

Session 7

Mixed TCP and UDP pairs

Running TCP and UDP side by side shows the central fairness problem of the Internet: UDP does not back off, so it takes whatever share it wants while TCP yields.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 16 to 18 of the manual: mixed tcp and udp pairs
- 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
Q16	Setup 4 nodes, two TCP client and server pair and two UDP client and server pair	Complete
Q17	Send packets to respective clients from both the servers	Complete
Q18	Monitor the traffic for both the pair and plot the no. of bytes received	Complete

Preparation

✕ Do not copy. Read for understanding and the viva

- Four nodes, two application pairs, distinct ports; keep both pairs on the same bottleneck so they compete.

- Log bytes received per interval for each sink separately (two trace files) and plot both series on one chart.
- Note in the record which pair suffers when the link is saturated and why.

Question 16

Problem Statement

■ Write in lab record

Setup 4 nodes, two TCP client and server pair and two UDP client and server pair.

Solution

■ Write in lab record

One script, `four-nodes.cc`, covers questions 16 to 18. The four nodes sit on one CSMA bus of 2 Mbit/s so every flow competes for the same medium. Each node sends exactly one flow and receives exactly one flow.

Steps

1. Copy the listing into `scratch/four-nodes.cc`.
2. `NodeContainer nodes; nodes.Create(4); CsmaHelper` with `DataRate` 2 Mbps and `Delay` 6560 ns (the LAN delay used in the ns-3 tutorial); `csma.Install(nodes)` puts all four on one channel.
3. `InternetStackHelper` on all nodes, `Ipv4AddressHelper` 10.1.1.0/24, so n0 to n3 get 10.1.1.1 to 10.1.1.4.
4. The four pairs are held in a small table (sender, receiver, port, protocol, name). A loop installs, for every row, a `PacketSinkHelper` on the receiver and either a `BulkSendHelper` (TCP) or an `OnOffHelper` at 500 kb/s with 1000-byte packets (UDP) on the sender. The socket factory string, `ns3::TcpSocketFactory` or `ns3::UdpSocketFactory`, is the only thing that differs between a TCP pair and a UDP pair.
5. Build and run: `./ns3 run scratch/four-nodes`.

Pair	Sender	Receiver	Port	Source application
TCP1	n0 (10.1.1.1)	n2 (10.1.1.3)	8080	BulkSend, unlimited bytes
TCP2	n1 (10.1.1.2)	n3 (10.1.1.4)	8081	BulkSend, unlimited bytes
UDP1	n2 (10.1.1.3)	n0 (10.1.1.1)	9000	OnOff CBR 500 kb/s, 1000 bytes
UDP2	n3 (10.1.1.4)	n1 (10.1.1.2)	9001	OnOff CBR 500 kb/s, 1000 bytes

Program

four-nodes.cc

```

1  /*
2  * four-nodes.cc  --  MCSL-223 Session 7, questions 16 to 18
3  *
4  * Purpose of the program:
5  *   Four nodes share one 2 Mbit/s CSMA bus.  Two TCP pairs and two UDP pair
6  *   run at the same time so that all four flows compete for the bus.  Every
7  *   0.5 s the cumulative bytes received at each of the four sinks is writte
8  *   to its own trace file (tcp1-bytes.txt, tcp2-bytes.txt, udp1-bytes.txt,
9  *   udp2-bytes.txt); plot_bytes.gp draws the four series on one chart.
10 *
11 * Topology (CSMA bus, 2 Mbit/s, 6560 ns):
12 *      n0 ----- n1 ----- n2 ----- n3
13 *      TCP1  n0 → n2  port 8080  (BulkSend → PacketSink)
14 *      TCP2  n1 → n3  port 8081  (BulkSend → PacketSink)
15 *      UDP1  n2 → n0  port 9000  (OnOff 500 kb/s → PacketSink)
16 *      UDP2  n3 → n1  port 9001  (OnOff 500 kb/s → PacketSink)
17 *   Each node sends exactly one flow, so the bus is the only shared resourc
18 *
19 * Build and run (ns-3.36 or later):
20 *   cp four-nodes.cc scratch/
21 *   ./ns3 run scratch/four-nodes
22 *   gnuplot plot_bytes.gp          (produces bytes.png)
23 */
24
25 #include "ns3/applications-module.h"
26 #include "ns3/core-module.h"
27 #include "ns3/csma-module.h"
28 #include "ns3/internet-module.h"
29 #include "ns3/network-module.h"
30
31 #include <vector>
32
33 using namespace ns3;
34
35 NS_LOG_COMPONENT_DEFINE("FourNodes");
36
37 /*
38 * SampleBytes: writes "time bytes" for every sink into its own stream and
39 * re-schedules itself 0.5 s later.  GetTotalRx() is the cumulative byte
40 * count that PacketSink keeps for us.
41 */
42 static void
43 SampleBytes(ApplicationContainer sinks, std::vector<Ptr<OutputStreamWrapper>

```

```

44 {
45     for (uint32_t i = 0; i < sinks.GetN(); ++i)
46     {
47         Ptr<PacketSink> sink = DynamicCast<PacketSink>(sinks.Get(i));
48         *streams[i]→GetStream() << Simulator::Now().GetSeconds() << " " <<
49             << std::endl;
50     }
51     Simulator::Schedule(Seconds(0.5), &SampleBytes, sinks, streams);
52 }
53
54 /*
55  * main: builds the bus, installs the four sender/sink pairs, starts the
56  * sampler, runs for 10 s and prints bytes and throughput per sink.
57  */
58 int
59 main(int argc, char* argv[])
60 {
61     std::string busRate = "2Mbps";
62     std::string udpRate = "500kb/s";
63     double simTime = 10.0;
64
65     CommandLine cmd(__FILE__);
66     cmd.AddValue("busRate", "CSMA bus data rate", busRate);
67     cmd.AddValue("udpRate", "Rate of each UDP OnOff source", udpRate);
68     cmd.AddValue("simTime", "Simulation time in seconds", simTime);
69     cmd.Parse(argc, argv);
70
71     // ---- Q16: four nodes on one CSMA bus -----
72     NodeContainer nodes;
73     nodes.Create(4);
74
75     CsmaHelper csma;
76     csma.SetChannelAttribute("DataRate", StringValue(busRate));
77     csma.SetChannelAttribute("Delay", TimeValue(NanoSeconds(6560)));
78     NetDeviceContainer devices = csma.Install(nodes);
79
80     InternetStackHelper stack;
81     stack.Install(nodes);
82
83     Ipv4AddressHelper address;
84     address.SetBase("10.1.1.0", "255.255.255.0");
85     Ipv4InterfaceContainer ifaces = address.Assign(devices);
86
87     // sender index, receiver index, port, "tcp" or "udp"

```

```

88     struct Pair
89     {
90         uint32_t from;
91         uint32_t to;
92         uint16_t port;
93         bool tcp;
94         std::string name;
95     };
96     std::vector<Pair> pairs = {{0, 2, 8080, true, "tcp1"},
97                               {1, 3, 8081, true, "tcp2"},
98                               {2, 0, 9000, false, "udp1"},
99                               {3, 1, 9001, false, "udp2"}};
100
101
102     ApplicationContainer sinks;
103
104     std::vector<Ptr<OutputStreamWrapper>> streams;
105
106     AsciiTraceHelper ascii;
107
108     for (const Pair& p : pairs)
109     {
110         std::string factory = p.tcp ? "ns3::TcpSocketFactory" : "ns3::UdpSocketFactory";
111         InetSocketAddress remote(ifaces.GetAddress(p.to), p.port);
112
113         // ---- Q16 / Q17: sink on the receiver, source on the sender ----
114         PacketSinkHelper sinkHelper(factory, InetSocketAddress(Ipv4Address::Zero, remote.port));
115         ApplicationContainer sinkApp = sinkHelper.Install(nodes.Get(p.to));
116         sinkApp.Start(Seconds(0.0));
117         sinkApp.Stop(Seconds(simTime));
118         sinks.Add(sinkApp);

```

```
11
6
11
7     ApplicationContainer srcApp;
11
8     if (p.tcp)
11
9     {
12
10         BulkSendHelper bulk(factory, remote);
12
11         bulk.SetAttribute("MaxBytes", UIntegerValue(0)); // send until
12
12         srcApp = bulk.Install(nodes.Get(p.from));
12
13     }
12
14     else
12
15     {
12
16         OnOffHelper onoff(factory, remote);
12
17         onoff.SetConstantRate(DataRate(udpRate), 1000);
12
18         srcApp = onoff.Install(nodes.Get(p.from));
12
19     }
13
20     srcApp.Start(Seconds(1.0));
13
21     srcApp.Stop(Seconds(simTime));
13
22
23     // ---- Q18: one trace file per sink -----
13
24     streams.push_back(ascii.CreateFileStream(p.name + "-bytes.txt"));
13
25 }
13
26
13
27 Simulator::Schedule(Seconds(0.0), &SampleBytes, sinks, streams);
```

```
13
8
13
9     Simulator::Stop(Seconds(simTime));
14
10     Simulator::Run();
14
11
14
12     // ---- report -----
14
13     double active = simTime - 1.0;
14
14     for (uint32_t i = 0; i < pairs.size(); ++i)
14
15     {
14
16         Ptr<PacketSink> sink = DynamicCast<PacketSink>(sinks.Get(i));
14
17         std::cout << pairs[i].name << " n" << pairs[i].from << "→n" << pairs[i].port << " : " << sink->GetTotalRx() << " bytes" << sink->GetTotalRx() * 8.0 / active / 1000.0 << " kbit/s"
14
18         << pairs[i].port << " : " << sink->GetTotalRx() << " bytes"
14
19         << sink->GetTotalRx() * 8.0 / active / 1000.0 << " kbit/s"
15
20     }
15
21
16
22     Simulator::Destroy();
15
23     return 0;
15
24 }
```

Diagram

```

1      n0 10.1.1.1      n1 10.1.1.2      n2 10.1.1.3      n3 10.1.1.4
2      |                |                |                |
3      ===+=====+=====+=====+===== CSMA bus, 2
4      TCP1: n0 -----> n2 :8080
5      TCP2:      n1 -----> n3 :8081
6      UDP1: n0 :9000 <----- n2
7      UDP2:      n1 :9001 <----- n3

```

Output

Expected output (ns-3 is not installed here; the listing was checked by reading against the ns-3.36 helper API). The TCP figures depend on how the two TCP flows share what the UDP flows leave; treat them as approximate:

```

1 $ ./ns3 run scratch/four-nodes
2 tcp1 n0→n2 port 8080 : 428000 bytes, 380.444 kbit/s
3 tcp2 n1→n3 port 8081 : 421000 bytes, 374.222 kbit/s
4 udp1 n2→n0 port 9000 : 562000 bytes, 499.556 kbit/s
5 udp2 n3→n1 port 9001 : 562000 bytes, 499.556 kbit/s

```

Explanation

The bus is the bottleneck. The two UDP sources together offer 1 Mbit/s of payload, about 1.06 Mbit/s on the wire with UDP, IP and Ethernet headers, and they keep sending at that rate no matter what. The two TCP flows share what is left, roughly 0.9 Mbit/s, and their ACK frames also take bus time. That is why each TCP flow ends near 380 kbit/s while each UDP flow gets its full 500 kbit/s. Both UDP flows deliver 562 packets of 1000 bytes: one packet every 16 ms (from the formula sheet, 8 times 1000 over 500000) for 9 s.

Question 17

Problem Statement

■ Write in lab record

Send packets to respective clients from both the servers.

Solution

■ Write in lab record

The manual calls the node that pushes data the server. In the script that is the `BulkSendHelper` node for TCP and the `OnOffHelper` node for UDP; the `PacketSink` node is the client that receives.

Steps

1. Sinks start at 0 s so they are listening before any data arrives; every source starts at 1 s and stops at 10 s (`srcApp.Start(Seconds(1.0))`).
2. `BulkSendHelper` with `MaxBytes 0` keeps the TCP socket's send buffer full until the stop time, so TCP sends as fast as its congestion window allows.
3. `OnOffHelper::SetConstantRate(DataRate("500kb/s"), 1000)` makes each UDP source send a 1000-byte datagram every 16 ms regardless of what the bus is doing.
4. Confirm the traffic in a pcap if the examiner asks: add `csma.EnablePcapAll("four-nodes")` before `Simulator::Run()` and open `four-nodes-2-0.pcap` in Wireshark; you will see TCP segments to port 8080 and UDP datagrams from port 9000 on the same interface.

Output

With `LogComponentEnable("PacketSink", LOG_LEVEL_INFO)` added at the top of `main`, every delivery prints a line like these (expected format, first four shown):

```
1 At time +1.00419s packet sink received 1000 bytes from 10.1.1.3 port 49153 t
2 At time +1.00838s packet sink received 1000 bytes from 10.1.1.4 port 49153 t
3 At time +1.01262s packet sink received 536 bytes from 10.1.1.1 port 49153 to
4 At time +1.01566s packet sink received 536 bytes from 10.1.1.2 port 49153 to
```

Explanation

The UDP sinks see 1000-byte datagrams at a steady 16 ms spacing from the first second onwards. The TCP sinks see 536-byte segments (the ns-3 default `SegmentSize`) arriving in bursts that grow as the congestion window opens, then settle to whatever rate the bus leaves free. Source port 49153 is the first ephemeral port ns-3 hands out on each node, which is why all four flows show it.

Question 18

Problem Statement

■ Write in lab record

Monitor the traffic for both the pair and plot the no. of bytes received.

Solution

■ Write in lab record

Steps

1. `PacketSink::GetTotalRx()` already keeps the cumulative byte count for each sink, so monitoring means sampling it. `SampleBytes` is scheduled at 0 s and re-schedules itself every 0.5 s; on each call it writes `time bytes` for every sink to that sink's own stream.
2. The streams come from `AsciiTraceHelper::CreateFileStream`, one per pair: `tcp1-bytes.txt`, `tcp2-bytes.txt`, `udp1-bytes.txt`, `udp2-bytes.txt`.
3. After the run, `gnuplot plot_bytes.gp` reads the four files and writes `bytes.png` with all four series on one chart.
4. Paste the chart in the record and mark on it where the UDP lines are straight (constant rate) and where the TCP lines bend.

Program

plot_bytes.gp

```
1 # plot_bytes.gp -- MCSL-223 Session 7, Q18
2 # Run after the simulation:  gnuplot plot_bytes.gp
3 # Reads the four "time bytes" files and draws them on one chart.
4
5 set terminal pngcairo size 900,540
6 set output "bytes.png"
7 set title "Cumulative bytes received at each sink (2 Mbit/s CSMA bus)"
8 set xlabel "Time (s)"
9 set ylabel "Bytes received"
10 set key left top
11 set grid
12
13 plot "tcp1-bytes.txt" using 1:2 with lines lw 2 title "TCP1 n0 to n2 (8080)"
14      "tcp2-bytes.txt" using 1:2 with lines lw 2 title "TCP2 n1 to n3 (8081)"
15      "udp1-bytes.txt" using 1:2 with lines lw 2 title "UDP1 n2 to n0 (9000)"
16      "udp2-bytes.txt" using 1:2 with lines lw 2 title "UDP2 n3 to n1 (9001)"
```

Output

Expected first lines of two of the trace files:

```
1 $ head -5 udp1-bytes.txt
2 0 0
3 0.5 0
4 1 0
5 1.5 31000
6 2 62000
7 $ head -5 tcp1-bytes.txt
8 0 0
9 0.5 0
10 1 0
11 1.5 14472
12 2 38056
```

Expected chart: four cumulative lines starting at 1 s. The two UDP lines are straight with slope 62.5 kB/s and lie on top of each other. The two TCP lines start below them, curve upward during slow start and then run straight with a smaller slope, about 47 kB/s, ending near 425 kB at 10 s.

Explanation

The slope of each line is that flow's throughput (the formula-sheet throughput is exactly rise over run on this plot, times 8). UDP is straight because the OnOff source never changes its rate and the bus still has room for it. TCP bends because its rate is set by the congestion window: it grows until a queue overflows at the sending device, halves, and grows again, so the line is a sequence of slightly different slopes averaging to the share the bus leaves.

If you raise `--udpRate=1Mb/s`, both UDP sources together offer 2 Mbit/s, the whole bus. The UDP lines stay nearly straight while the TCP lines almost flatten: TCP interprets every lost segment as congestion and slows down; UDP has no such rule. Write that observation in the record as the answer to "which pair suffers and why".

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 a CSMA bus and not four point-to-point links? **A:** With only four nodes, a shared bus is the simplest way to give all four flows one common bottleneck; separate point-to-point links would never compete.
- **Q:** What is the only difference between installing a TCP pair and a UDP pair? **A:** The socket factory string given to `PacketSinkHelper` and to the source helper: `ns3::TcpSocketFactory` against `ns3::UdpSocketFactory`.
- **Q:** Why does `BulkSendHelper` take `MaxBytes` 0? **A:** Zero means unlimited; the application keeps the socket buffer full until its stop time.
- **Q:** Which application counts the bytes you plot? **A:** `PacketSink`; `GetTotalRx()` returns the cumulative bytes delivered to it.
- **Q:** Why are the UDP lines straight and the TCP lines bent? **A:** UDP sends at a fixed rate with no feedback; TCP's rate follows its congestion window, which grows and halves with losses.

- **Q:** Which flow suffers when the bus is saturated? **A:** TCP. It treats every loss as congestion and slows down; UDP keeps sending, so it keeps most of its share.
- **Q:** What is the slope of a line in the bytes-versus-time plot? **A:** The throughput in bytes per second; multiply by 8 for bit/s.
- **Q:** Why sample every 0.5 s instead of logging every packet? **A:** A sampled cumulative count gives a smooth, small file that plots directly; per-packet logging gives thousands of lines that still need summing.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Using a CSMA `DeLay` of milliseconds. With a propagation delay comparable to a frame time the bus spends most of its time in collisions and backoff and the throughput figures make no sense; use the tutorial's 6560 ns.
- Starting the sources before the sinks. A TCP SYN sent to a port with no listener is refused and the flow never starts.
- Writing all four sinks to one stream but reading it as one series; either keep one file per sink, as here, or one file with five columns.
- Scheduling `SampleBytes` once and forgetting to re-schedule it inside the function, which gives a single line per file.
- Plotting the trace file and calling the y axis throughput. The file holds cumulative bytes; throughput is its slope.
- Putting two sources on the same node without noting that they share that node's device queue; the record should explain where the competition happens.

Session Summary

■ Write in lab record

- Source listing of `four-nodes.cc` with the header comment and the bus topology diagram with the four pairs marked
- The pair table (sender, receiver, port, protocol)
- Run output with bytes and throughput for all four sinks
- Four `PacketSink` log lines showing both a UDP and a TCP delivery
- The first lines of `udp1-bytes.txt` and `tcp1-bytes.txt`

- `plot_bytes.gp` and the chart `bytes.png` with the four series, annotated with which pair suffers when the bus saturates and why

 [Edit this page](#)

[<](#) Previous
Session 6

Next [>](#)
Session 8