NS3:

C++ program for simulating a network using the Network Simulator 3 (ns-3) framework. It includes various headers for different ns-3 modules and functionality, as well as some constants and macros that are used throughout the program.

There is also a template class called **SimpleHistogram** that is used for recording the frequency of different values, and a namespace containing various classes that are derived from NS3 classes.

It is simulation of a client-server communication over a wireless network using the ns-3 network simulator. It sets up a simulation environment with two nodes (representing the client and server), a wireless network connecting them, and a TCP/IP stack on each node.

The simulation is defined by a number of parameters, such as the client's data rate, the number and size of packets sent by the client, the start and stop times for the client and server, and the length of the simulation.

The code begins by including a number of ns-3 modules, which provide various functions for simulating networks and applications. Next, a number of constants are defined, such as CLIENT_MACHINE_NUM and SERVER_MACHINE_NUM, which represent the indices of the client and server nodes in the simulation. There are also a number of constants related to the configuration of the simulation, such as the client's data rate and the number of packets the client will send.

There is also a template class called **SimpleHistogram**, which is used to count occurrences of a particular value. This class has a std::map member variable called m_map, which stores the counts of values, and a number of methods for adding values to the histogram and printing it out.

The **namespace** ns3 contains a class called PacketCreationTimeTag, which is a subclass of the ns3::Tag class. This class is used to tag packets with the time they were created, so that the delay between when a packet is created and when it is received can be measured. The PacketCreationTimeTag class has a member variable called m_simpleValue, which stores the time a packet was created, and a number of methods for setting and getting this value, as well as for serializing and deserializing the tag.

TcpEchoServer::HandleRead is a member function of the TcpEchoServer class that is called when there is data to be read from the socket. It reads the data from the socket and then sends it back to the sender, essentially echoing the data. This is commonly known as an "echo server" because it simply echoes back any data that it receives. The HandleRead function is using the Send method of the Socket class to send the data back to the sender.

This function reads data from the socket using the RecvFrom method of the Socket class, and then sends the data back to the sender using the Send method. It continues to do this until there are no more data to be read from the socket.

General overview of how packets might flow in a simple network simulation using ns-3:

1. A packet is created at the source node, which is typically a client application running on a simulated host machine. The packet includes a header and a payload, which contains the data that the client wants to send to the destination node.
2. The packet is passed down through the different layers of the network stack on the source node. This includes the transport layer (e.g., TCP or UDP), the internet layer (e.g., IPv4 or IPv6), and the link layer (e.g., Ethernet or WiFi). Each layer adds its own header to the packet to provide additional information about the packet, such as the source and destination addresses, the protocol being used, and the packet size.
3. The packet is transmitted over the simulated network link, which could be a point-to-point link, a CSMA link, or a WiFi link, depending on the configuration of the simulation.
4. The packet is received by the destination node, which is typically a server application running on a simulated host machine. The packet is passed up through the different layers of the network stack on the destination node, with each layer stripping off its own header as the packet is passed up.
5. The server application processes the packet, possibly echoing it back to the client if it is an echo server, or performing some other action with the data contained in the packet.
6. The server sends a response packet back to the client, following the same process in reverse.

In this code, packets are being sent and received over a network using the Network Simulator 3 (NS3) library. The code includes a number of NS3 modules, such as core-module, point-to-point-module, csma-module, internet-module, and applications-module, which provide various networking functionalities such as creating nodes, setting up point-to-point links, installing the internet stack on nodes, and creating application layer protocols.

The code also includes several standard C++ modules, such as map, as well as modules for WiFi networking and mobility, such as yans-wifi-helper, mobility-helper, ipv4-address-helper, and yans-wifi-channel.

There are several compile-time constants defined in the code, such as CLIENT_MACHINE_NUM, SERVER_MACHINE_NUM, CLIENT_DATA_RATE, CLIENT_PACKETS_COUNT, and CLIENT_PACKET_SIZE, which are used to specify various network parameters.

The code also defines a class called SimpleHistogram, which is used to record the frequency of different values.

Finally, the code includes a class called PacketCreationTimeTag, which is used to store meta information (a time stamp) in the header of a TCP packet. This time stamp is used to calculate the delay of the packet as it is transmitted over the network.

Overall, the flow of packets in this code can be described as follows:

1. Nodes are created and a point-to-point link is installed between them.
2. The internet stack is installed on the nodes.
3. IP addresses are assigned to the nodes.
4. A TCP echo server application is installed on the server node, and a TCP echo client application is installed on the client node.
5. The client node sends packets to the server node using the TCP echo client application.
6. The server node receives the packets and echoes them back to the client node using the TCP echo server application.
7. A packet sink application is installed on the server node to receive the echoed packets.
8. The simulation is run and statistics, such as the total number of packets received and the packet loss rate, are printed out.

simulation of a network using the NS3 (Network Simulator 3) library. It includes various modules and headers from NS3, as well as some standard C++ headers and a custom template class called SimpleHistogram.

The code also defines several constants, such as CLIENT_MACHINE_NUM and SERVER_MACHINE_NUM, which are used as reference numbers for client and server objects in the code. There are also several constants related to logging, packet creation and transmission, and simulation parameters.

The code also includes a class called PacketCreationTimeTag, which is a type of "tag" that can be added to the header of a TCP packet. The tag stores the time at which the packet was created on the sender side, and the receiver can use this information to calculate the round-trip time for the packet.

Finally, the code includes several NS3 classes, such as TcpEchoServer and TcpEchoClient, which are used to simulate a server and client in the network. These classes define various methods, such as HandleRead and SendData, which are used to process and transmit data in the simulation. Overall, it seems that the purpose of this code is to simulate a network with a server and client, transmit data between them, and collect various statistics about the transmission.

This code is a simulation program written in the NS3 (Network Simulator 3) framework, which is a tool for simulating the behavior of networks and communication systems. The code is written in C++ and includes a number of header files that provide various functions and classes for use in the simulation.

The first group of header files includes "ns3/core-module.h", "ns3/point-to-point-module.h", "ns3/csma-module.h", "ns3/internet-module.h", and "ns3/applications-module.h". These header files provide basic functionality for simulating networks and communication systems in NS3. The "core-module" provides core classes and functions for defining and manipulating nodes, channels, and other basic elements of a network. The "point-to-point-module" and "csma-module" provide classes and functions for simulating point-to-point and CSMA (Carrier Sense Multiple Access) networks, respectively. The "internet-module" provides classes and functions for simulating internet protocols such as IPv4 and IPv6, and the "applications-module" provides classes and functions for simulating various types of network applications such as servers and clients.

The next group of header files includes "ns3/netanim-module.h" and "ns3/flow-monitor-module.h". These header files provide functions and classes for visualizing and monitoring the simulation. The "netanim-module" provides classes for creating animations of the simulation, and the "flow-monitor-module" provides classes for monitoring various aspects of the simulation such as flow statistics, packet size distributions, and delay statistics.

The client and server classes in this code is implementations of the NS3 Application class, which is a class for defining network applications in NS3. These classes are used to simulate the behavior of a client and a server in a network.

The client class, called TcpEchoClient, is responsible for sending packets to the server and receiving responses from the server. It has a number of member functions and variables that are used to configure and control the client's behavior. For example, the **client** has a member function called **Setup** that is used to configure the client's parameters such as the data rate, packet size, and number of packets to send. It also has a member function called **StartApplication** that is called when the client starts sending packets to the server, and a member function called **StopApplication** that is called when the client stops sending packets.

The **server** class, called TcpEchoServer, is responsible for receiving packets from the client and sending responses back to the client. It has similar member functions and variables as the client class, including a Setup function for configuring the server's parameters and StartApplication and StopApplication functions for controlling the server's behavior. The server also has a member function called HandleRead that is called when the server receives a packet from the client. This function reads the packet, modifies it as needed (for example, by adding a delay or increasing the size of the packet), and then sends it back to the client as a response.

Overall, the client and server classes work together to simulate the behavior of a client and a server in a network. The client sends packets to the server, and the server receives and responds to these

The TcpEchoClient class have the following methods:

- **Setup**: This function is used to configure the client's parameters such as the data rate, packet size, and number of packets to send.
- **StartApplication**: This function is called when the client starts sending packets to the server. It creates a socket and starts sending packets to the server.
- **StopApplication**: This function is called when the client stops sending packets. It closes the socket and stops sending packets.
- **SendPacket**: This function is used to send a packet to the server. It takes a packet size as an input and creates a packet with the specified size. The packet is then sent to the server using the socket created in the StartApplication function.
- **SetPacketSize**: This function is used to set the size of the packets that the client will send to the server.
- **SetPacketCount**: This function is used to set the number of packets that the client will send to the server.
- **SetDataRate**: This function is used to set the data rate at which the client will send packets to the server.
- **GetHistogram**: This function returns a histogram object that records the frequency of every packet size that the client has sent to the server.
- **GetPacketsSent**: This function returns the total number of packets that the client has sent to the server.
- **HandleRead**: This function is called when the server receives a packet from the client. It extracts the packet's creation time from the packet header and calculates the round trip time (RTT) by subtracting the current time from the packet's creation time. The server then sends an echo packet back to the client with the same size as the original packet. The echo packet is sent with a delay specified by the echo delay parameter.
- **GetHistogram**: This function returns a histogram object that records the frequency of every packet size that the server has received from the client.
- **GetPacketsReceived**: This function returns the total number of packets that the server has received from the client.
- **GetEchoedPacketsSent**: This function returns the total number of echo packets that the server has sent back to the client.

how does **TchoClient::ScheduleTx** work?

The TcpEchoClient class have a method called ScheduleTx, which is used to schedule the transmission of packets to the server. The purpose of this method is to send a specified number of packets to the server at a fixed rate.

Here is how the **ScheduleTx** method works:

1. It first checks if the number of packets that have been sent so far is less than the total number of packets to be sent (CLIENT_PACKETS_COUNT). If all the packets have already been sent, the method returns without doing anything.
2. It creates a new packet with the specified size (CLIENT_PACKET_SIZE).
3. It attaches a tag to the packet that contains the current time. This tag is used by the server to calculate the RTT for each packet.
4. It schedules the transmission of the packet to the server at a fixed rate specified by the CLIENT_SEND_TIME parameter. This is done using the Simulator::Schedule method.
5. It increments the counter for the number of packets sent so far.
6. It calls itself again using Simulator::Schedule, with a delay specified by the CLIENT_SEND_TIME parameter. This causes the method to be called repeatedly, sending packets to the server at a fixed rate.

The ScheduleTx method is called by the StartApplication method, which is called when the client starts sending packets to the server. This causes the client to send a specified number of packets to the server at a fixed rate until the StopApplication method is called, which stops the client from sending any more packets.

Here is a detailed explanation of the other methods in the TcpEchoClient class:

TcpEchoClient::TcpEchoClient()

- This is the constructor for the TcpEchoClient class. It initializes the instance variables for the object with default values.

TcpEchoClient::Setup()

- This method sets up the client machine node and creates an object for the TCP socket that will be used to send data to the server. It also sets up the event handler for when the socket is connected to the server.

TcpEchoClient::SendPacket()

- This method sends a packet to the server. It creates a new packet with the specified size and adds a tag to it with the current simulation time. It then sends the packet through the socket.

TcpEchoClient::ConnectionSucceeded()

- This method is called when the socket successfully connects to the server. It sets up the event handler for when the socket receives data and starts sending packets to the server.

TcpEchoClient::ConnectionFailed()

- This method is called when the socket fails to connect to the server. It logs an error message and stops the simulation.

TcpEchoClient::DataReceived()

- This method is called when the socket receives data from the server. It reads the data from the packet and logs the packet size and delay. It also updates the histogram for the packet size.

TcpEchoClient::ScheduleTx()

- This method schedules the next packet transmission. It calculates the interval between packet transmissions based on the data rate and packet size, and schedules the next transmission with this interval. It also checks if the maximum number of packets has been reached, and stops the simulation if necessary.

TcpEchoClient::StopApplication()

- This method stops the client application. It closes the socket and cancels any scheduled packet transmissions.

Execution order:

1. The client first initiates the process by calling the TcpEchoClient::StartApplication method. This method sets up the client's socket, binds it to an address and port, and starts the process of sending data to the server.
2. The client then calls the TcpEchoClient::ScheduleTx method to schedule the transmission of a packet to the server. This method creates a new packet with a specified size and sends it to the server using the client's socket.
3. When the server receives the packet, it calls the TcpEchoServer::HandleRead method to handle the incoming packet. This method extracts the data from the packet, and if the server is configured to echo the data back to the client, it creates a new packet with the same data and sends it back to the client.
4. When the client receives the echoed packet, it calls the TcpEchoClient::HandleRead method to handle the incoming packet. This method extracts the data from the packet and compares it to the data that was originally sent to the server. If the data matches, it increments a counter to keep track of the number of successful echo replies received.

5. The process repeats until the client has sent all of the packets that it was configured to send, or until the simulation time has reached the specified stop time.

The execution order of the server methods when a packet is being sent and then echoed would be as follows:

1. The StartApplication method of the TcpEchoServer class is called, which initializes the server and sets it up to listen for incoming connections.
2. The ConnectionSucceeded event is triggered when a client successfully connects to the server.
3. The ConnectionSucceeded event handler calls the Accept method of the TcpEchoServer class to accept the incoming connection and create a TcpSocket object for it.
4. The Accept method also sets up a callback function for the NewConnectionCreated event, which is triggered whenever a new connection is created.
5. The NewConnectionCreated event handler calls the StartRx method of the TcpEchoServer class, which sets up a callback function for the DataArrived event.
6. The DataArrived event is triggered whenever data is received on the socket.
7. The DataArrived event handler calls the HandleRead method of the TcpEchoServer class, which reads the data from the socket and echoes it back to the client.
8. The HandleRead method also updates the m_totalRx variable to keep track of the total amount of data received from the client.
9. The HandleRead method calls the ScheduleTx method to schedule the next transmission of data to the client.
10. The ScheduleTx method sets up a timer to trigger the SendData method at a later time.
11. The SendData method is called when the timer expires, and it sends the data to the client using the Send method of the TcpSocket class.
12. The Send method sends the data to the client and updates the m_totalTx variable to keep track of the total amount of data sent to the client.
13. The process continues until the StopApplication method is called, which stops the server and closes all connection

Here is the method execution order when a packet is sent by the client and received by the server, then echoed by the server and received by the client:

1. TcpEchoClient::StartApplication
2. TcpEchoClient::Connect
3. TcpEchoClient::SendPacket

4. TcpEchoClient::ConnectionSucceeded
5. TcpEchoClient::ConnectionFailed
6. TcpEchoServer::StartApplication
7. TcpEchoServer::ConnectionSucceeded
8. TcpEchoServer::ConnectionFailed
9. TcpEchoServer::HandleRead
10. TcpEchoServer::SendPacket
11. TcpEchoClient::HandleRead
12. TcpEchoClient::StopApplication
13. TcpEchoServer::StopApplication

1. TcpEchoClient::StartApplication: called when the client application is started
2. TcpEchoClient::SendPacket: called when the client sends a packet to the server
3. TcpEchoServer::StartApplication: called when the server application is started
4. TcpEchoServer::HandleRead: called when the server receives a packet from the client
5. TcpEchoServer::EchoPacket: called when the server echoes the received packet back to the client
6. TcpEchoClient::HandleRead: called when the client receives a packet from the server (the echoed packet)
7. TcpEchoClient::HandleSuccess: called when the client successfully receives the echoed packet
8. TcpEchoClient::ScheduleTx: called to schedule the next packet transmission from the client
9. TcpEchoServer::HandleSuccess: called when the server successfully echoes the packet back to the client

the list of method execution in the order that they would be called, along with a description of the conditions under which each method is called:

1. TcpEchoClient::StartApplication: This method is called when the client application is started, which happens at the time specified by the CLIENT_START_TIME define. This method sets up the client socket and schedules the first packet transmission.
2. TcpEchoClient::SendPacket: This method is called when it is time for the client to send a new packet. This happens at the interval specified by the CLIENT_SEND_TIME define. This method sends the packet using the client socket, and also schedules the next packet transmission.
3. TcpEchoServer::StartApplication: This method is called when the server application is started, which happens at the time specified by the

SERVER_START_TIME define. This method sets up the server socket and starts listening for incoming connections.
4. TcpEchoServer::ConnectionSucceeded: This method is called when the server socket successfully establishes a connection with the client socket.
5. TcpEchoServer::HandleRead: This method is called when the server socket receives a packet from the client. This method echoes the packet back to the client, according to the SERVER_ECHO_FACTOR and SERVER_ECHO_DELAY defines.
6. TcpEchoClient::HandleRead: This method is called when the client socket receives a packet from the server. This method checks if the packet is the expected echo of a packet that was previously sent by the client, and updates the appropriate statistics.
7. TcpEchoClient::StopApplication: This method is called when the client application is stopped, which happens at the time specified by the CLIENT_STOP_TIME define. This method closes the client socket and stops the packet transmission

complete list of method execution in the order they are called when a packet is sent by the client and received by the server, then echoed by the server and received by the client:

1. TcpEchoClient::StartApplication: This method is called when the client application is started, typically at the beginning of the simulation. It creates and schedules the first packet transmission using the ScheduleTx method.
2. TcpEchoClient::ScheduleTx: This method is called to schedule the transmission of a new packet. It sets up the packet with the appropriate size and payload, and sends it using the socket's Send method.
3. TcpEchoServer::StartApplication: This method is called when the server application is started, typically at the beginning of the simulation. It creates a socket and binds it to the server's IP address and port number.
4. TcpEchoServer::HandleAccept: This method is called when the server's socket receives a new incoming connection request from a client. It creates a new socket and accepts the connection, allowing data to be exchanged between the client and server.
5. TcpEchoServer::HandleRead: This method is called when the server's socket receives a new packet from the client. It reads the packet and checks if it contains a PacketCreationTimeTag. If it does, it calculates the round trip time (RTT) by comparing the current time to the packet creation time. The server then echoes the packet back to the client by calling the socket's Send method.
6. TcpEchoClient::HandleRead: This method is called when the client's socket receives a new packet from the server. It reads the packet and compares its size to the expected size. If the sizes match, the client increments the

received packet count. If the received packet count equals the total number of packets expected,

the list of method execution when a packet is sent by the client and received by the server, then echoed by the server and received by the client:

1. TcpEchoClient::StartApplication: called when the client application is started
2. TcpEchoClient::Connect: called to initiate a connection to the server
3. TcpEchoClient::ConnectionSucceeded: called when the connection to the server is established
4. TcpEchoClient::SendData: called to send data to the server
5. TcpEchoServer::StartApplication: called when the server application is started
6. TcpEchoServer::HandleConnection: called when a new connection is established with the client
7. TcpEchoServer::HandleRead: called when data is received from the client
8. TcpEchoServer::EchoData: called to echo the data back to the client
9. TcpEchoServer::HandleWrite: called to write the echoed data back to the client
10. TcpEchoClient::HandleRead: called when data is received from the server (the echoed data)

The conditions under which these methods are called are as follows:

- TcpEchoClient::StartApplication: called when the client application is started, either manually or through the Simulator::Schedule method
- TcpEchoClient::Connect: called when the client application is started and a connection to the server needs to be established
- TcpEchoClient::ConnectionSucceeded: called when the connection to the server is established, indicated by a successful return value from the TcpSocket::Connect method
- TcpEchoClient::SendData: called when the client wants to send data to the server, either manually or through the Simulator::Schedule method
- TcpEchoServer::StartApplication: called when the server application is started, either manually or through the Simulator::Schedule method
- TcpEchoServer::HandleConnection: called when a new connection is established with the client, indicated by a successful return value from the TcpSocket::Accept method
- TcpEchoServer::HandleRead: called when data is received from the client, indicated by a successful return value from the TcpSocket::Recv method
- TcpEchoServer::EchoData: called to echo the data back to the client
- TcpEchoServer::HandleWrite: called to write the echoed data back to the client, indicated by a successful return value from the TcpSocket::Send method

- TcpEchoClient::HandleRead: called when data is received from the server (the echoed data), indicated by a successful return value from the TcpSocket::Recv method

One aspect of the code that stands out is the use of the PacketCreationTimeTag class, which is used to attach a timestamp to packets as they are created. This is done in the TcpEchoClient::CreatePacket method, where a new PacketCreationTimeTag object is created and attached to the packet using the AddPacketTag method:

```
PacketCreationTimeTag tag;

tag.SetSimpleValue(Simulator::Now().GetSeconds());

packet->AddPacketTag(tag);
```

The PacketCreationTimeTag class is derived from the Tag class, which is part of the ns-3 library and provides a way to attach metadata to packets. The Tag class has several pure virtual methods that must be implemented by any derived classes, including GetTypeId, GetInstanceTypeId, GetSerializedSize, Serialize, Deserialize, and Print. These methods are used to serialize and deserialize the tag data and to perform other operations on the tag.

The PacketCreationTimeTag class adds a single member variable, m_simpleValue, which is used to store the timestamp value. It also provides SetSimpleValue and GetSimpleValue methods to set and retrieve the value of this member variable.

Once the PacketCreationTimeTag has been attached to the packet, it can be retrieved later using the GetPacketTag method, as is done in the TcpEchoServer::HandleRead method:

```
PacketCreationTimeTag tag;

packet->PeekPacketTag(tag);

double creationTime = tag.GetSimpleValue();
```

The PacketCreationTimeTag is used in this code to measure the round-trip time (RTT) of packets sent between the client and server. When the client sends a packet, it includes the current time as a timestamp in the tag. When the server receives the packet, it retrieves the timestamp from the tag and calculates the RTT by subtracting the timestamp from the current time. This RTT value is then logged to the console.

how is echoing implemented?

In the provided code, echoing is implemented by the TcpEchoServer class. When the server receives a packet from a client, it creates a new packet with the same data as the received packet and sends it back to the client. This process is referred to as echoing because the server is simply sending back the data that it received from the client.

Echoing is implemented in the TcpEchoServer class by the HandleRead method. This method is called when the server receives a packet from a client. The method first creates a new packet with the same data as the received packet, using the Create<Packet> method and the data from the received packet. It then sends the new packet back to the client using the Send method.

The echoing process is repeated for each packet received by the server until the client stops sending packets or the simulation ends. The echoing process is triggered by the reception of packets from the client, and the server continues to echo packets back to the client until there are no more packets to be received.

callback functions that are registered with various objects in the ns-3 simulation. They are called at specific points in the simulation, such as when a packet is transmitted, dropped, or received.

For example, the "ClientMacTxDrop" function is registered as a callback for when a packet is dropped due to an error in the MAC layer of the client. Similarly, the "ClientIpV4L3SendOutgoingCallback" function is called whenever a packet is transmitted by the client's IPv4 layer.

The "getAddress" function is a template function that takes a function object as an argument and returns the memory address of the function. This is done by using the "target" method of the std::function object to get a pointer to the function, and then returning the value of the pointer as an integer.

The variables "ClientIpV4L3SendOutGoingCount", "ClientIpV4L3STxCount", "ServerIpV4L3SendOutGoingCount", and "ServerIpV4L3STxCount" are all counters that are incremented each time the corresponding callback function is called. They are used to track how many times the corresponding event occurs during the simulation.

This code defines several callback functions, which are essentially functions that are called when a specific event occurs. The purpose of these callback functions is to log or record information about the event when it happens.

- The ServerMacTxDrop callback function is called when a packet is dropped during transmission at the MAC layer on the server side. It logs a message indicating that the packet was dropped at the current simulation time.

- The ServerPhyTxDrop callback function is called when a packet is dropped during transmission at the PHY layer on the server side. It logs a message indicating that the packet was dropped at the current simulation time.
- The ServerPhyRxDrop callback function is called when a packet is dropped during reception at the PHY layer on the server side. It logs a message indicating that the packet was dropped at the current simulation time.
- The ClientMacTxDrop callback function is called when a packet is dropped during transmission at the MAC layer on the client side. It logs a message indicating that the packet was dropped at the current simulation time.
- The ClientPhyTxDrop callback function is called when a packet is dropped during transmission at the PHY layer on the client side. It logs a message indicating that the packet was dropped at the current simulation time.
- The ClientPhyRxDrop callback function is called when a packet is dropped during reception at the PHY layer on the client side. It logs a message indicating that the packet was dropped at the current simulation time.
- The SocketRxDrop callback function is called when a packet is dropped during reception at the socket level. It logs a message indicating that the packet was dropped at the current simulation time.
- The IpV4L3DropPkt callback function is called when a packet is dropped during transmission or reception at the IP layer. It logs a message indicating the reason for the packet drop, as well as the current simulation time.

These callback functions are typically called by the corresponding

ServerMacTxDrop is a callback function that is called when a packet is dropped due to an error during transmission at the MAC layer on the server side. The function simply logs a message indicating that the drop occurred at the current simulation time.

ServerPhyTxDrop is similar to ServerMacTxDrop, but is called when a packet is dropped due to an error during transmission at the PHY layer on the server side.

ServerPhyRxDrop is called when a packet is dropped due to an error during reception at the PHY layer on the server side.

ClientMacTxDrop, ClientPhyTxDrop, and ClientPhyRxDrop are the corresponding callback functions for the client side, and are called when a packet is dropped due to an error during transmission or reception at the MAC or PHY layer on the client side.

SocketRxDrop is a callback function that is called when a packet is dropped due to an error during reception at the socket level.

IpV4L3DropPkt is called when a packet is dropped by the IPv4 layer on either the client or server side. The function logs the reason for the drop and the current simulation time.

ClientIpV4L3SendOutgoingCallback and ServerIpV4L3SendOutgoingCallback are callback functions that are called whenever a packet is sent by the IPv4 layer on the client or server side, respectively. They increment a counter variable to track the number of packets sent.

ClientIpV4L3TX and ServerIpV4L3TX are callback functions that are called whenever a packet is transmitted by the IPv4 layer on the client or server side, respectively. They also increment a counter