1 0 0
6 2 113 0
7 3 120 0
8 4 120 0
9 ...
10 20 120 0
11 21 58 62
12 22 58 63
13 23 58 62
14 ...
15 40 58 62

```

Expected chart: the TCP line sits at 120 packets/s from 2 s to 20 s, drops to about 58 packets/s at 20 s and stays there; the UDP line is zero until 20 s and then flat at about 62 packets/s.

Explanation

Before 20 s only TCP uses the bridge. A 1000-byte segment plus 20 bytes TCP header (ns-3 adds a further 12 bytes of options), 20 bytes IP and 2 bytes PPP is 1042 bytes on the wire, and the bridge carries

$$\frac{1 \times 10^6}{1042 \times 8} \approx 120 \text{ packets/s}$$

After 20 s the UDP source adds one 1000-byte datagram every 16 ms, 62.5 packets/s or about 515 kbit/s on the wire, and it never slows down. TCP is left with about 485 kbit/s, which is 58 segments per second. The plot of packets received is therefore two flat levels with a step at 20 s: the packets that UDP takes are exactly the packets TCP loses.

Question 20

Problem Statement

■ Write in lab record

Start the TCP application at Time 1 second.

Solution

■ Write in lab record

Steps

1. `PacketSinkHelper("ns3::TcpSocketFactory", InetSocketAddress(Ipv4Address::GetAny(), 8080))` on n4, started at 0 s so it is listening before the SYN arrives.
2. `BulkSendHelper("ns3::TcpSocketFactory", InetSocketAddress(10.1.4.2, 8080))` on n0 with `MaxBytes 0` (unlimited); `tcpSrc.Start(Seconds(1.0))` and `tcpSrc.Stop(Seconds(simTime))`.
3. `Config::SetDefault("ns3::TcpSocket::SegmentSize", UIntegerValue(1000))` gives round segment sizes, and `Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(TcpNewReno::GetTypeId()))` selects NewReno so the halving in the formula sheet is what you will see (ns-3.35 and later default to Cubic).

Output

The `packets.txt` line for the first second shows the start: `1 0 0` (nothing delivered at exactly 1 s) followed by `2 113 0`. With `LogComponentEnable("PacketSink", LOG_LEVEL_INFO)` the first delivery is expected as

```
1 At time +1.03748s packet sink received 1000 bytes from 10.1.1.1 port 49153 t
```

Explanation

The three-way handshake takes one round trip. The round-trip time on the dumbbell is twice the sum of the one-way delays,

$$RTT = 2 \times (1 + 10 + 1) \text{ ms} = 24 \text{ ms}$$

plus the transmission times, so the first data segment reaches n4 at about 1.037 s. ns-3 starts with an initial window of 10 segments (10000 bytes). The bandwidth-delay product of the bridge is only

$$BDP = 10^6 \times 0.024 = 24,000 \text{ bits} = 3000 \text{ bytes}$$

three segments, so the initial window already fills the bridge and the sink sees 120 packets/s almost from the first RTT. Everything beyond three segments sits in the DropTail queue at n2.

Question 21

Problem Statement

■ Write in lab record

After 20 seconds, start the UDP application at Rate1 which clogs the half of the dumbbell bridge capacity.

Solution

■ Write in lab record

Steps

1. Bridge capacity is 1 Mbit/s, so Rate1 is 500 kbit/s: `OnOffHelper onoff("ns3::UdpSocketFactory", InetSocketAddress(10.1.5.2, 9000))` with `SetConstantRate(DataRate("500kb/s"), 1000)`.
2. Install on n1, `udpSrc.Start(Seconds(20.0))`, stop at `simTime`. The `PacketSink` for UDP on n5 starts at 0 s.
3. Both parameters are command-line options: `./ns3 run "scratch/dumbbell-cwnd --rate1=250kb/s --udpStart=15"` runs the variant the examiner may ask for.

Output

Expected totals from the run above: the UDP sink receives about 1238 of the 1250 datagrams offered in 20 s (495 kbit/s); the few missing ones were tail-dropped at n2 while the queue was

full of TCP segments. The TCP sink's rate drops from 120 to 58 packets/s at 20 s, visible in `packets.txt` and in `packets.png`.

Explanation

From the formula sheet the UDP packet interval is

$$\Delta t = \frac{8 \times 1000}{500,000} = 16 \text{ ms}$$

The queue at n2 is shared. UDP arrivals are constant; TCP keeps the queue full because its window is larger than the pipe. When the queue overflows, DropTail discards whichever packet arrives next, so both flows lose packets, but only TCP reacts to loss. The result is that UDP keeps essentially all of its 500 kbit/s and TCP is pushed down to the remainder.

Question 22

Problem Statement

■ Write in lab record

Using ns-3 tracing mechanism, plot the changes in the TCP window size over the time.

Solution

■ Write in lab record

Steps

1. The TCP socket's `CongestionWindow` is a traced value. Its config path is `/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow`: node 0, the `TcpL4Protocol` object aggregated to it, its first socket.
2. The socket is created when `BulkSend` starts at 1 s, so the connect is scheduled at 1.001 s: `Simulator::Schedule(Seconds(1.001), &TraceCwnd, stream)`. Connecting at 0 s finds no socket and `Config::ConnectWithoutContext` aborts.
3. `TraceCwnd` calls `Config::ConnectWithoutContext(path, MakeBoundCallback(&CwndChange, stream))`. `CwndChange(stream, oldCwnd, newCwnd)` writes `time newCwnd` on every change to `cwnd.txt`.

4. Plot with gnuplot, `plot "cwnd.txt" using 1:2 with steps`, or with the matplotlib script from Session 9, which also draws the 20 s marker.

Output

Expected first and later lines of `cwnd.txt` (bytes):

```
1 $ head -4 cwnd.txt
2 1.04942 11000
3 1.0495 12000
4 1.04958 13000
5 1.04966 14000
6 $ awk '$1>19.9 && $1<21.2' cwnd.txt | head -4
7 20.7311 82000
8 20.7318 41000
9 21.0965 42000
10 21.1002 43000
```

Expected shape of the plot:

Interval	cwnd behaviour	Why
1.0 to 1.9 s	10 kB rising to about 110 kB, doubling every RTT	slow start; the RTT grows as the queue at n2 fills
1.9 s	falls to about 55 kB	first overflow of the 100-packet queue, NewReno halves the window
2 to 20 s	slow straight climb from 55 kB to about 82 kB	congestion avoidance, one segment per RTT, with the RTT near 0.5 s because of the queue
20.7 s	halves to about 41 kB	UDP filled the last free slots of the queue and TCP lost a segment
21 to 40 s	sawtooth between about 25 kB and 45 kB	losses now come every few seconds because UDP keeps the queue near full

Explanation

The trace records the state variable itself, not the throughput. While $cwnd$ is above the bandwidth-delay product (3000 bytes) the bridge is full whatever the window does; what changes with $cwnd$ is how many segments wait in the queue, which is why the RTT climbs from 24 ms to almost a second before the first loss. The first peak is the queue limit: three segments in the pipe plus 100 in the queue plus the segment that overflowed, about 104 kB. NewReno sets $cwnd$ to half of that after fast retransmit, then adds one segment per RTT (formula sheet, congestion avoidance). After 20 s the queue is shared with UDP, the drop probability seen by TCP rises, and the sawtooth becomes shorter and lower: the average window, and so the TCP share of the bridge, roughly halves, which matches the 120 to 58 packets/s step in question 19.

Formula Sheet

× Do not copy. Read for understanding and the viva

Throughput

Bytes received at the sink divided by the time the flow was active, converted to bits per second:

$$\text{Throughput} = \frac{8 \times \text{bytes received}}{t_{\text{end}} - t_{\text{start}}} \text{ bit/s}$$

Bandwidth-delay product

The amount of data in flight on a link of capacity B and round-trip time RTT . A TCP window smaller than this cannot fill the link:

$$BDP = B \times RTT$$

For the Session 10 defaults, $B = 5 \text{ Mbit/s}$ and one-way delay 5 ms give $RTT = 10 \text{ ms}$ and $BDP = 5 \times 10^6 \times 0.010 = 50,000 \text{ bits} \approx 6.25 \text{ kB}$.

Transmission and propagation delay

$$t_{\text{trans}} = \frac{L}{B}, \quad t_{\text{prop}} = \frac{d}{v}, \quad t_{\text{total}} = t_{\text{trans}} + t_{\text{prop}} + t_{\text{queue}} + t_{\text{proc}}$$

where L is packet size in bits, B link rate, d distance and v signal speed (about $2 \times 10^8 \text{ m/s}$ in copper or fibre).

Constant bit rate traffic

An OnOff application with packet size L bytes and rate R bit/s sends one packet every

$$\Delta t = \frac{8L}{R} \text{ seconds}$$

TCP congestion window

Slow start doubles the window every RTT until the threshold; congestion avoidance adds one segment per RTT; a loss halves it (TCP NewReno):

$$cwnd_{ss}(n) = 2^n \text{ MSS}, \quad cwnd_{ca} \leftarrow cwnd + \frac{MSS^2}{cwnd}, \quad cwnd_{loss} \leftarrow \frac{cwnd}{2}$$

The maximum throughput of one TCP flow is bounded by

$$\text{Throughput}_{\max} = \frac{cwnd_{\max}}{RTT}$$

Packet loss

$$\text{Loss rate} = \frac{\text{packets sent} - \text{packets received}}{\text{packets sent}}$$

Viva Questions

× Do not copy. Read for understanding and the viva

- **Q:** Why is the cwnd connect scheduled at 1.001 s and not done in `main`? **A:** The socket is created when `BulkSend` starts at 1 s; before that the config path matches nothing and `Config::ConnectWithoutContext` aborts.
- **Q:** What does each part of `/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow` mean? **A:** Node 0, the `TcpL4Protocol` object aggregated to it (the dollar sign means aggregated object), its first socket, and that socket's traced value.
- **Q:** What is Rate1 here and why? **A:** 500 kbit/s: the bridge is 1 Mbit/s and the question asks for half of it.
- **Q:** Why does the TCP rate drop to about 58 packets/s and not exactly 60? **A:** UDP takes 62.5 datagrams of 1028 bytes per second, about 515 kbit/s on the wire; TCP gets the remaining 485 kbit/s, and a TCP frame is 1042 bytes.

- **Q:** Why does cwnd keep growing after 2 s when the throughput is already flat? **A:** The bridge is full once cwnd passes the bandwidth-delay product (3000 bytes); extra window only lengthens the queue and the RTT, until the queue overflows.
- **Q:** Which ns-3 object drops packets in this experiment? **A:** The `DropTailQueue` (100 packets by default) on n2's bridge device.
- **Q:** What is the difference between the `Rx` trace and `GetTotalRx()`? **A:** `Rx` is a trace source fired per packet, so you can count packets or timestamp them; `GetTotalRx()` is a cumulative byte counter you poll.
- **Q:** Why NewReno and not the default Cubic? **A:** The formula sheet describes NewReno (halving on loss, one segment per RTT); Cubic reduces to 0.7 and grows with a cubic curve, so the plot would not match the theory you are asked to explain.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Connecting the cwnd trace at time 0. The run aborts with a message that the path matched no object.
- Creating the routers or servers before the clients, so `NodeList/0` is not the TCP sender and the trace path points at a node with no TCP socket.
- Forgetting `PopulateRoutingTables()`; the SYN is dropped at n2 with no route and nothing is ever received.
- Leaving the bridge at 2 Mbit/s from Session 2 and still calling 500 kbit/s "half the bridge".
- Plotting cwnd with lines instead of steps, which draws slopes where the window actually jumped.
- Reporting the UDP delivered rate as exactly 500 kbit/s without checking; the tail drops at the shared queue make it a little lower, and the record should say so.

Session Summary

■ Write in lab record

- Source listing of `dumbbell-cwnd.cc` with the header comment and the dumbbell diagram showing link rates, delays and the two flows
- Run output with the TCP and UDP totals and the computed 120 and 58 packets/s levels

- `packets.txt` excerpt and the chart `packets.png` from `plot_packets.gp`, with the 20 s step marked
- The RTT (24 ms) and bandwidth-delay product (3000 bytes) calculations
- `cwnd.txt` excerpt and a cwnd-versus-time plot with the five intervals of the table labelled
- One paragraph on why UDP keeps its share and TCP halves

 [Edit this page](#)

[< Previous](#)
Session 7

Next >
[Session 9](#)