

Session 5

UDP over Wi-Fi with tracing and pcap

This session sends UDP traffic across the wireless network from Session 4 and records what arrives, both as a plot of bytes over time and as a pcap trace of the receiver's Wi-Fi interface.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 10 to 12 of the manual: udp over wi-fi with tracing and pcap
- 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
Q10	Create a UDP client on a node n1 and a UDP server on a node n2	Complete
Q11	Send packets to node n2 from node n1 and plot the number of bytes received with...	Complete
Q12	Show the pcap traces at node n2's Wi-Fi interface	Complete

Preparation

✕ Do not copy. Read for understanding and the viva

- Attach a trace to the sink's `Rx` callback and accumulate bytes per second into a file; plot with gnuplot or matplotlib.

- `YansWifiPhyHelper::EnablePcap("node2", devices.Get(1))` writes the Wi-Fi pcap for node n2 only.
- In Wireshark, filter on `udp` and use Statistics, IO Graph to cross-check your plot.

Question 10

Problem Statement

■ Write in lab record

Create a UDP client on a node n1 and a UDP server on a node n2.

Solution

■ Write in lab record

Questions 10, 11 and 12 share one program, `wifi_udp_trace.cc`: question 10 is the network and the two applications, question 11 the `Rx` trace and the plot, question 12 the pcap. The code indices 0 and 1 are the manual's n1 and n2. Positions are fixed 50 m apart so the plot is repeatable; the OLSR routing from Session 4 stays in place.

Steps

1. Save the program as `scratch/wifi_udp_trace.cc` and run `./ns3 run scratch/wifi_udp_trace`.
2. Check the last two console lines: 900 packets received, 0 lost, 921600 bytes.
3. Rate check with the formula sheet: 1024 bytes every 10 ms is $R = 8L/\Delta t = 8192/0.01 = 819.2$ kbit/s, well under the 11 Mbit/s channel.
4. Try `--interval=1` (1 ms, 8.19 Mbit/s) and watch `lost` become non-zero as the channel saturates.

Program

wifi_udp_trace.cc

```
1  /*
2   * MCSL-223 Section 1, Session 5, Questions 10, 11 and 12
3   * UDP client on n1, UDP server on n2 over the Session 4 ad-hoc Wi-Fi.
4   * Writes bytes-received-versus-time to rx-bytes.dat and a pcap of n2's
5   * Wi-Fi interface.
6   *
7   * Build: copy to ns-3.36+/scratch/ and run
8   *   ./ns3 run scratch/wifi_udp_trace
9   *   gnuplot rx_bytes.plt           (plot)
10  *   wireshark wifi-udp-1-0.pcap    (n2 trace)
11  *
12  *   n1 (10.1.1.1, UdpClient) ---- 50 m ---- n2 (10.1.1.2, UdpServer port 9)
13  *   Node index 0 and 1 in the code = n1 and n2 in the manual.
14  *   Positions are fixed so the plot is repeatable; the Session 4 OLSR
15  *   setup is kept so a third node could forward if added.
16  */
17  #include "ns3/core-module.h"
18  #include "ns3/network-module.h"
19  #include "ns3/internet-module.h"
20  #include "ns3/wifi-module.h"
21  #include "ns3/mobility-module.h"
22  #include "ns3/olsr-module.h"
23  #include "ns3/applications-module.h"
24  #include <fstream>
25
26  using namespace ns3;
27
28  NS_LOG_COMPONENT_DEFINE("WifiUdpTrace");
29
30  static uint64_t g_rxBytes = 0;      // bytes received at n2 so far
31  static std::ofstream g_traceFile;  // rx-bytes.dat
32
33  /*
34   * RxTrace: called by the UdpServer "Rx" trace source for every packet
35   * that reaches the application on n2. Adds its size to the running total.
36   */
37  static void
38  RxTrace(Ptr<const Packet> packet)
39  {
40      g_rxBytes += packet->GetSize();
41  }
42
43  /*
```

```
44  * SampleRx: writes "time cumulative_bytes" once per sample interval so
45  * gnuplot can draw bytes received against time.
46  */
47  static void
48  SampleRx(double interval)
49  {
50      g_traceFile << Simulator::Now().GetSeconds() << "\t" << g_rxBytes << std
51      Simulator::Schedule(Seconds(interval), &SampleRx, interval);
52  }
53
54  /*
55  * main: builds the two-node ad-hoc Wi-Fi link (Q10), installs UdpServer on
56  * n2 and UdpClient on n1 at 1024 B every 10 ms, records bytes over time
57  * (Q11) and a pcap on n2's Wi-Fi device (Q12).
58  */
59  int
60  main(int argc, char* argv[])
61  {
62      double simTime = 10.0;
63      uint32_t packetSize = 1024;
64      double intervalMs = 10.0;      // 1024 B / 10 ms = 819.2 kbit/s
65      double sample = 0.5;          // trace sample period in seconds
66      double distance = 50.0;
67
68      CommandLine cmd(__FILE__);
69      cmd.AddValue("simTime", "Simulation length in seconds", simTime);
70      cmd.AddValue("packetSize", "UDP payload in bytes", packetSize);
71      cmd.AddValue("interval", "Client send interval in ms", intervalMs);
72      cmd.AddValue("distance", "Distance between n1 and n2 in metres", distance);
73      cmd.Parse(argc, argv);
74
75      LogComponentEnable("UdpServer", LOG_LEVEL_INFO);
76
77      // ---- Q10: two nodes on an ad-hoc Wi-Fi channel ----
78      NodeContainer nodes;
79      nodes.Create(2);
80
81      YansWifiChannelHelper channel = YansWifiChannelHelper::Default();
82      YansWifiPhyHelper phy;
83      phy.SetChannel(channel.Create());
84
85      WifiHelper wifi;
86      wifi.SetStandard(WIFI_STANDARD_80211b);
87      wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
```

```
88         "DataMode", StringValue("DsssRate11Mbps"),
89         "ControlMode", StringValue("DsssRate11Mbps")
90     WifiMacHelper mac;
91     mac.SetType("ns3::AdhocWifiMac");
92     NetDeviceContainer devices = wifi.Install(phy, mac, nodes);
93
94     // Fixed positions: n1 at the origin, n2 `distance` metres along x.
95     Ptr<ListPositionAllocator> positions = CreateObject<ListPositionAllocato
96     positions→Add(Vector(0.0, 0.0, 0.0));
97     positions→Add(Vector(distance, 0.0, 0.0));
98     MobilityHelper mobility;
99     mobility.SetPositionAllocator(positions);
100
101     mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
102
103     mobility.Install(nodes);
104
105     2
106
107     3     OlSrHelper olsr;
108
109     4     InternetStackHelper stack;
110
111     5     stack.SetRoutingHelper(olsr);
112
113     6     stack.Install(nodes);
114
115     7
116
117     8     Ipv4AddressHelper address;
118
119     9     address.SetBase("10.1.1.0", "255.255.255.0");
120
121     0     Ipv4InterfaceContainer interfaces = address.Assign(devices);
122
123     1
124
125     2     // UDP server on n2 (port 9) and UDP client on n1.
126
127     3     uint16_t port = 9;
128
129     4     UdpServerHelper server(port);
130
131     5     ApplicationContainer serverApp = server.Install(nodes.Get(1));
```

```
11
6    serverApp.Start(Seconds(0.5));
11
7    serverApp.Stop(Seconds(simTime));
11
8
11
9    UdpClientHelper client(interfaces.GetAddress(1), port);
12
10    client.SetAttribute("MaxPackets", IntegerValue(4294967295u)); // until
12
11    client.SetAttribute("Interval", TimeValue(MicroSeconds(static_cast<uint6
12
12    client.SetAttribute("PacketSize", IntegerValue(packetSize));
12
13    ApplicationContainer clientApp = client.Install(nodes.Get(0));
12
14    clientApp.Start(Seconds(1.0));
12
15    clientApp.Stop(Seconds(simTime));
12
16
17
18    // ---- Q11: bytes received versus time ----
19
20    Ptr<UdpServer> udpServer = DynamicCast<UdpServer>(serverApp.Get(0));
21
22    udpServer->TraceConnectWithoutContext("Rx", MakeCallback(&RxTrace));
23
24    g_traceFile.open("rx-bytes.dat");
25
26    g_traceFile << "# time(s)\tbytes_received_at_n2" << std::endl;
27
28    Simulator::Schedule(Seconds(0.0), &SampleRx, sample);
29
30
31
32
33    // ---- Q12: pcap on n2's Wi-Fi interface only → wifi-udp-1-0.pcap ----
34
35    phy.SetPcapDataLinkType(WifiPhyHelper::DLT_IEEE802_11_RADIO);
36
37    phy.EnablePcap("wifi-udp", devices.Get(1));
38
39
```

```

13
8     Simulator::Stop(Seconds(simTime));
13
9     Simulator::Run();
14
10
14
1     std::cout << "Packets received at n2: " << udpServer->GetReceived()
14
2         << ", lost: " << udpServer->GetLost() << std::endl;
14
3     std::cout << "Bytes received at n2 : " << g_rxBytes << " in " << simTim
14
4         << " s = " << g_rxBytes * 8.0 / (simTime - 1.0) / 1e3 << " kbi
14
5
14
6     g_traceFile.close();
14
7     Simulator::Destroy();
14
8     return 0;
14
9 }

```

Output

Expected console output. `UdpServer` logs one line per packet; the delay per packet is the Wi-Fi hop delay computed in Session 4 (about 1.3 ms for a 1024-byte frame), so time stamps are illustrative.

```

1 TraceDelay: RX 1024 bytes from 10.1.1.1 Sequence Number: 0 Uid: 6 TXtime: +1
2 TraceDelay: RX 1024 bytes from 10.1.1.1 Sequence Number: 1 Uid: 8 TXtime: +1
3 TraceDelay: RX 1024 bytes from 10.1.1.1 Sequence Number: 2 Uid: 10 TXtime: +
4 ...
5 TraceDelay: RX 1024 bytes from 10.1.1.1 Sequence Number: 899 Uid: 1804 TXtim
6 Packets received at n2: 900, lost: 0
7 Bytes received at n2 : 921600 in 9 s = 819.2 kbit/s

```

Explanation

`UdpServerHelper(9)` installs a `UdpServer` that binds UDP port 9, counts packets by the sequence number `UdpClient` puts in a 12-byte `SeqTsHeader`, and reports lost ones from gaps in the sequence. `UdpClientHelper` on `n1` sends `PacketSize` bytes every `Interval`; `MaxPackets` is set to the maximum so the `Stop` time ends the flow. Unlike the echo pair of Session 1 this is one-way traffic, which is what the plot in question 11 needs. The two nodes are one hop apart, so OLSR only fills in the direct route.

Question 11

Problem Statement

■ Write in lab record

Send packets to node `n2` from node `n1` and plot the number of bytes received with respect to time at node `n2`.

Solution

■ Write in lab record

Steps

1. In the program, `RxTrace` is connected to the server's `Rx` trace source with `TraceConnectWithoutContext`; it adds each packet size to a counter.
2. `SampleRx` runs every 0.5 s and appends `time bytes` to `rx-bytes.dat`.
3. After the run, plot with `gnuplot rx_bytes.plt`; it writes `rx-bytes.png`.
4. Paste the plot in the record with the axis labels and the two numbers it must agree with: 0 bytes until 1 s, 921600 bytes at 10 s.

Program

The trace part of `wifi_udp_trace.cc`:

```

1 static void RxTrace(Ptr<const Packet> packet) { g_rxBytes += packet->GetSize()
2
3 static void SampleRx(double interval)
4 {
5     g_traceFile << Simulator::Now().GetSeconds() << "\t" << g_rxBytes << std
6     Simulator::Schedule(Seconds(interval), &SampleRx, interval);
7 }
8
9 Ptr<UdpServer> udpServer = DynamicCast<UdpServer>(serverApp.Get(0));
10 udpServer->TraceConnectWithoutContext("Rx", MakeCallback(&RxTrace));

```

The gnuplot script:

rx_bytes.plt

```

1 # gnuplot script: bytes received at n2 against time
2 # usage: gnuplot rx_bytes.plt (reads rx-bytes.dat written by wifi_udp_trac
3 set terminal pngcairo size 800,500
4 set output "rx-bytes.png"
5 set title "UDP bytes received at n2 (Session 5)"
6 set xlabel "Time (s)"
7 set ylabel "Cumulative bytes received"
8 set grid
9 set key left top
10 plot "rx-bytes.dat" using 1:2 with linespoints lw 2 pt 7 title "bytes at n2"

```

Output

Expected `rx-bytes.dat` (every packet arrives 1.34 ms after it is sent, so the sample at each half second counts 50 packets more than the previous one; the sample at 10 s is not written because the simulator stops first):

```
1 # time(s)    bytes_received_at_n2
2 0      0
3 0.5    0
4 1      0
5 1.5    51200
6 2      102400
7 2.5    153600
8 3      204800
9 3.5    256000
10 4      307200
11 4.5    358400
12 5      409600
13 5.5    460800
14 6      512000
15 6.5    563200
16 7      614400
17 7.5    665600
18 8      716800
19 8.5    768000
20 9      819200
21 9.5    870400
```

The plot is a straight line from (1 s, 0) to (10 s, 921600) with slope $921600/9 = 102400 \text{ bytes/s} = 819.2 \text{ kbit/s}$, the offered rate. A flat section would mean the channel was busy or the nodes moved out of range.

Explanation

NS-3 tracing separates the event source from the consumer. `UdpServer` fires its `Rx` trace source for every packet it receives; the script attaches a plain C++ function to it with `TraceConnectWithoutContext` (the same mechanism `Config::ConnectWithoutContext` uses with a path string, as in Session 8 for the congestion window). Sampling on a timer instead of writing per packet keeps the file small and gives evenly spaced points, which gnuplot draws directly with `using 1:2`. The throughput formula from the sheet is the slope of this line: bytes received divided by elapsed time, times 8.

Question 12

Problem Statement

■ Write in lab record

Show the pcap traces at node n2's Wi-Fi interface.

Solution

■ Write in lab record

Steps

1. The program calls `phy.EnablePcap("wifi-udp", devices.Get(1))`, which captures only n2's Wi-Fi device and writes `wifi-udp-1-0.pcap` (node 1, device 0).
2. `SetPcapDataLinkType(WifiPhyHelper::DLT_IEEE802_11_RADIO)` adds a radiotap header so Wireshark shows the data rate and signal strength of every frame.
3. Open it: `wireshark wifi-udp-1-0.pcap`.
4. Apply the filter `udp.dstport == 9` to see only the client's datagrams; `wlan.fc.type_subtype == 0x1d` shows the 802.11 ACKs n2 sent back; `olsr` shows the routing traffic.
5. Statistics, I/O Graph with the filter `udp.dstport == 9` and Y axis set to Bytes reproduces the plot from question 11 straight from the capture.

Output

Expected Wireshark packet list for `wifi-udp-1-0.pcap` with filter `udp.dstport == 9` (900 frames displayed):

1	No.	Time	Source	Destination	Protocol	Length	Info
2	12	1.001340	10.1.1.1	10.1.1.2	UDP	1112	49153 → 9 Len=1024
3	14	1.011340	10.1.1.1	10.1.1.2	UDP	1112	49153 → 9 Len=1024
4	16	1.021340	10.1.1.1	10.1.1.2	UDP	1112	49153 → 9 Len=1024
5	...						
6	1810	9.991340	10.1.1.1	10.1.1.2	UDP	1112	49153 → 9 Len=1024

Frame length 1112 is 1024 payload + 8 UDP + 20 IP + 8 LLC/SNAP + 24 802.11 header + 4 FCS = 1088 bytes on air, plus the radiotap header NS-3 writes (24 bytes here; the exact size varies by

version). Without the filter the list also contains OLSR HELLO messages every 2 s from both nodes and the 802.11 ACK n2 returns after each data frame. Wireshark's Statistics, Capture File Properties shows about 921600 bytes of UDP payload and an average of 100 packets per second.

Explanation

`YansWifiPhyHelper::EnablePcap` hooks the phy's `MonitorSnifferRx` and `MonitorSnifferTx` trace sources on the given device only, so the file holds every frame n2's radio sent or received, including frames that were not for it. That is the wireless equivalent of a network tap and is why the manual asks for the trace at n2 rather than n1: it shows what arrived, not what was sent. The radiotap link type is worth enabling because the plain 802.11 format drops the rate and signal fields. The three views from this session agree with each other: the server's packet count, the trace file's slope and the pcap byte total are the same 921600 bytes measured at the application, the trace and the radio.

Viva Questions

× Do not copy. Read for understanding and the viva

- **Q:** What is the difference between `UdpClient` and `UdpEchoClient` ? **A:** `UdpClient` sends one way with a sequence and time stamp header; `UdpEchoClient` expects a reply.
- **Q:** How does `UdpServer` know a packet was lost? **A:** From a gap in the sequence numbers carried in the `SeqTsHeader` .
- **Q:** What is a trace source and a trace sink? **A:** The source is the event NS-3 fires, the sink is the function you attach with `TraceConnect` .
- **Q:** Why sample every 0.5 s instead of logging each packet? **A:** Fewer, evenly spaced points; the plot is the same line.
- **Q:** What does the slope of the bytes-versus-time line mean? **A:** Throughput in bytes per second; times 8 gives bit/s.
- **Q:** Why does the pcap file name contain 1-0? **A:** Node index 1 (n2), device index 0 on that node.
- **Q:** What does the radiotap header add? **A:** Per-frame data rate, channel and signal strength for Wireshark to display.
- **Q:** Why do OLSR packets appear in the trace when only UDP data was sent? **A:** OLSR broadcasts HELLO and TC messages periodically regardless of data traffic.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Connecting the `Rx` trace before the server application exists, or with a wrong `Config` path, so the counter stays at zero.
- Passing a fraction to `Milliseconds`; it truncates to an integer, so 0.5 becomes 0 ms and the client floods the channel. Use `MicroSeconds`.
- Leaving `MaxPackets` at its default of 100, so the flow stops after one second and the plot goes flat.
- Calling `EnablePcap` on `devices` (all nodes) and then opening n1's file to show n2's interface.
- Forgetting the `#` header line in the data file and getting a parse error from gnuplot on the label row.
- Reporting the pcap byte count (with headers) as the received bytes instead of the application count.

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 2×10^8 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

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

Packet loss

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

Session Summary

Write in lab record

- `wifi_udp_trace.cc` listing, the `UdpServer` log and the final packet and byte counts
- `rx-bytes.dat`, `rx_bytes.plt` and the bytes-versus-time plot with its slope
- Wireshark view of `wifi-udp-1-0.pcap` filtered on `udp.dstport == 9` and the I/O Graph

[Edit this page](#)

< Previous
Session 4

Next >
Session 6