

Session 2

Throughput versus latency and the dumbbell topology

This session measures how end-to-end throughput changes as link delay grows, then builds the dumbbell topology (two clients, a bottleneck link, two servers) used by Sessions 3, 8 and 9.

Objectives

✕ Do not copy. Read for understanding and the viva

- Complete questions 3 to 4 of the manual: throughput versus latency and the dumbbell topology
- 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
Q3	Measure the throughput (end to end) while varying latency in the network created in...	Complete
Q4	Create a simple network topology having two client node on left side and two server...	Complete

Preparation

✕ Do not copy. Read for understanding and the viva

- Vary the point-to-point Delay attribute (1 ms, 10 ms, 50 ms, 100 ms) and read bytes received from the sink or FlowMonitor; tabulate throughput against delay.

- Dumbbell: nodes n0, n1 (clients), n2 and n3 (routers), n4, n5 (servers); five point-to-point links; give each link its own subnet.
- Enable global routing with `Ipv4GlobalRoutingHelper::PopulateRoutingTables()` so packets cross the bridge.

Question 3

Problem Statement

■ Write in lab record

Measure the throughput (end to end) while varying latency in the network created in Session 1.

Solution

■ Write in lab record

Steps

1. Save the program as `scratch/throughput_vs_latency.cc`. It is the Session 1 link with a TCP `BulkSendApplication` on n0, a `PacketSink` on n1 and `FlowMonitor` on both nodes. The delay comes from the command line.
2. Run it four times, once per latency:

```
1 ./ns3 run "scratch/throughput_vs_latency --delay=1ms"
2 ./ns3 run "scratch/throughput_vs_latency --delay=10ms"
3 ./ns3 run "scratch/throughput_vs_latency --delay=50ms"
4 ./ns3 run "scratch/throughput_vs_latency --delay=100ms"
```

BASH

3. From each run copy `Rx Bytes` and `Throughput` of flow 1 (the data flow 10.1.1.1 to 10.1.1.2; flow 2 is the ACK stream) into the table.
4. Each run also writes a FlowMonitor XML file (`throughput-1ms.xml`, `throughput-10ms.xml` and so on); keep them as evidence.
5. Plot throughput against delay by hand or with gnuplot (`plot "table.dat" using 1:2 with linespoints`).

Program

throughput_vs_latency.cc

```

1  /*
2   * MCSL-223 Section 1, Session 2, Question 3
3   * End-to-end TCP throughput on the Session 1 link while the delay varies.
4   *
5   * Build: copy to ns-3.36+/scratch/ and run once per delay:
6   *   ./ns3 run "scratch/throughput_vs_latency --delay=1ms"
7   *   ./ns3 run "scratch/throughput_vs_latency --delay=10ms"
8   *   ./ns3 run "scratch/throughput_vs_latency --delay=50ms"
9   *   ./ns3 run "scratch/throughput_vs_latency --delay=100ms"
10  *
11  *   n0 (BulkSend) ----- n1 (PacketSink, port 5000)
12  *   10.1.1.1   5 Mbps, delay   10.1.1.2
13  */
14  #include "ns3/core-module.h"
15  #include "ns3/network-module.h"
16  #include "ns3/internet-module.h"
17  #include "ns3/point-to-point-module.h"
18  #include "ns3/applications-module.h"
19  #include "ns3/flow-monitor-module.h"
20
21  using namespace ns3;
22
23  NS_LOG_COMPONENT_DEFINE("ThroughputVsLatency");
24
25  /*
26   * main: builds the two-node link with the delay given on the command line,
27   * runs a TCP bulk transfer from n0 to n1 for simTime seconds and prints
28   * FlowMonitor statistics plus throughput = 8 * rxBytes / (tLastRx - tFirstT
29   */
30  int
31  main(int argc, char* argv[])
32  {
33      std::string delay = "10ms";
34      double simTime = 10.0;
35
36      CommandLine cmd(__FILE__);
37      cmd.AddValue("delay", "One-way link delay, e.g. 1ms, 10ms, 50ms, 100ms",
38      cmd.AddValue("simTime", "Simulation length in seconds", simTime);
39      cmd.Parse(argc, argv);
40
41      // NewReno so the formula sheet describes what the simulator does.
42      Config::SetDefault("ns3::TcpL4Protocol::SocketType", StringValue("ns3::T
43

```

```

44     NodeContainer nodes;
45     nodes.Create(2);
46
47     PointToPointHelper p2p;
48     p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
49     p2p.SetChannelAttribute("Delay", StringValue(delay));
50     NetDeviceContainer devices = p2p.Install(nodes);
51
52     InternetStackHelper stack;
53     stack.Install(nodes);
54
55     Ipv4AddressHelper address;
56     address.SetBase("10.1.1.0", "255.255.255.0");
57     Ipv4InterfaceContainer interfaces = address.Assign(devices);
58
59     uint16_t port = 5000;
60
61     // Sink on n1 accepts the TCP connection and counts bytes.
62     PacketSinkHelper sink("ns3::TcpSocketFactory",
63                           InetSocketAddress(Ipv4Address::GetAny(), port));
64     ApplicationContainer sinkApp = sink.Install(nodes.Get(1));
65     sinkApp.Start(Seconds(0.0));
66     sinkApp.Stop(Seconds(simTime));
67
68     // BulkSend on n0 sends as fast as TCP allows (MaxBytes 0 = unlimited).
69     BulkSendHelper source("ns3::TcpSocketFactory",
70                           InetSocketAddress(interfaces.GetAddress(1), port))
71     source.SetAttribute("MaxBytes", UIntegerValue(0));
72     ApplicationContainer sourceApp = source.Install(nodes.Get(0));
73     sourceApp.Start(Seconds(1.0));
74     sourceApp.Stop(Seconds(simTime));
75
76     FlowMonitorHelper flowmon;
77     Ptr<FlowMonitor> monitor = flowmon.InstallAll();
78
79     Simulator::Stop(Seconds(simTime));
80     Simulator::Run();
81
82     // Per-flow statistics. Flow 1 is data n0→n1, flow 2 is the ACK stream.
83     monitor->CheckForLostPackets();
84     Ptr<Ipv4FlowClassifier> classifier =
85         DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
86     std::cout << "Delay = " << delay << std::endl;
87     for (auto const& flow : monitor->GetFlowStats())

```

```

88     {
89         Ipv4FlowClassifier::FiveTuple t = classifier→FindFlow(flow.first);
90         const FlowMonitor::FlowStats& s = flow.second;
91         double duration = s.timeLastRxPacket.GetSeconds() - s.timeFirstTxPac
92         double throughput = duration > 0 ? s.rxBytes * 8.0 / duration / 1e6
93         std::cout << "Flow " << flow.first << " (" << t.sourceAddress << " -
94             << t.destinationAddress << ")" << std::endl;
95         std::cout << " Tx Packets: " << s.txPackets << std::endl;
96         std::cout << " Rx Packets: " << s.rxPackets << std::endl;
97         std::cout << " Tx Bytes:   " << s.txBytes << std::endl;
98         std::cout << " Rx Bytes:   " << s.rxBytes << std::endl;
99         std::cout << " Lost:       " << s.lostPackets << std::endl;
100
101         std::cout << " Duration:   " << duration << " s" << std::endl;
102
103         std::cout << " Mean delay: "
104
105             << (s.rxPackets ? s.delaySum.GetSeconds() / s.rxPackets *
106
107             << " ms" << std::endl;
108
109         std::cout << " Throughput: " << throughput << " Mbit/s" << std::endl
110     }
111
112     monitor→SerializeToXmlFile("throughput-" + delay + ".xml", false, false
113
114     Simulator::Destroy();
115
116     return 0;
117 }

```

Output

Expected console output for the 10 ms run (values are plausible for TCP NewReno on a 5 Mbit/s link, not from an actual run; your numbers will differ slightly):

```

1 Delay = 10ms
2 Flow 1 (10.1.1.1 → 10.1.1.2)
3   Tx Packets: 9520
4   Rx Packets: 9512
5   Tx Bytes:   5484520
6   Rx Bytes:   5480000
7   Lost:       8
8   Duration:   9.0012 s
9   Mean delay: 26.4 ms
10  Throughput: 4.87 Mbit/s
11 Flow 2 (10.1.1.2 → 10.1.1.1)
12  Tx Packets: 4760
13  Rx Packets: 4760
14  Tx Bytes:   190400
15  Rx Bytes:   190400
16  Lost:       0
17  Duration:   8.9905 s
18  Mean delay: 10.9 ms
19  Throughput: 0.17 Mbit/s

```

Table for the record (flow 1, simulation 10 s, source active from 1 s):

One-way delay	RTT	Rx Bytes (expected)	Duration	Throughput = 8 x Rx Bytes / duration
1 ms	2 ms	5,520,000	9.00 s	4.91 Mbit/s
10 ms	20 ms	5,480,000	9.00 s	4.87 Mbit/s
50 ms	100 ms	5,290,000	9.00 s	4.70 Mbit/s
100 ms	200 ms	4,980,000	9.00 s	4.42 Mbit/s

Sample calculation for the 100 ms row: $8 \times 4,980,000 / 9.00 = 4.43 \times 10^6$ bit/s.

Why the curve falls: from the formula sheet, $BDP = B \times RTT = 5 \times 10^6 \times 0.2 = 10^6$ bits = 125 kB at 100 ms. NS-3's default send and receive buffers are 128 kB, so the window can just cover the pipe and the steady state still fills the link; what is lost is the slow-start ramp. With the default initial window of 10 segments of 536 bytes, slow start needs about five RTTs (5.36, 10.7, 21.4, 42.9, 85.8, then 125 kB) to fill the pipe: 10 ms at RTT 2 ms, but one full second at RTT 200 ms. The window bound $\text{Throughput}_{\max} = cwnd_{\max} / RTT = 131072 \times 8 / 0.2 = 5.24$ Mbit/s is still above the link rate,

so the link, not the window, is the limit at every delay in the table. Run `--delay=200ms` as an extra row: the bound becomes 2.62 Mbit/s and the measured value drops below it.

Explanation

`BulkSendHelper` with `MaxBytes` 0 keeps the TCP socket's send buffer full, so the connection runs as fast as congestion control and the link allow. `PacketSinkHelper` on n1 accepts the connection and discards data while counting bytes. `FlowMonitorHelper::InstallAll` attaches probes to the IPv4 layer of every node and groups packets by the five-tuple (addresses, ports, protocol) into flows; the TCP connection therefore appears as two flows, data and ACKs. The script computes the formula-sheet throughput, $8 \times \text{rxBytes} / (t_{\text{last rx}} - t_{\text{first tx}})$, from `FlowStats`. `rxBytes` counts IP packets including headers, which is why it is a little larger than what `PacketSink::GetTotalRx` (payload only) would report. `Config::SetDefault` for `TcpNewReno` makes the run match the congestion-window formulas on the sheet instead of the CUBIC default.

Question 4

Problem Statement

■ Write in lab record

Create a simple network topology having two client node on left side and two server nodes on the right side. Both clients are connected with another node n1. Similarly, both server node connecting to node n2. Also connect node n1 and n2 thus forming a dumbbell shape topology. Use point to point link only.

Solution

■ Write in lab record

Steps

1. Number the nodes so the code index equals the label: n0, n1 clients; n2, n3 routers (the manual's n1 and n2); n4, n5 servers.
2. Save the program as `scratch/dumbbell.cc` and run `./ns3 run scratch/dumbbell`.
3. Check the address line and the routing table of n2 printed at 1 s: it must hold routes to 10.1.4.0 and 10.1.5.0 through 10.1.3.2.

4. Check that both echo clients get their reply; the reply proves the packet crossed both routers and the bridge.
5. `ls dumbbell-*.pcap` shows ten files, one per device (`EnablePcapAll` covers every point-to-point device). Open `dumbbell-2-2.pcap` , the n2 side of the bridge, and confirm both echo requests pass through it.

Program

dumbbell.cc

```
1  /*
2   * MCSL-223 Section 1, Session 2, Question 4
3   * Dumbbell topology with point-to-point links only.
4   *
5   * Build: copy to ns-3.36+/scratch/ and run
6   *   ./ns3 run scratch/dumbbell
7   *
8   *   n0 (client) --\                               /-- n4 (server)
9   *                   n2 ---- bridge ---- n3
10  *   n1 (client) --/    2 Mbps, 10 ms    \-- n5 (server)
11  *
12  * Access links 10 Mbps, 1 ms. Subnets:
13  *   n0-n2 10.1.1.0/24   n1-n2 10.1.2.0/24   n2-n3 10.1.3.0/24
14  *   n3-n4 10.1.4.0/24   n3-n5 10.1.5.0/24
15  * The manual calls the routers n1 and n2; here they are n2 and n3 so the
16  * node index matches the NodeContainer index.
17  */
18  #include "ns3/core-module.h"
19  #include "ns3/network-module.h"
20  #include "ns3/internet-module.h"
21  #include "ns3/point-to-point-module.h"
22  #include "ns3/applications-module.h"
23
24  using namespace ns3;
25
26  NS_LOG_COMPONENT_DEFINE("Dumbbell");
27
28  /*
29   * main: creates six nodes, five point-to-point links, five subnets,
30   * fills the routing tables with global routing and verifies the paths with
31   * one UDP echo from each client to its server.
32   */
33  int
34  main(int argc, char* argv[])
35  {
36      CommandLine cmd(__FILE__);
37      cmd.Parse(argc, argv);
38
39      LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
40      LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
41
42      NodeContainer nodes;
43      nodes.Create(6);
```

```
44
45 // Node pairs for the five links.
46 NodeContainer n0n2(nodes.Get(0), nodes.Get(2));
47 NodeContainer n1n2(nodes.Get(1), nodes.Get(2));
48 NodeContainer n2n3(nodes.Get(2), nodes.Get(3));
49 NodeContainer n3n4(nodes.Get(3), nodes.Get(4));
50 NodeContainer n3n5(nodes.Get(3), nodes.Get(5));
51
52 PointToPointHelper access;
53 access.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
54 access.SetChannelAttribute("Delay", StringValue("1ms"));
55
56 PointToPointHelper bridge;
57 bridge.SetDeviceAttribute("DataRate", StringValue("2Mbps"));
58 bridge.SetChannelAttribute("Delay", StringValue("10ms"));
59
60 NetDeviceContainer d0d2 = access.Install(n0n2);
61 NetDeviceContainer d1d2 = access.Install(n1n2);
62 NetDeviceContainer d2d3 = bridge.Install(n2n3);
63 NetDeviceContainer d3d4 = access.Install(n3n4);
64 NetDeviceContainer d3d5 = access.Install(n3n5);
65
66 InternetStackHelper stack;
67 stack.Install(nodes);
68
69 // One subnet per link.
70 Ipv4AddressHelper address;
71 address.SetBase("10.1.1.0", "255.255.255.0");
72 Ipv4InterfaceContainer i0i2 = address.Assign(d0d2);
73 address.SetBase("10.1.2.0", "255.255.255.0");
74 Ipv4InterfaceContainer i1i2 = address.Assign(d1d2);
75 address.SetBase("10.1.3.0", "255.255.255.0");
76 Ipv4InterfaceContainer i2i3 = address.Assign(d2d3);
77 address.SetBase("10.1.4.0", "255.255.255.0");
78 Ipv4InterfaceContainer i3i4 = address.Assign(d3d4);
79 address.SetBase("10.1.5.0", "255.255.255.0");
80 Ipv4InterfaceContainer i3i5 = address.Assign(d3d5);
81
82 // Routers n2 and n3 need routes to every subnet.
83 Ipv4GlobalRoutingHelper::PopulateRoutingTables();
84
85 std::cout << "n0 " << i0i2.GetAddress(0) << " n1 " << i1i2.GetAddress(0)
86           << " n2 " << i0i2.GetAddress(1) << "/" << i1i2.GetAddress(1)
87           << i2i3.GetAddress(0) << " n3 " << i2i3.GetAddress(1) << "/"
```

```

88         << i3i4.GetAddress(0) << "/" << i3i5.GetAddress(0) << "  n4 "
89         << i3i4.GetAddress(1) << "  n5 " << i3i5.GetAddress(1) << std:
90
91     // Echo servers on n4 and n5; one echo from n0 to n4 and from n1 to n5.
92     UdpEchoServerHelper echoServer(9);
93     ApplicationContainer servers = echoServer.Install(NodeContainer(nodes.Ge
94 servers.Start(Seconds(1.0));
95 servers.Stop(Seconds(10.0));
96
97     UdpEchoClientHelper client0(i3i4.GetAddress(1), 9);
98     client0.SetAttribute("MaxPackets", UIntegerValue(1));
99     client0.SetAttribute("PacketSize", UIntegerValue(1024));
100
101     ApplicationContainer c0 = client0.Install(nodes.Get(0));
102
103     c0.Start(Seconds(2.0));
104
105     c0.Stop(Seconds(10.0));
106
107     3
108
109     UdpEchoClientHelper client1(i3i5.GetAddress(1), 9);
110
111     client1.SetAttribute("MaxPackets", UIntegerValue(1));
112
113     client1.SetAttribute("PacketSize", UIntegerValue(1024));
114
115     ApplicationContainer c1 = client1.Install(nodes.Get(1));
116
117     c1.Start(Seconds(3.0));
118
119     c1.Stop(Seconds(10.0));
120
121     0
122
123     // Print n2's routing table at 1 s to show the global routes.
124
125     Ptr<OutputStreamWrapper> routing = Create<OutputStreamWrapper>(&std::cou
126
127     Ipv4GlobalRoutingHelper::PrintRoutingTableAt(Seconds(1.0), nodes.Get(2),
128
129     4
130
131     bridge.EnablePcapAll("dumbbell");

```

```
11
6
11
7     Simulator::Stop(Seconds(10.0));
11
8     Simulator::Run();
11
9     Simulator::Destroy();
12
0     return 0;
12
1 }
```

Output

Topology for the record:

```
1   n0 (10.1.1.1) ---10 Mbit/s, 1 ms---\                               /---10
2                                     n2 ---2 Mbit/s, 10 ms (bridge)--- n3
3   n1 (10.1.2.1) ---10 Mbit/s, 1 ms---/                               \---10
4
5   subnets: 10.1.1.0/24 (n0-n2)  10.1.2.0/24 (n1-n2)  10.1.3.0/24 (n2-n3)
6             10.1.4.0/24 (n3-n4)  10.1.5.0/24 (n3-n5)
```

Expected console output (routing-table rows may print in a different order in your version; the echo time stamps are computed from the link parameters):

```

1 n0 10.1.1.1 n1 10.1.2.1 n2 10.1.1.2/10.1.2.2/10.1.3.1 n3 10.1.3.2/10.1.4.
2 Node: 2, Time: +1s, Local time: +1s, Ipv4ListRouting table
3   Priority: 0 Protocol: ns3::Ipv4StaticRouting
4 Node: 2, Time: +1s, Local time: +1s, Ipv4StaticRouting table
5 Destination      Gateway      Genmask      Flags Metric Ref    Use Ifac
6 127.0.0.0         0.0.0.0      255.0.0.0     U        0      -      -    0
7 10.1.1.0          0.0.0.0      255.255.255.0 U        0      -      -    1
8 10.1.2.0          0.0.0.0      255.255.255.0 U        0      -      -    2
9 10.1.3.0          0.0.0.0      255.255.255.0 U        0      -      -    3
10
11   Priority: -10 Protocol: ns3::Ipv4GlobalRouting
12 Node: 2, Time: +1s, Local time: +1s, Ipv4GlobalRouting table
13 Destination      Gateway      Genmask      Flags Metric Ref    Use Ifac
14 10.1.3.2          10.1.3.2     255.255.255.255 UH       -      -      -    3
15 10.1.4.1          10.1.3.2     255.255.255.255 UGH      -      -      -    3
16 10.1.4.2          10.1.3.2     255.255.255.255 UGH      -      -      -    3
17 10.1.5.1          10.1.3.2     255.255.255.255 UGH      -      -      -    3
18 10.1.5.2          10.1.3.2     255.255.255.255 UGH      -      -      -    3
19 10.1.4.0          10.1.3.2     255.255.255.0  UG      -      -      -    3
20 10.1.5.0          10.1.3.2     255.255.255.0  UG      -      -      -    3
21
22 At time +2s client sent 1024 bytes to 10.1.4.2 port 9
23 At time +2.0179s server received 1024 bytes from 10.1.1.1 port 49153
24 At time +2.0179s server sent 1024 bytes to 10.1.1.1 port 49153
25 At time +2.03581s client received 1024 bytes from 10.1.4.2 port 9
26 At time +3s client sent 1024 bytes to 10.1.5.2 port 9
27 At time +3.0179s server received 1024 bytes from 10.1.2.1 port 49153
28 At time +3.0179s server sent 1024 bytes to 10.1.2.1 port 49153
29 At time +3.03581s client received 1024 bytes from 10.1.5.2 port 9

```

Delay calculation for one echo request (1054 bytes on the wire), hop by hop with the formula sheet:

Hop	$t_{\text{trans}} = L/B$	t_{prop}	Sum
n0 to n2 (10 Mbit/s)	0.843 ms	1 ms	1.843 ms
n2 to n3 (2 Mbit/s)	4.216 ms	10 ms	14.216 ms
n3 to n4 (10 Mbit/s)	0.843 ms	1 ms	1.843 ms
One way			17.90 ms
Round trip			35.81 ms

The log shows the request at n4 at 2.0179 s and the reply at n0 at 2.03581 s, matching the table.

Explanation

Each `PointToPointHelper::Install` call takes a pair of nodes and produces two devices and one channel, so five calls build the five links; using two helpers lets the bridge have a lower rate and longer delay than the access links. Every link needs its own subnet because IPv4 forwarding chooses the outgoing interface by matching the destination against a network prefix; putting two links in one subnet would make that choice ambiguous. n2 and n3 have three interfaces each and forward because `InternetStackHelper` enables IP forwarding on every node. Without routes they would drop packets for 10.1.4.0 and 10.1.5.0; `Ipv4GlobalRoutingHelper::PopulateRoutingTables` reads the whole topology from the simulator (link-state style, like OSPF) and writes shortest-path routes into every node before the simulation starts. The bridge at 2 Mbit/s is the bottleneck that Sessions 3, 8 and 9 congest on purpose.

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** Why does FlowMonitor show two flows for one TCP connection? **A:** Data and ACKs have swapped source and destination, so they are different five-tuples.
- **Q:** Why does throughput fall as delay rises even when the window covers the BDP? **A:** Slow start needs a fixed number of RTTs to open the window; longer RTT means more seconds at low rate.
- **Q:** State the bandwidth-delay product for 5 Mbit/s and RTT 200 ms. **A:** 10 to the 6 bits, 125 kB.

- **Q:** Why is `rxBytes` from `FlowMonitor` larger than `GetTotalRx` from the sink? **A:** `FlowMonitor` counts at the IP layer, headers included; the sink counts payload.
- **Q:** Why does the dumbbell need five subnets? **A:** Each point-to-point link is a separate IP network; forwarding matches destination prefixes to interfaces.
- **Q:** What does `PopulateRoutingTables` do? **A:** Builds a global link-state view of the topology and installs shortest-path routes in every node.
- **Q:** What is the bottleneck of the dumbbell and where would packets be dropped? **A:** The 2 Mbit/s bridge; drops happen in the transmit queue of `n2`'s bridge device.
- **Q:** What is the one-way delay of a 1024-byte echo across the dumbbell? **A:** About 17.9 ms: $0.843 + 1$ on each access link and $4.216 + 10$ on the bridge.

Common Mistakes

✗ Do not copy. Read for understanding and the viva

- Reading throughput off the ACK flow (flow 2) instead of the data flow.
- Dividing by the simulation time (10 s) instead of the flow duration from first Tx to last Rx.
- Assigning the same subnet to two links, which makes routing fail silently.
- Forgetting `PopulateRoutingTables`, so the echo client sends and nothing ever comes back.
- Building the dumbbell with `CsmaHelper`; the manual says point-to-point only.
- Mixing up the manual's router names (`n1`, `n2`) with the code indices (`n2`, `n3`) in the record without saying so.

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

- `throughput_vs_latency.cc` listing and the FlowMonitor output of the four runs (1, 10, 50, 100 ms)
- Throughput-versus-delay table with the BDP working and the slow-start explanation
- `dumbbell.cc` listing, the dumbbell diagram with five subnets, n2's routing table and the two echo exchanges with the hop-by-hop delay table

 Edit this page

< Previous
Session 1

Next >
Session 3