NS3 UPDATE 2

Saem Hasan - 1705027

Topology

4 Simulation and analysis

4.1 Throughput

The TCP versions New Reno, TCPW and TCPW BR were simulated on the wireless link using the NS-2 [Weigle, Adurthi, Jeffay et al. (2006)] simulation platform, and their changes were recorded. Fig. 2 shows the network topology. The network-related parameters are: the bandwidth of the wired link is 100 Mb/s, and the one-way transmission time is 30 ms. The bottleneck link bandwidth is 100 Mb/s, the unidirectional transmission time is 10 ms. The wireless link bandwidth is 5 Mb/s, the unidirectional transmission time is 0.01 ms, and the transmission packet size is 1000 bytes.

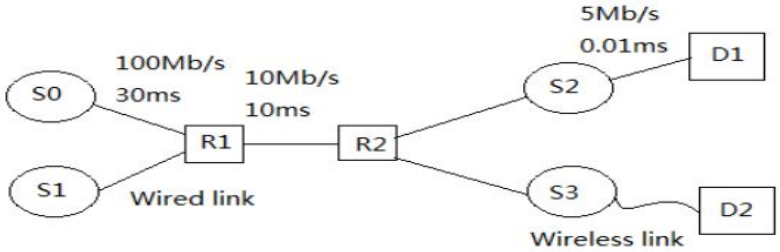
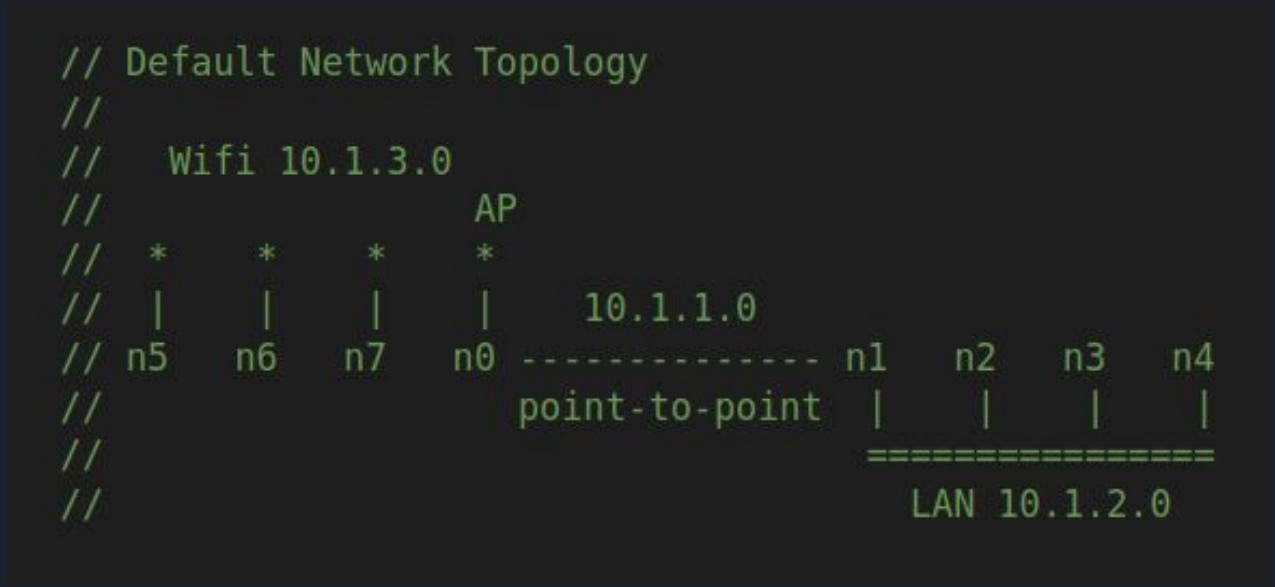


Figure 2: Network topology

Topology


// Default Network Topology
//
//   Wifi 10.1.3.0
//   AP
// *   *   *   *
// | | | | 10.1.1.0
// n5   n6   n7   n0 -------------- n1   n2   n3   n4
//   point-to-point | | | |
//   ================
//   LAN 10.1.2.0

Nodes

- Number Of Nodes : 18
- Number Of Flows : 16

```
uint32_t NUMOFNODES = 8;
uint32_t WIRED_BANDWIDTH = 100;     // Mbps
uint32_t WIRED_DELAY = 30;          // ms
uint32_t BOTTLENECKBANDWIDTH = 10; // Mbps
uint32_t BOTTLENECKDELAY = 10;      // ms
uint32_t WIRELESS_BANDWIDTH = 5;    // Mbps
uint32_t PACKET_SIZE = 1000;        // Bytes
float WIRELESS_DELAY = 0.01;        // ms
uint16_t port = 50000;
```

TCP Congestion Control Algorithm

Congestion Algorithm : TCP WestWood

```
// set TCP WESTWOOD
   Config::SetDefault("ns3::TcpL4Protocol::SocketType",
TypeIdValue(TcpWestwood::GetTypeId()));
   Config::SetDefault("ns3::TcpWestwood::ProtocolType",
EnumValue(TcpWestwood::WESTWOOD));
```

```
// set TCP WESTWOOD
Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(TcpWestwood::GetTypeId()));
Config::SetDefault("ns3::TcpWestwood::ProtocolType", EnumValue(TcpWestwood::WESTWOOD));

NodeContainer p2pNodes;
p2pNodes.Create(2);

PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute("DataRate", StringValue(std::to_string(BOTTLENECKBANDWIDTH) + "Mbps"));
pointToPoint.SetChannelAttribute("Delay", StringValue(std::to_string(BOTTLENECKDELAY) + "ms"));

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install(p2pNodes);

NodeContainer csmaNodes;
csmaNodes.Add(p2pNodes.Get(1));
csmaNodes.Create(nCsma);

CsmaHelper csma;
csma.SetChannelAttribute("DataRate", StringValue(std::to_string(WIRED_BANDWIDTH) + "Mbps"));
csma.SetChannelAttribute("Delay", StringValue(std::to_string(WIRED_DELAY) + "ms"));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install(csmaNodes);
```

```
NodeContainer wifiStaNodes;
wifiStaNodes.Create(nWifi);
NodeContainer wifiApNode = p2pNodes.Get(0);

YansWifiChannelHelper channel = YansWifiChannelHelper::Default();
YansWifiPhyHelper phy;
phy.SetChannel(channel.Create());

WifiHelper wifi;
wifi.SetRemoteStationManager("ns3::AarfWifiManager");

WifiMacHelper mac;
Ssid ssid = Ssid("ns-3-ssid");
mac.SetType("ns3::StaWifiMac",
            "Ssid", SsidValue(ssid),
            "ActiveProbing", BooleanValue(false));

NetDeviceContainer staDevices;
staDevices = wifi.Install(phy, mac, wifiStaNodes);

mac.SetType("ns3::ApWifiMac",
            "Ssid", SsidValue(ssid));

NetDeviceContainer apDevices;
apDevices = wifi.Install(phy, mac, wifiApNode);
```

```
Ptr<RateErrorModel> em = CreateObject<RateErrorModel>();
em->SetAttribute("ErrorRate", DoubleValue(.00001));


for (uint32_t i = 1; i <= nWifi; i++)
{
    Config::Set("/NodeList/" + std::to_string(i) +
    "/DeviceList/0/$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/PostReceptionErrorModel", PointerValue(em));
}

MobilityHelper mobility;

mobility.SetPositionAllocator("ns3::GridPositionAllocator",
                              "MinX", DoubleValue(0.0),
                              "MinY", DoubleValue(0.0),
                              "DeltaX", DoubleValue(5.0),
                              "DeltaY", DoubleValue(10.0),
                              "GridWidth", UintegerValue(3),
                              "LayoutType", StringValue("RowFirst"));

mobility.SetMobilityModel("ns3::RandomWalk2dMobilityModel",
                          "Bounds", RectangleValue(Rectangle(-50, 50, -50, 50)));
mobility.Install(wifiStaNodes);

mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install(wifiApNode);
```

```
for (uint16_t i = 0; i < nCsma; i++)
{
    BulkSendHelper source("ns3::TcpSocketFactory",
                          InetSocketAddress(csmaInterfaces.GetAddress(i), port));
    // Set the amount of data to send in bytes.  Zero is unlimited.
    source.SetAttribute("MaxBytes", UintegerValue(maxBytes * 10));
    ApplicationContainer sourceApps = source.Install(wifiStaNodes.Get(i));
    sourceApps.Start(Seconds(0.0));
    sourceApps.Stop(Seconds(SIMULATION_TIME));

    PacketSinkHelper sink("ns3::TcpSocketFactory",
                          InetSocketAddress(Ipv4Address::GetAny(), port));
    ApplicationContainer sinkApps = sink.Install(csmaNodes.Get(i));
    sinkApps.Start(Seconds(0.0));
    sinkApps.Stop(Seconds(SIMULATION_TIME));
}
```

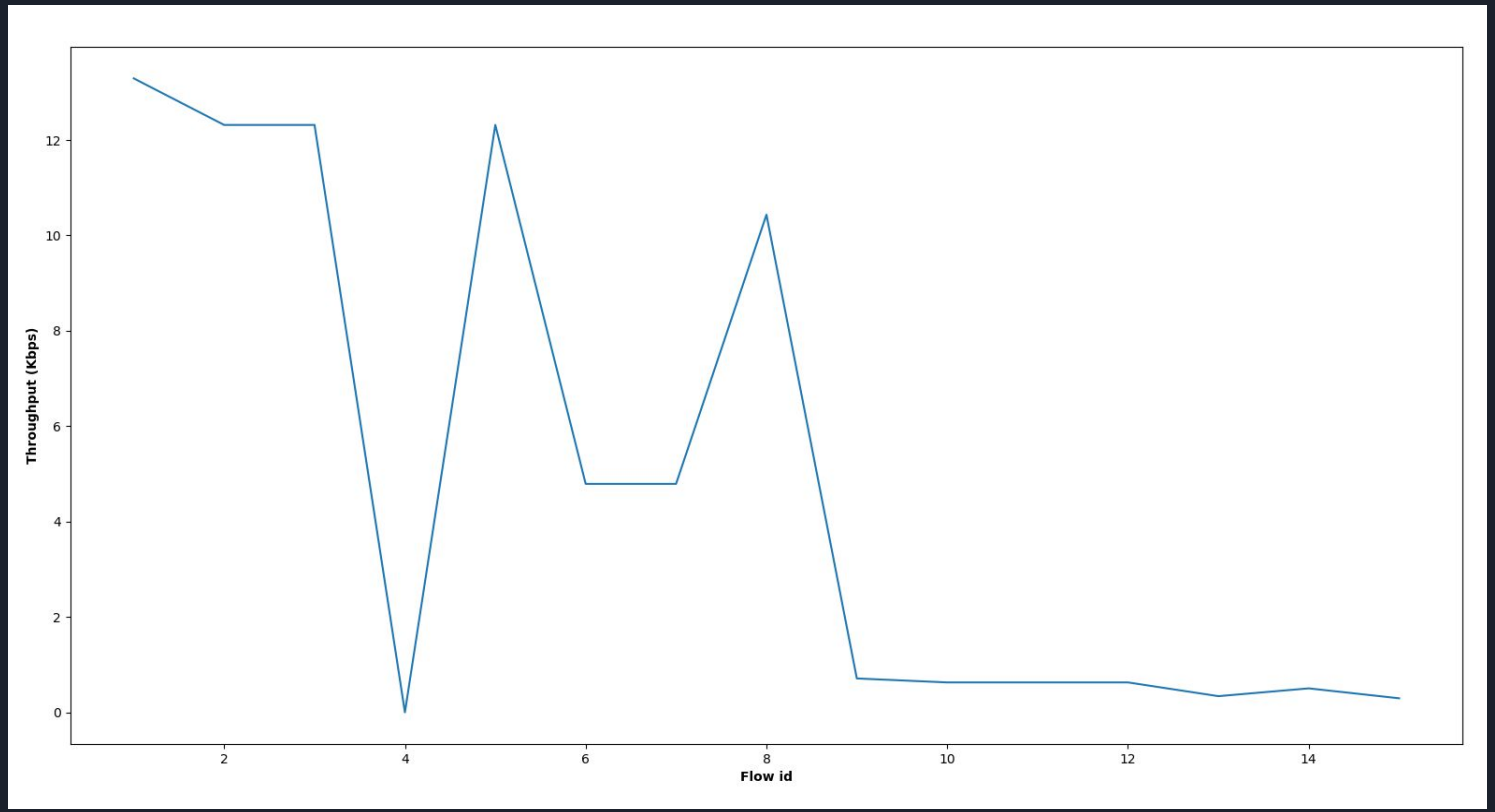
```
// Flow monitor
FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon.InstallAll();

std::ofstream thr("mytest-throughput.dat", std::ios::out);
Simulator::Schedule(Seconds(1.1 + 0.000001), &TraceThroughput, monitor);

Simulator::Stop(Seconds(SIMULATION_TIME));
Simulator::Run();

flowmon.SerializeToXmlFile("mytest.flowmonitor", true, true);

// Print per flow statistics
printFlow(&flowmon, monitor);

// Cleanup
Simulator::Destroy();
```

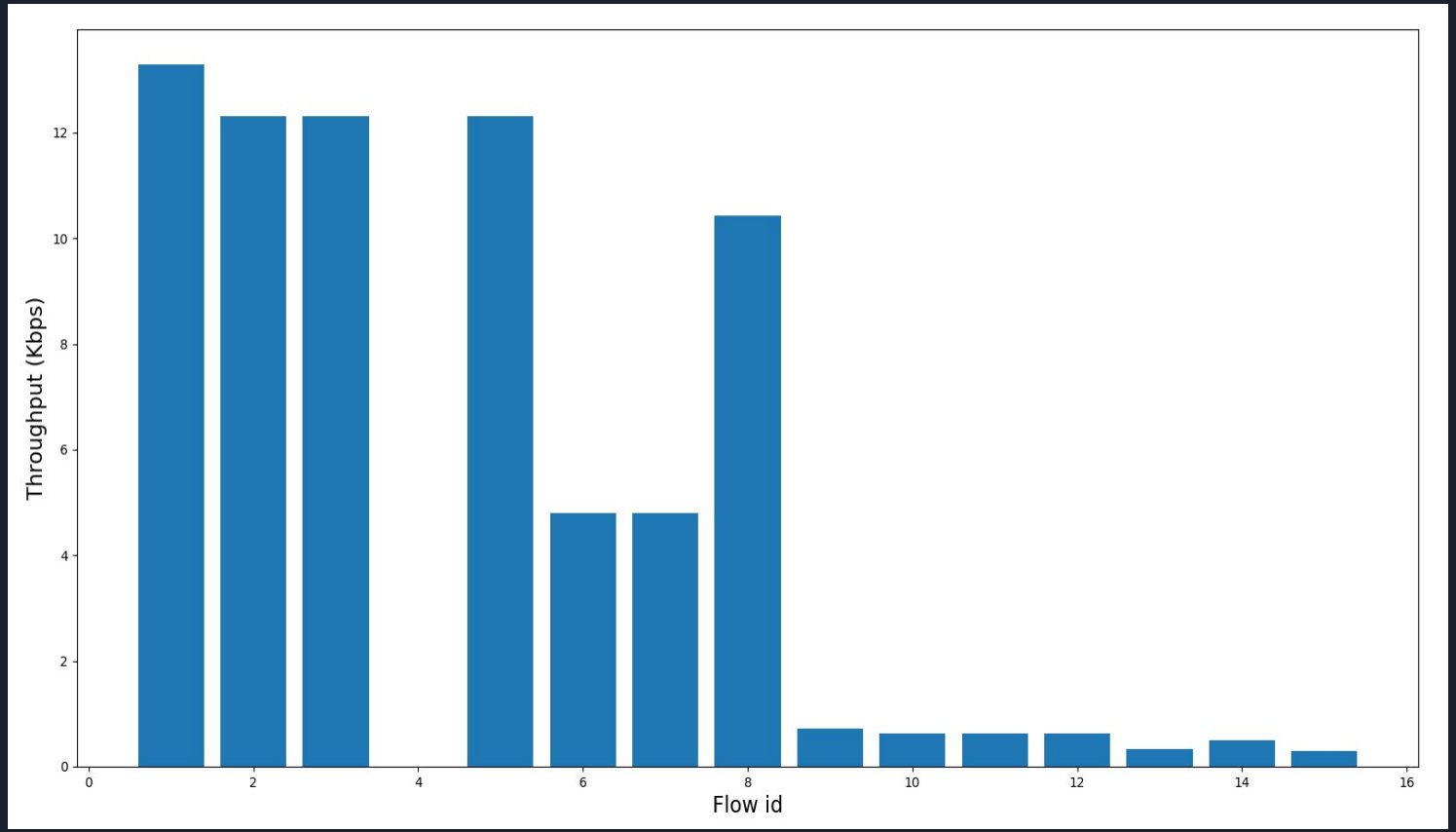
```cpp
void printFlow(FlowMonitorHelper *flowmon, Ptr<FlowMonitor> monitor)
{
    float throughPut = 0;
    uint32_t SentPackets = 0;
    uint32_t ReceivedPackets = 0;
    uint32_t LostPackets = 0;
    int j = 0;

    std::ofstream flowout("mytest-flow.dat", std::ios::out);

    monitor->CheckForLostPackets();
    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmon->GetClassifier());
    FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats();
    for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin(); i != stats.end(); ++i)
    {
        Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(i->first);
        std::cout << "Flow " << i->first << " (" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
        std::cout << "  Tx Packets: " << i->second.txPackets << "\n";
        std::cout << "  Tx Bytes:   " << i->second.txBytes << "\n";
        std::cout << "  TxOffered:  " << i->second.txBytes * 8.0 / SIMULATION_TIME / 1000 << " kbps\n";
        std::cout << "  Rx Packets: " << i->second.rxPackets << "\n";
        std::cout << "  Rx Bytes:   " << i->second.rxBytes << "\n";

        double calthroughput = i->second.rxBytes * 8.0 / SIMULATION_TIME / 1000;

        std::cout << "  Throughput: " << calthroughput << " kbps\n";

        SentPackets = SentPackets + (i->second.txPackets);
        ReceivedPackets = ReceivedPackets + (i->second.rxPackets);
        LostPackets = LostPackets + (i->second.txPackets - i->second.rxPackets);
        throughPut += calthroughput;

        j++;

        flowout << i->first << " " << calthroughput << std::endl;
    }
```

```
        double calthroughput = i->second.rxBytes * 8.0 / SIMULATION_TIME / 1000;

        std::cout << "  Throughput: " << calthroughput << " kbps\n";

        SentPackets = SentPackets + (i->second.txPackets);
        ReceivedPackets = ReceivedPackets + (i->second.rxPackets);
        LostPackets = LostPackets + (i->second.txPackets - i->second.rxPackets);
        throughPut += calthroughput;

        j++;

        flowout << i->first << " " << calthroughput << std::endl;
    }

    float avgthroughPut = throughPut / j;
    NS_LOG_UNCOND("\n\n--------Total Results of the simulation----------" << std::endl);
    NS_LOG_UNCOND("Total sent packets  : " << SentPackets);
    NS_LOG_UNCOND("Total Received Packets : " << ReceivedPackets);
    NS_LOG_UNCOND("Total Lost Packets : " << LostPackets);
    NS_LOG_UNCOND("Packet Loss ratio : " << ((LostPackets * 100.0) / SentPackets) << "%");
    NS_LOG_UNCOND("Packet delivery ratio : " << ((ReceivedPackets * 100.0) / SentPackets) << "%");
    NS_LOG_UNCOND("Average throughput : " << avgthroughPut << " kbps");
    NS_LOG_UNCOND("Total Flow id : " << j);
}
```

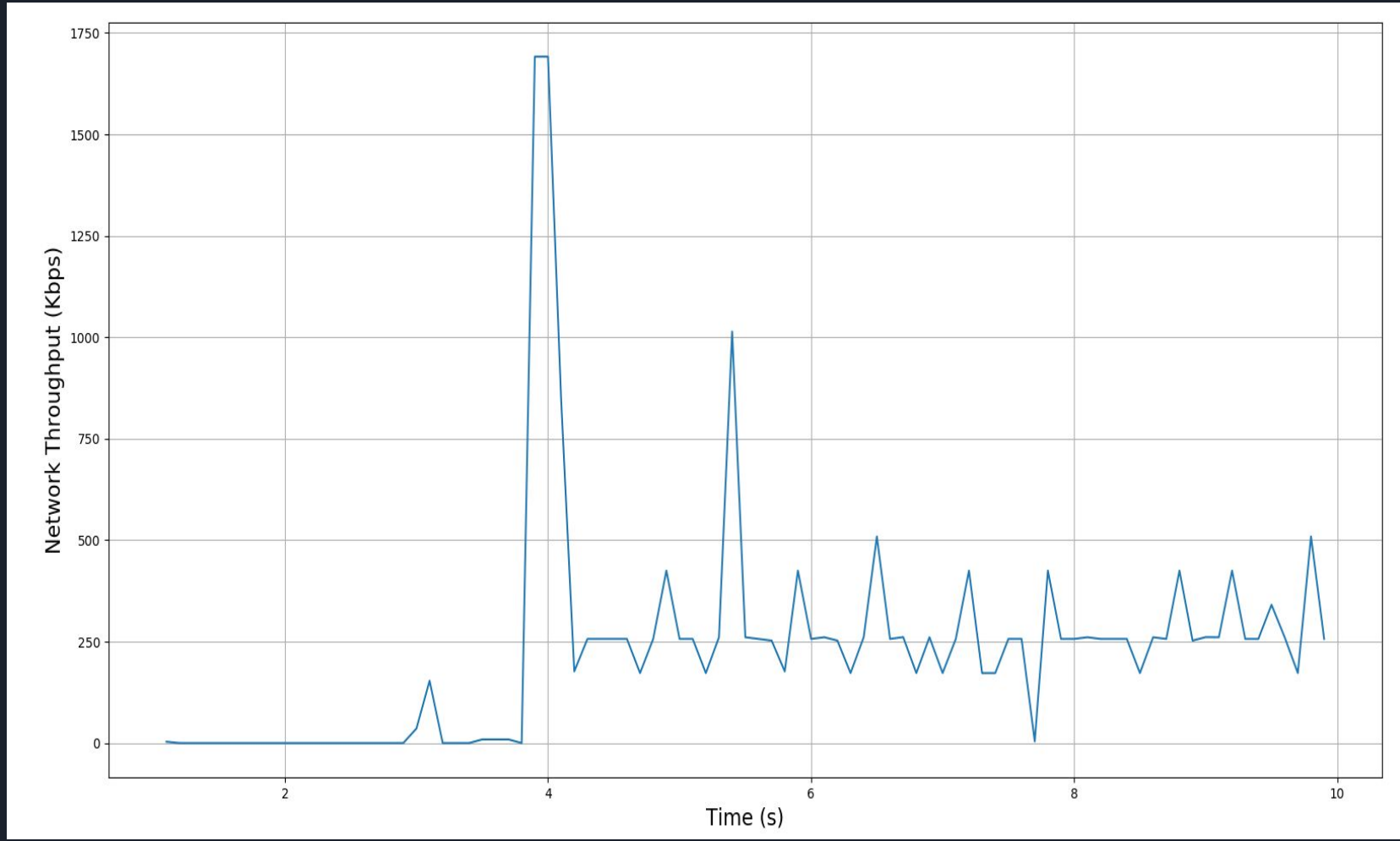
```
Flow 3 (10.1.3.3 -> 10.1.2.3)
  Tx Packets: 31
  Tx Bytes:   15724
  TxOffered:  12.5792 kbps
  Rx Packets: 28
  Rx Bytes:   15212
  Throughput: 12.1696 kbps
Flow 4 (10.1.3.4 -> 10.1.2.4)
  Tx Packets: 3
  Tx Bytes:   168
  TxOffered:  0.1344 kbps
  Rx Packets: 0
  Rx Bytes:   0
  Throughput: 0 kbps
Flow 5 (10.1.3.5 -> 10.1.2.5)
  Tx Packets: 23
  Tx Bytes:   11204
  TxOffered:  8.9632 kbps
  Rx Packets: 21
  Rx Bytes:   11096
  Throughput: 8.8768 kbps
```

```
--------Total Results of the simulation--

Total sent packets  : 264
Total Received Packets : 238
Total Lost Packets : 26
Packet Loss ratio : 9.84848%
Packet delivery ratio : 90.1515%
Average throughput : 4.34923 kbps
Total Flow id : 15
Done.
```


Throughput (Kbps)
0
2
4
6
8
10
12
2
4
6
8
10
12
14
Flow id


Throughput (Kbps)
0
2
4
6
8
10
12
0
2
4
6
8
10
12
14
16
Flow id

```
// Calculate throughput
static void
TraceThroughput(Ptr<FlowMonitor> monitor)
{
    FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats();
    int cnt = 0;
    int total = 0;
    Time curTime = Now();

    for (auto itr = stats.begin(); itr != stats.end(); itr++)
    {
        total += (itr->second.txBytes - prev[cnt]);
        prev[cnt] = itr->second.txBytes;
        cnt++;
    }
    std::ofstream thr("mytest-throughput.dat", std::ios::out | std::ios::app);

    thr << curTime.GetSeconds() << " " << 8 * (total) / (1000 * (curTime.GetSeconds() - prevTime.GetSeconds())) << std::endl;

    prevTime = curTime;

    Simulator::Schedule(Seconds(0.1), &TraceThroughput, monitor);
}
```


Network Throughput (Kbps)
1750
1500
1250
1000
750
500
250
0
2
4
6
8
10
Time (s)

Thank you.