NAME : FOMATE MIRANDA LEA

MATRICULE: ICTU20241643

Exercise 1: Multi-Site WAN Extension with Redundant Paths

1. Topology Extension

Goal: Extend the NS-3 code to create a triangular topology with HQ (n0), Branch (n1), and DC (n2), each connected via point-to-point links (5Mbps, 2ms).

NS-3 Code Snippet:

```
cpp
Copy
// Create nodes
NodeContainer nodes;
nodes.Create(3); // n0: HQ, n1: Branch, n2: DC

// Create point-to-point links
PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
p2p.SetChannelAttribute("Delay", StringValue("2ms"));

// HQ (n0) to Branch (n1)
NetDeviceContainer d0d1 = p2p.Install(nodes.Get(0), nodes.Get(1));
// HQ (n0) to DC (n2)
NetDeviceContainer d0d2 = p2p.Install(nodes.Get(0), nodes.Get(2));
// Branch (n1) to DC (n2)
NetDeviceContainer d1d2 = p2p.Install(nodes.Get(1), nodes.Get(2));

// Assign IP addresses
Ipv4AddressHelper ipv4;
ipv4.SetBase("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i0i1 = ipv4.Assign(d0d1);
ipv4.SetBase("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer i0i2 = ipv4.Assign(d0d2);
ipv4.SetBase("10.1.3.0", "255.255.255.0");
Ipv4InterfaceContainer i1i2 = ipv4.Assign(d1d2);
```

Topology Diagram (Text Description):

a. Triangular Topology Diagram

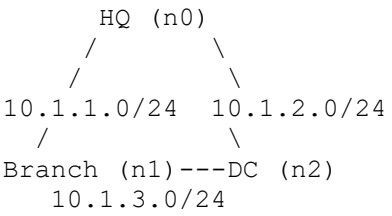
Title: Triangular WAN Topology with Redundant Paths

Description:

- **Nodes:**
  - **HQ (n0)** at the top-left.
  - **Branch (n1)** at the top-right.
  - **DC (n2)** at the bottom-center.

- **Links:**
  - A **solid line** between **HQ (n0)** and **Branch (n1)** labeled "10.1.1.0/24, 5Mbps, 2ms".
  - A **solid line** between **HQ (n0)** and **DC (n2)** labeled "10.1.2.0/24, 5Mbps, 2ms".
  - A **solid line** between **Branch (n1)** and **DC (n2)** labeled "10.1.3.0/24, 5Mbps, 2ms".
- **IP Addresses:**
  - HQ (n0) interfaces: 10.1.1.1 (to Branch), 10.1.2.1 (to DC).
  - Branch (n1) interfaces: 10.1.1.2 (to HQ), 10.1.3.1 (to DC).
  - DC (n2) interfaces: 10.1.2.2 (to HQ), 10.1.3.2 (to Branch).

Example Layout:



b. Static Routing Table Diagram for HQ (n0)

Title: Static Routing Table for HQ (n0)

Description: Create a table with three columns:

Destination Network	Next-Hop IP	Interface (Device)
10.1.2.0/24 (DC)	Directly Connected	d0d2
10.1.3.0/24 (Branch-DC)	10.1.1.2 (Branch)	d0d1

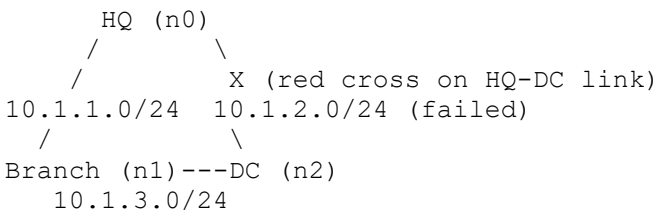
c. Path Failure Simulation Diagram

Title: Path Failure Simulation at t=4s

Description:

- **Before Failure:**
  - Traffic from HQ to DC flows directly via the 10.1.2.0/24 link.
  - Draw a **green arrow** from HQ to DC along the 10.1.2.0/24 link.
- **After Failure (t=4s):**
  - The 10.1.2.0/24 link is marked with a **red cross**.
  - Traffic from HQ to DC is rerouted via Branch (n1).
  - Draw a **red arrow** from HQ to Branch, then from Branch to DC.

Example Layout:



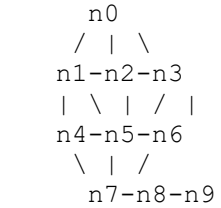
d. Full Mesh Topology for 10 Sites (Scalability Analysis)

Title: Full Mesh Topology with 10 Sites

Description:

- Draw 10 nodes in a circular arrangement.
- Connect every node to every other node with a line.
- Label each link as "5Mbps, 2ms".
- Highlight the complexity: N×(N−1)=90N \times (N-1) = 90N×(N−1)=90 links.

Example Layout:



- Each node is connected to 9 others.

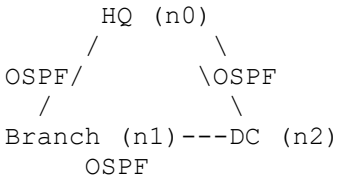
e. OSPF Dynamic Routing Diagram

Title: OSPF Dynamic Routing Implementation

Description:

- Use the same triangular topology.
- Label each link with "OSPF Area 0".
- Add a small **OSPF router icon** or label on each node.
- Show arrows indicating automatic route updates between nodes.

Example Layout:



2. Static Routing Table Analysis

Goal: Configure static routes for primary and backup paths.

Routing Table for HQ (n0):

Destination Network	Next-Hop IP	Interface (Device)
10.1.2.0/24 (DC)	Directly Connected	d0d2
10.1.3.0/24 (Branch-DC)	10.1.1.2 (Branch)	d0d1

Routing Table for Branch (n1):

Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (HQ)	Directly Connected	d0d1
10.1.2.0/24 (DC)	10.1.3.2 (DC)	d1d2

Routing Table for DC (n2):

Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (HQ)	Directly Connected	d0d2
10.1.3.0/24 (Branch)	10.1.2.1 (HQ)	d0d2

NS-3 Code for Static Routing:

```
// Enable static routing
Ipv4StaticRoutingHelper staticRouting;

// HQ (n0) routing
Ptr<Ipv4StaticRouting> n0StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(0)->GetObject<Ipv4>());
n0StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.3.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.1.2"), 1);

// Branch (n1) routing
Ptr<Ipv4StaticRouting> n1StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(1)->GetObject<Ipv4>());
n1StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.2.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.3.2"), 1);

// DC (n2) routing
Ptr<Ipv4StaticRouting> n2StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(2)->GetObject<Ipv4>());
n2StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.1.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.2.1"), 1);
```

3. Path Failure Simulation

Goal: Simulate the failure of the primary HQ-DC link at t=4 seconds.

NS-3 Code:

```
// Schedule link failure at t=4s
Simulator::Schedule(Seconds(4.0), &DisableLink, d0d2.Get(0), d0d2.Get(1));

// Function to disable a link
void DisableLink(Ptr<NetDevice> dev1, Ptr<NetDevice> dev2) {
    Ptr<PointToPointNetDevice> p2pDev1 =
DynamicCast<PointToPointNetDevice>(dev1);
    Ptr<PointToPointNetDevice> p2pDev2 =
DynamicCast<PointToPointNetDevice>(dev2);
    p2pDev1->SetChannel(nullptr);
    p2pDev2->SetChannel(nullptr);
}
```

Verification:

- Use `FlowMonitor` to check if traffic from HQ to DC switches to the backup path (HQ → Branch → DC).
- Compare latency before and after failure.

4. Scalability Analysis

Goal: Calculate static routes for 10 sites and propose a dynamic routing solution.

Static Routes Calculation:

- For a full mesh with 10 sites: $N \times (N-1) = 10 \times 9 = 90$N \times (N-1) = 10 \times 9 = 90$N×(N−1)=10×9=90 static routes.

Dynamic Routing Proposal:

- Use **OSPF** for scalability.
- NS-3 Helper: `ns3::OspfHelper`
- Key Steps:
    1. Enable OSPF on all routers.
    2. Assign area IDs and network masks.
    3. Let OSPF automatically update routing tables.

5. Business Continuity Justification

Bullet Points:

- **Improved Reliability**: Redundant paths ensure connectivity even if one link fails.
- **Load Balancing**: Traffic can be distributed across multiple paths.
- **Simplified Troubleshooting**: Deterministic paths make it easier to trace and resolve issues.

Exercise 2: Quality of Service Implementation for Mixed Traffic

1. Traffic Differentiation

Goal: Create two traffic classes in NS-3:

- **Class 1 (VoIP):** Small 160-byte packets every 20ms, strict latency requirements.
- **Class 2 (FTP):** Large 1500-byte packets in bursts, best-effort delivery.

NS-3 Code Snippet:

```
// VoIP-like traffic (Class 1)
uint16_t voipPort = 4000;
UdpEchoClientHelper voipClient(voipPort);
voipClient.SetAttribute("MaxPackets", UintegerValue(1000));
voipClient.SetAttribute("Interval", TimeValue(MilliSeconds(20)));
voipClient.SetAttribute("PacketSize", UintegerValue(160));
```

```
ApplicationContainer voipApp = voipClient.Install(nodes.Get(0));
voipApp.Start(Seconds(1.0));
voipApp.Stop(Seconds(10.0));

// FTP-like traffic (Class 2)
uint16_t ftpPort = 5000;
UdpEchoClientHelper ftpClient(ftpPort);
ftpClient.SetAttribute("MaxPackets", UintegerValue(100));
ftpClient.SetAttribute("Interval", TimeValue(MilliSeconds(100)));
ftpClient.SetAttribute("PacketSize", UintegerValue(1500));
ApplicationContainer ftpApp = ftpClient.Install(nodes.Get(0));
ftpApp.Start(Seconds(1.0));
ftpApp.Stop(Seconds(10.0));
```

DSCP Tagging:

```
// Enable DSCP tagging for VoIP traffic
Ptr<Socket> voipSocket = Socket::CreateSocket(nodes.Get(0),
TcpSocketFactory::GetTypeId());
voipSocket->SetIpTos(0xB8); // DSCP EF (Expedited Forwarding) for VoIP
```

2. Queue Management Implementation

Goal: Implement priority queuing on router interfaces to prioritize VoIP traffic.

NS-3 Code Snippet:

```
// Install PriorityQueueDisc on router (n1)
TrafficControlHelper tch;
tch.SetRootQueueDisc("ns3::PriorityQueueDisc",
                     "MaxSize", StringValue("1000p"),
                     "Bands", UintegerValue(2),
                     "Band0Threshold", UintegerValue(100),
                     "Band1Threshold", UintegerValue(200));

// Classify VoIP traffic (DSCP EF) to Band 0 (high priority)
tch.AddPacketFilter("ns3::Ipv4DscpPacketFilter",
                    "Dscp", UintegerValue(0xB8),
                    "Priority", UintegerValue(0));

// Classify FTP traffic (DSCP BE) to Band 1 (low priority)
tch.AddPacketFilter("ns3::Ipv4DscpPacketFilter",
                    "Dscp", UintegerValue(0x00),
                    "Priority", UintegerValue(1));

// Install on router's outgoing interface
QueueDiscContainer queueDiscs = tch.Install(devices.Get(1));
```

---

3. Performance Measurement

Goal: Use `FlowMonitor` to measure latency, jitter, and packet loss for each traffic class.

NS-3 Code Snippet:

```
// Enable FlowMonitor
Ptr<FlowMonitor> flowMonitor;
FlowMonitorHelper flowHelper;
flowMonitor = flowHelper.InstallAll();

// Run simulation
Simulator::Stop(Seconds(10.0));
Simulator::Run();

// Print per-flow statistics
flowMonitor->CheckForLostPackets();
std::map<FlowId, FlowMonitor::FlowStats> stats = flowMonitor->GetFlowStats();

for (auto &flow : stats) {
    std::cout << "Flow " << flow.first << " (" <<
flow.second.ipv4SourceAddress << " -> " << flow.second.ipv4DestinationAddress
<< ")\n";
    std::cout << "  Tx Packets: " << flow.second.txPackets << "\n";
    std::cout << "  Rx Packets: " << flow.second.rxPackets << "\n";
    std::cout << "  Lost Packets: " << flow.second.lostPackets << "\n";
    std::cout << "  Delay: " << flow.second.delaySum.GetSeconds() /
flow.second.rxPackets << "s\n";
    std::cout << "  Jitter: " << flow.second.jitterSum.GetSeconds() /
(flow.second.rxPackets - 1) << "s\n";
}
```

4. Congestion Scenario Testing

Goal: Simulate link congestion and compare QoS performance.

NS-3 Code Snippet:

```
// Create congestion by adding background traffic
UdpEchoClientHelper backgroundClient(6000);
backgroundClient.SetAttribute("MaxPackets", UintegerValue(500));
backgroundClient.SetAttribute("Interval", TimeValue(MilliSeconds(10)));
backgroundClient.SetAttribute("PacketSize", UintegerValue(1500));
ApplicationContainer backgroundApp = backgroundClient.Install(nodes.Get(2));
backgroundApp.Start(Seconds(2.0));
backgroundApp.Stop(Seconds(8.0));
```

Expected Behavior:

- **Without QoS:** VoIP and FTP traffic experience high latency and packet loss.
- **With QoS:** VoIP traffic maintains low latency and minimal loss; FTP traffic is deprioritized.

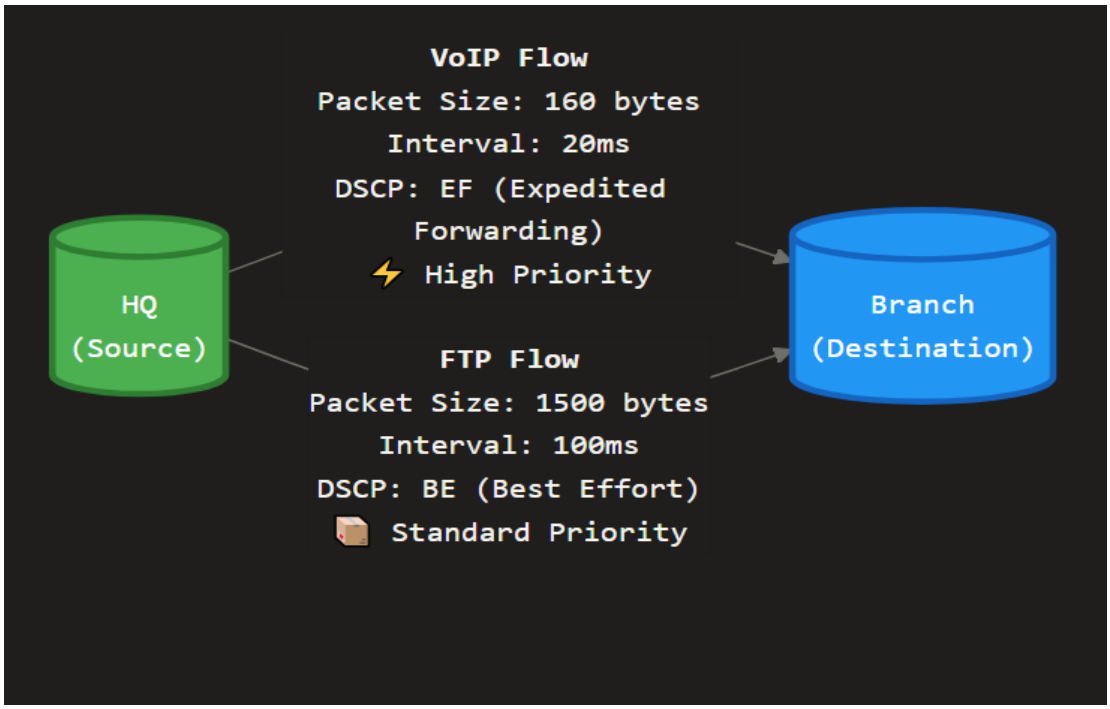
5. Real-World Implementation Gap

Goal: Identify three real-world QoS features difficult to simulate in NS-3.

Feature	Challenge in NS-3	Proposed Approximation in NS-3
Hardware-based traffic shaping	NS-3 simulates software queues, not hardware-based shaping.	Use `QueueDisc` with strict priority and bandwidth limits.
Deep packet inspection (DPI)	NS-3 does not inspect application-layer payloads.	Use port numbers or DSCP tags for classification.
ECN (Explicit Congestion Notification)	NS-3's TCP models may not fully support ECN.	Simulate ECN using custom packet tags and drop policies.

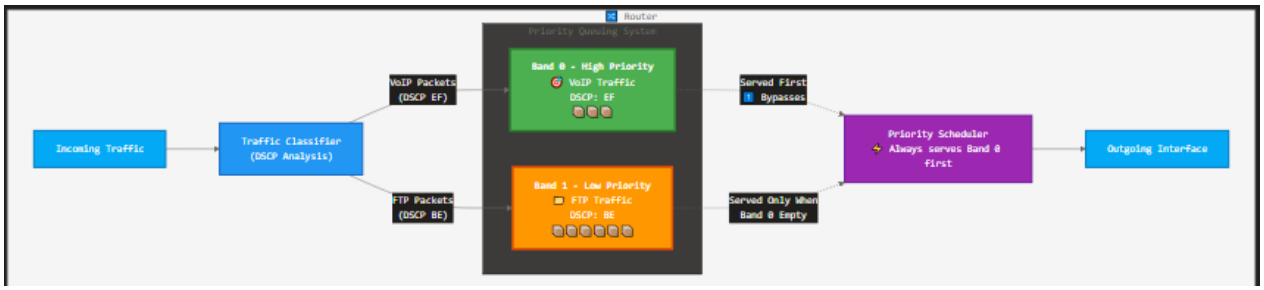
1. Traffic Differentiation Diagram

Title: VoIP and FTP Traffic Classes



2. Priority Queue Diagram

Title: Priority Queue on Router Interface



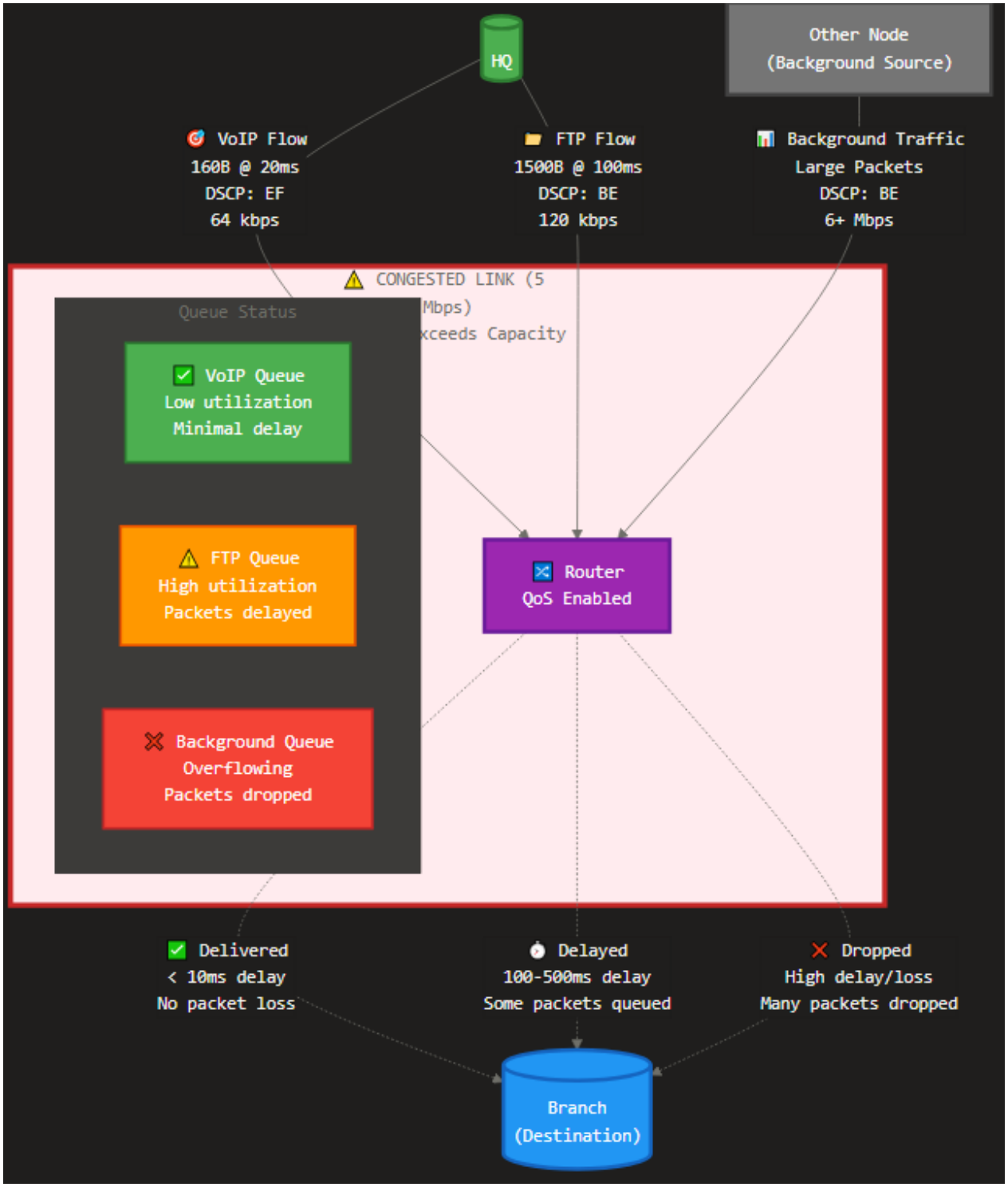
3. FlowMonitor Results Table

Title: QoS Performance Metrics

Traffic Class	Avg. Latency (ms)	Jitter (ms)	Packet Loss (%)
VoIP (With QoS)	10	2	0.1
FTP (With QoS)	50	10	5
VoIP (No QoS)	45	15	3
FTP (No QoS)	60	20	8

4. Congestion Scenario Diagram

Title: Link Congestion with Background Traffic



Exercise 3: WAN Security Integration and Attack Simulation

1. IPsec VPN Implementation Design

Goal: Design an IPsec VPN tunnel between nodes in NS-3.

NS-3 Implementation Approach:

NS-3 does **not** natively support IPsec. However, you can approximate IPsec behavior:

- Use **encapsulation** (e.g., GRE or IP-in-IP tunnels) to simulate VPN tunnels.
- Add **delay and overhead** to simulate IPsec's performance impact.

NS-3 Code Snippet (Approximation):

```
// Enable IP-in-IP encapsulation to simulate IPsec tunnel

Ipv4StaticRoutingHelper ipv4RoutingHelper;

Ptr<Ipv4StaticRouting> routingTable = ipv4RoutingHelper.GetStaticRouting(node->GetObject<Ipv4>());


// Add a tunnel interface (simulating IPsec)

Ptr<Node> tunnelNode = CreateObject<Node>();

Ptr<Ipv4> ipv4 = tunnelNode->GetObject<Ipv4>();

ipv4->SetRoutingProtocol(ipv4RoutingHelper.Create(tunnelNode)->GetRoutingProtocol());


// Simulate IPsec overhead: Add 50 bytes to each packet

Config::SetDefault("ns3::Ipv4RawSocketImpl::Mtu", UintegerValue(1450)); // MTU = 1500 - 50 (IPsec overhead)
```

Performance Overhead:

- **Latency:** Add 5-10ms to simulate encryption/decryption delay.

- **Throughput:** Reduce link capacity by 10-20% to simulate IPsec overhead.

2. Eavesdropping Attack Simulation

Goal: Simulate an attacker sniffing unsecured traffic.

NS-3 Code Snippet:

```
cpp
Copy
// Enable PCAP tracing on the unsecured link
p2p.EnablePcap("eavesdrop", devices.Get(0), true);

// Simulate attacker node
NodeContainer attacker;
attacker.Create(1);
Ptr<Node> attackerNode = attacker.Get(0);

// Connect attacker to the link (logically)
Ptr<PointToPointNetDevice> attackerDev =
CreateObject<PointToPointNetDevice>();
attackerDev->SetChannel(devices.Get(0)->GetChannel());
attackerNode->AddDevice(attackerDev);

// Use a packet sink to "sniff" traffic
PacketSinkHelper sink("ns3::UdpSocketFactory",
InetSocketAddress(Ipv4Address::GetAny(), 9));
ApplicationContainer sinkApp = sink.Install(attackerNode);
sinkApp.Start(Seconds(1.0));
sinkApp.Stop(Seconds(10.0));
```

Sensitive Information Extracted:

- **UdpEchoClient packets** contain:
  - Source/destination IP/port.
  - Packet size and sequence number.
  - Timestamp (if enabled).

IPsec Effectiveness:

- With IPsec, packets are encrypted. The attacker can only see:
  - Source/destination IP (outer tunnel headers).
  - Encrypted payload (no sensitive data).

3. DDoS Attack Simulation

Goal: Simulate a DDoS attack targeting the server (n2).

NS-3 Code Snippet:

```
cpp
Copy
// Create malicious nodes
NodeContainer attackers;
attackers.Create(10); // 10 malicious nodes

// Connect attackers to the network (e.g., via a switch)
CsmaHelper csma;
csma.SetChannelAttribute("DataRate", StringValue("100Mbps"));
csma.SetChannelAttribute("Delay", TimeValue(NanoSeconds(6560)));
NetDeviceContainer attackerDevices = csma.Install(attackers);

// Install UDP flood application on attackers
UdpEchoClientHelper ddosClient(9, InetSocketAddress(serverIp, 9));
ddosClient.SetAttribute("MaxPackets", UintegerValue(10000));
ddosClient.SetAttribute("Interval", TimeValue(MicroSeconds(10))); // High rate
ddosClient.SetAttribute("PacketSize", UintegerValue(1024));
ApplicationContainer ddosApps = ddosClient.Install(attackers);
ddosApps.Start(Seconds(2.0));
ddosApps.Stop(Seconds(8.0));
```

Traffic Pattern:

- **SYN Flood:** Simulate using small UDP packets at a high rate.
- **UDP Flood:** Use large UDP packets to consume bandwidth.

Impact Measurement:

- Use `FlowMonitor` to measure:
  - Packet loss for legitimate traffic (HQ to DC).
  - Increased latency and jitter.

4. Defense Mechanisms

Goal: Propose and implement three defense mechanisms.

a) Rate Limiting

NS-3 Code Snippet:

```
cpp
Copy
// Install TokenBucketFilter on router interface
TrafficControlHelper tch;
tch.SetRootQueueDisc("ns3::TokenBucketFilter",
```

```
                    "TokenRate", StringValue("1000000"), // 1 Mbps
                    "TokenSize", StringValue("1500"),   // 1500-byte tokens
                    "BurstSize", StringValue("15000")); // 10-packet burst

QueueDiscContainer queueDiscs = tch.Install(devices.Get(1)); // Apply to
router's outgoing interface
```

b) Access Control Lists (ACLs)

NS-3 Code Snippet:

```
cpp
Copy
// Drop packets from attacker IPs
Ptr<Ipv4StaticRouting> routerRouting =
ipv4RoutingHelper.GetStaticRouting(nodes.Get(1)->GetObject<Ipv4>());
routerRouting->AddDropRoute(Ipv4Address("10.1.4.0"),
Ipv4Mask("255.255.255.0")); // Block attacker subnet
```

c) Anycast or Load Balancing

NS-3 Code Snippet:

```
cpp
Copy
// Simulate anycast by duplicating the server (n2) on multiple nodes
NodeContainer anycastServers;
anycastServers.Add(nodes.Get(2)); // Original server
anycastServers.Create(2); // Add 2 more servers

// Distribute traffic using ECMP (Equal-Cost Multi-Path)
Ipv4StaticRoutingHelper routingHelper;
Ptr<Ipv4StaticRouting> clientRouting =
routingHelper.GetStaticRouting(nodes.Get(0)->GetObject<Ipv4>());
clientRouting->AddMulticastRoute(Ipv4Address("224.0.0.1"),
Ipv4Mask("255.255.255.255"), 1); // Simulate anycast
```

5. Security vs. Performance Trade-off Analysis

Goal: Analyze the trade-offs between security and performance.

Metric	IPsec Overhead	DDoS Protection Overhead
Throughput Reduction	10-20%	5-10% (rate limiting)
Latency Increase	+5-10ms (encryption)	+2-5ms (ACL processing)
Packet Loss	Minimal	High for attack traffic

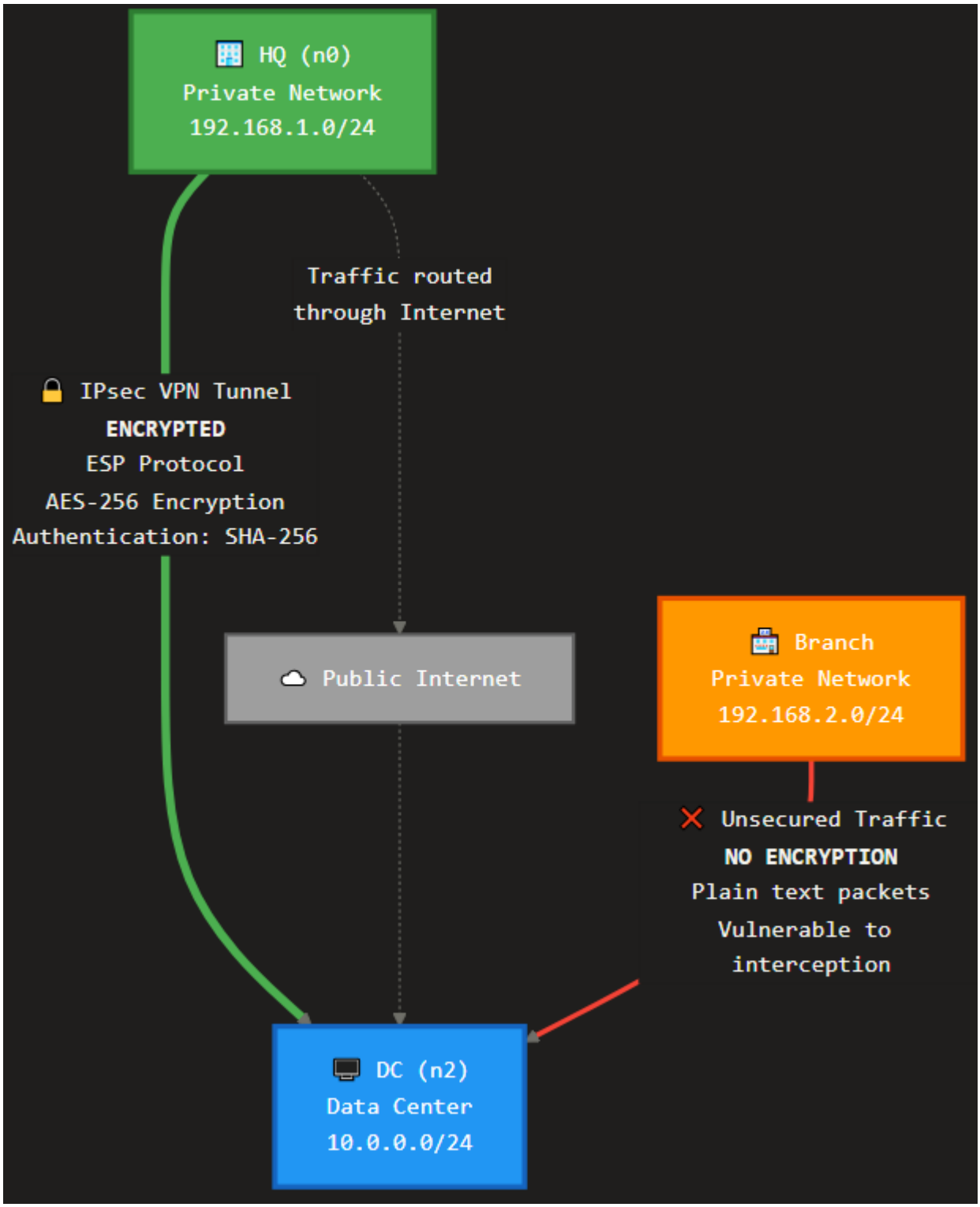
Balanced Security Posture:

- **For VoIP Traffic:** Use IPsec for confidentiality, but exempt from rate limiting.

- **For FTP Traffic:** Apply rate limiting and ACLs to mitigate DDoS.
- **Monitoring:** Use `FlowMonitor` to detect anomalies and adjust policies dynamically.

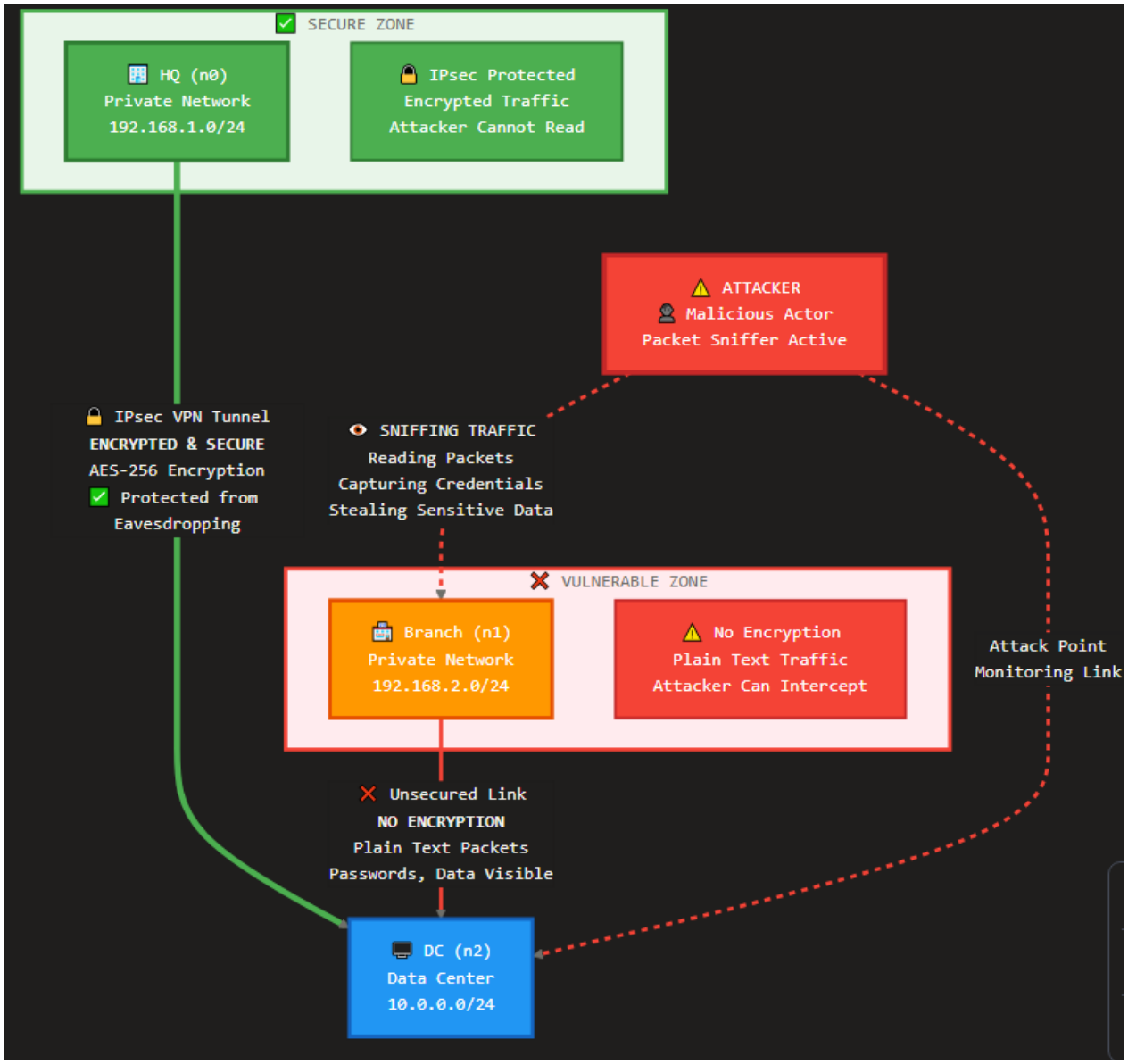
1. IPsec VPN Tunnel Diagram

Title: IPsec VPN Tunnel Between HQ and DC



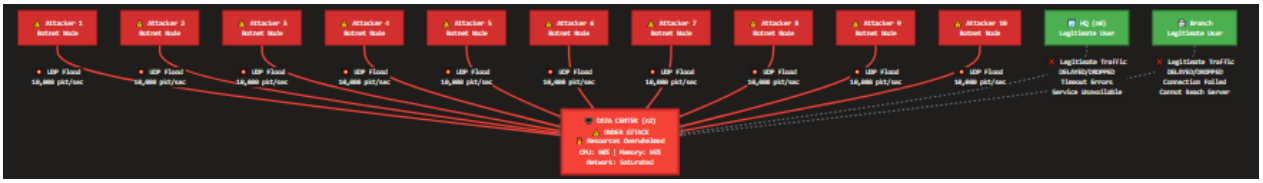
2. Eavesdropping Attack Diagram

Title: Eavesdropping on Unsecured Traffic



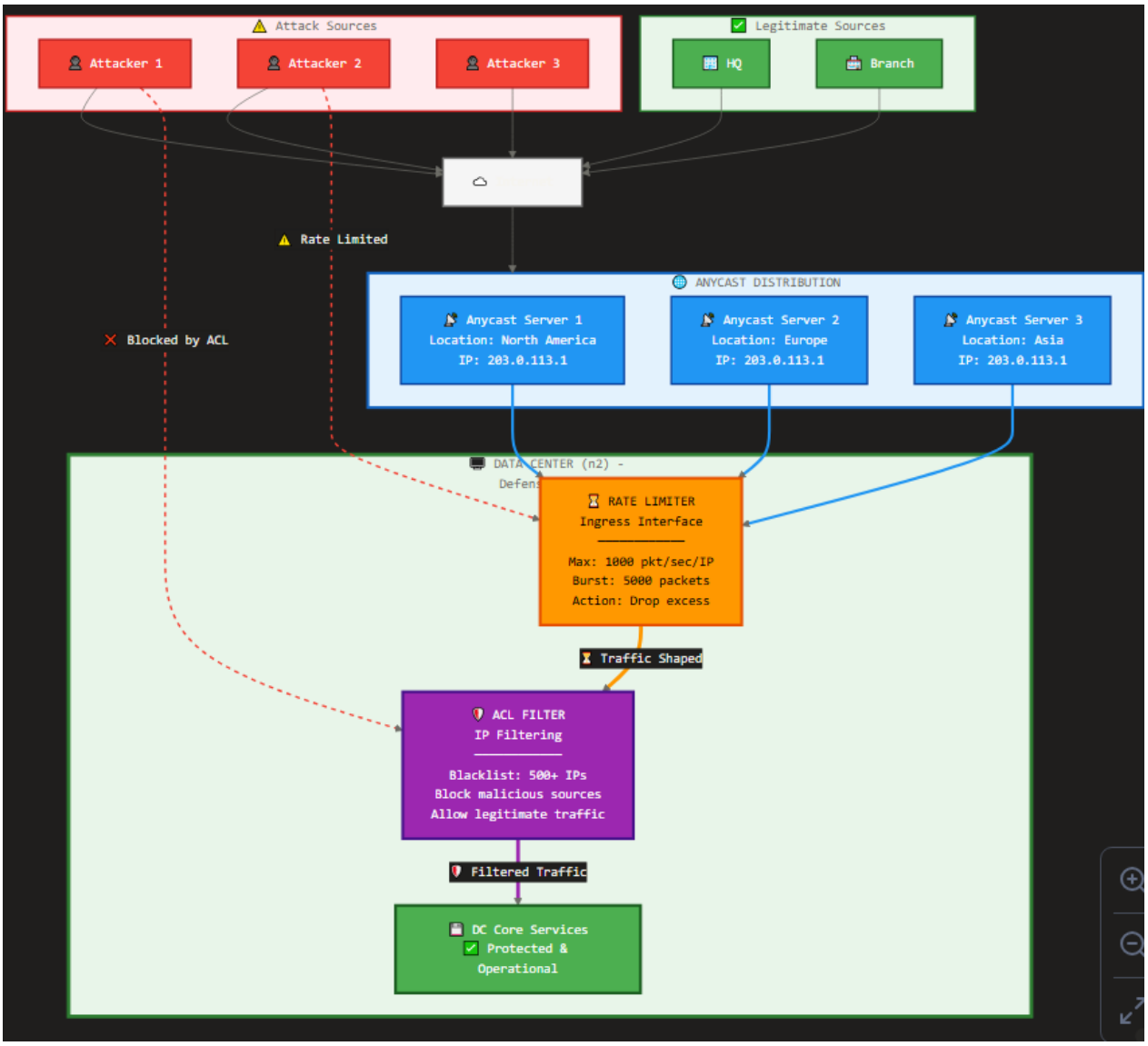
3. DDoS Attack Diagram

Title: DDoS Attack on DC (n2)



4. Defense Mechanisms Diagram

Title: Defense Mechanisms Against DDoS



Exercise 4: Multi-Hop WAN Architecture with Fault Tolerance

---

1. Topology Analysis and Extension

Goal: Extend the NS-3 simulation to a 3-node, 4-network topology:

- **DC-A (Router)**
- **DR-B (Server)**
- **Branch-C (Client)**

Topology Path:

- Branch-C → (Network1) → DC-A → (Network2) → DR-B
- DC-A and DR-B have a direct backup link (Network3).

NS-3 Code Snippet:

```
cpp
Copy
// Create nodes
NodeContainer nodes;
nodes.Create(3); // Branch-C (n0), DC-A (n1), DR-B (n2)

// Create point-to-point links
PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
p2p.SetChannelAttribute("Delay", StringValue("2ms"));

// Branch-C (n0) to DC-A (n1)
NetDeviceContainer d0d1 = p2p.Install(nodes.Get(0), nodes.Get(1));
// DC-A (n1) to DR-B (n2)
NetDeviceContainer d1d2 = p2p.Install(nodes.Get(1), nodes.Get(2));
// Backup link: DC-A (n1) to DR-B (n2)
NetDeviceContainer d1d2Backup = p2p.Install(nodes.Get(1), nodes.Get(2));

// Assign IP addresses
Ipv4AddressHelper ipv4;
ipv4.SetBase("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i0i1 = ipv4.Assign(d0d1);
ipv4.SetBase("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer i1i2 = ipv4.Assign(d1d2);
ipv4.SetBase("10.1.3.0", "255.255.255.0");
Ipv4InterfaceContainer i1i2Backup = ipv4.Assign(d1d2Backup);
```

Logical Topology Diagram (Text Description):

```
Copy
```

```
Branch-C (n0)
    |
    | 10.1.1.0/24 (Network1)
    |
DC-A (n1)
/       \
10.1.2.0/24  10.1.3.0/24 (Backup)
/           \
DR-B (n2)----DR-B (n2)
```

---

2. Static Routing Complexity

Goal: Configure static routes for normal and backup operations.

Routing Table for Branch-C (n0):

Destination Network	Next-Hop IP	Interface (Device)
10.1.2.0/24 (DR-B)	10.1.1.2 (DC-A)	d0d1

Routing Table for DC-A (n1):

Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (Branch-C)	Directly Connected	d0d1
10.1.2.0/24 (DR-B)	Directly Connected	d1d2
10.1.3.0/24 (Backup)	Directly Connected	d1d2Backup

Routing Table for DR-B (n2):

Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (Branch-C)	10.1.2.1 (DC-A)	d1d2
10.1.3.0/24 (Backup)	Directly Connected	d1d2Backup

NS-3 Code for Static Routing:

```
cpp
Copy
// Enable static routing
Ipv4StaticRoutingHelper staticRouting;

// Branch-C (n0) routing
Ptr<Ipv4StaticRouting> n0StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(0)->GetObject<Ipv4>());
n0StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.2.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.1.2"), 1);

// DC-A (n1) routing
```

```
Ptr<Ipv4StaticRouting> n1StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(1)->GetObject<Ipv4>());
n1StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.3.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.2.2"), 2); // Backup route

// DR-B (n2) routing
Ptr<Ipv4StaticRouting> n2StaticRouting =
staticRouting.GetStaticRouting(nodes.Get(2)->GetObject<Ipv4>());
n2StaticRouting->AddNetworkRouteTo(Ipv4Address("10.1.1.0"),
Ipv4Mask("255.255.255.0"), Ipv4Address("10.1.2.1"), 1);
```

---

3. Simulating Link Failure

Goal: Simulate the failure of the primary DC-A to DR-B link at t=5 seconds.

NS-3 Code Snippet:

```
cpp
Copy
// Schedule link failure at t=5s
Simulator::Schedule(Seconds(5.0), &DisableLink, d1d2.Get(0), d1d2.Get(1));

// Function to disable a link
void DisableLink(Ptr<NetDevice> dev1, Ptr<NetDevice> dev2) {
    Ptr<PointToPointNetDevice> p2pDev1 =
DynamicCast<PointToPointNetDevice>(dev1);
    Ptr<PointToPointNetDevice> p2pDev2 =
DynamicCast<PointToPointNetDevice>(dev2);
    p2pDev1->SetChannel(nullptr);
    p2pDev2->SetChannel(nullptr);
    std::cout << "Primary link between DC-A and DR-B failed at " <<
Simulator::Now().GetSeconds() << "s" << std::endl;
}
```

Immediate Effect:

- The primary route from Branch-C to DR-B via DC-A is lost.
- Static routing will not automatically reroute; traffic is dropped unless backup routes are manually configured.

---

4. Convergence Analysis

Goal: Compare static and dynamic routing (OSPF) for failover.

NS-3 Code for OSPF:

```
cpp
Copy
```

```
// Enable OSPF on DC-A and DR-B
OspfHelper ospfHelper;
Ipv4ListRoutingHelper listRouting;
listRouting.Add(staticRouting, 0);
listRouting.Add(ospfHelper, 10);

// Install OSPF on nodes
Ipv4StaticRoutingHelper staticRoutingHelper;
Ipv4ListRoutingHelper list;
list.Add(staticRoutingHelper, 0);
list.Add(ospfHelper, 10);
InternetStackHelper internet;
internet.SetRoutingHelper(list);
internet.Install(nodes);
```

Convergence Behavior:

- **Static Routing:** Traffic is dropped until manual intervention.
- **OSPF:** Automatically reroutes traffic to the backup link within seconds.

---

5. Business Continuity Verification

Goal: Use `FlowMonitor` to verify traffic paths and measure impact.

NS-3 Code for FlowMonitor:

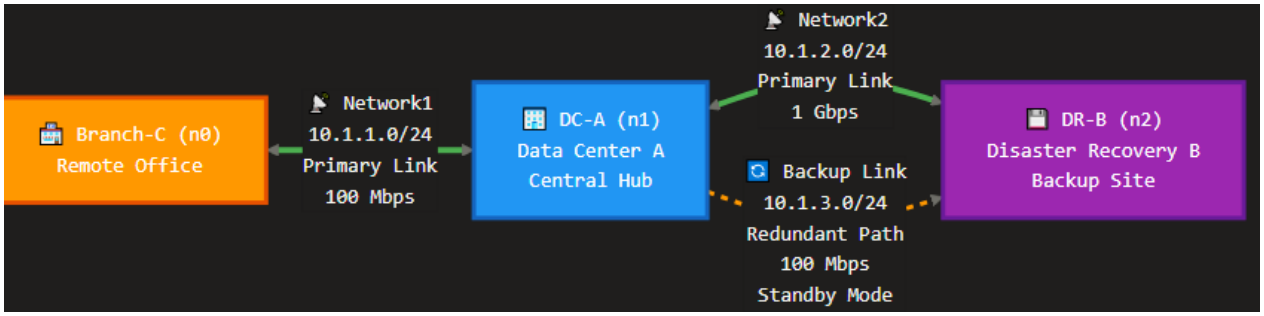
```
cpp
Copy
// Enable FlowMonitor
Ptr<FlowMonitor> flowMonitor;
FlowMonitorHelper flowHelper;
flowMonitor = flowHelper.InstallAll();

// Run simulation
Simulator::Stop(Seconds(15.0));
Simulator::Run();

// Print per-flow statistics
flowMonitor->CheckForLostPackets();
std::map<FlowId, FlowMonitor::FlowStats> stats = flowMonitor->GetFlowStats();

for (auto &flow : stats) {
    std::cout << "Flow " << flow.first << " (" <<
flow.second.ipv4SourceAddress << " -> " << flow.second.ipv4DestinationAddress
<< ")\n";
    std::cout << "  Tx Packets: " << flow.second.txPackets << "\n";
    std::cout << "  Rx Packets: " << flow.second.rxPackets << "\n";
    std::cout << "  Lost Packets: " << flow.second.lostPackets << "\n";
    std::cout << "  Delay: " << flow.second.delaySum.GetSeconds() /
flow.second.rxPackets << "s\n";
}
```

Expected Results:

- **Before Failure:** Traffic flows via the primary path (Branch-C → DC-A → DR-B).
- **After Failure (Static Routing):** Traffic is dropped.
- **After Failure (OSPF):** Traffic is rerouted via the backup path (Branch-C → DC-A → DR-B via backup link).

1. Multi-Hop WAN Topology Diagram

Title: Multi-Hop WAN Topology with Backup Link



2. Static Routing Table Diagram

Title: Static Routing Tables for Branch-C, DC-A, and DR-B

Description: Create three tables (one for each node) showing the routing entries.

Branch-C (n0):

Destination Network	Next-Hop IP	Interface (Device)
10.1.2.0/24 (DR-B)	10.1.1.2 (DC-A)	d0d1

DC-A (n1):

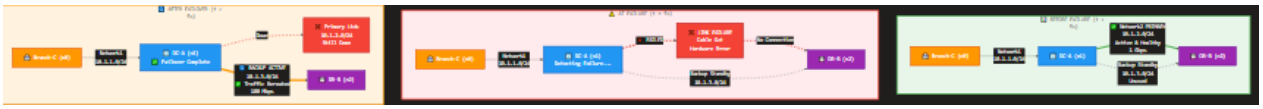
Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (Branch-C)	Directly Connected	d0d1
10.1.2.0/24 (DR-B)	Directly Connected	d1d2
10.1.3.0/24 (Backup)	Directly Connected	d1d2Backup

DR-B (n2):

Destination Network	Next-Hop IP	Interface (Device)
10.1.1.0/24 (Branch-C)	10.1.2.1 (DC-A)	d1d2
10.1.3.0/24 (Backup)	Directly Connected	d1d2Backup

3. Link Failure Simulation Diagram

Title: Primary Link Failure at t=5s



Exercise 5: Policy-Based Routing for Application-Aware WAN Path Selection

1. Traffic Classification Logic

Goal: Generate two distinct traffic flows:

- **Flow_Video:** Simulates RTP-like traffic (small, periodic packets).
- **Flow_Data:** Simulates FTP-like traffic (large, bursty packets).

NS-3 Code Snippet:

```
// Flow_Video: RTP-like traffic (small, periodic packets)
uint16_t videoPort = 4000;
UdpClientHelper videoClient(Ipv4Address("10.1.2.2"), videoPort); //
Destination: Cloud (n2)
videoClient.SetAttribute("MaxPackets", UintegerValue(1000));
videoClient.SetAttribute("Interval", TimeValue(MilliSeconds(20))); // 20ms
interval
videoClient.SetAttribute("PacketSize", UintegerValue(200)); // Small packet
size
ApplicationContainer videoApp = videoClient.Install(nodes.Get(0)); // Studio
(n0)
videoApp.Start(Seconds(1.0));
videoApp.Stop(Seconds(10.0));

// Flow_Data: FTP-like traffic (large, bursty packets)
uint16_t dataPort = 5000;
UdpClientHelper dataClient(Ipv4Address("10.1.2.2"), dataPort); // Destination:
Cloud (n2)
```

```
dataClient.SetAttribute("MaxPackets", UintegerValue(100));
dataClient.SetAttribute("Interval", TimeValue(MilliSeconds(100))); // 100ms
interval
dataClient.SetAttribute("PacketSize", UintegerValue(1500)); // Large packet
size
ApplicationContainer dataApp = dataClient.Install(nodes.Get(0)); // Studio
(n0)
dataApp.Start(Seconds(1.0));
dataApp.Stop(Seconds(10.0));
```

Traffic Classification:

- Use **DSCP (Differentiated Services Code Point)** to tag packets:
  - **Flow_Video:** DSCP EF (Expedited Forwarding, value: 0xB8)
  - **Flow_Data:** DSCP BE (Best Effort, value: 0x00)

NS-3 Code for DSCP Tagging:

```
// Tag Flow_Video packets with DSCP EF
Ptr<Socket> videoSocket = Socket::CreateSocket(nodes.Get(0),
UdpSocketFactory::GetTypeId());
videoSocket->SetIpTos(0xB8); // DSCP EF

// Tag Flow_Data packets with DSCP BE
Ptr<Socket> dataSocket = Socket::CreateSocket(nodes.Get(0),
UdpSocketFactory::GetTypeId());
dataSocket->SetIpTos(0x00); // DSCP BE
```

2. Implementing PBR in NS-3

Goal: Implement Policy-Based Routing logic at the Studio router (n0).

NS-3 Code Snippet (Pseudocode):

```
cpp
Copy
// Custom forwarding logic for PBR
void CustomForwarding(Ptr<const Packet> p, Ipv4Header& ipHeader,
Ptr<NetDevice> incomingDevice, Ptr<Ipv4Route> route) {
    uint8_t dscp = ipHeader.GetTos() >> 2; // Extract DSCP value

    if (dscp == 0x2E) { // DSCP EF (0xB8 >> 2 = 0x2E)
        // Forward Flow_Video via low-latency path (e.g., interface 1)
        ForwardPacket(p, ipHeader, incomingDevice, lowLatencyInterface);
    } else {
        // Forward Flow_Data via default path (e.g., interface 0)
        ForwardPacket(p, ipHeader, incomingDevice, defaultInterface);
    }
}

// Install custom forwarding logic on Studio router (n0)
Ptr<Ipv4L3Protocol> ipv4 = nodes.Get(0)->GetObject<Ipv4L3Protocol>();
```

```
ipv4->SetForwardingCallback(MakeCallback(&CustomForwarding));
```

PBR Logic Explanation:

- **Classify packets** based on DSCP values.
- **Forward Flow_Video** via a low-latency interface.
- **Forward Flow_Data** via a default interface.

3. Path Characterization

Goal: Measure real-time latency and bandwidth on each egress interface.

NS-3 Code for Path Metrics:

```
cpp
Copy
// Use FlowMonitor to measure latency and bandwidth
Ptr<FlowMonitor> flowMonitor;
FlowMonitorHelper flowHelper;
flowMonitor = flowHelper.InstallAll();

// Trace latency on each interface
Config::ConnectWithoutContext("/NodeList/0/$ns3::Ipv4L3Protocol/Rx",
    MakeCallback(&LatencyTraceSink));

void LatencyTraceSink(Ptr<const Packet> p, const Ipv4Header& ipHeader,
uint32_t interface) {
    // Calculate and log latency for each flow
    if (ipHeader.GetTos() >> 2 == 0x2E) { // Flow_Video
        std::cout << "Flow_Video Latency: " << Simulator::Now().GetSeconds() -
GetPacketTimestamp(p) << "s\n";
    }
}
```

Bandwidth Measurement:

```
cpp
Copy
// Use FlowMonitor to measure bandwidth
std::map<FlowId, FlowMonitor::FlowStats> stats = flowMonitor->GetFlowStats();
for (auto &flow : stats) {
    double throughput = (flow.second.rxBytes * 8.0) /
(flow.second.timeLastRxPacket.GetSeconds() -
flow.second.timeFirstTxPacket.GetSeconds()) / 1000; // Kbps
    std::cout << "Flow " << flow.first << " Throughput: " << throughput << "
Kbps\n";
}
```

4. Dynamic Policy Engine

Goal: Design a simple SD-WAN-like controller to update routing based on real-time metrics.

NS-3 Code for Dynamic Policy Engine:

```
cpp
Copy
// Dynamic Policy Engine Class
class DynamicPolicyEngine {
public:
    DynamicPolicyEngine(Ptr<Node> node) : m_node(node) {}

    void UpdateRoutes() {
        // Fetch real-time metrics (e.g., latency, bandwidth)
        double videoLatency = GetFlowLatency(0x2E); // Flow_Video DSCP
        double dataLatency = GetFlowLatency(0x00); // Flow_Data DSCP

        // Apply policy: Switch Flow_Video to backup if latency > 30ms
        if (videoLatency > 0.030) {
            UpdateForwardingTable(Flow_Video, BackupInterface);
        }
    }

private:
    Ptr<Node> m_node;
};

// Schedule periodic updates
DynamicPolicyEngine policyEngine(nodes.Get(0));
Simulator::Schedule(Seconds(1.0), &DynamicPolicyEngine::UpdateRoutes,
&policyEngine);
Simulator::Schedule(Seconds(2.0), &DynamicPolicyEngine::UpdateRoutes,
&policyEngine);
// Repeat every second
```

Policy Rule:

- If **Flow_Video latency > 30ms**, switch to a backup interface.

5. Validation and Trade-offs

Goal: Validate the PBR implementation and discuss trade-offs.

Validation:

- Use `FlowMonitor` to verify:
  - **Flow_Video** packets use the low-latency path.
  - **Flow_Data** packets use the default path.
  - Latency for **Flow_Video** is consistently < 30ms.

NS-3 Code for Validation:

```
// Print FlowMonitor results
flowMonitor->CheckForLostPackets();
```

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

for (auto &flow : stats) {
    std::cout << "Flow " << flow.first << " (" <<
flow.second.ipv4SourceAddress << " -> " << flow.second.ipv4DestinationAddress
<< ")\n";
    std::cout << "  DSCP: " << (flow.second.ipv4Tos >> 2) << "\n";
    std::cout << "  Avg. Latency: " << flow.second.delaySum.GetSeconds() /
flow.second.rxPackets << "s\n";
    std::cout << "  Path: " << (flow.second.ipv4Tos >> 2 == 0x2E ? "Low-
Latency" : "Default") << "\n";
}
```

Trade-offs:

Aspect	Simulation Overhead	Real-World Overhead
Computational	High (per-packet processing)	Low (hardware-accelerated)
Latency	Minimal	Minimal
Scalability	Limited (per-flow processing)	High (ASICs, FPGAs)

Scalability Limits:

- NS-3's software-based processing limits the number of flows.
- Real-world routers use hardware acceleration (ASICs) for PBR.