OMNET++

Classes:

1. **CustomTcpEchoApp** is a class that represents a TCP echo server application. It extends the CustomTcpServerHostApp class and overrides certain methods to implement the desired behavior of the echo server.
2. **CustomTcpEchoAppThread** is a class that represents a thread of execution in the echo server. It extends the TcpServerThreadBase class and overrides certain methods to implement the desired behavior of the thread.
3. **CustomTcpServerHostApp** is a class that represents a TCP server application. It extends the ApplicationBase class and implements the TcpSocket::ICallback interface. It manages a collection of TcpServerThreadBase objects, one for each connected client.
4. **TcpServerThreadBase** is a class that represents a thread of execution in a TCP server. It extends the cSimpleModule class and implements the TcpSocket::ICallback interface. It represents a connection with a client and handles the communication with that client.
5. **TcpSocket** is a class that represents a TCP socket. It provides a number of methods for sending and receiving data, connecting and disconnecting, and managing the state of the socket.
6. **ICallback** is an interface that defines a set of callback methods that can be implemented by a class to receive notifications about the state and events of a TcpSocket. Classes that implement this interface can register themselves with a TcpSocket to receive these notifications.
7. **Packet** is a class that represents a data packet in the simulation. It provides methods for accessing and modifying the header and payload of the packet, as well as methods for adding and removing packet header tags.
8. **Protocol** is a class that represents a network protocol. It provides a number
9. **DispatchProtocolReq** is a class that represents a packet header tag that specifies the protocol to be used for sending the packet. It is used by the sendDown method of CustomTcpEchoApp to specify that the packet should be sent using the TCP protocol.
10. **CreationTimeTag** is a class that represents a packet header tag that stores the time at which the packet was created. It is used by the dataArrived method of CustomTcpEchoAppThread to calculate the delay of the received packet.
11. **LifecycleOperation** is a class that represents an operation that can be performed on a module during its lifecycle (e.g., start, stop, crash). The handleStartOperation, handleStopOperation, and handleCrashOperation methods of CustomTcpServerHostApp are called when the server module receives a start, stop, or crash operation, respectively.
12. **NodeStatus** is a class that represents the current status of a simulation node (e.g., up, down, starting up, shutting down). The handleStartOperation and handleStopOperation methods of CustomTcpServerHostApp use this class to check the status of the node before starting or stopping the server.

We have provided the code for a custom TCP echo application and server host in the INET Framework for the OMNeT++ network simulator.

The **CustomTcpEchoApp** class is derived from the **CustomTcpServerHostApp** class, which itself is derived from the ApplicationBase class in the INET Framework. **CustomTcpEchoApp** overrides the initialize, finish, and refreshDisplay methods of the **CustomTcpServerHostApp** class. It also has a sendDown method which is used to send packets down the protocol stack and a delay variable which represents the delay of the echo application.

Methods and member variables:

The **CustomTcpEchoApp** class has several member variables such as *echoFactor*, *bytesRcvd*, *bytesSent*, *packetsSent, packetsRcvd, avrgDelay, avrgDelayCounter, firstSentPacketTime, lastSentPacketTime, m_PreviousPacketDelay, and m_jitterAccumulator*. **The purpose of these variables is not clear from their names.**

The **CustomTcpEchoApp** class has a nested class called **CustomTcpEchoAppThread**, which is derived from the **TcpServerThreadBase** class. **CustomTcpEchoAppThread** overrides the established, dataArrived, handleMessage, timerExpired, and init methods of the TcpServerThreadBase class. It also has a member variable echoAppModule which is a pointer to a CustomTcpEchoApp object.

The **CustomTcpServerHostApp** class is responsible for creating and managing TCP server threads that handle incoming connections. It has a **TcpSocket** member called serverSocket which is used to listen for incoming connections and a **SocketMap** member called socketMap which stores the active sockets.

The **CustomTcpServerHostApp** class also has a ThreadSet member called **threadSet** which stores the set of server threads. it overrides several methods of the **ApplicationBase**, **TcpSocket::ICallback**, and cSimpleModule classes. These include initialize, numInitStages, handleMessageWhenUp, finish, refreshDisplay, socketAvailable, socketClosed, handleStartOperation, handleStopOperation, and handleCrashOperation.

The **TcpServerThreadBase** class is the base class for server threads in the INET Framework. It is derived from the cSimpleModule class and has a CustomTcpServerHostApp member called hostmod. It also has virtual methods such as established, dataArrived, handleMessage, timerExpired, and init.

==========

In the **CustomTcpEchoApp** class, the initialize method is used to initialize the delay and echoFactor member variables. It also sets the bytesRcvd and bytesSent variables to 0 and starts watching these variables.

The **sendDown** method is used to send a packet down the protocol stack. It adds a DispatchProtocolReq tag to the packet and sets the protocol to TCP. It then sends the packet on the socketOut gate.

The **refreshDisplay** method updates the display string for the module with the number of threads and the number of bytes received and sent.

The **finish** method is called at the end of the simulation and is used to print some statistics such as the number of bytes and packets received and sent, the throughput, and the average delay.

In the **CustomTcpEchoAppThread** class, the established method is called when a connection with a client is established. It sends the first packet to the client with a delay of echoDelay.

The dataArrived method is called when data is received from the client. It sends the received data back to the client with a delay of echoDelay.

The **handleMessage** method handles self-messages and messages from the client. If the message is a self-message, it sends the next packet to the client with a delay of echoDelay. If the message is from the client, it sends the received data back to the client with a delay of echoDelay.

The **timerExpired** method is called when a timer expires. It sends the next packet to the client with a delay of echoDelay.

The **init** method is called to initialize the server thread with a pointer to the CustomTcpEchoApp module and a TcpSocket object. It sets the echoAppModule member variable to the pointer to the CustomTcpEchoApp module.

==========

1. The **CustomTcpEchoApp** and **CustomTcpEchoAppThread** classes are part of a custom TCP echo application and server host in the INET Framework for the OMNeT++ network simulator.
2. The **CustomTcpEchoApp** class is derived from the **CustomTcpServerHostApp** class and overrides several of its methods, including initialize, finish, and refreshDisplay.
3. The **CustomTcpEchoApp** class has a delay member variable which represents the delay of the echo application, and an echoFactor member variable which has an unknown purpose.
4. The **CustomTcpEchoApp** class has several member variables for tracking statistics such as the number of bytes and packets received and sent, and the average delay.

5. The **CustomTcpEchoApp** class has a nested class called **CustomTcpEchoAppThread** which is derived from the **TcpServerThreadBase** class and overrides several of its methods.
6. The **CustomTcpEchoAppThread** class has an echoAppModule member variable which is a pointer to a CustomTcpEchoApp object.
7. The **CustomTcpServerHostApp** class is responsible for creating and managing TCP server threads that handle incoming connections. It has a TcpSocket member called serverSocket and a SocketMap member called socketMap for storing active sockets.
8. The **CustomTcpServerHostApp** class has a ThreadSet member called threadSet which stores the set of server threads.
9. The **CustomTcpServerHostApp** class overrides several methods of the ApplicationBase, TcpSocket::ICallback, and cSimpleModule classes to handle events such as incoming connections, data arrival, and socket closure.
10. The **TcpServerThreadBase** class is the base class for server threads in the INET Framework. It has a CustomTcpServerHostApp member called hostmod and several virtual methods.
11. The **CustomTcpEchoApp** class has a sendDown method which is used to send packets down the protocol stack. It adds a DispatchProtocolReq tag to the packet and sets the protocol to TCP before sending it on the socketOut gate.
12. The **CustomTcpEchoApp** class has a finish method which is called at the end of the simulation to print statistics such as the number of bytes and packets received and sent, the throughput, and the average delay.
13. In the **CustomTcpEchoAppThread** class, the established method is called when a connection with a client is established. It sends the first packet to the client with a delay of echoDelay.
14. The **dataArrived** method is called when data is received from the client. It sends the received data back to the client with a delay of echoDelay.
15. The **handleMessage** method handles self-messages and messages from the client. If the message is a self-message, it sends the next packet to the client with a delay of echoDelay. If the message is from the client, it sends the received data back to the client with a delay of echoDelay.
16. The **timerExpired** method is called when a timer expires. It sends the next packet to the client with a delay of echoDelay.
17. The `

==========

CustomTcpEchoAppThread class has an init method which is called to initialize the server thread with a pointer to theCustomTcpEchoAppmodule and aTcpSocketobject. It sets theechoAppModulemember variable to the pointer to theCustomTcpEchoAppmodule.

18. The **CustomTcpServerHostApp** class has aremoveThread` method which is called to remove a server thread from the set of active threads.

19. The **CustomTcpServerHostApp** class has a threadClosed method which is called when a server thread is closed. It removes the thread from the set of active threads and deletes the associated socket.

20. The **CustomTcpServerHostApp** class has a handleStartOperation method which is called when the module is starting. It binds the serverSocket to a local address and listens for incoming connections.
21. The **CustomTcpServerHostApp** class has a handleStopOperation method which is called when the module is stopping. It closes the serverSocket and deletes all active sockets.
22. The **CustomTcpServerHostApp** class has a handleCrashOperation method which is called when the module crashes. It closes the serverSocket and deletes all active sockets.
23. The **CustomTcpServerHostApp** class has a socketAvailable method which is called when a new connection is available on the serverSocket. It creates a new server thread and adds it to the set of active threads.
24. The **CustomTcpServerHostApp** class has a socketClosed method which is called when a socket is closed. It removes the associated server thread from the set of active threads and deletes the socket.
25. The **CustomTcpServerHostApp** class has a refreshDisplay method which updates the display string for the module with the number of active threads.
26. The **TcpServerThreadBase** class has a handleMessage method which handles messages from the client and self-messages. If the message is a self-message, it calls the timerExpired method. If the message is from the client, it calls the dataArrived method.
27. The **TcpServerThreadBase** class has a dataArrived method which is called when data is received from the client. It has an empty implementation and is intended to be overridden by derived classes.
28. The **TcpServerThreadBase** class has a timerExpired method which is called when a timer expires. It has an empty implementation and is intended to be overridden by derived classes.
29. The **TcpServerThreadBase** class has a socketDataArrived method which is called when data is received on the socket. It has an empty implementation and is intended to be overridden by derived classes.
30. The **TcpServerThreadBase** class has a socketPeerClosed method which is called when the peer closes the socket. It has an empty implementation and is intended to be overridden by derived classes.

==========

- The **CustomTcpEchoApp** and **CustomTcpEchoAppThread** classes are implementations of a TCP echo application.
- The **CustomTcpEchoApp** class is a subclass of **CustomTcpServerHostApp** and the **CustomTcpEchoAppThread** class is a subclass of **TcpServerThreadBase**.
- The **CustomTcpEchoApp** class has several member variables, including delay, echoFactor, bytesRcvd, bytesSent, packetsSent, packetsRcvd, avrgDelay, avrgDelayCounter, firstSentPacketTime, lastSentPacketTime, m_PreviousPacketDelay, and m_jitterAccumulator.
- The **delay** variable stores the value of the echoDelay parameter, which specifies the delay between the sending of each packet.
- The **echoFactor** variable stores the value of the echoFactor parameter, which is not used in the provided code.

- The bytesRcvd, bytesSent, packetsSent, and packetsRcvd variables store the number of bytes and packets received and sent by the server.
- The avrgDelay, avrgDelayCounter, firstSentPacketTime, lastSentPacketTime, m_PreviousPacketDelay, and m_jitterAccumulator variables are used to calculate various statistics about the packets, such as the average delay, the first and last sent packet times, the previous packet delay, and the jitter.
- The initialize method of the CustomTcpEchoApp class initializes the delay and echoFactor variables and sets up watches on the bytesRcvd, bytesSent, `packetsSent

  ==========

  Calculating packet delay

packet header tags are used to calculate the delay of packets by adding a CreationTimeTag to each packet when it is sent and then retrieving the value of the tag when the packet is received.

Here's how it works in more detail:

1. When a packet is sent, a CreationTimeTag is added to the packet's header using the addTagIfAbsent method and the current simulation time is set as the tag's creation time.
2. When a packet is received, the dataArrived method is called and the packet is passed as an argument.
3. The packet's data is accessed using the peekData method and the CreationTimeTags are retrieved using the getAllTags method.
4. The creation time of each CreationTimeTag is retrieved using the getCreationTime method.
5. The delay of the packet is calculated by subtracting the creation time from the current simulation time.
6. The delay is then used to update various statistics, such as the average delay, the previous packet delay, and the jitter.

==========

Here is a list of the methods that would be called in the correct order when a packet is sent by the client and received by the server, and then echoed by the server and received by the client, along with an explanation of the conditions under which each method is called:

1. sendDown method of CustomTcpEchoApp on the server side: This method is called when the server wants to send a packet down to the transport layer.
2. dataArrived method of CustomTcpEchoAppThread on the server side: This method is called when data arrives at the server from the transport layer.
3. sendDown method of CustomTcpEchoApp on the server side: This method is called when the server wants to send a packet down to the transport layer as a response to the packet received from the client.

4. socketDataArrived method of CustomTcpServerHostApp on the server side: This method is called when data arrives at the server's socket from the transport layer.
5. dataArrived method of CustomTcpEchoAppThread on the client side: This method is called when data arrives at the client from the transport layer.
6. sendDown method of CustomTcpEchoApp on the client side: This method is called when the client wants to send a packet down to the transport layer.
7. dataArrived method of CustomTcpEchoAppThread on the client side: This method is called when data arrives at the client from the transport layer.
8. sendDown method of CustomTcpEchoApp on the client side: This method is called when the client wants to send a packet down to the transport layer as a response to the packet received from the server.
9. socketDataArrived method of CustomTcpServerHostApp on the client side: This method is called when data arrives at the client's socket from the transport layer.
10. finish method of CustomTcpEchoApp on the server and client sides: This method is called when the simulation finishes and the server and client want to print out statistics about the communication between them.

==========

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

1. sendDown method of CustomTcpEchoApp on the client side: This method is called when the client wants to send a packet down to the transport layer.
2. socketDataArrived method of CustomTcpServerHostApp on the server side: This method is called when data arrives at the server's socket from the transport layer.
3. dataArrived method of CustomTcpEchoAppThread on the server side: This method is called when data arrives at the server from the transport layer.
4. sendDown method of CustomTcpEchoApp on the server side: This method is called when the server wants to send a packet down to the transport layer as a response to the packet received from the client.
5. socketDataArrived method of CustomTcpServerHostApp on the client side: This method is called when data arrives at the client's socket from the transport layer.
6. sendDown method of CustomTcpEchoApp on the client side: This method is called when the client wants to send a packet down to the transport layer.
7. dataArrived method of CustomTcpEchoAppThread on the client side: This method is called when data arrives at the client from the transport layer.
8. sendDown method of `CustomT

==========

- CustomTcpEchoApp::sendDown: This method is called when the server is ready to send the echoed packet back to the client. It is called with the packet to be sent as the argument.
- CustomTcpEchoApp::initialize: This method is called during the initialization stage of the server application. It sets up various variables and registers them for watch.

- CustomTcpEchoAppThread::established: This method is called when a connection is established between the server and the client.
- CustomTcpEchoAppThread::dataArrived: This method is called when a packet is received by the server from the client. It is called with the received packet and a bool indicating whether the packet is marked as urgent or not.
- CustomTcpEchoAppThread::handleMessage: This method is called when a message is received by the server application. It determines the type of message and takes appropriate action.
- CustomTcpEchoAppThread::timerExpired: This method is called when a timer set by the server application expires.
- CustomTcpEchoApp::finish: This method is called when the simulation finishes. It prints out various statistics about the packets sent and received by the server.
- CustomTcpEchoApp::refreshDisplay: This method is called to update the display string of the server application. It shows the number of threads and the number of bytes sent and received.

==========

There is two main classes: CustomTcpEchoApp and CustomTcpEchoAppThread. CustomTcpEchoApp is a subclass of CustomTcpServerHostApp, which is a subclass of ApplicationBase. This means that CustomTcpEchoApp has all the functionality of a basic application in INET, as well as the specific functionality of a TCP server. CustomTcpEchoAppThread is a subclass of TcpServerThreadBase, which is a simple module that handles the communication between a server and a single client.

When the CustomTcpEchoApp is initialized, it sets the value of the "echoDelay" and "echoFactor" parameters and initializes counters for bytes received and sent, and packets received and sent. The sendDown method is used to send a packet down the stack towards the TCP layer, adding a DispatchProtocolReq tag to the packet to specify that it should be sent via TCP. The method also updates the bytesSent and packetsSent counters and emits a packetSentSignal signal.

The refreshDisplay method updates the display string of the CustomTcpEchoApp object to show the number of threads (connections with clients), and the number of bytes and packets received and sent. The finish method is called when the simulation is ending, and it displays some statistics about the server's performance, including the throughput, the time of the first and last sent packets, the average delay, and the average jitter.

CustomTcpEchoAppThread handles the communication with a single client. When a connection with a client is established, the established method is called. The dataArrived method is called when a packet is received from the client, and it updates the bytesReceived and packetsReceived counters and emits a packetReceivedSignal signal. The handleMessage method is called when a self-message (timer) or another kind of message is received, and it delegates the handling to the timerExpired method if the message is a timer. The init method is used to initialize the thread with a pointer to the CustomTcpEchoApp object and a TcpSocket for the connection with the client. The threadClosed method is called when the thread is closed.

Finally, the CustomTcpServerHostApp class is responsible for creating and managing the threads (CustomTcpEchoAppThread objects) that handle the communication with the clients. It maintains a SocketMap object to keep track of the sockets and a ThreadSet object to keep track of the threads. The handleMessageWhenUp method is called when a message is received and the application is in the

==========

CustomTcpEchoApp and CustomTcpEchoAppThread. CustomTcpEchoApp is a subclass of CustomTcpServerHostApp, which is a class that provides functionality for a TCP server application. It has a number of methods that are called at different stages of the server's execution.

The initialize method is called when the server is first set up. It sets some member variables, such as the delay and echo factor, based on parameters provided in the simulation configuration. It also sets up some "watches" on other member variables, which allows them to be monitored during the simulation.

The sendDown method is called when the server wants to send a packet down the stack, towards the transport layer. It takes a packet as an argument and updates some counters to keep track of the number of bytes and packets sent. It then adds a special "tag" to the packet to specify that it should be sent using the TCP protocol. Finally, it sends

the packet down the stack using the "socketOut" gate.

The refreshDisplay method is called periodically during the simulation to update the visual representation of the server in the simulation environment. It sets the text that is displayed next to the server's icon to show the number of threads (connections) the server has, and the number of bytes and packets received and sent.

The finish method is called when the simulation is ending. It prints out some statistics about the server's performance, such as the number of bytes and packets received and sent, the throughput (amount of data transmitted per second), and the delay (amount of time it takes for a packet to be sent and received). It also calculates and prints the average jitter (variation in delay) for the packets received by the server.

The CustomTcpEchoAppThread class represents a separate thread (or connection) for each client that connects to the server. It has several methods that are called at different points during the execution of the thread.

The established method is called when a connection with a client is established.

The dataArrived method is called when a packet is received from a client. It updates some counters to keep track of the number of bytes and packets received and emits a signal to indicate that a packet was received. It also extracts a special "tag" from the packet that indicates the time at which the packet was created, and uses this to calculate the delay for the packet. It also calculates the jitter for the received packets by comparing the current delay with the previous delay. Finally, it echoes the received packet back to the client by calling the sendDown method of the CustomTcpEchoApp class.

The connectionClosed method is called when the connection with a client is closed. It updates some counters to keep track of the number of bytes and packets received and sent and prints out some statistics about the thread's performance.

The peerClosed method is called when the connection with a client is closed from the client's side. It updates some counters to keep track of the number of bytes and packets received and sends a message to close the connection from the server's side as well.

==========

A TCP echo application in C++, which is essentially a program that sends back any data that it receives. There are two main classes in this code: CustomTcpEchoApp and CustomTcpEchoAppThread. CustomTcpEchoApp is a subclass of CustomTcpServerHostApp, which is a class that handles the creation and management of socket connections for the application. CustomTcpEchoApp has several member variables such as delay, echoFactor, bytesRcvd, bytesSent, packetsSent, and packetsRcvd, which are used to store information about the packets that are sent and received by the application. The class also has several member functions, including sendDown(), initialize(), finish(), and refreshDisplay(). These functions are called at different stages of the application's execution to perform specific tasks.

CustomTcpEchoAppThread is a subclass of TcpServerThreadBase, which is a class that handles the communication between the application and individual clients that are connected to it. CustomTcpEchoAppThread has a member variable called echoAppModule, which is a pointer to an instance of the CustomTcpEchoApp class. The class also has several member functions, including established(), dataArrived(), handleMessage(), timerExpired(), and init(). These functions are called at different stages of the communication with a client to perform specific tasks.

When a client sends a packet to the server, the packet is received by the server and passed to the dataArrived() function of the CustomTcpEchoAppThread class. This function processes the packet and then calls the sendDown() function of the CustomTcpEchoApp class to send the packet back to the client. The sendDown() function adds a TCP protocol tag to the packet and then sends it to the client through the "socketOut" gate. When the packet is received by the client, it is passed to the handleMessage() function of the CustomTcpEchoAppThread class, which processes the packet and updates the appropriate member variables of the CustomTcpEchoApp class.

==========

This is a C++ implementation of a TCP echo application. This application is a server that listens for incoming client connections and echoes any data received back to the client.

There are two main classes in the code: CustomTcpEchoApp and CustomTcpEchoAppThread. CustomTcpEchoApp is derived from the CustomTcpServerHostApp class, which is itself derived from the ApplicationBase class. This class is responsible for setting up the server socket, handling incoming client connections, and creating a new thread for each connected client.

CustomTcpEchoAppThread is a class that is responsible for handling the communication with a single client. It is derived from the TcpServerThreadBase class, which provides the basic functionality for a server thread. CustomTcpEchoAppThread overrides several virtual methods to implement the echo functionality.

When a client sends data to the server, it is received by the CustomTcpEchoAppThread::dataArrived method. This method processes the received data and sends it back to the client by calling the CustomTcpEchoApp::sendDown method. This method adds the necessary TCP and dispatch protocol headers to the packet and sends it out through the socket connection.

The CustomTcpEchoApp class also includes several variables and methods for collecting statistics on the server's performance. These include the number of bytes and packets received and sent, the average delay between when a packet is received and when it is echoed back to the client, and the throughput of the server (measured in kbps). These statistics are displayed at the end of the simulation in the CustomTcpEchoApp::finish method.

In addition to echoing data back to the client, the CustomTcpEchoAppThread class also calculates the jitter (variation in delay) for the packets received from the client. It does this by comparing the delay of each packet to the delay of the previous packet and adding the difference to an accumulator. The average jitter is then calculated by dividing the accumulator by the number of packets received.

==========

CustomTcpEchoApp is a subclass of CustomTcpServerHostApp, which is a class that extends ApplicationBase and implements the TcpSocket::ICallback interface. This class is responsible for handling incoming connections and managing a socket map to keep track of the open sockets. It has a method threadClosed that removes a TcpServerThreadBase instance from the thread set when it is closed. It also has methods to handle start, stop, and crash operations, as well as methods to handle messages when the application is up and to refresh the display.

CustomTcpEchoApp has several member variables, including delay, echoFactor, bytesRcvd, bytesSent, packetsSent, packetsRcvd, avrgDelay, avrgDelayCounter, firstSentPacketTime, lastSentPacketTime, m_PreviousPacketDelay, and m_jitterAccumulator. It also has a method sendDown to send a message down the protocol stack, and methods initialize, finish, and refreshDisplay to perform initialization, finalization, and display tasks, respectively.

CustomTcpEchoAppThread is a subclass of TcpServerThreadBase, which is a class that extends cSimpleModule and implements the TcpSocket::ICallback interface. It has a method init to initialize the thread with a pointer to the CustomTcpServerHostApp instance and a TcpSocket instance. It also has methods established, dataArrived, handleMessage, and timerExpired to handle various events related to the socket.

CustomTcpEchoAppThread has a member variable echoAppModule, which is a pointer to the CustomTcpEchoApp instance that owns the

========== delay and jitter ==========

CustomTcpEchoApp and CustomTcpEchoAppThread. CustomTcpEchoApp is a subclass of CustomTcpServerHostApp, which is itself a subclass of ApplicationBase. CustomTcpEchoApp serves as the server in a client-server communication, and it listens for incoming connections and handles them using instances of CustomTcpEchoAppThread.

CustomTcpEchoApp has several member variables that keep track of various statistics, such as the number of bytes and packets sent and received, and the average delay of received packets. It also has a method called sendDown that is used to send packets to the client, and a method called finish that is called when the simulation is complete and is used to print out various statistics about the communication.

CustomTcpEchoAppThread is a subclass of TcpServerThreadBase, and it is used to handle individual connections from clients. It has several methods that are called in response to various events, such as dataArrived, which is called when a packet is received from the client, and timerExpired, which is called when a timer expires. It also has a method called established, which is called when a connection is established with the client.

In the dataArrived method, the server calculates the delay of the received packet by looking at the packet header tag CreationTimeTag, which records the time at which the packet was created. The server also calculates the average jitter of received packets by comparing the delay of