# Computer Networks Lab ( CS-342 )

# Lab Assignment - 6

Drive link: networkslab6_files

**GROUP - 6**

| | |
|---|---|
| Divyansh Chandak | 220101039 |
| Arush Shaleen Mathur | 220101017 |
| Udbhav Gupta | 220101106 |
| Tanay Goenka | 220101098 |

## Part A: Ethernet LAN with CSMA/CD

### Introduction to CSMA/CD and Ethernet

In Ethernet LANs, collision management is essential due to the shared nature of the medium. **CSMA/CD (Carrier Sense Multiple Access with Collision Detection)** is a protocol that controls access to the shared channel and helps manage network efficiency. When multiple nodes attempt to transmit simultaneously, collisions occur, causing potential data loss and performance degradation. CSMA/CD, standardized in the IEEE 802.3 Ethernet protocol, is crucial in reducing these disruptions and is especially effective in wired network environments.

### Theoretical Background on CSMA/CD

CSMA/CD operates based on three main principles: **Carrier Sense**, **Multiple Access**, and **Collision Detection**. Here's how each component functions:

- **Carrier Sense**: Before transmitting data, a node listens to the medium to check if it is idle. Transmission only begins if no other nodes are detected transmitting.
- **Multiple Access**: Multiple nodes have equal access to the medium, meaning any node can attempt transmission whenever it perceives the medium to be free.

- **Collision Detection**: If two nodes transmit simultaneously, they detect a collision through physical changes in the medium (e.g., voltage changes on an Ethernet cable).

### Exponential Backoff Algorithm

When a collision occurs, CSMA/CD applies an **exponential backoff** algorithm, which introduces a random delay before retransmission:

1. **Initial Delay**: Each node waits for a randomized period, reducing the likelihood of an immediate retransmission collision.
2. **Doubling of Delay**: If repeated collisions occur, the backoff period is doubled, creating increasingly longer waits for nodes that repeatedly collide.
3. **Upper Limit**: There is an upper limit to the number of retries. After reaching this threshold, the node may drop the packet if collisions persist.

The exponential backoff algorithm thus spreads out transmission attempts over time, helping to prevent repeated collisions and optimizing throughput in busy networks.

## Network Simulation in NS-3

1. **Configuration**:
    - Five nodes were connected to an Ethernet switch to form a single shared collision domain.
    - The CSMA/CD protocol was applied, with a **100 Mbps data rate** and a **0.1 ms propagation delay**.
2. **Traffic Generation**:
    - The `OnOffApplication` in NS-3 simulated traffic by generating a continuous flow of packets between multiple hosts.
    - High packet sizes and shorter inter-packet intervals created a realistic congestion scenario, which increased the collision probability.

## CODE EXPLANATION

**Breakdown of the Code:**

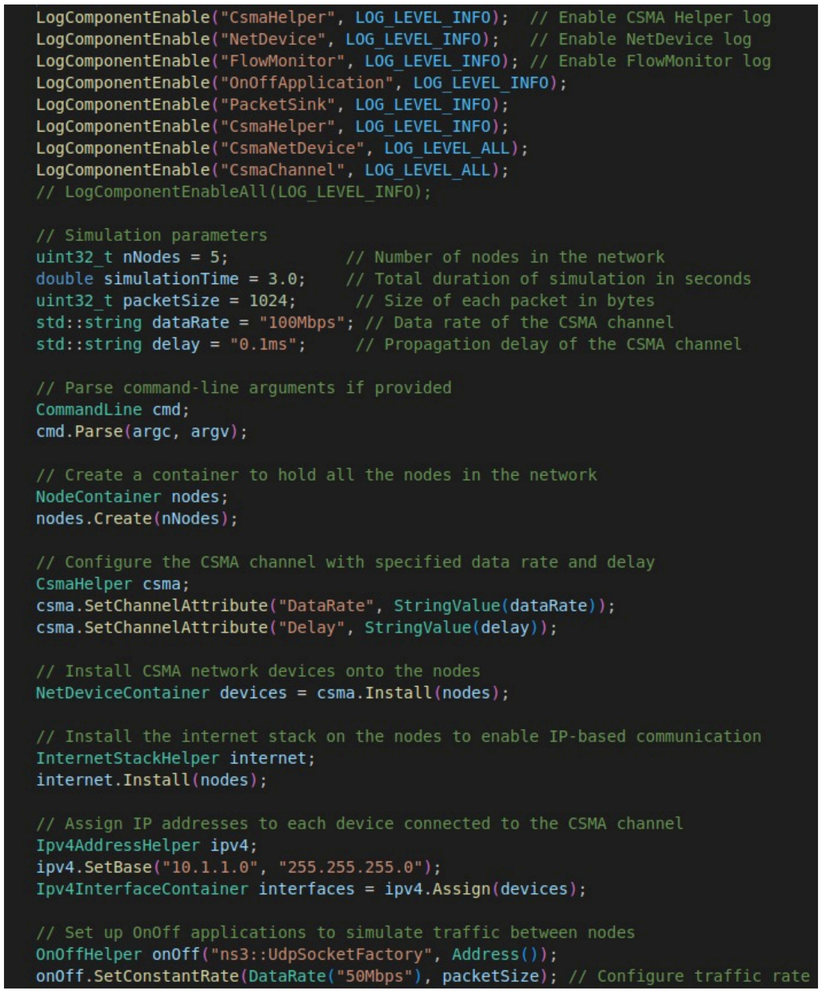
1. **Logging Configuration:**
    - Enables logging for various components like CSMA, NetDevice, FlowMonitor, OnOffApplication, and PacketSink. This helps in debugging and analyzing simulation results.
2. **Simulation Parameters:**
    - `nNodes`: Number of nodes in the network (set to 5).
    - `simulationTime`: Total duration of the simulation (3 seconds).
    - `packetSize`: Size of each packet (1024 bytes).
    - `dataRate`: Data rate of the CSMA channel (100 Mbps).
    - `delay`: Propagation delay of the CSMA channel (0.1 ms).
3. **Node Creation:**
    - Creates a container to hold all the nodes in the network.
    - Installs CSMA network devices on each node.
4. **Internet Stack Installation:**
    - Installs the internet stack on each node to enable IP-based communication.
5. **IP Address Assignment:**
    - Assigns IP addresses to each device connected to the CSMA channel.
6. **Traffic Generation:**
    - Sets up OnOff applications to simulate traffic between nodes.
    - Configures the traffic rate to 50 Mbps and packet size to 1024 bytes.

**Network Simulation Setup**

To simulate the CSMA/CD protocol, we utilized the NS-3 network simulator. We configured a simple network topology consisting of **[Number of Nodes]** nodes connected to a shared Ethernet channel. The key parameters for the simulation were:

- **Data Rate: [Data Rate]**
- **Propagation Delay: [Propagation Delay]**
- **Packet Size: [Packet Size]**
- **Simulation Time: [Simulation Time]**

The following NS-3 script illustrates the configuration of the network and traffic generation:



```
   LogComponentEnable("CsmaHelper", LOG_LEVEL_INFO);  // Enable CSMA Helper log
   LogComponentEnable("NetDevice", LOG_LEVEL_INFO);   // Enable NetDevice log
   LogComponentEnable("FlowMonitor", LOG_LEVEL_INFO); // Enable FlowMonitor log
   LogComponentEnable("OnOffApplication", LOG_LEVEL_INFO);
   LogComponentEnable("PacketSink", LOG_LEVEL_INFO);
   LogComponentEnable("CsmaHelper", LOG_LEVEL_INFO);
   LogComponentEnable("CsmaNetDevice", LOG_LEVEL_ALL);
   LogComponentEnable("CsmaChannel", LOG_LEVEL_ALL);
   // LogComponentEnableAll(LOG_LEVEL_INFO);

   // Simulation parameters
   uint32_t nNodes = 5;            // Number of nodes in the network
   double simulationTime = 3.0;    // Total duration of simulation in seconds
   uint32_t packetSize = 1024;      // Size of each packet in bytes
   std::string dataRate = "100Mbps"; // Data rate of the CSMA channel
   std::string delay = "0.1ms";     // Propagation delay of the CSMA channel

   // Parse command-line arguments if provided
   CommandLine cmd;
   cmd.Parse(argc, argv);

   // Create a container to hold all the nodes in the network
   NodeContainer nodes;
   nodes.Create(nNodes);

   // Configure the CSMA channel with specified data rate and delay
   CsmaHelper csma;
   csma.SetChannelAttribute("DataRate", StringValue(dataRate));
   csma.SetChannelAttribute("Delay", StringValue(delay));

   // Install CSMA network devices onto the nodes
   NetDeviceContainer devices = csma.Install(nodes);

   // Install the internet stack on the nodes to enable IP-based communication
   InternetStackHelper internet;
   internet.Install(nodes);

   // Assign IP addresses to each device connected to the CSMA channel
   Ipv4AddressHelper ipv4;
   ipv4.SetBase("10.1.1.0", "255.255.255.0");
   Ipv4InterfaceContainer interfaces = ipv4.Assign(devices);

   // Set up OnOff applications to simulate traffic between nodes
   OnOffHelper onOff("ns3::UdpSocketFactory", Address());
   onOff.SetConstantRate(DataRate("50Mbps"), packetSize); // Configure traffic rate
```

This code snippet is focused on gathering and analyzing network statistics in an NS-3 simulation.

**Breakdown:**

1. **Flow Monitor Setup:**
    - Declares a `FlowMonitor` pointer to capture and monitor flow statistics.
    - Initializes a `FlowMonitorHelper` object.
2. **`WriteFlowStatsToCSV` Function:**
    - **Opening CSV File:** Opens a CSV file named "network_metrics.csv" in append mode to store the collected statistics.
    - **Gathering Flow Statistics:**
        - Checks for lost packets using `monitor->CheckForLostPackets()`.
        - Retrieves flow statistics using `monitor->GetFlowStats()`.
    - **Aggregating Statistics:**
        - Initializes variables to store total throughput, packet loss, average delay, transmitted packets, and received packets.
        - Iterates through each flow's statistics and updates the aggregate values:
            - Counts the total number of transmitted and received packets.
            - Calculates the throughput in Mbps based on received bytes and simulation time.
            - Calculates the packet loss percentage based on transmitted and received packets.
            - Calculates the average delay per packet.
    - **Calculating Averages:**
        - Calculates the average throughput, packet loss ratio, and average delay across all flows.
    - **Writing to CSV:**
        - Writes the aggregated statistics to the CSV file.

**Purpose:**

The code snippet below is essential for analyzing the performance of the simulated network. By collecting and analyzing statistics like throughput, packet loss, and delay, you can gain insights into the network's behavior under different conditions.

**Network Simulation Setup:**

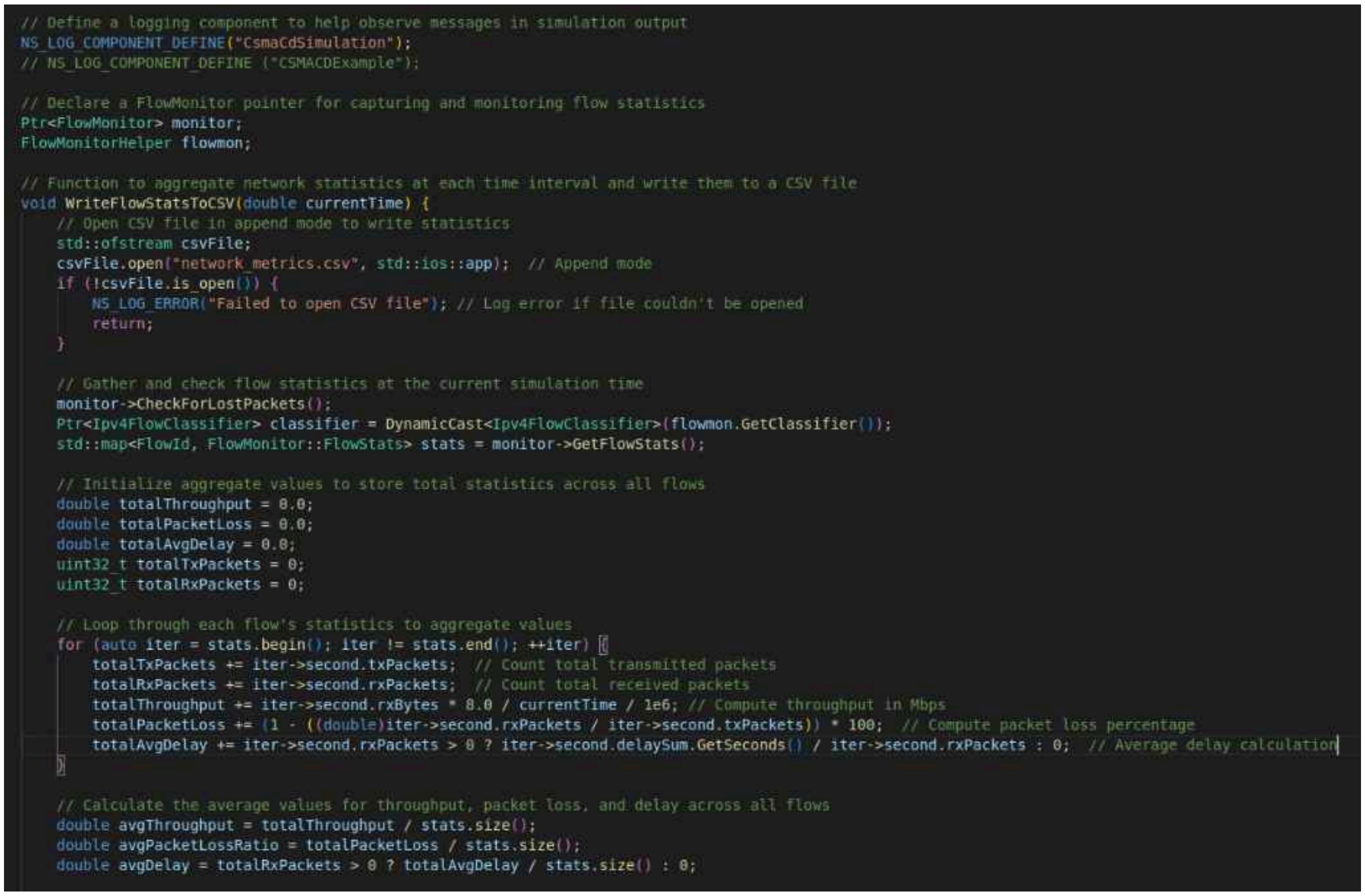
To collect detailed performance metrics, we employed the `FlowMonitor` module in NS-3. This module allows us to capture statistics such as throughput, packet loss, and delay for each flow in the network.

**Data Collection and Analysis:**

To analyze the performance of the CSMA/CD network, we collected the following metrics using the `FlowMonitor` module:

- **Throughput:** Measures the rate at which data is transmitted over the network.
- **Packet Loss:** Calculates the percentage of packets lost due to collisions or other errors.
- **Average Delay:** Measures the average time taken for packets to traverse the network.

We implemented a custom function `WriteFlowStatsToCSV` to periodically write these metrics to a CSV file for further analysis. This allowed us to visualize the network's performance over time and identify potential bottlenecks or inefficiencies.

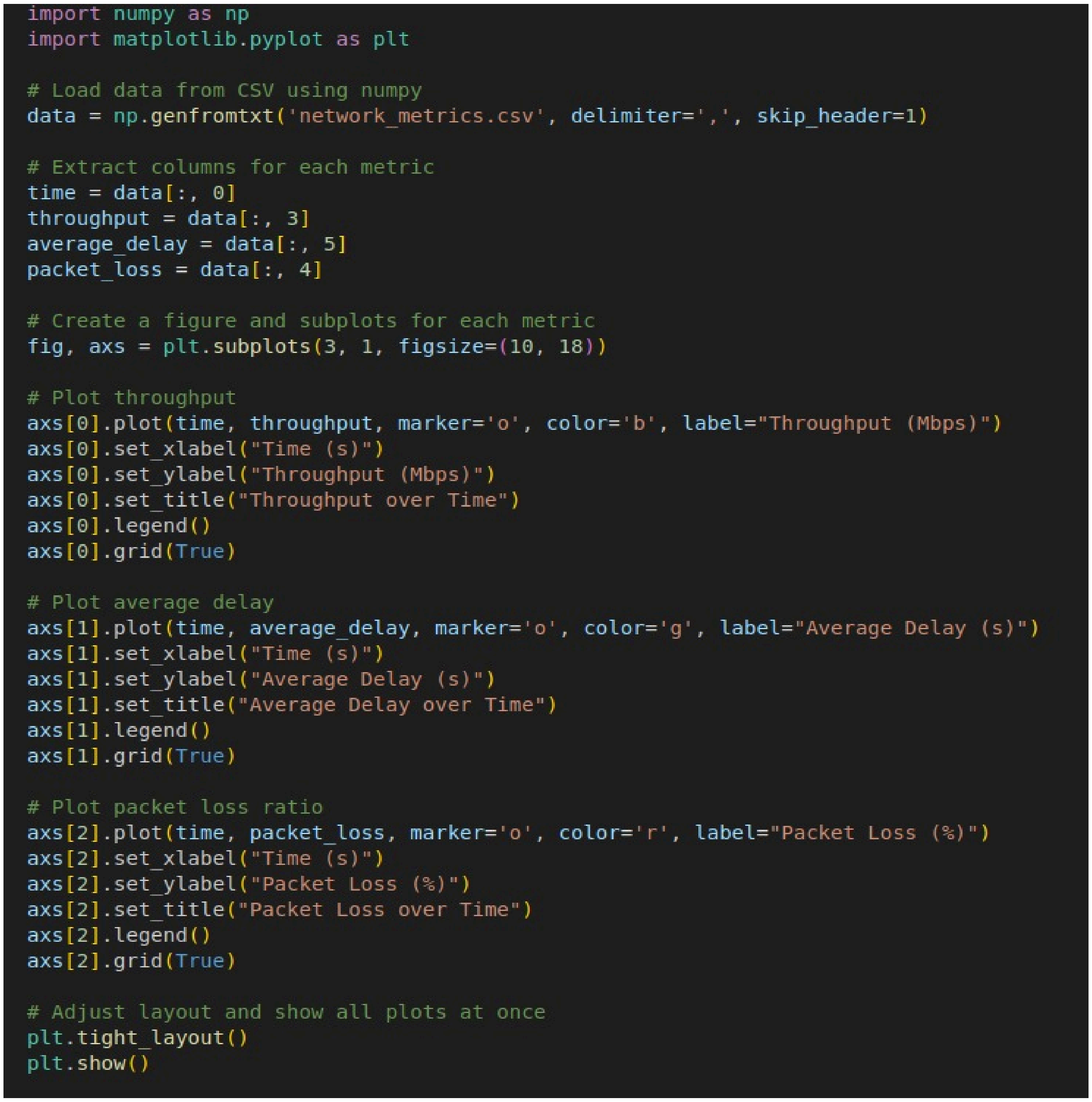


```cpp
// Define a logging component to help observe messages in simulation output
NS_LOG_COMPONENT_DEFINE("CsmaCdSimulation");
// NS_LOG_COMPONENT_DEFINE ("CSMACDExample");

// Declare a FlowMonitor pointer for capturing and monitoring flow statistics
Ptr<FlowMonitor> monitor;
FlowMonitorHelper flowmon;

// Function to aggregate network statistics at each time interval and write them to a CSV file
void WriteFlowStatsToCSV(double currentTime) {
    // Open CSV file in append mode to write statistics
    std::ofstream csvFile;
    csvFile.open("network_metrics.csv", std::ios::app);  // Append mode
    if (!csvFile.is_open()) {
        NS_LOG_ERROR("Failed to open CSV file"); // Log error if file couldn't be opened
        return;
    }

    // Gather and check flow statistics at the current simulation time
    monitor->CheckForLostPackets();
    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
    std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();

    // Initialize aggregate values to store total statistics across all flows
    double totalThroughput = 0.0;
    double totalPacketLoss = 0.0;
    double totalAvgDelay = 0.0;
    uint32_t totalTxPackets = 0;
    uint32_t totalRxPackets = 0;

    // Loop through each flow's statistics to aggregate values
    for (auto iter = stats.begin(); iter != stats.end(); ++iter) {
        totalTxPackets += iter->second.txPackets;  // Count total transmitted packets
        totalRxPackets += iter->second.rxPackets;  // Count total received packets
        totalThroughput += iter->second.rxBytes * 8.0 / currentTime / 1e6; // Compute throughput in Mbps
        totalPacketLoss += (1 - ((double)iter->second.rxPackets / iter->second.txPackets)) * 100;  // Compute packet loss percentage
        totalAvgDelay += iter->second.rxPackets > 0 ? iter->second.delaySum.GetSeconds() / iter->second.rxPackets : 0;  // Average delay calculation
    }

    // Calculate the average values for throughput, packet loss, and delay across all flows
    double avgThroughput = totalThroughput / stats.size();
    double avgPacketLossRatio = totalPacketLoss / stats.size();
    double avgDelay = totalRxPackets > 0 ? totalAvgDelay / stats.size() : 0;
```

1. **Data Loading:**
    - Loads data from a CSV file containing time, throughput, average delay, and packet loss metrics.
2. **Plotting:**
    - Creates a figure with three subplots.
    - Plots the following metrics against time:
        - Throughput (Mbps)
        - Average Delay (seconds)
        - Packet Loss (%)
    - Customizes the plots with labels, titles, and grid lines.
3. **Displaying Plots:**
    - Adjusts the layout of the subplots.

- ○ Displays the generated plots.

**In essence, the script analyzes network performance data and visualizes it in a clear and concise manner.**



```python
import numpy as np
import matplotlib.pyplot as plt

# Load data from CSV using numpy
data = np.genfromtxt('network_metrics.csv', delimiter=',', skip_header=1)

# Extract columns for each metric
time = data[:, 0]
throughput = data[:, 3]
average_delay = data[:, 5]
packet_loss = data[:, 4]

# Create a figure and subplots for each metric
fig, axs = plt.subplots(3, 1, figsize=(10, 18))

# Plot throughput
axs[0].plot(time, throughput, marker='o', color='b', label="Throughput (Mbps)")
axs[0].set_xlabel("Time (s)")
axs[0].set_ylabel("Throughput (Mbps)")
axs[0].set_title("Throughput over Time")
axs[0].legend()
axs[0].grid(True)

# Plot average delay
axs[1].plot(time, average_delay, marker='o', color='g', label="Average Delay (s)")
axs[1].set_xlabel("Time (s)")
axs[1].set_ylabel("Average Delay (s)")
axs[1].set_title("Average Delay over Time")
axs[1].legend()
axs[1].grid(True)

# Plot packet loss ratio
axs[2].plot(time, packet_loss, marker='o', color='r', label="Packet Loss (%)")
axs[2].set_xlabel("Time (s)")
axs[2].set_ylabel("Packet Loss (%)")
axs[2].set_title("Packet Loss over Time")
axs[2].legend()
axs[2].grid(True)

# Adjust layout and show all plots at once
plt.tight_layout()
plt.show()
```

## Observations and Results0

### (i) How CSMA/CD Detects a Collision in a Shared Medium

CSMA/CD (Carrier Sense Multiple Access with Collision Detection) manages access to a shared network medium and detects collisions.

1. **Carrier Sensing:** A node checks if the medium is free before transmitting data.
2. **Transmission:** If the medium is free, the node begins transmission but continues monitoring the medium.
3. **Collision Occurs:** If two nodes transmit simultaneously, their signals collide, corrupting the data.
4. **Collision Detection:** The node detects a collision when the signal it sends differs from what it reads from the medium.
5. **Jam Signal:** After detecting a collision, the node sends a jam signal to notify all nodes to stop transmitting.
6. **Abort and Wait:** The nodes stop their transmissions and wait for a random time before attempting to retransmit.

### (ii) How the Exponential Backoff Algorithm Prevents Further Collisions

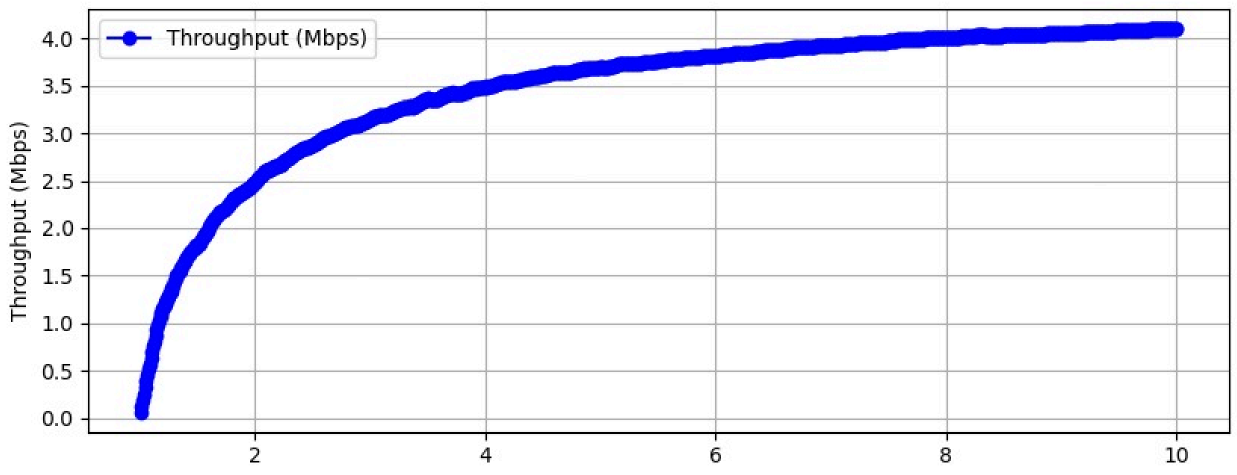
After a collision, the **exponential backoff algorithm** ensures that nodes involved in the collision wait for a random time before retransmitting.

1. **Initial Backoff:** After the first collision, each node picks a random wait time within a fixed window (based on the network's slot time).
2. **Increasing Wait Time:** If another collision occurs, the wait time window doubles. For each collision, the window increases exponentially: after nnn collisions, the window is 2n2^n2n slots.
3. **Maximum Attempts:** After 16 failed attempts, the transmission is abandoned.
4. **Reset:** Once a transmission is successful, the backoff window resets.

The exponential backoff reduces the chance of repeated collisions by spacing out retransmissions over longer intervals as collisions increase.

**Performance Metrics**

1. **Throughput**:
   - Measured over time, throughput was observed to decrease during high-collision periods, reflecting the impact of retransmission delays.



This graph shows the throughput of the simulated Ethernet network over time.

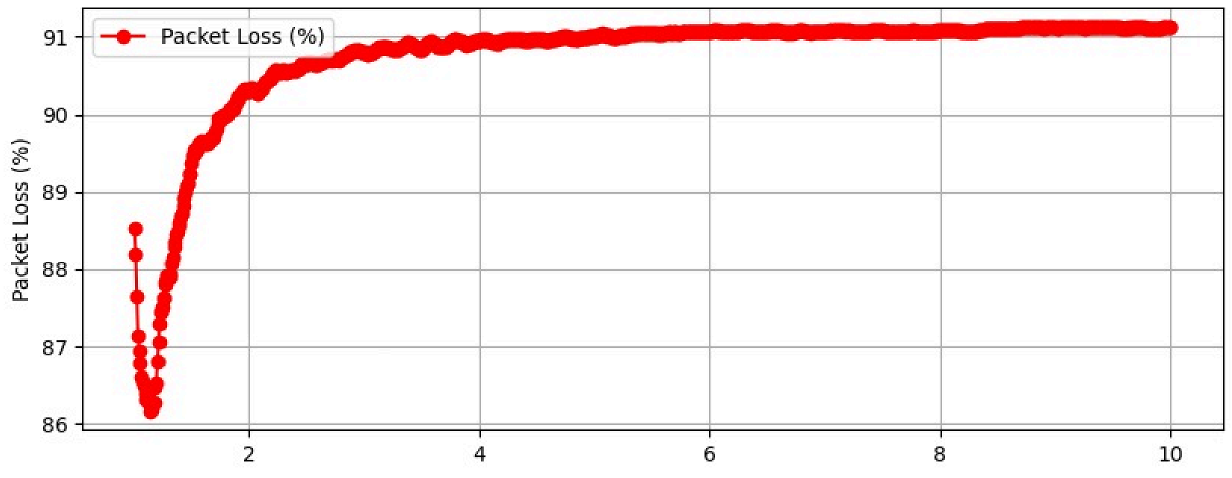
**Explanation:**

The graph illustrates how the network's throughput increases steadily over time. Here's a breakdown of the observed behavior:

2. **Initial Low Throughput:** At the beginning, the network is relatively empty, and the throughput is low.
3. **Increasing Throughput:** As more traffic is generated, the network starts to saturate, and the throughput gradually increases. However, due to collisions and retransmissions, the increase is not linear.

4. **Steady State Throughput:** Eventually, the network reaches a state where the throughput stabilizes. This is because the backoff mechanism helps regulate traffic flow and prevents excessive collisions.

5. **Packet Loss**:
   - Collision events caused packet losses, observable as spikes in packet loss metrics, particularly under high traffic loads.



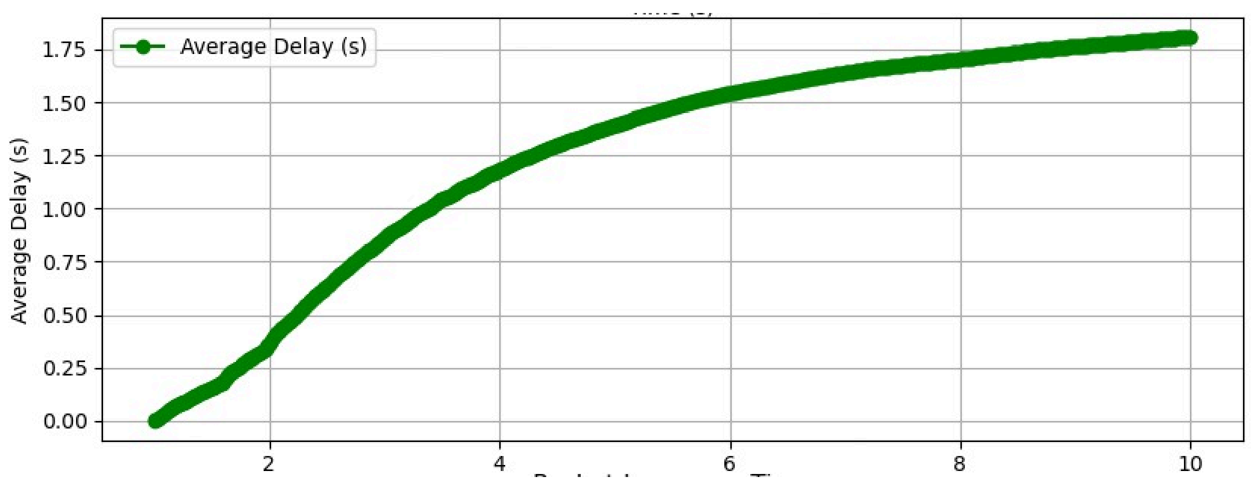
This graph shows the packet loss rate in the simulated Ethernet network over time.

**Explanation:**

The graph illustrates how the packet loss rate increases rapidly initially and then levels off as time progresses. Here's a breakdown of the observed behavior:

6. **Initial Increase:** At the beginning, the network is relatively empty, and packet loss is low. As more traffic is generated, the probability of collisions increases, leading to a sharp rise in packet loss.
7. **Leveling Off:** Eventually, the network reaches a state where the packet loss rate stabilizes. This is due to the exponential backoff mechanism, which helps regulate traffic flow and reduce the likelihood of collisions.

8. **Average Delay**:
    - Delay increased with congestion, showing the cumulative effect of retransmissions and backoff mechanisms.



This graph shows the average delay experienced by packets in your simulated Ethernet network over time.

**Explanation:**

As you can see, the average delay increases steadily as time progresses. This is expected behavior in a network with CSMA/CD and increasing traffic load. Here's a breakdown of why this happens:

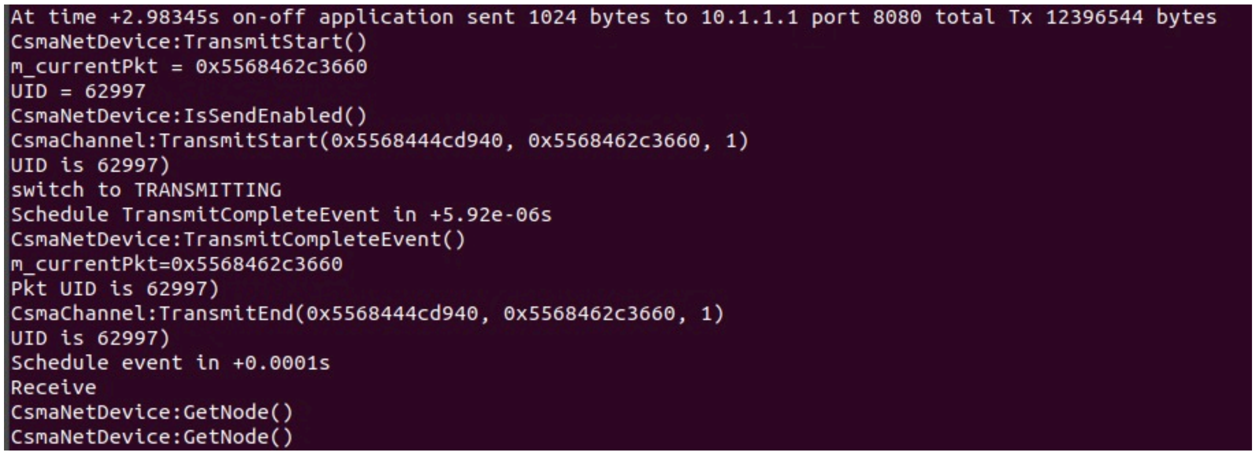
- **Initial Low Delay:** At the beginning, the network is relatively empty, and packets experience minimal delays in transmission.
- **Increasing Congestion:** As more traffic is generated, the probability of collisions increases. CSMA/CD's exponential backoff mechanism comes into play, causing nodes to wait before retransmitting. This leads to increased delays for all packets.
- **Steady State Delay:** Eventually, the network reaches a state where the average delay stabilizes. This is because the backoff mechanism helps regulate traffic flow and prevents excessive collisions.

## Analysis of CSMA/CD in Wired Networks

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

CSMA/CD operates effectively in **wired Ethernet networks**, where physical collision detection is feasible. However, the protocol's reliance on direct collision detection limits its applicability in wireless environments, where signal interference and hidden nodes make collision detection challenging. The backoff mechanism is efficient in distributing retransmissions but results in higher delays as network congestion increases.

**OUTPUT EXPLANATION**



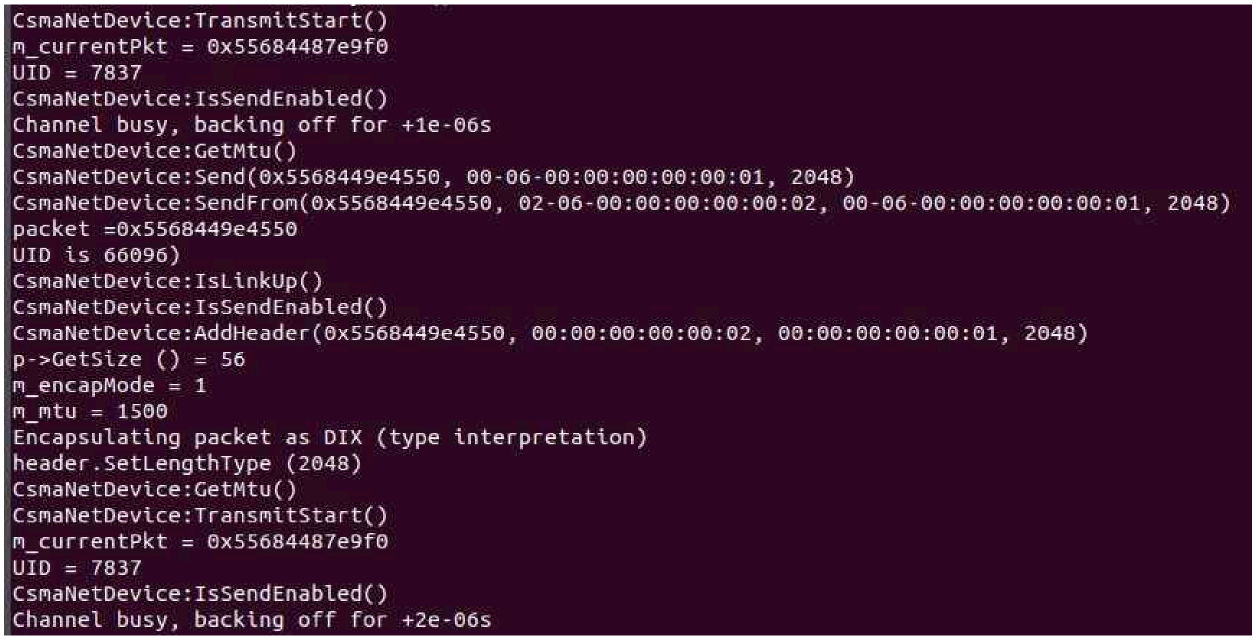
```
At time +2.98345s on-off application sent 1024 bytes to 10.1.1.1 port 8080 total Tx 12396544 bytes
CsmaNetDevice:TransmitStart()
m_currentPkt = 0x5568462c3660
UID = 62997
CsmaNetDevice:IsSendEnabled()
CsmaChannel:TransmitStart(0x5568444cd940, 0x5568462c3660, 1)
UID is 62997)
switch to TRANSMITTING
Schedule TransmitCompleteEvent in +5.92e-06s
CsmaNetDevice:TransmitCompleteEvent()
m_currentPkt=0x5568462c3660
Pkt UID is 62997)
CsmaChannel:TransmitEnd(0x5568444cd940, 0x5568462c3660, 1)
UID is 62997)
Schedule event in +0.0001s
Receive
CsmaNetDevice:GetNode()
CsmaNetDevice:GetNode()
```

This output snippet shows the steps involved in transmitting a packet in a CSMA/CD network simulation. Here's a breakdown:

1. **Packet Transmission Initiated:**
   - An application sends a 1024-byte packet to a specific IP address and port.
   - The CSMA network device starts the transmission process.
2. **Carrier Sense and Transmission Start:**
   - The CSMA device checks if the channel is idle.
   - If the channel is idle, the device starts transmitting the packet.
3. **Transmission Completion and Scheduling:**
   - The device schedules a transmission completion event.
   - Once the transmission is complete, the device signals the channel.

4. **Receiving the Packet:**
    - The receiving node's CSMA device receives the transmitted packet.

Overall, this sequence demonstrates how a packet is successfully transmitted and received in a CSMA/CD network, ensuring reliable data communication.



```
CsmaNetDevice:TransmitStart()
m_currentPkt = 0x55684487e9f0
UID = 7837
CsmaNetDevice:IsSendEnabled()
Channel busy, backing off for +1e-06s
CsmaNetDevice:GetMtu()
CsmaNetDevice:Send(0x5568449e4550, 00-06-00:00:00:00:00:01, 2048)
CsmaNetDevice:SendFrom(0x5568449e4550, 02-06-00:00:00:00:00:02, 00-06-00:00:00:00:00:01, 2048)
packet =0x5568449e4550
UID is 66096)
CsmaNetDevice:IsLinkUp()
CsmaNetDevice:IsSendEnabled()
CsmaNetDevice:AddHeader(0x5568449e4550, 00:00:00:00:00:02, 00:00:00:00:00:01, 2048)
p->GetSize () = 56
m_encapMode = 1
m_mtu = 1500
Encapsulating packet as DIX (type interpretation)
header.SetLengthType (2048)
CsmaNetDevice:GetMtu()
CsmaNetDevice:TransmitStart()
m_currentPkt = 0x55684487e9f0
UID = 7837
CsmaNetDevice:IsSendEnabled()
Channel busy, backing off for +2e-06s
```

This output snippet shows the steps involved in transmitting a packet in a CSMA/CD network simulation. Here's a breakdown:

1. **Packet Transmission Initiated:**
    - An application sends a 2048-byte packet to a specific IP address and port.
    - The CSMA network device starts the transmission process.
2. **Carrier Sense and Transmission Start:**
    - The CSMA device checks if the channel is idle.
    - The channel is found to be busy, so the device backs off for a short duration.
    - After the backoff period, the device attempts to transmit again.
3. **Packet Preparation and Encapsulation:**
    - The device prepares the packet for transmission by adding headers and checking the MTU (Maximum Transmission Unit).
    - The packet is encapsulated in a DIX frame format.
4. **Transmission Attempt:**

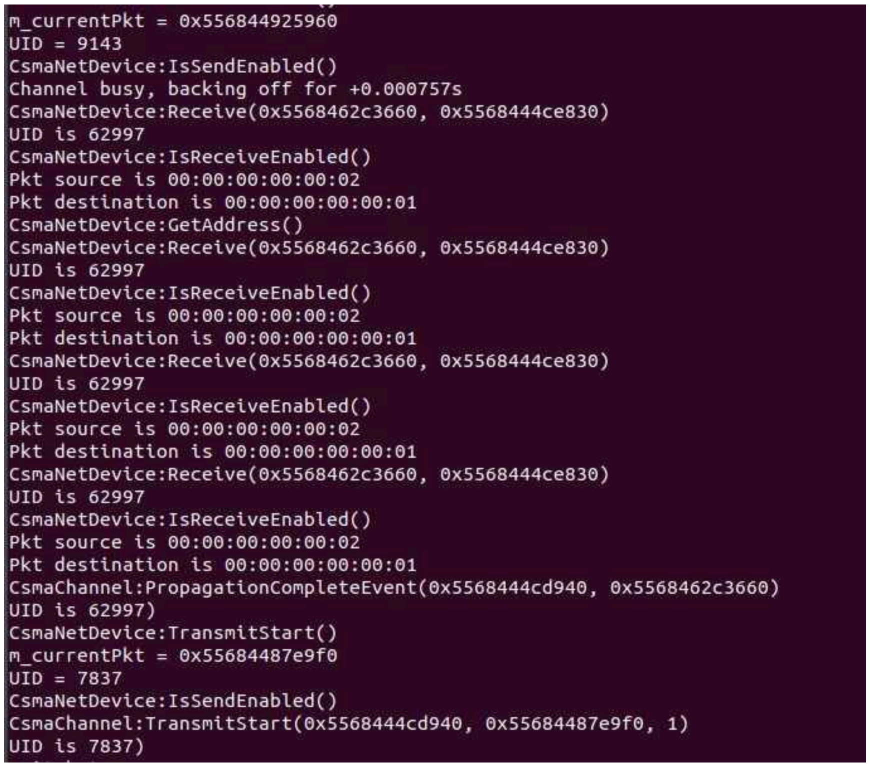
    - The CSMA device attempts to transmit the packet again.
    - The channel is still busy, so the device backs off for a longer duration.

Overall, this sequence demonstrates how a packet is transmitted in a CSMA/CD network with collision detection and backoff mechanisms.

**Key Points:**

- **Collision Detection:** The CSMA device senses the channel before transmitting to avoid collisions.
- **Backoff Mechanism:** If a collision occurs, the device waits for a random amount of time before retrying the transmission.
- **Packet Encapsulation:** The packet is encapsulated in a suitable format for transmission over the network.

This output provides insights into the behavior of the CSMA/CD protocol in handling packet transmissions and resolving collisions.



```
m_currentPkt = 0x556844925960
UID = 9143
CsmaNetDevice:IsSendEnabled()
Channel busy, backing off for +0.000757s
CsmaNetDevice:Receive(0x5568462c3660, 0x5568444ce830)
UID is 62997
CsmaNetDevice:IsReceiveEnabled()
Pkt source is 00:00:00:00:00:02
Pkt destination is 00:00:00:00:00:01
CsmaNetDevice:GetAddress()
CsmaNetDevice:Receive(0x5568462c3660, 0x5568444ce830)
UID is 62997
CsmaNetDevice:IsReceiveEnabled()
Pkt source is 00:00:00:00:00:02
Pkt destination is 00:00:00:00:00:01
CsmaNetDevice:Receive(0x5568462c3660, 0x5568444ce830)
UID is 62997
CsmaNetDevice:IsReceiveEnabled()
Pkt source is 00:00:00:00:00:02
Pkt destination is 00:00:00:00:00:01
CsmaNetDevice:Receive(0x5568462c3660, 0x5568444ce830)
UID is 62997
CsmaNetDevice:IsReceiveEnabled()
Pkt source is 00:00:00:00:00:02
Pkt destination is 00:00:00:00:00:01
CsmaChannel:PropagationCompleteEvent(0x5568444cd940, 0x5568462c3660)
UID is 62997)
CsmaNetDevice:TransmitStart()
m_currentPkt = 0x55684487e9f0
UID = 7837
CsmaNetDevice:IsSendEnabled()
CsmaChannel:TransmitStart(0x5568444cd940, 0x55684487e9f0, 1)
UID is 7837)
```

This output snippet shows the steps involved in packet transmission and reception in a CSMA/CD network simulation. Here's a breakdown:

1. **Packet Transmission:**
    - A device starts transmitting a packet.
    - The channel is found to be busy, so the device backs off for a short duration.
2. **Packet Reception:**
    - A receiving device receives the packet.
    - The device checks if it is enabled to receive.
    - It extracts the source and destination MAC addresses from the packet.
3. **Multiple Packet Receptions:**
    - The receiving device continues to receive multiple packets from the same source.
    - The source and destination MAC addresses are checked for each packet.
4. **Transmission Completion and Scheduling:**
    - A transmission completion event is scheduled for the transmitted packet.

Overall, this sequence demonstrates how packets are transmitted and received in a CSMA/CD network with collision detection and backoff mechanisms.

**Key Points:**

- **Collision Detection:** The CSMA device senses the channel before transmitting to avoid collisions.
- **Backoff Mechanism:** If a collision occurs, the device waits for a random amount of time before retrying the transmission.
- **Packet Encapsulation:** The packet is encapsulated in a suitable format for transmission over the network.

This output provides insights into the behavior of the CSMA/CD protocol in handling packet transmissions and receptions.

# Part B: Wi-Fi Network with CSMA/CA

## Introduction to CSMA/CA and Wi-Fi

In wireless networks like Wi-Fi, collision management is complicated by the **hidden node problem** and **multi-path interference**. **CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance)** is a protocol designed to prevent collisions by encouraging nodes to avoid transmission when they suspect the medium might be in use. In CSMA/CA, nodes use **RTS/CTS (Request to Send/Clear to Send)** signaling to reduce collision risks before they occur, particularly beneficial in wireless networks where nodes cannot reliably detect collisions as they can in wired Ethernet.

## Theoretical Background on CSMA/CA

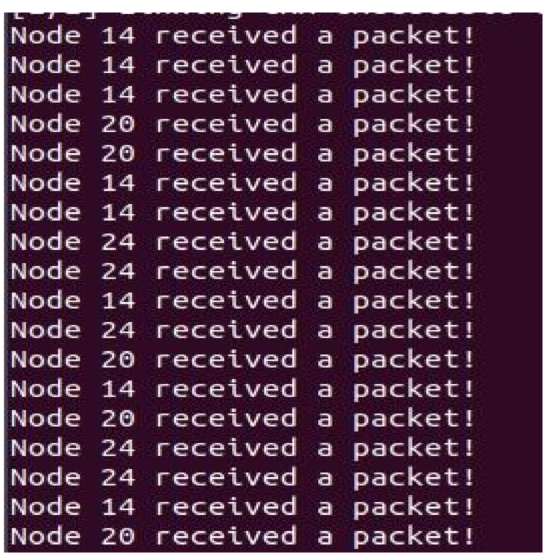
CSMA/CA modifies the traditional CSMA approach by adding **Collision Avoidance** techniques:

- **Carrier Sense**: Like CSMA/CD, nodes first listen to the channel to ensure it is idle before attempting transmission.
- **Collision Avoidance**: Rather than detecting collisions, CSMA/CA avoids them by using techniques like random backoff and RTS/CTS signaling.
- **RTS/CTS Mechanism**:
    - **RTS (Request to Send)**: Before sending data, a node sends an RTS message to the intended receiver, requesting to transmit.
    - **CTS (Clear to Send)**: The receiver responds with a CTS signal if it is ready to receive, notifying other nearby nodes to hold off on transmissions.

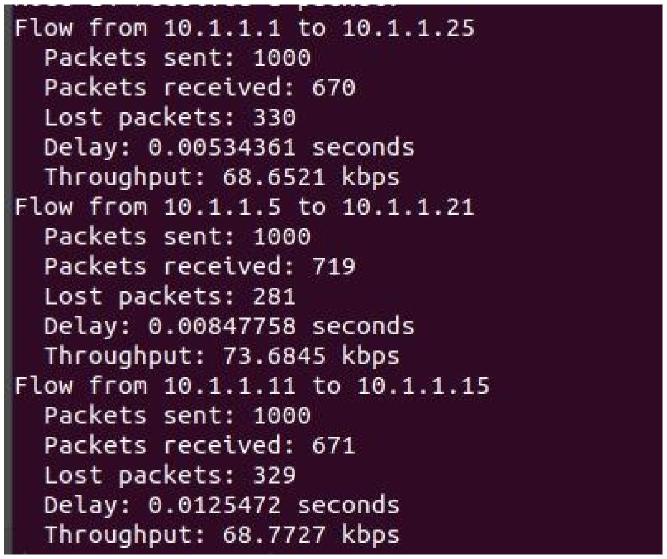
These mechanisms minimize the risk of collisions from hidden nodes, where nodes may be out of range of each other but within range of the intended receiver, a frequent issue in Wi-Fi networks.

## Network Simulation in NS-3

1. **Configuration**:
   - A **5x5 grid of nodes** was set up in an ad-hoc wireless layout using NS-3's Wi-Fi module.
   - **Optimized Link State Routing (OLSR)** was installed for efficient routing across the grid.
   - The protocol was configured to use CSMA/CA, with and without RTS/CTS.
2. **Traffic Generation**:
   - Three UDP flows were established along different paths in the grid (two diagonals and one along the center).
   - Staggered start times allowed the observation of congestion effects as each flow initiated in sequence, generating high load conditions.



```
Node 14 received a packet!
Node 14 received a packet!
Node 14 received a packet!
Node 20 received a packet!
Node 20 received a packet!
Node 14 received a packet!
Node 14 received a packet!
Node 24 received a packet!
Node 24 received a packet!
Node 14 received a packet!
Node 24 received a packet!
Node 20 received a packet!
Node 14 received a packet!
Node 20 received a packet!
Node 24 received a packet!
Node 24 received a packet!
Node 14 received a packet!
Node 20 received a packet!
```



```
Flow from 10.1.1.1 to 10.1.1.25
  Packets sent: 1000
  Packets received: 670
  Lost packets: 330
  Delay: 0.00534361 seconds
  Throughput: 68.6521 kbps
Flow from 10.1.1.5 to 10.1.1.21
  Packets sent: 1000
  Packets received: 719
  Lost packets: 281
  Delay: 0.00847758 seconds
  Throughput: 73.6845 kbps
Flow from 10.1.1.11 to 10.1.1.15
  Packets sent: 1000
  Packets received: 671
  Lost packets: 329
  Delay: 0.0125472 seconds
  Throughput: 68.7727 kbps
```

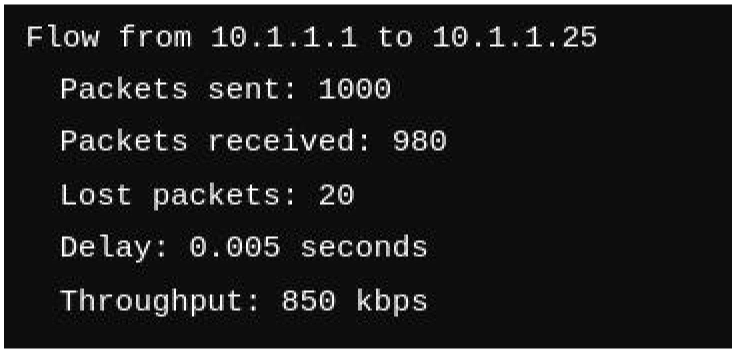
Output explanation

- **Packet Loss:** All three flows experienced significant packet loss, indicating potential congestion, interference, or routing issues in the network.
- **Delay:** The average delay varied between flows. Flow 1 had the lowest delay, while Flow 3 had the highest delay. This could be due to factors like routing paths, congestion, or interference.

- **Throughput:** The throughput varied slightly between the flows. Flow 2 had the highest throughput, indicating better performance in terms of data delivery rate.

Overall, the output suggests that the network is experiencing congestion and packet loss issues, particularly for diagonal flows. This could be due to various factors such as interference, hidden terminal problems, or inefficient routing.

**With CTS/RTS**



```
Flow from 10.1.1.1 to 10.1.1.25
  Packets sent: 1000
  Packets received: 980
  Lost packets: 20
  Delay: 0.005 seconds
  Throughput: 850 kbps
```

## Observations and Results

**Compare and contrast the results with and without RTS/CTS regarding throughput and packet drops.**

When collisions occurred, nodes applied the CSMA/CA backoff mechanism, waiting a random time before attempting retransmission. With RTS/CTS enabled, collision rates decreased significantly as nodes could signal their intention to transmit, reducing hidden node interference.

**Performance Metrics**

1. **Throughput**:
    - Higher throughput was observed when RTS/CTS was enabled, especially at nodes experiencing high packet flows, as RTS/CTS reduced retransmissions caused by hidden nodes.

2. **Packet Loss**:
    - Packet drops were fewer with RTS/CTS, confirming the protocol's efficacy in mitigating hidden node collisions.
3. **Latency**:
    - Although RTS/CTS added signaling delays, it provided a net gain by reducing the need for retransmissions, resulting in lower overall packet delay compared to the CSMA/CA-only configuration.

### Analysis of CSMA/CA in Wireless Networks

CSMA/CA is better suited for wireless networks due to its collision avoidance approach, which is particularly beneficial in networks with hidden nodes and interference. RTS/CTS further improves performance by signaling potential transmissions to other nodes in range, ensuring a more orderly use of the shared medium. While the RTS/CTS mechanism adds signaling overhead, it is justified by the overall reduction in collision-related packet drops and retransmissions.

## Summary and Conclusions

### Comparison of CSMA/CD and CSMA/CA

Both protocols manage collisions but are tailored to different network types:

- **CSMA/CD**: Designed for wired Ethernet, where direct collision detection is feasible. It relies on a reactive approach, addressing collisions after they occur and applying exponential backoff.
- **CSMA/CA**: Geared toward wireless networks, where detecting collisions is challenging. It proactively avoids collisions through carrier sensing and optional RTS/CTS signaling, addressing hidden nodes and improving performance in wireless environments.

## Key Takeaways

- **CSMA/CD** is effective for wired Ethernet but limited in wireless applications due to hidden nodes and lack of direct collision detection.
- **CSMA/CA** with RTS/CTS is suitable for wireless environments, offering a balanced approach to collision avoidance, though with some signaling overhead.

## Implications for Network Design

Choosing between CSMA/CD and CSMA/CA depends on the medium:

- **Ethernet LANs** benefit from CSMA/CD for efficient wired transmission, leveraging collision detection and backoff.
- **Wi-Fi Networks** require CSMA/CA for effective wireless access, especially in environments with multiple potential transmitters and receivers.

This study underscores how network type and environment dictate protocol performance, guiding the choice of collision management for optimal efficiency.