```cpp
#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/netanim-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 ("50Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("5ms"));

  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 (32);

  ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));

  UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 32);
  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));

  AnimationInterface anim("first.xml");
  anim.SetConstantPosition(nodes.Get(0), 20.0, 20.0);
  anim.SetConstantPosition(nodes.Get(1), 60.0, 60.0);

  AsciiTraceHelper ascii;
  pointToPoint.EnableAsciiAll(ascii.CreateFileStream("first.tr"));

  pointToPoint.EnablePcapAll("first_pcap");

  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

Visualizing Networking

NS3 ASSIGNMENT

Atharva Deshpande | S20190010043 | CCN Sec-A | 06-05-2021

Installing Network Simulator & NetAnim

The installation of NS-3 went easy especially after reading the manual provided as a part of lab tutorial. As it was suggested, I didn't use command line argument to install NetAnim rather used the commands given in the tutorial document.

Downloading the required packages prior is extremely important and resolving the errors coming through them is even more critical. Rest if all the dependencies are met, the applications work well and function smoothly.

Importing the required modules & modifying the source code

In the cover page of this document, the image shows the source code that was executed for the entire simulation of the client-server communication. This source code was derived from the ns3 application's tutorial folder in the examples provided. It had a simple UDP communication between a client and a server. The module that was specially included in the code was the NetAnim module which allowed the program to derive certain attributes that enabled the simulation of network inside the NetAnim application. The generated xml file was used for simulation of the network.

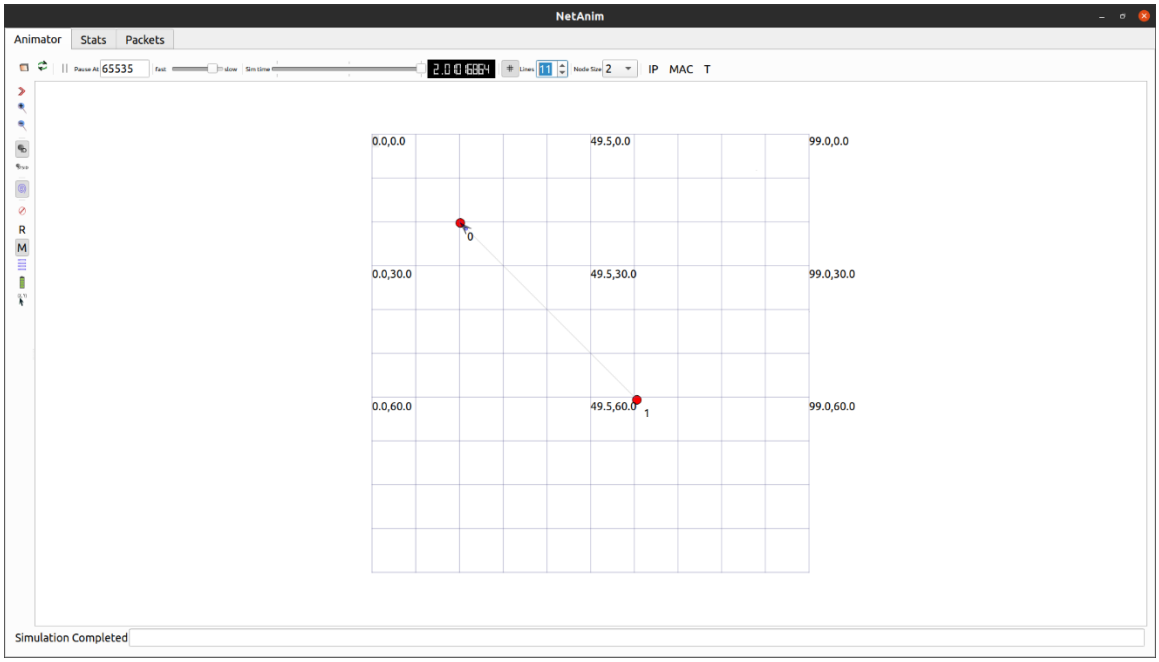
Data rate and delay were specified as instructed in the assignment. The server port 32 was mentioned too and it worked well as per the output received after executing the code. Server was supposed to start at 1 second after the simulation and the client was supposed to send the data exactly at 2 seconds of the simulation time. The coordinates were specified and fixed as per the values given in the assignment.

NetAnim output

After modifying the source code as per the requirements, it was executed in the scratch folder. Here's what was received from the terminal:

```
atharva@atharva-Lenovo:~/ns-allinone-3.31/ns-3.31$ ./waf --run scratch/first
Waf: Entering directory `/home/atharva/ns-allinone-3.31/ns-3.31/build'
[2803/2873] Compiling scratch/first.cc
[2833/2873] Linking build/scratch/first
Waf: Leaving directory `/home/atharva/ns-allinone-3.31/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (3.610s)
AnimationInterface WARNING:Node:0 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:0 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
At time 2s client sent 1024 bytes to 10.1.1.2 port 32
At time 2.00517s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00517s server sent 1024 bytes to 10.1.1.1 port 49153
At time 2.01034s client received 1024 bytes from 10.1.1.2 port 32
atharva@atharva-Lenovo:~/ns-allinone-3.31/ns-3.31$ cd ..
atharva@atharva-Lenovo:~/ns-allinone-3.31$ ls
bake  build.py  constants.py  netanim-3.108  ns-3.31  pybindgen-0.21.0  __pycache__  README  util.py
atharva@atharva-Lenovo:~/ns-allinone-3.31$ cd netanim-3.108/
atharva@atharva-Lenovo:~/ns-allinone-3.31/netanim-3.108$ ./NetAnim
```

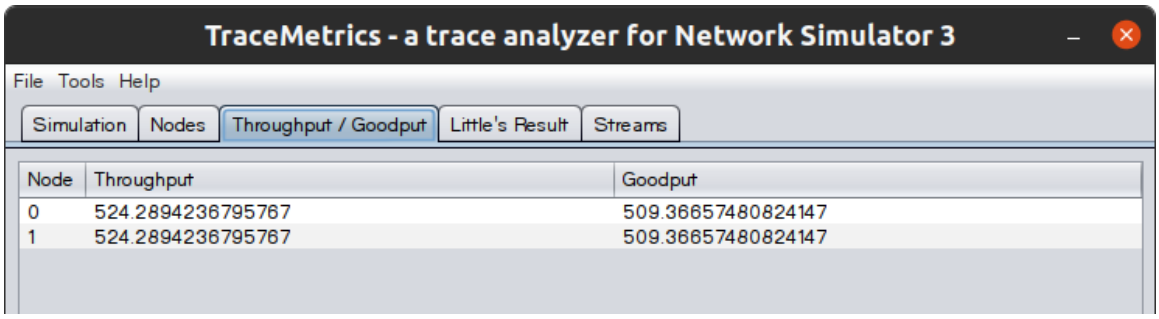
As instructed in the tutorial class, the warning was ignored and the output received suggested that the back-and-forth communication between the client and server was successful. As one can see the last line activated the NetAnim application and after getting through it, the appropriate xml file was selected and the packets were visualized.



As it is evident from the figure above that both the client and server were located correctly [I grid represents ten units so (20,20) and (60,60) located perfectly]. After running the program on the xml file, packets were exchanged between the two and the simulation ended in 2.010 seconds. The data that was shared was encapsulated in an IPv4 packet and since the size of data was less than 1500 bytes, no fragmentation occurred.

Throughput & Goodput at a glance

In order to extract the information regarding the throughput and goodput of the link, a trace file was first generated using Ascii trace helper and then using the third-party application 'tracemetrics' which required java to be installed, the throughput and goodput of the link was captured and displayed as below.



Let's analyze how these values were received.

As per the given parameters,

Data sent and received by both client and server = 1024 bytes

But, as the packets in which the data was encapsulated was IPv4, we have minimum 20 bytes of header additional. Not to forget the source and destination that constituted a total of 8 bytes of the packets (32 bits or 4 bytes each). Later we'll observe in Wireshark that that each frame consisted of 1054 bytes.

Now, the transmission delay, d_{trans} = (1054 x 8) b / 50 Mbps = 0.16864 msec

And we defined the propagation delay, d_{prop} = 5 msec

So, the total delay = d_{prop} + d_{trans} = 5.16864 msec

Adding the simulation time, we can say it takes 2 + 0.0051684 sec = 2.00516864 seconds to transmit the frame from client to server. In the complete round trip, it takes 2. 00516864 + 00516864 = 2.01034 seconds to send and receive the data back.

Hence, Throughput of the link = 1054 bytes / 2.01034 sec = **524.2894367 bytes/seconds**

And Goodput of the link = 1024 bytes / 2.01034 sec = **509.36657480 bytes/seconds** as it considers the actual data (payload) that is transmitted to the communication link.

Exploring Pcap with the help of Wireshark and tcpdump

By enabling Pcap we could generate a Pcap file that had information about the packet traversal between the two nodes. So, the code generated two Pcap files, one for the client and one for the server. Basically, each node will have the detailed information about the arrival and departures of the packets on the communication line.

```
atharva@atharva-Lenovo:~/ns-allinone-3.31/ns-3.31$ tcpdump -nn -tt -r first_pcap-0-0.pcap
reading from file first_pcap-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.1.2.32: UDP, length 1024
2.010337 IP 10.1.1.2.32 > 10.1.1.1.49153: UDP, length 1024
atharva@atharva-Lenovo:~/ns-allinone-3.31/ns-3.31$ tcpdump -nn -tt -r first_pcap-1-0.pcap
reading from file first_pcap-1-0.pcap, link-type PPP (PPP)
2.005168 IP 10.1.1.1.49153 > 10.1.1.2.32: UDP, length 1024
2.005168 IP 10.1.1.2.32 > 10.1.1.1.49153: UDP, length 1024
atharva@atharva-Lenovo:~/ns-allinone-3.31/ns-3.31$
```

As we can see in the picture above, after using tcpdump as a command line utility and using exactly the flags prescribed in the tutorial class over the obtained Pcap files, we get:

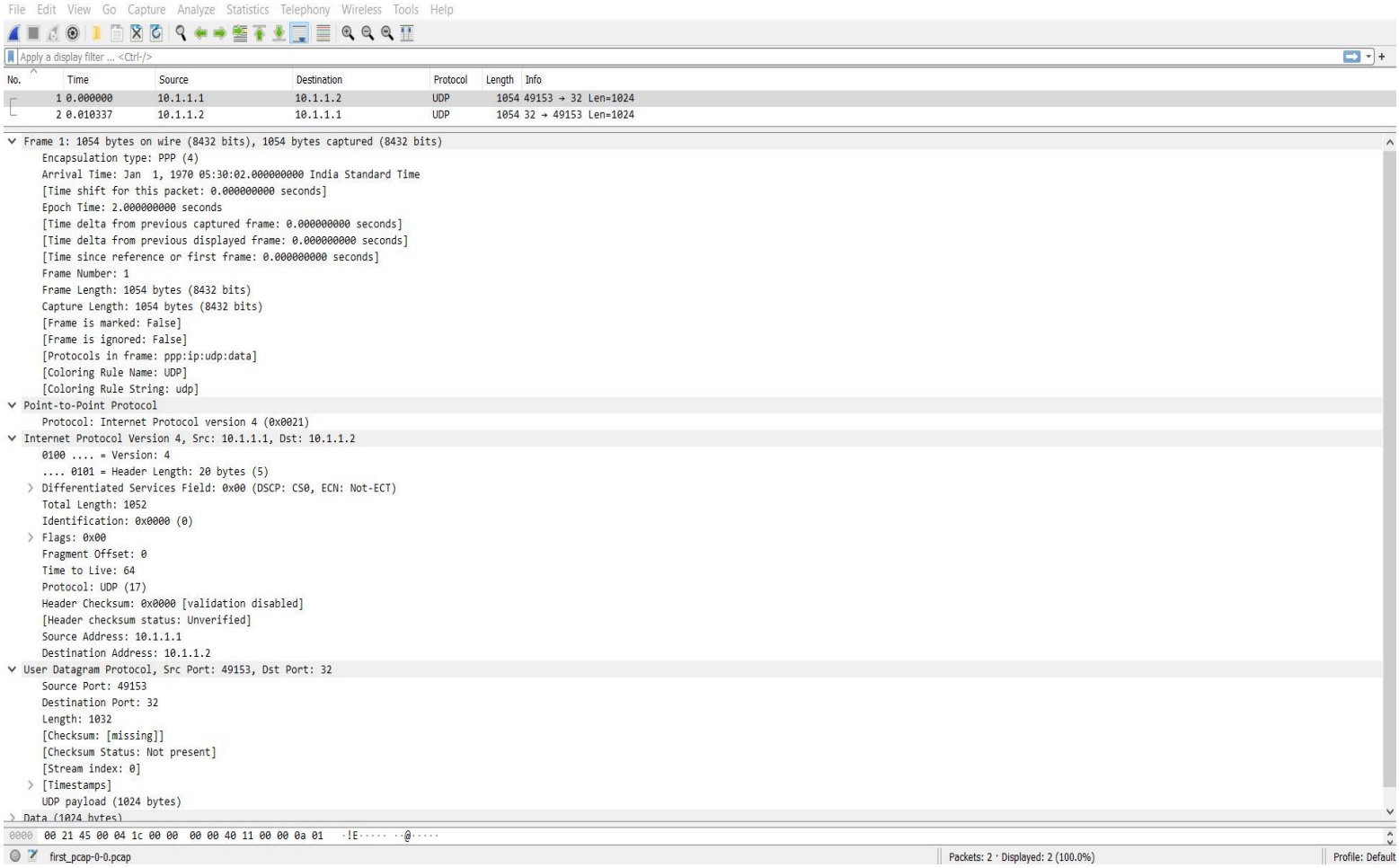
- Timestamp of the activities
- Source and destination IPv4 addresses (as IP governs in the Network layer)
- Transport Protocol Used (which is UDP for both)
- Link Layer Protocol (which is in this case PPP i.e., Point to Point Protocol)
- Payload Length (1024 bytes here)

In the Application layer, we already know that the server port is 32 from the executed code and the output displayed in the first pic.

From the picture above, we can deduce from the output of 'first_pcap-0-0.pcap' that the client (10.1.1.1, 49153) sends data of length 1024 to port 32 of the server through UDP protocol at 2 seconds from the beginning and receives a response packet of payload length 1024 from the server (10.1.1.2, 32) through UDP at 2.010337 seconds (explained in throughput section how!).

From the same picture, we can deduce from the output of 'first_pcap-1-0.pcap' that the server (10.1.1.2, 32) receives data of length 1024 from the client (10.1.1.2, 49153) through UDP protocol at 2.005168 seconds from the beginning and immediately sends a response packet of payload length 1024 to the client through UDP.

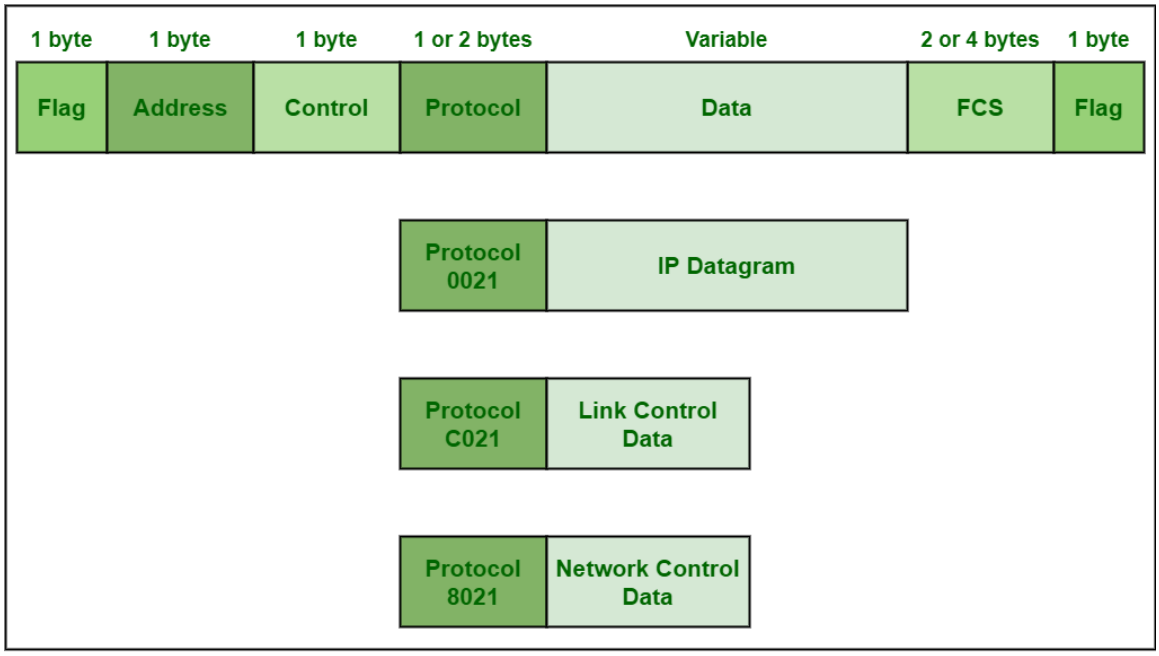
Now let's come back to the mystery of how exactly the total amount of data transmitted in the communication link at once, was 1054 bytes. Let's analyze the frame 1 in file 'first_pcap-0-0.pcap'



After running Wireshark and selecting the aforementioned file, on clicking Serial no.1 we can analyze the frame 1. The very first line says 1054 bytes captured (8432 bits).

This can be understood by the fact that we were sending 1024 bytes of data in the IPv4 packet. Then in the 3rd section starting with "Internet Protocol Version 4…", we can notice that the total length including the 20 bytes header of the IPv4 packet and 8 bytes of UDP segment header (as we can see total length of UDP segment is 1032 at the end) is 1052 bytes (1024+20+8=1052). Here comes the trickiest part. If the total length of IPv4 packet was of 1052 bytes, how were 1054 bytes captured and not 1052?

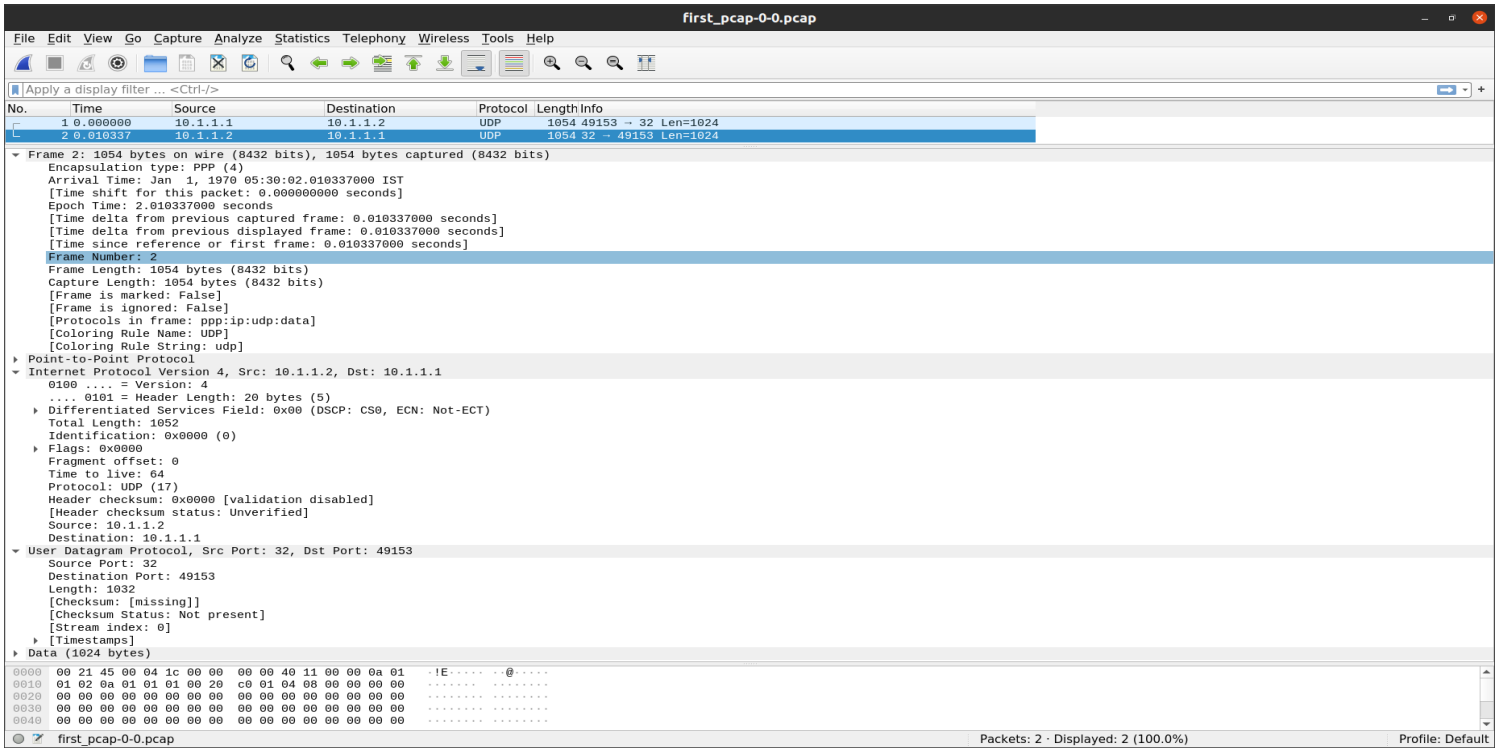
The answer lies in the process of encapsulating the 1052 bytes sized IPv4 packet inside a link layer frame that typically uses PPP. The structure of PPP frame starts with a 2 bytes protocol ID and then trails the IP datagram. Reference is given below:



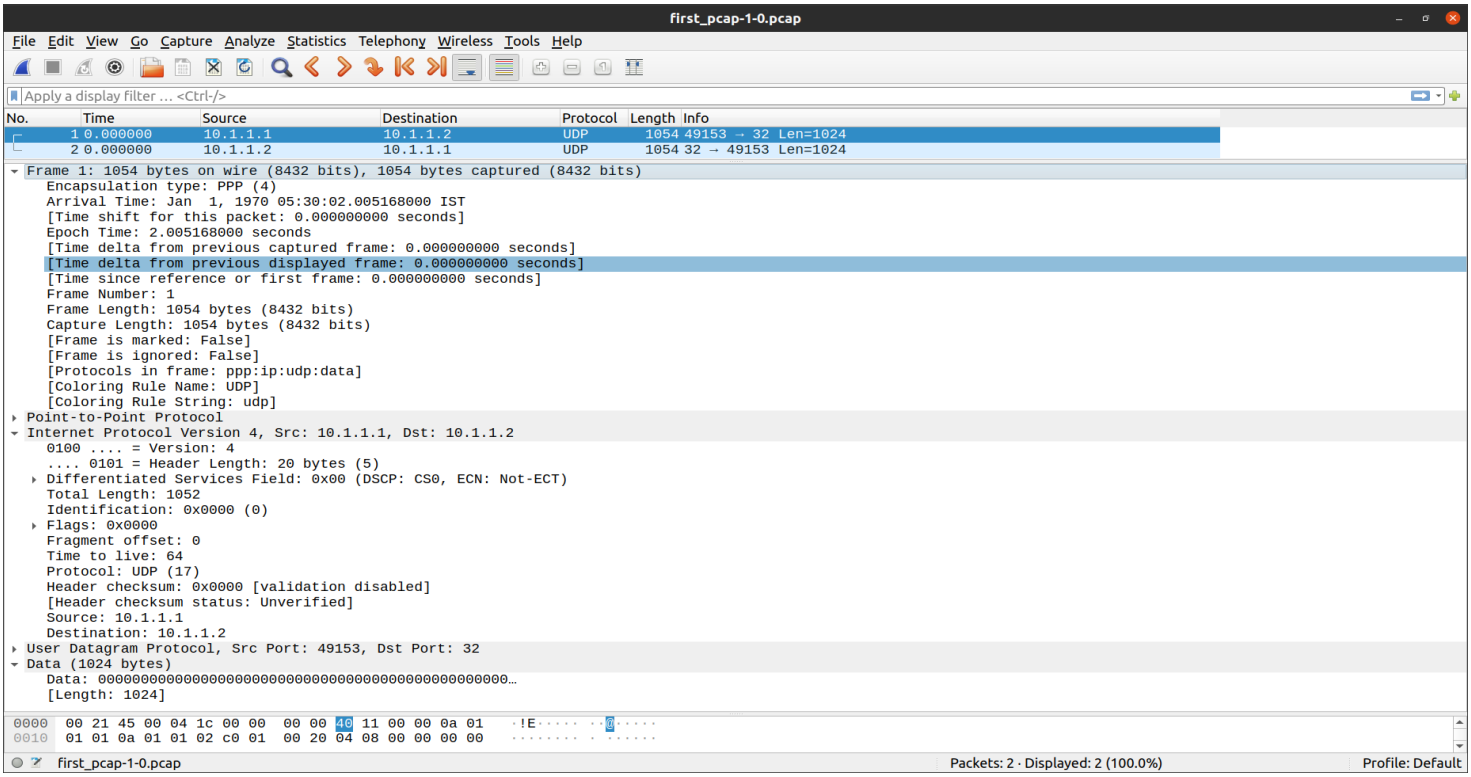
PPP Frame Format

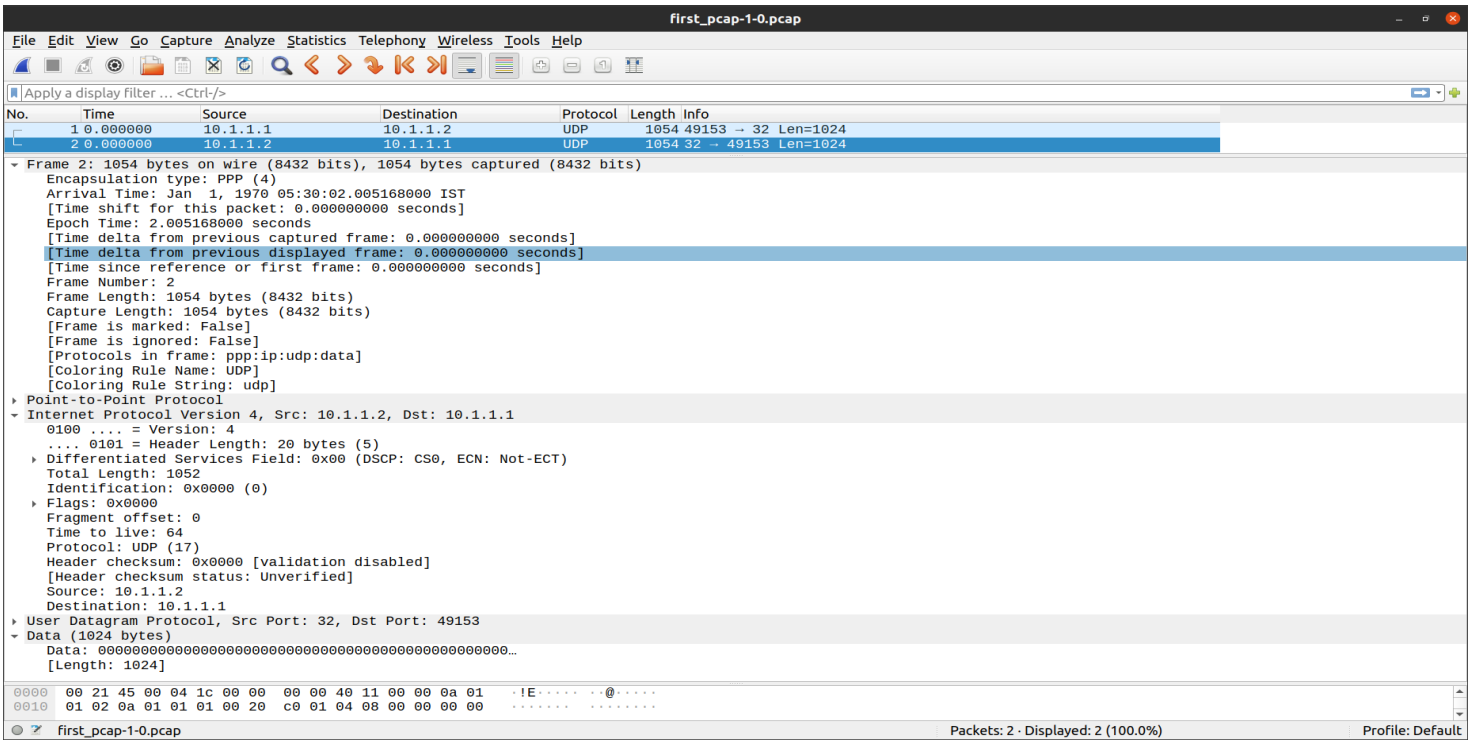
Like we can see in the Point-to-Point Protocol section of the Wireshark screenshot added that the protocol used in the encapsulated packet is indeed Internet Protocol version 4 (0x0021). Comparing this information with the picture shared above that shows the PPP frame we can conclude that yes, a 2 bytes Protocol ID was used which pumped the total data length to 1054 bytes from 1052 bytes. A PPP frame does not require more than 2 bytes as it is very simple and all other information like flags, source and destination address is present inside the encapsulated packet. [Courtesy for the image above: GeekforGeeks]

Similarly, we can watch the frame 2 received by node 0 i.e., our client:

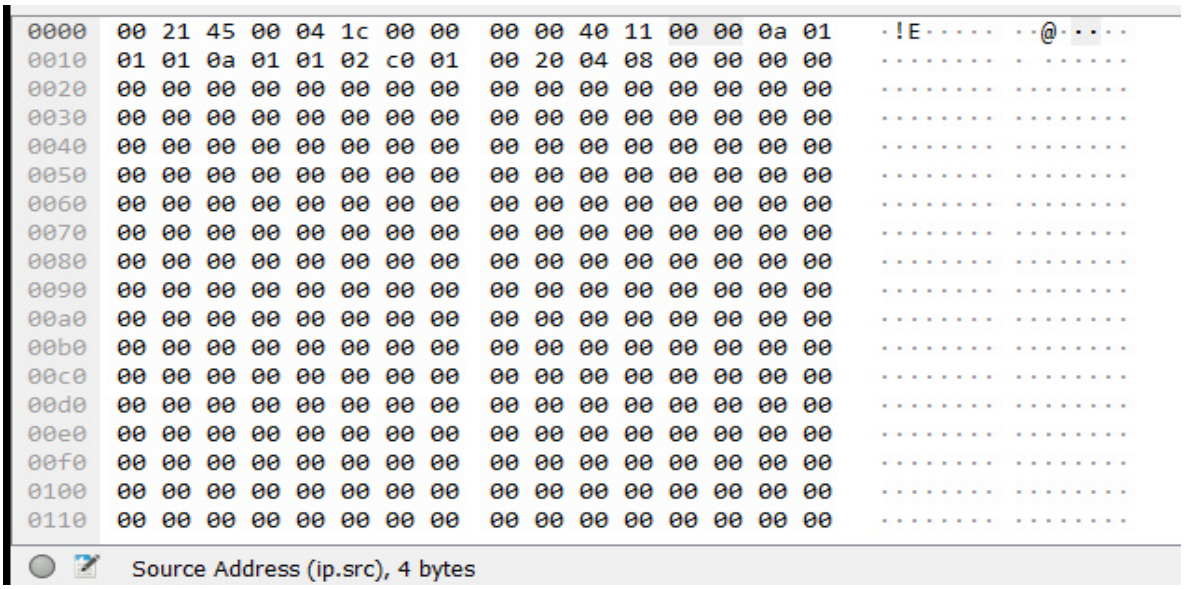


We can also see the frames 1 and 2 at node 1 (server):





Then we are now going to analyze each field in the frame 1 of server (the one it receives from the client at first):



Let's analyze the topmost lines one by one precisely using Wireshark:

00 21 45 00 04 1c 00 00 00 00 40 11 00 00 0a 01 01 01 00 0a 01 01 02: All the following revelations and analysis has been made by using Wireshark.

- Here, 00 21 stands for the Protocol ID in PPP; it indicates IP being used.
- 45 in hexadecimal is the IP header length.
- The succeeding 00 denotes Explicit Congestion Notification (ip.dsfield.ecn). In IPv4 datagram structure, we have a TOS field just after the header length. It is this field which is storing ECN and (via Wikipedia we get to know) when ECN is negotiated, an ECN-aware router may set a mark in the IP header instead of dropping a packet in order to impeding congestion.
- 04 1c together as a byte stores the total length of an IPv4 packet which in decimal is **1052**.
- Next two consecutive pair of bytes having 00 00 as the value each, are the identification tags and flag offset of the IP packet ordinally. No fragmentation happened; hence we are getting that as the output.
- While the very next byte, 40 represents the TTL, here it conveys that the value is 256 seconds.
- Then the byte next, 11, says that in the upper layer i.e., the Transport layer, UDP protocol has been used.
- Then the pair of bytes, 00 00, represents the IP header checksum.
- The following 32-bit chunk **0a 01 01 01**, keeps the IP address of the source which is 10.1.1.2 in decimal and that indeed happens to be our client's as shown in the

previous screenshots. While the next 32-bit chunk **0a 01 01 02** shows the destination address 10.1.1.2 that happens to be of our server's address.

c0 01 00 20 04 08 00 00: After the above sequence of bytes, we now enter into the sequence of bytes that holds the credentials of our UDP packet.

- The first two bytes c0 01 give us the source port i.e., 49153, which is true and the next two bytes 00 20 rightly show us the destination port number which is 32.
- The pair of bytes, 00 00 then represents the UDP header checksum.

And ultimately, we have the next 1024 bytes as the data inside the UDP packet. Likewise, same analysis can be done for frame 2 of the server and both the frames of client.

Summary

To summarize, we fetched the required files from tutorial folder in ns3 examples, imported necessary modules, added coordinates and some more specifications. Then we used NetAnim to visualize the simulation of network on a grid. Found out the correct values of Throughput as well as Goodput and also proved & analyzed how we got the same results as in tracemetrics. We then explored Pcap file in detail with the help of Wireshark and tcpdump.