Name: BAGNAWE GNOGA Emmanuel Dylan
ID Number: ICTU20241556
E-Mail: bagnawe.gnoga@ictuniversity.edu.cm
GitHub Account: https://github.com/daa92
Level: 3 – Semester I
Course – ID: Wide Area Networks – ISN 3132
Instructor: Engr. Daniel MOUNE
Submission: 2025-12-16

CONTINUOUS ASSESSMENT

NOTE: To better understand the provided code, I did rewrite it by myself with my own variable and object names.

`Exercise 1`: *Multi-Site WAN Extension with Redundant Paths*

1. **Specific modifications needed to extend the provided NS-3 code to create a triangular topology:**

- Adding a new network (network3) which will link the client to the server:

```
//==============network3============= client <---> server

NodeContainer network3 (client, server); //yes, even networks are objects
                                         //we directly assign nodes to each network
        //adding NIC to each node
NetDeviceContainer network3devices = p2p.Install (network3);

        //Assigning IP (version 4)  addresses to the network
Ipv4AddressHelper network3address;
network3address.SetBase ("10.1.3.0", "255.255.255.0");
Ipv4InterfaceContainer network3interface = network3address.Assign (network3devices);
// it will be like this:
// 0 client 2 <------> 2 server 0
// index 0 is for the loopback (127.0.0.0)
// index 2 (client) = 10.1.3.2 (server'ip)  and then index 2 (server) = 10.1.3.1 (client's ip)


//==============End_network3=============
```

2. **Static Routing Table Analysis**

- Client: we tell the client where to route the packet on the direct path, and after on the backup path

```
        // ***Static routing configuration***
        //enabling IP forwarding on the router
        Ptr <Ipv4> ipv4router = router-> GetObject <Ipv4> ();
        ipv4router->SetAttribute ("IpForward", BooleanValue (true));

        // Configure routing on the client
        // for now, client needs to know that to reach 10.1.2.0/24, it should go through 10.1.1.2 (router's inte
rface)
        Ptr <Ipv4StaticRouting> staticroutingclient = staticroutinghelper.GetStaticRouting (client -> GetObject
<Ipv4> ());


        // direct path client < ----- > server
        staticroutingclient->AddHostRouteTo (
                        Ipv4Address ("10.1.3.2"), //ip of the target network
                        2, // the interface attached to ( 0 client 2 -> 2 server 0 )
                        1 // high priority
        );
```

```
        //backup path client < ------- > router < ------- > server
        staticroutingclient->AddHostRouteTo (
                        Ipv4Address ("10.1.3.2"), //ip of the target node
                        Ipv4Address ("10.1.1.2"), // the hop (intermediary)
                        1, // interface
                        10 // metric (low-priority)
        );

        // Route to reach server's other interface (10.1.2.2) via router
        staticroutingclient->AddNetworkRouteTo(
                Ipv4Address("10.1.2.0"), //target network
                Ipv4Mask("255.255.255.0"), // subnet
                Ipv4Address("10.1.1.2"), //hop
                1 // interface
        );
```

- Server: same logic for the server

```
        // Route to reach client's other interface (10.1.1.1) via router
        staticroutingserver->AddNetworkRouteTo(
                Ipv4Address("10.1.1.0"),
                Ipv4Mask("255.255.255.0"),
                Ipv4Address("10.1.2.1"),
                1
        );

                Ipv4Address("10.1.3.1"),   // Destination ip
                Ipv4Address("10.1.2.1"),   // the intermediary (router) (its ip address attached to the
network2)
                1,                          // the interface attached to ( 0 client 1 -> 1 router 0  ->
1 client )
                10S
            );
```

- Router: noticing the router which interface is attached which network and enabling routing

```
    // ============ ROUTER ROUTING TABLE ============
    Ptr<Ipv4StaticRouting> staticroutingrouter = staticroutinghelper.GetStaticRouting(router->GetObject<Ipv4
>());

    // Route to client's networks
    staticroutingrouter->AddNetworkRouteTo(
            Ipv4Address("10.1.1.0"),
            Ipv4Mask("255.255.255.0"),
            1
    );

    staticroutingrouter->AddNetworkRouteTo(
            Ipv4Address("10.1.3.0"),
            Ipv4Mask("255.255.255.0"),
            Ipv4Address("10.1.1.1"),       // Via client
            1
    );
```

```
        // Route to server's networks
        staticroutingrouter->AddNetworkRouteTo(
                Ipv4Address("10.1.2.0"),
                Ipv4Mask("255.255.255.0"),
                2
        );

        staticroutingrouter->AddNetworkRouteTo(
                Ipv4Address("10.1.3.0"),
                Ipv4Mask("255.255.255.0"),
                Ipv4Address("10.1.2.2"),        // Via server
                2
        );
```

3. **Path Failure Simulation**

Here we just turn off the NIC link (Point-To-Point) at +4 seconds and then turn it off at +7 to simulate the unavailability.

```
// ============ SIMULATE LINK FAILURE ============
// Get Ipv4 objects
Ptr<Ipv4> ipv4Client = client->GetObject<Ipv4>();
Ptr<Ipv4> ipv4Server = server->GetObject<Ipv4>();

// Get interface indices for the direct link devices
uint32_t clientIfIndex = ipv4Client->GetInterfaceForDevice(network3devices.Get(0));
uint32_t serverIfIndex = ipv4Server->GetInterfaceForDevice(network3devices.Get(1));

// Set direct link DOWN at 4 seconds (forces backup path)
Simulator::Schedule(Seconds(4.0), &Ipv4::SetDown, ipv4Client, clientIfIndex);
Simulator::Schedule(Seconds(4.0), &Ipv4::SetDown, ipv4Server, serverIfIndex);

// Set direct link UP at 7 seconds (restore direct path)
Simulator::Schedule(Seconds(7.0), &Ipv4::SetUp, ipv4Client, clientIfIndex);
Simulator::Schedule(Seconds(7.0), &Ipv4::SetUp, ipv4Server, serverIfIndex);
```

The method used to check if the backup works is just the outputs, the routing table and the Netanim animation (check the video **exercise1.webm**).

Using of **FlowMonitor** to compare the latency between the direct and the backup paths

```
#include "ns3/flow-monitor-module.h" //for comparing the latency between the direct and the backup path
 /* // ============ FLOWMONITOR ANALYSIS ============
 monitor->CheckForLostPackets();

 Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
 FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats();
```

Check the **router-static-routing.routes** for the routing table.

4. **Scalability Analysis**

Static routes would be needed if the company expanded to 10 sites in a full mesh: **10 x 9 = 90 routes**

More scalable approach using a dynamic routing protocol: **OSPF** (Open Shortest Path First) or **BGP** (Border Gateway Protocol)

For realistic **OSPF** protocol behavior, Using the **`ns3::QuaggaHelper`** to integrate the real-world Quagga routing suite (which includes OSPFv2/v3 daemons) into your NS-3 nodes is the ideal. How to step it up:

* Import its library

```
#include "ns3/dce-module.h"
#include "ns3/quagga-helper.h"
```

* Install nodes and stacks

```
// Create nodes
NodeContainer routers;
routers.Create(N);

// Install Internet Stack
InternetStackHelper stack;
stack.Install(routers);

// Install DCE Manager
DceManagerHelper dce;
dce.Install(routers);
```

* Configure Quagga

```
router ospf
ospf router-id 1.1.1.1
network 10.1.1.0/24 area 0  // Network attached to R1, placed in Area 0
network 10.1.2.0/24 area 0  // Router-to-router link, placed in Area 0
```

* Simulation

5. **Business Continuity Justification**

- **Improved Reliability via Immediate Redundancy (Failover):** By having two distinct, static routes to the same destination (one primary, one through a different peer), any single link failure (e.g., the direct Client ↔ Server link) is instantly mitigated. The packet is simply routed through the alternate, working path (ex: Client → Router → Server) without the need for a dynamic protocol to converge. This achieves near **zero-downtime failover** for link failures.
- **Load Balancing Potential through Equal-Cost Paths:** Static routing can be configured with **multiple equal-cost next hops** for the same destination network (often done by simply adding two identical routes to the routing table). This allows the operating system's kernel to perform simple **per-packet or per-flow load balancing** across the parallel, redundant links (e.g., traffic between the Client and Server is spread across the direct link and the link through the Router).
- **Simplified Troubleshooting through Deterministic Paths:** Unlike complex dynamic protocols that can frequently change paths, static routing forces traffic along an **explicitly defined and constant path**. This **determinism** is a major benefit for troubleshooting, as you always know *exactly* which link (or series of links) a packet should traverse for a specific destination, making it easier to isolate problems like latency, packet drops, or policy issues.

`Exercise 2`: *QoS implementation for mixed traffic*

1. **Traffic Differentiation**

Class 1: VoIP. Here we just used **G.711 VoIP codec** for **applications** module

// for enable the outputs

```
// for the VoIP
LogComponentEnable("UdpClient", LOG_LEVEL_INFO);
LogComponentEnable("UdpServer", LOG_LEVEL_INFO);
```

//configuration itself

```
//=======================VoIP configuration of sending udp packets
/*Here we have no reply from the server
 * 160 bytes (G.711 VoIP codec)
 * 20ms between packets (6.5s → 6.52s → 6.54s)
 * Consistent delay: 4.608ms on all packets (good network quality)_this info comes from the output
 * Delay: +4.608e+06ns*/
UdpServerHelper voipserver (5060); // port = 5060
ApplicationContainer serverapps = voipserver.Install (server);
serverapps.Start (Seconds (6));
serverapps.Stop (Seconds (10));

UdpClientHelper voipclient (network2interface.GetAddress (1), 5060);
voipclient.SetAttribute ("MaxPackets", UintegerValue (3));
voipclient.SetAttribute ("Interval", TimeValue (Seconds (0.02))); //20ms
voipclient.SetAttribute ("PacketSize", UintegerValue (160)); //160b

ApplicationContainer clientapps = voipclient.Install (client);
clientapps.Start (Seconds (6.5)); // begins at +6.5s
clientapps.Stop (Seconds (10));

NS_LOG_INFO("VoIP applications configured");
```

Class 2: FTP

```
// ============ FTP SERVER (PacketSink - receives data) ============
uint16_t ftpPort = 21; // 21 is the port number of commands of FTP protocol
PacketSinkHelper sinkHelper("ns3::TcpSocketFactory",
                            InetSocketAddress(Ipv4Address::GetAny(), ftpPort));
ApplicationContainer sinkApp = sinkHelper.Install(server);
sinkApp.Start(Seconds(2.0));
sinkApp.Stop(Seconds(10.0));

// ============ FTP CLIENT - BURST 1 (BulkSend) ============
BulkSendHelper ftpBurst1("ns3::TcpSocketFactory",
                         InetSocketAddress(network2interface.GetAddress(1), ftpPort));

// Simulate transferring a 5MB file in first burst
ftpBurst1.SetAttribute("MaxBytes", UintegerValue(5 * 1024 * 1024)); // 1.5kB
ftpBurst1.SetAttribute("SendSize", UintegerValue(1500));            // 1.5KB packets

ApplicationContainer burst1App = ftpBurst1.Install(client);
burst1App.Start(Seconds(2.5));   // First burst at 2.5s
burst1App.Stop(Seconds(10.0));

NS_LOG_INFO("FTP Burst 1: 5MB file transfer (1-5s)");
```

DSCP (Differentiated Services Code Point): **DSCP** is a 6-bit field in the IP header used for **Quality of Service (QoS)** to prioritize different types of network traffic.

Common DSCP Values:

DSCP Value	Name	Priority	Typical Use
0	Best Effort (BE)	Lowest	Regular internet traffic, FTP
8	CS1	Low	Background traffic
10	AF11	Medium-Low	Bulk data
18	AF21	Medium	Standard applications
26	AF31	Medium-High	Streaming video
34	AF41	High	Video conferencing
46	EF (Expedited Forwarding)	Highest	VoIP, real-time traffic
48	CS6	Critical	Network control

DSCP tag for **UDP** echo: here the value is 0, meaning that the priority is **very low**

```
// SET DSCP FOR UDP Echo - BE (0) = 0x00
Simulator::Schedule(Seconds(2.001), [&clientApps]() {
    Ptr<UdpEchoClient> app = DynamicCast<UdpEchoClient>(clientApps.Get(0));
    if (app) {
        Ptr<Socket> socket = app->GetSocket();
        if (socket) {
            socket->SetIpTos(0x00);  // DSCP 0 (BE)
            NS_LOG_INFO("UDP Echo: DSCP BE (0) = 0x00 - LOWEST PRIORITY (Best Effort)");
        }
    } else {
        NS_LOG_ERROR("Failed to set DSCP for UDP Echo");
    }
});
```

DSCP tag for **VoIP**: here the value is 46, meaning that the priority is **very high**

```
 // SET DSCP FOR VoIP - EF (46) = 0xB8
 Simulator::Schedule(Seconds(3.001), [&clientapps]() {
     Ptr<UdpClient> app = DynamicCast<UdpClient>(clientapps.Get(0));
     if (app) {
         Ptr<Socket> socket = app->GetSocket();
         if (socket) {
             socket->SetIpTos(0xB8);  // DSCP 46 (EF)
             NS_LOG_INFO("VoIP: DSCP EF (46) = 0xB8 - HIGHEST PRIORITY");
         }
     } else {
         NS_LOG_ERROR("Failed to set DSCP for VoIP");
     }
 });
```

DSCP tag for **FTP**: here the value is 24, meaning that the priority is **medium**.

```
// SET DSCP FOR FTP - CS3 (24) = 0x60
Simulator::Schedule(Seconds(2.001), [&burst1App]() {
    Ptr<BulkSendApplication> app = DynamicCast<BulkSendApplication>(burst1App.Get(0));
    if (app) {
        Ptr<Socket> socket = app->GetSocket();
        if (socket) {
            socket->SetIpTos(0x60);  // DSCP 24 (CS3)
            NS_LOG_INFO("FTP: DSCP CS3 (24) = 0x60 - MEDIUM PRIORITY");
        }
    } else {
        NS_LOG_ERROR("Failed to set DSCP for FTP");
    }
});
```

2. **Queue Management Implementation**

- Queue discipline to use: '**PrioQueueDisc**' (Priority Queue Discipline) because it supports multiple priority bands (queues), routes packets to different queues based on DSCP values and higher priority are always served first.

- Configuration parameters: 100 packets max for each band, lowest priority for the UDP Echo, medium for the FTP and then high priority for the VoIP.

- Mapping the traffic classes to different queue: we map different classes by their value of priority (0, 24 and 46).

```
TrafficControlHelper tchPrio;

// PrioQueueDisc uses the IP TOS field to determine priority
// We just need to install it - it will automatically use DSCP values
tchPrio.SetRootQueueDisc("ns3::PrioQueueDisc");

// Install on router's interface to server (network2device.Get(0))
QueueDiscContainer qdiscs = tchPrio.Install(network2device.Get(0));

// Get the PrioQueueDisc and manually add child queues
Ptr<QueueDisc> qdisc = qdiscs.Get(0);
Ptr<PrioQueueDisc> prioQdisc = DynamicCast<PrioQueueDisc>(qdisc);

if (prioQdisc)
{
    // Create 3 child FIFO queues for 3 priority bands
    ObjectFactory factory;
    factory.SetTypeId("ns3::FifoQueueDisc");
    factory.Set("MaxSize", StringValue("100p"));

    for (uint32_t i = 0; i < 3; i++)
    {
        Ptr<QueueDisc> childQdisc = factory.Create<QueueDisc>();
        Ptr<QueueDiscClass> qclass = CreateObject<QueueDiscClass>();
        qclass->SetQueueDisc(childQdisc);
        prioQdisc->AddQueueDiscClass(qclass);
    }

    NS_LOG_INFO("PrioQueueDisc with 3 bands installed successfully");
```

3. Performance measurement:
- NS-3 tool used: FlowMonitor
- We are going to compare latency, jitter, packet loss.
- The results should be displays on a table during the execution of the program.

Check the code for the modifications.

4. Simulation of congestion
- For this, we will set the delay to 10ms
- Set smaller queue to 20 packets

5. Real-World Implementation Gap

Feature	Real-World Capability	NS-3 Limitation	Approximation Quality
Hardware Shaping	Nanosecond precision, parallel processing	Discrete events, sequential	**60%** - Timing inaccurate but behavior similar
DPI	Payload inspection, ML classification	No payload, 5-tuple only	**40%** - Can use heuristics but missing key capability
Adaptive QoS	Real-time ML, control plane	Static config, no OS model	**70%** - Can simulate decisions but not real ML

`Exercise 3`: *WAN security integration and attack simulations*

1. The is no NS-3 module which allows us to simulate IPSec VPN. Besides this we actually made up two classes (**VpnEncryptionDelayApp** and **IPSecNetDevice**) which simulate it. We added encryption, decryption and **HMAC** (Hash-based Message Authentication Code) delays, increased the size of each packet, due to **ESP** (Encapsulating Security Payload) in tunnel mode by: New outer IP header: 20 bytes, ESP header: 8 bytes, Initialization Vector (IV): 16 bytes, ESP trailer: 2 bytes, Authentication data (**ICV**): 12 bytes.

Expected Performance Overhead

Real-World IPSec Performance Impact

Metric	Without VPN	With IPSec VPN	Overhead
Latency	2 ms	2.05-2.15 ms	+50-150 µs
Throughput	5 Mbps	4.75-4.85 Mbps	-3-5%
CPU Usage	Baseline	+10-20%	Varies by CPU
Packet Size	1500 bytes	1558 bytes	+58 bytes
Effective MTU	1500 bytes	1442 bytes	-58 bytes
Jitter	Low	Slightly higher	+5-10%

2. **Eavesdropping Attack Simulation**: The eavesdropper simulates a **Man-in-the-Middle (MITM) attacker** who is passively monitoring network traffic between the client and router. It uses NS-3's **promiscuous mode** to capture all packets on the link, regardless of whether they're addressed to the eavesdropper. Information which might be extracted are sequence numbers, timestamps, Echo request/reply patterns, source/destination IP addresses, port numbers, Packet sizes and timing, communication frequency, data volume, connection endpoints. To prevent this attack, the VPN will encrypt data so that the hacker can only see cipher text.

3. **DDoS**: here, we create 5 other nodes (attackers), which send multiple packets to the router in order to overload it. The attack type is **UDP** which can be totally changeable (via arguments before executing the code). During the simulation, instead of receiving 10 packets, the router received 7654, and didn't drop anything.

4. **Defense Mechanisms**:
   a) Rate limiting on the router interfaces

This is about install a queue-discipline (token-bucket or leaky-bucket style) on the router's two physical devices so that the router will only forward at a configured maximum rate and will buffer/drop excess packets. Use the Traffic Control module (**TrafficControlHelper + a TBF-like QueueDisc**) and attach it to the router's NetDevices. When incoming traffic exceeds the configured token rate, packets will queue and eventually be dropped when the queue fills; attackers' high-rate traffic will be throttled at the router egress.

Limitations: The disc limits bit-rate leaving the device; it does not distinguish good Vs bad flows, NS-3 queue-disc models are idealized and Making sure to pick a limit that reflects the real link capacity Vs overhead (you already modified DataRate for VPN).

b) Access Control Lists (ACLs) to block malicious traffic

When the router NetDevice receives a packet, the callback inspects the IPv4 header; if it returns false the packet is treated as dropped/blocked and will not continue to be processed/forwarded by the stack (this behavior depends on NetDevice semantics in this NS-3 version; **SetPromiscReceiveCallback** can be used to observe but not necessarily block; prefer **SetReceiveCallback** if you need to block).

Limitations: The NetDevice receive callback API differs between device types and NS-3 versions, ACL rules based purely on IP addresses are ineffective vs. IP spoofing (SYN floods often spoof sources), Order/placement matters, ACLs implemented this way are instantaneous and idealized; they don't model processing cost of applying complex rules.

c) Any-cast or load balancing to distribute attack traffic

This is just about using multiple routers and forward the packet to the nearest one.

Limitations: NS-3's default IPv4 routing doesn't do true ECMP (Equal-Cost Multi-Path) load balancing (it picks first route), you'd need to implement custom routing logic or per-packet hash-based selection, doesn't simulate BGP any cast announcements.

5. **Security vs. Performance Trade-off Analysis**:

- How much throughput reduction would you expect when enabling IPSec: the expected throughput impact is 3-8% of reduction. The components contributing to reduction are the protocol overhead (bandwidth loss) and Processing Overhead.
- What latency increase would DDoS protection mechanisms introduce: the components contributing to latency are Firewall/ACL Processing, Rate Limiting and Deep Packet Inspection (DPI)

- Proposition of a balanced security posture for the company's WAN based on your simulation insights, considering both protection levels and performance impact: enable full IPSec and DDoS protection everywhere except:
  - ✗ Verified internal trusted networks (Layer 2 switches)
  - ✗ Development/testing environments
  - ✗ Already-encrypted application traffic (HTTPS/TLS)

  Rationale from Simulation:

- ✗ 8% throughput reduction is **acceptable** for 100% confidentiality
- ✗ 1-2ms latency increase is **imperceptible** to users
- ✗ 85-90% attack mitigation is **excellent** protection
- ✗ Cost of NOT implementing >> Cost of performance loss

The simulation proves: Modern encryption and DDoS protection are **efficient enough** that security should be the DEFAULT, not the exception.

`Exercise 4`: *Multi-hop WAN architecture with fault tolerance*

a) **Topology Analysis and Extension**: the key implementation here are the multi-path routing (DR-B has two routes to Branch-C), link characteristics (different bandwidth/latency simulate real WAN conditions), resilience testing (we can simulate link failures by disabling Network2 to test failover to Network3), PCAP analysis (three separate network captures allow detailed traffic analysis per segment) and NetAnim visualization (triangle topology clearly shows redundant paths)

This design provides the foundation for testing disaster recovery scenarios, failover mechanisms, and multi-hop WAN performance in the RegionalBank environment.

b) **Static Routing Complexity**: to implement proper failover, we need to add redundant routes with different metrics on DR-B using custom functions such as **SetupInitialRouting**, **FailoverToBackup** and **RecoverToPrimary**.

In production routers (Cisco IOS fo examples), here's how it’s implemented :

```
! ============================================================
! DR-B Router Configuration (Real Cisco Router)
! ============================================================

interface GigabitEthernet0/0
 description Primary Link to DC-A (Network 2)
 ip address 10.1.2.2 255.255.255.0
 no shutdown

interface GigabitEthernet0/1
 description Backup Link to DC-A (Network 3)
 ip address 10.1.3.2 255.255.255.0
 no shutdown

! PRIMARY ROUTE (Administrative Distance = 1, default for static)
ip route 10.1.1.0 255.255.255.0 10.1.2.1 1 name PRIMARY_TO_BRANCH

! BACKUP ROUTE (Administrative Distance = 10, floating static)
ip route 10.1.1.0 255.255.255.0 10.1.3.1 10 name BACKUP_TO_BRANCH

! Result: Primary route always preferred unless link goes down
! When GigabitEthernet0/0 fails, route is removed from routing table
! Backup route (AD=10) is automatically installed
```

Metrics are used **within the same routing protocol** to choose the best path when multiple routes to the same destination exist with the same AD.

Administrative Distance Table:

Route Source	Administrative Distance	Use Case
Directly Connected	0	Always preferred
Static Route (Primary)	**1**	**Preferred manual route**
EIGRP Summary	5	EIGRP internal
OSPF	110	Dynamic routing protocol
Static Route (Backup)	**10-250**	**Floating static route**
External EIGRP	170	Less trusted
BGP	200	Internet routes
Unknown	255	Never used

Key Concept: Lower AD = More trusted = Preferred route

c) **Simulating Link Failure & Business Continuity Verification**: to do this, we will use the 'Error' Model, which simulates packet loss by injecting errors.

a) STATIC vs OSPF CONVERGENCE COMPARISON

STATIC ROUTING:

- Convergence: MANUAL
- Time: Depends on administrator action
- Typical: Minutes to hours (human response time)
- In simulation: ~1-2ms (scripted failover)
- Advantages:
  Predictable routing paths
  No protocol overhead
  Full administrative control
- Disadvantages:
  NO automatic failover
  Requires manual intervention
  Extended outages during failures

OSPF DYNAMIC ROUTING:

- Convergence: AUTOMATIC
- Time: ~1-5 seconds (typical)
- In simulation: <100ms (NS-3 global routing)
- Process:
  1. Link failure detected

2. LSA flooded to all routers
3. SPF recalculated (Dijkstra)
4. New routes installed

• Advantages:
- AUTOMATIC failover (no human needed)
- Fast convergence (seconds)
- Always uses optimal path
- Scales to large networks

• Disadvantages:
- Protocol overhead (Hello packets, LSAs)
- CPU/memory for SPF calculation
- More complex configuration

RECOMMENDATION FOR REGIONALBANK:
• Deploy OSPF for critical WAN links because:
• Automatic failover ensures business continuity
• Sub-second convergence minimizes downtime
• Meets banking industry uptime requirements (99.99%)
• Reduces operational overhead (no manual intervention)

For a 3-node topology, overhead is negligible Vs the massive benefit of automatic failover.

d) **Business Continuity Verification**:

a)

Added tools: custom trace sinks (**PrimaryPathTxTrace**, **PrimaryPathRxTrace**) attached to network2devices, **path metrics tracking** (**g_primaryPathMetrics struct**) counting packets on each path and **physical layer monitoring** using **PhyTxEnd** and **PhyRxEnd** traces on the primary link.

b)

Added tools: backup path trace sinks (**BackupPathTxTrace**, **BackupPathRxTrace**) on Network 3 devices, **convergence detection** in existing **PacketReceivedCallback** that timestamps when traffic resumes and **path activity comparison** showing which links carried traffic at what times.

c)

Added tools: flowMonitor installation (**g_flowMonitorHelper.InstallAll()**) tracking all UDP flows, **per-flow metrics** extraction: delay, jitter, loss ratio, throughput, **aggregate statistics** calculation across all transactions and **XML export** for detailed post-simulation analysis.

`Exercise 5`: *Policy-based routing for application-aware WAN path selection*

1. Traffic Classification Logic

- **Protocol Differentiation**: UDP vs TCP allows router to classify at L4
- **Port-based Classification**: Different ports (5004 vs 2100) enable port-based PBR rules
- **DSCP Marking**: ToS field (0xb8 vs 0x28) enables DiffServ-based QoS
- **Packet Size**: Dramatically different sizes (180 vs 1460) enable size-based filtering
- **Traffic Pattern**: Constant vs bursty allows statistical classification

Traffic Characteristics Summary

Feature	**Flow_Video (RTP-like)**	**Flow_Data (FTP-like)**
Protocol	UDP	TCP
Port	5004	2100
Packet Size	180 bytes	1460 bytes
Rate	72 kbps (constant)	4 Mbps (bursty)
Pattern	Periodic (50 pkt/s)	Exponential On/Off
DSCP Marking	EF (0xb8) - Priority	AF11 (0x28) - Best effort
Socket Type	UdpSocketFactory	TcpSocketFactory
Latency Sensitivity	HIGH (< 50ms)	LOW (100-200ms OK)

2. **Implementing PBR (Policy-Based Routing) in NS-3**:

- Custom routing decision function - bypasses default routing table. This is the CORE PBR logic that intercepts packets at IP layer.
- Device: The input device where packet arrived
- Packet: The packet to be routed
- IPHeader: The IPv4 Header
- UCB: Unicast Forward Callback (normal forwarding)
- MCB: Multicast Forward Callback
- LCB: Local Deliver CallBack
- ECB: Error CallBack

Description of packet processing

- [FLOW_VIDEO] RTP-like Control Traffic:

Source: Studio (10.1.1.1) -> Destination: Cloud (10.1.2.2)
Port: UDP/5004
Packet Size: 180 bytes
Rate: 72 kbps (50 packets/sec)
Pattern: Constant, periodic
DSCP Marking: EF (0xb8) - HIGH PRIORITY
Latency Requirement: < 50ms

- [FLOW_DATA] FTP-like Bulk Transfer:

Source: Studio (10.1.1.1) -> Destination: Cloud (10.1.2.2)
Port: TCP/2100
Packet Size: 1460 bytes
Rate: 4 Mbps (bursty)
Pattern: Exponential On/Off (2s ON, 1s OFF avg)
DSCP Marking: AF11 (0x28) - LOWER PRIORITY
Latency Tolerance: ~100-200ms acceptable

3. **Path Characterization**:
   a) Data collection layer

```
// NS-3 fires these callbacks automatically:
routerIpv4L3->TraceConnectWithoutContext("Tx", MakeCallback(&TxTraceCallback));
cloudIpv4L3->TraceConnectWithoutContext("Rx", MakeCallback(&RxTraceCallback));
routerIpv4L3->TraceConnectWithoutContext("Drop", MakeCallback(&DropTraceCallback));
```

   b) Metric computation layer

```
// Exponential Weighted Moving Average - same algorithm TCP uses for RTT
double alpha = 0.125;
averageLatency = alpha * newSample + (1 - alpha) * oldAverage;
```

   c) PBR decision engine

```
if (TRAFFIC_VIDEO)
   use Network2;
if (TRAFFIC_DATA)
  use Network3;
```

Sum Up: Path Characterization = Continuous Monitoring + Smart Decisions

1. **Collect** raw data (packet Tx/Rx times, bytes)
2. **Compute** meaningful metrics (latency, bandwidth, congestion)
3. **Decide** which path to use based on current conditions
4. **Adapt** dynamically as network state changes

This creates **adaptive PBR** that responds to actual network conditions rather than just static rules.

4. **Dynamic Policy Engine**:
   The behavior:

- T=0s: Controller starts Video → Network 2 (latency ~7ms) Data → Network
- T=1s: [SDWAN-CTRL] Policy Evaluation Video latency: 7ms < 30ms threshold → No action
- T=2s: [SDWAN-CTRL] Policy Evaluation All metrics normal → No action
- T=4s: [TEST] CONGESTION INJECTED Network 2 latency → 50ms
- T=5s: [SDWAN-CTRL] Policy Evaluation Video latency: 50ms > 30ms threshold [SDWAN-CTRL] SWITCHING to SECONDARY path Video → Network 3 (switched!)
- T=8s: [TEST] CONGESTION REMOVED Network 2 latency → 7ms
- T=13s: [SDWAN-CTRL] Policy Evaluation Cooldown expired (5s since T=5s) Primary recovered: 7ms < 30ms [SDWAN-CTRL] SWITCHING back to PRIMARY Video → Network 2 (switched back!)

5. **Validation and Trade-offs**:

   a) VALIDATION STRATEGY
   A.Functional Validation - Proving PBR Works Correctly
   B. Add Validation to main()

   b) COMPUTATIONAL OVERHEAD ANALYSIS
   Add Performance Profiling

   c) TRADE-OFFS DISCUSSION
   Add Scalability Analysis Report

Key Scalability Limits

- **Increased Processing Overhead**: Unlike traditional destination-based routing, which is often hardware-accelerated (using ASICs or NPUs), PBR typically relies on the device's main CPU to inspect each packet against a set of defined rules. As the number of flows and associated policies increases, the CPU must work harder to make forwarding decisions, leading to a performance bottleneck, increased latency, and potentially dropped packets.
- **Per-Flow State Requirement**: Some PBR solutions, particularly those in certain MPLS-based routing contexts, require maintaining a per-flow state. This creates a significant memory and processing burden as the number of active flows grows, ultimately limiting the scalability of the solution.
- **Administrative Complexity**: Each unique policy for a specific flow (defined by source/destination IP, port, protocol, etc.) must be manually configured and maintained. A large number of flows translates directly to a large number of potential policies, making the configuration and ongoing management a complex and time-consuming administrative task.
- **Configuration Management**: PBR configurations are generally locally significant to a specific device. To apply a consistent policy across multiple points in a network path, PBR must be configured hop-by-hop, which is difficult to scale across large or dynamic networks.

`Exercise 6`: *Inter-AS routing simulation for multi-provider WAN*

1.
2.
3.
4.
5.