CN-LAB ASSIGNMENT-6

Sushant Kumar-220101096
Sneh Dadhania-220123062
Priyanshu Pratyay-220101125
Simon Lalremsiama-220101095

https://github.com/LegendsDen/CN341_2024/tree/main/Assign6

Part A

```
TrafficGenerator() ->function
```

This function sets up the network traffic between nodes, where each node acts as both a sender and receiver. Here's a breakdown of its responsibilities:

- **Traffic Setup:** Each node is configured to send data to another node in a "round-robin" fashion, creating a high-traffic load.
- **On-Off Application:** An `OnOffApplication` is set up to generate UDP packets from each node, sending them to the next node in the container. The transmission rate and packet size are set to `100Mbps` and `1024` bytes, respectively.
- **Staggered Starts:** Each `OnOffApplication` is configured to start at staggered times (`0.5 * i` seconds), helping to increase potential collisions.

- **Packet Sink Application:** A `PacketSink` is installed on each node to receive incoming packets. This application runs for the full duration of the simulation (from 0.0 to 10.0 seconds).

```
void TrafficGenerator(NodeContainer &nodes, Ipv4InterfaceContainer &interfaces)
{
    uint16_t port = 9;

    // Set up high-traffic load and configure each node as both sender and receiver
    for (uint32_t i = 0; i < nodes.GetN(); ++i)
    {
        // Configure OnOffApplication to send data to a randomly selected node (e.g., next node in line)
        OnOffHelper onOff("ns3::UdpSocketFactory", Address(InetSocketAddress(interfaces.GetAddress((i + 1) % nodes.GetN()), port)));
        onOff.SetConstantRate(DataRate("100Mbps"), 1024); // High data rate with 1024-byte packets

        // Install the application on each node
        ApplicationContainer app = onOff.Install(nodes.Get(i));
        app.Start(Seconds(0.5 * i)); // Stagger start times to increase collisions
        app.Stop(Seconds(10.0));

        // Configure a PacketSink on each node to receive data on the specified port
        PacketSinkHelper sink("ns3::UdpSocketFactory", Address(InetSocketAddress(Ipv4Address::GetAny(), port)));
        ApplicationContainer sinkApp = sink.Install(nodes.Get(i));
        sinkApp.Start(Seconds(0.0));
        sinkApp.Stop(Seconds(10.0));
    }
}
```

`LogMetrics()`

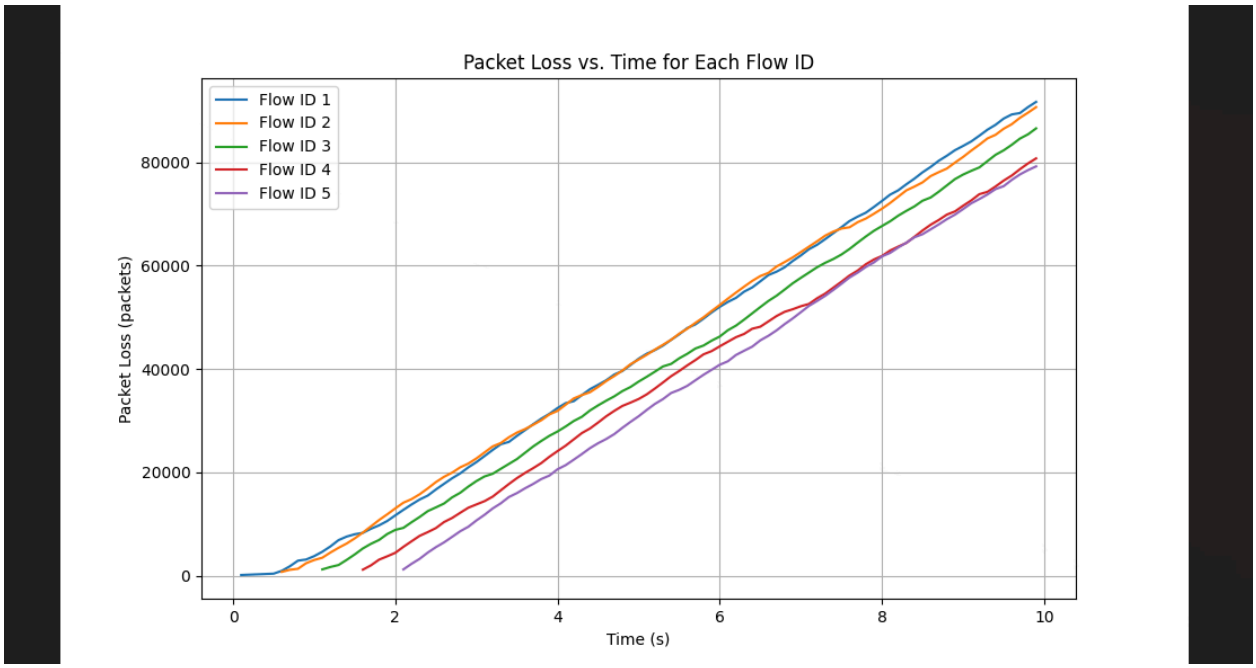
This function logs network performance metrics at regular intervals and appends them to a CSV file for later analysis. Here's a breakdown of its tasks:

- **Flow Statistics Collection**: Flow statistics like throughput, packet loss, and delay are collected for each active flow using the `FlowMonitor`.
- **Metric Calculation**:
    - **Throughput** is calculated by taking the total number of received bytes and dividing by the flow duration in seconds, with the result converted to Mbps.
    - **Delay** is averaged over the received packets.
    - **Packet Loss** is calculated as the difference between the transmitted packets and received packets.

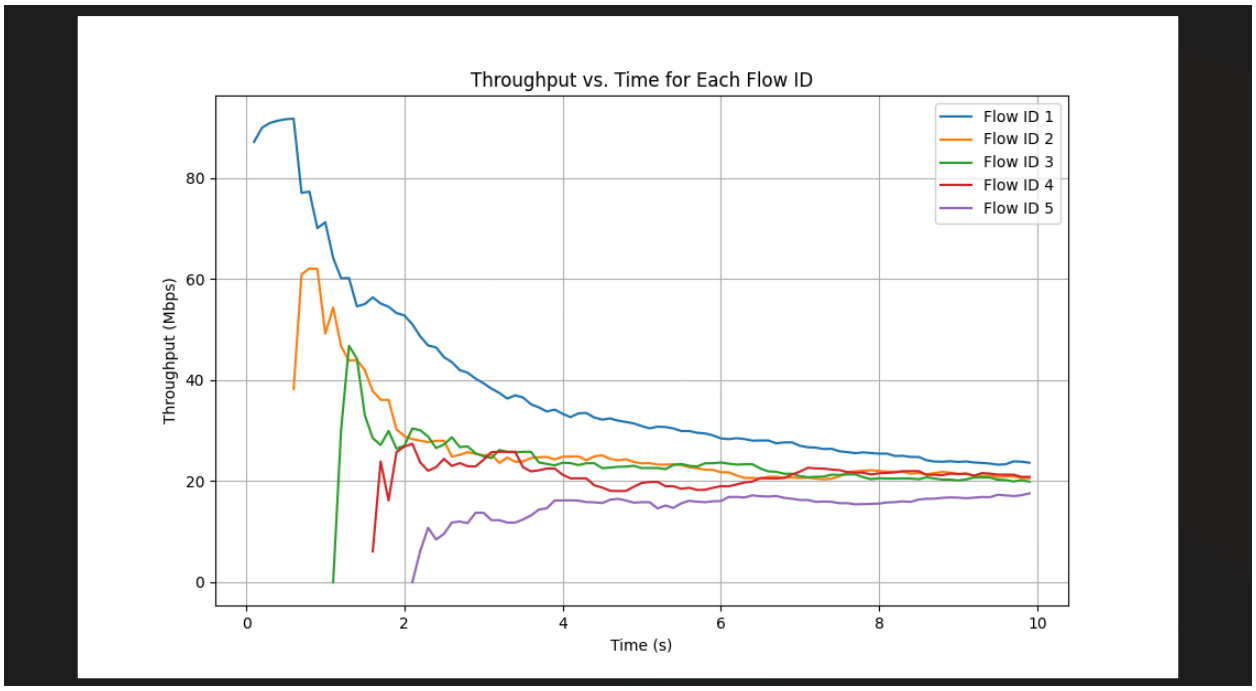
- **Logging to File**: These metrics are written to `csma_simulation_results.csv` with the format: `FlowId, Time, Throughput, PacketLoss, Delay`.
- **Scheduling**: This function schedules itself to run again after the specified interval (`interval`), continuously logging metrics throughout the simulation.

```cpp
void LogMetrics(Ptr<FlowMonitor> flowMonitor, FlowMonitorHelper &flowHelper, double interval)
{
    flowMonitor->CheckForLostPackets();
    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowHelper.GetClassifier());
    FlowMonitor::FlowStatsContainer stats = flowMonitor->GetFlowStats();

    std::ofstream resultsFile("csma_simulation_results.csv", std::ios_base::app);
    double currentTime = Simulator::Now().GetSeconds();

    for (auto iter = stats.begin(); iter != stats.end(); ++iter)
    {
        // Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(iter->first);
        double throughput = iter->second.rxBytes * 8.0 /
            (iter->second.timeLastRxPacket.GetSeconds() - iter->second.timeFirstTxPacket.GetSeconds()) / 1024 / 1024;
        double delay = iter->second.rxPackets > 0 ? iter->second.delaySum.GetSeconds() / iter->second.rxPackets : 0;
        double packetLoss = iter->second.txPackets - iter->second.rxPackets;

        resultsFile << iter->first << "," << currentTime << ","
                    << throughput << "," << packetLoss << "," << delay << "\n";
    }

    resultsFile.close();

    // Schedule the next logging event
    Simulator::Schedule(Seconds(interval), &LogMetrics, flowMonitor, std::ref(flowHelper), interval);
}
```

Packet loss vs time graph for various flows



Throughput vs time for various flows



Part 3 →

i. How CSMA/CD Detects a Collision in a Shared Medium?

Ans)

In a shared medium like Ethernet, CSMA/CD (Carrier Sense Multiple Access with Collision Detection) works by detecting collisions as follows:

Carrier Sensing: Each node first listens to the medium (carrier sensing) to check if any other device is transmitting. If the medium is free, the node begins its transmission.
Collision Detection: While transmitting, the node simultaneously monitors the signal on the medium. If the transmitted signal differs from what the node sent (indicating another node is transmitting), the node detects a collision.
Collision Handling: Upon detecting a collision, the node stops transmission immediately, sends a jamming signal to notify other nodes of the collision, and initiates a backoff procedure.

ii. How the Exponential Backoff Algorithm Works

Ans)

Ethernet receives datagram from network layer, creates frame
If Ethernet senses channel:
if idle: start frame transmission.
if busy: wait until channel idle, then transmit
If entire frame transmitted without collision - done!

If another transmission detected while sending: abort, send jam signal
After aborting, enter binary (exponential) backoff:
after mth collision, chooses K at random from {0,1,2, …, 2m-1}. Ethernet waits K·512 bit times, returns to Step 2
more collisions: longer backoff interval

Part 4 →

i. Why CSMA/CD works well in wired networks but may not be as effective in wireless scenarios.

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) is effective in wired networks because:

- **Collision Detection**: Wired networks allow each node to monitor (or "listen to") the shared channel for collisions as they transmit. This is possible because electrical signals travel through the cable, and nodes can sense any disturbances or interference caused by a collision.
- **Direct, Reliable Sensing**: Wired nodes have direct access to the shared medium and can reliably detect when it is free or busy.

In **wireless networks**, CSMA/CD is less effective due to:

- **Hidden Terminal Problem**: In wireless networks, two devices may not "see" each other (due to obstacles or distance) even though both are within range of the receiver. This lack of awareness can lead both devices to transmit simultaneously, causing a collision.
- **Difficult Collision Detection**: Wireless devices cannot easily detect collisions because the transmission and reception use the same frequency range. While a device is transmitting, it cannot "listen" to the channel to check for collisions effectively.
- **Signal Fading and Interference**: Wireless signals weaken over distance and can be affected by obstacles, which can lead to missed detections of active transmissions.

Due to these limitations, **CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance)** is used in wireless networks, where devices aim to avoid collisions rather than detect them, typically by using acknowledgment packets and waiting periods to verify successful transmissions.

ii. How the protocol detects collisions and how exponential backoff helps mitigate them.

Collision Detection:

- In CSMA/CD, when a device transmits data, it also "listens" to the channel to verify that its transmission is the only signal on the medium.
- If another device transmits at the same time, the two signals interfere, causing a change in the voltage level on the wire that is easily detected as a collision.
- When a collision is detected, all transmitting devices immediately stop sending data.

Exponential Backoff:

- After a collision, each device involved waits a random amount of time before attempting to retransmit. The waiting time is calculated using an exponential backoff algorithm, where the wait time doubles with each consecutive collision (up to a limit).
- For example, if two devices collide, each device chooses a random number of time slots (from a range that doubles after each collision) to wait before retransmitting.
- This backoff strategy reduces the probability that the same devices will collide repeatedly, which is especially useful when the network is busy.

Part B

1. Network Setup

a. Create a 5x5 wireless ad-hoc network in a grid layout

- **Node Creation:** The `NodeContainer` named `c` creates 25 nodes, representing the 5x5 grid.
- **Wifi Setup:**
  - The `WifiHelper` and `WifiMacHelper` objects configure each node with Wi-Fi settings using the `WIFI_STANDARD_80211b` standard.
  - The `YansWifiPhyHelper` sets up physical layer attributes such as `RxGain`, and connects nodes to a `YansWifiChannel` configured with `ConstantSpeedPropagationDelayModel` and `FriisPropagationLossModel` to simulate realistic signal loss and delay.
- **Grid Positioning:**
  - `MobilityHelper` arranges nodes in a grid layout, with `MinX` and `MinY` set to zero and `DeltaX` and `DeltaY` set to 100 meters (the `distance` variable), creating a grid with 5 columns (`GridWidth = 5`) and row-first layout.

b. Use `wifi-simple-adhoc-grid.cc` as a starting point

- The code structure follows that of `wifi-simple-adhoc-grid.cc`, implementing a similar Wi-Fi ad-hoc configuration, grid layout, and traffic flows.

c. Install OLSR for routing

- **Routing Protocol:** OLSR (Optimized Link State Routing) is enabled using `OlsrHelper` and added to `Ipv4ListRoutingHelper` to handle routing within the network. This protocol helps nodes dynamically find and maintain routes to each other in the ad-hoc network.

```
std::string phyMode("DsssRate1Mbps");
double distance = 100.0;      // m
uint32_t packetSize = 1000;   // bytes
uint32_t numPackets = 100;

uint32_t packetSize2 = 1000;  // bytes
uint32_t numPackets2 = 100;

uint32_t packetSize3 = 1000;  // bytes
uint32_t numPackets3 = 100;

uint32_t numNodes = 25;  // 5x5 grid
uint32_t sinkNode1 = 0, sourceNode1 = 24;  // Flow 1: Diagonal 1
uint32_t sinkNode2 = 4, sourceNode2 = 20;  // Flow 2: Diagonal 2
uint32_t sinkNode3 = 10, sourceNode3 = 14; // Flow 3: Middle horizontal

double interval = 0.004; // seconds
bool verbose = false;
bool tracing = true;
```

2. Traffic Configuration

a. Establish three UDP traffic flows

- **Flow 1**: From `n0` to `n24` along one diagonal.
- **Flow 2**: From `n4` to `n20` along the other diagonal.
- **Flow 3**: From `n10` to `n14` along the middle row.
- **Socket Configuration**:
  - Each flow is set up with a source and a sink socket. For example, `source1` transmits from `n0` to `n24` on port 80 for Flow 1, with separate sockets set up for Flow 2 and Flow 3 on unique ports (81 and 82).

b. Set traffic flows to operate at high transmission rates

- **Traffic Rate and Packet Size**:

- Each flow is configured with a high transmission rate by scheduling `GenerateTraffic` to send 1000-byte packets at a 0.004-second interval.

3. Flow Monitoring

a. Implement `FlowMonitor` to track UDP flow performance

- **Flow Monitor**: `FlowMonitorHelper` and `FlowMonitor` are configured to track each flow's performance, enabling detailed analysis of transmission statistics, such as throughput and packet loss.

4. Scheduling

a. Schedule the initiation of each traffic flow at staggered times

- **Scheduling Traffic Generation**:
  - Flow 1 is scheduled to start at 15.0 seconds, Flow 2 at 15.5 seconds, and Flow 3 at 16.0 seconds. These staggered start times allow observation of how different flows interact with each other in the network.

```
// Schedule traffic generation
Simulator::Schedule(Seconds(15.0), &GenerateTraffic, source1, packetSize, numPackets, interPacketInterval);
Simulator::Schedule(Seconds(15.5), &GenerateTraffic, source2, packetSize2, numPackets2, interPacketInterval);
Simulator::Schedule(Seconds(16.0), &GenerateTraffic, source3, packetSize3, numPackets3, interPacketInterval);
```

5. Data Collection

a. Record throughput for each UDP flow

- **Throughput Measurement**:
  - Throughput for each flow is calculated from `FlowMonitor` statistics as `rxBytes * 8.0 / (timeLastRxPacket - timeFirstTxPacket)` in Kbps. This data is printed in the console for each flow, providing an overview of transmission efficiency.

b. Monitor packet collisions and drops at intermediary nodes

- **Packet Drop and Collision Analysis**:
  - The `FlowMonitor` also tracks metrics like transmitted (`txPackets`) and received packets (`rxPackets`), lost packets (`lostPackets`), and times forwarded (`timesForwarded`), which indicate potential areas of congestion or collision in the network.
- **Tracing**:
  - ASCII and PCAP tracing is enabled with `AsciiTraceHelper` and `wifiPhy.EnablePcap`, allowing for a detailed record of packet-level events for offline analysis.

6. RTS/CTS Experimentation

a. Repeat the experiment with RTS/CTS enabled

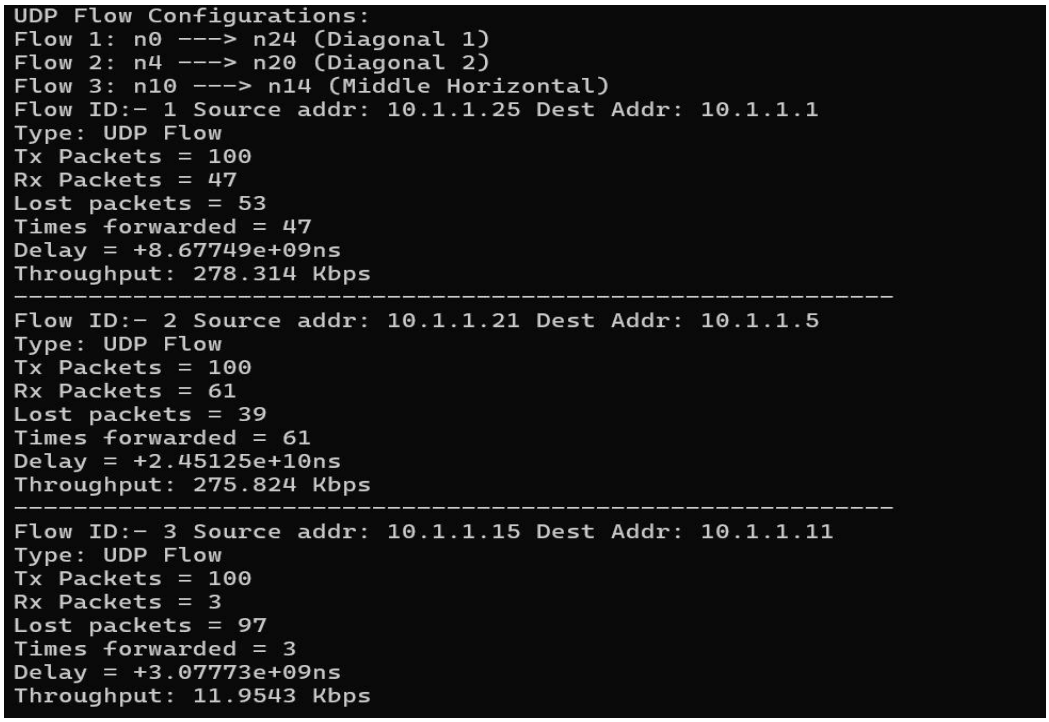
- **Enabling RTS/CTS**:
  - To enable RTS/CTS, configure the `RtsCtsThreshold` parameter in `Config::SetDefault` to a value that triggers RTS/CTS for all packets (typically set to a small packet size threshold, such as 0).

```
Config::SetDefault("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));
```

- **Comparison**:
  - The experiment should be repeated with RTS/CTS enabled, and results, such as throughput and packet drops, can be compared to those from the scenario without RTS/CTS.

Analysis and Reporting

- **Data Analysis**: We simulated the network for different packet sizes and observed that smaller the packet size lesser the packet loss.



500 bytes

```
UDP Flow Configurations:
Flow 1: n0 ---> n24 (Diagonal 1)
Flow 2: n4 ---> n20 (Diagonal 2)
Flow 3: n10 ---> n14 (Middle Horizontal)
Flow ID:- 1 Source addr: 10.1.1.25 Dest Addr: 10.1.1.1
Type: UDP Flow
Tx Packets = 100
Rx Packets = 46
Lost packets = 54
Times forwarded = 46
Delay = +1.66535e+10ns
Throughput: 401.283 Kbps
-----------------------------------------------------------
Flow ID:- 2 Source addr: 10.1.1.21 Dest Addr: 10.1.1.5
Type: UDP Flow
Tx Packets = 100
Rx Packets = 3
Lost packets = 97
Times forwarded = 3
Delay = +6.14016e+09ns
Throughput: 11.6958 Kbps
-----------------------------------------------------------
Flow ID:- 3 Source addr: 10.1.1.15 Dest Addr: 10.1.1.11
Type: UDP Flow
Tx Packets = 100
Rx Packets = 42
Lost packets = 58
Times forwarded = 42
Delay = +1.62223e+10ns
Throughput: 365.534 Kbps
-----------------------------------------------------------
```

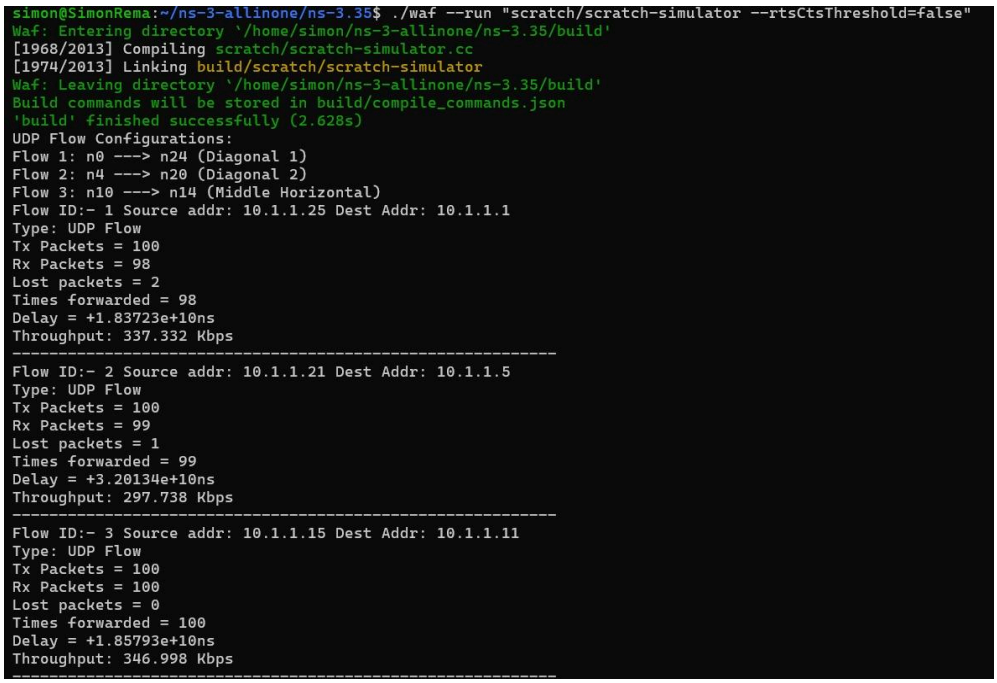
1000 bytes

```
UDP Flow Configurations:
Flow 1: n0 ---> n24 (Diagonal 1)
Flow 2: n4 ---> n20 (Diagonal 2)
Flow 3: n10 ---> n14 (Middle Horizontal)
Flow ID:- 1 Source addr: 10.1.1.25 Dest Addr: 10.1.1.1
Type: UDP Flow
Tx Packets = 100
Rx Packets = 91
Lost packets = 9
Times forwarded = 91
Delay = +1.04381e+10ns
Throughput: 182.228 Kbps
-----------------------------------------------------------
Flow ID:- 2 Source addr: 10.1.1.21 Dest Addr: 10.1.1.5
Type: UDP Flow
Tx Packets = 100
Rx Packets = 99
Lost packets = 1
Times forwarded = 99
Delay = +2.46519e+10ns
Throughput: 197.304 Kbps
-----------------------------------------------------------
Flow ID:- 3 Source addr: 10.1.1.15 Dest Addr: 10.1.1.11
Type: UDP Flow
Tx Packets = 100
Rx Packets = 97
Lost packets = 3
Times forwarded = 97
Delay = +4.2422e+10ns
Throughput: 190.478 Kbps
-----------------------------------------------------------
```

200 bytes

- **Observations**:

We simulated the network with and without RTS/CTS and saw a slight increase in the number of packets successfully received.



Without RTS/CTS

```
simon@SimonRema:~/ns-3-allinone/ns-3.35$ ./waf --run "scratch/scratch-simulator --rtsCtsThreshold=true"
Waf: Entering directory `/home/simon/ns-3-allinone/ns-3.35/build'
Waf: Leaving directory `/home/simon/ns-3-allinone/ns-3.35/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.269s)
UDP Flow Configurations:
Flow 1: n0 ---> n24 (Diagonal 1)
Flow 2: n4 ---> n20 (Diagonal 2)
Flow 3: n10 ---> n14 (Middle Horizontal)
Flow ID:- 1 Source addr: 10.1.1.25 Dest Addr: 10.1.1.1
Type: UDP Flow
Tx Packets = 100
Rx Packets = 98
Lost packets = 2
Times forwarded = 98
Delay = +2.48172e+10ns
Throughput: 287.099 Kbps
------------------------------------------------------------
Flow ID:- 2 Source addr: 10.1.1.21 Dest Addr: 10.1.1.5
Type: UDP Flow
Tx Packets = 100
Rx Packets = 100
Lost packets = 0
Times forwarded = 100
Delay = +3.37109e+10ns
Throughput: 272.765 Kbps
------------------------------------------------------------
Flow ID:- 3 Source addr: 10.1.1.15 Dest Addr: 10.1.1.11
Type: UDP Flow
Tx Packets = 100
Rx Packets = 100
Lost packets = 0
Times forwarded = 100
Delay = +2.5535e+10ns
Throughput: 294.175 Kbps
------------------------------------------------------------
```

With RTS/CTS

Observations from Flow Monitor for packet size 500 bytes.

Flow Id:1
======
UDP 10.1.1.25/49153---->10.1.1.1/80

Tx bitrate:1066.67kbps
Rx bitrate:365.031kbps
Mean delay:313.915ms
Packet Loss ratio:26%

timeFirstTxPacket= 1.5e+10ns
timeFirstRxPacket= 1.50486e+10ns
timeLastTxPacket= 1.5396e+10ns
timeLastRxPacket= 1.59049e+10ns
delaySum= 2.32297e+10ns
jitterSum= 4.87326e+08ns
lastDelay= 2.32297e+10ns
txBytes= 52800
rxBytes= 39072
txPackets= 100
rxPackets= 74
lostPackets= 26
timesForwarded= 74

delayHistogram nBins:521
Index:48 Start:0.048 Width:0.001 Count:1
Index:50 Start:0.05 Width:0.001 Count:1
Index:52 Start:0.052 Width:0.001 Count:2
Index:65 Start:0.065 Width:0.001 Count:1
Index:67 Start:0.067 Width:0.001 Count:1
Index:75 Start:0.075 Width:0.001 Count:1
Index:83 Start:0.083 Width:0.001 Count:1
Index:91 Start:0.091 Width:0.001 Count:1
Index:106 Start:0.106 Width:0.001 Count:1
Index:108 Start:0.108 Width:0.001 Count:1
Index:134 Start:0.134 Width:0.001 Count:1
Index:142 Start:0.142 Width:0.001 Count:1
Index:150 Start:0.15 Width:0.001 Count:1
Index:163 Start:0.163 Width:0.001 Count:1
Index:165 Start:0.165 Width:0.001 Count:1
Index:167 Start:0.167 Width:0.001 Count:1
Index:181 Start:0.181 Width:0.001 Count:1
Index:188 Start:0.188 Width:0.001 Count:1
Index:196 Start:0.196 Width:0.001 Count:1
Index:198 Start:0.198 Width:0.001 Count:1
Index:206 Start:0.206 Width:0.001 Count:1
Index:209 Start:0.209 Width:0.001 Count:1
Index:212 Start:0.212 Width:0.001 Count:1
Index:225 Start:0.225 Width:0.001 Count:1
Index:239 Start:0.239 Width:0.001 Count:1
Index:247 Start:0.247 Width:0.001 Count:1

Flow Id:2
======
UDP 10.1.1.21/49153---->10.1.1.5/81

Tx bitrate:1066.67kbps
Rx bitrate:336.893kbps
Mean delay:547.077ms
Packet Loss ratio:16%

timeFirstTxPacket= 3e+10ns
timeFirstRxPacket= 3.00253e+10ns
timeLastTxPacket= 3.0396e+10ns
timeLastRxPacket= 3.10785e+10ns
delaySum= 4.59545e+10ns
jitterSum= 7.61451e+08ns
lastDelay= 4.59545e+10ns
txBytes= 52800
rxBytes= 44352
txPackets= 100
rxPackets= 84
lostPackets= 16
timesForwarded= 84

delayHistogram nBins:739
Index:25 Start:0.025 Width:0.001 Count:1
Index:27 Start:0.027 Width:0.001 Count:1
Index:74 Start:0.074 Width:0.001 Count:1
Index:106 Start:0.106 Width:0.001 Count:1
Index:276 Start:0.276 Width:0.001 Count:1
Index:283 Start:0.283 Width:0.001 Count:1
Index:290 Start:0.29 Width:0.001 Count:1
Index:302 Start:0.302 Width:0.001 Count:1
Index:304 Start:0.304 Width:0.001 Count:1
Index:319 Start:0.319 Width:0.001 Count:1
Index:321 Start:0.321 Width:0.001 Count:1
Index:323 Start:0.323 Width:0.001 Count:1
Index:325 Start:0.325 Width:0.001 Count:1
Index:327 Start:0.327 Width:0.001 Count:1
Index:337 Start:0.337 Width:0.001 Count:1
Index:345 Start:0.345 Width:0.001 Count:1
Index:347 Start:0.347 Width:0.001 Count:1
Index:348 Start:0.348 Width:0.001 Count:1
Index:416 Start:0.416 Width:0.001 Count:1
Index:417 Start:0.417 Width:0.001 Count:1
Index:425 Start:0.425 Width:0.001 Count:1
Index:439 Start:0.439 Width:0.001 Count:1
Index:447 Start:0.447 Width:0.001 Count:1
Index:449 Start:0.449 Width:0.001 Count:1
Index:457 Start:0.457 Width:0.001 Count:1
Index:459 Start:0.459 Width:0.001 Count:1

Flow Id:3
======
UDP 10.1.1.15/49153---->10.1.1.11/82

Tx bitrate:1066.67kbps
Rx bitrate:365.881kbps
Mean delay:317.807ms
Packet Loss ratio:24%

timeFirstTxPacket= 4.5e+10ns
timeFirstRxPacket= 4.50351e+10ns
timeLastTxPacket= 4.5396e+10ns
timeLastRxPacket= 4.59125e+10ns
delaySum= 2.41533e+10ns
jitterSum= 5.62689e+08ns
lastDelay= 2.41533e+10ns
txBytes= 52800
rxBytes= 40128
txPackets= 100
rxPackets= 76
lostPackets= 24
timesForwarded= 76

delayHistogram nBins:534
Index:35 Start:0.035 Width:0.001 Count:1
Index:36 Start:0.036 Width:0.001 Count:1
Index:50 Start:0.05 Width:0.001 Count:1
Index:64 Start:0.064 Width:0.001 Count:1
Index:78 Start:0.078 Width:0.001 Count:1
Index:85 Start:0.085 Width:0.001 Count:1
Index:99 Start:0.099 Width:0.001 Count:1
Index:107 Start:0.107 Width:0.001 Count:1
Index:115 Start:0.115 Width:0.001 Count:1
Index:123 Start:0.123 Width:0.001 Count:1
Index:125 Start:0.125 Width:0.001 Count:1
Index:127 Start:0.127 Width:0.001 Count:1
Index:129 Start:0.129 Width:0.001 Count:1
Index:131 Start:0.131 Width:0.001 Count:1
Index:132 Start:0.132 Width:0.001 Count:1
Index:140 Start:0.14 Width:0.001 Count:1
Index:148 Start:0.148 Width:0.001 Count:1
Index:156 Start:0.156 Width:0.001 Count:1
Index:169 Start:0.169 Width:0.001 Count:1
Index:177 Start:0.177 Width:0.001 Count:1
Index:179 Start:0.179 Width:0.001 Count:1
Index:187 Start:0.187 Width:0.001 Count:1
Index:200 Start:0.2 Width:0.001 Count:1
Index:202 Start:0.202 Width:0.001 Count:1
Index:210 Start:0.21 Width:0.001 Count:1
Index:230 Start:0.23 Width:0.001 Count:1