Understanding NS3

What is NS3?

NS3 is an **open-source network simulator** written in C++ with optional Python bindings. Key features include:

- Discrete-event simulation engine
- Realistic network modeling capabilities
- Support for various network protocols and technologies
- Ability to connect to real networks
- Comprehensive tracing and visualization tools

NS3 is a **discrete event simulator**.

What is discrete event simulation?

Discrete event simulation is a method of modeling complex systems by representing them as a sequence of distinct events occurring at specific points in time. In the context of network simulation, such as in NS3, this approach is particularly useful for simulating network behavior and performance.

Key aspects of discrete event simulation include:

1. **Events:** These are occurrences that change the state of the system. In network simulations, events might include packet transmissions, receptions, or link state changes.
2. **Event Queue:** The simulator maintains a queue of scheduled events, ordered by their occurrence time.
3. **Simulation Clock:** Unlike real-time simulations, the simulation clock advances from one event to the next, skipping periods where no events occur.
4. **State Variables:** These represent the system's condition at any given time, such as buffer occupancy or link status.
5. **Statistical Accumulators:** Used to gather performance metrics during the simulation.

In NS3, the discrete event simulation engine manages these components, allowing researchers to model complex network scenarios efficiently. This approach enables the simulation of large-scale networks over extended periods without the need for real-time execution, making it possible to study long-term behavior and rare events that might be difficult to observe in real networks.

The simulation progresses by processing events in chronological order, updating state variables, and scheduling new events as necessary. This method provides a balance between accuracy and computational efficiency, making it ideal for studying network protocols, topologies, and performance under various conditions.

NS3 Core Concepts

1. **Nodes:** In NS3, a network is a collection of nodes. Everything is attached to a Node, or attached to things attached to a Node.

2. **NetDevices:** NetDevices represent network interface cards that connect nodes to channels.

3. **Channels:** Channels represent the medium through which data is transmitted between nodes.

4. **Applications:** Applications generate traffic in the network and are installed on nodes.

5. **Packets:** Packets are the basic units of data transmission in the network.

Understanding the simulation scripts

1. Create a new file in the `scratch` directory:

   ```
   cd ns-allinone-3.36.1/ns-3.36.1/

   gedit scratch/first-simulation.cc
   ```

2. Open the file in your favorite editor and add the following code:

   ```
   #include "ns3/core-module.h"
   #include "ns3/network-module.h"
   #include "ns3/internet-module.h"
   #include "ns3/point-to-point-module.h"
   #include "ns3/applications-module.h"
   ```

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE("FirstSimulation");

int main() {
    // Enable logging
    LogComponentEnable("UdpEchoClientApplication",
LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication",
LOG_LEVEL_INFO);

    // Create nodes
    NodeContainer nodes;
    nodes.Create(3);

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

    // Install devices on nodes
    NetDeviceContainer devices01 =
pointToPoint.Install(nodes.Get(0), nodes.Get(1));
    NetDeviceContainer devices12 =
pointToPoint.Install(nodes.Get(1), nodes.Get(2));

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

    // Assign IP addresses
    Ipv4AddressHelper address;
    address.SetBase("10.1.1.0", "255.255.255.0");
    Ipv4InterfaceContainer interfaces01 =
address.Assign(devices01);
```

```
    address.SetBase("10.1.2.0", "255.255.255.0");
    Ipv4InterfaceContainer interfaces12 =
address.Assign(devices12);

    // Create UDP server on node 2
    UdpEchoServerHelper echoServer(9);
    ApplicationContainer serverApps =
echoServer.Install(nodes.Get(2));
    serverApps.Start(Seconds(1.0));
    serverApps.Stop(Seconds(10.0));

    // Create UDP client on node 0
    UdpEchoClientHelper echoClient(interfaces12.GetAddress(1),
9);
    echoClient.SetAttribute("MaxPackets", UintegerValue(10));
    echoClient.SetAttribute("Interval",
TimeValue(Seconds(1.0)));
    echoClient.SetAttribute("PacketSize", UintegerValue(1024));

    ApplicationContainer clientApps =
echoClient.Install(nodes.Get(0));
    clientApps.Start(Seconds(2.0));
    clientApps.Stop(Seconds(10.0));

    // Enable pcap tracing
    pointToPoint.EnablePcapAll("first-simulation");

    // Run simulation
    Simulator::Run();
    Simulator::Destroy();

    return 0;
}
```

3. Run the simulation:

```
./waf --run scratch/first-simulation
```

Understanding the Simulation Structure

Every NS3 simulation follows this general structure:

1. **Include necessary modules**
2. **Create nodes**
3. **Create and configure network devices and channels**
4. **Install protocol stacks on nodes**
5. **Assign IP addresses**
6. **Create and configure applications**
7. **Set up tracing (optional)**
8. **Run the simulation**
9. **Clean up resources**

Analyzing Simulation Results

NS3 provides several ways to analyze simulation results:

1. **Console Output**: Basic information printed to the console
2. **PCAP Files**: Can be analyzed with tools like Wireshark
3. **Trace Files**: Custom trace files for detailed analysis
4. **Flow Monitor**: For collecting statistics on network flows

As you become more comfortable with NS3, you can explore:

1. Different network topologies (bus, star, mesh)
2. Wireless networks (Wi-Fi, LTE)
3. Routing protocols
4. Custom applications and protocols

5. Visualization using NetAnim

Debugging Tips

1. Use NS_LOG_COMPONENT_DEFINE and LogComponentEnable for logging
2. Enable pcap tracing to capture packet details
3. Use GDB for debugging C++ code
4. Check the NS3 documentation and examples for reference

Detailed Explanation of the NS3 Simulation Code

The provided C++ code implements Experiment 1 from the previous lab assignment, creating a three-node network topology with UDP client-server applications. Let me break down the code in detail:

Header Inclusions

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/flow-monitor-helper.h"
```

These lines include necessary NS3 modules:

- `core-module`: Basic NS3 functionality
- `network-module`: Network components like nodes and packets
- `internet-module`: IP addressing and routing
- `point-to-point-module`: Point-to-point connections

- `applications-module`: Network applications like UDP client/server
- `flow-monitor-module` and `flow-monitor-helper`: For monitoring network flows

Namespace and Logging Setup

```
using namespace ns3;
NS_LOG_COMPONENT_DEFINE("ThreeNodeTopology");
```

This sets up the NS3 namespace and defines a logging component named "ThreeNodeTopology".

Main Function

```
int main() {
    // Log level setting
    LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

The main function begins by enabling logging for UDP client and server applications at the INFO level, which will display basic information about packet transmission.

Node Creation

```
    // Create nodes
    NodeContainer nodes;
    nodes.Create(3);
```

This creates a container of three nodes (Node0, Node1, Node2) as specified in the assignment.

Link Configuration

```
    // Create point-to-point link attributes
    PointToPointHelper pointToPoint;
    pointToPoint.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
    pointToPoint.SetChannelAttribute("Delay", StringValue("1ms"));
```

This configures the point-to-point links with a data rate of 5 Mbps and a delay of 1 ms. The delay parameter is what would be varied to measure throughput changes as required in step 2 of the assignment.

Device Installation

```
    // Install devices and channels between Node0 and Node1, and Node1 and Node2
    NetDeviceContainer device0_1 = pointToPoint.Install(nodes.Get(0), nodes.Get(1));
    NetDeviceContainer device1_2 = pointToPoint.Install(nodes.Get(1), nodes.Get(2));
```

This creates two point-to-point links: one between Node0 and Node1, and another between Node1 and Node2.

Internet Stack Installation

cpp

```
    // Install Internet Stack
    InternetStackHelper stack;
    stack.Install(nodes);
```

This installs the TCP/IP protocol stack on all three nodes.

IP Address Assignment

cpp

```
    // Assign IP addresses
    Ipv4AddressHelper address;
    address.SetBase("10.1.1.0", "255.255.255.0");
    Ipv4InterfaceContainer interfaces0_1 = address.Assign(device0_1);
    address.SetBase("10.1.2.0", "255.255.255.0");
    Ipv4InterfaceContainer interfaces1_2 = address.Assign(device1_2);
```

This assigns IP addresses to the network interfaces:

- The link between Node0 and Node1 uses the 10.1.1.0/24 subnet
- The link between Node1 and Node2 uses the 10.1.2.0/24 subnet

First UDP Server Configuration

cpp

```
    // Create UDP echo server on Node2
    uint16_t serverPort = 9;
    UdpEchoServerHelper server1(serverPort);
    ApplicationContainer serverApp1 = server1.Install(nodes.Get(2));
    serverApp1.Start(Seconds(1.0));
    serverApp1.Stop(Seconds(10.0));
```

This creates a UDP echo server on Node2 listening on port 9. The server starts at 1 second into the simulation and stops at 10 seconds.

First UDP Client Configuration

cpp

```
    // Create UDP echo client on Node0
    UdpEchoClientHelper client1(interfaces1_2.GetAddress(1),
serverPort);
    client1.SetAttribute("MaxPackets", UintegerValue(10));
    client1.SetAttribute("Interval", TimeValue(Seconds(1.0)));
    client1.SetAttribute("PacketSize", UintegerValue(1024));
    ApplicationContainer clientApp1 = client1.Install(nodes.Get(0));
    clientApp1.Start(Seconds(2.0));
    clientApp1.Stop(Seconds(10.0));
```

This creates a UDP echo client on Node0 that sends packets to the server on Node2. The client:

- Targets the IP address of Node2 on port 9
- Sends a maximum of 10 packets
- Sends packets at 1-second intervals
- Uses 1024-byte packets
- Starts at 2 seconds and stops at 10 seconds

Second UDP Server Configuration

```
  // Adding another UDP echo server on Node2 on a different port
    uint16_t serverPort2 = 10;
    UdpEchoServerHelper server2(serverPort2);
    ApplicationContainer serverApp2 = server2.Install(nodes.Get(2));
    serverApp2.Start(Seconds(1.0));
    serverApp2.Stop(Seconds(10.0));
```

This creates a second UDP echo server on Node2, but listening on port 10 to avoid conflicts with the first server (addressing step 3 of the assignment).

Second UDP Client Configuration

```
  // Adding another UDP echo client on Node0
    UdpEchoClientHelper client2(interfaces1_2.GetAddress(1),
serverPort2);
    client2.SetAttribute("MaxPackets", UintegerValue(10));
    client2.SetAttribute("Interval", TimeValue(Seconds(1.0)));
    client2.SetAttribute("PacketSize", UintegerValue(1024));
    ApplicationContainer clientApp2 = client2.Install(nodes.Get(0));
    client2.Start(Seconds(2.0));
    client2.Stop(Seconds(10.0));
```

This creates a second UDP echo client on Node0 that sends packets to the second server on Node2 (port 10).

Packet Capture Setup

```
  // Enable pcap tracing
    pointToPoint.EnablePcapAll("three-node-topology");
```

This enables packet capture (pcap) for all point-to-point links, generating files that can be analyzed with Wireshark to verify correct packet delivery (addressing step 4 of the assignment).

Simulation Execution

```
  // Run the simulation
    Simulator::Run();
    Simulator::Destroy();
```

```
    return 0;
}
```

Finally, the simulation is run and then cleaned up before the program exits.

This code successfully implements all requirements from Experiment 1, creating a three-node topology with two point-to-point links, setting up UDP clients and servers, and enabling packet capture for analysis.

What is an Error Model?

In the context of NS3, an error model is a component used to simulate packet corruption or loss in network simulations. Error models are typically applied to network devices to mimic channel losses and evaluate how protocols behave under imperfect network conditions.

Error Model Fundamentals

Error models in NS3 are objects that determine whether packets should be considered corrupted or lost during transmission. The core functionality is provided through the `IsCorrupt()` method, which evaluates a packet and returns true if the packet should be considered errored.

There are two main types of error models in NS3:

1. **Stochastic Models**: These error packets according to random variable distributions. Examples include:
    - `RateErrorModel`: Corrupts packets based on a specified error rate and unit (bit, byte, or packet)
    - `BurstErrorModel`: Simulates burst error patterns
2. **Deterministic Models**: These error packets according to a prescribed pattern. Examples include:
    - `ListErrorModel`: Allows users to specify exactly which packets should be errored.
    - `ReceiveListErrorModel`: Similar to ListErrorModel but with different implementation details.

Adding an Error Model to Our Simulation

To add an error model to our previous three-node topology simulation, we can modify the code as follows:

```
// After setting up the network devices and before starting the
simulation
// Create an error model
Ptr<RateErrorModel> em = CreateObject<RateErrorModel>();
em->SetAttribute("ErrorRate", DoubleValue(0.01));  // 1% error rate
em->SetAttribute("ErrorUnit",
EnumValue(RateErrorModel::ERROR_UNIT_PACKET));

// Apply the error model to Node2's receiving device
// This will cause 1% of packets received by Node2 to be dropped
NetDeviceContainer device1_2 = pointToPoint.Install(nodes.Get(1),
nodes.Get(2));
device1_2.Get(1)->SetAttribute("ReceiveErrorModel", PointerValue(em));
```

This code creates a `RateErrorModel` with a 1% packet error rate and applies it to the receiving device on Node2. When packets are transmitted from Node1 to Node2, approximately 1% of them will be flagged as corrupted by the error model.

Analyzing the Impact of Errors

To analyze the impact of the error model, we can count the number of packets sent and received:

```
// Add counters for sent and received packets
uint32_t packetsSent = 0;
uint32_t packetsReceived = 0;

// Set up callbacks to count packets
Config::ConnectWithoutContext("/NodeList/0/ApplicationList/*/$ns3::Udp
EchoClient/Tx",
                              MakeCallback(&IncrementCounter,
&packetsSent));
Config::ConnectWithoutContext("/NodeList/2/ApplicationList/*/$ns3::Udp
EchoServer/Rx",
                              MakeCallback(&IncrementCounter,
&packetsReceived));
```

```
// Define the callback function
static void IncrementCounter(Ptr<const Packet> p, uint32_t* counter) {
  (*counter)++;
}

// After simulation completes
double errorRate = (packetsSent > 0) ?
                   (double)(packetsSent - packetsReceived) /
packetsSent : 0;
std::cout << "Packets sent: " << packetsSent << std::endl;
std::cout << "Packets received: " << packetsReceived << std::endl;
std::cout << "Error rate: " << errorRate * 100 << "%" << std::endl;
```

This will allow us to measure the actual error rate observed in the simulation and compare it to our configured value of 1%.

By incorporating error models into our simulations, we can create more realistic network scenarios and test how protocols and applications perform under various error conditions.

To analyze network performance in NS3, we can calculate metrics like throughput, round trip time, and packet loss using various built-in tools. Here's how to implement these measurements in our simulation:

Using Flow Monitor

Flow Monitor is a powerful NS3 module that collects statistics about network flows:

```
// Add at the top with other includes

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


// Add after setting up applications but before Simulator::Run()

// Create flow monitor

Ptr<FlowMonitor> flowMonitor;

FlowMonitorHelper flowHelper;
```

```
flowMonitor = flowHelper.InstallAll();


// Run simulation

Simulator::Run();


// Collect statistics after simulation

flowMonitor->CheckForLostPackets();

Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowHelper.GetClassifier());

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


// Print flow statistics

for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
stats.begin(); i != stats.end(); ++i) {

  Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(i->first);


  // Calculate throughput in Mbps

  double throughput = i->second.rxBytes * 8.0 /
(i->second.timeLastRxPacket.GetSeconds() -
i->second.timeFirstTxPacket.GetSeconds()) / 1000000;


  std::cout << "Flow " << i->first << " (" << t.sourceAddress << ":"
<< t.sourcePort

            << " -> " << t.destinationAddress << ":" <<
t.destinationPort << ")\n";
```

```
  std::cout << "  Throughput: " << throughput << " Mbps\n";

  std::cout << "  Packet Loss: " << i->second.lostPackets << " / " <<
(i->second.txPackets)

            << " (" <<
((double)i->second.lostPackets/i->second.txPackets * 100) << "%)\n";

  std::cout << "  Mean Delay: " << i->second.delaySum.GetSeconds() /
i->second.rxPackets * 1000 << " ms\n";

  std::cout << "  Mean Jitter: " << i->second.jitterSum.GetSeconds() /
i->second.rxPackets * 1000 << " ms\n";

}
```

Measuring Round Trip Time

To measure RTT specifically, we can use callbacks to track when packets are sent and when acknowledgments are received:

```
// Add these variables at the beginning of main()

Time sendTime[1000] = {};  // Array to store send times

uint32_t totalPackets = 0;

double totalRtt = 0;


// Create callback functions

static void PacketSent(Ptr<const Packet> packet, uint32_t packetId) {

  sendTime[packetId] = Simulator::Now();

}
```

```
static void PacketReceived(Ptr<const Packet> packet, uint32_t
packetId) {

  Time rtt = Simulator::Now() - sendTime[packetId];

  totalRtt += rtt.GetSeconds();

  totalPackets++;

  std::cout << "Packet " << packetId << " RTT: " << rtt.GetSeconds() *
1000 << " ms\n";

}



// Connect callbacks to the UdpEchoClient

Config::ConnectWithoutContext("/NodeList/0/ApplicationList/*/$ns3::Udp
EchoClient/TxWithAddresses",

                              MakeCallback(&PacketSent));

Config::ConnectWithoutContext("/NodeList/0/ApplicationList/*/$ns3::Udp
EchoClient/RxWithAddresses",

                              MakeCallback(&PacketReceived));



// After simulation, calculate average RTT

if (totalPackets > 0) {

  std::cout << "Average RTT: " << (totalRtt / totalPackets) * 1000 <<
" ms\n";

}
```

Measuring Throughput Over Time

To observe how throughput changes over time:

```
// Add these variables

uint32_t bytesReceived = 0;

Time lastSampleTime = Seconds(0);

std::vector<std::pair<double, double>> throughputSamples;


// Create callback function

static void PacketReceivedForThroughput(Ptr<const Packet> packet,
const Address &address) {

  bytesReceived += packet->GetSize();

  Time now = Simulator::Now();

  double interval = now.GetSeconds() - lastSampleTime.GetSeconds();


  if (interval >= 0.5) {  // Sample every 0.5 seconds

    double throughput = (bytesReceived * 8.0 / interval) / 1000000;
// Mbps

    throughputSamples.push_back(std::make_pair(now.GetSeconds(),
throughput));

    std::cout << "Time: " << now.GetSeconds() << "s, Throughput: " <<
throughput << " Mbps\n";


    bytesReceived = 0;

    lastSampleTime = now;

  }

}
```

```
// Connect callback to server application

Config::ConnectWithoutContext("/NodeList/2/ApplicationList/*/$ns3::Udp
EchoServer/Rx",


MakeCallback(&PacketReceivedForThroughput));
```

Varying Link Parameters to Analyze Performance

To study how link parameters affect performance, we can modify our simulation to run multiple times with different parameters:

cpp

```
// Replace the fixed delay value with a loop

std::vector<std::string> delays = {"1ms", "5ms", "10ms", "20ms",
"50ms"};

std::vector<double> throughputs;


for (const auto& delay : delays) {

  // Reset the simulation

  Simulator::Destroy();


  // Create nodes

  NodeContainer nodes;

  nodes.Create(3);
```

```
  // Configure point-to-point links with current delay
  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
  pointToPoint.SetChannelAttribute("Delay", StringValue(delay));

  // Continue with the rest of the setup...

  // Run simulation and collect results
  double currentThroughput = MeasureThroughput();  // Implement this function to return throughput
  throughputs.push_back(currentThroughput);

  std::cout << "Link delay: " << delay << ", Throughput: " << currentThroughput << " Mbps\n";
}
```

Generating Visual Reports

For visual analysis, we can export data to files that can be processed by graphing tools:

cpp

```
// After collecting all data
std::ofstream throughputFile("throughput-vs-delay.dat");
for (size_t i = 0; i < delays.size(); ++i) {
  throughputFile << delays[i] << " " << throughputs[i] << std::endl;
}
```

```
throughputFile.close();


// Also export Flow Monitor data to XML

flowMonitor->SerializeToXmlFile("flow-monitor-results.xml", true,
true);
```

By implementing these techniques, you can comprehensively analyze network performance under various conditions and understand how different parameters affect metrics like throughput, RTT, and packet loss.