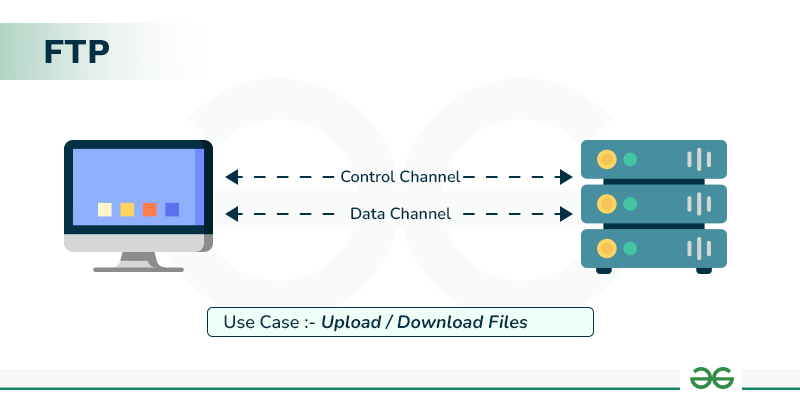
**Practical No 11**

**Aim: -** Write a Program to simulate FTP using TCP & Animate.

**Objective: -**

* Understanding of FTP
* Understanding of TCP
* Implementation and animation of FTP using TCP

**Theory: -**

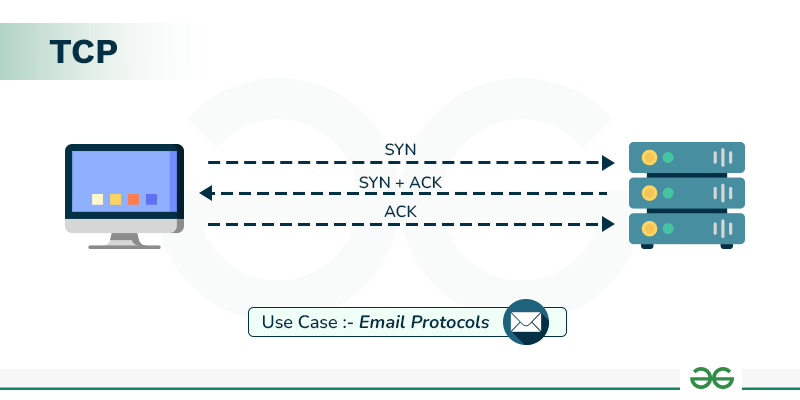
FTP stands for File Transfer Protocol. It is a standard network protocol used for transferring files between a client and a server on a computer network, such as the internet. FTP is built on a client-server architecture and operates on a reliable, connection-oriented transport protocol like TCP (Transmission Control Protocol).

**Advantages of FTP:**

* File sharing also comes in the category of advantages of FTP in this between two machines files can be shared on the network.
* Speed is one of the main benefits of FTP.
* Since we don’t have to finish every operation to obtain the entire file, it is more efficient.

**Disadvantages of FTP:**

* File size limit is the drawback of FTP only 2 GB size files can be transferred.
* More than one receiver is not supported by FTP.
* FTP does not encrypt the data this is one of the biggest drawbacks of FTP.

TCP stands for Transmission Control Protocol. It is one of the core protocols of the Internet Protocol (IP) suite and operates at the transport layer (Layer 4) of the OSI model. TCP provides reliable, connection-oriented communication between devices over a network, ensuring that data packets are delivered efficiently and in order.

**Advantages of TCP:**

* It is a reliable protocol.
* It provides an error-checking mechanism as well as one for recovery.
* It gives flow control.

**Disadvantages of TCP:**

* TCP is made for Wide Area Networks; thus, its size can become an issue for small networks with low resources.
* TCP runs several layers so it can slow down the speed of the network.
* It is not generic in nature. Meaning, it cannot represent any protocol stack other than the TCP/IP suite. E.g., it cannot work with a Bluetooth connection.

**Program: -**

#include <iostream>

#include <fstream>

#include <string>

#include "ns3/core-module.h"

#include "ns3/applications-module.h"

#include "ns3/network-module.h"

#include "ns3/internet-module.h"

#include "ns3/point-to-point-module.h"

#include "ns3/ipv4-global-routing-helper.h"

#include "ns3/netanim-module.h"

using namespace ns3;

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

// The number of bytes to send in this simulation.

static const uint32\_t totalTxBytes = 2000000;

static uint32\_t currentTxBytes = 0;

// Perform series of 1040 byte Writes (this is a multiple of 26 since

// we want to detect data splicing in the output stream)

static const uint32\_t WriteSize = 1040;

uint8\_t data[WriteSize];

std::string animFile = "ftp-animation.xml";

// These are for starting the writing process, and handling the sending

// socket's notification upcalls (events). These two together more or less

// implement a sending "Application", although not a proper ns3::Application

// subclass.

void StartFlow (Ptr<Socket>, Ipv4Address, uint16\_t);

void WriteUntilBufferFull (Ptr<Socket>, uint32\_t);

static void

CwndTracer (uint32\_t oldval, uint32\_t newval)

{

NS\_LOG\_INFO ("Moving cwnd from " << oldval << " to " << newval);

}

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

{

// Users may find it convenient to turn on explicit debugging

// for selected modules; the below lines suggest how to do this

// LogComponentEnable("TcpL4Protocol", LOG\_LEVEL\_ALL);

// LogComponentEnable("TcpSocketImpl", LOG\_LEVEL\_ALL);

// LogComponentEnable("PacketSink", LOG\_LEVEL\_ALL);

// LogComponentEnable("TcpLargeTransfer", LOG\_LEVEL\_ALL);

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

cmd.Parse (argc, argv);

// initialize the tx buffer.

for(uint32\_t i = 0; i < WriteSize; ++i)

{

char m = toascii (97 + i % 26);

data[i] = m;

}

// Here, we will explicitly create three nodes. The first container contains

// nodes 0 and 1 from the diagram above, and the second one contains nodes

// 1 and 2. This reflects the channel connectivity, and will be used to

// install the network interfaces and connect them with a channel.

NodeContainer n0n1;

n0n1.Create (2);

NodeContainer n1n2;

n1n2.Add (n0n1.Get (1));

n1n2.Create (1);

// We create the channels first without any IP addressing information

// First make and configure the helper, so that it will put the appropriate

// attributes on the network interfaces and channels we are about to install.

PointToPointHelper p2p;

p2p.SetDeviceAttribute ("DataRate", DataRateValue (DataRate (10000000)));

p2p.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (10)));

// And then install devices and channels connecting our topology.

NetDeviceContainer dev0 = p2p.Install (n0n1);

NetDeviceContainer dev1 = p2p.Install (n1n2);

// Now add ip/tcp stack to all nodes.

InternetStackHelper internet;

internet.InstallAll ();

// Later, we add IP addresses.

Ipv4AddressHelper ipv4;

ipv4.SetBase ("10.1.3.0", "255.255.255.0");

ipv4.Assign (dev0);

ipv4.SetBase ("10.1.2.0", "255.255.255.0");

Ipv4InterfaceContainer ipInterfs = ipv4.Assign (dev1);

// and setup ip routing tables to get total ip-level connectivity.

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

///////////////////////////////////////////////////////////////////////////

// Simulation 1

//

// Send 2000000 bytes over a connection to server port 50000 at time 0

// Should observe SYN exchange, a lot of data segments and ACKS, and FIN

// exchange. FIN exchange isn't quite compliant with TCP spec (see release

// notes for more info)

//

///////////////////////////////////////////////////////////////////////////

uint16\_t servPort = 50000;

// Create a packet sink to receive these packets on n2...

PacketSinkHelper sink ("ns3::TcpSocketFactory",

InetSocketAddress (Ipv4Address::GetAny (), servPort));

ApplicationContainer apps = sink.Install (n1n2.Get (1));

apps.Start (Seconds (0.0));

apps.Stop (Seconds (3.0));

// Create a source to send packets from n0. Instead of a full Application

// and the helper APIs you might see in other example files, this example

// will use sockets directly and register some socket callbacks as a sending

// "Application".

// Create and bind the socket...

Ptr<Socket> localSocket =

Socket::CreateSocket (n0n1.Get (0), TcpSocketFactory::GetTypeId ());

localSocket->Bind ();

// Trace changes to the congestion window

Config::ConnectWithoutContext

("/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow",

MakeCallback (&CwndTracer));

// ...and schedule the sending "Application"; This is similar to what an

// ns3::Application subclass would do internally.

Simulator::ScheduleNow (&StartFlow, localSocket,

ipInterfs.GetAddress (1), servPort);

// One can toggle the comment for the following line on or off to see the

// effects of finite send buffer modelling. One can also change the size of

// said buffer.

//localSocket->SetAttribute("SndBufSize", UintegerValue(4096));

//Ask for ASCII and pcap traces of network traffic

AsciiTraceHelper ascii;

p2p.EnableAsciiAll (ascii.CreateFileStream ("tcp-large-transfer.tr"));

p2p.EnablePcapAll ("tcp-large-transfer");

// Create the animation object and configure for specified output

AnimationInterface anim ("tcpfile.xml");

// Finally, set up the simulator to run. The 1000 second hard limit is a

// failsafe in case some change above causes the simulation to never end

Simulator::Stop (Seconds (1000));

Simulator::Run ();

Simulator::Destroy ();

}

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//begin implementation of sending "Application"

void StartFlow (Ptr<Socket> localSocket,

Ipv4Address servAddress,

uint16\_t servPort)

{

NS\_LOG\_LOGIC ("Starting flow at time " << Simulator::Now ().GetSeconds ());

localSocket->Connect (InetSocketAddress (servAddress, servPort)); //connect

// tell the tcp implementation to call WriteUntilBufferFull again

// if we blocked and new tx buffer space becomes available

localSocket->SetSendCallback (MakeCallback (&WriteUntilBufferFull));

WriteUntilBufferFull (localSocket, localSocket->GetTxAvailable ());

}

void WriteUntilBufferFull (Ptr<Socket> localSocket, uint32\_t txSpace)

{

while (currentTxBytes < totalTxBytes && localSocket->GetTxAvailable () > 0)

{

uint32\_t left = totalTxBytes - currentTxBytes;

uint32\_t dataOffset = currentTxBytes % WriteSize;

uint32\_t toWrite = WriteSize - dataOffset;

toWrite = std::min (toWrite, left);

toWrite = std::min (toWrite, localSocket->GetTxAvailable ());

int amountSent = localSocket->Send (&data[dataOffset], toWrite, 0);

if(amountSent < 0)

{

// we will be called again when new tx space becomes available.

return;

}

currentTxBytes += amountSent;

}

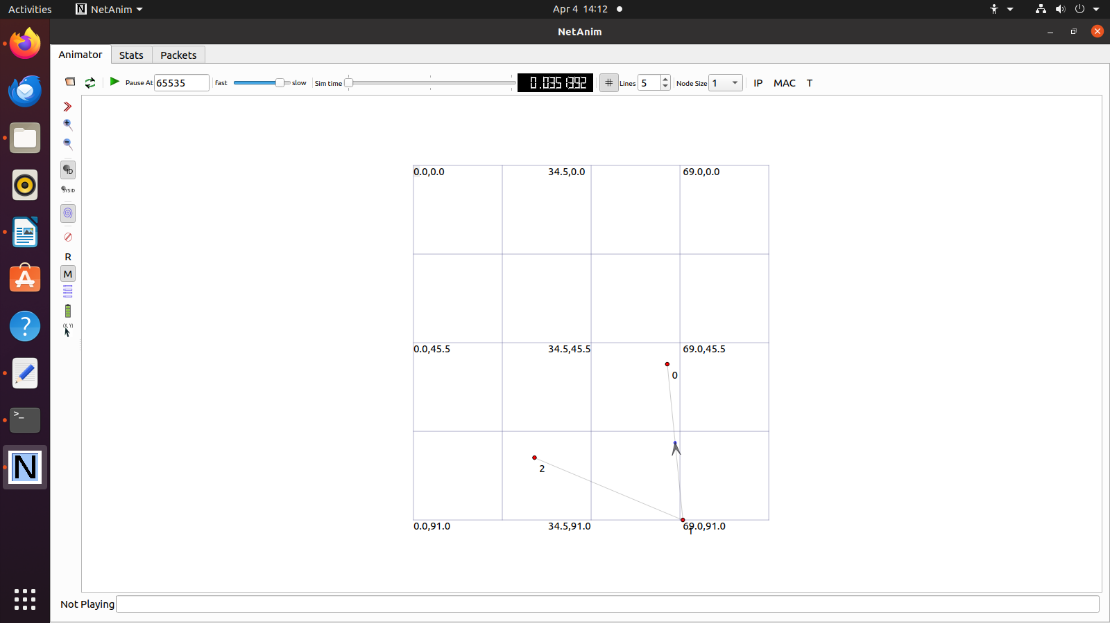
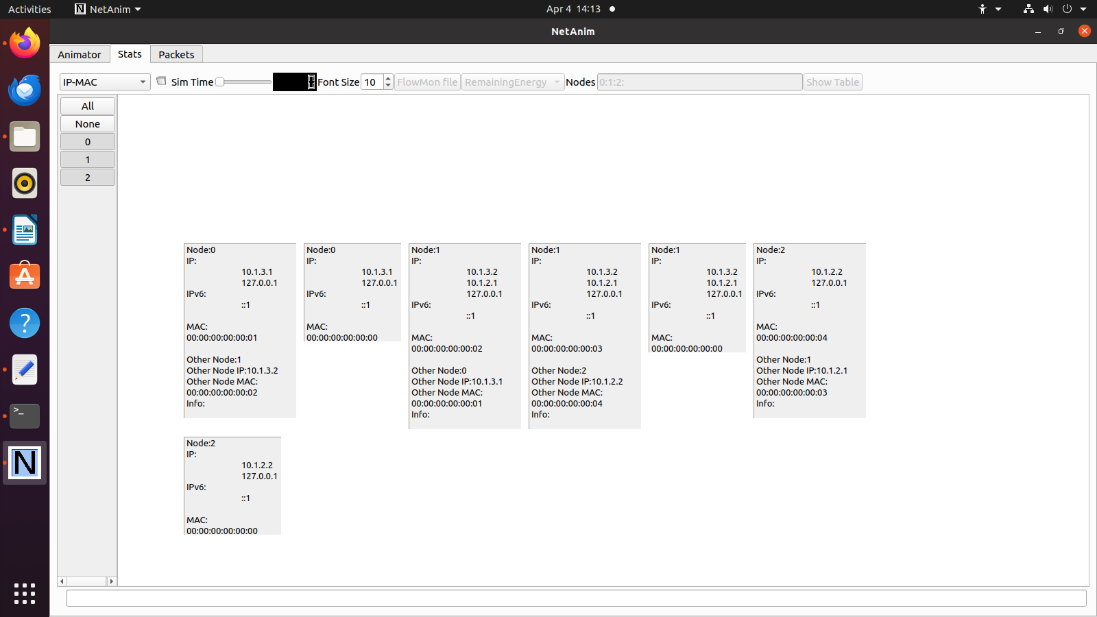
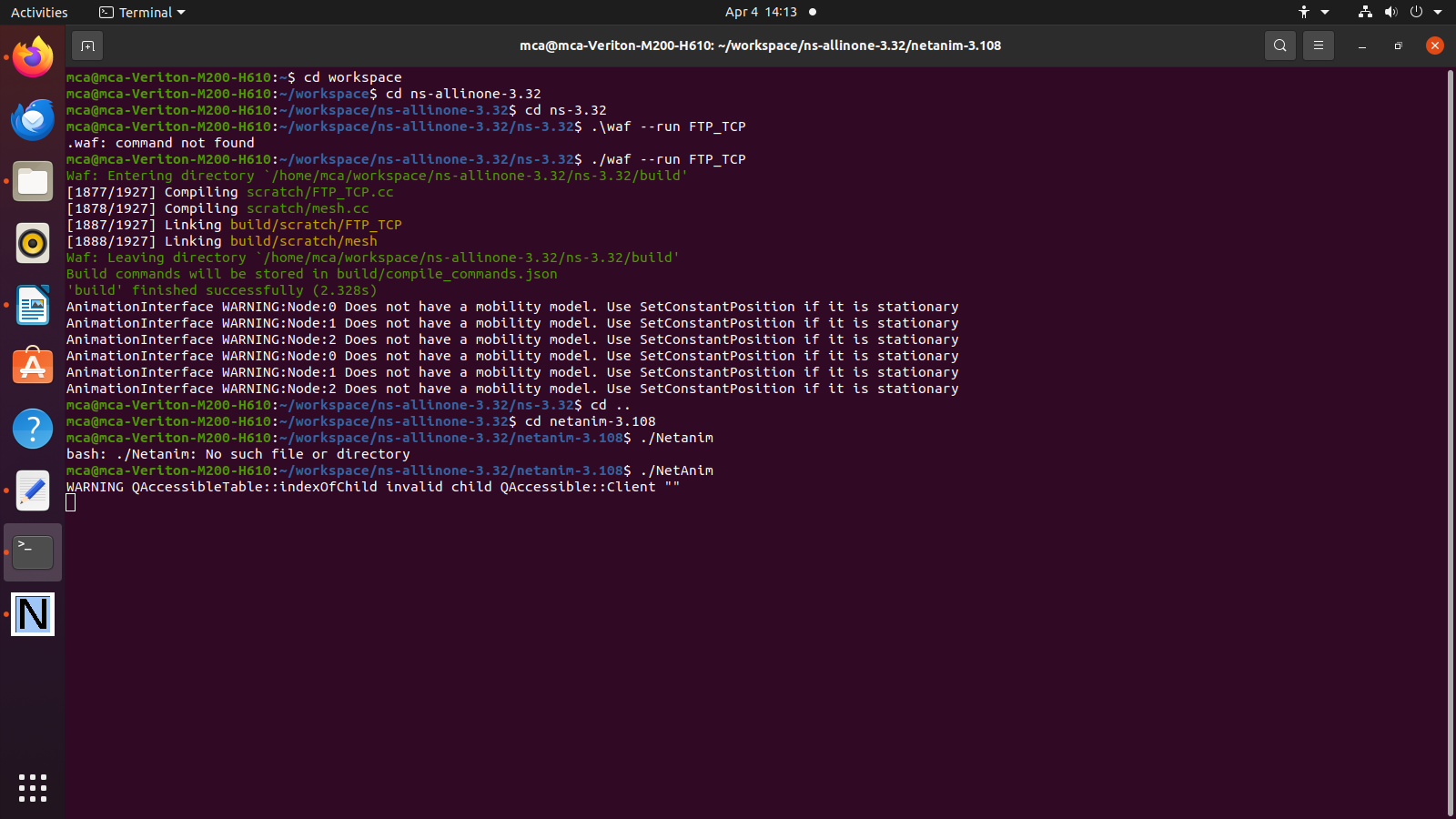
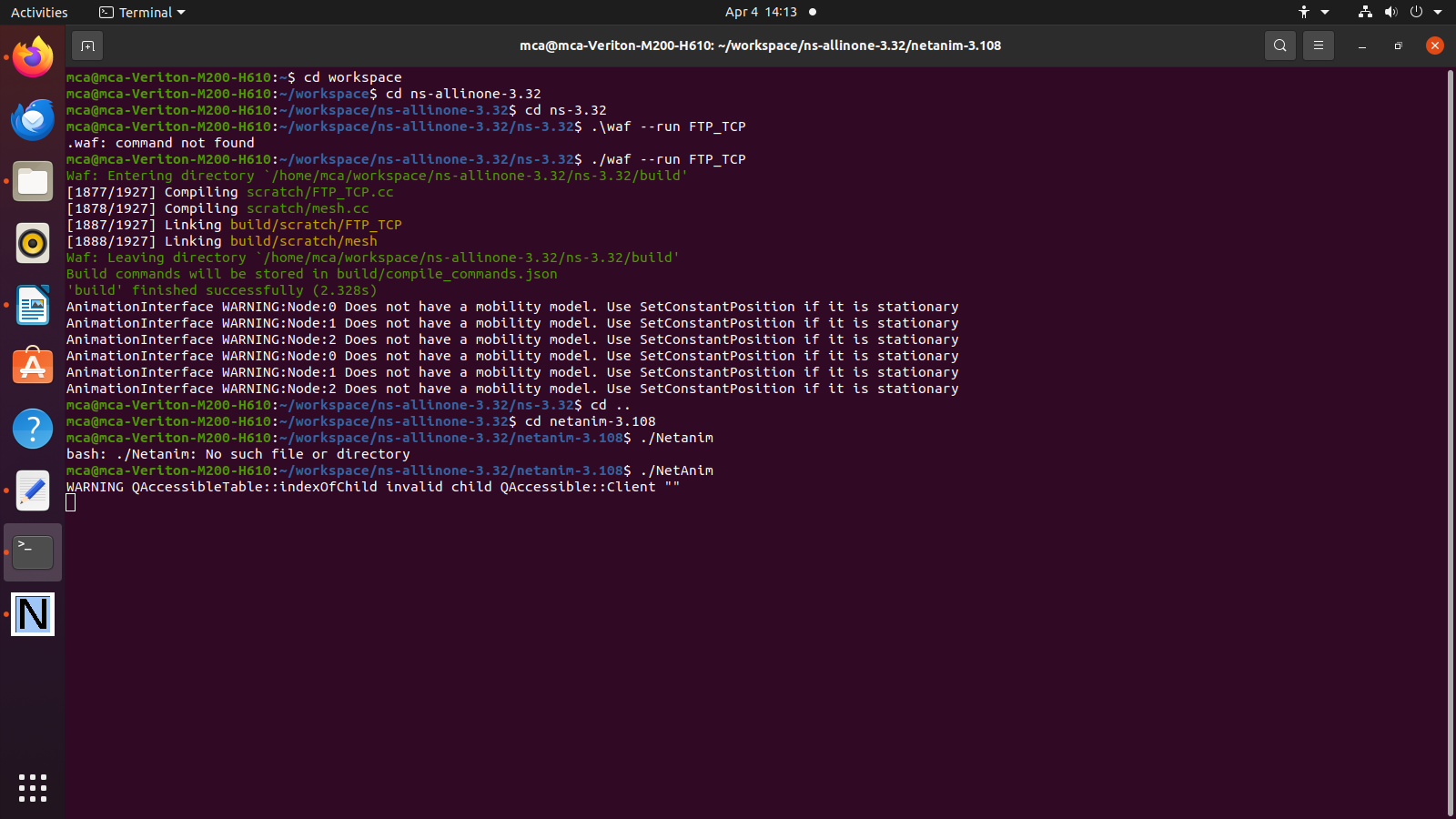
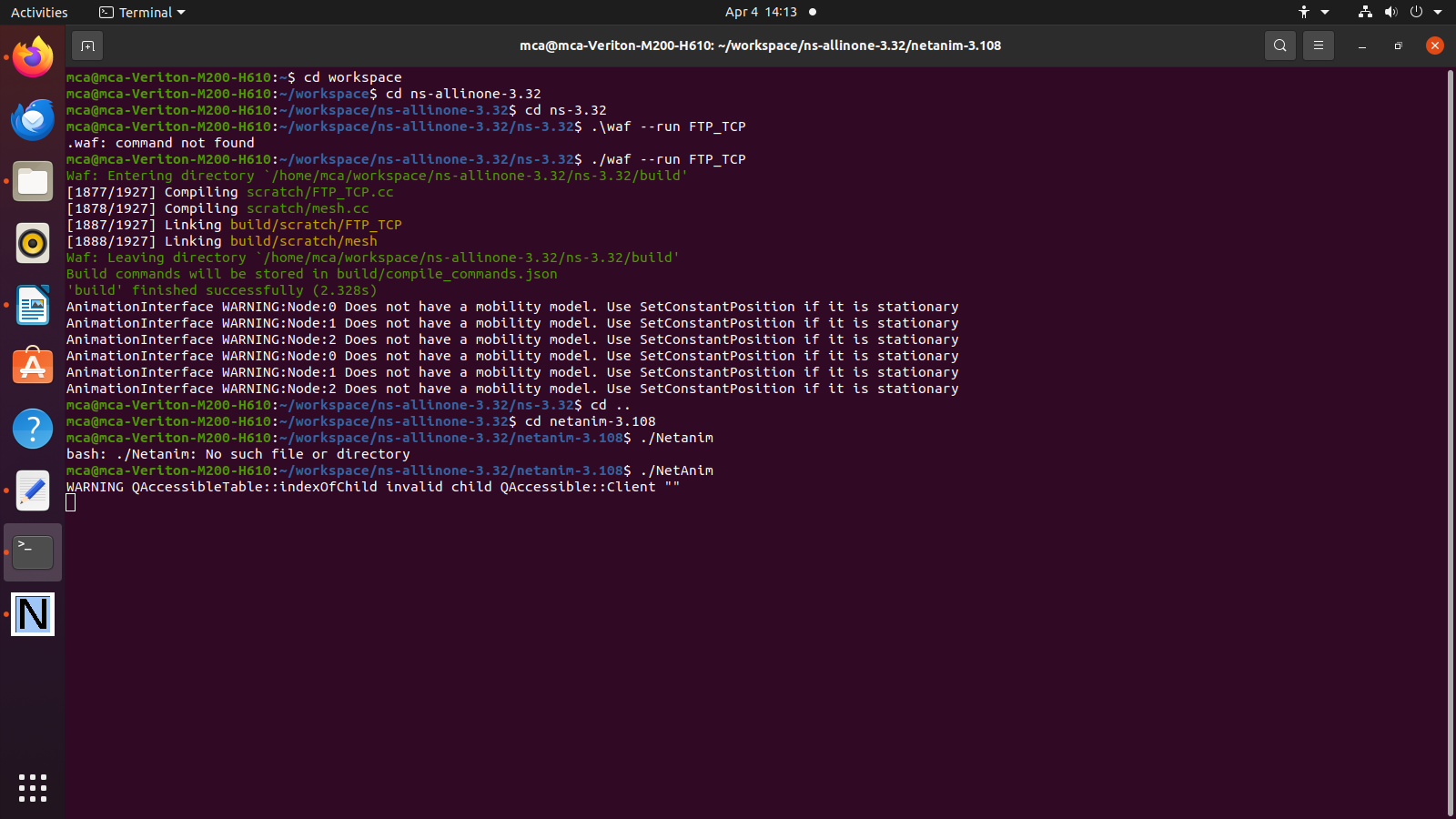
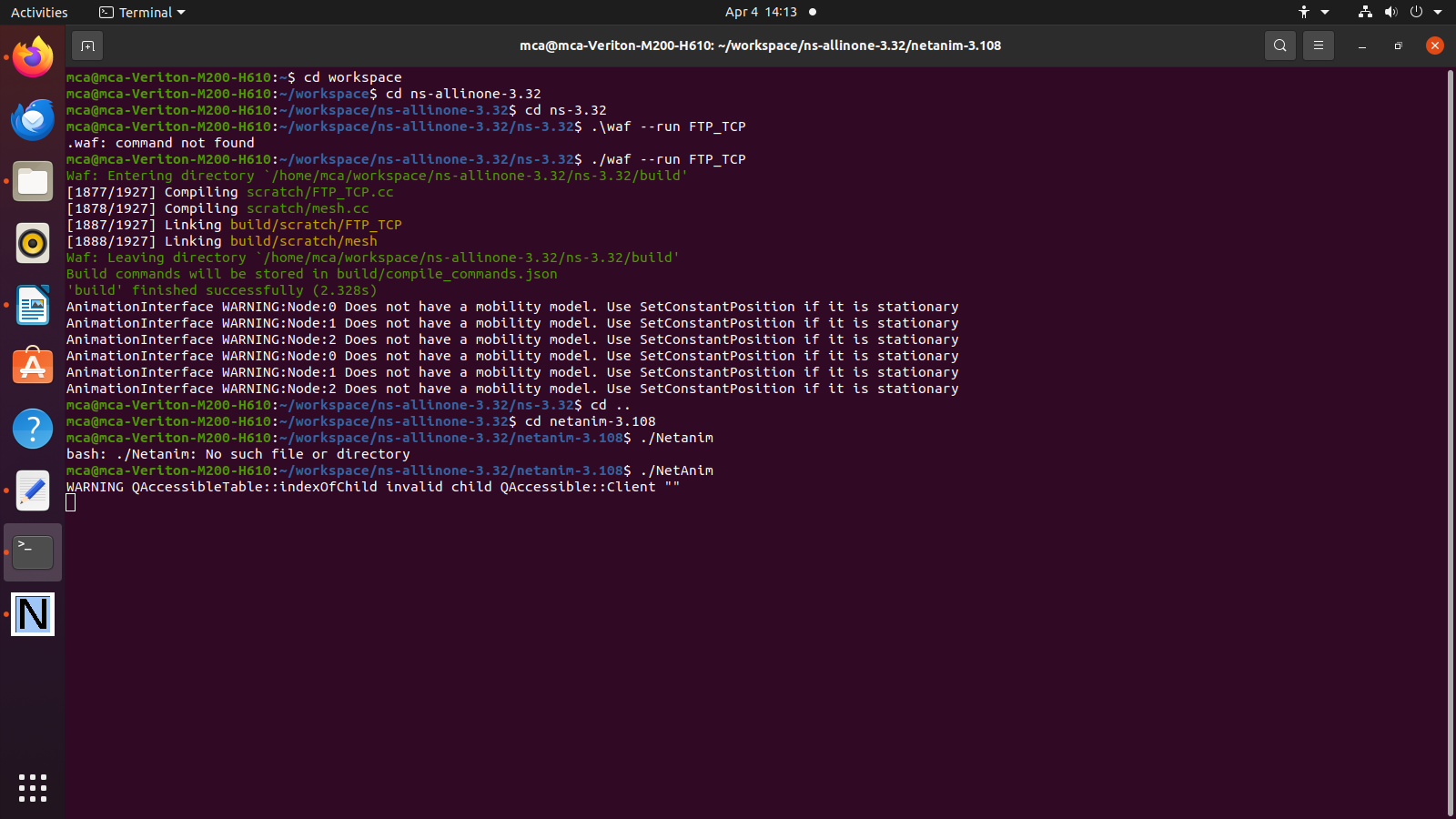
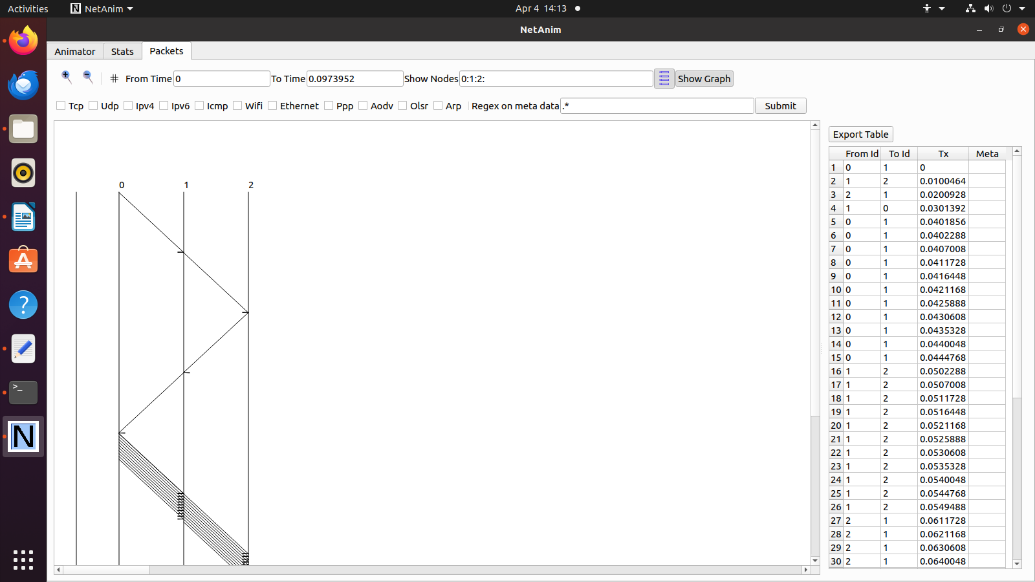
if (currentTxBytes >= totalTxBytes)

{

localSocket->Close ();

}

}

**Output: -**

**Conclusion: -**

Successfully implemented of FTP using TCP.