**Laboratory Journal**

*Submitted in partial fulfilment of the requirement*

*for the Laboratory*

**‘Communication Networks Laboratory’**

**(Code – EC 314)**

Prepared and Submitted by:

Mr. \_\_Danish Ahmed Mehmuda\_\_

(Admission No.: \_U19EC111\_)

B.Tech III (EC), Semester - VI

(2021-22)

Laboratory Teacher: Dr. Raghavendra Pal



**Department of Electronics Engineering**

**Sardar Vallabhbhai National Institute of Technology**

**Surat-395007, Gujarat, INDIA.**

**Sardar Vallabhbhai National Institute of Technology**

Surat, Gujarat, INDIA

**DEPARTMENT OF ELECTRONICS ENGINEERING**

**2021-22**



**Subject- Communication Networks Lab (Code – EC314)**

Certificate

This is to certify that the Laboratory Journal is prepared & submitted by **B.Tech III (Semester-VI)** student Mr. \_\_\_\_Danish Ahmed Mehmuda\_\_\_\_\_ bearing **Admission No.** \_\_\_U19EC111\_\_\_ in the partial fulfilment of the requirement for the laboratory **Communication Networks Lab (Code-EC314)** through ONLINE MODE.

We, certify that the work is comprehensive, complete and fit for evaluation.

**Laboratory Teacher:**

**Name Signature with Date**

1. Dr. Raghavendra Pal

**Jan-May 2022**

**Communication Networks Lab (Code – EC314)**

**Academic Year (2021-22)**

**LIST OF EXPERIMENTS**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **List of Experiments** | **Page No.** |
| **1.** | Introduction to TCP/IP Networking Commands. |  |
| **2.** | Cyclic Redundancy Check (CRC) Method for Error Detection. |  |
| **3.** | Hamming Code for Error Detection and Correction. |  |
| **4.** | Bit Stuffing. |  |
| **5.** | Shortest Path Routing Algorithm. |  |
| **6.** | Symmetric Key Ciphering and Deciphering using Classical Ciphers. |  |
| **7.** | Asymmetric Key Ciphering and Deciphering using Modern Ciphers. |  |
| **8.** | Introduction to Network Simulator (NS3) |  |
| **9.** | Elementary Network Model design using NS3 |  |
| **10.** | Dynamic Network Model design for Optimum Routing using NS3 |  |
| **11.** | Local Area Network Architecture and its Performance Analysis using NS3 |  |
| **12.** | Wireless Network Implementation using NS3. |  |

**Submitted By**

**Name: Danish Ahmed Mehmuda**

**Admission Number: U19EC111**

**Class/Year/Branch: B.Tech III ECED**

**Experiment 1**

**Date:**

**Aim:** To perform

**Tools Required:** MATLAB

**Theory:**

**Code:**

**Output Results/Graphs**

**Conclusions:**

**Experiment 2**

**Date:**

**Aim:** To perform cyclic Redundancy Check (CRC) Method for Error Detection.

**Tools Required:** MATLAB

**Theory:**

CRC or Cyclic Redundancy Check is a method of detecting accidental changes/errors in the communication channel.   
CRC uses **Generator Polynomial**which is available on both sender and receiver side. An example generator polynomial is of the form like x3 + x + 1. This generator polynomial represents key 1011. Another example is x2 + 1 that represents key 101.

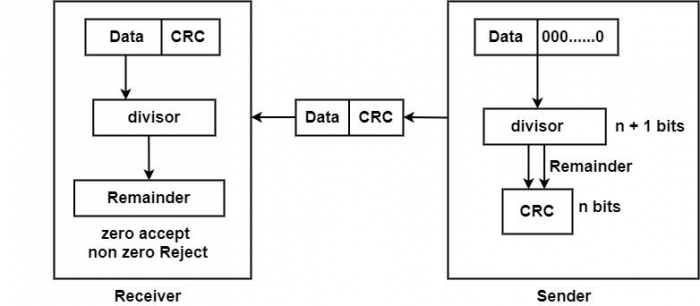
n : Number of bits in data to be sent from sender side.

k : Number of bits in the key obtained from generator polynomial.

**Qualities of CRC**

* It should have accurately one less bit than the divisor.
* Joining it to the end of the data unit should create the resulting bit sequence precisely divisible by the divisor.

**CRC generator and checker**

****

**Sender Side (Generation of Encoded Data from Data and Generator Polynomial (or Key)):**

1. The binary data is first augmented by adding k-1 zeros in the end of the data
2. Use ***modulo-2 binary division*** to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same

**Receiver Side (Check if there are errors introduced in transmission)**  
Perform modulo-2 division again and if the remainder is 0, then there are no errors.

In this article we will focus only on finding the remainder i.e. check word and the code word.

**Modulo 2 Division:**  
The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

* In each step, a copy of the divisor (or data) is XORed with the k bits of the dividend (or key).
* The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
* When there are no bits left to pull down, we have a result. The (n-1)-bit remainder which is appended at the sender side.

**Code:**

clear all;

close all;

clc;

d\_w=[1 1 0 1];

div=[1 0 1 0];

%% Transmitter

d\_w\_length=length(d\_w);

div\_length=length(div);

remainder=d\_w(1:div\_length);

for i=(div\_length):(d\_w\_length)

if (remainder(1) == div(1))

remainder=bitxor(remainder,div);

end

remainder = remainder(2:div\_length);

if (i ~= d\_w\_length)

remainder = [remainder d\_w(i+1)];

end

end

c\_w= [d\_w remainder];

disp('codeword')

disp(c\_w);

%% channel noise

c\_w=awgn(c\_w,10);

disp('codeword with noise')

disp(c\_w);

%% reciever preprocess

c\_w=round(c\_w);

c\_w= c\_w > 0;

disp('codeword after preprocessing')

disp(c\_w);

%% reciever check

c\_w\_length=length(c\_w);

syndrome=c\_w(1:div\_length);

for i=(div\_length):(c\_w\_length)

if (syndrome(1) == div(1))

syndrome=bitxor(syndrome,div);

end

syndrome = syndrome(2:div\_length);

if (i ~= c\_w\_length)

syndrome = [syndrome c\_w(i+1)];

end

end

disp('syndrome')

disp(syndrome);

for i=1:length(syndrome)

if(syndrome(i)==1)

disp('Error is present');

break;

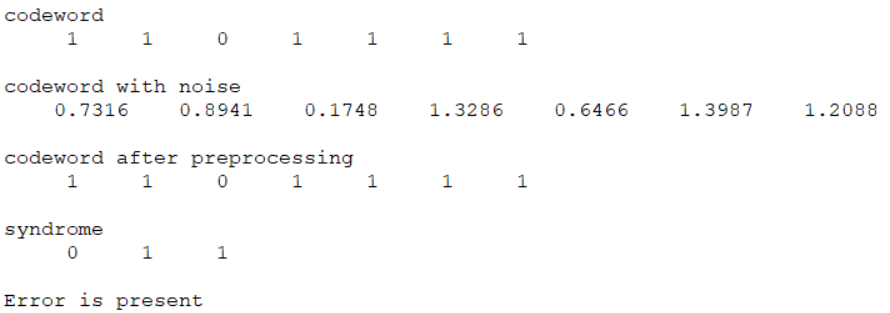
elseif (i==length(syndrome))

disp('Error is not present');

end

end

**Output:**

****

**Conclusions:**

**Experiment 3**

**Date:**

**Aim:** Use Hamming Codes for Error Detection and Correction.

**Tools Required:** MATLAB

**Theory:**

When bits are transmitted over the computer network, they are subject to get corrupted due to interference and network problems. The corrupted bits leads to spurious data being received by the receiver and are called errors.

Error-correcting codes (ECC) are a sequence of numbers generated by specific algorithms for detecting and removing errors in data that has been transmitted over noisy channels. Error correcting codes ascertain the exact number of bits that has been corrupted and the location of the corrupted bits, within the limitations in algorithm.

ECCs can be broadly categorized into two types −

* **Block codes** − The message is divided into fixed-sized blocks of bits, to which redundant bits are added for error detection or correction.
* **Convolutional codes** − The message comprises of data streams of arbitrary length and parity symbols are generated by the sliding application of a Boolean function to the data stream.

**Hamming Code**

Hamming code is a block code that is capable of detecting up to two simultaneous bit errors and correcting single-bit errors. It was developed by R.W. Hamming for error correction.

In this coding method, the source encodes the message by inserting redundant bits within the message. These redundant bits are extra bits that are generated and inserted at specific positions in the message itself to enable error detection and correction. When the destination receives this message, it performs recalculations to detect errors and find the bit position that has error.

Hamming code is a set of error-correction codes that can be used to **detect and correct the errors** that can occur when the data is moved or stored from the sender to the receiver. It is **technique developed by R.W. Hamming for error correction**.

**Redundant bits –**

Redundant bits are extra binary bits that are generated and added to the information-carrying bits of data transfer to ensure that no bits were lost during the data transfer.  
The number of redundant bits can be calculated using the following formula:

2^r ≥ m + r + 1

where, r = redundant bit, m = data bit

**Parity bits –**  
A parity bit is a bit appended to a data of binary bits to ensure that the total number of 1’s in the data is even or odd. Parity bits are used for error detection. There are two types of parity bits:

1. **Even parity bit:**  
   In the case of even parity, for a given set of bits, the number of 1’s are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurrences of 1’s an even number. If the total number of 1’s in a given set of bits is already even, the parity bit’s value is 0.
2. **Odd Parity bit –**  
   In the case of odd parity, for a given set of bits, the number of 1’s are counted. If that count is even, the parity bit value is set to 1, making the total count of occurrences of 1’s an odd number. If the total number of 1’s in a given set of bits is already odd, the parity bit’s value is 0.

**General Algorithm of Hamming code –**  
The Hamming Code is simply the use of extra parity bits to allow the identification of an error.

1. Write the bit positions starting from 1 in binary form.
2. All the bit positions that are a power of 2 are marked as parity bits.
3. All the other bit positions are marked as data bits.
4. Each data bit is included in a unique set of parity bits, as determined its bit position in binary form.

In general, each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

1. Since we check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd.
2. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

**Code:**

clc;  
close all;  
clear all;  
  
% message bits  
m = [0 1 0 1];  
n = length(m);  
r = 1;  
  
while (power(2,r) < (n+r+1))  
 r = r+1;  
end  
  
  
% Final result array  
res = zeros(1, n+r);  
  
% Find positions of redundant bits  
for i = 0:1:r-1  
 res(power(2, i)) = -1;  
end  
  
j=1;  
  
for i=1:1:r+n-1  
 if(res(i) ~= -1)  
 res(i) = m(j);  
 j = j+1;  
 end  
end  
  
for i=1:1:r+n-1  
 if(res(i) == -1)  
 x = log2(i);  
 ones = 0;  
  
 for j=i+2:1:r+n-1  
 if(j && bitsll(1, x))  
 if(res(j-1)==1)  
 ones = ones+1;  
 end  
 end  
 end  
  
 if(mod(ones,2)==0)  
 res(i) = 0;  
 else  
 res(i) = 1;  
 end  
 end  
end  
  
disp(res);

**Output Results/Graphs**

1 1 0 1 1 0 0

**Conclusions:**

**Experiment 4**

**Date:**

**Aim:** To perform bit stuffing for framing.

**Tools Required:** MATLAB

**Theory:**

Data link layer is responsible for something called Framing, which is the division of stream of bits from network layer into manageable units (called frames). Frames could be of fixed size or variable size. In variable-size framing, we need a way to define the end of the frame and the beginning of the next frame. However, if the pattern occurs in the message, then mechanisms needs to be incorporated so that this situation is avoided.

The two common approaches are −

* **Byte - Stuffing** − A byte is stuffed in the message to differentiate from the delimiter. This is also called character-oriented framing.
* **Bit - Stuffing** − A pattern of bits of arbitrary length is stuffed in the message to differentiate from the delimiter. This is also called bit - oriented framing.

Bit stuffing is the mechanism of inserting one or more non-information bits into a message to be transmitted, to break up the message sequence, for synchronization purpose.

Note that stuffed bits should not be confused with overhead bits. **Overhead bits** are non-data bits that are necessary for transmission (usually as part of headers, checksums etc.).

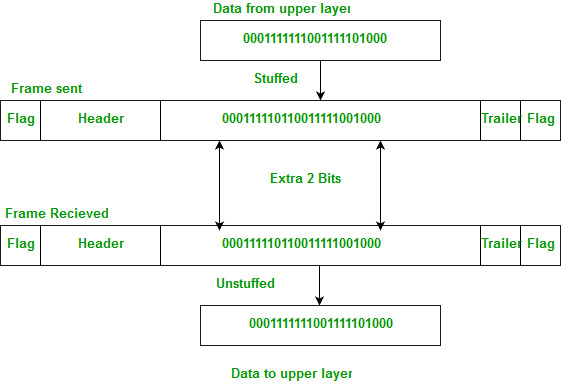
## Bit Stuffing Mechanism

In a data link frame, the delimiting flag sequence generally contains six or more consecutive 1s. In order to differentiate the message from the flag in case of the same sequence, a single bit is stuffed in the message. Whenever a 0 bit is followed by five consecutive 1bits in the message, an extra 0 bit is stuffed at the end of the five 1s.

When the receiver receives the message, it removes the stuffed 0s after each sequence of five 1s. The un-stuffed message is then sent to the upper layers.

**Applications of Bit Stuffing –**

1. synchronize several channels before multiplexing
2. rate-match two single channels to each other
3. run length limited coding



**Run length limited coding –** To limit the number of consecutive bits of the same value(i.e., binary value) in the data to be transmitted. A bit of the opposite value is inserted after the maximum allowed number of consecutive bits.

Bit stuffing technique does not ensure that the sent data is intact at the receiver side (i.e., not corrupted by transmission errors). It is merely a way to ensure that the transmission starts and ends at the correct places.

**Disadvantages of Bit Stuffing –**  
The code rate is unpredictable; it depends on the data being transmitted.

**Code:**

1. **Bit Stuffing**

clc;

clear all;

close all;

%msg=[ 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 ] %Unstuffed

%msg=[ 1 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0] %Stuffed

msg=input('Input Message binary bit stream :');

count=0;

stuffcount=0;

[M, N]=size(msg);

for j=1:N-5+stuffcount

j;

for i=j:j+5

i;

if msg(i)==1

count=count+1;

else

count=0;

break;

end

end

if(count ==6)

msg=[msg(1:j+4) 0 msg(j+5 : end)];

count=0;

stuffcount=stuffcount+1;

end

end

disp(msg);

1. **Bit Unstuffing**

clc;

clear all;

close all;

msg=input('Input Message binary bit stream :');

% msg=[ 1 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 0 ];

%umsg=[ 1 0 1 1 1 1 1 % 1 1 0 0 1 1 1 1 1 % 0 1 1 1 1 1 % 1 1 0 ];

umsg =msg;

ct=0;

cts=0;

N=length(msg);

for j=1:N-5-cts

ct=0;

for i=j:j+4

disp(i)

disp(msg(i))

if msg(i)==0

break;

else

ct=ct+1;

end

end

disp('ct')

disp(ct);

if ct==5

if msg(i+1)==0

disp('test')

umsg(i+1-cts) = [];

cts=cts+1;

j=i+1;

end

end

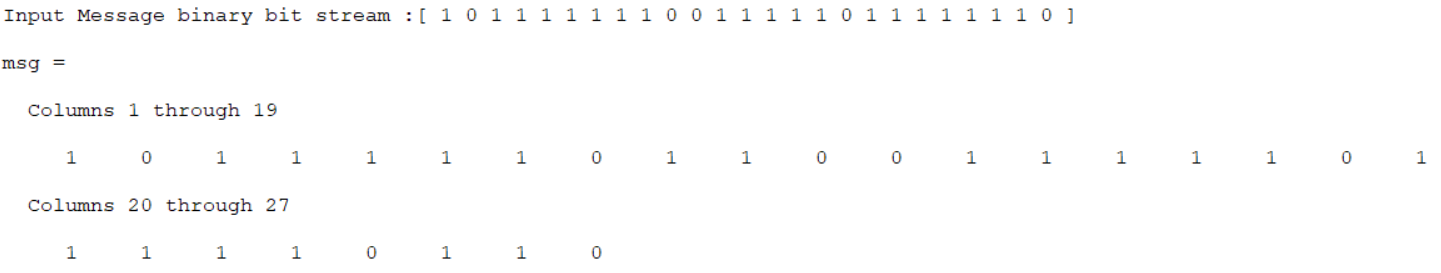
end

msg

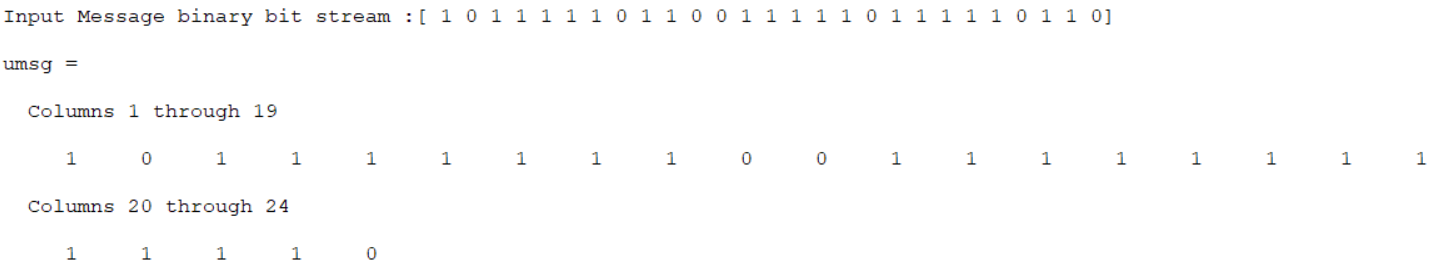
umsg

**Output:**

1. **Bit Stuffing**

****

1. **Bit Unstuffing**

****

**Conclusions:**

**Experiment 5**

**Date:**

**Aim:** To perform shortest path routing algorithm

**Tools Required:** MATLAB

**Theory:**

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF).

The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels are monotonically non-decreasing. This generalization is called the generic Dijkstra shortest-path algorithm.

**Code:**

close all;

matrix = [[0 7 7 9 Inf Inf Inf Inf],

[7 0 8 3 Inf Inf 1 Inf],

[7 8 0 Inf 1 Inf 4 Inf],

[9 3 Inf 0 Inf Inf Inf 6],

[Inf Inf 1 Inf 0 Inf 1 7],

[Inf Inf Inf Inf Inf 0 Inf 7],

[Inf 1 4 Inf 1 Inf 0 6],

[Inf Inf Inf 6 7 7 6 0]];

[path,cost]= dijkstra(matrix, 2);

display(path);

display(cost);

function [path,cost]= dijkstra(matrix, source)

nodes=length(matrix);

cost(1:nodes)=Inf;

known(1:nodes)=0;

path(1:nodes)=-1;

cost(source)=0;

curr=source;

while sum(known)<nodes

known(curr)=1;

for i=1:nodes

if (matrix(curr,i)~=Inf | matrix(curr,i)~=0) & (known(i)==0)

if(cost(i)>cost(curr)+matrix(curr,i))

cost(i)=matrix(curr,i)+cost(curr);

path(i)=curr;

end

end

end

minValue=Inf; %minimum cost from non visited nodes

minVertex=Inf; %vertex with minimum cost

for i=1:nodes

if known(i)==0

if cost(i)<minValue

minValue=cost(i);

minVertex=i;

end

end

end

curr=minVertex;

end

end

**Output Results/Graphs**

Diagram

Description automatically generated

Calendar

Description automatically generated with medium confidence

**Conclusions:**

**Experiment 6**

**Date:**

**Aim:** Implement Symmetric Key Ciphering and Deciphering using Classical Ciphers.

**Tools Required:** MATLAB

**Code:**

close all;

clear all;

file=fileread('test.txt');

n=strlength(file);

for i=1:1:n

encript(i)=bitxor(int32(file(i)),i); %Xoring to encript data

end

encripted\_data=char(encript);

display(encripted\_data);

n=strlength(encripted\_data);

for i=1:1:n

decript(i)=bitxor(int32(encripted\_data(i)),i); %Xoring again to decript data

end

decripted\_data=char(decript);

display(decripted\_data);

**Output Results/Graphs**

Graphical user interface, text, application, email

Description automatically generated

**Conclusions:**

**Experiment 7**

**Date:**

**Aim:** Implement Asymmetric Key Ciphering and Deciphering using Modern Ciphers.

**Tools Required:** MATLAB

**Code:**

close all;

text = 'This is communication networks practical';

p = 61;

q = 53;

n = p\*q;

phi = lcm((p-1),(q-1));

e=7;

d=1;

while 1

if mod(e\*d, phi) ==1

break

end

d=d+1;

end

text = double(text);

encrypt=powermod(text,e,n);

encrypted\_data = char(encrypt);

display(encrypted\_data);

t=double(encrypted\_data);

decrypt=powermod(t,d,n);

decrypted\_data=char(decrypt);

display(decrypted\_data);

**Output Results/Graphs**

Graphical user interface, text, application, email

Description automatically generated

**Conclusions:**

**Experiment 8**

**Date:**

**Aim:** Introduction to Network Simulator 3 (NS3)

**Tools Required:** Oracle Virtual Machine, Ubuntu 20.04 LTS

**Theory:**

Network simulator is a tool used for simulating the real world network on one computer by writing scripts in C++ or Python. Normally if we want to perform experiments, to see how our network works using various parameters. We don’t have required number of computers and routers for making different topologies. Even if we have these resources it is very expensive to build such a network for experiment purposes.

So to overcome these drawbacks we used NS3, which is a discrete event network simulator for Internet. NS3 helps to create various virtual nodes (i.e., computers in real life) and with the help of various Helper classes it allows us to install devices, internet stacks, application, etc to our nodes.

Using NS3 we can create PointToPoint, Wireless, CSMA, etc connections between nodes. PointToPoint connection is same as a LAN connected between two computers. Wireless connection is same as WiFi connection between various computers and routers. CSMA connection is same as bus topology between computers.

**Code for first.cc:**

/\* -\*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -\*- \*/

/\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation;

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*/

#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"

// Default Network Topology

//

// 10.1.1.0

// n0 -------------- n1

// point-to-point

//

using namespace ns3;

NS\_LOG\_COMPONENT\_DEFINE ("FirstScriptExample");

int

main (int argc, char \*argv[])

{

CommandLine cmd (\_\_FILE\_\_);

cmd.Parse (argc, argv);

Time::SetResolution (Time::NS);

LogComponentEnable ("UdpEchoClientApplication", LOG\_LEVEL\_INFO);

LogComponentEnable ("UdpEchoServerApplication", LOG\_LEVEL\_INFO);

NodeContainer nodes;

nodes.Create (2);

PointToPointHelper pointToPoint;

pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));

pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

NetDeviceContainer devices;

devices = pointToPoint.Install (nodes);

InternetStackHelper stack;

stack.Install (nodes);

Ipv4AddressHelper address;

address.SetBase ("10.1.1.0", "255.255.255.0");

Ipv4InterfaceContainer interfaces = address.Assign (devices);

UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));

serverApps.Start (Seconds (1.0));

serverApps.Stop (Seconds (10.0));

UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);

echoClient.SetAttribute ("MaxPackets", UintegerValue (1));

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));

Simulator::Run ();

Simulator::Destroy ();

return 0;

}

**Command Description:**

**NodeContainer:** Initialise a list of nodes and creates the nodes using “.Create (size)” command.

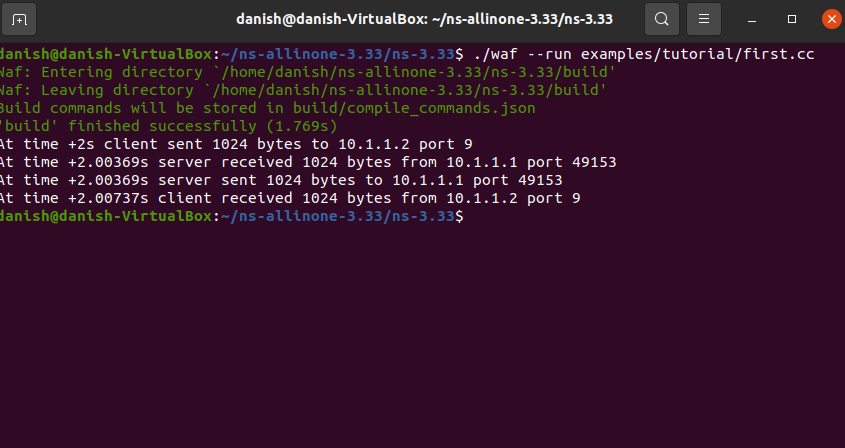
**PointToPointHelper:** Used to initialise network with point to point link.

**NetDeviceContainer:** Used to install devices that were instantiated previously.

**InternetStackHelper: A**ggregate IP/TCP/UDP functionality to existing nodes.

**UdpEchoServerHelper:** Create a server application which waits for input UDP packets and sends them back to the original sender.

**Output Results/Graphs**

****

**Conclusions:**