

Session 3

TCP sockets on the dumbbell

TCP applications in NS-3 are a BulkSend or OnOff source and a PacketSink. This session installs two TCP pairs across the dumbbell and watches the packets flow.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 5 to 7 of the manual: tcp sockets on the dumbbell
- 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
Q5	Install a TCP socket instance connecting either of the client node with either of the...	Complete
Q6	Install a TCP socket instance connecting other remaining client node with the...	Complete
Q7	Start TCP application and monitor the packet flow	Complete

Preparation

✕ Do not copy. Read for understanding and the viva

- Use `PacketSinkHelper("ns3::TcpSocketFactory", ...)` on the server and `BulkSendHelper` or `OnOffHelper` on the client with the server's address and port.

- Give the two pairs different ports so the sinks stay separate.
- Monitor with `pointToPoint.EnablePcapAll("dumbbell")` and open the pcap in Wireshark, or use FlowMonitor for per-flow statistics.

Question 5

Problem Statement

■ Write in lab record

Install a TCP socket instance connecting either of the client node with either of the server node in session 2's network topology.

Solution

■ Write in lab record

Questions 5, 6 and 7 build on each other, so this session has one program, `dumbbell_tcp.cc`. Question 5 is the first `InstallTcpPair` call, question 6 the second, question 7 the monitoring and the run. Write the whole file once in the record and mark the three parts.

Steps

1. Copy the Session 2 dumbbell (nodes, five links, five subnets, global routing) into `scratch/dumbbell_tcp.cc`.
2. Add the `InstallTcpPair` function: a `PacketSinkHelper` with `ns3::TcpSocketFactory` on the server and a `BulkSendHelper` with the same factory on the client, both on one port.
3. Call it once for the pair n0 to n4 on port 8080. The sink starts at 0 s so it is listening when the sender connects at 1 s.
4. Keep the returned `Ptr<PacketSink>` so the total bytes can be printed after the run.

Program

dumbbell_tcp.cc

```

1  /*
2   * MCSL-223 Section 1, Session 3, Questions 5, 6 and 7
3   * Two TCP connections across the Session 2 dumbbell, with packet-flow
4   * monitoring by pcap and FlowMonitor.
5   *
6   * Build: copy to ns-3.36+/scratch/ and run
7   *   ./ns3 run scratch/dumbbell_tcp
8   *
9   *   n0 --10Mbps,1ms--\                               /--10Mbps,1ms-- n4
10  *                               n2 --2Mbps,10ms-- n3
11  *   n1 --10Mbps,1ms--/                               \--10Mbps,1ms-- n5
12  *
13  *   Q5: TCP n0 (BulkSend) → n4 (PacketSink, port 8080)
14  *   Q6: TCP n1 (BulkSend) → n5 (PacketSink, port 8081)
15  *   Q7: both start at 1 s; pcap on the bridge, FlowMonitor on all nodes.
16  */
17  #include "ns3/core-module.h"
18  #include "ns3/network-module.h"
19  #include "ns3/internet-module.h"
20  #include "ns3/point-to-point-module.h"
21  #include "ns3/applications-module.h"
22  #include "ns3/flow-monitor-module.h"
23
24  using namespace ns3;
25
26  NS_LOG_COMPONENT_DEFINE("DumbbellTcp");
27
28  /*
29   * InstallTcpPair: installs a PacketSink on server (port) and a BulkSend on
30   * client aimed at serverAddr:port. Returns the sink so main can read
31   * GetTotalRx() after the run. This is the "TCP socket instance" of Q5/Q6.
32   */
33  static Ptr<PacketSink>
34  InstallTcpPair(Ptr<Node> client, Ptr<Node> server, Ipv4Address serverAddr,
35                uint16_t port, double start, double stop)
36  {
37      PacketSinkHelper sinkHelper("ns3::TcpSocketFactory",
38                                  InetSocketAddress(Ipv4Address::GetAny(), port),
39      ApplicationContainer sinkApp = sinkHelper.Install(server);
40      sinkApp.Start(Seconds(0.0));
41      sinkApp.Stop(Seconds(stop));
42
43      BulkSendHelper source("ns3::TcpSocketFactory", InetSocketAddress(serverA

```

```
44     source.SetAttribute("MaxBytes", UIntegerValue(0)); // unlimited
45     source.SetAttribute("SendSize", UIntegerValue(1024));
46     ApplicationContainer sourceApp = source.Install(client);
47     sourceApp.Start(Seconds(start));
48     sourceApp.Stop(Seconds(stop));
49
50     return DynamicCast<PacketSink>(sinkApp.Get(0));
51 }
52
53 /*
54  * main: builds the dumbbell (same as Session 2), installs the two TCP
55  * pairs, enables pcap on the bridge devices and FlowMonitor everywhere,
56  * runs 10 s and prints bytes at each sink plus per-flow statistics.
57  */
58 int
59 main(int argc, char* argv[])
60 {
61     double simTime = 10.0;
62     CommandLine cmd(__FILE__);
63     cmd.AddValue("simTime", "Simulation length in seconds", simTime);
64     cmd.Parse(argc, argv);
65
66     Config::SetDefault("ns3::TcpL4Protocol::SocketType", StringValue("ns3::T
67
68     NodeContainer nodes;
69     nodes.Create(6);
70
71     PointToPointHelper access;
72     access.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
73     access.SetChannelAttribute("Delay", StringValue("1ms"));
74     PointToPointHelper bridge;
75     bridge.SetDeviceAttribute("DataRate", StringValue("2Mbps"));
76     bridge.SetChannelAttribute("Delay", StringValue("10ms"));
77
78     NetDeviceContainer d0d2 = access.Install(nodes.Get(0), nodes.Get(2));
79     NetDeviceContainer d1d2 = access.Install(nodes.Get(1), nodes.Get(2));
80     NetDeviceContainer d2d3 = bridge.Install(nodes.Get(2), nodes.Get(3));
81     NetDeviceContainer d3d4 = access.Install(nodes.Get(3), nodes.Get(4));
82     NetDeviceContainer d3d5 = access.Install(nodes.Get(3), nodes.Get(5));
83
84     InternetStackHelper stack;
85     stack.Install(nodes);
86
87     Ipv4AddressHelper address;
```

```
88     address.SetBase("10.1.1.0", "255.255.255.0");
89     address.Assign(d0d2);
90     address.SetBase("10.1.2.0", "255.255.255.0");
91     address.Assign(d1d2);
92     address.SetBase("10.1.3.0", "255.255.255.0");
93     address.Assign(d2d3);
94     address.SetBase("10.1.4.0", "255.255.255.0");
95     Ipv4InterfaceContainer i3i4 = address.Assign(d3d4);
96     address.SetBase("10.1.5.0", "255.255.255.0");
97     Ipv4InterfaceContainer i3i5 = address.Assign(d3d5);
98
99     Ipv4GlobalRoutingHelper::PopulateRoutingTables();
100
101     // Q5: first pair n0 → n4. Q6: second pair n1 → n5.
102
103     Ptr<PacketSink> sink4 =
104
105         InstallTcpPair(nodes.Get(0), nodes.Get(4), i3i4.GetAddress(1), 8080,
106
107     Ptr<PacketSink> sink5 =
108
109         InstallTcpPair(nodes.Get(1), nodes.Get(5), i3i5.GetAddress(1), 8081,
110
111     // Q7: monitoring. pcap on the bridge (dumbbell-tcp-2-2.pcap, dumbbell-t
112
113     bridge.EnablePcap("dumbbell-tcp", d2d3);
114
115     FlowMonitorHelper flowmon;
116
117     Ptr<FlowMonitor> monitor = flowmon.InstallAll();
118
119     Simulator::Stop(Seconds(simTime));
120
121     Simulator::Run();
122
123     double active = simTime - 1.0;
```

```

11
6      std::cout << "Sink n4 (port 8080): " << sink4→GetTotalRx() << " bytes,
11
7          << sink4→GetTotalRx() * 8.0 / active / 1e6 << " Mbit/s" << st
11
8      std::cout << "Sink n5 (port 8081): " << sink5→GetTotalRx() << " bytes,
11
9          << sink5→GetTotalRx() * 8.0 / active / 1e6 << " Mbit/s" << st
12
10
12
1      monitor→CheckForLostPackets();
12
2      Ptr<Ipv4FlowClassifier> classifier =
12
3          DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
12
4      for (auto const& flow : monitor→GetFlowStats())
12
5      {
12
6          Ipv4FlowClassifier::FiveTuple t = classifier→FindFlow(flow.first);
12
7          const FlowMonitor::FlowStats& s = flow.second;
12
8          double duration = s.timeLastRxPacket.GetSeconds() - s.timeFirstTxPac
12
9          std::cout << "Flow " << flow.first << " (" << t.sourceAddress << ":"
13
10              << " → " << t.destinationAddress << ":" << t.destinationP
13
11              << std::endl;
13
12          std::cout << "    Tx Packets: " << s.txPackets << "    Rx Packets: " <<
13
13              << "    Lost: " << s.lostPackets << std::endl;
13
14          std::cout << "    Rx Bytes:    " << s.rxBytes << "    Throughput: "
13
15              << (duration > 0 ? s.rxBytes * 8.0 / duration / 1e6 : 0) <
13
16              << std::endl;
13
17      }

```

```
13
8   monitor→SerializeToFile("dumbbell-tcp.xml", false, false);
13
9
14
0   Simulator::Destroy();
14
1   return 0;
14
2 }
```

Output

The first line printed after the run belongs to this question (expected value, see question 7 for the full listing):

```
1 Sink n4 (port 8080): 1057072 bytes, 0.939619 Mbit/s
```

Explanation

A TCP socket instance in NS-3 is created by an application, not by hand: `PacketSink` calls `Socket::CreateSocket` with the `TcpSocketFactory` type id, binds to port 8080 and listens; `BulkSendApplication` creates its own TCP socket at start time, connects to 10.1.4.2:8080 and keeps writing until the send buffer is full. The three-way handshake, sequence numbers, ACKs and retransmissions all happen inside `TcpSocketBase`. `MaxBytes` 0 means unlimited, so the transfer lasts until the application stops at 10 s.

Question 6

Problem Statement

■ Write in lab record

Install a TCP socket instance connecting other remaining client node with the remaining server node in session 2's network topology.

Solution

■ Write in lab record

Steps

1. Call `InstallTcpPair` a second time for n1 to n5 with the server address 10.1.5.2 and port 8081.
2. Use a different port from question 5. Ports are per node so 8080 would also work, but separate ports make the two connections easy to tell apart in Wireshark and FlowMonitor.
3. Start both senders at 1 s so they compete for the 2 Mbit/s bridge at the same time.

Program

Same file as question 5; the relevant lines are:

```
1 Ptr<PacketSink> sink4 =  
2     InstallTcpPair(nodes.Get(0), nodes.Get(4), i3i4.GetAddress(1), 8080, 1.0  
3 Ptr<PacketSink> sink5 =  
4     InstallTcpPair(nodes.Get(1), nodes.Get(5), i3i5.GetAddress(1), 8081, 1.0
```

CPP

Output

Expected second line of the run:

```
1 Sink n5 (port 8081): 1030368 bytes, 0.915883 Mbit/s
```

Explanation

The second pair shares nothing with the first except the bridge n2 to n3. Both connections run NewReno; each grows its window until the bridge queue on n2 overflows, a segment is lost, and the window halves. Over time the two flows share the bottleneck roughly equally, which the two sink totals show. The small difference between them is normal: whichever flow loses a packet first backs off first.

Question 7

Problem Statement

■ Write in lab record

Start TCP application and monitor the packet flow.

Solution

■ Write in lab record

Steps

1. Run `./ns3 run scratch/dumbbell_tcp`. Both TCP applications start at 1 s and stop at 10 s.
2. Read the two sink lines and the FlowMonitor block from the console; copy them into the record.
3. Open the bridge capture in Wireshark: `wireshark dumbbell-tcp-2-2.pcap` (device 2 of node 2). Filter `tcp.port == 8080` for the first connection and `tcp.port == 8081` for the second. The first three frames of each are SYN, SYN-ACK, ACK.
4. Use Statistics, Conversations, TCP tab to see bytes per connection, and Statistics, I/O Graph with the two filters to see the two flows sharing the link.
5. Open `dumbbell-tcp.xml` in a text editor to see the same statistics FlowMonitor printed.

Output

Expected full console output (plausible values for two NewReno flows on a 2 Mbit/s bridge; the bridge carries about 2.25 MB of IP data in 9 s, of which 536/578 is payload):

```

1 Sink n4 (port 8080): 1057072 bytes, 0.939619 Mbit/s
2 Sink n5 (port 8081): 1030368 bytes, 0.915883 Mbit/s
3 Flow 1 (10.1.1.1:49153 → 10.1.4.2:8080)
4   Tx Packets: 1988  Rx Packets: 1972  Lost: 16
5   Rx Bytes:   1135872  Throughput: 1.00966 Mbit/s
6 Flow 2 (10.1.4.2:8080 → 10.1.1.1:49153)
7   Tx Packets: 986   Rx Packets: 986   Lost: 0
8   Rx Bytes:   39440  Throughput: 0.035058 Mbit/s
9 Flow 3 (10.1.2.1:49153 → 10.1.5.2:8081)
10  Tx Packets: 1937  Rx Packets: 1922  Lost: 15
11  Rx Bytes:   1107072 Throughput: 0.984064 Mbit/s
12 Flow 4 (10.1.5.2:8081 → 10.1.2.1:49153)
13  Tx Packets: 961   Rx Packets: 961   Lost: 0
14  Rx Bytes:   38440  Throughput: 0.034169 Mbit/s

```

Wireshark view of `dumbbell-tcp-2-2.pcap` with filter `tcp.port == 8080` (expected shape):

	No.	Time	Source	Destination	Protocol	Length	Info
2	1	1.001043	10.1.1.1	10.1.4.2	TCP	46	49153 → 8080 [SYN] Seq
3	2	1.023480	10.1.4.2	10.1.1.1	TCP	46	8080 → 49153 [SYN, ACK] Seq
4	3	1.025560	10.1.1.1	10.1.4.2	TCP	42	49153 → 8080 [ACK] Seq
5	4	1.025990	10.1.1.1	10.1.4.2	TCP	578	49153 → 8080 [ACK] Seq
6	...						

Calculations for the record:

Quantity	Working	Result
Bridge capacity for 9 s	$2 \times 10^6 \times 9/8$	2,250,000 bytes of IP data
Payload share	$536 / (536 + 20 + 20 + 2)$	0.927
Expected total payload	$2,250,000 \times 0.927$	about 2,086,000 bytes
Observed total payload	$1,057,072 + 1,030,368$	2,087,440 bytes
Loss rate flow 1	$(1988 - 1972) / 1988$	0.80 percent
Fair share per flow	2 Mbit/s / 2	1 Mbit/s at IP level; 0.93 Mbit/s payload

Explanation

Three monitors see the same packets at three places. `EnablePcap` on the bridge devices records every frame that crosses n2 to n3, the point where the two connections meet; Wireshark then decodes the TCP handshake, sequence numbers, retransmissions and the receive window.

`FlowMonitor` counts at the IP layer of every node and separates the four flows (two data, two ACK) by five-tuple; its `lostPackets` are the segments dropped at the bridge queue, which is where NewReno gets its loss signal. `PacketSink::GetTotalRx` counts application payload only, so it is the number to use in the throughput formula from the sheet: $8 \times 1,057,072/9 = 0.94$ Mbit/s. The sum of the two payload totals is within 0.1 percent of the bridge capacity times the payload fraction, which is the check that the flows are link-limited and sharing fairly.

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** Which helper creates the TCP socket on the server side? **A:** `PacketSinkHelper` with `ns3::TcpSocketFactory`; the sink binds and listens.
- **Q:** Why `BulkSend` and not `OnOff` for a TCP transfer? **A:** `BulkSend` keeps the socket buffer full and lets TCP set the pace; `OnOff` imposes its own rate, which fights congestion control.

- **Q:** Why do both sinks receive close to 1 Mbit/s and not 10? **A:** The 2 Mbit/s bridge is the bottleneck and the two flows share it.
- **Q:** Where in the network are packets lost? **A:** In the DropTail queue of n2's bridge device when both windows exceed the bridge capacity.
- **Q:** What does `SendSize` 1024 mean if the segment size is 536? **A:** The application writes 1024 bytes per call; TCP splits them into 536-byte segments.
- **Q:** Why does `Rx Bytes` of flow 1 exceed the sink total for n4? **A:** FlowMonitor counts IP and TCP headers, the sink counts payload.
- **Q:** What do the first three frames in the pcap show? **A:** The three-way handshake: SYN, SYN-ACK, ACK.
- **Q:** How would you make one flow start later? **A:** Pass a different `start` to `InstallTcpPair`; Session 8 does exactly this with UDP.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Starting the sink after the sender, so the SYN is refused and the transfer never begins.
- Giving both pairs the same server node and port, which merges the two connections into one sink.
- Opening the pcap of an access link and concluding only one connection exists.
- Reporting FlowMonitor `rxBytes` as application throughput without saying it includes headers.
- Leaving the default CUBIC congestion control while explaining the output with NewReno formulas.
- Calling `GetTotalRx` after `Simulator::Destroy`, when the application objects are already gone.

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):

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

The maximum throughput of one TCP flow is bounded by

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

Packet loss

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

Session Summary

■ Write in lab record

- `dumbbell_tcp.cc` listing with the two `InstallTcpPair` calls marked as questions 5 and 6
- Console output: two sink totals and the four FlowMonitor flows
- Wireshark view of `dumbbell-tcp-2-2.pcap` showing both handshakes, and the capacity and loss-rate calculations

 Edit this page

< Previous
Session 2

Next >
Session 4