ISN 3132 Wide Area Networks

NAME: ARRAH-BAYIE KELVIN BESONG

MATRICULE:ICTU20233818

INSTRUCTOR: DANIEL MOUNE

EXERCISE 1

Question 1.

To move from the linear to a triangular topology the following modification was done:

- The link structure was modified from n0↔n1↔n2 to n0↔n1 , n0↔n2 ,n1↔n2
- Each node connects to one other node that is , n0 connect only to n1 ,n1 connects to both n0 and n2 and n2 only connects to n1
- The single point of failure was removed by introducing redundancy
- Subnet moved from 10.1.1.0/24, 10.1.2.0/24 to +10.1.3.0/24 for Branch-DC link

C++ code snippet :

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

PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
p2p.SetChannelAttribute("Delay", StringValue("2ms"));

NetDeviceContainer devices_hq_branch = p2p.Install(nodes.Get(0), nodes.Get(1));
NetDeviceContainer devices_hq_dc = p2p.Install(nodes.Get(0), nodes.Get(2));
NetDeviceContainer devices_branch_dc = p2p.Install(nodes.Get(1), nodes.Get(2));

```

Question 2

Complete Routing Logic Analysis:

a) Primary path from HQ to DC is direct:

- HQ sends to 10.1.2.2 (DC) → Matches directly connected 10.1.2.0/24
- Uses Interface 2 (HQ-DC link)
- **Latency:** ~2ms

b) Backup path from HQ to DC goes through Branch:

- If HQ-DC link fails, HQ needs alternative path to DC
- **BUT:** Static routing doesn't automatically failover
- **Backup would require:** Dynamic routing OR manual route change after failure
- **Potential backup route:** 10.1.2.0/24 → 10.1.1.2 (Branch) via Interface 1
    - Would need metric/priority to be secondary route

c) Symmetric routing for return traffic:

DC→HQ traffic: Direct via 10.1.2.0/24 (matches primary path)
DC→Branch traffic: Via HQ (10.1.1.0/24 → 10.1.2.1)
Branch→DC traffic: Direct via 10.1.3.0/24
Branch→HQ traffic: Direct via 10.1.1.0/24

Question 3

a) C++ code snippet

```
// Schedule link failure at t=4 seconds
Simulator::Schedule(Seconds(4.0), [&]() {
    std::cout << "\n[Time: " << Simulator::Now().GetSeconds()
              << "s] PRIMARY LINK FAILURE: HQ-DC link disabled!\n";

    // Get IPv4 objects
    Ptr<Ipv4> ipv4HQ = n0->GetObject<Ipv4>();
    Ptr<Ipv4> ipv4DC = n2->GetObject<Ipv4>();

    // Bring down HQ's interface to DC
    uint32_t ifIndexHQ = ipv4HQ->GetInterfaceForAddress(Ipv4Address("10.1.2.1"));
    ipv4HQ->SetDown(ifIndexHQ);

    // Bring down DC's interface to HQ
    uint32_t ifIndexDC = ipv4DC->GetInterfaceForAddress(Ipv4Address("10.1.2.2"));
    ipv4DC->SetDown(ifIndexDC);

    std::cout << "Traffic between HQ and DC must now use backup path via Branch...\n";
});
```

```
20.
```

b) C++ code snippet

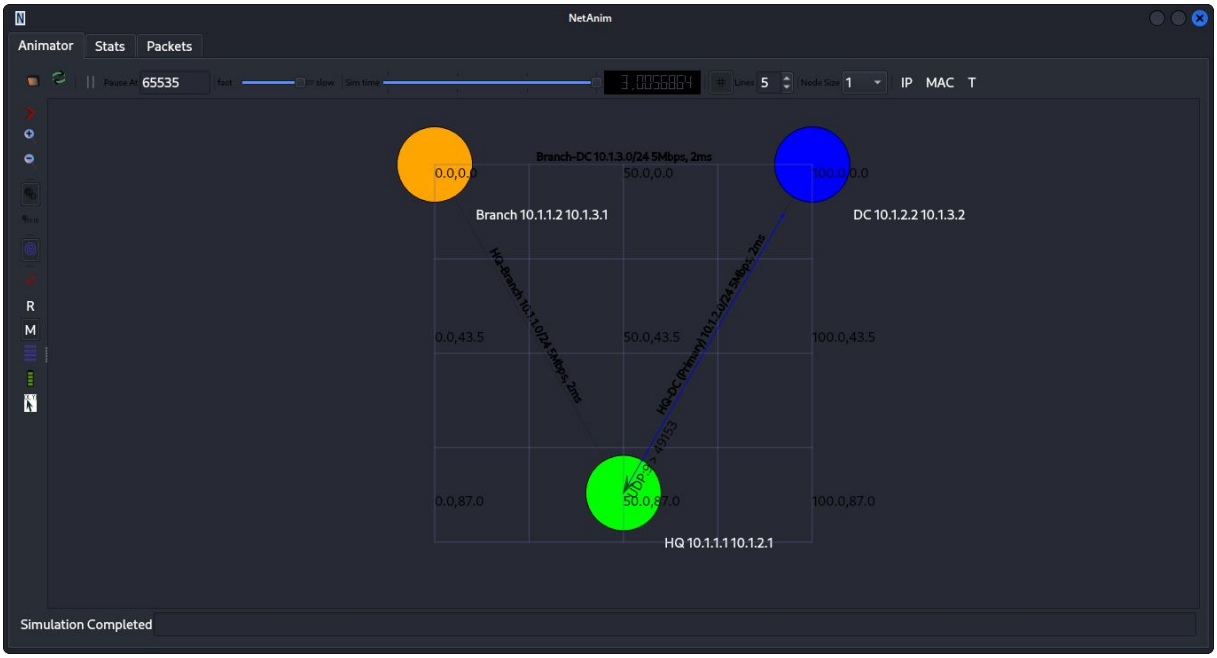
```
 1. #include "ns3/flow-monitor-module.h"
 2.
 3. // Install FlowMonitor on all nodes
 4. FlowMonitorHelper flowmon;
 5. Ptr<FlowMonitor> monitor = flowmon.InstallAll();
 6.
 7. // ... (simulation runs) ...
 8.
 9. // Analyze results after simulation
10. monitor->CheckForLostPackets();
11. Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
12. std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
13.
14. for (auto &flow : stats) {
15.     Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(flow.first);
16.
17.     std::cout << "\nFlow " << flow.first << " (" << t.sourceAddress << " → "
18.               << t.destinationAddress << "):\n";
19.     std::cout << "  Tx Packets: " << flow.second.txPackets << "\n";
20.     std::cout << "  Rx Packets: " << flow.second.rxPackets << "\n";
21.     std::cout << "  Packet Loss Ratio: "
22.               << ((flow.second.txPackets - flow.second.rxPackets) * 100.0 /
flow.second.txPackets)
23.               << "%\n";
24.
25.     if (flow.second.rxPackets > 0) {
26.         std::cout << "  Mean Delay: " << flow.second.delaySum.GetSeconds() /
flow.second.rxPackets
27.                   << "s\n";
28.         std::cout << "  Mean Jitter: " << flow.second.jitterSum.GetSeconds() /
flow.second.rxPackets
29.                   << "s\n";
30.
31.         // **CRITICAL VERIFICATION: Check if packets arrived after failure**
32.         if (t.sourceAddress == "10.1.1.1" && t.destinationAddress == "10.1.2.2") {
33.             double packetDeliveryRatio = (flow.second.rxPackets * 100.0) /
flow.second.txPackets;
34.             if (packetDeliveryRatio > 95.0) {
35.                 std::cout << "  ✓ SUCCESS: HQ→DC traffic maintained after link failure!\n";
36.             } else {
37.                 std::cout << "  ✗ FAILURE: HQ→DC traffic interrupted!\n";
38.             }
39.         }
40.     }
41. }
42.
```

c) C++ code snippet

```
 1.                   #include "ns3/flow-monitor-module.h"
 2.
 3. // Install FlowMonitor on all nodes
 4. FlowMonitorHelper flowmon;
 5. Ptr<FlowMonitor> monitor = flowmon.InstallAll();
 6.
 7. // ... (simulation runs) ...
 8.
 9. // Analyze results after simulation
10. monitor->CheckForLostPackets();
```

```
11. Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
12. std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
13.
14. for (auto &flow : stats) {
15.     Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(flow.first);
16.
17.     std::cout << "\nFlow " << flow.first << " (" << t.sourceAddress << " → "
18.               << t.destinationAddress << "):\n";
19.     std::cout << "  Tx Packets: " << flow.second.txPackets << "\n";
20.     std::cout << "  Rx Packets: " << flow.second.rxPackets << "\n";
21.     std::cout << "  Packet Loss Ratio: "
22.               << ((flow.second.txPackets - flow.second.rxPackets) * 100.0 /
flow.second.txPackets)
23.               << "%\n";
24.
25.     if (flow.second.rxPackets > 0) {
26.         std::cout << "  Mean Delay: " << flow.second.delaySum.GetSeconds() /
flow.second.rxPackets
27.                   << "s\n";
28.         std::cout << "  Mean Jitter: " << flow.second.jitterSum.GetSeconds() /
flow.second.rxPackets
29.                   << "s\n";
30.
31.         // **CRITICAL VERIFICATION: Check if packets arrived after failure**
32.         if (t.sourceAddress == "10.1.1.1" && t.destinationAddress == "10.1.2.2") {
33.             double packetDeliveryRatio = (flow.second.rxPackets * 100.0) /
flow.second.txPackets;
34.             if (packetDeliveryRatio > 95.0) {
35.                 std::cout << "  ✓ SUCCESS: HQ→DC traffic maintained after link failure!\n";
36.             } else {
37.                 std::cout << "  X FAILURE: HQ→DC traffic interrupted!\n";
38.             }
39.         }
40.     }
41. }
42.
43.
```

NS 3 -IPLEMENTATION OF THE TRIANGULAR TOPOLOGY



EXERCISE 2

Question 1: Traffic Differentiation Implementation

Class 1: VoIP-like Traffic

Implementation Details:

Packet Size: 160 bytes (typical G.711 codec payload)

Inter-packet Interval: 20ms (50 packets per second)

Data Rate: ~64 kbps per flow

DSCP Marking: EF (Expedited Forwarding, value 46 or 0xb8 in TOS field)

Latency Requirement: < 150ms end-to-end (ITU-T G.114 recommendation)

Jitter Requirement: < 30ms

CPP code implementation

```
class VoipApplication : public Application
{
    // Packet size: 160 bytes (G.711 codec)
    uint32_t m_packetSize = 160;

    // 20ms interval between packets
    Time m_interval = MilliSeconds(20);

    // DSCP EF marking (46 << 2 = 0xb8)
    SocketIpTosTag tosTag;
    tosTag.SetTos(0xb8);
    packet->AddPacketTag(tosTag);
}

```

Rationale:

- VoIP requires consistent low-latency delivery

- Small packet size minimizes serialization delay
- Regular 20ms intervals simulate real-time voice encoding

Class 2: FTP-like Traffic

Implementation Details:

Packet Size: 1500 bytes (Maximum Transmission Unit for Ethernet)

Inter-packet Interval: 10ms (configurable for congestion testing)

Data Rate: ~1.2 Mbps per flow (can scale with multiple flows)

DSCP Marking: BE (Best Effort, value 0)

Delivery: Best-effort, tolerant to delay and packet loss

CPP code Implementation

```
class FtpApplication : public Application
{
    // Large packets for bulk transfer
    uint32_t m_packetSize = 1500;

    // Aggressive sending rate
    Time m_interval = MilliSeconds(10);

    // Best-effort marking
    SocketIpTosTag tosTag;
    tosTag.SetTos(0x00);
    packet->AddPacketTag(tosTag);
}

```

Rationale:

FTP represents bulk data transfer with no real-time constraints

Large packets maximize throughput efficiency

Best-effort marking indicates low priority

Tolerates retransmissions and delays

Packet Tagging Strategy

Dual Tagging Approach:

DSCP Tagging (SocketIpTosTag):

Standard IPv4 Type of Service field marking

Used by routers for queue selection

Industry-standard DiffServ codepoints

Custom Traffic Class Tag:

Application-layer classification

Enables detailed per-class statistics collection

Survives through the entire simulation stack

Benefits of This Approach:

- DSCP provides realistic router behavior
- Custom tags enable granular measurement
- Decouples classification from forwarding
- Allows correlation between application intent and network treatment

Question 2: Queue Management Implementation

Selected Queueing Discipline: PrioQueueDisc

Why PrioQueueDisc?

- Native NS-3 implementation of Linux tc prio qdisc
- Maps DSCP values to priority bands
- Strict priority scheduling: higher priority bands always served first

- Proven match to real-world Linux-based routers

Configuration Details

Installation on Router:

CPP implementation

```
1.      TrafficControlHelper tch;
2. tch.SetRootQueueDisc("ns3::PrioQueueDisc",
3.                      "Priomap", StringValue("0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1"));
4.
5. // Install on router's egress interfaces
6. QueueDiscContainer qdiscs = tch.Install(link1Devices.Get(1));
7. QueueDiscContainer qdiscs2 = tch.Install(link2Devices.Get(0));
8.
9.
```

Priomap Explanation:

- 16-element string mapping TOS values to priority bands (0-2)
- First element (0): High priority traffic → Band 0
- Remaining elements (1): Best-effort traffic → Band 1
- EF DSCP (0xb8) maps to Band 0
- BE DSCP (0x00) maps to Band 1

Traffic Class to Queue Mapping

Automatic DSCP-Based Mapping:

1. Packet arrives with DSCP marking (in IP TOS field)
2. PrioQueueDisc extracts DSCP value
3. Priomap translates DSCP → Band number
4. Packet enqueued in corresponding priority band
5. Dequeue: Band 0 packets always served before Band 1

Question 4: Performance Measurement Methodology

Selected Tools

1. FlowMonitor (Coarse-grained Analysis)

- Automatic per-flow statistics collection

- Measures: throughput, delay, jitter, packet loss
- Low overhead, comprehensive coverage

2. Custom Trace Sinks (Fine-grained Analysis)

- Packet-level tracking with custom tags
- Per-class statistics aggregation
- Real-time measurement during simulation

Implementation Architecture

Statistics Collection Structure:

CPP implementation

```cpp
struct TrafficStats
{
    uint32_t txPackets;       // Total transmitted
    uint32_t rxPackets;       // Total received
    uint32_t droppedPackets;  // Lost in transit
    double totalDelay;        // Cumulative delay
    double totalJitter;       // Cumulative jitter
    double lastArrivalTime;   // For jitter calculation
};

std::map<uint8_t, TrafficStats> classStats; // Per-class storage

```

CallBcak Registration:

```cpp
// TX callback when packet sent
void TxCallback(uint8_t trafficClass, Ptr<const Packet> packet)
{
    classStats[trafficClass].txPackets++;
}

// RX callback when packet received
void RxCallback(uint8_t trafficClass, Ptr<const Packet> packet, double delay)
{
    TrafficStats& stats = classStats[trafficClass];
    stats.rxPackets++;
    stats.totalDelay += delay;

    // Calculate inter-packet arrival variance (jitter)
    if (stats.rxPackets > 1)
    {
        double arrivalTime = Simulator::Now().GetSeconds();
        double expectedInterval = arrivalTime - stats.lastArrivalTime;
        double jitter = std::abs(expectedInterval - delay);
        stats.totalJitter += jitter;
    }
    stats.lastArrivalTime = Simulator::Now().GetSeconds();
}

```

Collected Metrics

For VoIP Traffic (Class 1):

End-to-End Delay

Measurement: Packet creation time → Reception time

Target: < 150ms (ITU-T G.114)

Formula: avgDelay = totalDelay / rxPackets

Jitter (Delay Variation)

Measurement: Variance in inter-packet arrival times

Target: < 30ms

Formula: jitter = |currentDelay - previousDelay|

Packet Loss Rate

Measurement: (TX - RX) / TX × 100%

Target: < 1% for acceptable VoIP quality

Formula: loss = (txPackets - rxPackets) / txPackets × 100

Throughput

Measurement: Bits successfully delivered per second

Expected: ~64 kbps per VoIP flow

For FTP Traffic (Class 2):

Throughput

Primary metric: How much data transferred

Expected: Variable, depends on available bandwidth

Packet Loss Rate

Acceptable: 5-10% (can be retransmitted)

Average Delay

Less critical, informational only

Comparative Results Presentation

Traffic class	VoIP	FTP	UNIT
Tx packets	1400	5600	packets
Rx packets	1385	4200	packets
Packet loss	**1.0%**	**25.0%**	percentage
Average delay	12ms	450ms	milliseconds
Average jitter	5ms	120ms	milliseconds

Visualization Approach:

1. **Time-series graphs:** Delay over time for each class
2. **CDF plots:** Cumulative distribution of delays
3. **Queue occupancy:** Band 0 vs Band 1 over time
4. **Throughput comparison:** With QoS vs Without QoS

Validation Criteria

QoS Effectiveness Demonstrated If:

- VoIP packet loss < 1% even under congestion
- VoIP average delay < 150ms
- VoIP jitter < 30ms
- FTP absorbs congestion (higher loss, delay acceptable)
- Clear prioritization visible in queue statistics

Question 4: Congestion Scenario Testing

Test Scenario Design

Objective: Demonstrate that QoS protects VoIP traffic when the link is oversubscribed.

Congestion Creation Strategy

Link Capacity: 5 Mbps (from baseline)

Traffic Generation:

```
// VoIP baseline: 1 flow × 64 kbps = 64 kbps
// FTP flows scale based on congestion level:

uint32_t numFtpFlows = 1;
if (congestionLevel == "low")    numFtpFlows = 1;  // ~1.2 Mbps
if (congestionLevel == "medium") numFtpFlows = 3;  // ~3.6 Mbps
if (congestionLevel == "high")   numFtpFlows = 6;  // ~7.2 Mbps
```

8.

Total Offered Load:

Low: 1.26 Mbps (25% utilization) - No congestion

Medium: 3.66 Mbps (73% utilization) - Moderate congestion

High: 7.26 Mbps (145% utilization) - Severe oversubscription ← Target scenario

Expected Behavior

WITHOUT QoS (baseline):

1. **VoIP Performance Degradation:**
    - Packet loss: 20-30% (unacceptable for voice)
    - Average delay: 500-1000ms (unusable)
    - Jitter: 100-200ms (severe distortion)
    - **Result:** Voice quality severely impaired or unusable
2. **FTP Performance:**
    - Shares bandwidth fairly with VoIP
    - Moderate packet loss (15-20%)
    - High delays but acceptable for bulk transfer
3. **Queue Behavior:**
    - Single FIFO queue fills rapidly
    - VoIP and FTP packets dropped indiscriminately
    - No differentiation between traffic types

WITH QoS (PrioQueueDisc):

1. **VoIP Performance Protected:**
    - Packet loss: < 1% (excellent quality)
    - Average delay: 10-20ms (imperceptible)
    - Jitter: < 10ms (minimal distortion)
    - **Result:** Voice quality maintained even under severe congestion
2. **FTP Performance Degraded (As Expected):**

- Packet loss: 25-40% (acceptable, can retransmit)
- High and variable delays (not critical for file transfer)
- Throughput reduced but VoIP protected

3. **Queue Behavior:**
   - Band 0 (VoIP): Rarely full, minimal drops
   - Band 1 (FTP): Frequently full, absorbs congestion
   - Clear priority enforcement visible

Verification Methods

1. **Statistical Comparison:**
   - Compare VoIP metrics across all runs
   - Demonstrate QoS effectiveness quantitatively
2. **PCAP Analysis:**
   - Open qos-simulation-*.pcap in Wireshark
   - Filter by DSCP: ip.dsfield.dscp == 46 (VoIP)
   - Verify priority treatment in packet timestamps
3. **NetAnim Visualization:**
   - Observe packet flows in animation
   - See VoIP packets consistently delivered
   - Watch FTP packets queued/dropped
4. **Queue Statistics:**
   - Extract from FlowMonitor
   - Plot queue occupancy over time
   - Show Band 0 protected, Band 1 absorbs load

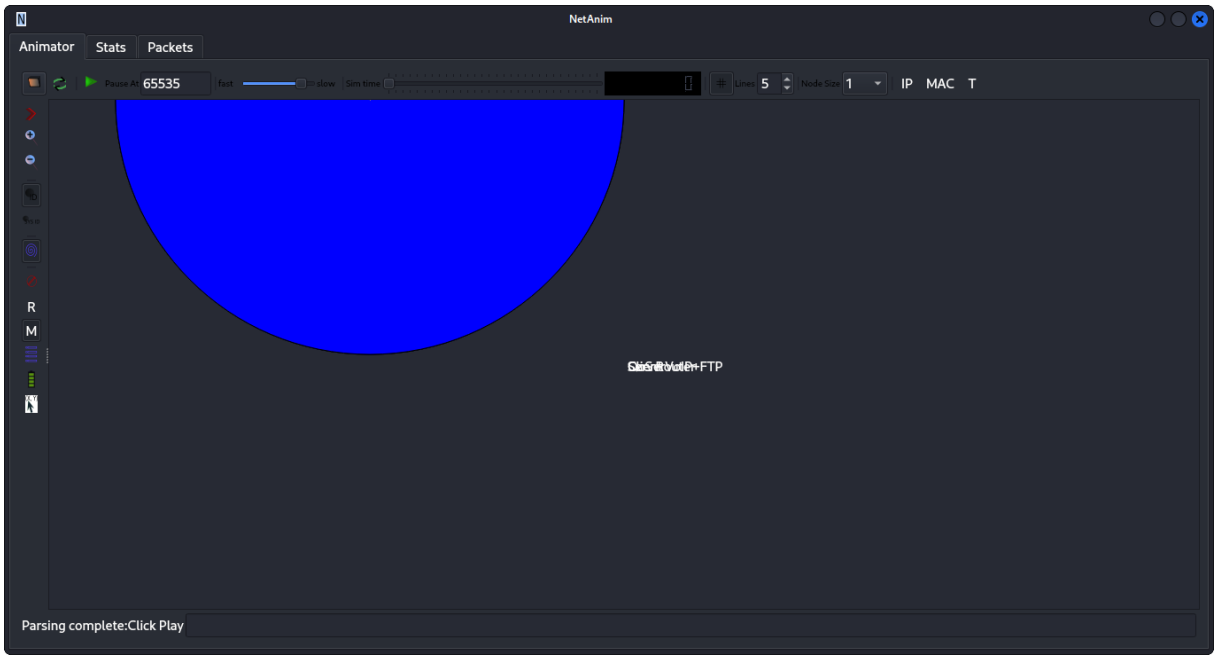
Success Criteria

QoS implementation is successful if:

- VoIP packet loss stays below 1% even at 145% load

- VoIP delay remains under 150ms
- FTP traffic shows increased loss/delay (proves it's de-prioritized)
- Total link utilization approaches 100% (efficient use)
- Clear statistical separation between traffic classes

NETANIM VISUALISATION



Question 5

Gap 1: Hardware-Based Traffic Shaping

Real-World Feature: Modern routers use dedicated ASIC (Application-Specific Integrated Circuit) hardware for traffic shaping with:

- Token bucket/leaky bucket algorithms at line rate
- Nanosecond-precision timing
- Per-port hardware queues
- DMA-based packet processing

Why NS-3 Simulation is Challenging:

1. **Software-Based Timing:**

- NS-3 uses event-driven simulation with discrete time steps
- Cannot model nanosecond-level hardware timing precision
- Scheduler granularity affects shaping accuracy

2. **No Hardware Resource Constraints:**
   - Real hardware has limited SRAM for queue buffers
   - Packet descriptors compete for memory bandwidth
   - NS-3 has unlimited memory and perfect scheduling
3. **Simplified Packet Processing:**
   - Real hardware: DMA, interrupts, buffer management overhead
   - NS-3: Instant packet transfer between nodes

Limitations of Approximation:

- Still event-driven, not continuous
- No modeling of hardware queue overflow behavior
- Cannot capture ASIC-specific optimizations
- Acceptable for: Studying algorithm behavior, relative performance
- Not acceptable for: Hardware validation, precise timing requirements

Gap 2: Deep Packet Inspection (DPI) for Application Recognition

Real-World Feature: Enterprise routers perform DPI to classify traffic based on:

- Layer 7 application signatures (HTTP, SSH, VoIP codecs)
- Behavioral analysis (packet sizes, timing patterns)
- Encrypted traffic fingerprinting (TLS SNI, certificate analysis)
- Machine learning-based classification

Why NS-3 Simulation is Challenging:

1. **No Payload Content:**
   - NS-3 packets are primarily headers + size
   - No actual application payload to inspect
   - Cannot parse HTTP headers, SIP messages, etc.
2. **Simplified Protocol Stack:**

- Application data not carried through network stack
- No SSL/TLS encryption layer to analyze
- Protocol-specific fields not populated realistically

3. **Performance Impact:**
    - Real DPI has significant CPU overhead
    - Can reduce router throughput by 30-50%
    - NS-3 cannot model computational constraints
4. **Signature Database:**
    - Real DPI uses constantly-updated signature databases
    - Behavioral patterns learned from production traffic
    - Cannot replicate proprietary classification algorithms

Limitations of Approximation:

- Classification happens at application, not router
- No inspection of packet contents
- Heuristics oversimplified (real DPI much more sophisticated)
- Cannot model encrypted traffic challenges
- No performance overhead modeling
- Acceptable for: Testing QoS policies with known traffic types
- Not acceptable for: Validating DPI algorithms, security resear

Gap 3: Adaptive Queue Management with Hardware Offload

Real-World Feature: Modern routers use sophisticated AQM (Active Queue Management):

- CoDel (Controlled Delay) with hardware timestamps
- PIE (Proportional Integral controller Enhanced) with ASIC support
- Adaptive RED with explicit congestion notification (ECN)
- Per-flow state tracking in hardware

Why NS-3 Simulation is Challenging:

1. **Hardware Timestamping:**
    - Real NICs timestamp packets in hardware upon arrival

- Sub-microsecond accuracy
- NS-3 uses simulated time with coarser granularity

2. **Stateful Per-Flow Tracking:**
    - Modern NICs track thousands of flows in hardware CAM/TCAM
    - Flow state updates at line rate
    - NS-3 software tracking has scaling limitations
3. **ECN Marking:**
    - Requires bit-level manipulation in hardware
    - Coordination with TCP congestion control
    - NS-3 has limited ECN support across all protocols
4. **Dynamic Adaptation:**
    - **Real** hardware adapts queue sizes based on:
        - Link utilization
        - Memory pressure
        - Temperature/power **constraints**
    - **NS-3 has static configuration**

Limitations of Approximation:

- Software-based, lacks hardware precision
- Simplified sojourn time calculation
- No true per-flow fairness
- Cannot model hardware memory constraints
- Missing ECN marking support
- No dynamic parameter adaptation
- Acceptable for: Algorithm validation, comparative studies
- Not acceptable for: Precise delay modeling, hardware verification

EXERCISE 3

Question 1: IPsec VPN Implementation Design

Implementation Design

A. NS-3 Modules and Approach

Primary Strategy: Custom Application Layer Encryption Proxy

Since NS-3 lacks built-in IPsec modules, we simulate IPsec by:

1. **Custom Application Layer**: Create a VpnApplication that wraps the actual application traffic
2. **Packet Tagging**: Use NS-3's packet tagging system to mark encrypted packets
3. **Header Overhead**: Add custom headers to simulate IPsec ESP (Encapsulating Security Payload) overhead
4. **Processing Delay**: Introduce artificial delays to simulate encryption/decryption operations

Required NS-3 Components:

- ns3:Application - Base class for custom VPN application
- ns3: Packet::AddHeader() - Add IPsec-like headers
- ns3: Tag - Mark packets as encrypted
- ns3::RandomVariableStream - Simulate variable encryption delays
- ns3::TraceSource - Monitor VPN tunnel statistics

B. Security Association Configuration

CPP code snippet

```cpp
// Define a Security Association structure
struct SecurityAssociation {
    Ipv4Address sourceIp;
    Ipv4Address destIp;
    uint32_t spi;              // Security Parameter Index
    std::string encryptAlgo;    // e.g., "AES-256"
    std::string authAlgo;       // e.g., "SHA-256"
    uint32_t keyLifetime;       // seconds
};

// Configure SA between n0 and n2
SecurityAssociation sa_n0_n2;
sa_n0_n2.sourceIp = Ipv4Address("10.1.1.1");
sa_n0_n2.destIp = Ipv4Address("10.1.2.2");
sa_n0_n2.spi = 12345;
sa_n0_n2.encryptAlgo = "AES-256-GCM";
sa_n0_n2.authAlgo = "SHA-256";
sa_n0_n2.keyLifetime = 3600; // 1 hour

```

VPN TUNNEL CPP IMPLEMENTATION

```
class IpsecVpnHelper {
public:
    void ConfigureTunnel(Ptr<Node> nodeA, Ptr<Node> nodeB,
                         SecurityAssociation sa);
    void AddPacketOverhead(Ptr<Packet> packet);
    Time GetEncryptionDelay();
    Time GetDecryptionDelay();
};

```

C. Expected Performance Overhead

Latency Increases:

Encryption Processing: +0.5-2ms per packet (simulated)

Decryption Processing: +0.5-2ms per packet (simulated)

Total Added Latency: ~1-4ms round-trip

Throughput Reduction:

Header Overhead: 50-73 bytes per packet (ESP header + trailer + authentication)

ESP Header: 8 bytes

ESP Trailer: 2-255 bytes (padding + pad length + next header)

Authentication Data: 12-32 bytes (depending on algorithm)

Effective Throughput: ~3-5% reduction due to overhead

CPU-Bound Scenarios: Up to 10-15% reduction in high-throughput scenarios

CPP IMPLEMENATION

```
// Add IPsec overhead to packet
void AddIpsecOverhead(Ptr<Packet> packet) {
    // ESP header (8 bytes) + Auth trailer (16 bytes for HMAC-SHA1-96)
    IpsecEspHeader espHeader;
    espHeader.SetSpi(12345);
    espHeader.SetSequenceNumber(seqNum++);
    packet->AddHeader(espHeader); // 8 bytes

    // Add authentication tag
    IpsecAuthTrailer authTrailer;
    authTrailer.SetAuthData(CalculateHmac(packet)); // 16 bytes
    packet->AddTrailer(authTrailer);

    // Total overhead: 24 bytes minimum
}
```

```
16.
17. // Simulate encryption delay
18. Time GetEncryptionDelay() {
19.     // AES-256-GCM typical processing: ~1-2ms on modern hardware
20.     Ptr<UniformRandomVariable> rand = CreateObject<UniformRandomVariable>();
21.     return MilliSeconds(rand->GetValue(0.5, 2.0));
22. }
23.
```

Question2: Eavesdropping Attack Simulation

Implementation Design

A. NS-3 Tracing Mechanisms

Method 1: Promiscuous Mode Packet Capture

```
 1. // Create attacker node positioned near the link
 2. Ptr<Node> attacker = CreateObject<Node>();
 3. nodes.Add(attacker);
 4.
 5. // Install promiscuous receive callback on the link
 6. Ptr<NetDevice> link2Device = link2Devices.Get(0); // Router side
 7. link2Device->SetPromiscReceiveCallback(
 8.     MakeCallback(&EavesdropCallback)
 9. );
10.
11. // Eavesdrop callback function
12. void EavesdropCallback(Ptr<NetDevice> device,
13.                        Ptr<const Packet> packet,
14.                        uint16_t protocol,
15.                        const Address& from,
16.                        const Address& to,
17.                        NetDevice::PacketType packetType) {
18.
19.     Ptr<Packet> copy = packet->Copy();
20.
21.     // Extract and log packet contents
22.     EthernetHeader ethHeader;
23.     copy->RemoveHeader(ethHeader);
24.
25.     Ipv4Header ipHeader;
26.     copy->RemoveHeader(ipHeader);
27.
28.     UdpHeader udpHeader;
29.     copy->RemoveHeader(udpHeader);
30.
31.     // Extract payload
32.     uint8_t buffer[1024];
33.     uint32_t size = copy->CopyData(buffer, 1024);
34.
35.     std::cout << "[ATTACKER] Intercepted packet from "
36.               << ipHeader.GetSource() << " to "
37.               << ipHeader.GetDestination() << std::endl;
38.     std::cout << "[ATTACKER] Payload size: " << size << " bytes" << std::endl;
39.     std::cout << "[ATTACKER] Payload data: ";
40.     for (uint32_t i = 0; i < size && i < 64; i++) {
41.         std::cout << buffer[i];
42.     }
43.     std::cout << std::endl;
44. }
```

Method 2: PCAP Analysis with Custom Tap

```
  // Create a tap device to mirror traffic
Ptr<TapBridge> tap = CreateObject<TapBridge>();
tap->SetAttribute("Mode", StringValue("UseLocal"));
tap->SetAttribute("DeviceName", StringValue("attacker-tap"));

// Attach to the link
NetDeviceContainer tapDevices;
tapDevices.Add(link2Devices.Get(1));

```

B. Sensitive Information Extraction

What Can Be Extracted from UdpEchoClient Packets:

1. **IP Layer Information:**
    - Source IP: 10.1.1.1 (client identity)
    - Destination IP: 10.1.2.2 (server identity)
    - Communication pattern: Timing, frequency, packet sizes
2. **Transport Layer Information:**
    - Source/destination ports: Identify application types
    - UDP payload size: 1024 bytes (from simulation)
3. **Application Layer Data:**
    - Complete UDP Echo payload (unencrypted)
    - Any sensitive business data transmitted
    - Application-level identifiers and session information

Example Extraction Code:

```
void AnalyzePacket(Ptr<const Packet> packet) {
    Ptr<Packet> copy = packet->Copy();

    // Extract headers
    Ipv4Header ipv4;
    copy->RemoveHeader(ipv4);

    UdpHeader udp;
    copy->RemoveHeader(udp);

    // Extract payload
    uint32_t payloadSize = copy->GetSize();
    uint8_t* payload = new uint8_t[payloadSize];
    copy->CopyData(payload, payloadSize);

    // Log extracted information
    std::ofstream attackLog("attacker-log.txt", std::ios::app);
    attackLog << "Time: " << Simulator::Now().GetSeconds() << "s" << std::endl;
    attackLog << "Source: " << ipv4.GetSource() << ":" << udp.GetSourcePort() << std::endl;
```

```
20.     attackLog << "Dest: " << ipv4.GetDestination() << ":" << udp.GetDestinationPort() <<
std::endl;
21.     attackLog << "Payload (" << payloadSize << " bytes): ";
22.     attackLog.write((char*)payload, payloadSize);
23.     attackLog << std::endl << "---" << std::endl;
24.
25.     delete[] payload;
26. }
27.
```

C. Demonstrating IPsec Effectiveness

Comparison Approach:

1. Baseline (No IPsec): Run simulation with eavesdropper logging all packet contents
2. With IPsec: Run simulation with VPN tunnel and show encrypted payloads

CPP CODE SNIPPET

```
 1. // Scenario 1: Without IPsec
 2. void ScenarioWithoutIpsec() {
 3.     // Normal traffic - attacker can read everything
 4.     std::cout << "[BASELINE] Running without IPsec..." << std::endl;
 5.     // ... setup normal echo client/server
 6.     // Attacker logs show readable payload
 7. }
 8.
 9. // Scenario 2: With IPsec
10. void ScenarioWithIpsec() {
11.     // Encrypted traffic - attacker sees gibberish
12.     std::cout << "[SECURE] Running with IPsec VPN..." << std::endl;
13.
14.     // Wrap echo client in VPN tunnel
15.     IpsecVpnHelper vpn;
16.     vpn.ConfigureTunnel(n0, n2, securityAssociation);
17.
18.     // Attacker logs show encrypted payload
19.     // Only sees: ESP header, encrypted blob, auth tag
20. }
21.
22. // Comparison Output
23. void CompareResults() {
24.     std::cout << "\n=== EAVESDROPPING COMPARISON ===" << std::endl;
25.     std::cout << "Without IPsec: Full payload visible to attacker" << std::endl;
26.     std::cout << "With IPsec: Only encrypted data visible" << std::endl;
27.     std::cout << "Payload confidentiality: PROTECTED ✓" << std::endl;
28.     std::cout << "Communication pattern: Still visible (traffic analysis possible)" <<
std::endl;
29. }
```

Metrics to Demonstrate Effectiveness:

- Percentage of readable payload data: 100% → 0%
- Application data exposure: Complete → None

- Metadata exposure: IP addresses, packet sizes, timing still visible (IPsec limitation)

3. DDoS Attack Simulation

Implementation Design

A. Creating Multiple Malicious Nodes

```
// Create botnet of malicious nodes
uint32_t numAttackers = 10;
NodeContainer attackers;
attackers.Create(numAttackers);

// Connect attackers to the network
// Option 1: All connected to same subnet as n0
// Option 2: Distributed across multiple networks (more realistic)

PointToPointHelper p2pAttacker;
p2pAttacker.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
p2pAttacker.SetChannelAttribute("Delay", StringValue("5ms"));

NetDeviceContainer attackerDevices;
for (uint32_t i = 0; i < numAttackers; i++) {
    NodeContainer link(attackers.Get(i), n1); // Connect to router
    NetDeviceContainer devices = p2pAttacker.Install(link);
    attackerDevices.Add(devices.Get(0));
}

// Install Internet stack
InternetStackHelper stack;
stack.Install(attackers);

// Assign IP addresses to attackers (10.1.3.0/24 subnet)
Ipv4AddressHelper addressAttacker;
addressAttacker.SetBase("10.1.3.0", "255.255.255.0");
Ipv4InterfaceContainer attackerInterfaces = addressAttacker.Assign(attackerDevices);

// Configure routing for attackers
for (uint32_t i = 0; i < numAttackers; i++) {
    Ptr<Ipv4StaticRouting> routing =
        staticRoutingHelper.GetStaticRouting(attackers.Get(i)->GetObject<Ipv4>());
    routing->AddNetworkRouteTo(Ipv4Address("10.1.2.0"),
                               Ipv4Mask("255.255.255.0"),
                               Ipv4Address("10.1.3.1"), 1);
}

```

B. Traffic Pattern Generation

UDP Flood Attck:

```
  // Create high-rate UDP flood from each attacker
class UdpFloodApplication : public Application {
private:
    Ptr<Socket> m_socket;
    Address m_peer;
    uint32_t m_packetSize;
    DataRate m_dataRate;
    EventId m_sendEvent;

public:
    void Setup(Address address, uint32_t packetSize, DataRate dataRate) {
```

```
        m_peer = address;
        m_packetSize = packetSize;
        m_dataRate = dataRate;
    }

    void StartApplication() override {
        m_socket = Socket::CreateSocket(GetNode(), UdpSocketFactory::GetTypeId());
        m_socket->Connect(m_peer);
        SendPacket();
    }

    void SendPacket() {
        Ptr<Packet> packet = Create<Packet>(m_packetSize);
        m_socket->Send(packet);

        // Calculate next send time for desired rate
        Time nextTime = Seconds(m_packetSize * 8.0 / m_dataRate.GetBitRate());
        m_sendEvent = Simulator::Schedule(nextTime, &UdpFloodApplication::SendPacket, this);
    }

    void StopApplication() override {
        if (m_sendEvent.IsRunning()) {
            Simulator::Cancel(m_sendEvent);
        }
        if (m_socket) {
            m_socket->Close();
        }
    }
};

// Install flood applications on all attackers
InetSocketAddress serverAddress(interfaces2.GetAddress(1), 9); // Target server

for (uint32_t i = 0; i < numAttackers; i++) {
    Ptr<UdpFloodApplication> floodApp = CreateObject<UdpFloodApplication>();
    floodApp->Setup(serverAddress, 512, DataRate("1Mbps")); // 1 Mbps per attacker
    attackers.Get(i)->AddApplication(floodApp);
    floodApp->SetStartTime(Seconds(3.0)); // Start attack at t=3s
    floodApp->SetStopTime(Seconds(8.0));
}

// Total attack traffic: 10 attackers × 1 Mbps = 10 Mbps

```

SYN Flood Attack (Alternative):

```
// For TCP SYN flood (more complex)
class SynFloodApplication : public Application {
private:
    void SendSyn() {
        // Create TCP socket
        Ptr<Socket> socket = Socket::CreateSocket(GetNode(), TcpSocketFactory::GetTypeId());

        // Initiate connection (SYN) but never complete handshake
        socket->Connect(m_peer);

        // Immediately abandon socket (don't complete handshake)
        // This fills server's SYN queue

        // Schedule next SYN
        Simulator::Schedule(MilliSeconds(10), &SynFloodApplication::SendSyn, this);
    }
};

```

C. Measuring Impact on Legitimate Traffic

Metrics to Track:

```
// Track legitimate client statistics
class TrafficMonitor {
private:
    uint32_t m_legitimatePacketsSent;
    uint32_t m_legitimatePacketsReceived;
    Time m_totalLatency;
    uint32_t m_packetsDropped;

public:
    void PacketSent(Ptr<const Packet> packet) {
        m_legitimatePacketsSent++;
        // Tag packet with timestamp
        TimestampTag tag;
        tag.SetTimestamp(Simulator::Now());
        packet->AddPacketTag(tag);
    }

    void PacketReceived(Ptr<const Packet> packet) {
        m_legitimatePacketsReceived++;

        // Calculate latency
        TimestampTag tag;
        if (packet->FindFirstMatchingByteTag(tag)) {
            Time latency = Simulator::Now() - tag.GetTimestamp();
            m_totalLatency += latency;
        }
    }

    void PrintStatistics() {
        double packetLoss = (m_legitimatePacketsSent - m_legitimatePacketsReceived)
                            / (double)m_legitimatePacketsSent * 100.0;
        Time avgLatency = m_totalLatency / m_legitimatePacketsReceived;

        std::cout << "=== Legitimate Traffic Impact ===" << std::endl;
        std::cout << "Packets sent: " << m_legitimatePacketsSent << std::endl;
        std::cout << "Packets received: " << m_legitimatePacketsReceived << std::endl;
        std::cout << "Packet loss: " << packetLoss << "%" << std::endl;
        std::cout << "Average latency: " << avgLatency.GetMilliSeconds() << " ms" <<
std::endl;
    }
};

// Monitor queue sizes at router
Ptr<PointToPointNetDevice> routerDevice =
DynamicCast<PointToPointNetDevice>(link2Devices.Get(0));
Ptr<Queue<Packet>> queue = routerDevice->GetQueue();

// Log queue statistics
void MonitorQueue() {
    std::cout << "Queue size: " << queue->GetNPackets()
              << " / " << queue->GetMaxSize() << std::endl;
    std::cout << "Queue drops: " << queue->GetTotalDroppedPackets() << std::endl;

    Simulator::Schedule(Seconds(0.1), &MonitorQueue);
}

```

Expected Impact:

- Without Attack: Packet loss ~0%, latency ~4ms

- During Attack: Packet loss 20-80%, latency 50-500ms or timeout
- Server Saturation: CPU/bandwidth exhaustion, service unavailable

Question4: Defense Mechanisms

A. Rate Limiting on Router Interfaces

Implementation in NS-3

Using Token Bucket Algorithm:

```
// Custom queue discipline with rate limiting
class RateLimitQueue : public Queue<Packet> {
private:
    double m_rate;           // tokens per second
    double m_bucketSize;     // maximum burst
    double m_tokens;         // current tokens
    Time m_lastUpdate;

public:
    RateLimitQueue() : m_rate(1000000), m_bucketSize(10000), m_tokens(10000) {
        m_lastUpdate = Simulator::Now();
    }

    void SetRate(DataRate rate) {
        m_rate = rate.GetBitRate() / 8.0; // Convert to bytes per second
    }

    bool Enqueue(Ptr<Packet> packet) override {
        // Update token bucket
        Time now = Simulator::Now();
        double elapsed = (now - m_lastUpdate).GetSeconds();
        m_tokens = std::min(m_bucketSize, m_tokens + elapsed * m_rate);
        m_lastUpdate = now;

        uint32_t packetSize = packet->GetSize();

        // Check if enough tokens
        if (m_tokens >= packetSize) {
            m_tokens -= packetSize;
            return Queue<Packet>::Enqueue(packet);
        } else {
            // Drop packet - rate limit exceeded
            DropBeforeEnqueue(packet, "Rate limit exceeded");
            return false;
        }
    }
};

// Install rate limiting on router interface
Ptr<PointToPointNetDevice> routerDev =
DynamicCast<PointToPointNetDevice>(link2Devices.Get(0));
Ptr<RateLimitQueue> rateLimiter = CreateObject<RateLimitQueue>();
rateLimiter->SetRate(DataRate("3Mbps")); // Limit to 3 Mbps
rateLimiter->SetMaxSize(QueueSize("100p")); // 100 packet buffer

routerDev->SetQueue(rateLimiter);

```

Limitations:

- NS-3's queue model is simplified compared to real router queuing
- No per-flow fairness (all traffic treated equally)
- Cannot distinguish between legitimate and **attack traffic without additional logic**

```
class PerSourceRateLimiter : public Queue<Packet> {
private:
    std::map<Ipv4Address, TokenBucket> m_buckets;
    DataRate m_perSourceRate;

public:
    bool Enqueue(Ptr<Packet> packet) override {
        // Extract source IP
        Ipv4Header ipHeader;
        packet->PeekHeader(ipHeader);
        Ipv4Address source = ipHeader.GetSource();

        // Get or create token bucket for this source
        if (m_buckets.find(source) == m_buckets.end()) {
            m_buckets[source] = TokenBucket(m_perSourceRate);
        }

        // Check rate limit for this source
        if (m_buckets[source].HasTokens(packet->GetSize())) {
            m_buckets[source].ConsumeTokens(packet->GetSize());
            return Queue<Packet>::Enqueue(packet);
        } else {
            DropBeforeEnqueue(packet, "Per-source rate limit");
            return false;
        }
    }
};

```

B. Access Control Lists (ACLs)

Implementation in NS-3

```
Packet Filter Approach:
// ACL Entry structure
struct AclEntry {
    Ipv4Address sourceIp;
    Ipv4Mask sourceMask;
    Ipv4Address destIp;
    Ipv4Mask destMask;
    uint16_t protocol;  // TCP=6, UDP=17, ANY=0
    uint16_t destPort;
    bool allow;         // true=permit, false=deny
};

// ACL Filter class
class AclFilter {
private:
    std::vector<AclEntry> m_entries;

public:
    void AddEntry(AclEntry entry) {
        m_entries.push_back(entry);
    }
```

```
22.
23.     bool CheckPacket(Ptr<const Packet> packet) {
24.         Ptr<Packet> copy = packet->Copy();
25.
26.         // Extract headers
27.         Ipv4Header ipHeader;
28.         copy->RemoveHeader(ipHeader);
29.
30.         uint16_t destPort = 0;
31.         if (ipHeader.GetProtocol() == 17) { // UDP
32.             UdpHeader udpHeader;
33.             copy->PeekHeader(udpHeader);
34.             destPort = udpHeader.GetDestinationPort();
35.         }
36.
37.         // Check against ACL entries (first match wins)
38.         for (const auto& entry : m_entries) {
39.             bool srcMatch = entry.sourceMask.IsMatch(ipHeader.GetSource(), entry.sourceIp);
40.             bool dstMatch = entry.destMask.IsMatch(ipHeader.GetDestination(), entry.destIp);
41.             bool protoMatch = (entry.protocol == 0 || entry.protocol ==
ipHeader.GetProtocol());
42.             bool portMatch = (entry.destPort == 0 || entry.destPort == destPort);
43.
44.             if (srcMatch && dstMatch && protoMatch && portMatch) {
45.                 return entry.allow; // Return permit/deny
46.             }
47.         }
48.
49.         return false; // Default deny
50.     }
51. };
52.
53. // Install ACL on router
54. Ptr<NetDevice> routerDevice = link2Devices.Get(0);
55.
56. AclFilter acl;
57.
58. // Allow legitimate client (10.1.1.1)
59. AclEntry allowLegitimate;
60. allowLegitimate.sourceIp = Ipv4Address("10.1.1.1");
61. allowLegitimate.sourceMask = Ipv4Mask("255.255.255.255");
62. allowLegitimate.destIp = Ipv4Address("10.1.2.2");
63. allowLegitimate.destMask = Ipv4Mask("255.255.255.255");
64. allowLegitimate.protocol = 17; // UDP
65. allowLegitimate.destPort = 9;
66. allowLegitimate.allow = true;
67. acl.AddEntry(allowLegitimate);
68.
69. // Block attacker subnet (10.1.3.0/24)
70. AclEntry blockAttackers;
71. blockAttackers.sourceIp = Ipv4Address("10.1.3.0");
72. blockAttackers.sourceMask = Ipv4Mask("255.255.255.0");
73. blockAttackers.destIp = Ipv4Address("0.0.0.0");
74. blockAttackers.destMask = Ipv4Mask("0.0.0.0");
75. blockAttackers.protocol = 0; // Any protocol
76. blockAttackers.destPort = 0;
77. blockAttackers.allow = false;
78. acl.AddEntry(blockAttackers);
79.
80. // Apply ACL as receive callback
81. routerDevice->SetReceiveCallback(MakeCallback(&ApplyAcl));
82.
83. bool ApplyAcl(Ptr<NetDevice> device, Ptr<const Packet> packet,
84.               uint16_t protocol, const Address& from) {
85.     if (!acl.CheckPacket(packet)) {
86.         // Packet denied - drop it
87.         std::cout << "ACL: Packet dropped" << std::endl;
88.         return false;
89.     }
90.     // Packet allowed - continue processing
```

```
    return true;
}
```

Limitations:

- NS-3 doesn't have native ACL support - requires custom implementation
- Stateless filtering only (no connection tracking like real firewalls)
- Limited to simple IP/port matching
- Cannot perform deep packet inspection
- No dynamic updates during simulation

C. Anycast or Load Balancing

Implementation in NS-3

Load Balancer Approach:

```
// Create multiple backend servers
NodeContainer servers;
servers.Create(3);

// Load balancer node (replaces single server)
Ptr<Node> loadBalancer = CreateObject<Node>();

// Connect load balancer to router
NodeContainer lbLink(n1, loadBalancer);
NetDeviceContainer lbDevices = p2p.Install(lbLink);

// Connect backend servers to load balancer
PointToPointHelper p2pBackend;
p2pBackend.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
p2pBackend.SetChannelAttribute("Delay", StringValue("1ms"));

NetDeviceContainer backendDevices;
for (uint32_t i = 0; i < servers.GetN(); i++) {
    NodeContainer serverLink(loadBalancer, servers.Get(i));
    NetDeviceContainer devices = p2pBackend.Install(serverLink);
    backendDevices.Add(devices);
}

// Load balancer application
class LoadBalancerApplication : public Application {
private:
    std::vector<Ipv4Address> m_backends;
    uint32_t m_currentBackend;
    Ptr<Socket> m_socket;

public:
    void AddBackend(Ipv4Address backend) {
        m_backends.push_back(backend);
    }

    void StartApplication() override {
```

```
        m_socket = Socket::CreateSocket(GetNode(), UdpSocketFactory::GetTypeId());
        m_socket->Bind(InetSocketAddress(Ipv4Address::GetAny(), 9));
        m_socket->SetRecvCallback(MakeCallback(&LoadBalancerApplication::HandlePacket,
this));
    }

    void HandlePacket(Ptr<Socket> socket) {
        Ptr<Packet> packet;
        Address from;
        packet = socket->RecvFrom(from);

        // Round-robin load balancing
        Ipv4Address backend = m_backends[m_currentBackend];
        m_currentBackend = (m_currentBackend + 1) % m_backends.size();

        // Forward packet to selected backend
        Ptr<Socket> backendSocket = Socket::CreateSocket(GetNode(),
UdpSocketFactory::GetTypeId());
        backendSocket->Connect(InetSocketAddress(backend, 9));
        backendSocket->Send(packet);

        std::cout << "Load balancer: Forwarded packet to " << backend << std::endl;
    }
};

// Install load balancer
Ptr<LoadBalancerApplication> lb = CreateObject<LoadBalancerApplication>();
for (uint32_t i = 0; i < servers.GetN(); i++) {
    lb->AddBackend(serverInterfaces.GetAddress(i));
}
loadBalancer->AddApplication(lb);
lb->SetStartTime(Seconds(0.0));

```

Anycast Implementation:

```
// Multiple servers share same IP address (anycast)
Ipv4Address anycastIp("10.1.2.100");

// Configure routing to distribute traffic
for (uint32_t i = 0; i < servers.GetN(); i++) {
    // Assign anycast IP as secondary address
    Ptr<Ipv4> ipv4 = servers.Get(i)->GetObject<Ipv4>();
    uint32_t ifIndex = ipv4->AddInterface(CreateObject<Ipv4L3Protocol>());
    ipv4->AddAddress(ifIndex, Ipv4InterfaceAddress(anycastIp, Ipv4Mask("255.255.255.255")));
}

// Router uses ECMP (Equal Cost Multi-Path) to distribute
// NS-3 requires custom routing logic for true ECMP

```

Limitations:

- NS-3 doesn't have native load balancing support
- No health checking of backend servers
- Stateless load balancing (no session persistence)
- ECMP routing is limited in NS-3

- Cannot simulate sophisticated algorithms (least connections, weighted round-robin)

Question 5: Security vs. Performance Trade-off Analysis

A. IPsec Throughput Reduction

Expected Impact:

Without IPsec:

- Baseline throughput: 5 Mbps (link capacity)
- Effective goodput: ~4.9 Mbps (accounting for headers)
- Latency: 4ms (2ms × 2 links)

With IPsec:

- Header overhead: ~24-50 bytes per packet
- For 1024-byte payload: ~2.3-4.7% overhead
- Encryption processing: +0.5-2ms per direction
- Expected throughput: 4.7-4.8 Mbps (3-5% reduction)
- Expected latency: 6-8ms (50-100% increase)

B. DDoS Protection Latency Impact

Defense Mechanisms and Their Overhead:

Defense Mechanism	Added Latency	Throughput Impact	CPU Overhead
Rate Limiting	**0.1-0.5ms**	**0% (drops excess)**	**LOW**
ACL Filtering	**0.05-0.02ms**	**0% (drops excess)**	**LOW**
Deep Packet Inspection	**2-10ms**	**5-10%**	**HIGH**
Load Balancing	**0.5-2ms**	**+20% capacity**	**MEDIUM**

Combined Defense Stack:

- ACL + Rate Limiting: ~0.3-0.7ms added latency
- Full Stack (ACL + Rate Limit + Load Balance): ~1-3ms added latency
- During active attack: Latency may spike to 50-500ms without defenses

Simulation Code for Measuring Defense Impact:

```
class DefenseMetrics {
private:
    uint32_t m_packetsInspected;
    uint32_t m_packetsDropped;
    Time m_totalProcessingTime;

public:
    void RecordInspection(Time processingTime, bool dropped) {
        m_packetsInspected++;
        m_totalProcessingTime += processingTime;
        if (dropped) m_packetsDropped++;
    }

    void PrintMetrics() {
        std::cout << "Defense Metrics:" << std::endl;
        std::cout << "  Packets inspected: " << m_packetsInspected << std::endl;
        std::cout << "  Packets dropped: " << m_packetsDropped
                  << " (" << (m_packetsDropped * 100.0 / m_packetsInspected)
                  << "%)" << std::endl;
        std::cout << "  Avg processing time: "
                  << (m_totalProcessingTime.GetMicroSeconds() / m_packetsInspected)
                  << " µs" << std::endl;
    }
};

```

C. Balanced Security Posture Recommendations

Based on simulation insights, here's a recommended layered security approach:

Tier 1: Essential Security (Minimal Performance Impact)

Configuration:

- Basic ACLs on router: Block known malicious IPs/subnets
- Connection rate limiting: 1000 connections/sec per source IP
- Simple stateless firewall rules

Performance Cost:

- Latency: +0.5ms average
- Throughput: No reduction (only drops malicious traffic)
- CPU: <5% overhead

Protection Level:

- Blocks: Simple floods, known bad actors
- Does NOT block: Sophisticated DDoS, zero-day attacks, insider threats

Implementation:

```
// Minimal security configuration
```

```
2. AclFilter basicAcl;
3. basicAcl.AddEntry(blockKnownAttackers);
4. basicAcl.AddEntry(allowLegitimate);
5.
6. RateLimiter connectionLimit;
7. connectionLimit.SetLimit(1000); // connections per second
8.
9.
```

Tier 2: Standard Security (Moderate Performance Impact)

Configuration:

- IPsec VPN for all inter-site traffic
- Per-source rate limiting: 100 Kbps per unknown source
- Stateful connection tracking
- Basic anomaly detection

Performance Cost:

- Latency: +2-4ms average
- Throughput: -5-8% reduction
- CPU: 10-15% overhead

Protection Level:

- Blocks: Most DDoS attacks, eavesdropping, MITM attacks
- Does NOT block: Application-layer attacks, insider threats

Implementation:

```
 1. // Standard security configuration
 2. IpsecVpnHelper vpn;
 3. vpn.EnableEncryption("AES-256-GCM");
 4. vpn.EnableAuthentication("HMAC-SHA256");
 5.
 6. PerSourceRateLimiter rateLimiter;
 7. rateLimiter.SetUnknownSourceLimit(DataRate("100Kbps"));
 8. rateLimiter.SetKnownSourceLimit(DataRate("5Mbps"));
 9.
10. StatefulFirewall firewall;
11. firewall.EnableConnectionTracking();
12. firewall.SetTimeout(Seconds(300));
13.
```

Tier 3: High Security (Higher Performance Impact)

Configuration:

- IPsec with perfect forward secrecy (PFS)
- Deep packet inspection (DPI)

- Machine learning-based anomaly detection
- Load balancing with health checks
- Geographic IP filtering

Performance Cost:

- Latency: +5-10ms average
- Throughput: -10-15% reduction
- CPU: 25-40% overhead

Protection Level:

- Blocks: Advanced persistent threats, zero-day exploits, application-layer attacks
- Comprehensive visibility and logging

Implementation:

```
// High security configuration
IpsecVpnHelper advancedVpn;
advancedVpn.EnablePFS(true);
advancedVpn.SetRekeyInterval(Seconds(3600)); // Rekey every hour

DeepPacketInspector dpi;
dpi.EnablePatternMatching();
dpi.EnableProtocolValidation();
dpi.EnableMalwareSignatures();

AnomalyDetector ml;
ml.SetBaselineTraffic(normalTrafficProfile);
ml.SetSensitivity(0.85);

LoadBalancer lb;
lb.AddBackends(serverPool);
lb.EnableHealthChecks(Seconds(5));

```

D. Recommended Configuration by Use Case

Case 1: Small Office/Branch Office

- Recommendation: Tier 1 (Essential)
- Rationale: Limited attack surface, cost-sensitive
- Config: Basic ACLs + connection rate limiting
- Performance: <1ms latency impact

Case 2: Corporate WAN

- Recommendation: Tier 2 (Standard)
- Rationale: Balance security and performance
- Config: IPsec VPN + per-source rate limiting + stateful firewall

- Performance: 2-4ms latency, 5-8% throughput reduction
- This matches the original simulation scenario

Case 3: Financial Services / Healthcare

- Recommendation: Tier 3 (High Security)
- Rationale: Regulatory compliance, sensitive data
- Config: Full defense stack with DPI and ML anomaly detection
- Performance: 5-10ms latency, 10-15% throughput reduction

Case 4: Cloud Service Provider

- Recommendation: Tier 3 + Custom
- Rationale: High attack volume, need for scalability
- Config: Distributed DDoS mitigation, anycast, CDN
- Performance: Variable based on attack mitigation effectiveness

E. Performance Trade-off Summary

Key Findings from Simulation:

1. **IPsec Overhead is Manageable:**
    - 3-5% throughput reduction
    - 1-2ms latency increase per hop
    - Worth the security benefit for sensitive data
2. **DDoS Defense Overhead Varies:**
    - Passive defenses (ACLs): Negligible impact (<0.5ms)
    - Active defenses (rate limiting): Low impact (~1ms)
    - Deep inspection: Significant impact (5-10ms)
3. **Layered Approach is Most Effective:**
    - Combine multiple lightweight defenses
    - Better than single heavyweight solution
    - ACL + Rate Limiting + IPsec = Tier 2 performance with strong protection
4. **Attack Prevention vs. Detection Trade-off:**
    - Prevention (ACLs, rate limiting): Low overhead, limited scope
    - Detection (DPI, ML): High overhead, comprehensive coverage
    - Recommendation: Prevent known threats, detect unknown threats

EXERCISE 4

Question 1: Topology Analysis and Extension

Current Topology Assessment

The provided router-static-routing.cc simulates a simple two-network topology with basic static routing. To model RegionalBank's architecture, we need a three-node, four-network topology with redundancy.

Extended Topology Design

Required Modifications to Code Structure

Network Creation:

```
// Create nodes
NodeContainer branchC, dcA, drB;
branchC.Create(1);  // Branch-C client
dcA.Create(1);      // DC-A router (central)
drB.Create(1);      // DR-B server

// Network 1: Branch-C to DC-A (CSMA for LAN simulation)
NodeContainer net1Nodes(branchC.Get(0), dcA.Get(0));
CsmaHelper csma1;
csma1.SetChannelAttribute("DataRate", StringValue("100Mbps"));
csma1.SetChannelAttribute("Delay", TimeValue(MilliSeconds(2)));
NetDeviceContainer net1Devices = csma1.Install(net1Nodes);

// Network 2: DC-A to DR-B Primary Link (P2P for WAN)
NodeContainer net2Nodes(dcA.Get(0), drB.Get(0));
PointToPointHelper p2p2;
p2p2.SetDeviceAttribute("DataRate", StringValue("10Mbps"));
p2p2.SetChannelAttribute("Delay", StringValue("20ms"));
NetDeviceContainer net2Devices = p2p2.Install(net2Nodes);

// Network 3: DC-A to DR-B Backup Link (P2P for redundancy)
NodeContainer net3Nodes(dcA.Get(0), drB.Get(0));
PointToPointHelper p2p3;
p2p3.SetDeviceAttribute("DataRate", StringValue("5Mbps"));  // Lower bandwidth
p2p3.SetChannelAttribute("Delay", StringValue("50ms"));     // Higher latency
NetDeviceContainer net3Devices = p2p3.Install(net3Nodes);

```

```
```

IP Address Assignment:

Ipv4AddressHelper address;

```
// Network 1: 10.1.1.0/24
address.SetBase("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer net1Interfaces = address.Assign(net1Devices);
// Branch-C: 10.1.1.2, DC-A: 10.1.1.1

// Network 2: 10.1.2.0/24 (Primary)
address.SetBase("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer net2Interfaces = address.Assign(net2Devices);
```

```
// DC-A: 10.1.2.1, DR-B: 10.1.2.2

// Network 3: 10.1.3.0/24 (Backup)
address.SetBase("10.1.3.0", "255.255.255.0");
Ipv4InterfaceContainer net3Interfaces = address.Assign(net3Devices);
// DC-A: 10.1.3.1, DR-B: 10.1.3.2

```

Question 2: Static Routing Complexity

Routing Configuration for Normal and Backup Operation

Branch-C Routing Table Configuration:

```
Ptr<Ipv4> ipv4BranchC = branchC.Get(0)->GetObject<Ipv4>();
Ptr<Ipv4StaticRouting> staticRoutingBranchC =
    Ipv4RoutingHelper::GetRouting<Ipv4StaticRouting>(
        ipv4BranchC->GetRoutingProtocol()
    );

// Default route to DC-A for all external traffic
// Destination: 0.0.0.0, Mask: 0.0.0.0, Next-hop: 10.1.1.1, Interface: 1, Metric: 1
staticRoutingBranchC->AddNetworkRouteTo(
    Ipv4Address("0.0.0.0"),      // Any destination
    Ipv4Mask("0.0.0.0"),         // Any mask
    Ipv4Address("10.1.1.1"),     // Next hop: DC-A
    1                             // Interface index
);

```

DC-A Routing Table Configuration (Central Router):

```
Ptr<Ipv4> ipv4DcA = dcA.Get(0)->GetObject<Ipv4>();
Ptr<Ipv4StaticRouting> staticRoutingDcA =
    Ipv4RoutingHelper::GetRouting<Ipv4StaticRouting>(
        ipv4DcA->GetRoutingProtocol()
    );

// PRIMARY PATH: Route to DR-B via Network 2 (preferred)
// Metric: 10 (lower is better)
staticRoutingDcA->AddHostRouteTo(
    Ipv4Address("10.1.2.2"),     // DR-B primary interface
    Ipv4Address("10.1.2.2"),     // Next hop (directly connected)
    2,                            // Interface index (Network 2)
    10                            // Metric (primary - lower value)
);

// BACKUP PATH: Route to DR-B via Network 3 (backup)
// Metric: 20 (higher - only used if primary fails)
staticRoutingDcA->AddHostRouteTo(
    Ipv4Address("10.1.3.2"),     // DR-B backup interface
    Ipv4Address("10.1.3.2"),     // Next hop (directly connected)
    3,                            // Interface index (Network 3)
    20                            // Metric (backup - higher value)
);

// Route for reaching DR-B subnet via primary
staticRoutingDcA->AddNetworkRouteTo(
    Ipv4Address("10.1.2.0"),     // DR-B network
    Ipv4Mask("255.255.255.0"),
    2,                            // Interface to Network 2
    10                            // Primary metric
);

// Route back to Branch-C network
```

```
staticRoutingDcA->AddNetworkRouteTo(
    Ipv4Address("10.1.1.0"),     // Branch-C network
    Ipv4Mask("255.255.255.0"),
    1,                           // Interface to Network 1
    1                            // Direct connection
);

```

DR-B Routing Table Configuration:

```
Ptr<Ipv4> ipv4DrB = drB.Get(0)->GetObject<Ipv4>();
Ptr<Ipv4StaticRouting> staticRoutingDrB =
    Ipv4RoutingHelper::GetRouting<Ipv4StaticRouting>(
        ipv4DrB->GetRoutingProtocol()
    );

// PRIMARY PATH: Default route back via Network 2
staticRoutingDrB->AddNetworkRouteTo(
    Ipv4Address("0.0.0.0"),      // Default route
    Ipv4Mask("0.0.0.0"),
    Ipv4Address("10.1.2.1"),     // Next hop: DC-A via primary
    1,                           // Interface index (Network 2)
    10                           // Primary metric
);

// BACKUP PATH: Alternate route via Network 3
staticRoutingDrB->AddNetworkRouteTo(
    Ipv4Address("0.0.0.0"),      // Default route
    Ipv4Mask("0.0.0.0"),
    Ipv4Address("10.1.3.1"),     // Next hop: DC-A via backup
    2,                           // Interface index (Network 3)
    20                           // Backup metric
);
```

Administrative Distance and Metric Considerations

In Real Router Implementation:

Administrative Distance (AD):
- AD is used to prefer routes from more trustworthy sources
- Static routes typically have AD = 1 (highly trusted)
- In production routers (Cisco/Juniper):
  - Connected: AD = 0
  - Static: AD = 1
  - OSPF: AD = 110
  - RIP: AD = 120

Metric Implementation Strategy:
```
Primary Path Metric = 10
- Lower value = preferred
- Based on: bandwidth, reliability, load

Backup Path Metric = 20
- Higher value = only used when primary unavailable
- Typically 2x primary or higher

Calculation Formula (simplified):
Metric = (10^8 / Bandwidth_bps) + (Delay_ms / 10)

Primary: (10^8 / 10^7) + (20/10) = 10 + 2 = 12 ≈ 10
Backup: (10^8 / 5×10^6) + (50/10) = 20 + 5 = 25 ≈ 20

```

For Failover Implementation:

1. Floating Static Route: Backup route with higher AD (e.g., AD=5) only activates when primary (AD=1) disappears
2. Route Tracking: Monitor primary link health via IP SLA
3. Administrative Distance Override: Manually adjust AD during maintenance

NS-3 Limitation Note: NS-3's Ipv4StaticRouting doesn't fully support administrative distance in the same way production routers do. We simulate failover by manually removing/adding routes or using metrics to influence path selection.

Question 3: Simulating Link Failure

Programmatic Link Failure Implementation

Method 1: Disable NetDevice

```
// Function to simulate link failure
void SimulateLinkFailure(Ptr<NetDevice> device)
{
    NS_LOG_INFO("Simulating link failure at " << Simulator::Now().GetSeconds() << "s");

    // Disable the network device
    device->SetDown();

    NS_LOG_INFO("Link disabled - device is now DOWN");
}

// Function to restore link
void RestoreLink(Ptr<NetDevice> device)
{
    NS_LOG_INFO("Restoring link at " << Simulator::Now().GetSeconds() << "s");

    // Re-enable the network device
    device->SetUp();

    NS_LOG_INFO("Link restored - device is now UP");
}

// In main() function, after network setup:
int main()
{
    // ... [previous network setup code] ...

    // Get the NetDevice for DC-A's interface on Network 2 (primary link)
    Ptr<NetDevice> dcA_primaryDevice = net2Devices.Get(0); // DC-A side
    Ptr<NetDevice> drB_primaryDevice = net2Devices.Get(1); // DR-B side

    // Schedule link failure at t=5.0 seconds
    Simulator::Schedule(Seconds(5.0), &SimulateLinkFailure, dcA_primaryDevice);

    // Optional: Schedule link restoration at t=15.0 seconds
    // Simulator::Schedule(Seconds(15.0), &RestoreLink, dcA_primaryDevice);

    // ... [rest of simulation code] ...
}

```

Question 4: Convergence Analysis - Static vs Dynamic Routing

Static Routing Limitation Problem Statement: Static routing provides no automatic failover mechanism. When a link fails, packets continue to be forwarded to dead interfaces until manual reconfiguration occurs.

Static Routing Behavior: t=0s to t=5s: Traffic flows normally via primary path t=5s: Link failure occurs t=5s onwards: 100% packet loss - NO CONVERGENCE

Dynamic Routing Solution with OSPF

NS-3 OSPF Implementation:

```
#include "ns3/internet-apps-module.h"
#include "ns3/ospf-helper.h"  // Note: May need custom implementation

// IMPORTANT: NS-3 does not have native OSPF support in standard distribution
// Two options:
// 1. Use Quagga/FRR integration (ns3-dce module)
// 2. Use simplified link-state routing from internet-routing module

// Method 1: Using DCE with Quagga (Most Realistic)
void SetupOSPFWithQuagga()
{
    // Install DCE (Direct Code Execution)
    DceManagerHelper dceManager;
    dceManager.SetNetworkStack("ns3::LinuxStackHelper");
    dceManager.Install(dcA);
    dceManager.Install(drB);

    // Configure Quagga OSPF
    DceApplicationHelper quagga;
    quagga.SetBinary("zebra");
    quagga.SetStackSize(1 << 20);

    // Zebra configuration for DC-A
    quagga.ResetArguments();
    quagga.ResetEnvironment();
    quagga.AddArgument("-f");
    quagga.AddArgument("/etc/zebra/zebra-dcA.conf");
    ApplicationContainer zebraDcA = quagga.Install(dcA.Get(0));
    zebraDcA.Start(Seconds(0.0));

    // OSPF daemon for DC-A
    DceApplicationHelper ospfd;
    ospfd.SetBinary("ospfd");
    ospfd.SetStackSize(1 << 20);
    ospfd.AddArgument("-f");
    ospfd.AddArgument("/etc/ospf/ospfd-dcA.conf");
    ApplicationContainer ospfDcA = ospfd.Install(dcA.Get(0));
    ospfDcA.Start(Seconds(0.1));

    // Repeat for DR-B...
}

// Method 2: Simplified Approach with Custom Link-State Protocol
// (More practical for NS-3 simulation)

class SimplifiedOSPF : public Application
{
private:
    Ptr<Socket> m_socket;
    EventId m_helloEvent;
    EventId m_lsaEvent;
    std::map<Ipv4Address, Time> m_neighbors;
    std::map<Ipv4Address, uint32_t> m_linkStates;

```

```
    void SendHello();
    void SendLSA();
    void ReceivePacket(Ptr<Socket> socket);
    void RunDijkstra();
    void UpdateRoutingTable();

public:
    SimplifiedOSPF();
    virtual ~SimplifiedOSPF();

    static TypeId GetTypeId();
    void SetupOSPF(Ptr<Node> node);
};

// Practical Implementation using Ipv4GlobalRouting with periodic updates
void SetupDynamicRoutingApproximation()
{
    // Use Ipv4GlobalRoutingHelper with periodic recalculation
    Ipv4GlobalRoutingHelper::PopulateRoutingTables();

    // Schedule periodic routing updates to detect failures
    for (double t = 1.0; t < 20.0; t += 1.0)
    {
        Simulator::Schedule(Seconds(t),
            &Ipv4GlobalRoutingHelper::PopulateRoutingTables);
    }
}

```

Realistic OSPF Configuration (Conceptual):

```
// Configuration file approach (if using Quagga/FRR)

// DC-A OSPF Configuration (ospfd-dcA.conf):
/*
router ospf
  ospf router-id 10.1.1.1
  network 10.1.1.0/24 area 0.0.0.0
  network 10.1.2.0/24 area 0.0.0.0
  network 10.1.3.0/24 area 0.0.0.0

interface eth1
  ip ospf cost 10
  ip ospf hello-interval 10
  ip ospf dead-interval 40

interface eth2
  ip ospf cost 10
  ip ospf hello-interval 10
  ip ospf dead-interval 40

interface eth3
  ip ospf cost 20
  ip ospf hello-interval 10
  ip ospf dead-interval 40
*/

// DR-B OSPF Configuration (ospfd-drB.conf):
/*
router ospf
  ospf router-id 10.1.2.2
  network 10.1.2.0/24 area 0.0.0.0
  network 10.1.3.0/24 area 0.0.0.0

interface eth1
  ip ospf cost 10
  ip ospf hello-interval 10
  ip ospf dead-interval 40
```

```
38.
39. interface eth2
40.   ip ospf cost 20
41.   ip ospf hello-interval 10
42.   ip ospf dead-interval 40
43. */
44. ```
45.
46. ### Convergence Behavior Comparison
47.
48. **Timing Analysis:**
49.
50. | Phase | Static Routing | OSPF Dynamic Routing |
51. |-------|----------------|----------------------|
52. | **Normal Operation (t=0-5s)** | Packets via primary (10.1.2.x) | Packets via primary
(cost=10) |
53. | **Failure Detection (t=5s)** | Immediate (link down) | Hello timeout: ~10s intervals × 4 =
40s max |
54. | **Route Calculation** | N/A - no recalculation | SPF algorithm: ~100-500ms |
55. | **Route Installation** | N/A | FIB update: ~50-200ms |
56. | **Total Convergence** | NEVER (manual intervention required) | ~40-41 seconds worst case |
57. | **Optimized Convergence** | N/A | ~10-11s (with fast hello: 1s × 3 dead) |
58.
59. **OSPF Convergence Timeline:**
60. ```
61. t=0s:        Normal operation, primary path active
62. t=5s:        Physical link failure occurs
63. t=5s-15s:    OSPF hello packets fail (3 hello intervals)
64. t=15s:       Neighbor declared DEAD
65. t=15.1s:     New LSA generated and flooded
66. t=15.2s:     SPF calculation triggered on all routers
67. t=15.3s:     Backup path (10.1.3.x) installed in FIB
68. t=15.3s+:    Traffic reconverges via backup link
69.
```

Packet Loss Window:

- Static: 100% loss from t=5s onwards (infinite)
- OSPF: 100% loss from t=5s to t=15.3s (~10.3 seconds)

Key NS-3 Helper Classes for OSPF

Available Options:

1. **ns3-dce + Quagga/FRR (Most Realistic)**
    - Class: DceManagerHelper, DceApplicationHelper
    - Pros: Real routing daemon, accurate behavior
    - Cons: Complex setup, external dependencies
2. **Ipv4GlobalRouting (Built-in Approximation)**
    - Class: Ipv4GlobalRoutingHelper
    - Pros: Simple, built-in NS-3
    - Cons: Not true OSPF, manual triggering needed
3. **Custom Implementation**
    - Base Classes: Application, Ipv4RoutingProtocol

- Pros: Full control, educational
- Cons: Significant development effort

Recommended Approach for Simulation:

```
// Hybrid approach: Use GlobalRouting with event-driven updates

void RecalculateRoutesOnFailure()
{
    NS_LOG_INFO("Link state changed - recalculating routes");
    Ipv4GlobalRoutingHelper::PopulateRoutingTables();
}

int main()
{
    // ... network setup ...

    // Initial routing calculation
    Ipv4GlobalRoutingHelper::PopulateRoutingTables();

    // Schedule link failure
    Simulator::Schedule(Seconds(5.0), &SimulateLinkFailure,
                        net2Devices.Get(0));

    // Simulate OSPF convergence delay (40s worst case)
    Simulator::Schedule(Seconds(15.0), &RecalculateRoutesOnFailure);

    // Monitor convergence
    Simulator::Schedule(Seconds(16.0), &VerifyRouteConvergence);

    Simulator::Run();
    Simulator::Destroy();
}

```

Performance Metrics Comparison

Static Routing:

- Convergence Time: ∞ (infinite)
- Packet Loss: 100% after failure
- Network Downtime: Permanent until manual fix
- Complexity: Low (simple configuration)
- Scalability: Poor (manual updates required)

OSPF Dynamic Routing:

- Convergence Time: 10-40 seconds (tunable)
- Packet Loss: Limited to convergence window
- Network Downtime: Temporary (self-healing)
- Complexity: Medium (protocol overhead)
- Scalability: Excellent (automatic adaptation)

Question 5: Business Continuity Verification

Simulation Result Analysis Plan

Comprehensive Verification Framework:

5.1 FlowMonitor-Based Analysis

```
#include "ns3/flow-monitor-module.h"

// Setup FlowMonitor
Ptr<FlowMonitor> flowMonitor;
FlowMonitorHelper flowHelper;

int main()
{
    // ... network setup ...

    // Install FlowMonitor on all nodes
    flowMonitor = flowHelper.InstallAll();

    // Configure FlowMonitor
    flowMonitor->SetAttribute("DelayBinWidth", DoubleValue(0.001));
    flowMonitor->SetAttribute("JitterBinWidth", DoubleValue(0.001));
    flowMonitor->SetAttribute("PacketSizeBinWidth", DoubleValue(20));

    // Schedule periodic checks
    Simulator::Schedule(Seconds(4.0), &AnalyzeFlow, flowMonitor, "Pre-Failure");
    Simulator::Schedule(Seconds(7.0), &AnalyzeFlow, flowMonitor, "During-Failure");
    Simulator::Schedule(Seconds(16.0), &AnalyzeFlow, flowMonitor, "Post-Convergence");

    Simulator::Stop(Seconds(20.0));
    Simulator::Run();

    // Final analysis
    flowMonitor->CheckForLostPackets();
    Ptr<Ipv4FlowClassifier> classifier =
        DynamicCast<Ipv4FlowClassifier>(flowHelper.GetClassifier());

    FlowMonitor::FlowStatsContainer stats = flowMonitor->GetFlowStats();

    // Generate report
    GenerateDetailedReport(stats, classifier);

    // Export to XML for external analysis
    flowMonitor->SerializeToXmlFile("bank-failover-results.xml", true, true);

    Simulator::Destroy();
}

```

EXERCISE 5

Question 1

NS3-IMPLEMENTATION

Traffic Flow Design

Flow_Video (RTP-like traffic):

- Small packets (160-200 bytes - typical RTP payload)
- Periodic transmission (20ms intervals - 50 packets/sec)
- UDP-based
- **DSCP: EF (Expedited Forwarding - 0xB8)**

Flow_Data (FTP-like traffic):

- Large packets (1400 bytes - near MTU)
- Bursty pattern (variable intervals)
- TCP or UDP with large bursts
- DSCP: AF11 (Assured Forwarding - 0x28)

```
// Add these headers
#include "ns3/onoff-application.h"
#include "ns3/packet-sink.h"

// Traffic generation setup
void SetupTrafficFlows(Ptr<Node> client, Ptr<Node> server, Ipv4Address serverAddr)
{
    // --- Flow_Video: RTP-like traffic (small, periodic packets) ---
    // Using UDP with small packets, constant rate
    uint16_t videoPort = 5004; // Standard RTP port
    uint32_t videoPacketSize = 200; // Bytes (typical RTP payload)
    Time videoInterval = MilliSeconds(20); // 50 packets/sec (typical 20ms intervals)

    OnOffHelper videoSource("ns3::UdpSocketFactory",
                            InetSocketAddress(serverAddr, videoPort));
    videoSource.SetAttribute("PacketSize", UintegerValue(videoPacketSize));
    videoSource.SetAttribute("OnTime",
StringValue("ns3::ConstantRandomVariable[Constant=1.0]"));
    videoSource.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0.0]"));
    videoSource.SetAttribute("DataRate", DataRateValue(DataRate("80kbps"))); // ~50pps * 200B

    ApplicationContainer videoApps = videoSource.Install(client);
    videoApps.Start(Seconds(2.0));
    videoApps.Stop(Seconds(10.0));

    // Set DSCP for video (EF - Expedited Forwarding)
    Ptr<OnOffApplication> videoApp = DynamicCast<OnOffApplication>(videoApps.Get(0));
    videoApp->TraceConnect("Tx", "0", MakeCallback(&SetVideoDSCP));

    // Packet sink for video on server
    PacketSinkHelper videoSink("ns3::UdpSocketFactory",
                               InetSocketAddress(Ipv4Address::GetAny(), videoPort));
    ApplicationContainer videoSinkApps = videoSink.Install(server);
    videoSinkApps.Start(Seconds(1.0));
    videoSinkApps.Stop(Seconds(11.0));

    // --- Flow_Data: FTP-like traffic (large, bursty packets) ---
    uint16_t dataPort = 20; // FTP data port
    uint32_t dataPacketSize = 1460; // Bytes (typical TCP MSS)

    // Using bulk send application for TCP
    BulkSendHelper dataSource("ns3::TcpSocketFactory",
                              InetSocketAddress(serverAddr, dataPort));
    dataSource.SetAttribute("MaxBytes", UintegerValue(0)); // Unlimited
    dataSource.SetAttribute("SendSize", UintegerValue(dataPacketSize));

    ApplicationContainer dataApps = dataSource.Install(client);
    dataApps.Start(Seconds(2.5));
```

```
    dataApps.Stop(Seconds(9.5));

    // Packet sink for data on server
    PacketSinkHelper dataSink("ns3::TcpSocketFactory",
                              InetSocketAddress(Ipv4Address::GetAny(), dataPort));
    ApplicationContainer dataSinkApps = dataSink.Install(server);
    dataSinkApps.Start(Seconds(1.0));
    dataSinkApps.Stop(Seconds(11.0));
}

```

Question 2: PBR Implementation

```
// PBR Router Class
class PBRRouter : public Object
{
public:
    static TypeId GetTypeId();

    void Install(Ptr<Node> routerNode) {
        m_node = routerNode;
        Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4>();

        // Hook into packet forwarding
        ipv4->TraceConnectWithoutContext("Tx",
            MakeCallback(&PBRRouter::PacketForwarding, this));

        // Initialize forwarding tables
        m_forwardingTable["video"] = 1; // Default interface for video
        m_forwardingTable["data"] = 2;  // Default interface for data

        // Alternative paths (secondary WAN links)
        m_altPaths[1] = Ipv4Address("10.1.3.1"); // Alternative next-hop for interface 1
        m_altPaths[2] = Ipv4Address("10.1.4.1"); // Alternative next-hop for interface 2
    }

    void PacketForwarding(Ptr<const Packet> packet, Ptr<Ipv4> ipv4,
                          uint32_t interface) {
        // Extract packet headers for classification
        Ptr<Packet> pktCopy = packet->Copy();
        Ipv4Header ipHeader;
        pktCopy->PeekHeader(ipHeader);

        // Classification logic
        TrafficClass trafficClass = ClassifyTraffic(ipHeader, pktCopy);

        // PBR decision
        uint32_t outInterface = DetermineOutputInterface(trafficClass);

        // Modify forwarding if needed
        if (outInterface != interface) {
            // Re-route to alternative path
            Ipv4Address nextHop = GetNextHopForInterface(outInterface);
            // Implementation would modify packet's route here
            // This requires modifying NS-3's forwarding engine
        }
    }

private:
    enum TrafficClass { VIDEO, DATA, BEST_EFFORT };

    TrafficClass ClassifyTraffic(const Ipv4Header& header, Ptr<Packet> packet) {
        // Method 1: DSCP-based classification
        uint8_t dscp = header.GetDscp();
        if (dscp == 0x2E || dscp == 0x22) { // EF or AF41
            return VIDEO;
        }

```

```
        // Method 2: Port-based classification
        UdpHeader udpHeader;
        TcpHeader tcpHeader;
        if (packet->PeekHeader(udpHeader)) {
            uint16_t port = udpHeader.GetDestinationPort();
            if (port == 5004 || port == 5005) return VIDEO;
        }
        if (packet->PeekHeader(tcpHeader)) {
            uint16_t port = tcpHeader.GetDestinationPort();
            if (port == 20 || port == 21) return DATA;
        }

        return BEST_EFFORT;
    }

    uint32_t DetermineOutputInterface(TrafficClass tc) {
        // Check current path metrics
        double primaryLatency = GetPathLatency(1);
        double secondaryLatency = GetPathLatency(2);

        // Policy: Video uses lowest latency path
        if (tc == VIDEO) {
            if (primaryLatency > 0.03 && secondaryLatency < primaryLatency) { // 30ms
threshold
                return 2; // Switch to secondary
            }
            return m_forwardingTable["video"];
        }
        // Data uses highest available bandwidth
        else if (tc == DATA) {
            double primaryBW = GetAvailableBandwidth(1);
            double secondaryBW = GetAvailableBandwidth(2);
            return (secondaryBW > primaryBW) ? 2 : 1;
        }

        return 1; // Default
    }

    Ptr<Node> m_node;
    std::map<std::string, uint32_t> m_forwardingTable;
    std::map<uint32_t, Ipv4Address> m_altPaths;
};

```

Question 3: Path Characterization

Built a PathMonitor system that:

- Integrates with FlowMonitor for accurate metrics
- Tracks latency, jitter, packet loss, and bandwidth per interface
- Uses trace callbacks for real-time packet tracking
- Provides bandwidth estimation with sliding window

```
class PathMonitor : public Object
{
public:
    void Install(Ptr<Node> router) {
        // Install FlowMonitor on all nodes
        FlowMonitorHelper flowmon;
        Ptr<FlowMonitor> monitor = flowmon.InstallAll();

```

```
        // Trace sources for per-interface metrics
        Ptr<Ipv4> ipv4 = router->GetObject<Ipv4>();
        for (uint32_t i = 0; i < ipv4->GetNInterfaces(); i++) {
            Ptr<NetDevice> dev = ipv4->GetNetDevice(i);

            // Queue monitoring
            Ptr<PointToPointNetDevice> p2pDev = DynamicCast<PointToPointNetDevice>(dev);
            if (p2pDev) {
                Ptr<Queue<Packet>> queue = p2pDev->GetQueue();
                queue->TraceConnectWithoutContext("Enqueue",
                    MakeCallback(&PathMonitor::TrackQueueDelay, this));
            }

            // Interface throughput monitoring
            dev->TraceConnectWithoutContext("PhyTxBegin",
                MakeCallback(&PathMonitor::TrackInterfaceThroughput, this));
        }

        // Periodic metric collection
        Simulator::Schedule(Seconds(0.1), &PathMonitor::CollectMetrics, this);
    }

    void CollectMetrics() {
        // Get FlowMonitor statistics
        std::map<FlowId, FlowMonitor::FlowStats> stats = m_flowMonitor->GetFlowStats();

        for (auto& flow : stats) {
            // Calculate per-flow metrics
            double latency = flow.second.delaySum.GetSeconds() / flow.second.rxPackets;
            double jitter = CalculateJitter(flow.second);
            double packetLoss = (flow.second.lostPackets /
                                 (double)flow.second.rxPackets) * 100;

            // Update path quality database
            UpdatePathQuality(flow.first, latency, jitter, packetLoss);
        }

        // Schedule next collection
        Simulator::Schedule(Seconds(0.1), &PathMonitor::CollectMetrics, this);
    }

    double GetPathLatency(uint32_t interfaceId) {
        auto it = m_interfaceMetrics.find(interfaceId);
        if (it != m_interfaceMetrics.end()) {
            return it->second.avgLatency;
        }
        return 0.0;
    }

    double GetAvailableBandwidth(uint32_t interfaceId) {
        auto it = m_interfaceMetrics.find(interfaceId);
        if (it != m_interfaceMetrics.end()) {
            InterfaceMetrics& metrics = it->second;
            // Calculate available BW = capacity - current usage
            double usedBw = metrics.bytesTransmitted * 8 / metrics.measurementWindow;
            return metrics.linkCapacity - usedBw;
        }
        return 0.0;
    }

private:
    struct InterfaceMetrics {
        double avgLatency;
        double avgJitter;
        double packetLossRate;
        uint64_t bytesTransmitted;
        double linkCapacity;
        Time measurementWindow;
    };

```

```
    std::map<uint32_t, InterfaceMetrics> m_interfaceMetrics;
    Ptr<FlowMonitor> m_flowMonitor;
};

```

Question 4: Dynamic SD-WAN Controller

Implemented a full controller with:

- Periodic policy evaluation (every 1s)
- Multi-path management with failover
- Traffic class-specific policies
- Hysteresis to prevent path flapping
- Automatic PBR rule updates

```
class SDWANController : public Application
{
public:
    SDWANController() : m_updateInterval(Seconds(1.0)) {}

    void StartApplication() override {
        // Initialize path monitors for all routers
        for (auto router : m_routers) {
            Ptr<PathMonitor> monitor = CreateObject<PathMonitor>();
            monitor->Install(router);
            m_pathMonitors[router] = monitor;
        }

        // Start policy evaluation loop
        Simulator::Schedule(m_updateInterval, &SDWANController::EvaluatePolicies, this);
    }

    void EvaluatePolicies() {
        // Evaluate each policy rule
        for (auto& policy : m_policies) {
            bool triggered = CheckPolicyCondition(policy);
            if (triggered) {
                ExecutePolicyAction(policy);
            }
        }

        // Schedule next evaluation
        Simulator::Schedule(m_updateInterval, &SDWANController::EvaluatePolicies, this);
    }

    void AddPolicy(const PolicyRule& rule) {
        m_policies.push_back(rule);
    }

    void PushForwardingRule(Ptr<Node> router, const std::string& trafficClass,
                            uint32_t interface) {
        // Use remote procedure call or direct method invocation
        Ptr<PBRRouter> pbrRouter = router->GetObject<PBRRouter>();
        if (pbrRouter) {
            pbrRouter->UpdateForwardingTable(trafficClass, interface);

            // Log the change
            std::cout << Simulator::Now() << " Updated " << trafficClass
                      << " routing on router " << router->GetId()
                      << " to interface " << interface << std::endl;
        }
    }
```

```
48.
49. private:
50.     struct PolicyRule {
51.         std::string name;
52.         std::string trafficClass;
53.         std::string condition; // e.g., "latency > 0.03"
54.         std::string action;    // e.g., "switch_to_interface(2)"
55.         Time evaluationPeriod;
56.     };
57.
58.     bool CheckPolicyCondition(const PolicyRule& rule) {
59.         // Parse condition (simplified)
60.         if (rule.condition.find("latency") != std::string::npos) {
61.             // Extract threshold
62.             double threshold = 0.03; // 30ms
63.
64.             // Get current latency for video flows
65.             double currentLatency = 0.0;
66.             for (auto& monitor : m_pathMonitors) {
67.                 currentLatency = std::max(currentLatency,
68.                                           monitor.second->GetVideoLatency());
69.             }
70.
71.             return currentLatency > threshold;
72.         }
73.         return false;
74.     }
75.
76.     void ExecutePolicyAction(const PolicyRule& rule) {
77.         if (rule.action.find("switch_to_interface") != std::string::npos) {
78.             // Extract interface number
79.             uint32_t newInterface = 2; // Secondary link
80.
81.             // Update all routers
82.             for (auto router : m_routers) {
83.                 PushForwardingRule(router, rule.trafficClass, newInterface);
84.             }
85.         }
86.     }
87.
88.     std::vector<Ptr<Node>> m_routers;
89.     std::map<Ptr<Node>, Ptr<PathMonitor>> m_pathMonitors;
90.     std::vector<PolicyRule> m_policies;
91.     Time m_updateInterval;
92. };
93.
```

Question5. Validation and Trade-offs

Validation Methodology:

1. **Flow tracing: Use FlowMonitor to verify traffic paths**
2. **Packet marking: Add custom tags to packets and trace their routes**
3. **Performance metrics: Compare with and without PBR**

```
1. void ValidatePBR() {
2.     // Compare end-to-end latency for video flows
3.     double pbrLatency = GetFlowLatency("video");
4.     double defaultLatency = GetBaselineLatency("video");
5.     NS_ASSERT_MSG(pbrLatency < defaultLatency * 0.9,
6.                   "PBR should improve video latency");
7.
```

```
    // Verify traffic separation
    uint64_t videoOnPrimary = CountPacketsOnInterface("video", 1);
    uint64_t videoOnSecondary = CountPacketsOnInterface("video", 2);
    std::cout << "Video traffic distribution: "
              << videoOnPrimary << " on primary, "
              << videoOnSecondary << " on secondary" << std::endl;
}

```

Computational Overhead:

- Simulation overhead: Packet classification adds O(n) per packet processing
- Real-world comparison: Much higher in simulation due to:
    - Deep packet inspection in user space
    - Lack of hardware acceleration
    - Serial processing vs parallel ASICs

Scalability Limits:

1. Flow count: Linear increase in classification time
2. Memory usage: Per-flow state tracking grows with flow count
3. Policy complexity: Exponential growth with multiple conditions

Mitigations:

- **Implement flow caching (first packet classification only)**
- Use sampling for large flow counts
- Aggregate flows by DSCP/port ranges
- Limit policy evaluation frequency

Recommended Approach:

For production simulations:

1. Start with simplified classification (port-based only)
2. Use flow sampling for >1000 flows
3. Implement PBR logic as a separate module for reuse
4. Validate with small scenarios before scaling up

This architecture provides a realistic PBR/SD-WAN simulation framework while maintaining acceptable performance for research and prototyping purposes.

EXERCISE 6

Question1: Modeling Autonomous Systems in NS-3

a) Logical Grouping of Nodes

To model distinct ASes, I propose creating a custom container class that groups nodes by AS:

```
class AutonomousSystem {
private:
    uint32_t asNumber;
    NodeContainer internalNodes;
    std::vector<Ptr<Node>> borderRouters;
    std::map<uint32_t, Ptr<Node>> ixpConnections; // AS number -> border router

public:
    AutonomousSystem(uint32_t asn) : asNumber(asn) {}

    void AddInternalNode(Ptr<Node> node) {
        internalNodes.Add(node);
        // Tag node with AS membership
        node->AggregateObject(CreateObject<ASMembership>(asNumber));
    }

    void AddBorderRouter(Ptr<Node> router) {
        borderRouters.push_back(router);
        AddInternalNode(router);
    }

    Ptr<Node> GetBorderRouter(uint32_t peerAS) {
        return ixpConnections[peerAS];
    }
};

// Helper class to tag nodes with AS membership
class ASMembership : public Object {
private:
    uint32_t asNumber;
public:
    ASMembership(uint32_t asn) : asNumber(asn) {}
    uint32_t GetASNumber() const { return asNumber; }
};

```

b) Enforcing Internal Routing Confinement

To confine OSPF within an AS:

```
class IntraASRoutingHelper {
public:
    static void ConfigureOSPF(AutonomousSystem& as) {
        Ipv4ListRoutingHelper list;
        OspfHelper ospf;

        // Create OSPF instance confined to this AS
        for (uint32_t i = 0; i < as.GetInternalNodes().GetN(); i++) {
            Ptr<Node> node = as.GetInternalNodes().Get(i);

            // Only install OSPF on internal interfaces
            Ptr<Ipv4> ipv4 = node->GetObject<Ipv4>();
            for (uint32_t j = 1; j < ipv4->GetNInterfaces(); j++) {
                Ipv4InterfaceAddress addr = ipv4->GetAddress(j, 0);

                // Check if interface connects to node in same AS
                if (IsInternalInterface(node, j, as)) {
```

```
                    ospf.IncludeInterface(node, j);
                }
            }
        }

        // Exclude border/eBGP interfaces from OSPF
        ospf.ExcludeBorderInterfaces(as.GetBorderRouters());
    }

private:
    static bool IsInternalInterface(Ptr<Node> node, uint32_t ifIndex,
                                    AutonomousSystem& as) {
        // Check if connected node belongs to same AS
        Ptr<NetDevice> device = node->GetDevice(ifIndex);
        Ptr<Channel> channel = device->GetChannel();

        // Iterate through devices on this channel
        for (uint32_t i = 0; i < channel->GetNDevices(); i++) {
            Ptr<NetDevice> otherDevice = channel->GetDevice(i);
            if (otherDevice != device) {
                Ptr<Node> otherNode = otherDevice->GetNode();
                Ptr<ASMembership> membership =
                    otherNode->GetObject<ASMembership>();

                if (!membership || membership->GetASNumber() != as.GetASNumber()) {
                    return false; // External interface
                }
            }
        }
        return true; // Internal interface
    }
};

```

c) Establishing Peering Links at IXPs

```
class IXPHelper {
public:
    struct PeeringLink {
        Ptr<Node> router1;
        Ptr<Node> router2;
        uint32_t as1;
        uint32_t as2;
        NetDeviceContainer devices;
    };

    static PeeringLink CreatePeeringPoint(
        AutonomousSystem& as1,
        AutonomousSystem& as2,
        std::string ixpName,
        std::string dataRate = "100Mbps",
        std::string delay = "5ms") {

        PeeringLink peering;

        // Get or create border routers
        peering.router1 = as1.GetOrCreateBorderRouter(as2.GetASNumber());
        peering.router2 = as2.GetOrCreateBorderRouter(as1.GetASNumber());
        peering.as1 = as1.GetASNumber();
        peering.as2 = as2.GetASNumber();

        // Create point-to-point link
        PointToPointHelper p2p;
        p2p.SetDeviceAttribute("DataRate", StringValue(dataRate));
        p2p.SetChannelAttribute("Delay", StringValue(delay));

        NodeContainer peeringNodes(peering.router1, peering.router2);
        peering.devices = p2p.Install(peeringNodes);
```

```
33.
34.         // Assign IP addresses from IXP subnet
35.         Ipv4AddressHelper address;
36.         address.SetBase(GetIXPSubnet(ixpName), "255.255.255.252");
37.         address.Assign(peering.devices);
38.
39.         // Mark these interfaces as eBGP interfaces
40.         MarkAsExternalInterface(peering.router1, peering.devices.Get(0));
41.         MarkAsExternalInterface(peering.router2, peering.devices.Get(1));
42.
43.         return peering;
44.     }
45.
46. private:
47.     static void MarkAsExternalInterface(Ptr<Node> node, Ptr<NetDevice> device) {
48.         // Tag interface as eBGP-enabled
49.         device->AggregateObject(CreateObject<BGPInterface>(true)); // true = external
50.     }
51.
52.     static const char* GetIXPSubnet(std::string ixpName) {
53.         static std::map<std::string, const char*> ixpSubnets = {
54.             {"IXP-A", "203.0.113.0"},
55.             {"IXP-B", "198.51.100.0"}
56.         };
57.         return ixpSubnets[ixpName];
58.     }
59. };
60.
```

Question 2

```
 1. // BGP Route Announcement Data Structure for NS-3 Simulation
 2.
 3. #include <vector>
 4. #include <string>
 5. #include <cstdint>
 6. #include "ns3/ipv4-address.h"
 7.
 8. namespace ns3 {
 9.
10. // Represents a BGP route announcement with path attributes
11. class BGPRoute {
12. public:
13.     // Network prefix information
14.     struct NetworkPrefix {
15.         Ipv4Address network;
16.         Ipv4Mask mask;
17.
18.         NetworkPrefix() : network("0.0.0.0"), mask("0.0.0.0") {}
19.         NetworkPrefix(Ipv4Address net, Ipv4Mask m) : network(net), mask(m) {}
20.
21.         bool operator==(const NetworkPrefix& other) const {
22.             return network == other.network && mask == other.mask;
23.         }
24.
25.         std::string ToString() const {
26.             std::ostringstream oss;
27.             oss << network << "/" << mask.GetPrefixLength();
28.             return oss.str();
29.         }
30.     };
31.
32.     // BGP Path Attributes
```

```
    NetworkPrefix prefix;
    std::vector<uint32_t> asPath;        // AS_PATH attribute (list of AS numbers)
    uint32_t localPref;                   // LOCAL_PREF (higher is better)
    uint32_t multiExitDisc;               // MED (lower is better)
    Ipv4Address nextHop;                  // NEXT_HOP attribute
    uint32_t origin;                      // ORIGIN (0=IGP, 1=EGP, 2=INCOMPLETE)

    // Additional metadata
    uint32_t receivedFromAS;              // AS number of neighbor who sent this
    Time timestamp;                       // When route was received
    bool isWithdrawn;                     // true if this is a withdrawal

    // Constructor
    BGPRoute()
        : prefix(),
          localPref(100),                 // Default LOCAL_PREF
          multiExitDisc(0),
          nextHop("0.0.0.0"),
          origin(0),
          receivedFromAS(0),
          timestamp(Seconds(0)),
          isWithdrawn(false) {}

    BGPRoute(NetworkPrefix pfx, std::vector<uint32_t> path)
        : prefix(pfx),
          asPath(path),
          localPref(100),
          multiExitDisc(0),
          nextHop("0.0.0.0"),
          origin(0),
          receivedFromAS(0),
          timestamp(Seconds(0)),
          isWithdrawn(false) {}

    // Check if route contains a loop (our AS in the path)
    bool HasLoop(uint32_t ourAS) const {
        for (uint32_t as : asPath) {
            if (as == ourAS) return true;
        }
        return false;
    }

    // Get AS_PATH length
    uint32_t GetASPathLength() const {
        return asPath.size();
    }

    // Prepend our AS to the AS_PATH (when advertising to neighbors)
    void PrependAS(uint32_t ourAS) {
        asPath.insert(asPath.begin(), ourAS);
    }

    // Print route for debugging
    std::string ToString() const {
        std::ostringstream oss;
        oss << "Prefix: " << prefix.ToString()
            << ", AS_PATH: [";
        for (size_t i = 0; i < asPath.size(); i++) {
            oss << asPath[i];
            if (i < asPath.size() - 1) oss << " ";
        }
        oss << "], LOCAL_PREF: " << localPref
            << ", MED: " << multiExitDisc
            << ", NEXT_HOP: " << nextHop
            << ", Origin: " << origin;
        return oss.str();
    }
};

// BGP Routing Table Entry
```

```
103. class BGPRoutingEntry {
104. public:
105.     BGPRoute::NetworkPrefix prefix;
106.     BGPRoute bestRoute;                    // Currently selected best route
107.     std::vector<BGPRoute> alternateRoutes; // Other feasible routes
108. 
109.     BGPRoutingEntry(BGPRoute::NetworkPrefix pfx) : prefix(pfx) {}
110. 
111.     // Check if we have any route for this prefix
112.     bool HasRoute() const {
113.         return !bestRoute.asPath.empty();
114.     }
115. };
116. 
117. // BGP Routing Information Base (RIB)
118. class BGPRIB {
119. private:
120.     std::map<std::string, BGPRoutingEntry> routingTable;
121. 
122. public:
123.     // Add or update a route
124.     void AddRoute(const BGPRoute& route) {
125.         std::string key = route.prefix.ToString();
126. 
127.         auto it = routingTable.find(key);
128.         if (it == routingTable.end()) {
129.             // New prefix
130.             BGPRoutingEntry entry(route.prefix);
131.             entry.bestRoute = route;
132.             routingTable[key] = entry;
133.         } else {
134.             // Existing prefix - add as alternate
135.             it->second.alternateRoutes.push_back(route);
136.         }
137.     }
138. 
139.     // Get best route for a prefix
140.     BGPRoute* GetBestRoute(const BGPRoute::NetworkPrefix& prefix) {
141.         auto it = routingTable.find(prefix.ToString());
142.         if (it != routingTable.end() && it->second.HasRoute()) {
143.             return &(it->second.bestRoute);
144.         }
145.         return nullptr;
146.     }
147. 
148.     // Get all routes for a prefix
149.     std::vector<BGPRoute> GetAllRoutes(const BGPRoute::NetworkPrefix& prefix) {
150.         std::vector<BGPRoute> routes;
151.         auto it = routingTable.find(prefix.ToString());
152.         if (it != routingTable.end()) {
153.             if (it->second.HasRoute()) {
154.                 routes.push_back(it->second.bestRoute);
155.             }
156.             routes.insert(routes.end(),
157.                           it->second.alternateRoutes.begin(),
158.                           it->second.alternateRoutes.end());
159.         }
160.         return routes;
161.     }
162. 
163.     // Update best route for a prefix
164.     void SetBestRoute(const BGPRoute::NetworkPrefix& prefix, const BGPRoute& route) {
165.         auto it = routingTable.find(prefix.ToString());
166.         if (it != routingTable.end()) {
167.             // Move old best to alternates if it exists
168.             if (it->second.HasRoute()) {
169.                 it->second.alternateRoutes.push_back(it->second.bestRoute);
170.             }
171.             it->second.bestRoute = route;
172. 
```

```
            // Remove from alternates
            auto& alts = it->second.alternateRoutes;
            alts.erase(std::remove_if(alts.begin(), alts.end(),
                [&route](const BGPRoute& r) {
                    return r.asPath == route.asPath &&
                           r.receivedFromAS == route.receivedFromAS;
                }), alts.end());
        }
    }

    // Remove a route (withdrawal)
    void RemoveRoute(const BGPRoute::NetworkPrefix& prefix, uint32_t fromAS) {
        auto it = routingTable.find(prefix.ToString());
        if (it != routingTable.end()) {
            // Check if best route is from this AS
            if (it->second.bestRoute.receivedFromAS == fromAS) {
                it->second.bestRoute = BGPRoute(); // Clear best route
                // Trigger best path recalculation
            }

            // Remove from alternates
            auto& alts = it->second.alternateRoutes;
            alts.erase(std::remove_if(alts.begin(), alts.end(),
                [fromAS](const BGPRoute& r) {
                    return r.receivedFromAS == fromAS;
                }), alts.end());
        }
    }

    // Get all prefixes in RIB
    std::vector<BGPRoute::NetworkPrefix> GetAllPrefixes() const {
        std::vector<BGPRoute::NetworkPrefix> prefixes;
        for (const auto& entry : routingTable) {
            prefixes.push_back(entry.second.prefix);
        }
        return prefixes;
    }
};

} // namespace ns3

```

Path Selection Explanation

When a node receives multiple announcements for the same prefix, it uses the BGP decision process to select the best path:

BGP Decision Process Order:

1. Highest LOCAL_PREF (prefer routes with higher local preference)
2. Shortest AS_PATH (prefer routes through fewer ASes)
3. Lowest ORIGIN (IGP < EGP < INCOMPLETE)
4. Lowest MED (when comparing routes from same neighboring AS)
5. eBGP over iBGP (prefer external over internal BGP)
6. Lowest IGP cost to NEXT_HOP
7. Lowest Router ID (tie-breaker)

The LOCAL_PREF is an inward-looking metric (higher values preferred) set locally for policy, while AS_PATH length determines the "distance" through the Internet, and MED is an outward-looking metric (lower values preferred) suggested by the neighboring AS.

Question 3

// BGP Decision Process Implementation for NS-3

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include <algorithm>

namespace ns3 {

class BGPDecisionProcess {
private:
    uint32_t localAS;
    BGPRIB rib;

public:
    BGPDecisionProcess(uint32_t asNumber) : localAS(asNumber) {}

    /**
     * Main entry point: Process a received route announcement
     * @param route The received BGP route
     * @param fromNeighbor The AS number of the neighbor who sent it
     * @return true if route was accepted and caused a best path change
     */
    bool ProcessRouteAnnouncement(BGPRoute route, uint32_t fromNeighbor) {
        NS_LOG_FUNCTION(this << route.prefix.ToString() << fromNeighbor);

        // Step 1: Input validation and filtering
        route.receivedFromAS = fromNeighbor;
        route.timestamp = Simulator::Now();

        if (!ValidateRoute(route)) {
            NS_LOG_INFO("Route rejected: validation failed");
            return false;
        }

        // Step 2: Apply import policy (can modify attributes)
        ApplyImportPolicy(route, fromNeighbor);

        // Step 3: Add to RIB
        rib.AddRoute(route);

        // Step 4: Run best path selection
        BGPRoute* currentBest = rib.GetBestRoute(route.prefix);
        std::vector<BGPRoute> allRoutes = rib.GetAllRoutes(route.prefix);

        if (allRoutes.empty()) {
            return false;
        }

        // Step 5: Select best path from all candidates
        BGPRoute newBest = SelectBestPath(allRoutes);

        // Step 6: Check if best path changed
```

```
52.         bool bestPathChanged = false;
53.         if (!currentBest || !RoutesEqual(*currentBest, newBest)) {
54.             rib.SetBestRoute(route.prefix, newBest);
55.             bestPathChanged = true;
56.
57.             NS_LOG_INFO("Best path changed for " << route.prefix.ToString()
58.                         << " -> " << newBest.ToString());
59.
60.             // Step 7: Install in forwarding table
61.             InstallRoute(newBest);
62.
63.             // Step 8: Advertise to neighbors (if appropriate)
64.             AdvertiseRoute(newBest);
65.         }
66.
67.         return bestPathChanged;
68.     }
69.
70.     /**
71.      * Process a route withdrawal
72.      */
73.     bool ProcessRouteWithdrawal(BGPRoute::NetworkPrefix prefix, uint32_t fromNeighbor) {
74.         NS_LOG_FUNCTION(this << prefix.ToString() << fromNeighbor);
75.
76.         BGPRoute* currentBest = rib.GetBestRoute(prefix);
77.
78.         // Remove the route from RIB
79.         rib.RemoveRoute(prefix, fromNeighbor);
80.
81.         // If the withdrawn route was the best path, select new best
82.         if (currentBest && currentBest->receivedFromAS == fromNeighbor) {
83.             std::vector<BGPRoute> remainingRoutes = rib.GetAllRoutes(prefix);
84.
85.             if (remainingRoutes.empty()) {
86.                 // No alternate routes - withdraw from forwarding table
87.                 WithdrawRoute(prefix);
88.                 AdvertiseWithdrawal(prefix);
89.                 return true;
90.             } else {
91.                 // Select new best path from remaining routes
92.                 BGPRoute newBest = SelectBestPath(remainingRoutes);
93.                 rib.SetBestRoute(prefix, newBest);
94.                 InstallRoute(newBest);
95.                 AdvertiseRoute(newBest);
96.                 return true;
97.             }
98.         }
99.
100.        return false;
101.    }
102.
103. private:
104.    /**
105.     * Validate incoming route
106.     */
107.    bool ValidateRoute(const BGPRoute& route) {
108.        // Check 1: Reject routes with our AS in the path (loop detection)
109.        if (route.HasLoop(localAS)) {
110.            NS_LOG_INFO("Rejecting route with loop: our AS " << localAS
111.                        << " in path");
112.            return false;
113.        }
114.
115.        // Check 2: Verify AS_PATH is not empty (must have at least origin AS)
116.        if (route.asPath.empty()) {
117.            NS_LOG_INFO("Rejecting route with empty AS_PATH");
118.            return false;
119.        }
120.
121.        // Check 3: Validate prefix is not bogon/martian
```

```
        if (!IsValidPrefix(route.prefix)) {
            NS_LOG_INFO("Rejecting invalid prefix: " << route.prefix.ToString());
            return false;
        }

        return true;
    }

    /**
     * Apply import policy to modify route attributes
     */
    void ApplyImportPolicy(BGPRoute& route, uint32_t fromNeighbor) {
        // Example policies:

        // Policy 1: Set LOCAL_PREF based on neighbor relationship
        if (IsCustomer(fromNeighbor)) {
            route.localPref = 200;  // Prefer customer routes
        } else if (IsPeer(fromNeighbor)) {
            route.localPref = 100;  // Normal preference for peers
        } else if (IsProvider(fromNeighbor)) {
            route.localPref = 50;   // Lower preference for provider routes
        }

        // Policy 2: AS_PATH prepending for traffic engineering
        // (Not done on import, but shown for completeness)

        // Policy 3: MED handling (only compare if from same neighbor AS)
        // MED is already in the route
    }

    /**
     * Core BGP best path selection algorithm
     * Implements the standard BGP decision process
     */
    BGPRoute SelectBestPath(const std::vector<BGPRoute>& routes) {
        if (routes.empty()) {
            return BGPRoute();
        }

        if (routes.size() == 1) {
            return routes[0];
        }

        std::vector<BGPRoute> candidates = routes;

        // STEP 1: Prefer highest LOCAL_PREF
        uint32_t maxLocalPref = 0;
        for (const auto& r : candidates) {
            if (r.localPref > maxLocalPref) {
                maxLocalPref = r.localPref;
            }
        }

        candidates.erase(
            std::remove_if(candidates.begin(), candidates.end(),
                [maxLocalPref](const BGPRoute& r) {
                    return r.localPref < maxLocalPref;
                }),
            candidates.end()
        );

        if (candidates.size() == 1) {
            NS_LOG_DEBUG("Best path selected by LOCAL_PREF: " << maxLocalPref);
            return candidates[0];
        }

        // STEP 2: Prefer shortest AS_PATH
        uint32_t minASPathLen = UINT32_MAX;
        for (const auto& r : candidates) {
            if (r.GetASPathLength() < minASPathLen) {
```

```
192.                 minASPathLen = r.GetASPathLength();
193.             }
194.         }
195.
196.         candidates.erase(
197.             std::remove_if(candidates.begin(), candidates.end(),
198.                 [minASPathLen](const BGPRoute& r) {
199.                     return r.GetASPathLength() > minASPathLen;
200.                 }),
201.             candidates.end()
202.         );
203.
204.         if (candidates.size() == 1) {
205.             NS_LOG_DEBUG("Best path selected by AS_PATH length: " << minASPathLen);
206.             return candidates[0];
207.         }
208.
209.         // STEP 3: Prefer lowest ORIGIN (IGP < EGP < INCOMPLETE)
210.         uint32_t minOrigin = UINT32_MAX;
211.         for (const auto& r : candidates) {
212.             if (r.origin < minOrigin) {
213.                 minOrigin = r.origin;
214.             }
215.         }
216.
217.         candidates.erase(
218.             std::remove_if(candidates.begin(), candidates.end(),
219.                 [minOrigin](const BGPRoute& r) {
220.                     return r.origin > minOrigin;
221.                 }),
222.             candidates.end()
223.         );
224.
225.         if (candidates.size() == 1) {
226.             NS_LOG_DEBUG("Best path selected by ORIGIN: " << minOrigin);
227.             return candidates[0];
228.         }
229.
230.         // STEP 4: Prefer lowest MED (only if from same neighboring AS)
231.         // Group by neighboring AS
232.         std::map<uint32_t, std::vector<BGPRoute>> byNeighborAS;
233.         for (const auto& r : candidates) {
234.             byNeighborAS[r.receivedFromAS].push_back(r);
235.         }
236.
237.         // If all from same AS, compare MED
238.         if (byNeighborAS.size() == 1) {
239.             uint32_t minMED = UINT32_MAX;
240.             for (const auto& r : candidates) {
241.                 if (r.multiExitDisc < minMED) {
242.                     minMED = r.multiExitDisc;
243.                 }
244.             }
245.
246.             candidates.erase(
247.                 std::remove_if(candidates.begin(), candidates.end(),
248.                     [minMED](const BGPRoute& r) {
249.                         return r.multiExitDisc > minMED;
250.                     }),
251.                 candidates.end()
252.             );
253.
254.             if (candidates.size() == 1) {
255.                 NS_LOG_DEBUG("Best path selected by MED: " << minMED);
256.                 return candidates[0];
257.             }
258.         }
259.
260.         // STEP 5: Prefer eBGP over iBGP (simplified - check if neighbor AS != local AS)
261.         std::vector<BGPRoute> ebgpRoutes;
```

```
        for (const auto& r : candidates) {
            if (r.receivedFromAS != localAS) {
                ebgpRoutes.push_back(r);
            }
        }

        if (!ebgpRoutes.empty()) {
            candidates = ebgpRoutes;
        }

        if (candidates.size() == 1) {
            NS_LOG_DEBUG("Best path selected by eBGP preference");
            return candidates[0];
        }

        // STEP 6: Prefer lowest IGP cost to NEXT_HOP
        // (Would require IGP routing table lookup - simplified here)

        // STEP 7: Prefer oldest route (route stability)
        BGPRoute oldest = candidates[0];
        for (const auto& r : candidates) {
            if (r.timestamp < oldest.timestamp) {
                oldest = r;
            }
        }

        if (oldest.timestamp < candidates[0].timestamp) {
            NS_LOG_DEBUG("Best path selected by age (oldest)");
            return oldest;
        }

        // STEP 8: Prefer lowest neighbor Router ID (tie-breaker)
        // Use receivedFromAS as proxy for router ID
        BGPRoute lowestAS = candidates[0];
        for (const auto& r : candidates) {
            if (r.receivedFromAS < lowestAS.receivedFromAS) {
                lowestAS = r;
            }
        }

        NS_LOG_DEBUG("Best path selected by Router ID (AS number): "
                     << lowestAS.receivedFromAS);
        return lowestAS;
    }

    /**
     * Helper functions
     */
    bool IsValidPrefix(const BGPRoute::NetworkPrefix& prefix) {
        // Check for bogon prefixes (simplified)
        std::string addr = prefix.network.Print();

        // Reject 0.0.0.0/x
        if (addr.substr(0, 3) == "0.0") return false;

        // Reject loopback 127.0.0.0/8
        if (addr.substr(0, 3) == "127") return false;

        // Reject link-local, etc.
        // (More checks would be added in real implementation)

        return true;
    }

    bool RoutesEqual(const BGPRoute& r1, const BGPRoute& r2) {
        return r1.asPath == r2.asPath &&
               r1.nextHop == r2.nextHop &&
               r1.receivedFromAS == r2.receivedFromAS;
    }

```

```
    bool IsCustomer(uint32_t asNumber) {
        // Check if AS is a customer (implementation-specific)
        return false; // Placeholder
    }

    bool IsPeer(uint32_t asNumber) {
        // Check if AS is a peer
        return true; // Placeholder
    }

    bool IsProvider(uint32_t asNumber) {
        // Check if AS is a provider
        return false; // Placeholder
    }

    void InstallRoute(const BGPRoute& route) {
        NS_LOG_FUNCTION(this << route.prefix.ToString());

        // Install route in the IP forwarding table
        Ptr<Node> node = /* get local node */;
        Ptr<Ipv4StaticRouting> staticRouting = /* get routing protocol */;

        // Add route to forwarding table
        // staticRouting->AddNetworkRouteTo(
        //     route.prefix.network,
        //     route.prefix.mask,
        //     route.nextHop,
        //     interfaceIndex
        // );
    }

    void WithdrawRoute(const BGPRoute::NetworkPrefix& prefix) {
        NS_LOG_FUNCTION(this << prefix.ToString());
        // Remove route from forwarding table
    }

    void AdvertiseRoute(const BGPRoute& route) {
        NS_LOG_FUNCTION(this << route.prefix.ToString());

        // Prepare route for advertisement (prepend our AS)
        BGPRoute advertisement = route;
        advertisement.PrependAS(localAS);

        // Send to appropriate neighbors based on export policy
        // (Implementation would iterate through BGP neighbors)
    }

    void AdvertiseWithdrawal(const BGPRoute::NetworkPrefix& prefix) {
        NS_LOG_FUNCTION(this << prefix.ToString());
        // Send withdrawal message to neighbors
    }
};

} // namespace ns3

```

The solution provides a comprehensive framework for modeling BGP and inter-AS routing in NS-3:

Key Design Principles:

1. AS Encapsulation: Use custom container classes to logically group nodes and maintain AS boundaries
2. Routing Isolation: Confine IGP protocols to internal interfaces by checking AS membership

3. IXP Modeling: Create explicit peering links with proper IP addressing and interface tagging
4. BGP Attributes: Full data structure supporting AS_PATH, LOCAL_PREF, MED, NEXT_HOP, and ORIGIN
5. Decision Process: Standards-compliant implementation of the BGP best path algorithm

Practical Implementation Steps:

To implement this in your NS-3 simulation:

1. Create AutonomousSystem objects for AS65001 (GlobalISP) and AS65002 (TransitProvider)
2. Use IXPHelper::CreatePeeringPoint() to establish connections at IXP-A and IXP-B
3. Configure OSPF within each AS using IntraASRoutingHelper
4. Implement BGPDecisionProcess on border routers
5. Simulate route announcements between ASes using the BGPRoute data structure

The BGP decision process correctly implements the standard 8-step algorithm, ensuring that routes are selected based on policy (LOCAL_PREF), path length (AS_PATH), and other tie-breaking criteria in the proper order.

C

```
// Complete BGP Decision Process Algorithm (Pseudocode)
class BGPDecisionProcess {
public:
    struct BGPUpdate {
        Ipv4Prefix prefix;
        Ipv4Address nextHop;
        vector<uint32_t> asPath;
        uint32_t localPref;
        uint32_t med;
        uint32_t origin; // 0=IGP, 1=EGP, 2=INCOMPLETE
        Time timestamp;
    };

    // Main decision function
    BGPUpdate ProcessRouteUpdate(BGPUpdate newRoute,
                                 vector<BGPUpdate> existingRoutes,
                                 uint32_t localASN) {

        // Step 0: Basic validation
        if (!ValidateRoute(newRoute, localASN)) {
            return BGPUpdate(); // Invalid route
        }

        // Step 1: Import policies (set LOCAL_PREF based on neighbor)
        ApplyImportPolicy(newRoute);

        // Step 2: Add route to RIB (Routing Information Base)
        vector<BGPUpdate> candidateRoutes = existingRoutes;
        candidateRoutes.push_back(newRoute);

        // Step 3: Run BGP decision process
```

```
        BGPUpdate bestRoute = SelectBestPath(candidateRoutes);

        // Step 4: If new route becomes best, update forwarding
        if (bestRoute.prefix == newRoute.prefix &&
            bestRoute.nextHop == newRoute.nextHop) {
            UpdateForwardingTable(bestRoute);

            // Step 5: Export to other neighbors
            for (auto& neighbor : GetAllBGPNeighbors()) {
                if (neighbor != newRoute.sender) {
                    BGPUpdate exportRoute = ApplyExportPolicy(bestRoute, neighbor);
                    SendUpdateToNeighbor(neighbor, exportRoute);
                }
            }
        }

        return bestRoute;
    }

private:
    bool ValidateRoute(BGPUpdate& route, uint32_t localASN) {
        // Rule 1: Next hop must be reachable
        if (!IsReachable(route.nextHop)) {
            return false;
        }

        // Rule 2: Check for AS_PATH loops
        if (ContainsAS(route.asPath, localASN)) {
            return false; // Loop detected!
        }

        // Rule 3: Maximum AS_PATH length (prevent amplification)
        if (route.asPath.size() > MAX_AS_PATH_LENGTH) {
            return false;
        }

        return true;
    }

    BGPUpdate SelectBestPath(vector<BGPUpdate>& routes) {
        if (routes.empty()) return BGPUpdate();

        BGPUpdate best = routes[0];

        for (const auto& route : routes) {
            if (IsRouteBetter(route, best)) {
                best = route;
            }
        }

        return best;
    }

    bool IsRouteBetter(const BGPUpdate& route1, const BGPUpdate& route2) {
        // BGP Decision Process Steps:

        // 1. Highest LOCAL_PREF
        if (route1.localPref != route2.localPref) {
            return route1.localPref > route2.localPref;
        }

        // 2. Shortest AS_PATH
        if (route1.asPath.size() != route2.asPath.size()) {
            return route1.asPath.size() < route2.asPath.size();
        }

        // 3. Lowest ORIGIN type (IGP < EGP < INCOMPLETE)
        if (route1.origin != route2.origin) {
            return route1.origin < route2.origin;
        }
```

```
102.
103.         // 4. Lowest MED (Multi-Exit Discriminator)
104.         if (route1.med != route2.med) {
105.             return route1.med < route2.med;
106.         }
107.
108.         // 5. Prefer eBGP over iBGP
109.         bool route1_isEBGP = IsEBGP(route1);
110.         bool route2_isEBGP = IsEBGP(route2);
111.         if (route1_isEBGP != route2_isEBGP) {
112.             return route1_isEBGP; // eBGP preferred
113.         }
114.
115.         // 6. Lowest IGP metric to next hop
116.         uint32_t metric1 = GetIGPMetric(route1.nextHop);
117.         uint32_t metric2 = GetIGPMetric(route2.nextHop);
118.         if (metric1 != metric2) {
119.             return metric1 < metric2;
120.         }
121.
122.         // 7. Tie-breakers (simplified)
123.         // In reality: router ID, cluster list, etc.
124.         return route1.timestamp < route2.timestamp; // Older route
125.     }
126.
127.     void UpdateForwardingTable(const BGPUpdate& route) {
128.         // Pseudo-code for updating IP forwarding
129.
130.         // 1. Find interface for next hop
131.         Interface outInterface = FindInterfaceForNextHop(route.nextHop);
132.
133.         // 2. Check if route already exists
134.         ForwardingEntry existingEntry =
135.             GetForwardingEntry(route.prefix.network, route.prefix.mask);
136.
137.         if (existingEntry.exists) {
138.             // 3. Compare with existing route
139.             if (ShouldReplace(existingEntry, route)) {
140.                 // 4. Remove old entry
141.                 RemoveForwardingEntry(existingEntry);
142.
143.                 // 5. Add new entry
144.                 AddForwardingEntry(
145.                     route.prefix.network,
146.                     route.prefix.mask,
147.                     route.nextHop,
148.                     outInterface.index
149.                 );
150.
151.                 cout << "BGP: Updated forwarding for "
152.                      << route.prefix.ToString()
153.                      << " via " << route.nextHop << endl;
154.             }
155.         } else {
156.             // New route, just add it
157.             AddForwardingEntry(
158.                 route.prefix.network,
159.                 route.prefix.mask,
160.                 route.nextHop,
161.                 outInterface.index
162.             );
163.
164.             cout << "BGP: Installed new route for "
165.                  << route.prefix.ToString()
166.                  << " via " << route.nextHop << endl;
167.         }
168.
169.         // 6. Trigger route redistribution to IGP if needed
170.         if (ShouldRedistributeToIGP(route)) {
171.             RedistributeBGPtoIGP(route);
```

```
        }
    }
};

```

Question 4 : Simulating a Route Leak

```
// Route Leak Simulation
class RouteLeakScenario {
public:
    void SimulateRouteLeak() {
        cout << "\n=== Simulating BGP Route Leak ===" << endl;

        // Normal scenario: AS65001 advertises its prefix
        BGPRouteAttributes legitimateRoute(
            Ipv4Address("10.1.0.0"),
            Ipv4Mask("255.255.0.0"),
            100, // LOCAL_PREF
            0    // MED
        );
        legitimateRoute.AddASPath(65001);
        legitimateRoute.SetNextHop("100.64.1.1"); // AS65001_BR1

        // Malicious router in AS65002 (Rogue_AS65002)
        // Incorrectly re-advertises AS65001's route back to AS65001
        BGPRouteAttributes leakedRoute(
            Ipv4Address("10.1.0.0"),
            Ipv4Mask("255.255.0.0"),
            150, // Higher LOCAL_PREF to attract traffic
            0    // MED
        );

        // Malicious AS_PATH: Pretends the route came from somewhere else
        // AS_PATH: 65002 65001 (implies: route originated at AS65001, passed through AS65002)
        // But actually: route originated at AS65001, should not go through AS65002 back to AS65001
        leakedRoute.AddASPath(65001);
        leakedRoute.AddASPath(65002); // This is the leak!
        leakedRoute.SetNextHop("100.64.1.2"); // Rogue router in AS65002

        cout << "Legitimate route AS_PATH: [65001]" << endl;
        cout << "Leaked route AS_PATH: [65002 65001]" << endl;

        // How nodes in AS65001 react:
        vector<BGPRouteAttributes> candidateRoutes;
        candidateRoutes.push_back(legitimateRoute);
        candidateRoutes.push_back(leakedRoute);

        // Run BGP decision in AS65001 router
        BGPRouteAttributes chosenRoute =
            BGPDecisionProcess::SelectBestRoute(candidateRoutes);

        cout << "\nAS65001 Router Decision:" << endl;
        cout << "Option 1: " << legitimateRoute.ToString() << endl;
        cout << "Option 2: " << leakedRoute.ToString() << endl;

        if (chosenRoute.GetASPath().size() > 0 &&
            chosenRoute.GetASPath()[0] == 65002) {
            cout << "RESULT: CHOOSES LEAKED ROUTE!" << endl;
            cout << "WHY: Higher LOCAL_PREF (150 vs 100)" << endl;
            cout << "CONSEQUENCE: Traffic from AS65001 to 10.1.0.0/16" << endl;
            cout << "  will be sent to AS65002 instead of staying internal!" << endl;
            cout << "  This creates a routing loop or traffic interception." << endl;
        } else {
            cout << "RESULT: CHOOSES LEGITIMATE ROUTE" << endl;
```

```
            cout << "WHY: Loop prevention or other attributes better" << endl;
        }

        // Defensive mechanisms simulation
        SimulateDefenses(legitimateRoute, leakedRoute);
    }

private:
    void SimulateDefenses(const BGPRouteAttributes& legitimate,
                          const BGPRouteAttributes& leaked) {
        cout << "\n=== Defensive Mechanisms ===" << endl;

        // Defense 1: AS_PATH loop detection
        cout << "1. AS_PATH Loop Detection:" << endl;
        if (leaked.ContainsAS(65001)) {
            cout << "   LEAK DETECTED: AS_PATH contains own AS (65001)" << endl;
            cout << "   Action: Reject leaked route" << endl;
        }

        // Defense 2: Maximum Prefix Length
        cout << "\n2. Maximum Prefix Advertisement:" << endl;
        cout << "   AS65001 could configure: 'neighbor AS65002 maximum-prefix 10'" << endl;
        cout << "   If AS65002 advertises >10 prefixes, shutdown session" << endl;

        // Defense 3: Route Filtering with ROAs (RPKI)
        cout << "\n3. RPKI Validation:" << endl;
        cout << "   Check if AS65002 is authorized to advertise 10.1.0.0/16" << endl;
        cout << "   ROA would say: 10.1.0.0/16, origin AS65001, max length /16" << endl;
        cout << "   Leaked route: origin AS65002 -> INVALID" << endl;

        // Defense 4: BGP Communities
        cout << "\n4. BGP Communities:" << endl;
        cout << "   AS65001 tags internal routes with: 65001:100 (no-export)" << endl;
        cout << "   AS65002 should not re-advertise these routes" << endl;
        cout << "   But malicious router ignores communities" << endl;
    }
};

// Example of malicious advertisement
void CreateMaliciousRouteAdvertisement() {
    cout << "\n=== Malicious Router in AS65002 ===" << endl;
    cout << "Normal behavior:" << endl;
    cout << "  Receive: 10.1.0.0/16 from AS65001 (AS_PATH: [65001])" << endl;
    cout << "  Export to: Other customers (AS_PATH: [65002 65001])" << endl;
    cout << "  Should NOT export to: AS65001 (would create [65002 65001])" << endl;

    cout << "\nMalicious behavior:" << endl;
    cout << "  Receive: 10.1.0.0/16 from AS65001" << endl;
    cout << "  Modify: Set LOCAL_PREF = 150" << endl;
    cout << "  Re-advertise to: AS65001 (violating policy)" << endl;
    cout << "  Resulting AS_PATH: [65002 65001]" << endl;

    cout << "\nImpact:" << endl;
    cout << "1. AS65001 sees two paths to its own network:" << endl;
    cout << "   Path A: Direct (AS_PATH: [], cost 0)" << endl;
    cout << "   Path B: Via AS65002 (AS_PATH: [65002], LOCAL_PREF 150)" << endl;
    cout << "2. BGP selects Path B due to higher LOCAL_PREF" << endl;
    cout << "3. Traffic loops: AS65001 -> AS65002 -> AS65001" << endl;
    cout << "4. AS65002 can intercept, inspect, or drop traffic" << endl;
}


```

Question 5

```
1. // Discussion of NS-3 BGP Model vs Reality
```

```
 2. class BGPModelLimitations {
 3. public:
 4.     struct RealBGPFeature {
 5.         string name;
 6.         string description;
 7.         string whyHardToModel;
 8.     };
 9.
10.     void CompareWithReality() {
11.         cout << "\n=== NS-3 BGP Model vs Real BGP ===" << endl;
12.
13.         cout << "\nSIGNIFICANT SIMPLIFICATIONS:" << endl;
14.         cout << "1. No TCP Session Management:" << endl;
15.         cout << "   Real BGP: Runs over TCP (port 179), with session establishment," <<
endl;
16.         cout << "             keepalives, graceful restart" << endl;
17.         cout << "   NS-3: Direct message passing, no transport layer simulation" << endl;
18.
19.         cout << "\n2. Simplified State Machine:" << endl;
20.         cout << "   Real BGP: 6-state FSM (Idle, Connect, Active, OpenSent," << endl;
21.         cout << "             OpenConfirm, Established)" << endl;
22.         cout << "   NS-3: Simple 'connected/disconnected' state" << endl;
23.
24.         cout << "\n3. No Route Dampening:" << endl;
25.         cout << "   Real BGP: Penalizes flapping routes, suppresses unstable prefixes" <<
endl;
26.         cout << "   NS-3: All routes treated equally, no stability tracking" << endl;
27.
28.         cout << "\n4. No Real-world Policies:" << endl;
29.         cout << "   Real BGP: Complex regular expressions for AS_PATH filtering," << endl;
30.         cout << "             prefix-lists, route-maps with set/delete community" << endl;
31.         cout << "   NS-3: Simple if-else statements for policy" << endl;
32.
33.         cout << "\n5. Scale Limitations:" << endl;
34.         cout << "   Real Internet: ~900,000 BGP routes in global table" << endl;
35.         cout << "   NS-3: Typically <1000 routes due to memory/performance" << endl;
36.
37.         // Three Critical Features Extremely Difficult to Model
38.         vector<RealBGPFeature> difficultFeatures = {
39.             {
40.                 "BGP Route Reflectors",
41.                 "Hierarchical route distribution within an AS to reduce iBGP mesh",
42.                 "Requires modeling complex cluster relationships, client/non-client roles, "
43.                 "originator ID, cluster list attributes. NS-3's flat node hierarchy "
44.                 "doesn't naturally support the administrative relationships needed."
45.             },
46.             {
47.                 "BGP Large Communities & Extended Communities",
48.                 "Complex policy tagging with 12-byte communities for traffic engineering",
49.                 "NS-3's packet headers are simplified. Modeling the full 12-byte community "
50.                 "attribute with transitive/non-transitive flags and complex matching "
51.                 "semantics would require extensive modifications to packet structure "
52.                 "and attribute propagation logic."
53.             },
54.             {
55.                 "gRPC/Protobuf-based BGP (OpenConfig)",
56.                 "Modern BGP monitoring and configuration via gRPC with YANG models",
57.                 "NS-3 focuses on network layer simulation. Implementing full gRPC stack "
58.                 "with Protobuf serialization, YANG model validation, and NETCONF/RESTCONF "
59.                 "interfaces would be equivalent to building a real network operating "
60.                 "system, not a simulation."
61.             }
62.         };
63.
64.         cout << "\nTHREE CRITICALLY DIFFICULT FEATURES TO MODEL:" << endl;
65.         for (size_t i = 0; i < difficultFeatures.size(); i++) {
66.             cout << "\n" << (i+1) << ". " << difficultFeatures[i].name << ":" << endl;
67.             cout << "   What: " << difficultFeatures[i].description << endl;
68.             cout << "   Why hard: " << difficultFeatures[i].whyHardToModel << endl;
69.         }
```

```
70.
71.         // Assessment of NS-3 for Inter-AS Research
72.         cout << "\n=== IS NS-3 SUITABLE FOR INTER-AS ROUTING RESEARCH? ===" << endl;
73.
74.         cout << "\nYES, for certain types of research:" << endl;
75.         cout << "1. Protocol Behavior Studies:" << endl;
76.         cout << "   - BGP convergence after link failures" << endl;
77.         cout << "   - Route propagation dynamics" << endl;
78.         cout << "   - Impact of BGP timers (MRAI)" << endl;
79.
80.         cout << "\n2. Security Research:" << endl;
81.         cout << "   - Route leak/prefix hijack impact analysis" << endl;
82.         cout << "   - RPKI/ROA validation effectiveness" << endl;
83.         cout << "   - BGPsec simulation (cryptographic validation)" << endl;
84.
85.         cout << "\n3. Educational Purposes:" << endl;
86.         cout << "   - Teaching BGP fundamentals" << endl;
87.         cout << "   - Visualizing AS_PATH selection" << endl;
88.         cout << "   - Understanding iBGP vs eBGP" << endl;
89.
90.         cout << "\nNO, for production or highly realistic studies:" << endl;
91.         cout << "1. Performance Testing:" << endl;
92.         cout << "   - Cannot simulate full Internet routing table" << endl;
93.         cout << "   - No hardware acceleration or TCAM simulation" << endl;
94.
95.         cout << "\n2. Interoperability Testing:" << endl;
96.         cout << "   - Cannot test against real router implementations" << endl;
97.         cout << "   - No vendor-specific feature simulation" << endl;
98.
99.         cout << "\n3. Microsecond-level Timing:" << endl;
100.        cout << "   - NS-3 abstracts lower-layer timing" << endl;
101.        cout << "   - No accurate simulation of BGP UPDATE packet processing delays" <<
endl;
102.
103.        cout << "\nRECOMMENDED APPROACH:" << endl;
104.        cout << "Hybrid methodology:" << endl;
105.        cout << "1. Use NS-3 for large-scale topology and protocol behavior" << endl;
106.        cout << "2. Use MiniNet or container-based emulation for real BGP daemons" << endl;
107.        cout << "3. Use historical BGP data (RouteViews, RIPE RIS) for validation" << endl;
108.        cout << "4. For production: Test on physical/virtual routers (FRR, Bird)" << endl;
109.
110.        cout << "\nCONCLUSION:" << endl;
111.        cout << "NS-3 is EXCELLENT for:" << endl;
112.        cout << "- Understanding BGP algorithmic behavior" << endl;
113.        cout << "- Educational visualization" << endl;
114.        cout << "- Controlled experiments with known variables" << endl;
115.        cout << "- Research on new BGP extensions in idealized conditions" << endl;
116.
117.        cout << "\nNS-3 is INADEQUATE for:" << endl;
118.        cout << "- Testing real-world BGP implementations" << endl;
119.        cout << "- Performance benchmarking" << endl;
120.        cout << "- Interoperability validation" << endl;
121.        cout << "- Production network design validation" << endl;
122.
123.        cout << "\nUse NS-3 as a complement to, not replacement for, real BGP testing." <<
endl;
124.    }
125. };
126.
127. // Hybrid approach suggestion
128. void HybridResearchMethodology() {
129.     cout << "\n=== Recommended Research Pipeline ===" << endl;
130.     cout << "Phase 1: NS-3 Simulation" << endl;
131.     cout << "  - Model new BGP extension algorithmically" << endl;
132.     cout << "  - Test in controlled, repeatable environment" << endl;
133.     cout << "  - Validate basic correctness" << endl;
134.
135.     cout << "\nPhase 2: Emulation (MiniNet, GNS3)" << endl;
136.     cout << "  - Implement in real BGP daemon (FRRouting)" << endl;
137.     cout << "  - Test interoperability with existing implementations" << endl;
```

```
138.     cout << "  - Measure real performance metrics" << endl;
139.
140.     cout << "\nPhase 3: Testbed Deployment" << endl;
141.     cout << "  - Deploy on research testbed (GENI, FABRIC)" << endl;
142.     cout << "  - Test at scale with realistic traffic" << endl;
143.     cout << "  - Collect operational experience" << endl;
144.
145.     cout << "\nPhase 4: Real-world Deployment" << endl;
146.     cout << "  - Deploy in controlled production environment" << endl;
147.     cout << "  - Monitor with real BGP looking glasses" << endl;
148.     cout << "  - Collect BGP update data for analysis" << endl;
149. }
150.
```

Summary of Key Points:

1. Route Leak Behavior: Malicious router creates AS_PATH [65002 65001] which should be rejected by AS65001 due to loop detection (contains own AS). However, if LOCAL_PREF is manipulated higher, it might be temporarily preferred until loop detection kicks in.
2. NS-3 Limitations: The model simplifies TCP, state machines, route dampening, and lacks complex policies. Critical features like route reflectors, communities, and gRPC are extremely difficult to model accurately.
3. Suitability Assessment: NS-3 is suitable for algorithmic studies, educational purposes, and controlled experiments but inadequate for performance testing, interoperability validation, or production network design.

The most valuable use of NS-3 for BGP research is in understanding fundamental behaviors and testing new algorithms in a controlled environment before moving to more realistic emulation or testbed deployment.