ASSIGNMENT - 06

GROUP - 15 :

220101008 - Adloori Chandana
220101009 - Aetukuri Sri Durga
220101010 - Alampally Khushi
220123052 - Ramineni Hardhika

DRIVE LINK : CN_ASSIGNMENT_06

PART - A:

(Command to run file for part A - `./ns3 run scratch/ethernet`)

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) is a network protocol used in Ethernet networks to manage access to a shared communication channel. Here's a brief overview:

1. **Carrier Sense**: Before transmitting data, a device listens to the network (the carrier) to check if the channel is idle or if another device is transmitting. If the channel is busy, the device waits until it's free.
2. **Multiple Access**: Multiple devices can access the shared network medium, but they must follow the CSMA protocol to avoid collisions.
3. **Collision Detection**: While transmitting, the device also listens to the network. If it detects a collision (two devices transmitting at the same time), it stops transmitting, sends a jamming signal to notify other devices of the collision, and then waits for a random time before retransmitting.

This process helps ensure fair access to the network and minimizes the chances of collisions, although it is not used in modern switched Ethernet networks, where collisions are less likely.

Network Setup

Setting up a Simple Ethernet LAN:

This code creates a network setup in NS-3 with five nodes, each representing a host connected to an Ethernet LAN via CSMA.

The following code initializes the nodes and sets up a CSMA channel with attributes that match a standard Ethernet LAN configuration:

```
// Simulation parameters
uint32_t nNodes = 5;
double simulationTime = 10.0; // seconds
uint32_t packetSize = 1024;   // bytes
std::string dataRate = "100Mbps";
std::string delay = "0.1ms";
```

```
// Create nodes
NodeContainer nodes;
nodes.Create(nNodes);
```

```
// Create CSMA channel
CsmaHelper csma;
csma.SetChannelAttribute("DataRate", StringValue(dataRate));
csma.SetChannelAttribute("Delay", StringValue(delay));
```

CSMA Channel Attributes:

The data rate is set to **100 Mbps**, and delay is set to **0.1 ms**, reflecting a typical high-speed Ethernet LAN configuration.

Connecting Hosts to the Network:

The code uses the CSMA helper to install network devices on each node, enabling communication over the shared Ethernet network.

```

// Install Internet stack
InternetStackHelper internet;
internet.Install(nodes);
// Assign IP addresses
Ipv4AddressHelper ipv4;
ipv4.SetBase("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign(devices);
```

InternetStackHelper internet; internet.Install(nodes);

- This creates an instance of the InternetStackHelper class and installs the necessary network stack (IP, TCP, UDP, etc.) on the nodes in the simulation. This allows the nodes to use Internet protocols for communication.

Ipv4AddressHelper ipv4; ipv4.SetBase("10.1.1.0", "255.255.255.0");

- This sets up an IPv4 address pool for the network. The base IP address is 10.1.1.0 with a subnet mask 255.255.255.0. This defines the range of IP addresses that can be assigned to the devices in the simulation.

Ipv4InterfaceContainer interfaces = ipv4.Assign(devices);

- This assigns the IPv4 addresses from the pool to the network devices (devices) installed on the nodes. The interfaces container holds the IP addresses assigned to each device. This step ensures that each node has a unique IP address to communicate within the simulated network.

```
   // Set up applications

   OnOffHelper onOff("ns3::UdpSocketFactory", Address());

   onOff.SetConstantRate(DataRate("50Mbps"), packetSize);

   // Randomize application start times to allow organic collisions

   Ptr<UniformRandomVariable> startTime =
CreateObject<UniformRandomVariable>();

   for (uint32_t i = 0; i < nNodes; i++) {

       AddressValue remoteAddress(InetSocketAddress(interfaces.GetAddress((i +
1) % nNodes), 8080));

       onOff.SetAttribute("Remote", remoteAddress);

       ApplicationContainer app = onOff.Install(nodes.Get(i));

       app.Start(Seconds(1.0 + startTime->GetValue(0.0, 0.5))); // Start
between 1.0 and 1.5 seconds

       app.Stop(Seconds(simulationTime));

       // Schedule collision detection for each node after app start

    Simulator::Schedule(Seconds(1.1), &DetectAndHandleCollision, nodes.Get(i));
```

1. **OnOffHelper onOff("ns3::UdpSocketFactory", Address());**
   - Creates an application helper that sends UDP traffic from nodes.
2.
3. **onOff.SetConstantRate(DataRate("50Mbps"), packetSize);**
   - Sets the transmission rate to 50 Mbps and the packet size for the UDP traffic.
4. **Ptr<UniformRandomVariable> startTime = CreateObject<UniformRandomVariable>();**
   - Creates a random variable generator to randomize the start time of applications on each node.
5. **for (uint32_t i = 0; i < nNodes; i++) { … }**
   - Loops over each node and installs the On/Off UDP application with a randomized start time between 1.0 and 1.5 seconds. The application sends packets to a remote address (next node in the circular network).
6. **Simulator::Schedule(Seconds(1.1), &DetectAndHandleCollision, nodes.Get(i));**
   - Schedules a collision detection function for each node after 1.1 seconds. This simulates checking for collisions and handling them with backoff.

```
   // Enable logging for CSMA/CD to observe collisions and backoff

   LogComponentEnable("CsmaCdSimulation", LOG_LEVEL_INFO);

   // Set up FlowMonitor for statistics

   monitor = flowmon.InstallAll();

   // Open CSV file and write headers (only once)

   std::ofstream csvFile("metrics.csv");

   csvFile << "Time (s),Tx Packets,Rx Packets,Throughput (Mbps),Packet Loss Ratio
(%),Average Delay (s)\n";

   csvFile.close();
```

LogComponentEnable("CsmaCdSimulation", LOG_LEVEL_INFO); enables logging for the CSMA/CD simulation at the INFO level. Then, monitor = flowmon.InstallAll(); installs the flow monitor to track network traffic statistics, and std::ofstream csvFile("metrics.csv"); opens a CSV file to log header information for metrics such as transmission, reception, throughput, packet loss, and delay.

```
   // Schedule WriteFlowStatsToCSV at regular intervals

       for (double time = 1.0; time <= simulationTime; time += 0.01) {

        Simulator::Schedule(Seconds(time), &WriteFlowStatsToCSV, time);

   }

   // Run simulation

   Simulator::Stop(Seconds(simulationTime));

   Simulator::Run();

   // Cleanup

   Simulator::Destroy();
```

Schedules **WriteFlowStatsToCSV** at 0.01-second intervals.

Runs the simulation for **simulationTime** seconds.

Cleans up after the simulation.

```
/

// Function to simulate collision detection and exponential backoff

void DetectAndHandleCollision(Ptr<Node> node) {

   uint32_t nodeId = node->GetId();

   if (backoffCounters.find(nodeId) == backoffCounters.end()) {

       backoffCounters[nodeId] = 0;
```

```
    }

    uint32_t backoffExp = std::min(backoffCounters[nodeId], MAX_BACKOFF_EXP);

    double delay = std::pow(2, backoffExp) * 0.0001; // Base delay is 0.1ms,
exponentially increasing

    NS_LOG_INFO("Collision detected on node " << nodeId << " at time " <<
Simulator::Now().GetSeconds() << "s");

    std::cout << "Collision detected on node " << nodeId << " at time " <<
Simulator::Now().GetSeconds() << "s\n";

    std::cout << "Node " << nodeId << " backing off for " << delay << " seconds before
retransmitting.\n";

    backoffCounters[nodeId]++;

    Simulator::Schedule(Seconds(delay), &DetectAndHandleCollision, node);

}
```

The DetectAndHandleCollision function handles collision detection and applies exponential backoff for a node:

1. **Initialize Backoff Counter:** Checks if the node has a backoff counter. If not, it initializes it to 0.
2. **Calculate Exponential Delay:** Computes the delay as 0.1ms * 2^backoffExp, doubling the delay with each collision until capped by MAX_BACKOFF_EXP.
3. **Log Collision Info:** Logs and prints messages showing the collision, node ID, and backoff delay.
4. **Increment Counter and Schedule Retransmission:** Increments the backoff counter and reschedules the function to try retransmission after the calculated delay, achieving exponential backoff.

NS3 FLOW MONITOR :

In this code, the ns-3 Flow Monitor is used to collect and analyze traffic statistics (like packet transmission, reception, delay, throughput, and packet loss) during the simulation.

- FlowMonitorHelper flowmon; creates a helper object to manage flow monitoring.
- monitor = flowmon.InstallAll(); installs the flow monitor on all nodes to start tracking the network traffic flows between them.
- WriteFlowStatsToCSV(double currentTime) is a function that uses the flow monitor to gather statistics at regular intervals and write them to a CSV file. This includes metrics like throughput, packet loss, and delay, which are computed from the flow data provided by the monitor.

The flow monitor provides a detailed report of the performance of each flow in the network, which is essential for evaluating the quality of the communication and identifying issues such as packet loss or delay.

```cpp
// Function to aggregate statistics for all flows at the current timestamp

void WriteFlowStatsToCSV(double currentTime) {

   // Create and open CSV file for appending

   std::ofstream csvFile;

   csvFile.open("metrics.csv", std::ios::app);  // Append mode

   if (!csvFile.is_open()) {

       NS_LOG_ERROR("Failed to open CSV file");

       return;

   }

   // Process FlowMonitor data at the current time

   monitor->CheckForLostPackets();
   Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());

   std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();

   // Initialize aggregate values

   double totalThroughput = 0.0;

   double totalPacketLoss = 0.0;

   double totalAvgDelay = 0.0;

   uint32_t totalTxPackets = 0;

   uint32_t totalRxPackets = 0;

   // Iterate through each flow and aggregate statistics

   for (auto iter = stats.begin(); iter != stats.end(); ++iter) {

       totalTxPackets += iter->second.txPackets;

       totalRxPackets += iter->second.rxPackets;

       totalThroughput += iter->second.rxBytes * 8.0 / currentTime / 1e6; // Mbps

       totalPacketLoss += (1 - ((double)iter->second.rxPackets /
iter->second.txPackets)) * 100;

       totalAvgDelay += iter->second.rxPackets > 0 ?
iter->second.delaySum.GetSeconds() / iter->second.rxPackets : 0;

   }

   // Calculate averages for throughput, packet loss ratio, and delay

   double avgThroughput = totalThroughput / stats.size();

   double avgPacketLossRatio = totalPacketLoss / stats.size();

   double avgDelay = totalRxPackets > 0 ? totalAvgDelay / stats.size() : 0;

   // Write aggregated data to CSV file

   csvFile << currentTime << ","
```

```
<< totalTxPackets << "," << totalRxPackets << ","

<< avgThroughput << "," << avgPacketLossRatio << "," << avgDelay << "\n";

   // Close the file

   csvFile.close();

}
```

- **Purpose**: This function aggregates network metrics (throughput, packet
Purpose: This function aggregates network metrics (throughput, packet
  loss, and delay) and writes them to a CSV file.
- **Process**:

CSV File: Opens (or creates) metrics.csv to log data. Checks if the file is open, logs an error if it fails.

Get Stats: Retrieves the flow statistics and analyzes key metrics.

Aggregate Statistics: Calculates throughput (in Mbps), packet loss ratio, and average delay for all flows.

Write Data: Appends metrics to the CSV file and closes it.

Subpart 3 :

i. CSMA/CD Collision Detection:

- **Carrier Sensing**: Node listens to the channel before transmitting. If the channel is busy, it waits.
- **Transmission**: Once the channel is idle, the node transmits.
- **Collision Detection**: While transmitting, the node listens for discrepancies in the signal, indicating a collision.
- **Log of Collision**: The `DetectAndHandleCollision()` function logs the collision and triggers the backoff mechanism.

ii. Exponential Backoff Algorithm:

- **Initial Backoff**: After a collision, the node waits for a random time before retransmitting.
- **Exponential Increase**: Backoff time increases exponentially with each subsequent collision.
- **Randomized Delay**: The backoff time is randomly varied to prevent synchronized retries.
- **Retransmission**: The node retries after the calculated backoff time (`Simulator::Schedule()` is used in the code).

Subpart 4 :

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

- **Wired Networks:** In wired Ethernet networks, the communication medium is typically a physical cable where signal transmission and reception are relatively predictable. The protocol can effectively detect collisions because nodes can listen to the network and detect when another transmission occurs. The presence of the physical medium allows collision detection in real-time.
- **Wireless Networks:** In wireless networks, CSMA/CD becomes less effective due to the **hidden node problem** and **signal interference**. In wireless environments:

  Hidden Node Problem: A node might not be able to hear another node's transmission if it is too far away or obstructed by physical barriers, leading to potential collisions that go undetected by the transmitter.

  Signal Interference: Wireless signals are more prone to interference and fading, making collision detection less reliable. Also, the **physical layer** does not always provide clear feedback (like voltage levels in wired networks), making it difficult for nodes to detect collisions.

Due to these issues, wireless networks often use protocols like **CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance)**, which is designed to avoid collisions rather than detect them after they occur.

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

- **Collision Detection:**

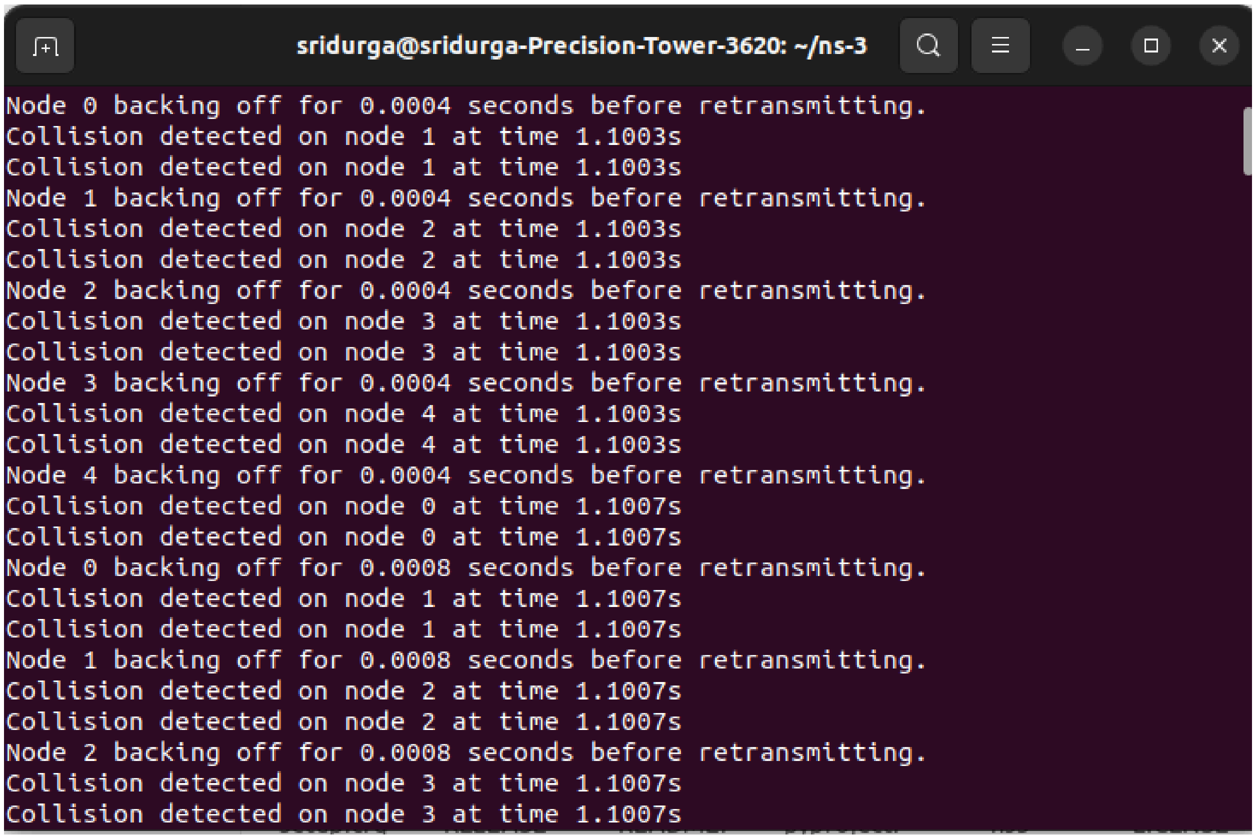
  In CSMA/CD, each node listens for collisions during transmission. If two nodes transmit at the same time, their signals will interfere, and the resulting collision will be detected by both nodes when they receive corrupted data. As soon as a collision is detected, the transmitting nodes stop sending data and send a "jam" signal to notify all nodes about the collision.

- **Exponential Backoff:**After a collision, nodes wait for a random backoff time before attempting to retransmit. The **exponential backoff** algorithm increases the backoff time with each successive collision to reduce the chances of repeated collisions. Specifically:
  i. After the first collision, the backoff time is chosen randomly from a small range.
  ii. After a second collision, the backoff range increases (the backoff time is doubled).
  iii. This process continues, exponentially increasing the range, making the protocol less likely to cause further collisions and allowing for a more efficient recovery.

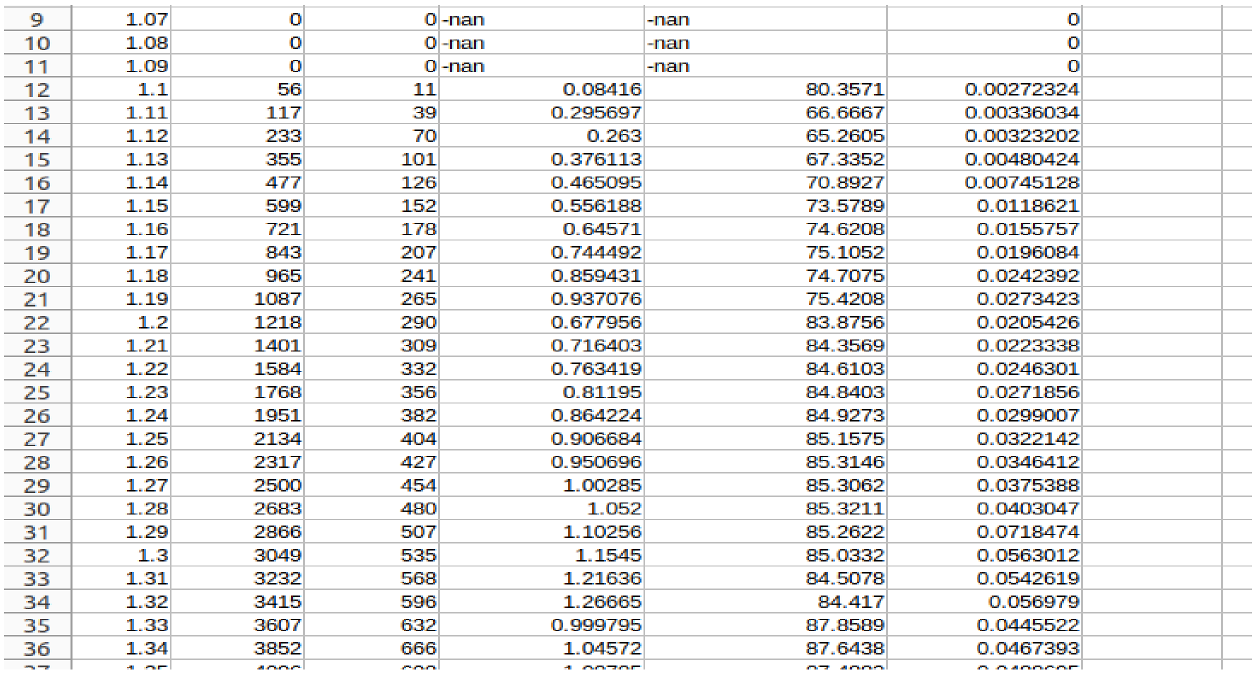
The combination of collision detection and exponential backoff ensures that CSMA/CD helps manage network traffic effectively, especially in networks with moderate traffic. However, in high-traffic environments or wireless networks, other protocols like **CSMA/CA** may be necessary to reduce collision probabilities and improve network performance.

Outputs:

This is the part of the output generated by the code in the terminal stating collision detection and backoff time intervals at each backoff.

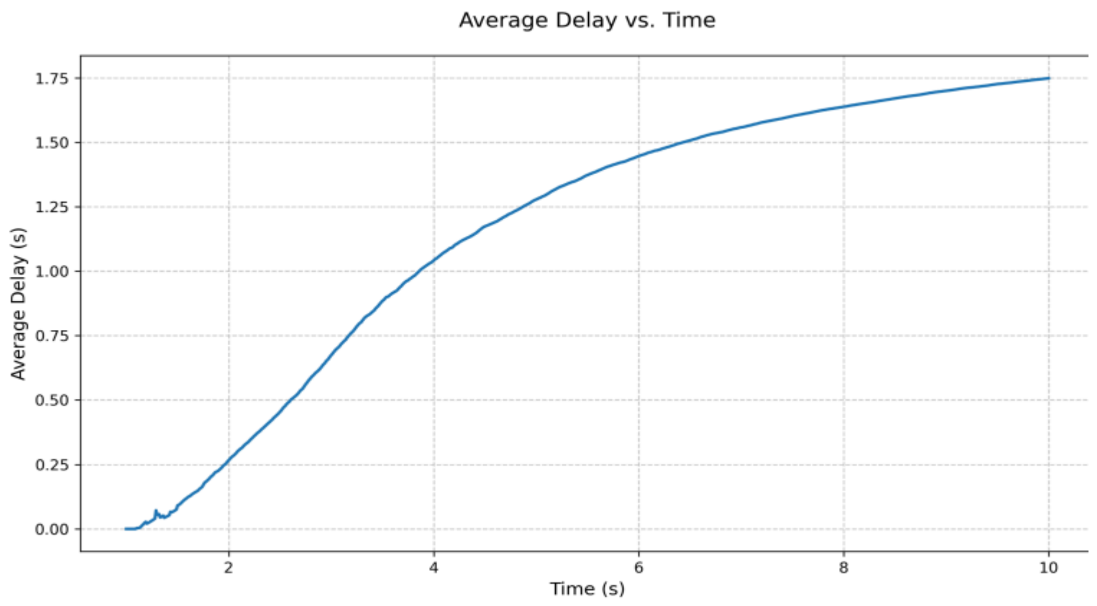


```
sridurga@sridurga-Precision-Tower-3620: ~/ns-3
Node 0 backing off for 0.0004 seconds before retransmitting.
Collision detected on node 1 at time 1.1003s
Collision detected on node 1 at time 1.1003s
Node 1 backing off for 0.0004 seconds before retransmitting.
Collision detected on node 2 at time 1.1003s
Collision detected on node 2 at time 1.1003s
Node 2 backing off for 0.0004 seconds before retransmitting.
Collision detected on node 3 at time 1.1003s
Collision detected on node 3 at time 1.1003s
Node 3 backing off for 0.0004 seconds before retransmitting.
Collision detected on node 4 at time 1.1003s
Collision detected on node 4 at time 1.1003s
Node 4 backing off for 0.0004 seconds before retransmitting.
Collision detected on node 0 at time 1.1007s
Collision detected on node 0 at time 1.1007s
Node 0 backing off for 0.0008 seconds before retransmitting.
Collision detected on node 1 at time 1.1007s
Collision detected on node 1 at time 1.1007s
Node 1 backing off for 0.0008 seconds before retransmitting.
Collision detected on node 2 at time 1.1007s
Collision detected on node 2 at time 1.1007s
Node 2 backing off for 0.0008 seconds before retransmitting.
Collision detected on node 3 at time 1.1007s
Collision detected on node 3 at time 1.1007s
```

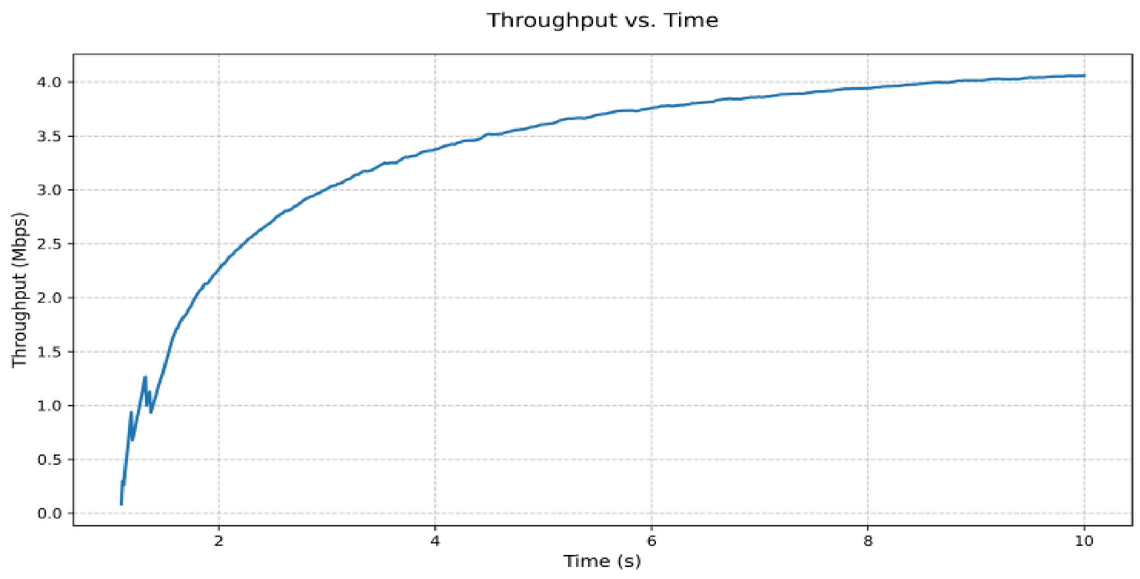


9	1.07	0	0	-nan	-nan	0
10	1.08	0	0	-nan	-nan	0
11	1.09	0	0	-nan	-nan	0
12	1.1	56	11	0.08416	80.3571	0.00272324
13	1.11	117	39	0.295697	66.6667	0.00336034
14	1.12	233	70	0.263	65.2605	0.00323202
15	1.13	355	101	0.376113	67.3352	0.00480424
16	1.14	477	126	0.465095	70.8927	0.00745128
17	1.15	599	152	0.556188	73.5789	0.0118621
18	1.16	721	178	0.64571	74.6208	0.0155757
19	1.17	843	207	0.744492	75.1052	0.0196084
20	1.18	965	241	0.859431	74.7075	0.0242392
21	1.19	1087	265	0.937076	75.4208	0.0273423
22	1.2	1218	290	0.677956	83.8756	0.0205426
23	1.21	1401	309	0.716403	84.3569	0.0223338
24	1.22	1584	332	0.763419	84.6103	0.0246301
25	1.23	1768	356	0.81195	84.8403	0.0271856
26	1.24	1951	382	0.864224	84.9273	0.0299007
27	1.25	2134	404	0.906684	85.1575	0.0322142
28	1.26	2317	427	0.950696	85.3146	0.0346412
29	1.27	2500	454	1.00285	85.3062	0.0375388
30	1.28	2683	480	1.052	85.3211	0.0403047
31	1.29	2866	507	1.10256	85.2622	0.0718474
32	1.3	3049	535	1.1545	85.0332	0.0563012
33	1.31	3232	568	1.21636	84.5078	0.0542619
34	1.32	3415	596	1.26665	84.417	0.056979
35	1.33	3607	632	0.999795	87.8589	0.0445522
36	1.34	3852	666	1.04572	87.6438	0.0467393

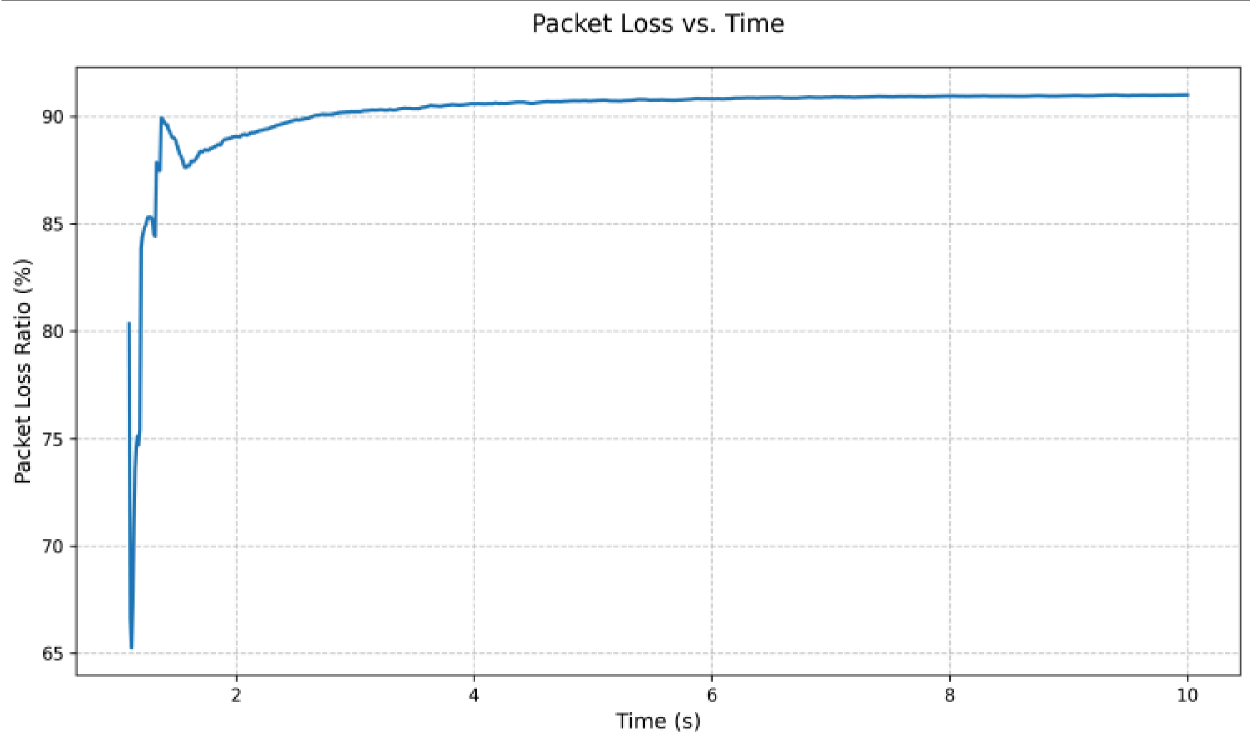
This is the metric data saved into metric.csv



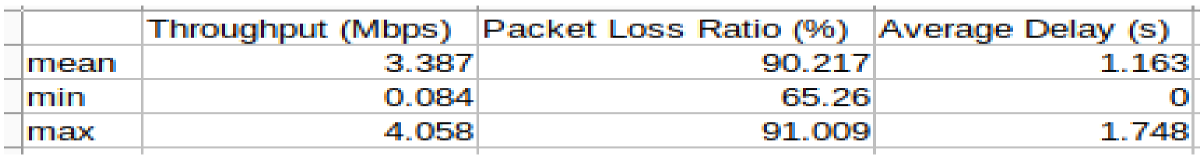
As time progresses, the average delay increases, which suggests that collisions or network congestion may be contributing to longer wait times for packets to be successfully transmitted. This is expected in a CSMA/CD setup with high traffic because as collisions occur, nodes need to wait and retransmit, which increases delay. The exponential increase indicates frequent backoffs and retransmissions due to ongoing collisions in the network.



The throughput initially rises sharply as packets are successfully transmitted. It eventually stabilizes, showing the maximum data transfer rate achieved by the network given the collisions and backoffs. The curve's shape suggests that the network reaches a stable state where throughput remains steady despite collisions. CSMA/CD helps maximize throughput by ensuring that, after each collision, nodes back off and avoid transmitting simultaneously, allowing data to flow smoothly after an initial period of congestion.



The packet loss ratio starts high and then gradually stabilizes, with the initial spikes likely indicating periods of heavy collisions. This high packet loss at the beginning is typical in CSMA/CD networks under high load, where multiple nodes are attempting to send packets simultaneously. As the network stabilizes, fewer packets are lost as nodes back off and synchronize their transmissions more effectively, reducing the collision rate.



	Throughput (Mbps)	Packet Loss Ratio (%)	Average Delay (s)
mean	3.387	90.217	1.163
min	0.084	65.26	0
max	4.058	91.009	1.748

The high throughput, packet loss ratio, and average delay are due to frequent collisions in a high-traffic CSMA/CD network:

1. Throughput: The mean throughput of 3.387 Mbps (max 4.058 Mbps) is below the theoretical max (e.g., 100 Mbps) because frequent collisions and retransmissions limit data flow.
2. Packet Loss Ratio: The high packet loss ratio (mean 90.217%) indicates that many transmissions fail due to collisions, common in a shared network with multiple hosts competing for bandwidth.
3. Average Delay: The average delay of 1.163 seconds (max 1.748 seconds) reflects the waiting time caused by CSMA/CD's exponential backoff after each collision, resulting in longer delivery times.

These metrics highlight the impact of collisions on network performance under heavy load.

The glitches in the graphs come from fluctuations caused by CSMA/CD handling high network traffic:

Average Delay vs. Time: Delay spikes occur when multiple nodes collide and back off with random delays, causing intermittent increases.

Throughput vs. Time: Throughput drops happen during collisions or backoff periods, while random retransmission timing creates temporary bursts.Packet Loss Ratio vs. Time: Initial high collisions lead to packet loss spikes, and random backoffs cause periodic increases.

PART-B

Routing Protocol:

- **OLSR** is suitable for mobile ad-hoc networks.
- It ensures stable and optimized routing, providing up-to-date routing tables even in dynamic network environments

OLSR (Optimized Link State Routing) is a proactive, table-driven routing protocol designed for mobile ad hoc networks (MANETs). It is an optimization of the traditional link-state routing protocol and is primarily used in scenarios with dynamic and wireless network topologies.

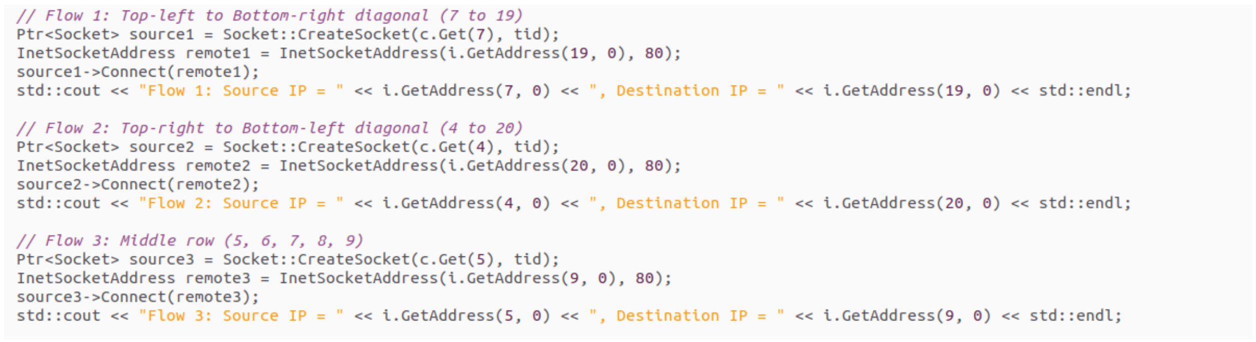
Key Features:

1. **Proactive Nature**: OLSR constantly updates the routing tables to ensure that routes are always available, even if no traffic is being sent.
2. **Optimized for MANETs**: It reduces overhead in terms of control message exchanges by introducing the concept of Multipoint Relays (MPRs). MPRs help minimize the number of nodes required to forward control messages, reducing the amount of routing information exchanged.
3. **Link-State Updates:** Each node periodically exchanges control information (Hello and Topology Control messages) to maintain up-to-date routing information.
4. **MPR Selection:** Nodes select a minimal set of neighbors, called MPRs, which are responsible for forwarding routing messages. This helps in reducing the flooding of control messages in the network.
5. **Routing Table:** The protocol constructs a complete view of the network topology using link-state advertisements, allowing it to compute optimal routes using Dijkstra's algorithm.

OLSR is efficient in terms of control message overhead and is suitable for dense, mobile networks where nodes frequently join or leave the network.

Traffic Configuration:

- Three UDP traffic flows were established:
  - One flow along each diagonal of the grid.
  - One flow along the middle row of the grid.
- These traffic flows were set to operate at high transmission rates to test network performance under heavy load.

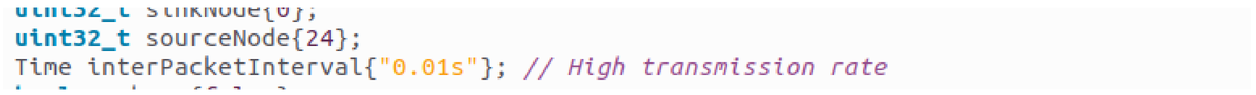


```
// Flow 1: Top-left to Bottom-right diagonal (7 to 19)
Ptr<Socket> source1 = Socket::CreateSocket(c.Get(7), tid);
InetSocketAddress remote1 = InetSocketAddress(i.GetAddress(19, 0), 80);
source1->Connect(remote1);
std::cout << "Flow 1: Source IP = " << i.GetAddress(7, 0) << ", Destination IP = " << i.GetAddress(19, 0) << std::endl;

// Flow 2: Top-right to Bottom-left diagonal (4 to 20)
Ptr<Socket> source2 = Socket::CreateSocket(c.Get(4), tid);
InetSocketAddress remote2 = InetSocketAddress(i.GetAddress(20, 0), 80);
source2->Connect(remote2);
std::cout << "Flow 2: Source IP = " << i.GetAddress(4, 0) << ", Destination IP = " << i.GetAddress(20, 0) << std::endl;

// Flow 3: Middle row (5, 6, 7, 8, 9)
Ptr<Socket> source3 = Socket::CreateSocket(c.Get(5), tid);
InetSocketAddress remote3 = InetSocketAddress(i.GetAddress(9, 0), 80);
source3->Connect(remote3);
std::cout << "Flow 3: Source IP = " << i.GetAddress(5, 0) << ", Destination IP = " << i.GetAddress(9, 0) << std::endl;
```

Performance Evaluation:

- The high transmission rates help evaluate the impact of CSMA/CA on throughput and packet loss.
- The setup simulates realistic communication scenarios to assess network efficiency under heavy traffic.

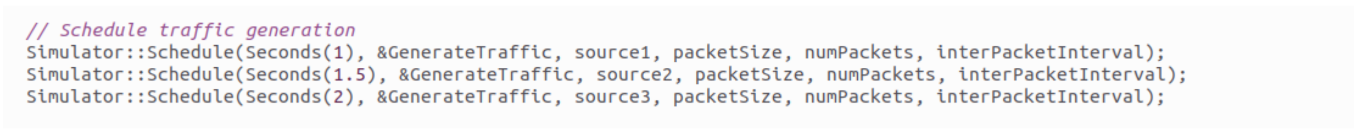
high transmission rate is simulated :



```
uint32_t sourceNode{24};
Time interPacketInterval{"0.01s"}; // High transmission rate
```

Scheduling:

- Staggered the initiation of each traffic flow to simulate a realistic start time:
    - Flow 1 started at 1 second.
    - Flow 2 started at 1.5 seconds.
    - Flow 3 started at 2 seconds.



```
// Schedule traffic generation
Simulator::Schedule(Seconds(1), &GenerateTraffic, source1, packetSize, numPackets, interPacketInterval);
Simulator::Schedule(Seconds(1.5), &GenerateTraffic, source2, packetSize, numPackets, interPacketInterval);
Simulator::Schedule(Seconds(2), &GenerateTraffic, source3, packetSize, numPackets, interPacketInterval);
```

Flow Monitoring:

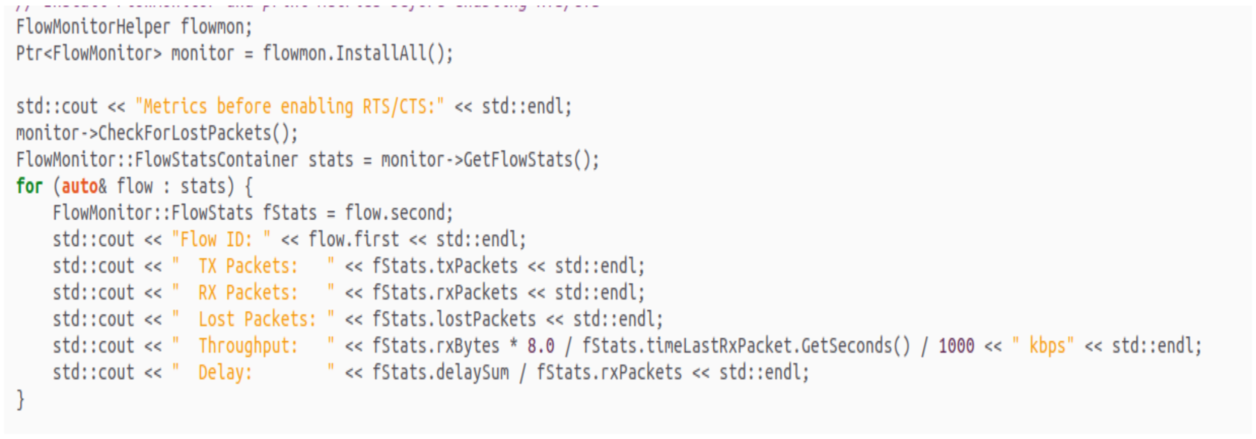
Three flows namely Flow1 ,Flow2 ,Flow 3 are used to simulate each diagonal flow of the grid and middle flow of the grid. They are clearly shown in the code below.

Data Collection:

- **Throughput Observation**: Used Flow Monitor to observe and record the throughput for each UDP flow, providing how efficiently the network is handling data transfer.
- **Packet Collisions and Drops**: Utilized the tracing mechanism to monitor packet collisions and drops at intermediary nodes, capturing any network congestion or failures.
- **Analysis:** Analyzed the collected data to identify which nodes experience the highest rates of packet collisions and drops, helping to pinpoint

network performance bottlenecks or issues in specific areas of the grid network.

ns-3 Flow Monitor is used to track the performance of each UDP flow



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

std::cout << "Metrics before enabling RTS/CTS:" << std::endl;
monitor->CheckForLostPackets();
FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats();
for (auto& flow : stats) {
    FlowMonitor::FlowStats fStats = flow.second;
    std::cout << "Flow ID: " << flow.first << std::endl;
    std::cout << "  TX Packets:   " << fStats.txPackets << std::endl;
    std::cout << "  RX Packets:   " << fStats.rxPackets << std::endl;
    std::cout << "  Lost Packets: " << fStats.lostPackets << std::endl;
    std::cout << "  Throughput:   " << fStats.rxBytes * 8.0 / fStats.timeLastRxPacket.GetSeconds() / 1000 << " kbps" << std::endl;
    std::cout << "  Delay:        " << fStats.delaySum / fStats.rxPackets << std::endl;
}
```

In **CSMA/CA** (Carrier Sense Multiple Access with Collision Avoidance), **RTS (Request to Send)** and **CTS (Clear to Send)** are control signals used to reduce collisions in wireless networks.

- **RTS (Request to Send):** When a device wants to transmit data, it first sends an RTS signal to indicate its intention and reserve the communication channel. This helps to alert other nearby devices of the pending transmission.
- **CTS (Clear to Send):** If the receiving device detects that the channel is free, it responds with a CTS signal, giving permission for the transmission to proceed. This helps prevent collisions by letting other devices know that they should wait before transmitting.

TO RUN THE CODE FOR PART-2 ,

This code is in the scratch folder so please go to that directory and run the commands.

Configure and build the code on ns-3 using the following commands:

./ns3 configure

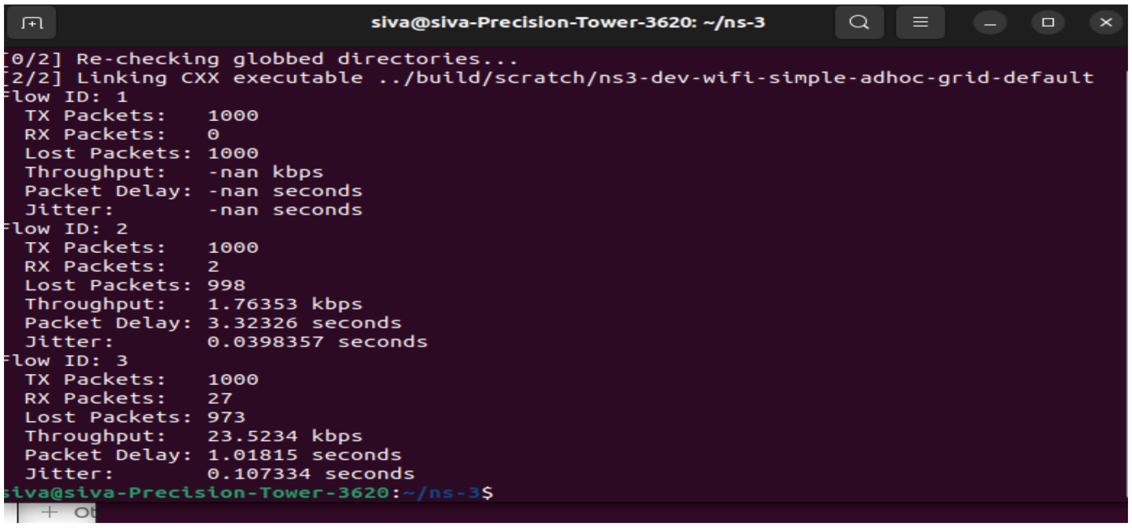
./ns3 build

The following command simulates the network grid without RTS/ CTS protocols.

./ns3 run wifi-simple-adhoc-grid

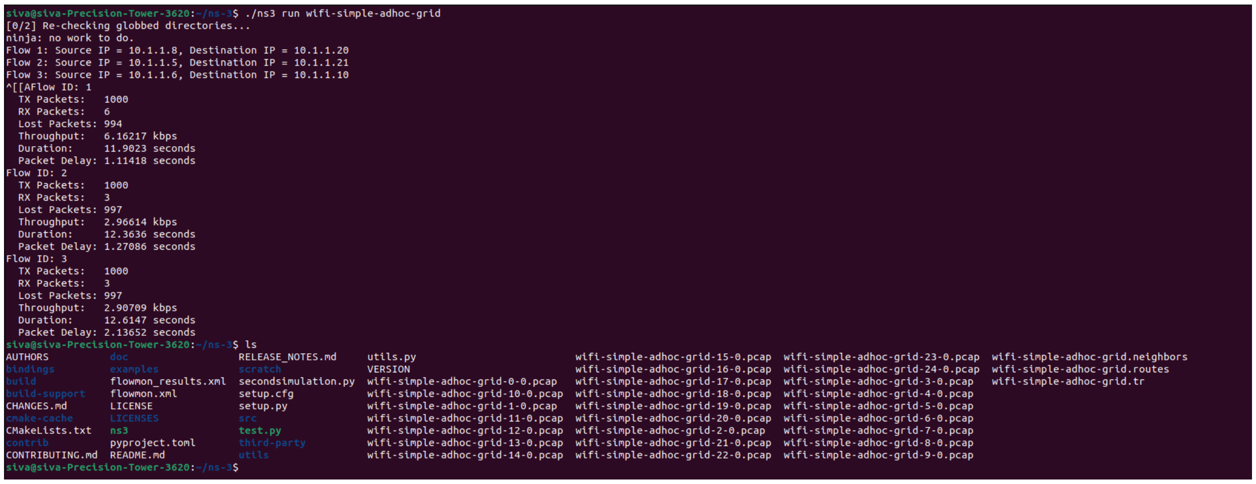
To run the code with RTS/CTS protocol run ./ns3 run wifi-hidden-terminal

Output before enabling RTS/CTS :



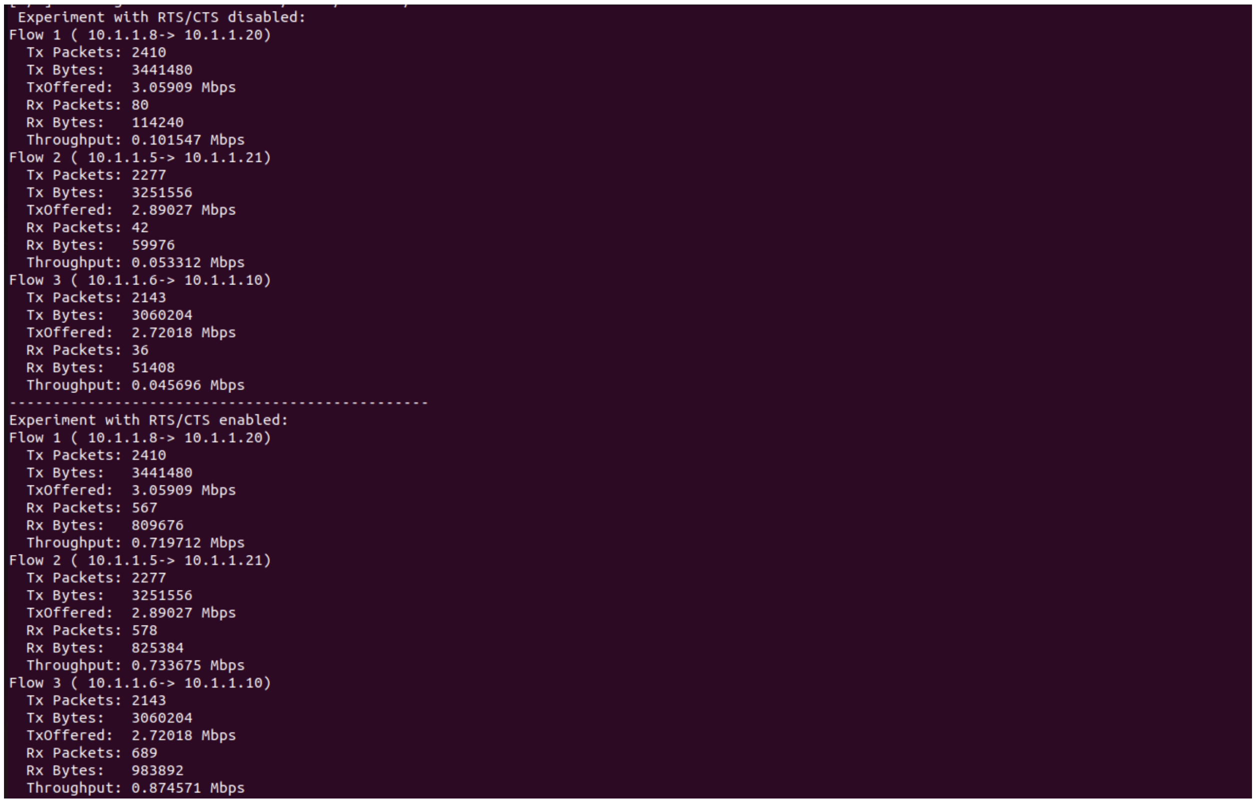
```
siva@siva-Precision-Tower-3620: ~/ns-3
[0/2] Re-checking globbed directories...
[2/2] Linking CXX executable ../build/scratch/ns3-dev-wifi-simple-adhoc-grid-default
Flow ID: 1
  TX Packets:    1000
  RX Packets:    0
  Lost Packets: 1000
  Throughput:    -nan kbps
  Packet Delay: -nan seconds
  Jitter:        -nan seconds
Flow ID: 2
  TX Packets:    1000
  RX Packets:    2
  Lost Packets: 998
  Throughput:    1.76353 kbps
  Packet Delay: 3.32326 seconds
  Jitter:        0.0398357 seconds
Flow ID: 3
  TX Packets:    1000
  RX Packets:    27
  Lost Packets: 973
  Throughput:    23.5234 kbps
  Packet Delay: 1.01815 seconds
  Jitter:        0.107334 seconds
siva@siva-Precision-Tower-3620:~/ns-3$
```

Below are the list of files that are formed after simulating traffic:



```
siva@siva-Precision-Tower-3620:~/ns-3$ ./ns3 run wifi-simple-adhoc-grid
[0/2] Re-checking globbed directories...
ninja: no work to do.
Flow 1: Source IP = 10.1.1.8, Destination IP = 10.1.1.20
Flow 2: Source IP = 10.1.1.5, Destination IP = 10.1.1.21
Flow 3: Source IP = 10.1.1.6, Destination IP = 10.1.1.10
^[[AFlow ID: 1
  TX Packets:   1000
  RX Packets:   6
  Lost Packets: 994
  Throughput:   6.16217 kbps
  Duration:     11.9023 seconds
  Packet Delay: 1.11418 seconds
Flow ID: 2
  TX Packets:   1000
  RX Packets:   3
  Lost Packets: 997
  Throughput:   2.96614 kbps
  Duration:     12.3636 seconds
  Packet Delay: 1.27086 seconds
Flow ID: 3
  TX Packets:   1000
  RX Packets:   3
  Lost Packets: 997
  Throughput:   2.90709 kbps
  Duration:     12.6147 seconds
  Packet Delay: 2.13652 seconds
siva@siva-Precision-Tower-3620:~/ns-3$ ls
AUTHORS          doc                 RELEASE_NOTES.md     utils.py                          wifi-simple-adhoc-grid-15-0.pcap  wifi-simple-adhoc-grid-23-0.pcap  wifi-simple-adhoc-grid.neighbors
bindings         examples            scratch              VERSION                           wifi-simple-adhoc-grid-16-0.pcap  wifi-simple-adhoc-grid-24-0.pcap  wifi-simple-adhoc-grid.routes
build            flowmon_results.xml secondsimulation.py  wifi-simple-adhoc-grid-0-0.pcap   wifi-simple-adhoc-grid-17-0.pcap  wifi-simple-adhoc-grid-3-0.pcap   wifi-simple-adhoc-grid.tr
build-support    flowmon.xml         setup.cfg            wifi-simple-adhoc-grid-10-0.pcap  wifi-simple-adhoc-grid-18-0.pcap  wifi-simple-adhoc-grid-4-0.pcap
CHANGES.md       LICENSE             setup.py             wifi-simple-adhoc-grid-1-0.pcap   wifi-simple-adhoc-grid-19-0.pcap  wifi-simple-adhoc-grid-5-0.pcap
cmake-cache      LICENSES            src                  wifi-simple-adhoc-grid-11-0.pcap  wifi-simple-adhoc-grid-20-0.pcap  wifi-simple-adhoc-grid-6-0.pcap
CMakeLists.txt   ns3                 test.py              wifi-simple-adhoc-grid-12-0.pcap  wifi-simple-adhoc-grid-2-0.pcap   wifi-simple-adhoc-grid-7-0.pcap
contrib          pyproject.toml      third-party          wifi-simple-adhoc-grid-13-0.pcap  wifi-simple-adhoc-grid-21-0.pcap  wifi-simple-adhoc-grid-8-0.pcap
CONTRIBUTING.md  README.md           utils                wifi-simple-adhoc-grid-14-0.pcap  wifi-simple-adhoc-grid-22-0.pcap  wifi-simple-adhoc-grid-9-0.pcap
siva@siva-Precision-Tower-3620:~/ns-3$
```

Comparison of Throughput and Packet Loss when RTS/CTS Disabled vs Enabled:



```
 Experiment with RTS/CTS disabled:
Flow 1 ( 10.1.1.8-> 10.1.1.20)
  Tx Packets: 2410
  Tx Bytes:   3441480
  TxOffered:  3.05909 Mbps
  Rx Packets: 80
  Rx Bytes:   114240
  Throughput: 0.101547 Mbps
Flow 2 ( 10.1.1.5-> 10.1.1.21)
  Tx Packets: 2277
  Tx Bytes:   3251556
  TxOffered:  2.89027 Mbps
  Rx Packets: 42
  Rx Bytes:   59976
  Throughput: 0.053312 Mbps
Flow 3 ( 10.1.1.6-> 10.1.1.10)
  Tx Packets: 2143
  Tx Bytes:   3060204
  TxOffered:  2.72018 Mbps
  Rx Packets: 36
  Rx Bytes:   51408
  Throughput: 0.045696 Mbps
-----------------------------------------------
Experiment with RTS/CTS enabled:
Flow 1 ( 10.1.1.8-> 10.1.1.20)
  Tx Packets: 2410
  Tx Bytes:   3441480
  TxOffered:  3.05909 Mbps
  Rx Packets: 567
  Rx Bytes:   809676
  Throughput: 0.719712 Mbps
Flow 2 ( 10.1.1.5-> 10.1.1.21)
  Tx Packets: 2277
  Tx Bytes:   3251556
  TxOffered:  2.89027 Mbps
  Rx Packets: 578
  Rx Bytes:   825384
  Throughput: 0.733675 Mbps
Flow 3 ( 10.1.1.6-> 10.1.1.10)
  Tx Packets: 2143
  Tx Bytes:   3060204
  TxOffered:  2.72018 Mbps
  Rx Packets: 689
  Rx Bytes:   983892
  Throughput: 0.874571 Mbps
```

Experiment 1: RTS/CTS Disabled

- **Flow 1** (10.1.1.8 -> 10.1.1.20):
  - Offered Throughput: 3.05909 Mbps
  - Received Throughput: 0.101547 Mbps
  - Packet Loss: High (indicating poor transmission efficiency)
- **Flow 2** (10.1.1.5 -> 10.1.1.21):
  - Offered Throughput: 2.89027 Mbps
  - Received Throughput: 0.053312 Mbps
  - Packet Loss: Significant (reflecting high transmission inefficiency)
- **Flow 3** (10.1.1.6 -> 10.1.1.10):
  - Offered Throughput: 2.72018 Mbps
  - Received Throughput: 0.045696 Mbps
  - Packet Loss: Major (indicating poor packet delivery)

Experiment 2: RTS/CTS Enabled

- **Flow 1** (10.1.1.8 -> 10.1.1.20):
  - Offered Throughput: 3.05909 Mbps
  - Received Throughput: 0.719712 Mbps

    - Packet Loss: Significantly reduced (indicating much better transmission success)
- **Flow 2** (10.1.1.5 -> 10.1.1.21):
    - Offered Throughput: 2.89027 Mbps
    - Received Throughput: 0.733675 Mbps
    - Packet Loss: Considerably lower (improved packet reception)
- **Flow 3** (10.1.1.6 -> 10.1.1.10):
    - Offered Throughput: 2.72018 Mbps
    - Received Throughput: 0.874571 Mbps
    - Packet Loss: Much lower (indicating improved delivery efficiency)

Key Observations:

1. **Throughput Gains:**
    - **Enabling RTS/CTS** consistently resulted in a dramatic **increase in throughput** across all flows. For instance, Flow 1's throughput improved from 0.101547 Mbps to 0.719712 Mbps, a substantial increase of over 7 times.
2. **Packet Loss Reduction:**
    - With RTS/CTS disabled, packet loss was notably high, as seen in the low received throughput values. However, when RTS/CTS was enabled, the **packet loss dropped significantly**, as indicated by the improved throughput figures in each flow.
3. **Flow-Specific Performance:**
    - **Flow 1:** The throughput improvement was substantial, with a rise from 0.101547 Mbps to 0.719712 Mbps.
    - **Flow 2:** A noticeable performance boost occurred, with throughput jumping from 0.053312 Mbps to 0.733675 Mbps.
    - **Flow 3:** Similarly, throughput increased from 0.045696 Mbps to 0.874571 Mbps, indicating a marked improvement in packet delivery.

Conclusion:

Enabling RTS/CTS has a clear positive impact on network performance. It significantly **improves throughput** and **reduces packet loss,** suggesting that RTS/CTS helps to **manage collisions and optimize data transmission** in environments with multiple competing flows. This results in much more **efficient network** behavior, especially in congested or high-interference settings.