JQUIC: Partial QUIC Protocol Implementation, API & Example Distributed File System (DFS) Application

COMP8047 - Major Project

KB - A00000000
April 23, 2021

Table of Contents

1 Introduction

1.1 Student Background

1.2 Acknowledgments

The student would like to thank my parents for support as well as Aman Abdulla who provided assistance in completing this project.

1.3 Project Description

The aim of this project was to create a partial Java server and client implementation based on the latest draft of Quick UDP Internet Connections (QUIC) protocol (Iyengar and Thomson) called JQUIC and an API based on it and then to create a Distributed File Server (DFS) application based on this Java QUIC API. The implementation of DFS application was fairly standard with a multi-threaded central server, that clients can connect to simultaneously, to edit, delete and store files on as it is mostly just a demonstration application for the API.

1.3.1 Essential Problems

There are two main problems that the student is trying to solve with this project. The first problem is that there is currently no easy-to-use API for QUIC protocol implemented in Java.

There is also currently no recent Java implementation of the QUIC server which is the second problem which this project aims to solve.

1.3.2 Goals and Objectives

As mentioned in the main project description, the goal of this project is to create a partial QUIC Java server and client implementation that supports multi-threading with an easy-to-use API similar to the one that Java already has for TCP sockets. This implementation will then be open-sourced so that other users help develop it into a full-fledged implementation of QUIC protocol in Java. The other goal is to demonstrate the API usage via Distributed File Server application built on top of this API. Both of these goals have been completed successfully, though with some stability and performance problems as discussed in "Implications of Implementation" section of this report.

2 Body

2.1 Background

Java Programming Language is one of the most used programming languages in the world based on the well-known TIOBE index ("TIOBE Index | TIOBE - The Software Quality Company"). The current commonly used transport layer protocol, Transport Control Protocol (TCP), already has a lot of implementations in Java including the official Socket API implementation by Oracle (Socket (Java Platform SE 7)), while **Quick UDP Internet Connections** (QUIC) protocol only has one called "kwik" and it's only an API for client-side ("Quicwg/Base-Drafts") which is a problem for the reasons mentioned below. QUIC is a new "multiplexed and secure general-purpose transport protocol", that provides "stream multiplexing, stream and connection-level flow control, low-latency connection establishment, connection migration and resilience to NAT rebinding, authenticated and encrypted header and payload"(Iyengar and Thomson). This protocol was first unveiled publicly by Google in June 2013 (Weiss) and is now in the stage of rapid development (Kakhki et al. 86) with multiple drafts of the protocol submitted to Internet Engineering Task Force (IETF) each year. QUIC is actually built on top of IP/UDP stack (Pandya et al. 27) so it is a part of application layer, but implements a majority of transport layer features for reliability like congestion control (Kakhki et al. 86) since UDP is unreliable and does not have them as per the original UDP specification (Postel). The diagram below shows where QUIC fits in as part of the network stack:

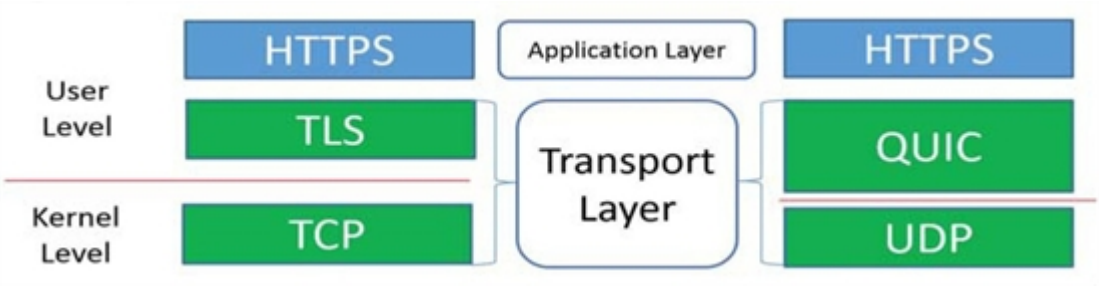


Figure 1: TLS vs QUIC protocol stack (Pandya et al. 27)

The reason to implement this new protocol is that QUIC has numerous advantages over Transport Control Protocol (TCP), which is the current standard protocol for reliable connections (Joshi and Krishnamurthy), due to both its modernity and its application layer implementation as mentioned before. For example, **Transport layer congestion control** is a very important feature of the TCP/IP network stack as it allows for "both fair and high utilization of Internet links shared by multiple flows" (Kakhki et al. 86). QUIC improves on TCP's congestion control by implementing "...better estimation of connection Round-trip Time (RTTs) and [detecting] and [recovering] from loss more efficiently" (Kakhki et al. 87). One of the largest benefits of QUIC over TCP is the much faster connection establishment due to ability to establish connections without the three-way handshake when the client has connected to the server before (Kakhki et al. 87). There are other beneficial features to QUIC such as fixing "head of line" blocking problem which is present in current IP+TCP+HTTP/2 stack. "Head of line" blocking happens because "HTTP/2 allows multiple objects to be fetched over the same connection, using multiple streams within a single flow" (Kakhki et al. 87). TCP causes all streams to stall when a loss occurs in even a single stream. In contrast, QUIC allows other streams to continue exchanging packets (Kakhki et al. 87). QUIC also has "forward error correction and improved privacy and flow compared to TCP" (Kakhki et al. 87). QUIC also has connection IDs which is useful for maintaining connection when changing networks like for example, changing from WiFi to mobile data network (Pandya et al. 27). This can be very useful when for example, driving around the city and automatically connecting to multiple different public WiFi networks.

In terms of popularity, QUIC is already "enabled by default in Google's Chrome browser and underlying Chromium open source browser code" (Mulks). Furthermore, HTTP/3, the next version of HTTP standard uses QUIC as its transport layer protocol and in fact was previously known as "HTTP over QUIC" (Ghedini). This solidifies QUIC as an upcoming reliable transport layer protocol standard that might overtake TCP in the future in terms of popularity.

The above reasons show that there are reasons to use QUIC over TCP which include its advantages outlined above as well as its upcoming usage as main transport-layer protocol for HTTP/3 stack which will no doubt be used as a standard protocol for majority of API and web browser communications in the future once more browsers support it so it makes sense to create a QUIC Implementation, Client/Server API and a test application like Distributed File Server in Java to server as foundation for an HTTP/3 implementation or other applications.

However, there are some drawbacks to QUIC. While “QUIC outperforms TCP+HTTPS [stack] in nearly every scenario” according to the tests done by both Google and the paper authors (Kakhki et al. 87), QUIC has problems with out-of-order packet delivery which it interprets as loss and so, “performs significantly worse than TCP in many scenarios”(Kakhki et al. 87). QUIC also suffers performance problems on mobile devices due to its “application-layer packet processing and encryption” (Kakhki et al. 87) which can be a huge setback due to popularity of mobile devices with more than 50% of worldwide traffic being mobile phones (“Mobile Vs. Desktop Usage (Latest 2019 Data)”). These drawbacks might be rectified as the protocol is further developed and having more implementations for the protocol will only serve to further help the protocol be developed as it will find potential problems with the protocol in practice.

2.2 Project Statement

This project called JQUIC is about creating an open-source Java implementation of QUIC and an easy-to-use API for it as well as an example Distributed File Server (DFS) application for it for demonstration.

2.3 Possible Alternative Solutions

Alternative Solutions for this project mostly consist of QUIC implementations in other programming languages such as Facebook’s C++ implementation “mvfst” or Microsoft’s C implementation “MsQuic” (“Quicwg/Base-Drafts”). There’s also another QUIC implementation in Java called “kwik”, but it does not have a server implementation or an easy-to-use API (“Quicwg/Base-Drafts”).

2.4 Chosen Solution

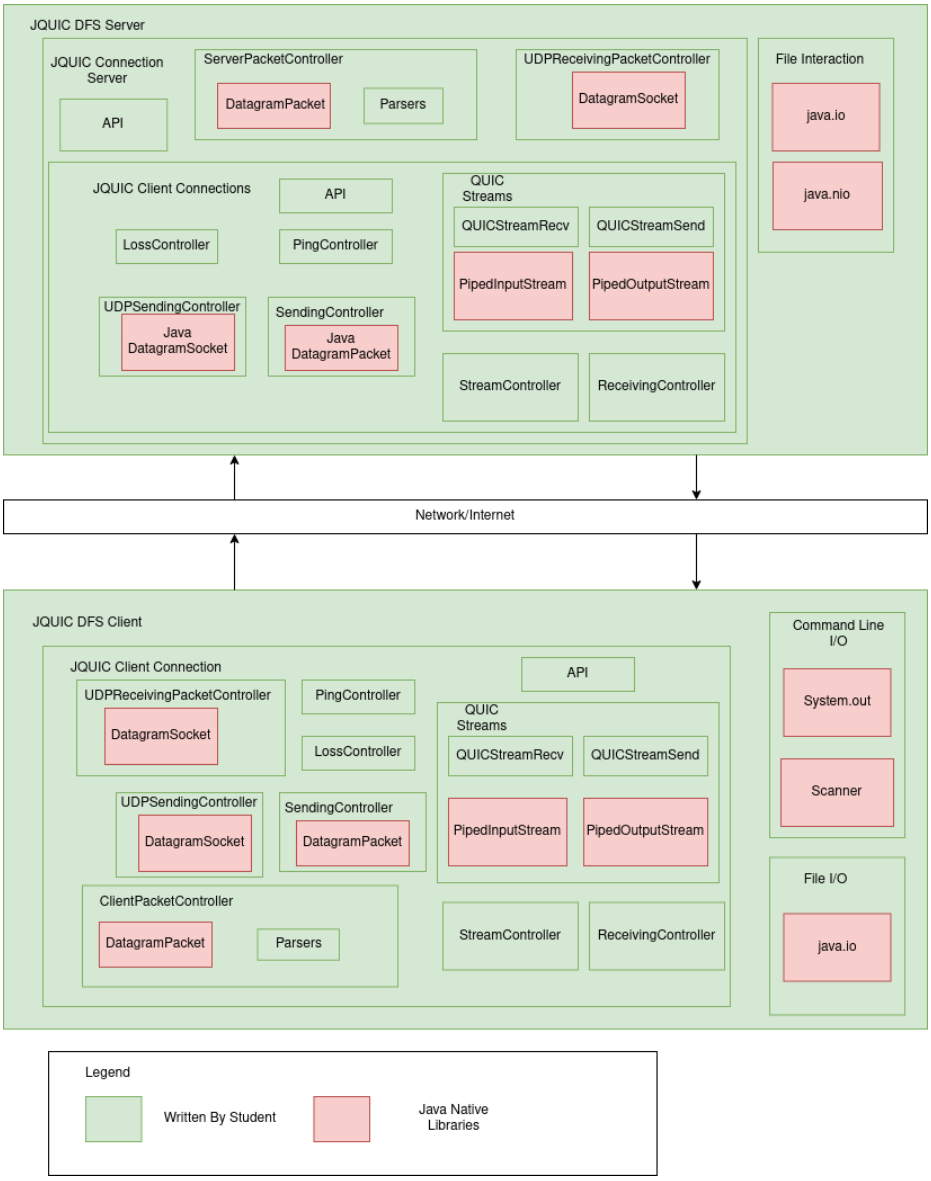
Based on the alternative solutions listed above, it was decided to implement QUIC protocol in Java programming language and to implement both Server and Client with an easy-to-use API as it would make this project unique in terms of different QUIC implementations and would create a basis for a more full-fledged QUIC implementation later with little change to the programs that were built based on the API.

2.5 Details of Design and Development

As this is a fairly large, multilevel project, there are multiple different aspects to it, each of which will be explained in this section.

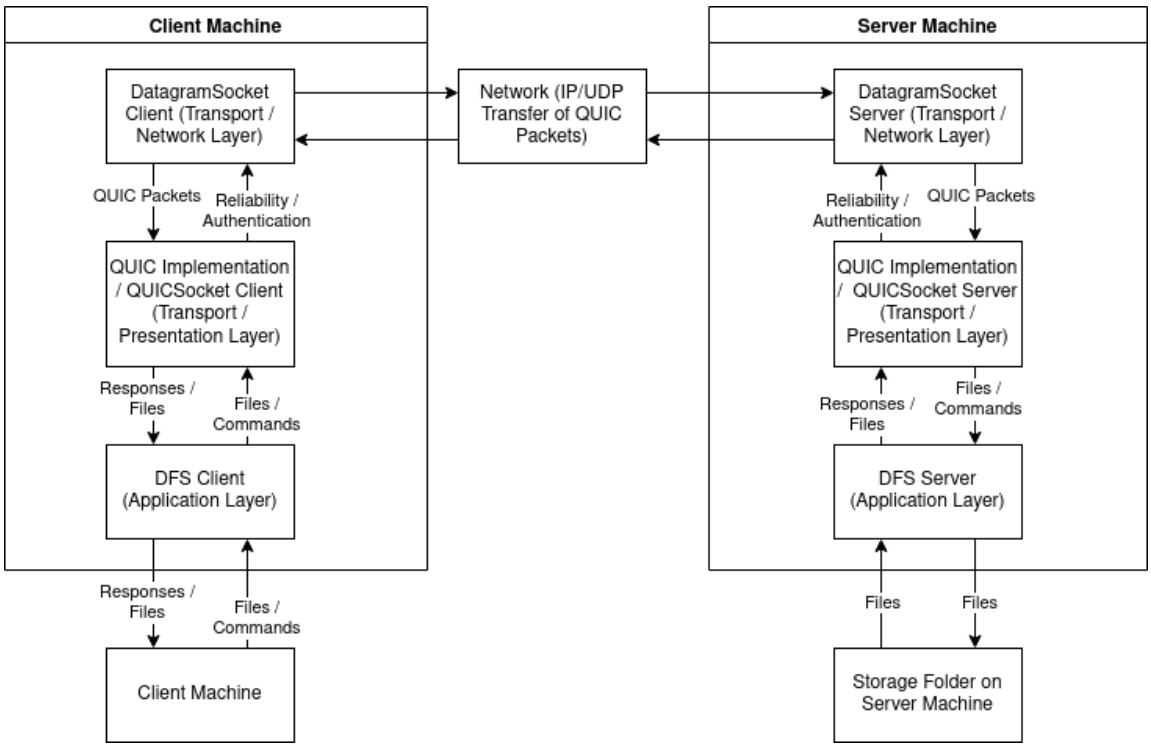
2.5.1 System Architecture Diagram

This is the system architecture diagram for this project:



Starting with architecture diagram as it gives the best overview, it shows that a lot of the building blocks for both client and server are very similar with some exceptions. For example, ServerPacketController and UDPReceivingPacketController are in purview of the QUIC connection server, not each specific connection while for client, they are all part of single client connection and all of them get closed when the connection is closed. For server, this is because server needs to accept more connections which requires the threads for those two controllers to keep running to do so. Each Controller will be explained in detail in Data Flow Diagrams section.

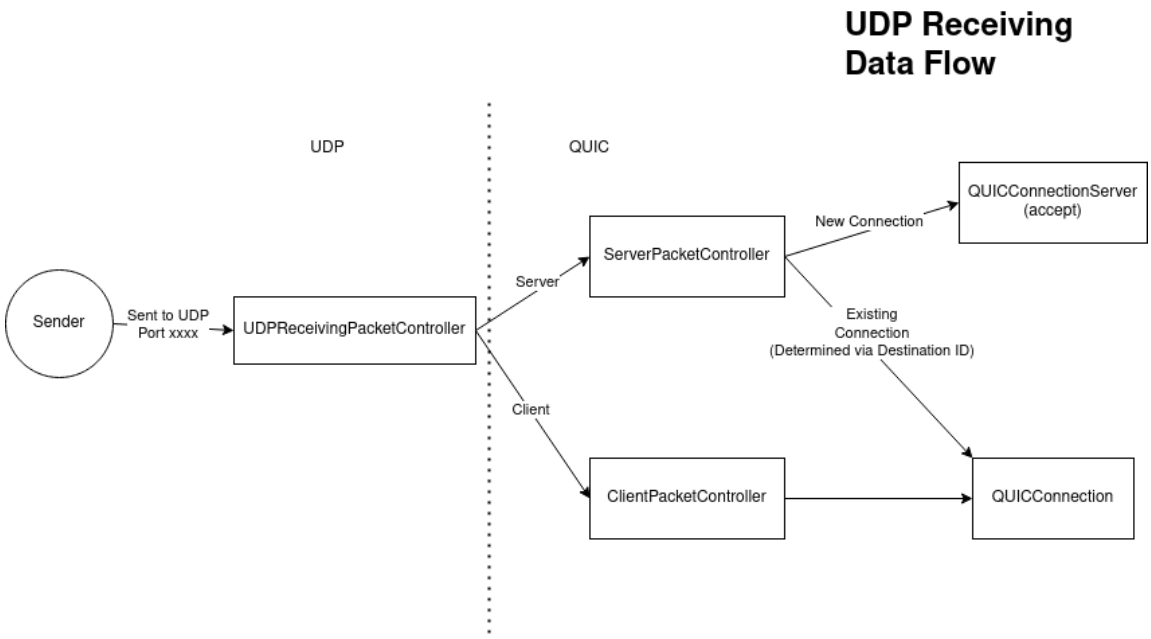
2.5.2 System Context Diagram



The above diagram shows the way that DFS Client and QUIC Implementation interact with other protocols in network stack as well as client and server machine.

2.5.3 Data Flow Diagrams

There are basically three different levels of Data Flow: UDP Receiving, QUIC Connection and QUIC Stream Flow. UDP Receiving has to do with receiving the UDP (Datagram) packet and parsing the QUIC packets from it. Here is a diagram of this (think of each box as its own thread/component of the whole system):

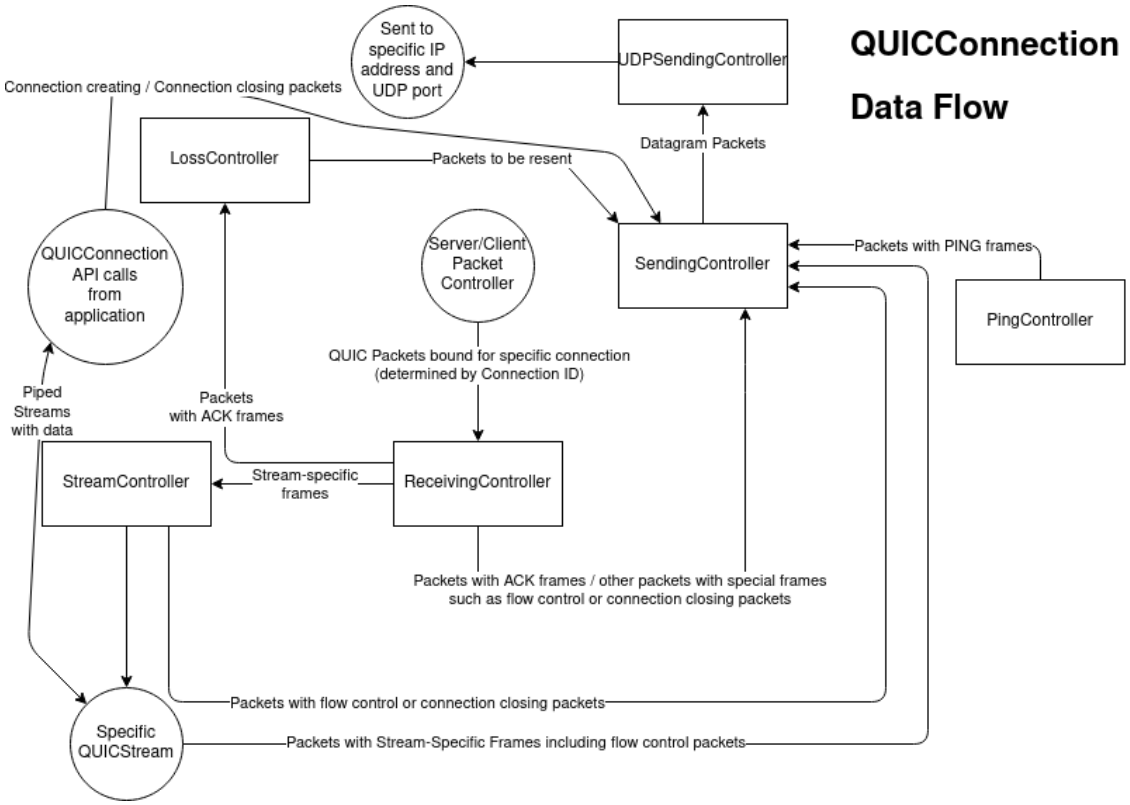


Here is the description of each component in the above diagram:

- **UDPReceivingPacketController**: receives UDP packets to a specific UDP port and quickly forwards them to either Server or Client Packet Controller without any checking; this Controller is designed to receive as fast as possible.
- **ServerPacketController**: gets UDP packets from UDPReceivingPacketController on the server-side and splits them into QUIC packets and determines which client thread to send it to depending on connection ID; it has other functions such as approving new connections.
- **ClientPacketController**: gets UDP packets from UDPReceivingPacketController on the client-side and splits them into QUIC packets and determines whether these packets are actually coming from the server and discards them if that is not the case; otherwise, it sends it to the single QUICConnection that was created as part of the client.

- **QUICConnectionServer**: gets an Initial Packet with packet number 0 from ServerPacketController and accepts new connection with “accept” method call similar to Java’s native class ServerSocket for TCP.
- **QUICConnection**: this is where all the main protocol actions take place; this is also where the API for QUIC protocol is implemented similar to Java’s native class Socket for TCP; this class is very similar for both client and server with some differences; this class will be explained more in the next Data Flow Diagram

Here is the data flow diagram for QUICConnection itself:



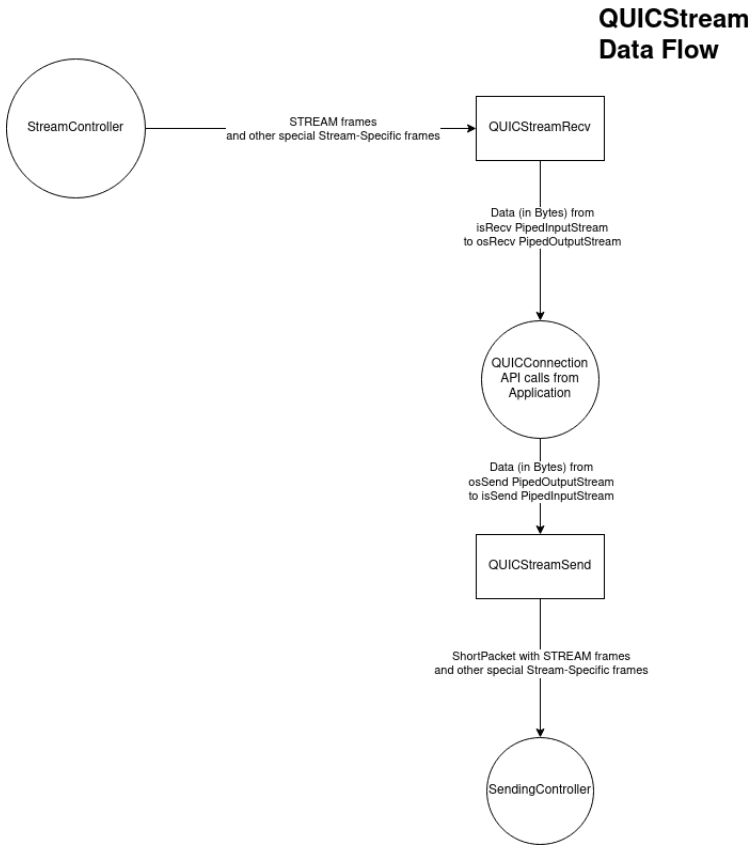
Here is the description of each component in the above diagram:

- **ReceivingController**: receives from ServerPacketController/ClientPacketController and processes QUIC packets and creates ACK frames and other packets with special frames to send back to SendingController; it also gives the packets for streams to

StreamController and packets with ACK frames to LossController and gives errors if packets are not compliant with the draft requirements; its other function is to manage connection-level flow control

- **SendingController**: creates Datagram (UDP) packets out of the different QUICPackets it receives from ReceivingController as well as other controllers; it then passes this DatagramPacket to UDPSendingController
- **UDPSendingController**: sends Datagram (UDP) packets that it gets from SendingController to specific to UDP address and port (determined by connection ID); it does so every few ticks to make sure the receiving side does not get overwhelmed
- **StreamController**: receives stream-specific frames from ReceivingController and forwards them to a specific stream; it also checks for some protocol violations and manages some parts of flow control for streams
- **Specific QUICStream**: streams is what QUIC protocol uses to send and receive application data in a connection; there can be multiple streams for a single connection which makes it different from TCP; more will be explained in the next data flow diagram
- **LossController**: Determines whether a loss has occurred on sending side by processing packets with ACK frames; it also controls which packets needs to be resent and then sends them to SendingController
- **PingController**: only on client-side and only if the “defer_timeout” option is set to true; sends packets with PING frames to SendingController to make sure the connection does not time out
- **QUICConnection API calls from Application**: these are API calls that application sends to the QUICConnection; these will be explained more in the “API Documentation” sub-section

Here is the data flow diagram for QUICStream:



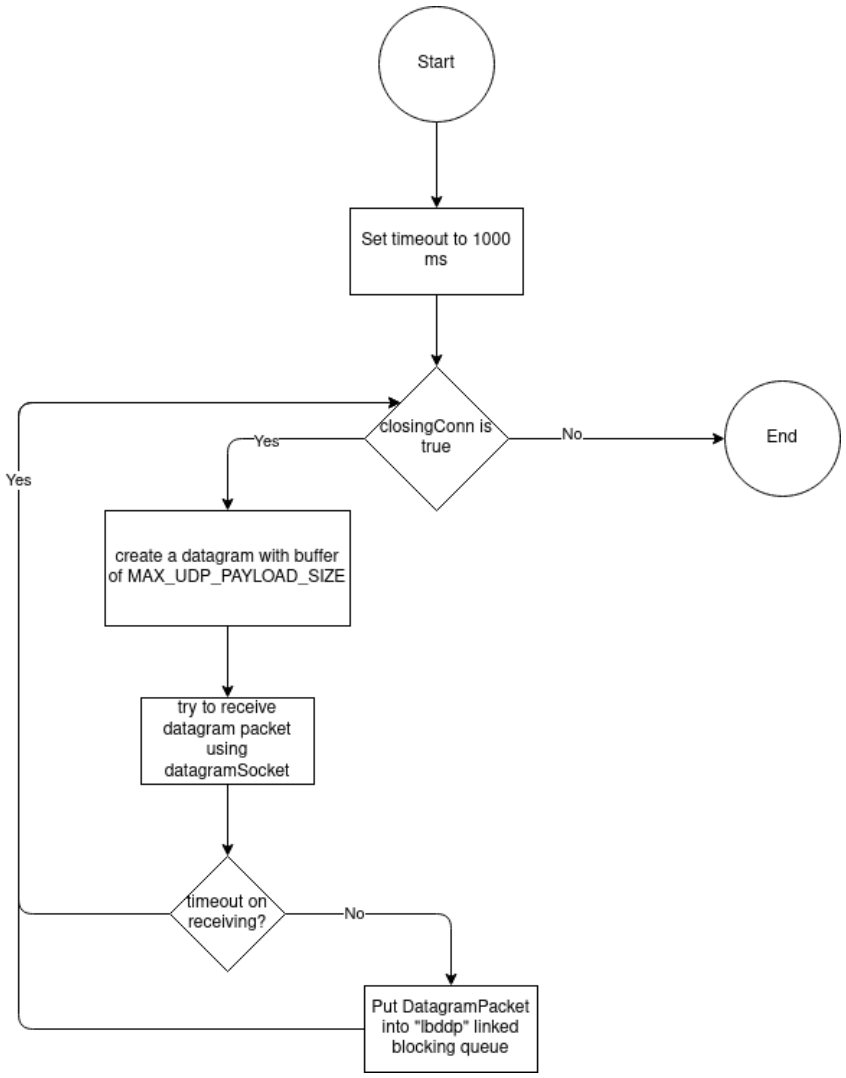
Here is the description of each component in the above diagram:

- **QUICStreamRecv**: processes different stream-specific frames including STREAM frames (frames with data) which it then sends through isRecv Piped Input Stream to application; also looks for protocol violations, manages out-of-order data (more on this in “Implications of Implementation”) as well as stream-specific flow control
- **QUICStreamSend**: gets data from application through isSend Piped Input Stream and sends it to the SendingController; also manages both connection-level and stream-level flow control to make sure it does not send too much data

2.5.4 JQUIC State Diagrams

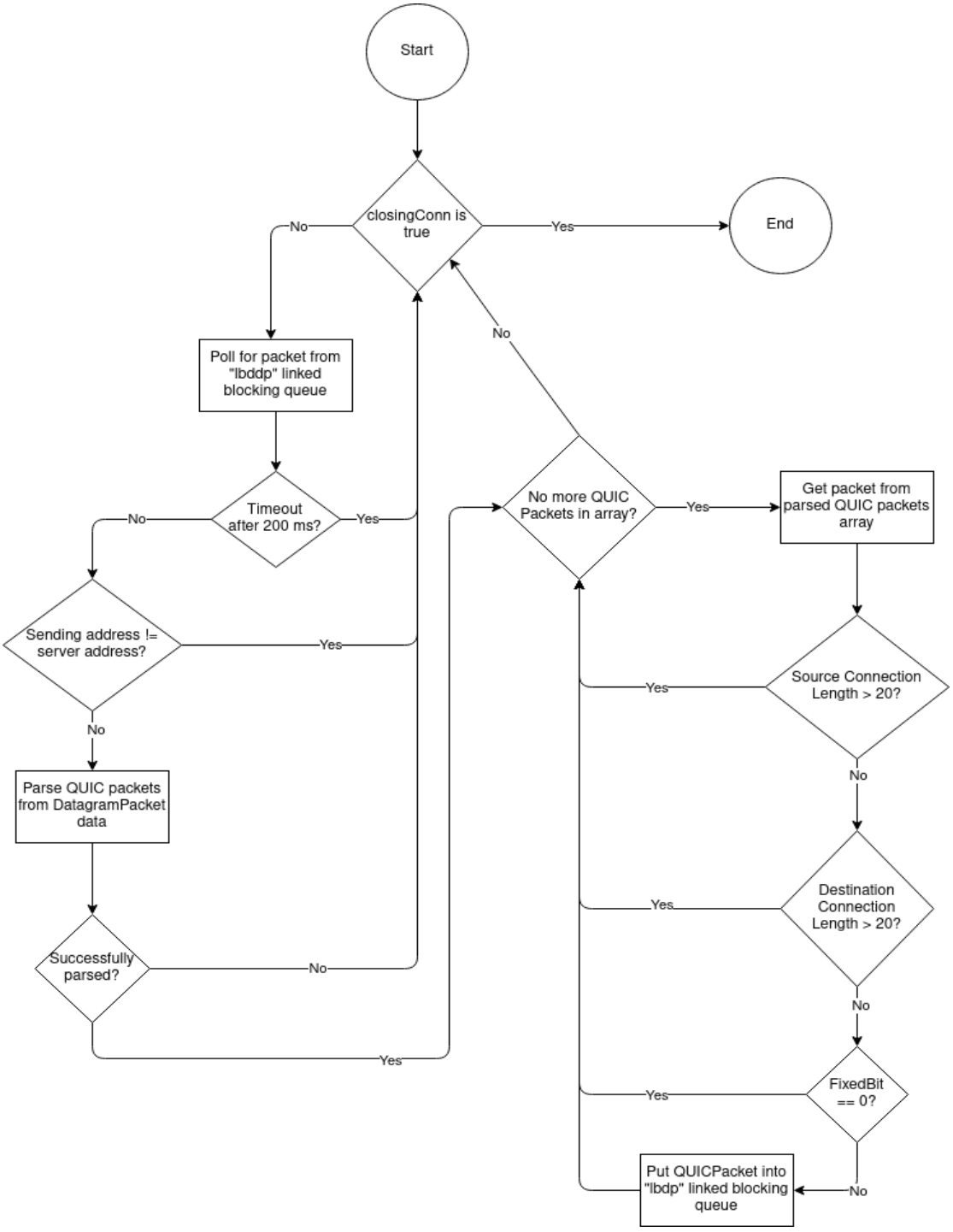
This section contains state diagrams for majority of the components of JQUIC protocol implementation which were described in the Data Flow Diagrams section. Note that these diagrams aren't as detailed as the implementation and were mostly used as an extra guide to not lose track of the different components so they might be somewhat simplified in nature. Data Flow, Class and System Architecture diagrams give extra description for the below controllers as well as show how they all fit into the overall architecture of this project.

2.5.4.1 UDPReceivingController



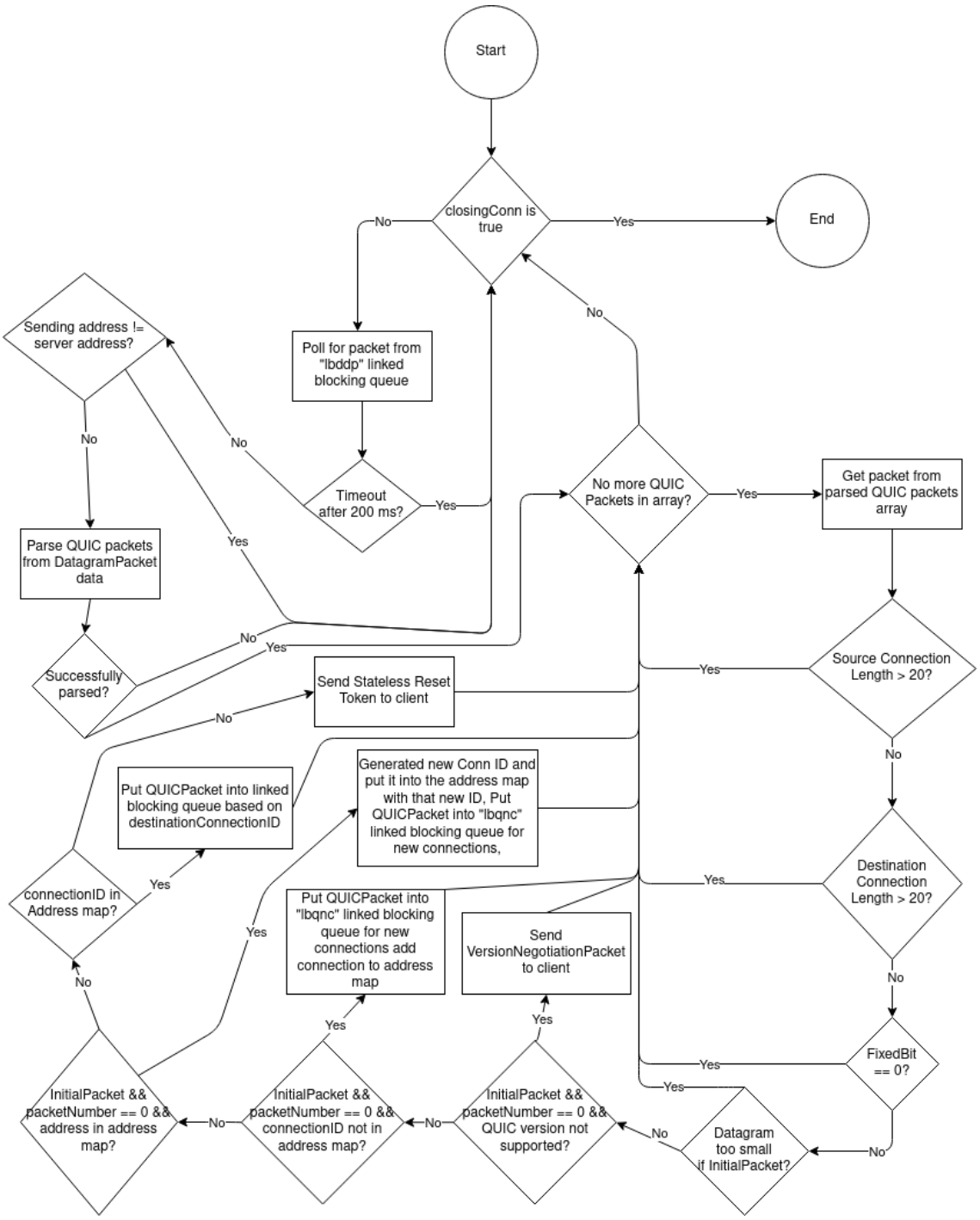
The above diagram is the state diagram for UDPReceivingController. The reason that there is a timeout set is so that closingConn condition (which is what regulates whether the connection is closing or not) is checked and the thread does not hang after QUIC connection has ended. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.2 ClientPacketController



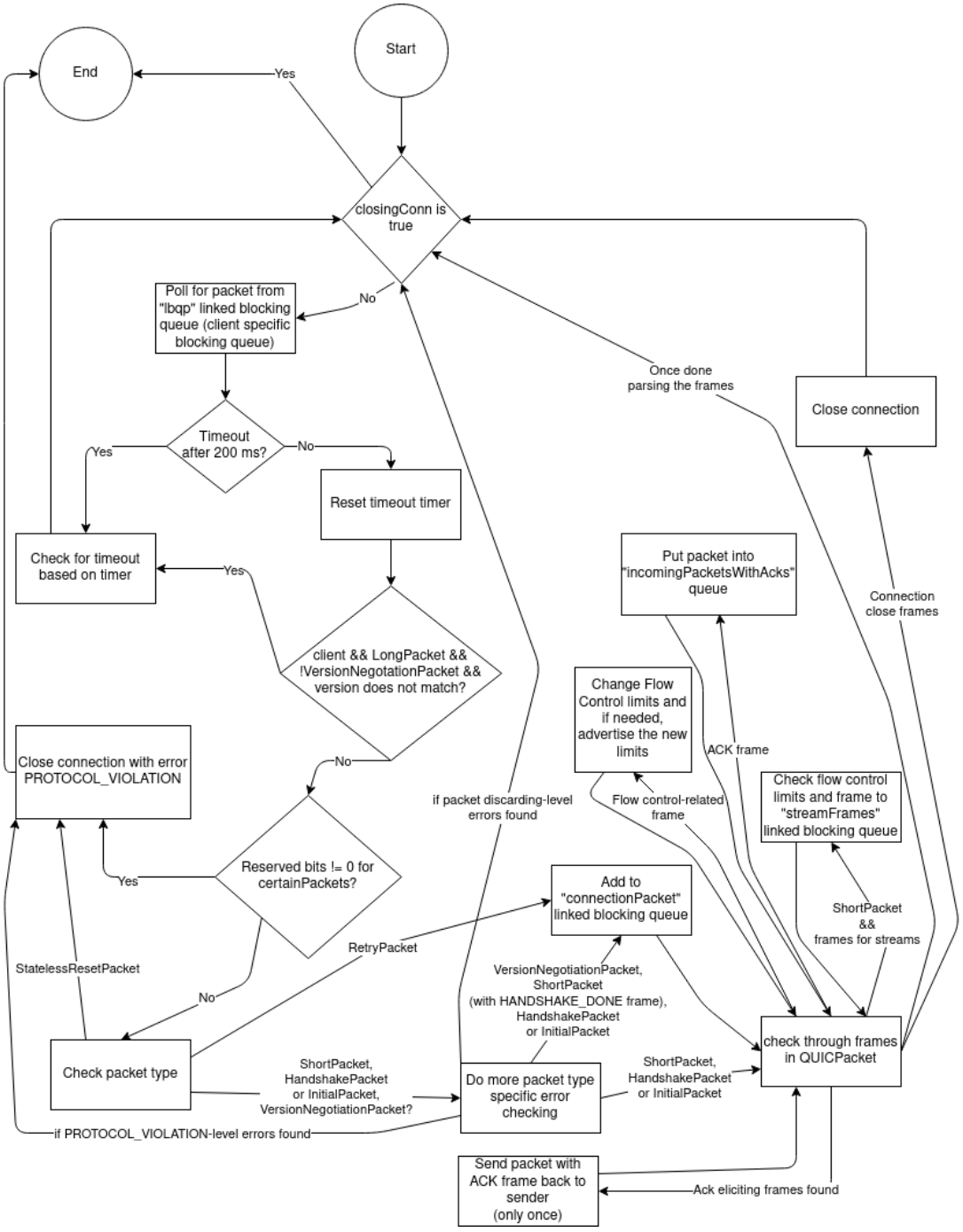
The above diagram is the state diagram for ClientPacketController. The reason for the timeout is similar to UDPReceivingController: so that the thread does not hang when connection is closed. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.3 ServerPacketController



The above diagram is the state diagram for ServerPacketController. ServerPacketController has a lot more checks compared to ClientPacketController as it has to make sure that the ConnectionID is correct or new as this is how it determines which client queue to send the QUIC packet to. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

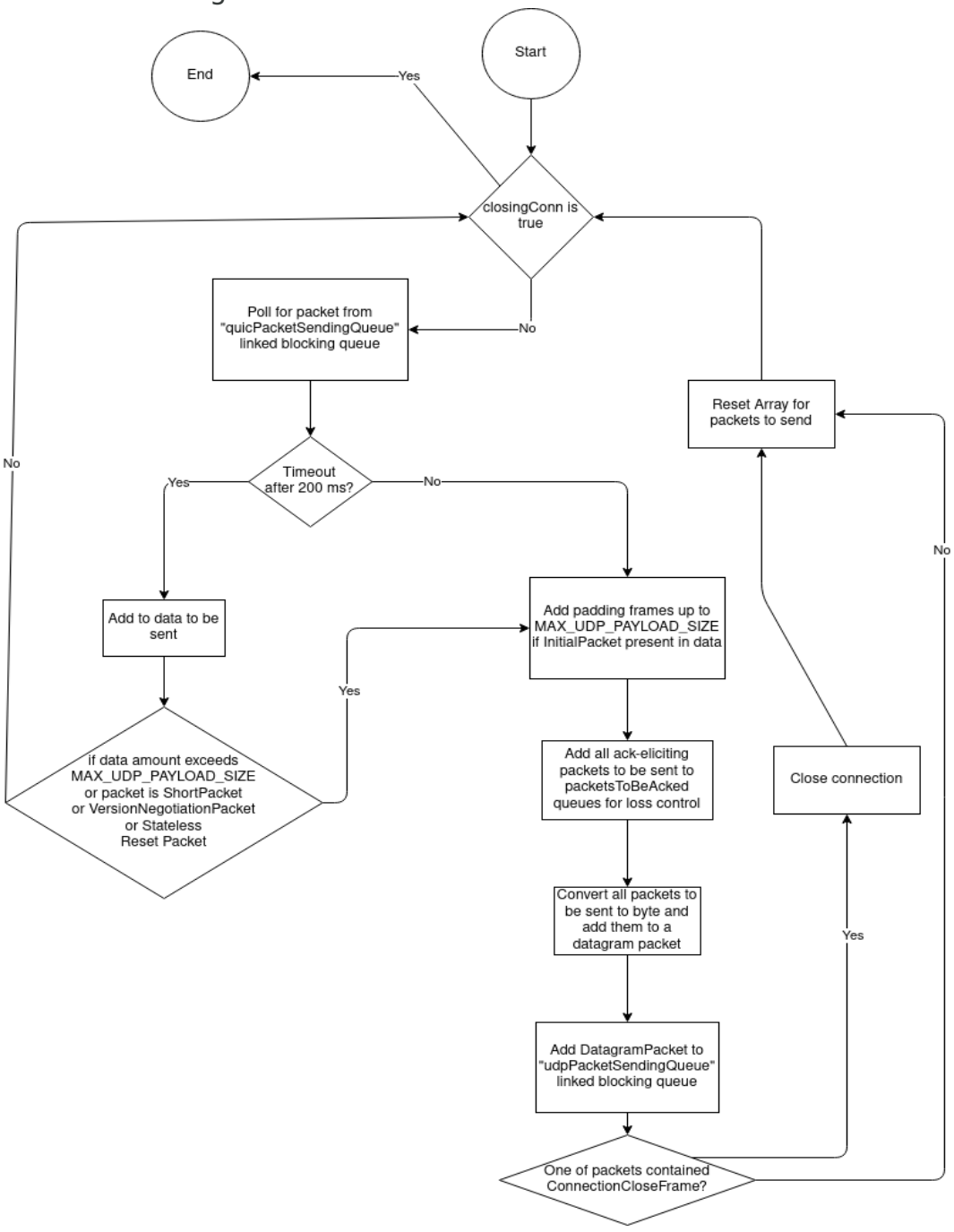
2.5.4.4 ReceivingController



The above diagram is the state diagram for ReceivingController. As ReceivingController is a fairly complex class, some parts of it have been omitted (like packet-specific error checking

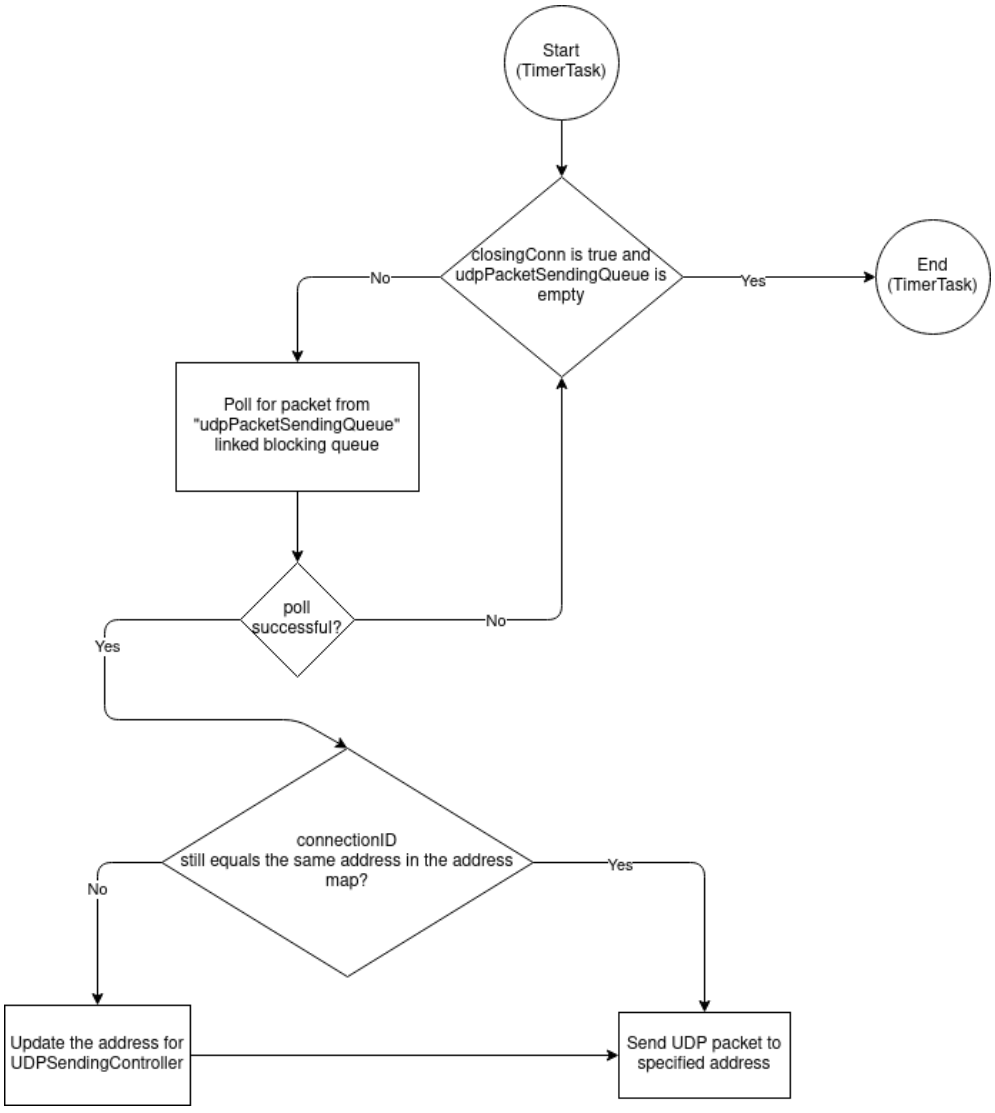
and specific flow control proceedings) for the diagram to save space as well as to make it more readable. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.5 SendingController



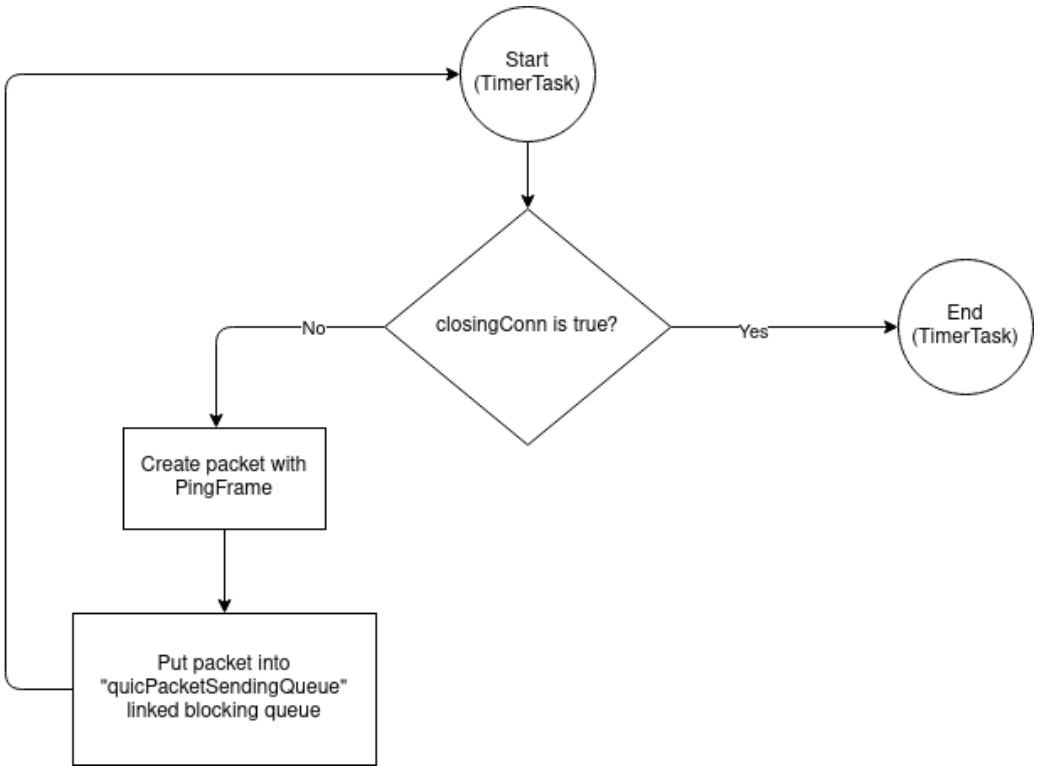
The above diagram is the state diagram for SendingController. It has been somewhat simplified compared to the actual implementation for easier readability. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.6 UDPSendingController



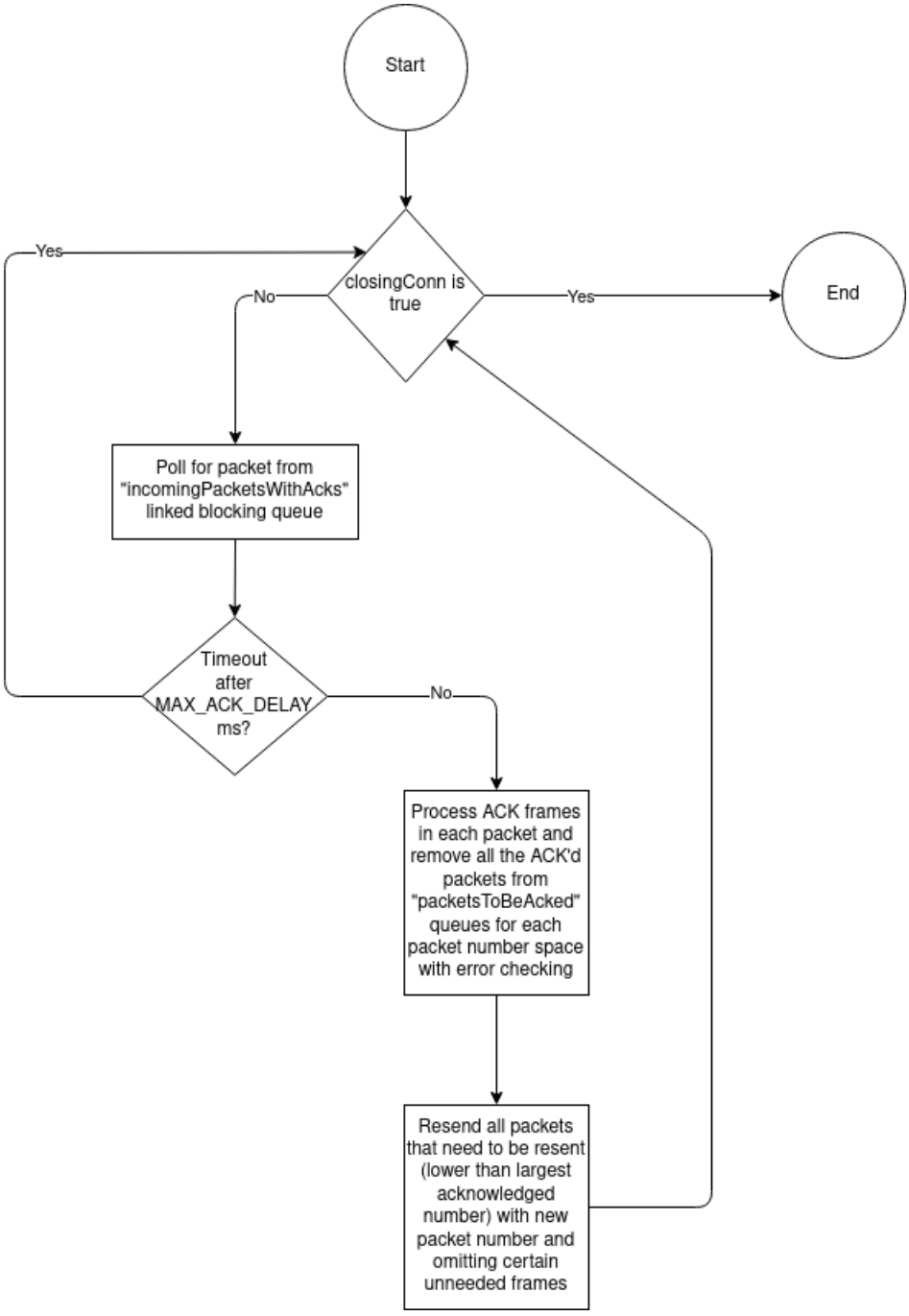
The above diagram is the state diagram for UDPSendingController. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.7 PingController



The above diagram shows the state diagram for PingController that makes sure that the client does not timeout when “defer_timeout” option is set to true. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

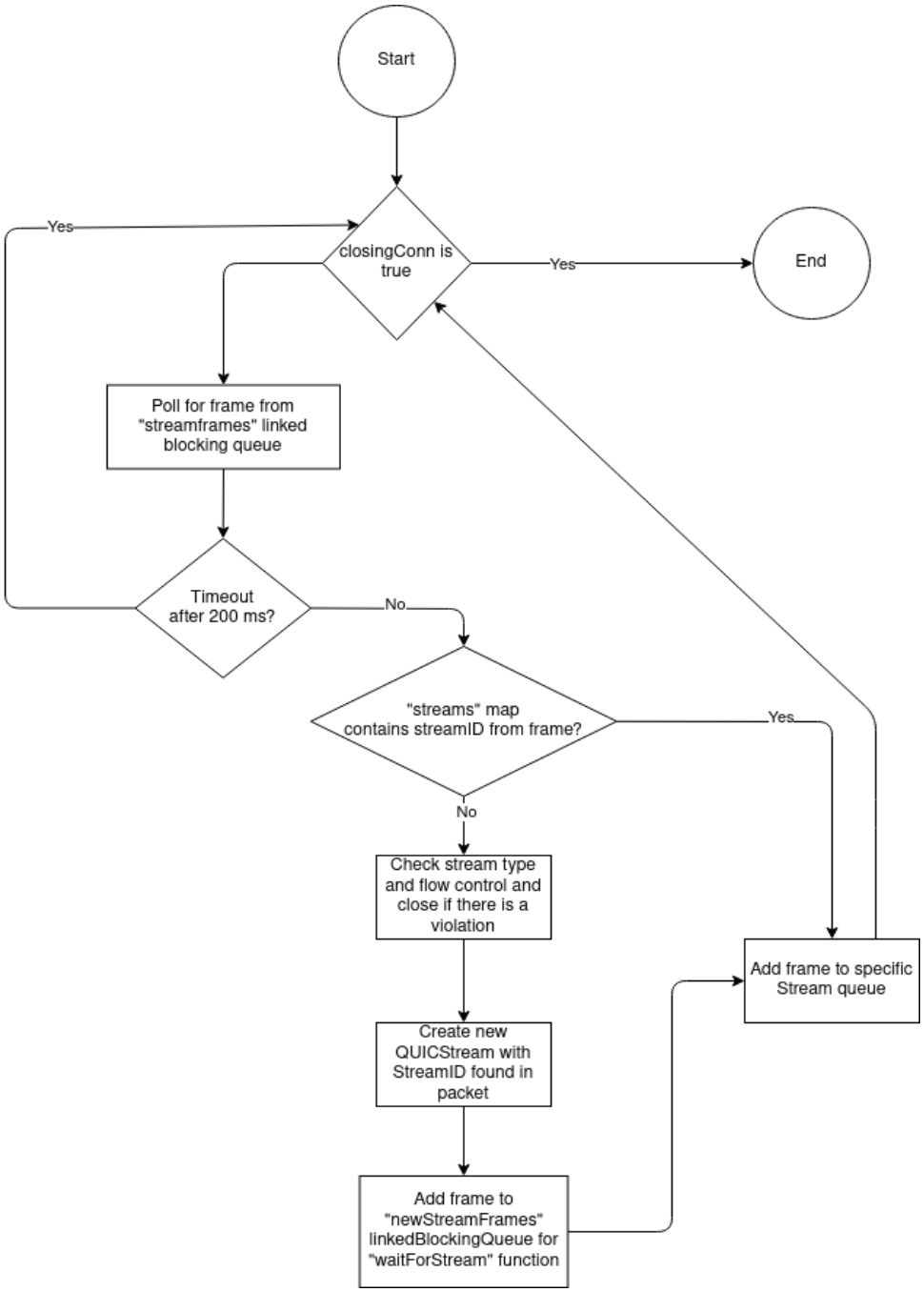
2.5.4.8 LossController



The above diagram is the state diagram for LossController that makes sure that the lost QUIC packets are resent. This state diagram is greatly simplified compared to the implementation to make it more readable. System Architecture, Class and Data Flow Diagrams show more

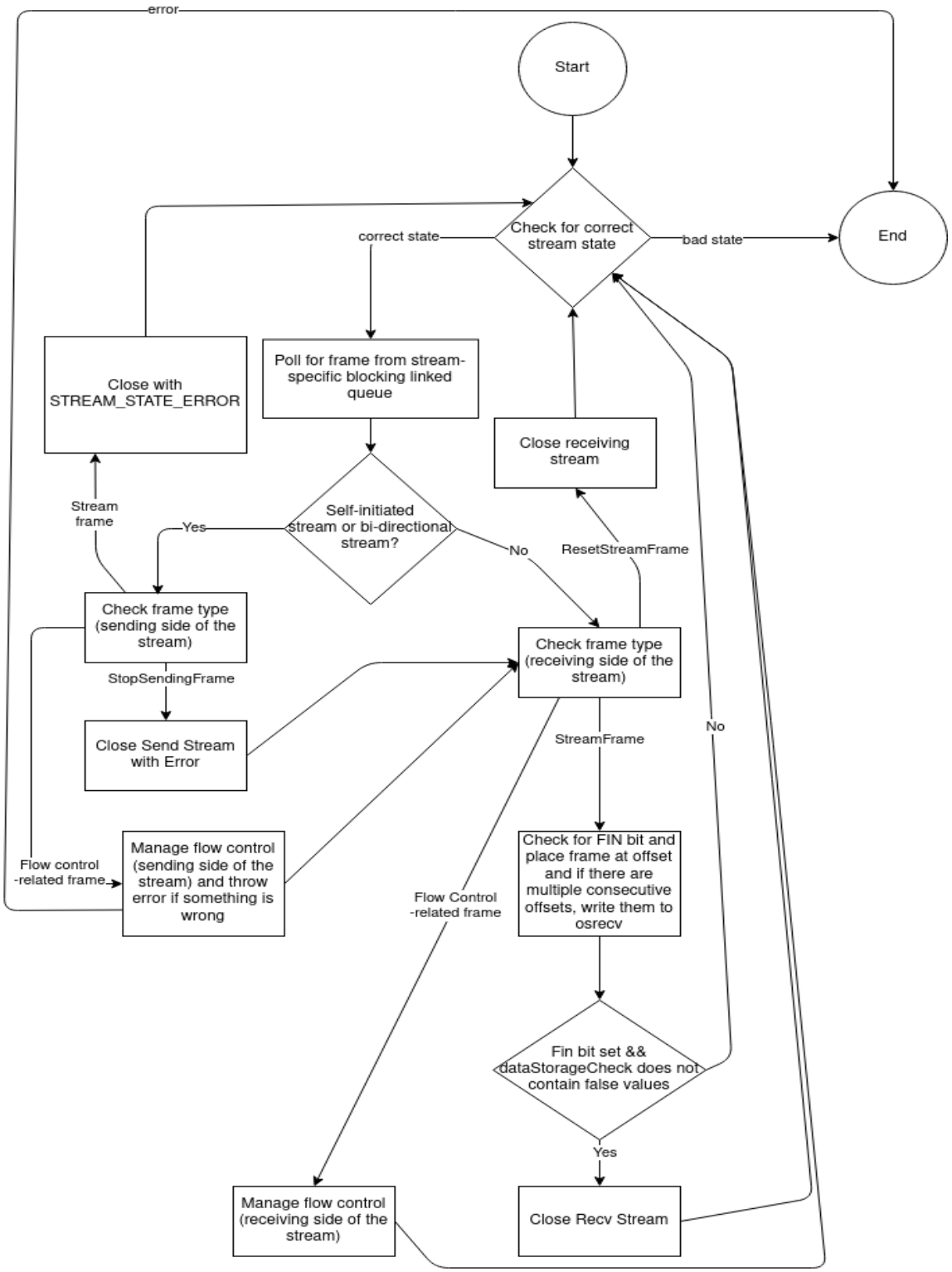
information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.9 StreamController



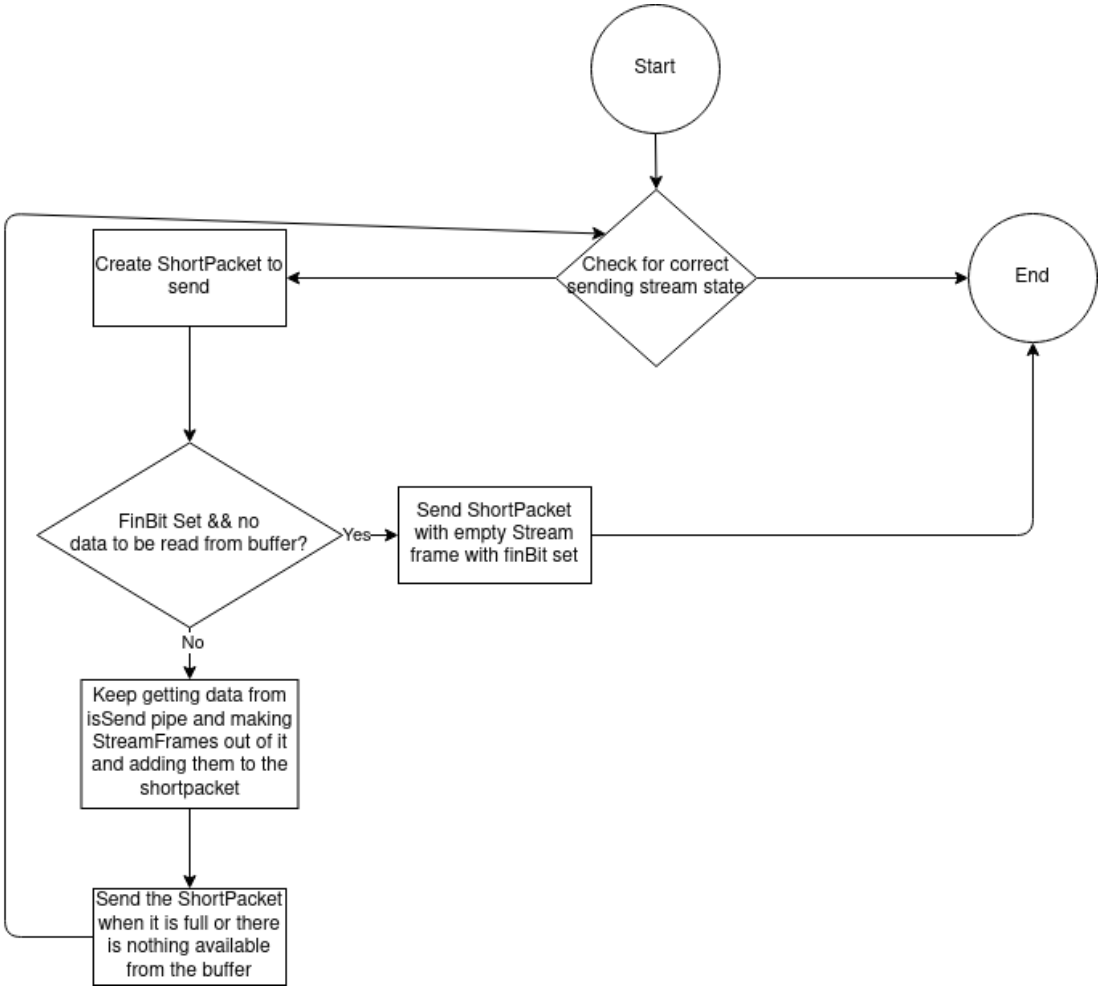
The above diagram is the state diagram for StreamController. Flow control and error-checking has been simplified in the diagram for easier reading. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.10 QUICStreamRecv



The above diagram is the state diagram for QUICStreamRecv. The diagram has been greatly simplified (especially some error checks and flow control) compared to the implementation to save space and easier readability. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

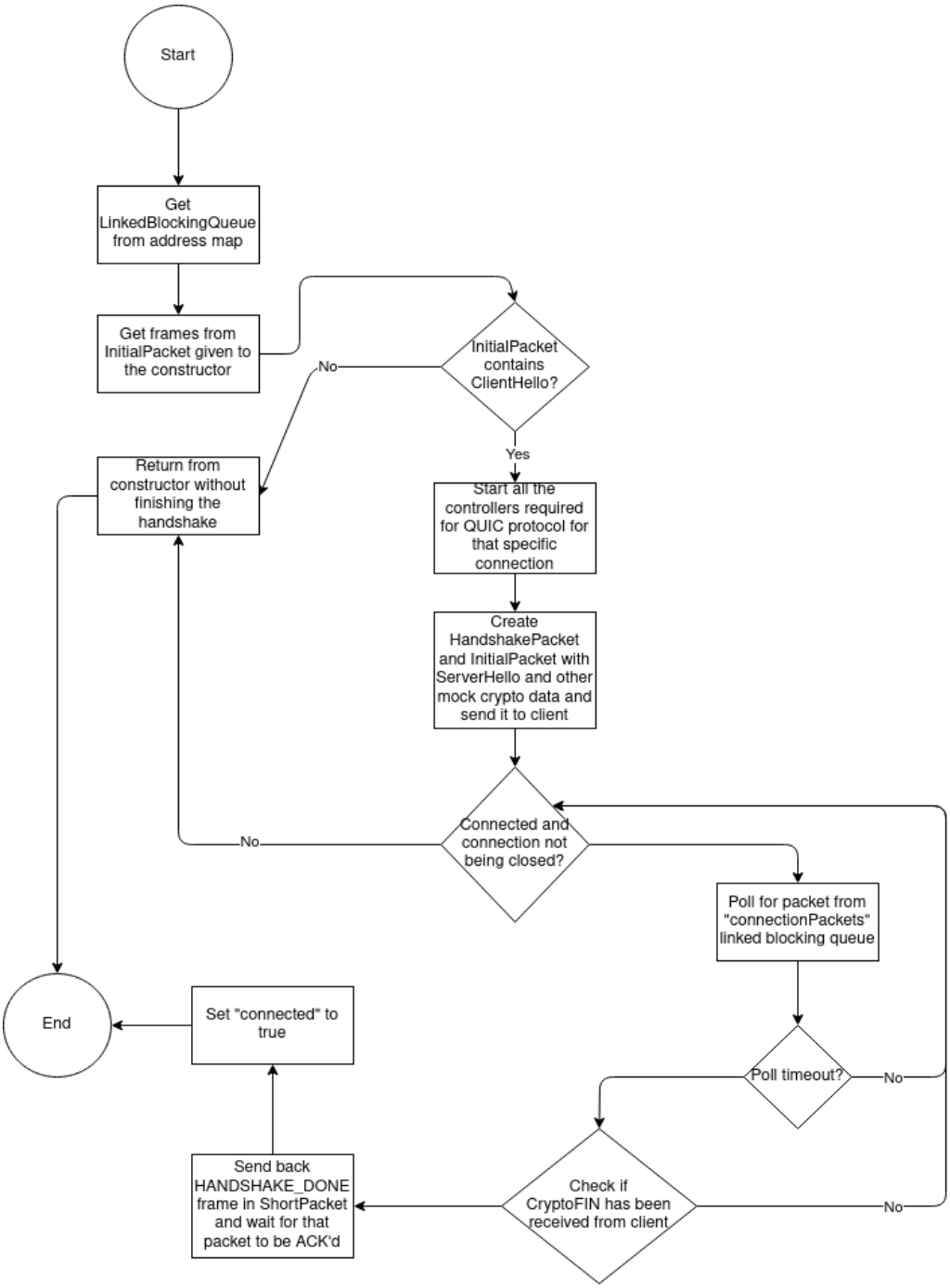
2.5.4.11 QUICStreamSend



The above diagram is the state diagram for QUICStreamSend. The above diagram has been somewhat simplified from the actual implementation for easier readability. System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.12 QUICConnection (Server-Side Constructor)

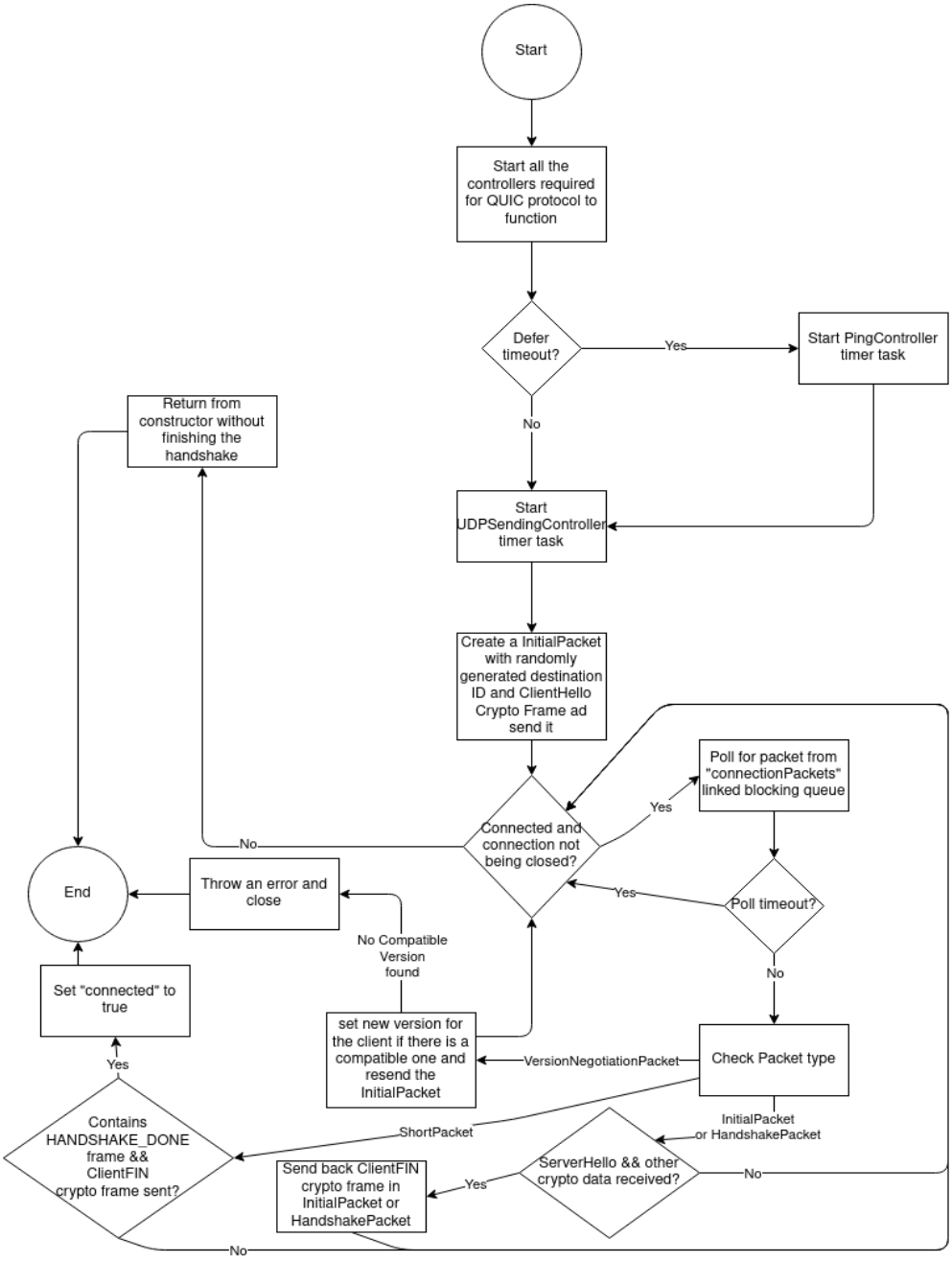
The following state diagram shows QUICConnection when it is constructed from QUICConnectionServer server-side which makes it complete the handshake with QUICConnection on client-side:



System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.13 QUICConnection (“Connect” method)

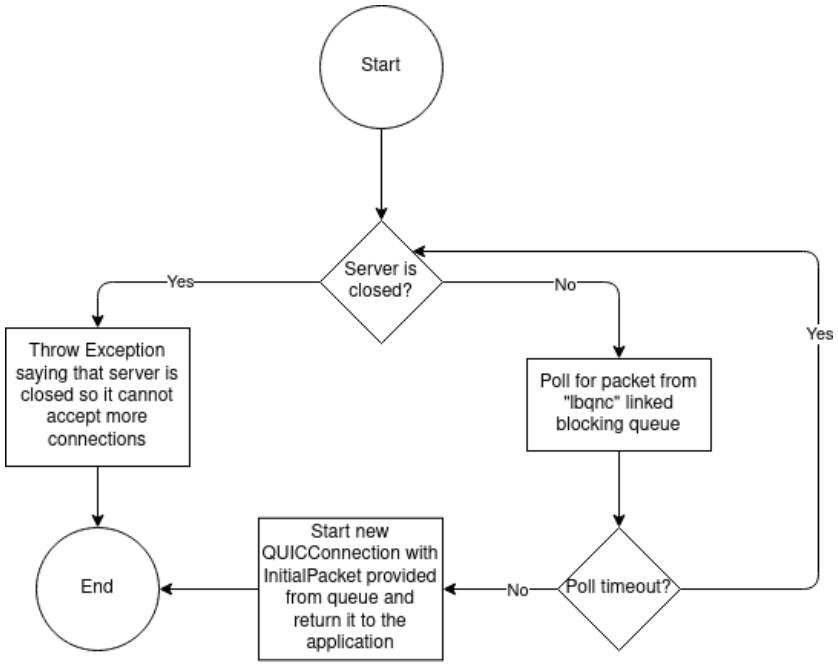
The following state diagram is for the method that QUICConnection uses to connect to QUICConnectionServer from client-side:



System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.4.14 QUICServerConnection (“Accept” Method)

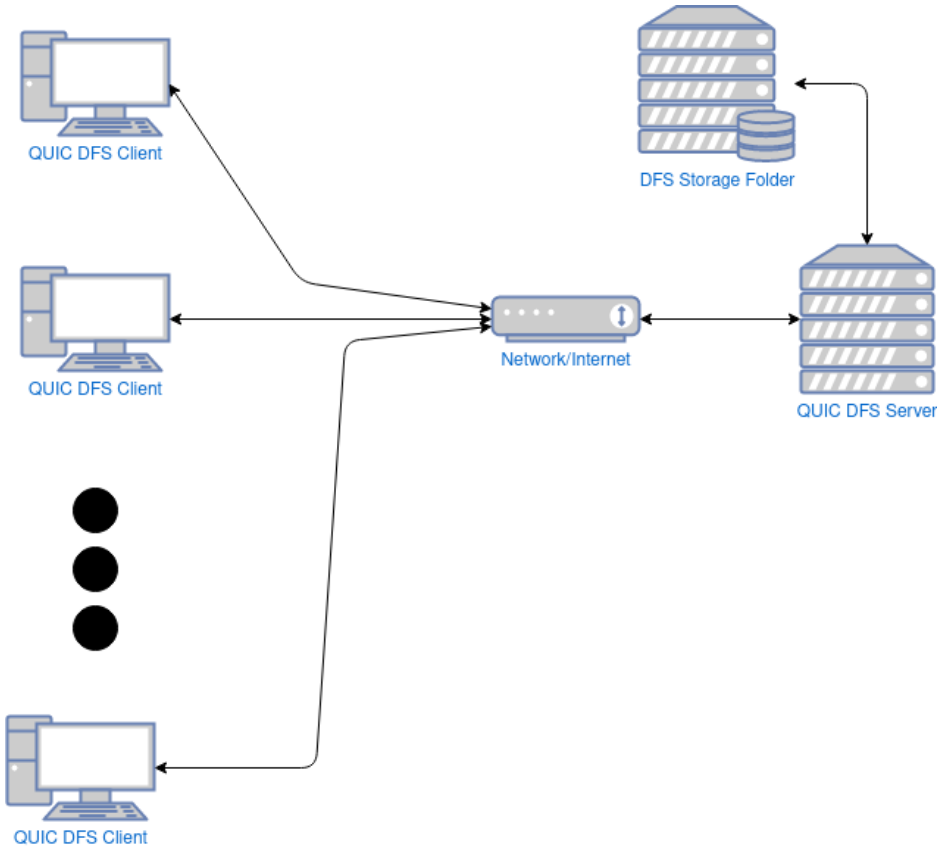
The following diagram is for the method that QUICServerConnection uses to accept clients:



System Architecture, Class and Data Flow Diagrams show more information about this controller as well as how it fits into the overall architecture of this project.

2.5.5 Network Diagram

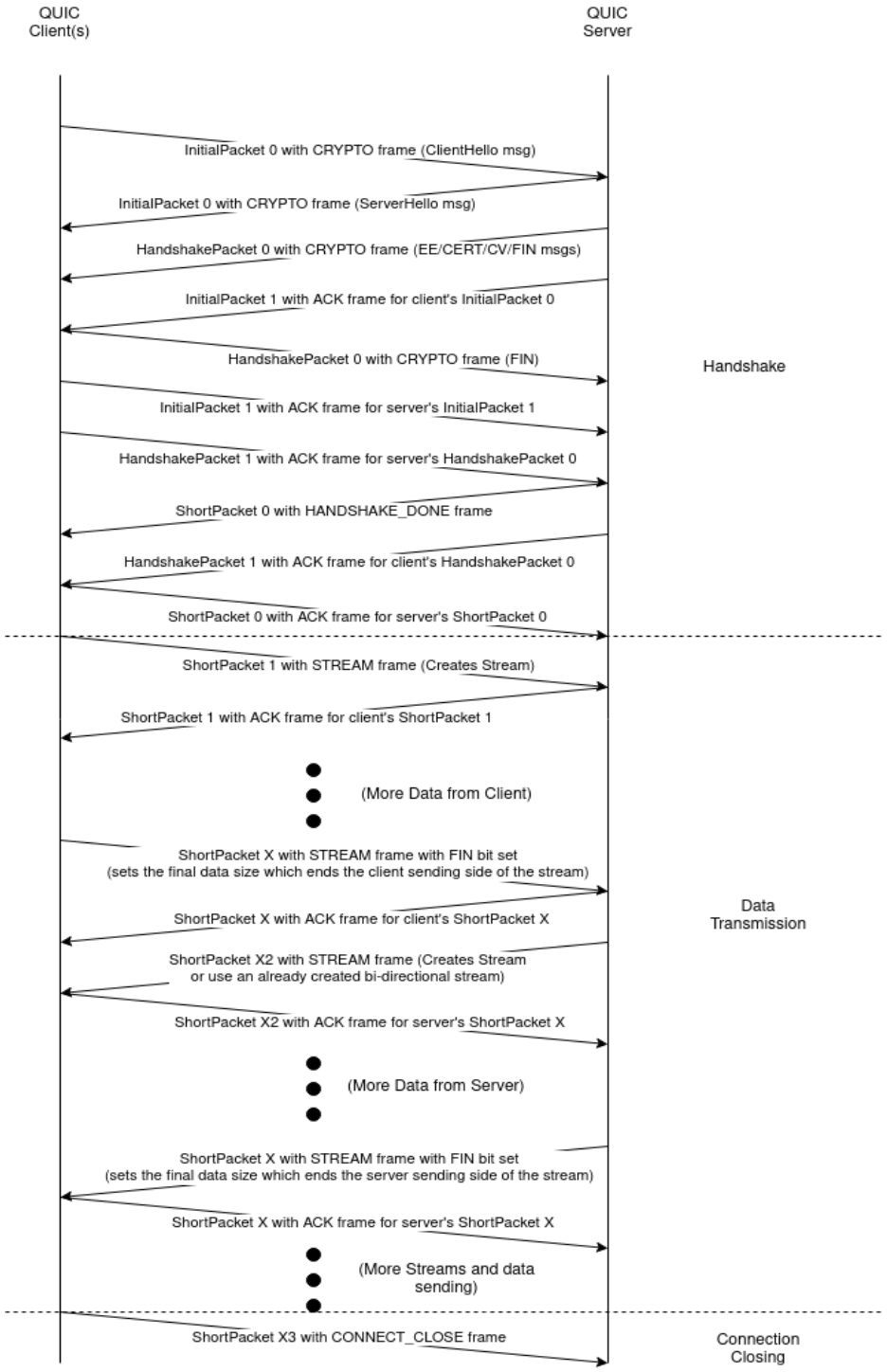
Here is the network diagram for JQUIC DFS Application:



This diagram illustrates that multiple clients can connect to DFS Server at once and send, update and receive files at the same time with some exceptions (as will be shown in Acceptance Tests).

2.5.6 Timing Diagram

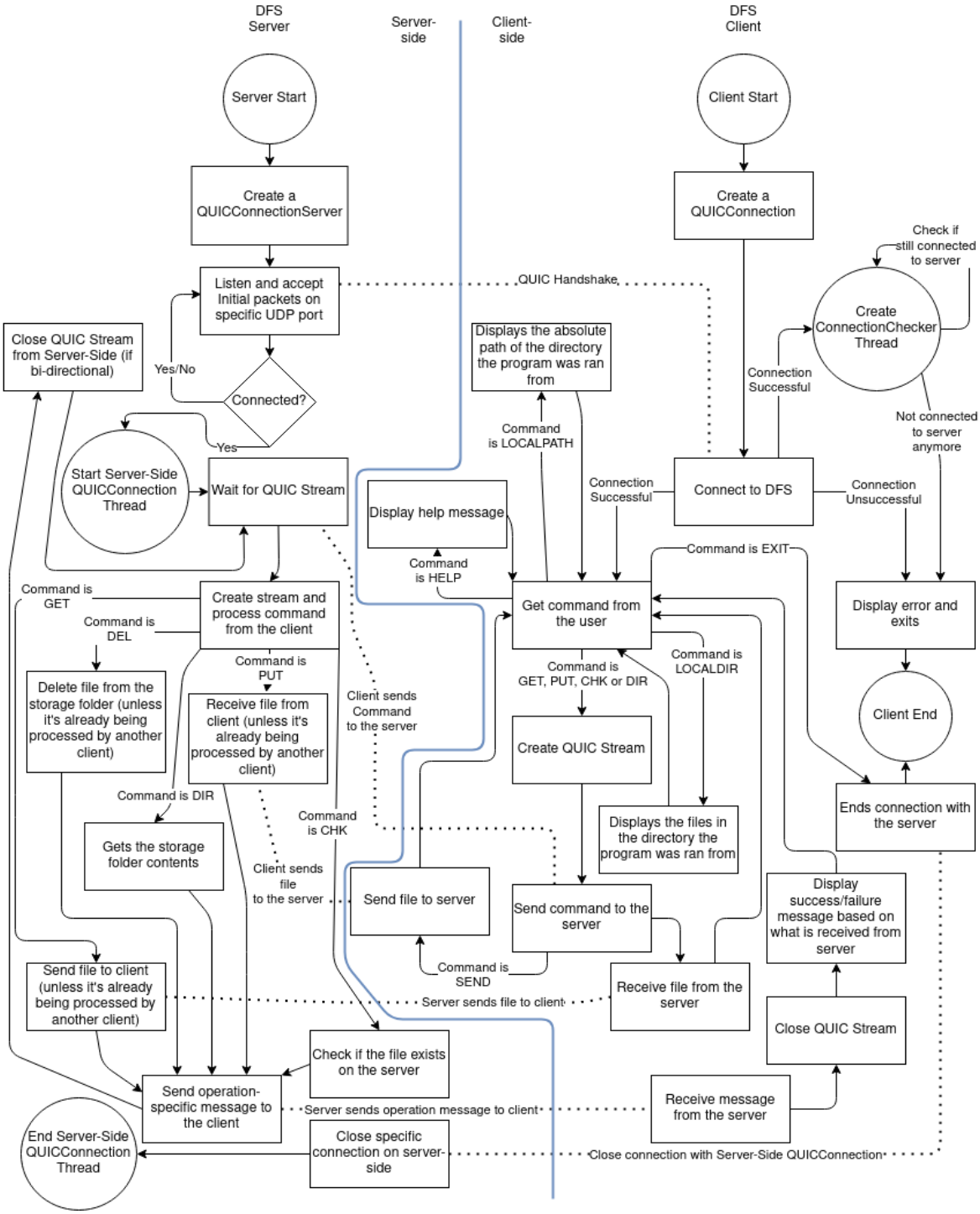
The below diagram shows QUIC client and server interact with each other throughout the connection:



The above diagram shows basic operations of QUIC client and server implementation with no major errors occurring during the connection which could potentially alter this timing diagram significantly as connection could end much earlier or specific streams could be interrupted via RESET_STREAM or STOP_SENDING frames. Notably absent from this diagram is the 0-RTT since they were not implemented and STREAM frames in ShortPackets during the handshake since they were not used which will be discussed in “Implications of Implementation” section of this report.

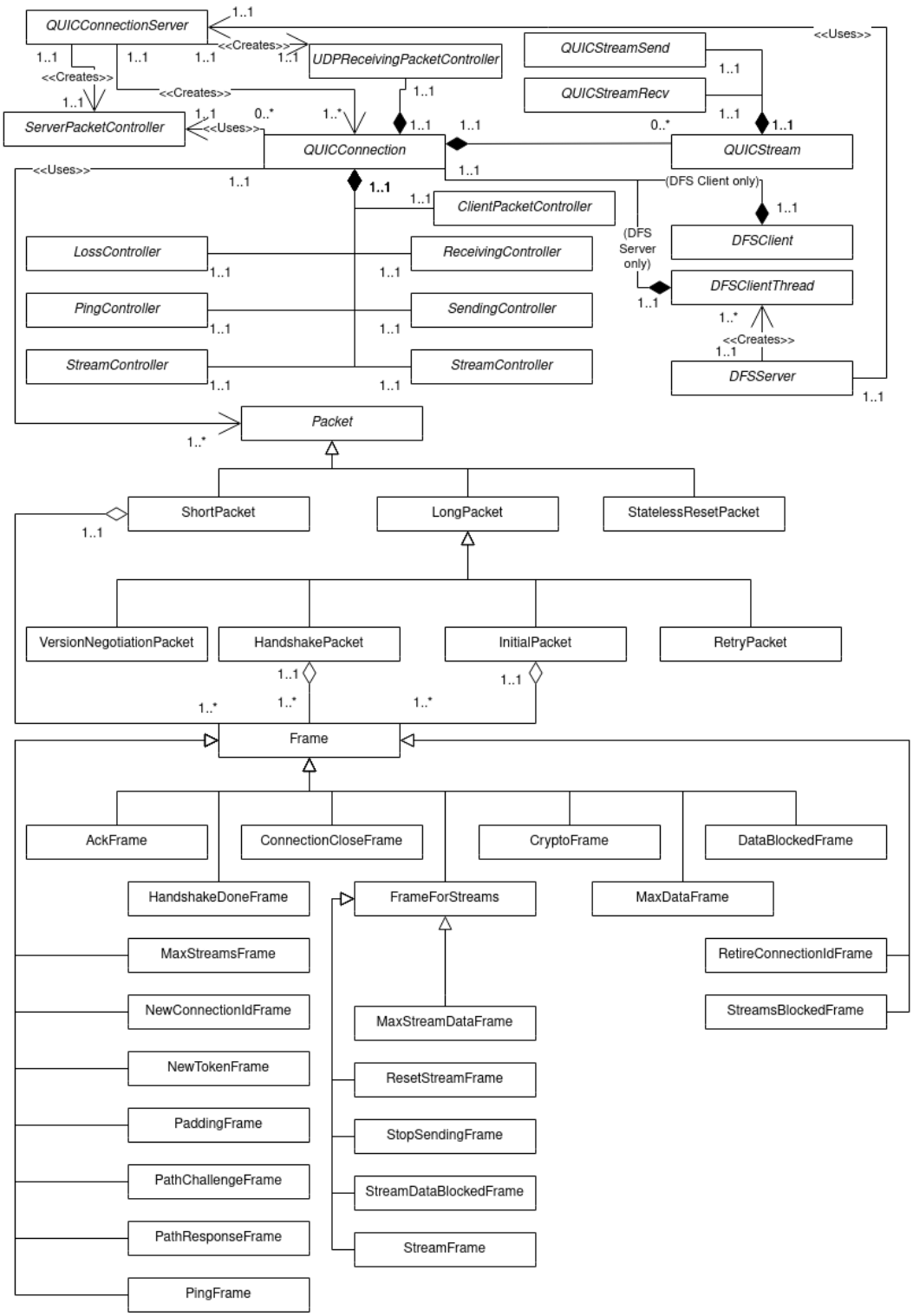
2.5.7 Distributed File Server (DFS) State Diagram

This is the state diagram for the DFS application itself which is very similar to the one in the proposal, but expanded:



Some of the error checks as well as back-and-forth confirmation messages have been omitted to save space in the diagram and make it less complicated. Almost all the paths on this diagram are tested in Acceptance Tests.

2.5.8 Class Diagram



The above diagram is a class diagram that shows how all the different classes in this project connect to each other. The classes in this project can be divided into multiple categories (all of these are shown in the diagram above):

- **Frame**: these classes inherit from parent/grandparent class called Frame; these Frame classes have a one-to-many relationship with HandshakePacket, ShortPacket or InitialPacket classes as per protocol description in the draft, meaning that HandshakePacket/InitialPacket/ShortPacket hold a lot of these Frame classes in an ArrayList to be sent across the network
- **Packet**: these classes inherit from parent/grandparent class called Packet which QUICConnection sends back and forth across network using its controllers as per protocol description in the draft
- **Controllers**: these classes are subclasses of QUICConnection which are used to manage different aspects of QUIC protocol as per protocol description in the draft; For more information, see System Architecture, Data Flow and JQUIC State Diagrams above
- **QUICConnection**: this class facilitates majority of QUIC protocol interactions between the client and its instance on the server using the controllers described above as per protocol description in the draft
- **QUICConnectionServer**: this class facilitates the QUIC protocol interactions when a client first connects to the server which creates a new QUICConnection on the server

2.5.9 API Documentation

The API for QUICConnection and QUICConnectionServer is fairly similar to Java Native Libraries Socket and ServerSocket (albeit somewhat simplified). Note that this is not an exhaustive list of methods, but rather the exposed methods that a user should be using when building an application using this API.

Below table shows the constructor for QUICConnectionServer:

Constructor	**Description**
QUICConnectionServer(int udpPort)	Creates a server QUIC Connection, bound to the specified UDP port

Below table shows the method summary for QUICConnectionServer:

Modifier and Type	**Method**	**Description**
boolean	isClosed()	Returns the closed state of the QUICConnectionServer
boolean	isBound()	Returns the binding state of the QUICConnectionServer
void	close()	Closes this QUICConnectionServer (though not all of the controllers that it created, see “Implications of Implementation” for more details)
QUICConnection	accept()	Listens for a connection being made to this QUICConnectionServer and accepts it

Below table shows the constructors for QUICConnection:

Constructor	**Description**
QUICConnection(boolean defer_timeout)	Creates an unconnected QUICConnection with defer_timeout determining whether the connection should be kept alive (similar to “SO_KEEPALIVE” for TCP)
QUICConnection()	Creates an unconnected QUICConnection with defer_timeout set to false
QUICConnection(InitialPacket initialPacket, ConcurrentHashMap<String, LinkedBlockingQueue<Packet>> IDToQueue, ConcurrentHashMap<String, InetSocketAddress> connIDToAddressMap, DatagramSocket ds, long version)	Creates a QUICConnection that negotiates with the client to create a connection; all the variables are required because it needs to be able to share them with ServerPacketController and UDPPacketController which it does when QUICConnectionServer uses this constructor

Below table shows the method summary for QUICConnection:

Modifier and Type	**Method**	**Description**
void	close(long errorCode, String reason)	Closes this QUICConnection by sending CONNECTION_CLOSE frame to the other side with specific errorCode and reason
void	close()	Closes this QUICConnection
void	connect(SocketAddress Endpoint)	Tries to connect to QUICConnectionServer on certain IP address and UDP port
boolean	isConnected()	Returns the connection state of the QUICConnection
boolean	isClosed()	Returns the closed state of the QUICConnection
boolean	isBound()	Returns the binding state of the QUICConnection
QUICStream	createStream(boolean bidirectional)	Creates and returns a local QUICStream with the boolean argument specifying whether the stream is bi-directional or not
QUICStream	waitForStream()	Waits to receive a QUIC Stream from the other side of the connection
void	sendSinglePacket(Packet packet)	Sends a single QUIC packet through the QUICConnection; used only for debugging, not recommended for production

Below table shows the constructors for QUICStream:

Constructor	**Description**
QUICStream(long streamID)	Creates a QUICStream for a specified QUICConnection with specified streamID

Below table shows the method summary for QUICStream:

Modifier and Type	**Method**	**Description**
OutputStream	getOutputStream()	Returns an output stream for this QUICStream
InputStream	getInputStream()	Returns an input stream for this QUICStream
void	closeStream()	Closes the SENDING side of this QUICStream
void	closeSendStreamWithError(long appErrorCode)	Closes the SENDING side of this QUICStream with an error with other side also being notified
void	closeRecvStreamWithError(long appErrorCode)	Closes the RECEIVING side of this QUICStream with an error with other side also being notified

2.5.10 Installation/User Manual

This section details the installation for Windows and Linux. Windows has not been tested as thoroughly, but does appear to work correctly. It is also not recommended to run a client on a more powerful machine than the server as that could lead to certain instability in DFS application (discussed more in “Implications of Implementation” section). Also note that UDP port that is selected for the server as well as UDP ports 1025-65535 must be open in Firewall rules.

2.5.10.1 Requirements, Installation and Usage

The only requirement on Linux is to have JDK 11 installed. Here are the steps to install JQUIC and DFS application on Linux:

1. Extract the contents provided zip “JQUIC.zip” to any directory
2. Open terminal and navigate to the directory that “JQUIC” folder is located in
3. Run the following command in the terminal: “chmod 755 -R JQUIC”
4. Navigate into “JQUIC” folder in the terminal (Command: “cd JQUIC”)
5. Run the following command in the terminal: “./gradlew”. This will install Gradle on your machine.
6. You can now run the client and the server on your machine

The only requirement on Windows is to have JDK 11 installed with JAVA_HOME environment variable properly configured. Here are the steps to install JQUIC and DFS application on Windows:

1. Extract the contents provided zip “JQUIC.zip” to any directory
2. Open command prompt and navigate to the directory that “JQUIC” folder is located in
3. Navigate into “JQUIC” folder in the command prompt (Command: “cd JQUIC”)
4. Run the following command in the terminal: “gradlew.bat”. This will install Gradle on your machine.
5. You can now run the DFS client and the DFS server on your machine

Here are the steps to run the DFS server after the installation steps above:

1. In the “JQUIC” folder, run the following command: “./gradlew -q --console=plain runServer”. This will provide the usage command for the server. For full usage, the server need two arguments: UDP port to run the JQUIC DFS application server on as well as a storage folder that all the files will be placed in and taken out of by the application clients. For example, if you want to run the server on UDP port 1024 and store in folder “files” located in “JQUIC” directory, you would run with the following the application like this: ‘./gradlew -q --console=plain runServer --args "1024 files"’. The only difference for Windows is that the command does not “./” at the end so to

run the command similar to the previous one, you would run ‘gradlew -q --console=plain runServer --args "1024 files"’ in the command prompt. Also note that the storage folder will be created automatically in the directory that the server was ran from (in case of relative path given) or a different folder if the absolute path was given as the storage folder argument.

Here are the steps to run the DFS client on your machine:

1. In the “JQUIC” folder, run the following command: “./gradlew -q --console=plain runClient”. This will provide the usage command for the server as well as the help message. For full usage, the client needs two arguments: IP address of the server to connect to as well as the UDP port to connect on. For example, if you want to connect to server with IP 192.168.1.72 and UDP port 1024, you run client as ‘./gradlew -q --console=plain runClient --args “192.168.1.72 1024”’. You cannot select specific UDP port for the client at this time. The only difference for Windows is that the command does not “./” at the end so to run the command similar to the previous one, you would run ‘gradlew -q --console=plain runClient --args “192.168.1.72 1024”’ in the command prompt.
2. After you run the program on either Linux or Windows, you will get a command prompt which you can input commands into. To check which commands are available, type “HELP” and press enter. All commands are case sensitive. Here is the list of commands available in the application (this list is identical to the help message):

   • PUT [Local File Path]: uploads the file to the server or edits the one already there if it's not being read or edited

   • GET [Server File]: downloads the file from the server or edits the one already there if it's not being read or edited

   • CHK [Server File] : checks if the file exists on the server

   • DEL [Server File]: deletes the file from the server

   • LOCALPATH: shows what is the current directory and path to it on the client

   • DIR: shows what files are currently available on the server

   • LOCALDIR: shows what files are currently available in the current directory on the client

   • HELP : shows all the available commands

   While on the server, all files are stored directly into the storage folder, client can give relative or absolute path of the directory from where the file is stored on its

local machine. For example, if you want to send file “midSizeTextFile.txt” from directory “files” which is “JQUIC” folder that you are running the application from, you can run the following command: “PUT files/midSizeTextFile.txt” which will upload the file “midSizeTextFile.txt” to the server. Directory uploading does not work properly at this time.

2.6 Testing Details and Results

As discussed in the proposal, the testing for this project consists of a combination of regression, manual and acceptance testing. Manual testing will be done manually by the student and will be mostly used for the protocol implementation to verify that all of the protocol features and requirements are implemented correctly according to the draft. Regression testing will be done for this same purpose, except in an automated way using JUnit testing framework. Acceptance testing will be mostly focused on the Distributed File Server portion of this project and its functionality.

2.6.1 Manual Tests

Manual tests are tests done manually by the student according to the “MUST”’s found in the proposal. “MUST”’s are the requirements in the draft document that the protocol must abide by. While in the proposal, it says that “Regression and Manual Testing will be done by first outlining ALL the draft requirements and translating them into test cases”, this was found not to be very practical due to the following reasons:

- There are requirements for features that are out of scope and as such are not really testable by the student; see the “Implications of Implementation” section for more information on which features are either absent or simplified
- Some of the requirements are not really testable like for example, ‘A sender MUST NOT send any of these frames from a terminal state ("Data Recvd" or "Reset Recvd"). A sender MUST NOT send a STREAM or STREAM_DATA_BLOCKED frame for a stream in the “Reset Sent" state or any terminal state, that is, after sending a RESET_STREAM frame. A receiver could receive any of these three frames in any state, due to the possibility of delayed delivery of packets carrying them’. Note that according to this, receiver could potentially receive those frames anyway so it’s not really possible to test it as a behavior, meaning that it is more of an in-code implementation requirement rather than a testable one. As such, requirements like this will not be tested with manual tests.
- Due to the mocking, simplification or certain implementation of features, some tests become pointless. Like for example, “An endpoint MUST treat the receipt of a frame of

unknown type as a connection error of type FRAME_ENCODING_ERROR." However, the current implementation just discards invalid packets/packets with invalid frames anyway so poorly constructed frames are just discarded instead of closing the whole connection. The reason for this is that the packets are not currently encrypted and as such, it becomes very hard to determine their validity and authenticity.

Majority of Connection ID requirements will also not be implemented as Connection ID's are currently simplified due to lack of connection migration as stated in the scope. Version Negotiation will also only be tested in very basic ways as there were no other versions of the protocol implemented other than draft version 34. Around 50 "MUST"'s in the draft document that could be tested for the current implementation were selected and captured in the following 38 tests:

No.	Draft Requirement	Test	Passing Criteria
1	"Endpoints MUST be able to deliver stream data to an application as an ordered byte-stream"	Endpoint 1 sends multiple data to Endpoint 2	Bunch of STREAM frames will be shown to be sent and received with their offsets and lengths
2	"An endpoint could receive data for a stream at the same stream offset multiple times. Data that has already been received can be discarded. The data at a given offset MUST NOT change if it is sent multiple times; an endpoint MAY treat receipt of different data at the same offset within a stream as a connection error of type PROTOCOL_VIOLATION."	Endpoint 1 changes the data in the STREAM frame and sends at the same stream offset as another STREAM frame to Endpoint 2 while in "Send" state.	Endpoint 1 receives back a CONNECTION_CLOSE frame with error type PROTOCOL_VIOLATION from Endpoint 2 and closes connection.
3	"An endpoint MUST NOT send data on any stream without ensuring that it is within the flow control limits set by its peer… Senders MUST NOT send data in excess of [Stream flow control or Connection flow control] limit."	Endpoint 1 sends more data to Endpoint 2 than initial flow control limit	Endpoint 1 sends DATA_BLOCKED frame to Endpoint 2 once limit is reached OR Endpoint 2 sends MAX_DATA frame to Endpoint 1 before updating the limit

4	“Before a stream is created, all streams of the same type with lower-numbered stream IDs MUST be created.”	Multiple streams will be created between Endpoint 1 and Endpoint 2	It will be shown that they are created in numerically sequential order
5	“An endpoint that receives a STOP_SENDING frame MUST send a RESET_STREAM frame if the stream is in the Ready or Send state.”	A stream is created on Endpoint 1 with Endpoint 2 waiting for a stream and a STOP_SENDING frame gets sent to Endpoint 1	Endpoint 2 receives a RESET_STREAM frame for the same stream as it sent the STOP_SENDING frame for
6	“A receiver MUST close the connection with a FLOW_CONTROL_ERROR error if the sender violates the advertised connection or stream data limits… A receiver MUST NOT wait for a STREAM_DATA_BLOCKED or DATA_BLOCKED frame before sending a MAX_STREAM_DATA or MAX_DATA frame”	Endpoint 1 keeps sending data to Endpoint 2 past the connection flow control limit	Endpoint 1 received back MAX_DATA and MAX_STREAM_DATA frames to increase the flow control parameters with asking for it
7	“A sender MUST ignore any MAX_STREAM_DATA or MAX_DATA frames that do not increase flow control limits.”	Endpoint 1 creates stream with Endpoint 2; Endpoint 2 sends a MAX_DATA and MAX_STREAM_DATA frames to Endpoint 1 with limits smaller than initial limits	Endpoint 1 does not change its connection data limit
8	“Both endpoints MUST maintain flow control state for the stream in the unterminated direction until that direction enters a terminal state.”	Endpoint 1 creates a bidirectional stream with Endpoint 2 and terminates its sending part of it	Both Endpoint 1 and 2 have the Endpoint 2-to-Endpoint 1 direction of the stream ready to send
9	“Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a frame with a stream ID exceeding the limit it has	Endpoint 1 keeps creating more and more streams to Endpoint 2	Endpoint 1 received back MAX_STREAMS frames to increase the flow control parameters

	sent MUST treat this as a connection error of type STREAM_LIMIT_ERROR… An endpoint MUST NOT wait to receive this signal before advertising additional credit, since doing so will mean that the peer will be blocked for at least an entire round trip, and potentially indefinitely if the peer chooses not to send STREAMS_BLOCKED frames.”		
10	“Receiver MUST ignore any MAX_STREAMS frame that does not increase the stream limit”	Endpoint 2 sends a MAX_STREAMS frame to Endpoint 1 with limit smaller than initial limit	Endpoint 1 does not change its connection stream limit
11	“If a client receives a packet that uses a different version than it initially selected, it MUST discard that packet.”	Endpoint 2 tries to send a packet with different version than Endpoint 1 has set and it’s not a Version Negotiation Packet	Endpoint 1 discards the packet
12	“A client MUST discard any Version Negotiation packet if it has received and successfully processed any other packet, including an earlier Version Negotiation packet.”	Endpoint 2 tries to send Version Negotiation Packet to Endpoint 1 after the connection has been established (meaning that other packets have already been sent)	Endpoint 1 discards the version negotiation packet
13	“A client MUST discard a Version Negotiation packet that lists the QUIC version selected by the client.”	Endpoint 2 sends a Version Negotiation Packet with the same version as the one Endpoint 1 tried to initiate the connection with	Endpoint 1 discards the received Version Negotiation Packet

14	“The client MUST check that the Destination and Source Connection ID fields [of Version Negotiation Packet] match the Source and Destination Connection ID fields in a packet that the client sent. If this check fails, the packet MUST be discarded.”	Source and Destination Fields are changed by Endpoint 2 before sending the packet to endpoint 1	Endpoint 1 discards the received packet
15	“A client MUST change the Destination Connection ID it uses for sending packets in response to only the first received Initial or Retry packet. A server MUST set the Destination Connection ID it uses for sending packets based on the first received Initial packet. Any further changes to the Destination Connection ID are only permitted if the values are taken from NEW_CONNECTION_ID frames; if subsequent Initial packets include a different Source Connection ID, they MUST be discarded.“	Source Connection ID is changed for new initial packet during handshake when sending from Endpoint 1 to Endpoint 2	Endpoint 2 discards the received packet
16	“After the handshake is confirmed, an endpoint MUST send any CONNECTION_CLOSE frames in a 1-RTT packet.”	Endpoint 1 connected to Endpoint 2 and then closes connection	The CONNECTION_CLOSE packet is sent in 1-RTT (Short) Packet.
17	“Endpoints MUST send stateless reset packets formatted as a packet with a short header. However, endpoints MUST treat any packet ending in a valid stateless reset token as a stateless reset, as other QUIC versions might allow the use of a long header.”	Endpoint 2 triggers a Endpoint 1 to send a stateless reset packet to it.	The form of stateless reset packet is short. Endpoint 2 confirms that it is a stateless reset packet based on the stateless reset token.
18	“Receivers MUST be able to	Endpoint 1 sends	Endpoint 2 is able to individually

	process coalesced packets…The receiver of coalesced QUIC packets MUST individually process each QUIC packet and separately acknowledge them, as if they were received as the payload of different UDP datagrams”	multiple packets in same batch to Endpoint 2	process the packets sent by Endpoint 1
19	“The payload of a packet that contains frames MUST contain at least one frame, and MAY contain multiple frames and multiple frame types. An endpoint MUST treat receipt of a packet containing no frames as a connection error of type PROTOCOL_VIOLATION. “	Endpoint 1 sends a packet with no frames to Endpoint 2	Endpoint 2 closes the connection with Endpoint 1 with connection error of type PROTOCOL_VIOLATION.
20	“An endpoint MUST treat receipt of a frame in a packet type that is not permitted as a connection error of type PROTOCOL_VIOLATION… The payload of [handshake] packet contains CRYPTO frames and could contain PING, PADDING, or ACK frames. Handshake packets MAY contain CONNECTION_CLOSE frames of type 0x1c. Endpoints MUST treat receipt of Handshake packets with other frames as a connection error of type PROTOCOL_VIOLATION.”	Endpoint 1 sends a Handshake Packet with a stream frame to Endpoint 2	Endpoint 2 closes the connection with Endpoint 1 with connection error of type PROTOCOL_VIOLATION.
21	“In IPv4, the DF bit MUST be set if possible, to prevent fragmentation on the path.”	Connection is made between endpoint 1 and endpoint 2 and a 1MB file is set	All the IPv4 packets for the transmission have DF bit set
22	“A client MUST expand the payload of all UDP datagrams carrying Initial packets to at least the smallest allowed maximum	Connection is made between endpoint 1 and endpoint 2	All the packets that contain Initial packets have PADDING frames to get the same of the payload to 1200 bytes

	datagram size of 1200 bytes by adding PADDING frames to the Initial packet or by coalescing the Initial packet.”		
23	“A server MUST discard an Initial packet that is carried in a UDP datagram with a payload that is smaller than the smallest allowed maximum datagram size of 1200 bytes.”	Endpoint 1(Client) will send a UDP datagram, containing Initial packet, that is smaller than the smallest allowed maximum datagram size of 1200 bytes to Endpoint 2 (Server).	Endpoint 2 discards the packet sent by Endpoint 1.
24	“Packets containing a zero value for [fixed bit] are not valid packets in this version and MUST be discarded.”	Endpoint 1 sends a ShortPacket to Endpoint 2 with fixed bit set to 0	Endpoint 2 discards the packet.
25	“In QUIC version 1, [Destination Connection ID Length] MUST NOT exceed 20. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet.”	Endpoint 1 sends a HandshakePacket to Endpoint 2 with Destination Connection ID length 21	Endpoint 2 discards the packet received from Endpoint 1.
26	“In QUIC version 1, [Source Connection ID Length] MUST NOT exceed 20 bytes. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet.”	Endpoint 1 sends a HandshakePacket to Endpoint 2 with Source Connection ID length 21	Endpoint 2 discards the packet received from Endpoint 1.
27	“An endpoint MUST treat receipt of a packet that has a non-zero value for [Reserved bits in Long Packet] after removing both packet and header protection as a connection error of type PROTOCOL_VIOLATION.”	Endpoint 1 sends a HandshakePacket to Endpoint 2 with Reserved bits not being 0.	Endpoint 2 closes the connection with Endpoint 1 with connection error of type PROTOCOL_VIOLATION.

28	“Initial packets sent by the server MUST set the Token Length field to zero; clients that receive an Initial packet with a non-zero Token Length field MUST either discard the packet or generate a connection error of type PROTOCOL_VIOLATION.”	Endpoint 1 (Server) sends an Initial Packet to Endpoint 2 (Client) token with Token Length field of 2.	Endpoint 2 closes the connection with Endpoint 1 with connection error of type PROTOCOL_VIOLATION.
29	“If the frame type is 0x03, ACK frames also contain the cumulative count of QUIC packets with associated ECN marks received on the connection up until this point. QUIC implementations MUST properly handle both types.”	Endpoint 1 sends an Initial Packet with ACK frame that contains ECN counts to Endpoint 2.	Endpoint 2 correctly parses the ACK frame with ECN counts received from Endpoint 1.
30	“If any computed packet number is negative, an endpoint MUST generate a connection error of type FRAME_ENCODING_ERROR.”	Endpoint 1 sends an Short Packet with ACK frame that would result in negative packet number.	Endpoint 2 closes the connection with Endpoint 1 with connection error of type FRAME_ENCODING_ERROR.
31	“An endpoint that receives a RESET_STREAM frame for a send-only stream MUST terminate the connection with error STREAM_STATE_ERROR.”	Endpoint 1 creates a send-only stream with Endpoint 2; Endpoint 2 sends RESET_STREAM frame to Endpoint 1.	Endpoint 1 terminates the connection with Endpoint 2 with error STREAM_STATE_ERROR.
32	“Receiving a STOP_SENDING frame for a locally-initiated stream that has not yet been created MUST be treated as a connection error of type STREAM_STATE_ERROR.”	Endpoint 1(Client) sends STOP_SENDING frame for non-existent locally-initiated stream on Endpoint 2 (Server).	Endpoint 2 terminates the connection with Endpoint 1 with error STREAM_STATE_ERROR.
33	“An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error STREAM_STATE_ERROR.”	Endpoint 1 creates a send-only stream with Endpoint 2; Endpoint 1 sends STOP_SENDING frame to Endpoint 1.	Endpoint 2 terminates the connection with Endpoint 1 with error STREAM_STATE_ERROR.
34	“Clients MUST NOT send	Endpoint 1 (Client)	Endpoint 2 terminates the connection

	NEW_TOKEN frames. A server MUST treat receipt of a NEW_TOKEN frame as a connection error of type PROTOCOL_VIOLATION. “	sends a Short Packet with NEW_TOKEN frame to Endpoint 2 (Server).	with Endpoint 1 with error PROTOCOL_VIOLATION.
35	“An endpoint MUST terminate the connection with error STREAM_STATE_ERROR if it receives a STREAM frame for a locally-initiated stream that has not yet been created…”	Endpoint 1 (Client) sends STREAM frame for non-existent locally-initiated stream on Endpoint 2 (Server).	Endpoint 2 terminates the connection with Endpoint 1 with error STREAM_STATE_ERROR.
36	“An endpoint MUST terminate the connection with error STREAM_STATE_ERROR if it receives a STREAM frame… for a send-only stream.”	Endpoint 1 creates a send-only stream with Endpoint 2; Endpoint 2 sends STREAM frame to Endpoint 1.	Endpoint 1 terminates the connection with Endpoint 2 with error STREAM_STATE_ERROR.
37	“An endpoint that receives a STREAM_DATA_BLOCKED frame for a send-only stream MUST terminate the connection with error STREAM_STATE_ERROR.”	Endpoint 1 creates a send-only stream with Endpoint 2; Endpoint 2 sends STREAM_DATA_BLOCKED frame to Endpoint 1.	Endpoint 1 terminates the connection with Endpoint 2 with error STREAM_STATE_ERROR.
38	“A server MUST treat receipt of a HANDSHAKE_DONE frame as a connection error of type PROTOCOL_VIOLATION.”	Endpoint 1 (Client) sends a Short Packet with HANDSHAKE_DONE frame to Endpoint 2 (Server).	Endpoint 2 terminates the connection with Endpoint 1 with error PROTOCOL_VIOLATION.

2.6.1.1 Manual Test Results

Here are the results for each test:

No.	Results (Screenshots, Packet Captures, etc.)	Verdict
0	This is not really a test and just shows what test environment consists of. Basically, one Endpoint connects to another and stays connected: Server: `QUICConnectionServer quicConnectionServer = new QUICConnectionServer( udpPort: 1024);` `try {` `  QUICConnection client = quicConnectionServer.accept();` `  if(client.isConnected()) {` `  System.out.println("New Client Connected: ID " + Base64.getEncoder().encodeToString(client.getSourceConnectionID()));` Client: `QUICConnection qs = new QUICConnection( defer_timeout: true);` `try {` `  /// Test 23` `  GlobalConstants.MAX_UDP_PAYLOAD_SIZE = 1199;` `  qs.connect(new InetSocketAddress( hostname: "127.0.0.1", port: 1024));` `  if(qs.isConnected()) {` `  System.out.println("Connected to Server " + Base64.getEncoder().encodeToString(qs.getDestConnectionID()) + ".");` Output for Server: `New Client Connected: ID PycLUsUsFwh7g25cSIqyfqIEPl0=` Output for Client:	-

	`` Connected to Server PycLUsUsFwh7g25cSIqyfqIEPl0=. ``` There is also printing by different threads like the ReceivingController which prints output for specific tests or all received packets.	
1	This is the code running on Endpoint 2 that receives the data from Endpoint 1: ``` /// Test 1 QUICConnection.QUICStream quicStream = qs.waitForStream(); InputStream input = quicStream.getInputStream(); BufferedReader reader = new BufferedReader(new InputStreamReader(input)); String line = reader.readLine(); try { for (int i = 0; i < 9; i++) { System.out.println(line); line = reader.readLine(); } } catch (IOException e) { e.printStackTrace(); if(quicStream.getRecvErrorCode() != 0) { System.out.println("Receiving Error code: " + quicStream.getRecvErrorCode()); } } ``` This is the code running on Endpoint 1 that sends the data to Endpoint 2:	**PASS**

```java
/// Test 1
QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);

OutputStream output = quicStream.getOutputStream();
PrintWriter writer = new PrintWriter(output,  autoFlush: true);
for(int i = 0; i < 10; i++) {
    writer.println("This is a message sent to the client");
}

quicStream.closeStream();
```

Below screenshot shows that Endpoint 2 received data from endpoint 1 in two different STREAM frames with offset in second frame's offset same as the length of the first frame:

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): +T+zwW4DFtPsmrmwzJ0h62V65/8=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: STREAM (LEN Bit Set)
StreamID: 3
Length: 37
Fin Bit: Not Set
Data: [84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32,
 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10]

Frame 2
-------
Type: STREAM (FIN, LEN, OFF Bit Set)
StreamID: 3
Offset: 37
Length: 333
Fin Bit: Set
Data: [84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32,
 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101,
  32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32, 105, 115,
 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105, 101, 110,
  116, 10, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32,
  116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103,
 101, 32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32, 105,
 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105, 101,
  110, 116, 10, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116,
 111, 32, 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 109, 101, 115, 115, 97,
 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105, 101, 110, 116, 10, 84, 104, 105, 115, 32,
 105, 115, 32, 97, 32, 109, 101, 115, 115, 97, 103, 101, 32, 115, 101, 110, 116, 32, 116, 111, 32, 116, 104, 101, 32, 99, 108, 105,
  101, 110, 116, 10]


}
```

This shows that the data is represented as ordered byte stream.

2	This code shows Endpoint 1 sending a 1-RTT packet with two STREAM frames that have different data at the same offset to Endpoint 2: `/// Test 2` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "goodbye".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "goodbye".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` This code shows Endpoint 2 just creating a stream so that it can receive the stream frames from Endpoint 1: `/// Test 2` `QUICConnection.QUICStream quicStream = qs.waitForStream();` The result is that connection 1 receives a CONNECTION_CLOSE frame with PROTOCOL_VIOLATION error (0x0A): `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): koMi0Tsx7zKl9rI1mNpxJjETZY8=` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 3` `Frames:` `{` `Frame 1` `-------` `Type: CONNECTION_CLOSE (Application Specific)` `Error Code: 10` `Reason Phrase Length: 29` `Reason Phrase: Different data at same offset` `}` `Client ID koMi0Tsx7zKl9rI1mNpxJjETZY8= Disconnected with Error Code 0x0A (PROTOCOL_VIOLATION): Different data at same offset.`	**PASS**
3	Both endpoint 1 and 2 use the same code as for test 1 for this test. Here are the initial limits for endpoint 1 and 2 (assume that these limits are the same for the other tests as well):	**PASS**

```
public static int INITIAL_MAX_DATA = 100;
public static int INITIAL_MAX_STREAM_DATA_BIDI_LOCAL = 10;
public static int INITIAL_MAX_STREAM_DATA_BIDI_REMOTE = 10;
public static int INITIAL_MAX_STREAM_DATA_UNI = 10;
```

Endpoint 1 sends DATA_BLOCKED frame once it reaches this limit:

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): tZWO4Qv8MIDCVtQrsTlS/cdv0pk=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 3
Frames:
{

Frame 1
-------
Type: DATA_BLOCKED
Maximum Data: 100


}
```

Endpoint 2 then increases its limit by sending MAX_DATA frame:

```
New Client Connected: ID sYoqxYnonxBTznRxAyD4at5Fi6o=
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): sYoqxYnonxBTznRxAyD4at5Fi6o=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: MAX_DATA
Maximum Data: 501



}
```

Thus, Endpoint 1 is able to wait for Endpoint 2 before increasing its limit as to not go over the flow control limit.

4	Code for Endpoint 1 (Server) that creates 8 streams (4 Uni-Directional and 4 Bi-Directional): `/// Test 4` `for(int i = 0; i < 4; i++ ) {` `  QUICConnection.QUICStream quicStreamUni = client.createStream( bidirectional: false);` `  System.out.println("New Uni-Directional Stream ID (Server): " + quicStreamUni.getStreamID());` `  QUICConnection.QUICStream quicStreamBidi = client.createStream( bidirectional: true);` `  System.out.println("New Bi-Directional Stream ID (Server): " + quicStreamBidi.getStreamID());` `}` Code for Endpoint 2 (Client) that creates 8 streams (4 Uni-Directional and 4 Bi-Directional):	**PASS**

```
/// Test 4
for(int i = 0; i < 4; i++ ) {
    QUICConnection.QUICStream quicStreamUni = qs.createStream( bidirectional: false);
    System.out.println("New Uni-Directional Stream ID (Client): " + quicStreamUni.getStreamID());
    QUICConnection.QUICStream quicStreamBidi = qs.createStream( bidirectional: true);
    System.out.println("New Bi-Directional Stream ID (Client): " + quicStreamBidi.getStreamID());
}
```

Output for Endpoint 1 showing increasing Stream IDs:

```
New Client Connected: ID LE5l03V6Cp+J3dikgT0DPIKDOK8=
New Uni-Directional Stream ID (Server): 3
New Bi-Directional Stream ID (Server): 1
New Uni-Directional Stream ID (Server): 7
New Bi-Directional Stream ID (Server): 5
New Uni-Directional Stream ID (Server): 11
New Bi-Directional Stream ID (Server): 9
New Uni-Directional Stream ID (Server): 15
New Bi-Directional Stream ID (Server): 13
```

Output for Endpoint 2 showing increasing Stream IDs:

```
New Uni-Directional Stream ID (Client): 2
New Bi-Directional Stream ID (Client): 0
New Uni-Directional Stream ID (Client): 6
New Bi-Directional Stream ID (Client): 4
New Uni-Directional Stream ID (Client): 10
New Bi-Directional Stream ID (Client): 8
New Uni-Directional Stream ID (Client): 14
New Bi-Directional Stream ID (Client): 12
```

5	Code for Endpoint 1 that creates a Uni-Directional Stream and sends some data to Endpoint 2 (as streams are only on receiving side when some data is received on them): `/// Test 5` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` Code for Endpoint 2 that waits for stream and then closes it from the receiving side by sending STOP_SENDING frame:	**PASS**

```
/// Test 5
QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);
OutputStream output = quicStream.getOutputStream();
PrintWriter writer = new PrintWriter(output, autoFlush: true);
writer.println("This is a message sent to the client");
```

STOP_SENDING frame received by Endpoint 1:

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): FkVfukrEzDH88ZMEFxog27t+JDU=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: STOP_SENDING
StreamID: 3
Application Protocol Error Code: 1


}
```

RESET_FRAME frame received by Endpoint 2 in response to STOP_SENDING frame (with final size of stream shown):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): MyzvfoInV/xJf0U3IsiWkbdQ3m8=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 5
Frames:
{

Frame 1
-------
Type: RESET_STREAM
StreamID: 3
Application Protocol Error Code: 1
Final Size: 37


}
```

6	Code for Endpoint 1 to keep sending data without stopping and updating the flow control limits: `/// Test 6` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `long offset = 0;` `while(true) {` `  ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `  sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset, "hello".getBytes(StandardCharsets.UTF_8).length,` `  fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `  client.sendSinglePacket(sp);` `  offset += "hello".getBytes(StandardCharsets.UTF_8).length;` `  Thread.sleep( l: 5);` `}` Code for Endpoint 2 to just create the Stream: `/// Test 6` `QUICConnection.QUICStream quicStream = qs.waitForStream();` Endpoint 1 receives MAX_DATA and MAX_STREAM_DATA frames from Endpoint 2 despite Endpoint 2 not sending any DATA_BLOCKED and STREAM_DATA_BLOCKED frames:	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): PmVLJkx6MTl7013R5XdeeAs5WSM=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 10
Frames:
{

Frame 1
-------
Type: MAX_DATA
Maximum Data: 501


}
```

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): PmVLJkx6MTl7013R5XdeeAs5WSM=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 12
Frames:
{

Frame 1
-------
Type: MAX_STREAM_DATA
StreamID: 3
Maximum Stream Data: 501


}
```

Note that this also means that client has very little chance to go over client limits at this time so FLOW_CONTROL_ERROR will never be triggered, but in case that it could be following code blocks show that it will be triggered in that theoretical scenario:

```
if((connRecvAmnt.get() + ((StreamFrame) frame).getData().length) > connRecvLimit.get()) {
    close(TransportErrorCodes.FLOW_CONTROL_ERROR.getErrorNum(), reason: "Exceeded Total Connection Limit");
    break;
}
```

```
if((streamRecvAmnt.get() + ((StreamFrame) ffs).getData().length) > streamRecvLimit.get()) {
    close(TransportErrorCodes.FLOW_CONTROL_ERROR.getErrorNum(), reason: "Exceeded Total Stream Limit");
    break;
}
```

7	Code for Endpoint 1 that creates a stream with Endpoint 2 by sending some data: `/// Test 7` `System.out.println("Connection Send Limit(Before): " + qs.getConnSendLimit());` `QUICConnection.QUICStream quicStream = qs.createStream( bidirectional: false);` `System.out.println("Stream Send Limit(Before): " + quicStream.getStreamSendLimit());` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `qs.sendSinglePacket(sp);` `Thread.sleep( l: 2000);` `System.out.println("Connection Send Limit(After): " + qs.getConnSendLimit());` `System.out.println("Stream Send Limit(After): " + quicStream.getStreamSendLimit());` Code for Endpoint 2 to get stream from Endpoint 1 and send MAX_DATA and MAX_STREAM_DATA frames to Endpoint 1 lower than initial limits (see Test 3): `/// Test 7` `QUICConnection.QUICStream quicStream = client.waitForStream();` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new MaxDataFrame( maximumData: 1));` `sp.addFrame(new MaxStreamDataFrame(quicStream.getStreamID(), maximumStreamData: 1));` `client.sendSinglePacket(sp);` Limits on Endpoint 1 not changing despite receiving MAX_DATA and MAX_STREAM_DATA:	**PASS**

```
Connection Send Limit(Before): 100
Stream Send Limit(Before): 100

Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): vQeAXwb1iPQaZxOsYP7CKQHceog=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: ACK
Largest Acknowledged: 3
ACK Delay: 4
ACK Range Count: 0
First ACK Range: 0


}

Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): vQeAXwb1iPQaZxOsYP7CKQHceog=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: MAX_DATA
Maximum Data: 1

Frame 2
-------
Type: MAX_STREAM_DATA
StreamID: 2
Maximum Stream Data: 1


}

Connection Send Limit(After): 100
Stream Send Limit(After): 100
```

8	Code for Endpoint 1 creates a bi-directional stream with endpoint 2 by sending some data and then closes its sending portion of the stream: `/// Test 8` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: true);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` `Thread.sleep( l: 1000);` `quicStream.closeStream();` Code for Endpoint 2 waits for stream from Endpoint 1 and checks send and recv state before and after Endpoint 1 closes the send portion of the stream: `/// Test 8` `QUICConnection.QUICStream quicStream = qs.waitForStream();` `System.out.println("Send State before recv portion of stream is closed: " + quicStream.getSendState());` `System.out.println("Recv State before recv portion of stream is closed: " + quicStream.getRecvState());` `System.out.println();` `Thread.sleep( l: 2000);` `System.out.println("Send State after recv portion of stream is closed: " + quicStream.getSendState());` `System.out.println("Recv State before recv portion of stream is closed: " + quicStream.getRecvState());` Send state for Endpoint 2 still stays 0 (Ready-to-Send) while Recv state changed to 2 (Data Recv; the reason Recv state did not change to 3 is because the app has not read the data and thus the stream still has to wait for the data to be read from it):	**PASS**

```
Send State before recv portion of stream is closed: 0
Recv State before recv portion of stream is closed: 0

Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): M/Yy/5OTn0F9gQ6q3T0gguw/KyE=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: STREAM (FIN Bit Set)
StreamID: 1
Fin Bit: Set
Data: []


}

Send State after recv portion of stream is closed: 0
Recv State before recv portion of stream is closed: 2
```

9

Code for Endpoint 1 to keep creating without stopping and updating the flow control limits:

```
/// Test 9
while(true) {
    QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);
    ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);
    sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,
            fin: false, "hello".getBytes(StandardCharsets.UTF_8)));
    client.sendSinglePacket(sp);
    Thread.sleep( l: 10);
}
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will automatically start to manage new streams and the flow control limits.

Endpoint 1 receives MAX_STREAMS frames from Endpoint 2 despite Endpoint 1 not sending any STREAMS_BLOCKED frames:

PASS

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): Y8HFXOWAKZUP7Tk+CG389l2RVL8=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 36
Frames:
{

Frame 1
-------
Type: MAX_STREAMS (Unidirectional)
Maximum Streams: 148


}
```

10	Code for Endpoint 1 that just prints limits before and after getting the two MAX_STREAMS frames: `/// Test 10` `System.out.println("Connection Bi-Directional Streams Limit(Before): " + qs.getConnSendBidiStreamLimit());` `System.out.println("Connection Uni-Directional Streams Limit(Before): " + qs.getConnSendUniStreamLimit());` `System.out.println();` `Thread.sleep( b 2000);` `System.out.println("Connection Bi-Directional Streams Limit(After): " + qs.getConnSendBidiStreamLimit());` `System.out.println("Connection Uni-Directional Streams Limit(After): " + qs.getConnSendUniStreamLimit());` Code for Endpoint 2 that sends a 1-RTT packet to Endpoint 1 with stream amount limit updates for both Uni-Directional and Bi-Directional streams, but with lower value than initial limits (see test 3): `/// Test 10` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new MaxStreamsFrame( maximumStreams: 1, streamType: 0));` `sp.addFrame(new MaxStreamsFrame( maximumStreams: 1, streamType: 1));` `client.sendSinglePacket(sp);` Limits on Endpoint 1 not changing despite receiving MAX_STREAMS for both Uni-Directional and Bi-Directional streams:	**PASS**

```
Connection Bi-Directional Streams Limit(Before): 5
Connection Uni-Directional Streams Limit(Before): 5

Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): Fxx3cYK2YbNzfuIoAK6WvA97D/c=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: MAX_STREAMS (Bidirectional)
Maximum Streams: 1

Frame 2
-------
Type: MAX_STREAMS (Unidirectional)
Maximum Streams: 1


}

Connection Bi-Directional Streams Limit(After): 5
Connection Uni-Directional Streams Limit(After): 5
```

11	Endpoint 2 (Server) tries to send a packet different than the QUIC version that was agreed upon (Version 34): `/// Test 11` `HandshakePacket hp = new HandshakePacket(client.getDestConnectionID(), client.getSourceConnectionID(), packetNum: 0, frames: null);` `hp.setVersion(GlobalConstants.SUPPORTED_QUIC_VERSIONS[0] + 1);` `hp.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));` `client.sendSinglePacket(hp);` There is no unique testing code for Endpoint 1 (Client) as just having a connection with Endpoint 2 will make it automatically start to receive and process packets.	**PASS**

	Endpoint 1 (Client) discards the packet sent to it by Endpoint 2 (Server): CLIENT VERSION: 34 DISCARDED PACKET (due to different version): Header Form: Long Fixed Bit: 1 Packet Type: Handshake Version: 35 Destination Connection ID Length: 20 Destination Connection ID (Base64): uVKMe97BHc5ivx4TrXbBRlgJpiw= Source Connection ID Length: 20 Source Connection ID (Base64): lykBA+8Bsrn6ZLaqigG8o9Yzoa8= Packet Number Length (Bytes): 1 (encoded as 0) Length: 505 Packet Number: 3 Frames: { Frame 1 ------- Type: CRYPTO Offset: 0 Length: 500 Crypto Data (Base 64): [-87, 41, 26, 39, -120, -86, 20, -52, -77, -72, -20, -74, -59, -19, 56, -8, 13, 12, -109, 127, 50, -116, 81, 126, 35, -50, -97, 6, 117, -117, 106, 84, 111, -28, 22, -55, 36, -21, 73, 30, -43, -21, 44, 56, 50, -49, -53, 106, 90, 2, -24, 102, -108, 16, 31, 73, 79, 80, -53, -40, -75, 118, 6, -74, -52, 26, -70, -39, -50, -41, -62, 88, -111, 125, 16, 49, -93, -62, 7, 96, -103, 52, 30, -14, -110, -98, 77, 72, 38, -61, 97, 99, 32, 119, -23, 64, 55, 27, 61, 13, -8, -35, 78, -121, -82, -31, 76, 20, -116, 87, 111, -32, -3, 60, -90, -112, 85, 123, -7, 64, 89, 10, -120, -64, -90, 117, -114, 84, -94, -59, 50, 52, 91, -121, -94, -115, -111, -91, 68, 0, -120, 4, -67, 15, -96, 103, -10, -37, -8, -120, 83, -44, -49, 15, 74, 64, -15, 34, 48, 1, -12, -29, -125, 105, 27, 107, -99, 69, -8, 64, -77, 28, -80, -98, 44, 107, 102, -104, 8, 24, -34, -70, -22, 83, 0, -3, -102, 3, -42, 89, 5, 113, 62, -30, -111, 14, -72, 69, -34, -81, 25, -118, 95, -117, 30, 83, 120, -16, -4, -17, -64, -128, 118, -43, 120, 24, 67, -100, -7, 98, 121, -76, -85, -12, 13, -6, -61, -97, 90, 67, -72, 116, 127, 39, -9, 105, -25, -81, -57, 52, 75, -70, -13, -82, -4, -43, -10, 80, 62, 52, 104, 103, 52, -64, -49, -91, -17, 83, 40, 118, 33, 83, -1, 73, 116, -97, -63, 62, -125, 26, 105, 52, -9, 86, 2, 39, 46, 30, 11, -61, -103, 76, -57, 32, 69, -41, 109, 89, -120, 103, -18, 109, -63, 58, 93, 51, 64, 64, 100, -64, 37, 34, 121, 115, 24, -76, -110, -21, 33, 21, -104, -14, -59, -8, -42, -6, 91, -40, -2, -32, 42, 87, -11, -95, -33, 7, 109, 55, -25, 106, -57, -106, -28, 32, -85, 120, -35, -68, -124, 34, 46, -19, 39, -39, -47, 39, -11, 6, 25, -18, 55, -60, 61, 91, -126, 53, 20, 103, 26, 63, -48, 101, 127, -55, 87, 48, -98, 89, 75, -82, 28, -24, 106, 55, 54, 27, -19, -29, 37, 119, -85, -121, 121, -50, 21, -22, 127, -14, -42, 24, -30, 50, 91, -1, -15, -28, 20, -6, 118, 51, -47, 34, 67, -54, -108, -45, 120, -115, 23, -78, 20, -121, 86, 91, 89, 72, 76, -55, 28, 36, -112, -19, -20, -61, 61, -70, 102, -60, -23, -30, -116, 99, 100, -22, -104, -49, 2, -113, -39, 44, -51, 1, 103, -42, -52, 108, -5, 60, -75, -42, -25, 63, -110, 121, -98, -99, -119, -74, 62, 80, 122, -94, -82, -64, -30, 79, 103, -101, -90, -38, 48, 95, -20, -83, 35, -25, 99, 116, 20, 76, 108, 127, 20, 112, -35, -124, 120, -8, 0, 112, -64, -83, 32, 37, -6, -83, -40, 94, 89, -82] }	
12	Endpoint 2 (Server) tries to send a Version Negotiation packet after connection has been already been established (meaning that other packets have already been sent): `/// Test 12` `long[] supported_versions = {35, 40, 33};` `VersionNegotiationPacket vnp = new VersionNegotiationPacket(client.getDestConnectionID(),` `client.getSourceConnectionID(), supported_versions);` `client.sendSinglePacket(vnp);` There is no unique testing code for Endpoint 1 as just having a connection with Endpoint 2 will make it automatically start to receive and process packets. Endpoint 1 (Client) discards the packet sent to it by Endpoint 2 (Server):	**PASS**

	`` DISCARDED PACKET (due to other packets already being received): Header Form: Long Fixed Bit: 1 Packet Type: Version Negotiation Destination Connection ID Length: 20 Destination Connection ID (Base64): o3/M+iEbUufbCARtBMT8xXIs1uM= Source Connection ID Length: 20 Source Connection ID (Base64): 88DQFZ5Gg5FF4aYUnn/nY+bhK3E= Supported Versions: [35 40 33] ```	
13	Endpoint 2 (Server) tries to send a Version Negotiation packet with same version as client already has set: ``` /// Test 13 long[] supported_versions = {34}; VersionNegotiationPacket vnp = new VersionNegotiationPacket(client.getDestConnectionID(), client.getSourceConnectionID(), supported_versions); client.sendSinglePacket(vnp); ``` There is no unique testing code for Endpoint 1 as just having a connection with Endpoint 2 will make it automatically start to receive and process packets. Endpoint 1 (Client) discards the packet sent to it by Endpoint 2 (Server):	**PASS**

```
CLIENT VERSION: 34
DISCARDED PACKET (due to supported versions containing a version selected by the client):

Header Form: Long
Fixed Bit: 1
Packet Type: Version Negotiation
Destination Connection ID Length: 20
Destination Connection ID (Base64): eLOGaPO2SRMCQgDo6EhDPFInA1k=
Source Connection ID Length: 20
Source Connection ID (Base64): asVrt3QpjJ2S4kex5gwahzKSTTM=
Supported Versions:
[
34
]
```

Note that this test would have failed similar to Test 12 if the condition for test 13 did not exist as there were already other packets received, but even if there weren't, test 13 would have still failed so this test is still valid.

14	Endpoint 2 (Server) tries to send a Version Negotiation packet with different Destination and Source Connection IDs than client has set:	**PASS**

```java
/// Test 14
long[] supported_versions = {35, 40, 33};
byte[] destConnectionID = new byte[20];
try {
    SecureRandom.getInstanceStrong().nextBytes(destConnectionID);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
byte[] srcConnectionID = new byte[20];
try {
    SecureRandom.getInstanceStrong().nextBytes(srcConnectionID);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
VersionNegotiationPacket vnp = new VersionNegotiationPacket(destConnectionID,
        client.getSourceConnectionID(), supported_versions);
client.sendSinglePacket(vnp);
VersionNegotiationPacket vnp2 = new VersionNegotiationPacket(client.getDestConnectionID(),
        srcConnectionID, supported_versions);
client.sendSinglePacket(vnp2);
```

There is no unique testing code for Endpoint 1 as just having a connection with Endpoint 2 will make it automatically start to receive and process packets.

Endpoint 1 (Client) discards the two Version Negotiation packets sent to it by Endpoint 2 (Server):

```
CLIENT Source Connection ID: UWmlxhAJKEPMy+kA9vFwkp6CX5A=
CLIENT Destination Connection ID: JTX+DFzEBo6IXrVYmgkjmxUpkQM=
DISCARDED PACKET (due to Source Connection ID not matching Destination Connection ID of the client or due to Destination Connection ID not
  matching Source Connection ID of the client):

Header Form: Long
Fixed Bit: 1
Packet Type: Version Negotiation
Destination Connection ID Length: 20
Destination Connection ID (Base64): ht7CJe1DzTehwbsdO8dGIy5nGS0=
Source Connection ID Length: 20
Source Connection ID (Base64): JTX+DFzEBo6IXrVYmgkjmxUpkQM=
Supported Versions:
[
35
40
33
]


CLIENT Source Connection ID: UWmlxhAJKEPMy+kA9vFwkp6CX5A=
CLIENT Destination Connection ID: JTX+DFzEBo6IXrVYmgkjmxUpkQM=
DISCARDED PACKET (due to Source Connection ID not matching Destination Connection ID of the client or due to Destination Connection ID not
  matching Source Connection ID of the client):

Header Form: Long
Fixed Bit: 1
Packet Type: Version Negotiation
Destination Connection ID Length: 20
Destination Connection ID (Base64): UWmlxhAJKEPMy+kA9vFwkp6CX5A=
Source Connection ID Length: 20
Source Connection ID (Base64): AlHWcdpKr1zWp8jrRiex9X/qql0=
Supported Versions:
[
35
40
33
]
```

Note that this test would have failed similar to Test 12 if the condition for test 14 did not exist as there were already other packets received, but even if there weren't, test 14 would have still failed so this test is still valid.

15	Endpoint 1 tries to send an Initial packet with different Source Connection ID than Endpoint 2 has set as Destination Connection ID: `/// Test 15` `byte[] srcConnectionID = new byte[20];` `try {` `  SecureRandom.getInstanceStrong().nextBytes(srcConnectionID);` `} catch (NoSuchAlgorithmException e) {` `  e.printStackTrace();` `}` `InitialPacket ip = new InitialPacket(client.getDestConnectionID(), srcConnectionID, new byte[]{}, packetNum: 0, frames: null);` `ip.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));` `client.sendSinglePacket(ip);` Endpoint 2 discards the packet sent by Endpoint 1:	**PASS**

```
DESTINATION CONNECTION ID: TwH0bE8A09dSro0w0JUZbQKzMo4=
DISCARDED PACKET (due to Source Connection ID of the packet not matching the Destination Connection ID of the endpoint):

Header Form: Long
Fixed Bit: 1
Packet Type: Initial
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): EMNBXCgSBC64l30mWCKoZT7Mn5c=
Source Connection ID Length: 20
Source Connection ID (Base64): 6eqsQuLuyBmu5FtAMhwvtrltd3k=
Packet Number Length (Bytes): 1 (encoded as 0)
Token Length: 0
Length: 1096
Packet Number: 1
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [111, -109, 50, 49, -20, 61, 82, 125, -89, -60, 76, 98, 27, 58, 35, 30, 28, 46, -8, 15, 111, -25, 106, 51, -128,
 -34, 56, 53, -4, -106, 36, 26, 60, 91, -24, 112, -90, 6, 70, 26, 100, -54, -102, -99, 90, 112, 69, 105, 57, 13, -85, 6, 79, -100, 68,
 -17, -113, -62, 47, 14, 81, 84, -55, 53, 124, -45, -112, -80, -86, -8, 0, 77, 55, -125, -71, -97, 104, -76, -127, -2, -75, -25, -13, -31,
  -78, -48, 66, 61, 99, 20, 31, -89, -120, -41, 109, -39, -108, -50, 87, -118, -36, -28, -18, -32, -47, -101, 39, 110, -34, -11, -7, -123,
  5, 114, -79, -78, -81, -99, 108, 124, -65, 28, -54, -8, 69, -116, -85, 109, 74, -13, -37, 110, -93, -64, -36, -53, -127, -123, 1, 113,
 -20, 34, -95, -37, -91, -82, 18, -102, -124, 49, 61, -28, -118, 108, -127, -56, 50, 16, 91, 61, 124, 49, 65, 47, -97, 127, -120, 124, -8,
  61, 56, 110, -79, -89, -18, 37, -53, 9, -10, -34, -99, 89, 14, 60, 101, 100, -20, -67, -74, 71, 61, -31, 48, 14, 53, 80, -64, -70, 54,
 -87, -122, 67, 41, 117, 47, -7, 96, 38, -34, 61, -97, 55, -55, -65, -26, 108, 52, -38, -123, 76, 58, -109, -118, 11, 82, 91, -128, -74,
 -36, -127, 100, 69, 91, -42, 22, 89, -20, 59, 2, -46, -108, -58, 11, 91, -126, 123, -71, 85, -80, -40, 75, -98, -8, 115, 29, -126, 66,
 -82, 7, 52, 95, 81, 12, -2, -102, -116, 12, -51, 91, -31, -121, 55, 13, -68, -114, 88, -125, 1, -126, 72, 58, -62, -116, 1, 42, 17, -31,
 48, -108, -40, 108, -110, -114, 77, -22, -18, -128, 122, -63, -24, 25, -73, 93, 52, 44, 20, 55, -45, 73, 74, -20, 100, -68, 94, 39, -114,
  -72, -45, 29, -14, 79, 118, 39, 105, 0, 76, -90, 113, 27, 83, -128, 35, 72, 61, 47, 23, 77, 113, 3, 118, 116, -1, -82, 69, -113, -66,
 -123, 116, -69, 41, -41, 105, 52, -113, -69, -39, -25, -115, 49, 78, 119, 71, -53, 33, 123, 40, -9, -14, 41, 17, 21, -95, -68, 17, -106,
 -111, 19, 60, 71, -108, 25, 54, -16, 76, 102, 96, 36, 5, 10, 114, 81, -96, -90, 85, 56, -77, -125, 103, -96, 17, 44, -26, 2, 67, -14, 92,
  50, 45, 29, 19, -3, -108, 99, 7, 75, -86, -98, 113, -109, -38, -41, 80, 59, -95, 53, -33, 10, 27, -7, -58, 99, -45, -111, -16, 58, -92,
 3, -20, 63, -41, -100, 12, 53, -63, 54, -63, 114, 11, 34, 124, 70, 85, 65, 16, 118, -50, -13, 47, -93, -5, -79, 103, -110, -61, 49, 88,
 -110, -107, -18, -81, -69, -22, 42, 79, -122, -97, 63, 54, -108, 56, -75, -95, 93, -29, -3, 50, 45, 17, -87, -128, -38, -65, 40, -19, 69,
  127, 0, -1, -40, -120]

PADDING Length: 591

}
```


16	Endpoint 1 closes connection with endpoint 2 after it has been established: `/// Test 16` `client.close();` EndPoint 2 receives CONNECTION_CLOSE frame in Short (1-RTT) packet:	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): um4BKcjSjmrFphjcRucPyv4kx3E=
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 0
Reason Phrase Length: 0
Reason Phrase:


}
```

17	Code for endpoint 2 (Client) that triggers a Stateless reset by changing the Destination Connection ID for an initial packet that has a packet number other than 0 so that Endpoint 1 (Server) cannot identify the connection and must send a stateless reset as it seems like it has no state with that client: There is no unique testing code for Endpoint 1 as just having a connection with Endpoint 2 will make it automatically start to receive and process packets. Endpoint 2 receives an stateless reset packet that it identifies by the stateless reset token:	**PASS**

```java
/// Test 17
byte[] dstConnectionID = new byte[20];
try {
    SecureRandom.getInstanceStrong().nextBytes(dstConnectionID);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
InitialPacket ip = new InitialPacket(dstConnectionID, qs.getSourceConnectionID(), new byte[]{}, packetNum: 0, frames: null);
ip.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));
qs.sendSinglePacket(ip);
```

	`` Header Form: Short Fixed Bit: 1 Stateless Reset Token: G9dA4PGd/Sw+gyD67WnQ2g== Closed Connection to Server GI8l5wIZZIQLx0E4ZPIdDcgOa2k= with Error Code 0x0A (PROTOCOL_VIOLATION): Stateless Reset Token Received. ``` Here is the code that shows that stateless reset packet is identify by the token rather than anything else so the header form would not matter: ``` if(packetBytes.length > 16) { byte[] possibleStateResetToken = new byte[16]; System.arraycopy(packetBytes, i: packetBytes.length - 16, possibleStateResetToken, i1: 0, i2: 16); if(Arrays.equals(possibleStateResetToken, GlobalConstants.STATELESS_RESET_TOKEN)) { packets.add(new StatelessResetPacket(packetBytes, packet_offset: 0)); statelessResetPacket = true; } } ```	
18	Neither Endpoint 1 (Client) nor Endpoint 2 (Server) have any special testing code for this one except the print statements because when the connection is getting created, the first two packets Endpoint 1 (server) sends to Endpoint 2 (client) get coalesced every time so I just print out these coalesced packets:	**PASS**

```
2 coalesced packets in one datagram:
[
PACKET 0
Header Form: Long
Fixed Bit: 1
Packet Type: Initial
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): E0eAinCksc3XSfgj6uyqKMq+1XM=
Source Connection ID Length: 20
Source Connection ID (Base64): /4HYovzww8kdu2QEL8+TwTF5nFo=
Packet Number Length (Bytes): 1 (encoded as 0)
Token Length: 0
Length: 596
Packet Number: 0
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 96
Crypto Data (Base 64): [-21, 52, -103, -63, 81, -123, -7, 104, -108, 125, -75, 15, 75, 67, -127, -102, 104, 31, 17, -123, 87, 86, 7,
 -128, 111, -74, 45, 90, -78, 49, -71, -50, 63, 19, -25, -119, -61, 127, -21, -52, -43, 50, -22, 96, -36, -111, -107, -42, 116, 100,
 -108, -127, 34, -124, -88, 117, 111, 32, -47, -110, -89, -53, 95, 59, -46, 0, 49, -60, -86, 16, 1, -77, 33, -12, -97, 59, -75, 8,
 -94, 63, 21, -39, -3, 99, 86, -29, 51, 105, 1, 35, -14, 13, -35, -67, 41, -44]

PADDING Length: 495

}

PACKET 1
Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): E0eAinCksc3XSfgj6uyqKMq+1XM=
Source Connection ID Length: 20
Source Connection ID (Base64): /4HYovzww8kdu2QEL8+TwTF5nFo=
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 505
Packet Number: 0
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [-116, -104, 92, 24, 54, -90, -10, -35, 30, 62, 23, 37, -123, -33, 11, 94, 21, 120, 26, 64, -77, 1, -29, -70,
 -21, 98, 28, -90, 84, -74, 96, -82, -99, 55, 17, -75, 126, -85, 23, -41, -103, -8, 97, 107, -109, -61, 15, -107, -88, 29, -122, -4,
 32, -27, 25, -73, -124, 84, -44, 15, -22, 24, -58, 119, -28, -111, -19, -41, -25, -77, -1, 52, 43, -75, 69, 42, 86, 69, 28, -63, -25,
  26, -70, 119, 61, -80, 88, -90, -86, -92, -58, 66, -70, -21, 70, -66, 113, -79, -16, -74, -113, 26, 19, 48, -113, -69, 30, -57, -49,
  -28, -11, 49, -29, 19, 101, -7, -96, 66, -109, 16, 17, -111, 45, -24, 8, -41, -108, 46, -88, 32, 79, -6, 101, -89, -77, -57, -69,
 19, 103, 16, 93, 20, 119, 23, 84, 47, 51, -76, 29, 62, -121, -43, -30, 25, 111, 106, -54, -41, -91, -89, -27, 26, -80, 116, 1, -81,
```

Here are the ACK's for these two packets (sent from Endpoint 2 to Endpoint 1) which shows that they have been received and processed correctly:

```
Header Form: Long
Fixed Bit: 1
Packet Type: Initial
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): /4HYovzww8kdu2QEL8+TwTF5nFo=
Source Connection ID Length: 20
Source Connection ID (Base64): E0eAinCksc3XSfgj6uyqKMq+1XM=
Packet Number Length (Bytes): 1 (encoded as 0)
Token Length: 0
Length: 942
Packet Number: 1
Frames:
{

Frame 1
-------
Type: ACK
Largest Acknowledged: 0
ACK Delay: 4
ACK Range Count: 0
First ACK Range: 0

PADDING Length: 936

}

Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): /4HYovzww8kdu2QEL8+TwTF5nFo=
Source Connection ID Length: 20
Source Connection ID (Base64): E0eAinCksc3XSfgj6uyqKMq+1XM=
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 6
Packet Number: 0
Frames:
{

Frame 1
-------
Type: ACK
Largest Acknowledged: 0
ACK Delay: 4
ACK Range Count: 0
First ACK Range: 0


}
```

19	Code for Endpoint 1 sends a packet to Endpoint 2 with no frames: `/// Test 19` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `qs.sendSinglePacket(sp);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1will make it automatically start to receive and process packets. Endpoint 2 receives packet from Endpoint 1 with no frames and closes connection due to error: `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): 8G/r3DA0ol/LijwQAt1qJHs+FWs=` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 1` `Frames: (None)` `New Client Connected: ID 8G/r3DA0ol/LijwQAt1qJHs+FWs=` `Client ID 8G/r3DA0ol/LijwQAt1qJHs+FWs= Disconnected with Error Code 0x0A (PROTOCOL_VIOLATION): No frames in packet.` Endpoint 1 receives connection close from Endpoint 2 with PROTOCOL_VIOLATION error code (0x0A): `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): 1XcJsgtODaRl7JNvAYJe75bM5r0=` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 1` `Frames:` `{` `Frame 1` `-------` `Type: CONNECTION_CLOSE (Application Specific)` `Error Code: 10` `Reason Phrase Length: 19` `Reason Phrase: No frames in packet` `}` `Closed Connection to Server 8G/r3DA0ol/LijwQAt1qJHs+FWs= with Error Code 0x0A (PROTOCOL_VIOLATION): No frames in packet.`	**PASS**
20	Code for Endpoint 1 to send a Handshake packet to Endpoint 2 with STREAM frame:	**PASS**

```
/// Test 20
HandshakePacket ip = new HandshakePacket(qs.getDestConnectionID(), qs.getSourceConnectionID(), packetNum: 0, frames: null);
ip.addFrame(new StreamFrame( streamID: 0, offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,
        fin: false, "hello".getBytes(StandardCharsets.UTF_8)));
qs.sendSinglePacket(ip);
```

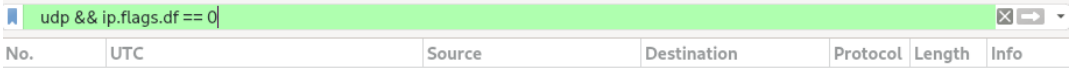
There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1will make it automatically start to receive and process packets.

Endpoint 2 receives Handshake packet from Endpoint 1 with STREAM frame and closes connection due to error:

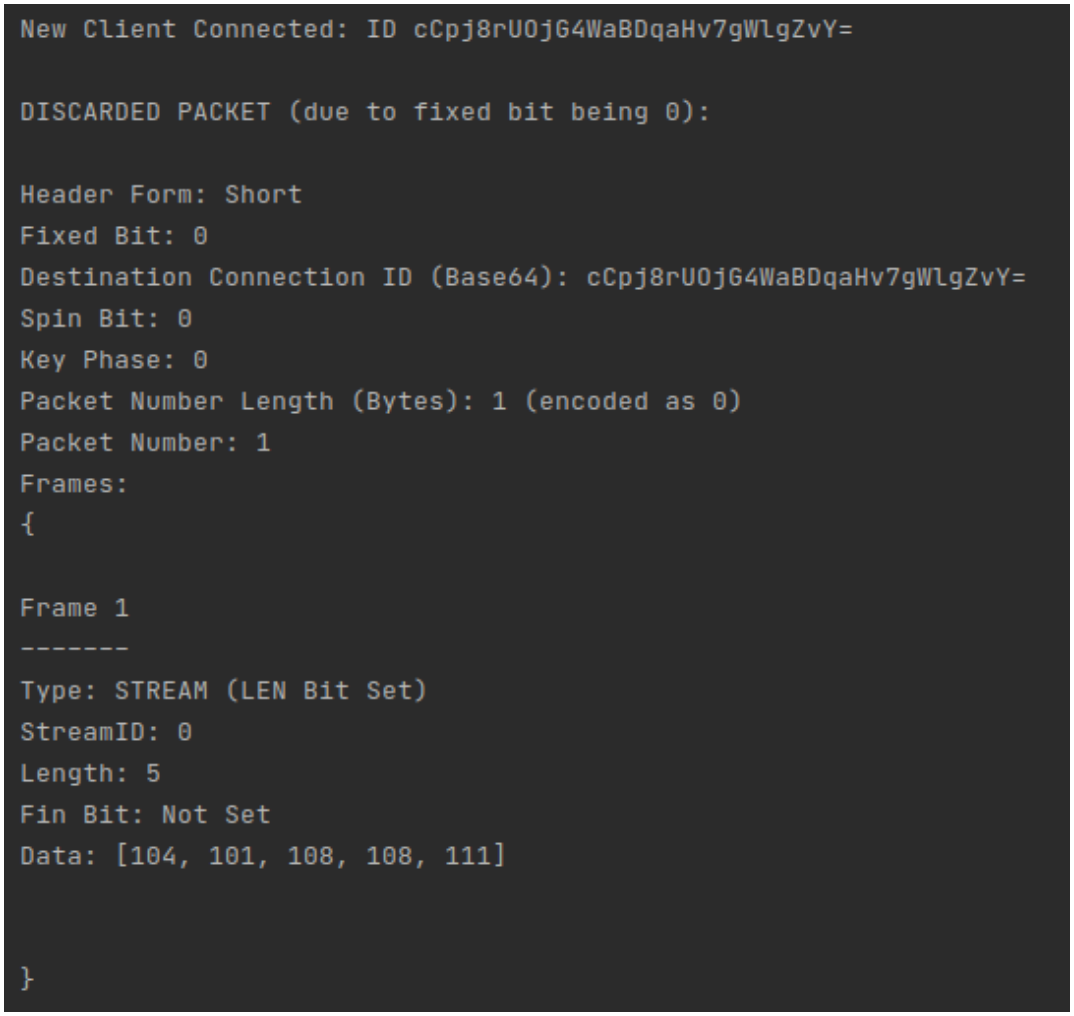
```
Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): bBLoe1a50em84aJKgLFmmc2zxhg=
Source Connection ID Length: 20
Source Connection ID (Base64): 8v3BrTNIu+dRr0Jp/qhrxVCDpSM=
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 9
Packet Number: 3
Frames:
{

Frame 1
-------
Type: STREAM (LEN Bit Set)
StreamID: 0
Length: 5
Fin Bit: Not Set
Data: [104, 101, 108, 108, 111]


}

Client ID bBLoe1a50em84aJKgLFmmc2zxhg= Disconnected with Error Code 0x0A (PROTOCOL_VIOLATION): Wrong frames in packet.
```

Endpoint 1 receives connection close from Endpoint 2 with PROTOCOL_VIOLATION error code (0x0A):

	<pre>Header Form: Short Fixed Bit: 1 Destination Connection ID (Base64): 8v3BrTNIu+dRr0Jp/qhrxVCDpSM= Spin Bit: 0 Key Phase: 0 Packet Number Length (Bytes): 1 (encoded as 0) Packet Number: 1 Frames: { Frame 1 ------- Type: CONNECTION_CLOSE (Application Specific) Error Code: 10 Reason Phrase Length: 22 Reason Phrase: Wrong frames in packet }</pre>	
21	This is achieved by having a fairly small Maximum Datagram Size (1200 bytes) which is smaller than most MTU sizes meaning that there is not reason for the packet to ever be fragmented: `public static int MAX_UDP_PAYLOAD_SIZE = 1200;` This is confirmed by sending a 1MB file using the QUIC Distributed File Server application and not seeing any IPv4 Packets that don't have Don't Fragment bit set in Wireshark for the transmission:	**PASS**

		
22	There is no unique testing code for this test for neither Endpoint 1 or 2. Endpoint 1 makes a connection to Endpoint 2 and both are recording whether Datagram packet contains Initial Packet and if it does, print out the size of that datagram. Here are the results: Endpoint 1: `DATAGRAM SIZE (with packet containing InitialPacket): 1200` `Connected to Server CzzCKQovog57iWm5v1RnDm4Zb1M=.` Endpoint 2: `DATAGRAM SIZE (with packet containing InitialPacket): 1200` `DATAGRAM SIZE (with packet containing InitialPacket): 1200` `New Client Connected: ID CzzCKQovog57iWm5v1RnDm4Zb1M=` As you can see, the Datagram size when packet contains InitialPacket is always 1200 bytes which is the minimum allowed maximum UDP payload size for this implementation: `public static int MAX_UDP_PAYLOAD_SIZE = 1200;`	**PASS**
23	Endpoint 1 (Client) tries to make a connection with Endpoint 2 (Server), but it only pads the size of Datagram containing Initial packets to 1199 bytes instead of 1200: `/// Test 23` `GlobalConstants.MAX_UDP_PAYLOAD_SIZE = 1199;` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1will make it automatically start to receive and process packets. Endpoint 2 discards the	**PASS**

	Initial packet sent to it by the client due to the UDP datagram that contains it being too small: `DATAGRAM SIZE (with packet containing InitialPacket): 1199` `DISCARDED PACKET (due to small size of datagram):` `Header Form: Long` `Fixed Bit: 1` `Packet Type: Initial` `Version: 34` `Destination Connection ID Length: 20` `Destination Connection ID (Base64): 9yQkRo/doyiJoPpoKkukmVTtQDI=` `Source Connection ID Length: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Token Length: 0` `Length: 1169` `Packet Number: 0` `Frames:` `{` `Frame 1` `-------` `Type: CRYPTO` `Offset: 0` `Length: 307` `Crypto Data (Base 64): [17, 99, 120, 39, -22, -19, -92, 113, -79, -105, -11, 102, 105, -37, 89, 105, 91, -3, -95, -65, 110, -11, -104, 49, -81, -106, 84, 86, -128, -10, 36, 108, 49, -61, -42, 45, 35, -55, 113, -43, -81, -95, 103, -48, -121, -78, 45, -35, -103, 74, -74, 14, 126, -67, 101, 124, -96, -95, 69, 66, -10, -113, 55, -45, -78, -50, 66, -7, -1, 91, 93, -67, -68, -50, -49, 7, 45, 109, 100, 13, 123, 25, -49, 48, 58, -87, 102, -93, -56, -81, 44, -100, -16, -5, 55, -10, 82, 43, 2, -1, -32, -106, -16, 65, 42, -14, -75, 121, -49, 68, -31, 55, -85, 75, 90, 117, -58, 17, 34, -1, 66, 42, -90, 11, -105, 48, -128, -126, 74, 28, -1, 74, 73, 103, -6, 48, -41, -49, 16, 114, -127, 93, -24, 103, -27, -117, 73, 80, 7, -50, -118, 81, 0, 6, 28, 17, -46, 55, 84, 80, -20, 74, 81, -86, -20, -97, 127, 32, 20, -33, 87, -96, -6, 52, -127, -6, -71, -12, -7, 59, 80, 87, -88, 55, 86, 77, 1, 85, 122, 106, 76, -62, 116, -37, -56, -52, -95, 95, -74, 73, 78, 107, -111, 112, -18, -68, -87, 101, -11, 70, 102, -101, -116, -28, 92, -33, 2, 52, -76, 9, 110, -41, -42, -108, -41, -72, -71, -62, -61, -16, -124, -96, 36, 23, 0, -88, 110, -78, -109, -85, 61, 46, 10, 27, -1, 51, -23, 68, 93, 2, -78, -70, 48, -23, 84, -64, 54, 119, 122, 42, 39, -87, 29, -22, 13, -87, -93, -62, 40, -6, 13, -6, 79, -99, -47, 92, 21, -49, -117, 114, 13, -109, -71, -87, -128, -43, -98, -105, -108, 15, 28, -33, 10, 33, 86, 37, -96, -32, -48, 0, -99, 4, 102, 67, 63, 12, -27]`  `PADDING Length: 857`  `}`	
24	Code for Endpoint 1 to send a Short (1-RTT) Packet to Endpoint 2 with fixed bit set to 0: `/// Test 24` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame( streamID: 0, offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `sp.setFixedBit(0);` `qs.sendSinglePacket(sp);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1will make it automatically start to receive and process packets. Endpoint 2 discards the Short Packet received from Endpoint 1:	**PASS**

		
25	Code for Endpoint 1 to send a Handshake Packet to Endpoint 2 with Destination Connection ID being more than 21 bytes:	**PASS**

```
/// Test 25
byte[] destConnectionID = new byte[21];
try {
    SecureRandom.getInstanceStrong().nextBytes(destConnectionID);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
HandshakePacket hp = new HandshakePacket(destConnectionID, qs.getSourceConnectionID(), packetNum: 0, frames: null);
hp.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));
qs.sendSinglePacket(hp);
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 discards the Handshake Packet received from Endpoint 1:

```
New Client Connected: ID LlVG9rwMfnaW4QLKPeGLz57tWxE=

DISCARDED PACKET (due to Destination Connection ID being more than 20):

Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 21
Destination Connection ID (Base64): UsgJdo7GIrp7FPhCQH2gnYXdK0vm
Source Connection ID Length: 20
Source Connection ID (Base64): zPGLAABhHR6uef3JhnJ+o9kdoDE=
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 505
Packet Number: 3
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [-40, 85, 4, -126, 67, 68, 101, -123, 95, 127, 118, -127, -19, 75, -38, -63, 97, -58, -112, 123, 81, 117, 105,
 -123, -46, 71, 21, -18, 115, -115, -90, -45, 42, 71, 48, 94, 67, -46, 70, 3, 62, 4, 113, 112, 22, -13, -100, -98, -125, 112, 40, 21,
 -13, -118, 112, -74, -92, -127, -21, 44, -38, 83, 21, -97, -47, -103, 57, 5, -17, 97, 55, -53, 52, 120, 98, 19, -73, -114, -117, 15,
 -37, -80, 105, -78, 34, -34, 60, -118, 94, 57, 4, -9, -88, 70, -66, 14, -89, 76, 85, -99, -92, -116, -42, -16, 24, -98, -82, -121,
 -53, -90, 20, 95, -50, -31, 123, -40, 13, -95, 22, -59, 21, -59, -93, -74, 95, -116, 14, -49, 78, 64, -61, 85, 51, 48, 15, 21, 80,
 -113, 24, -101, -82, 61, -67, -116, 64, -89, -72, -101, -44, 99, -13, 69, 97, -63, 82, 5, -35, 26, 56, 95, -100, -126, 40, -123, -59,
  -96, -109, -99, -16, -75, 75, -69, 32, -46, 16, -7, -108, 32, -109, -108, -116, 57, 27, -14, 110, -8, 89, -11, 6, 50, -46, 91, 67,
 -38, 118, 39, -46, 90, 41, -106, 28, 113, -110, 3, 49, 20, -45, 73, -77, -94, 16, 57, -62, 93, -2, -108, -1, -102, 72, 24, -71, 71,
 65, 24, 90, 23, 111, 21, 18, -128, -75, -111, 100, 103, 112, -115, 80, 31, 94, 113, 8, 125, -10, -126, 52, 8, 19, 100, -49, 9, -83,
 -68, -82, 6, 29, 69, -124, -102, -98, 74, -124, -24, 120, -64, -47, 36, -96, -95, -41, 77, 70, 55, 17, -76, 124, 22, -100, 39, 1,
 -24, -29, -123, 32, 53, -19, 22, 25, 17, 53, 47, -51, -43, 37, -100, -31, -110, 26, 101, -72, -103, 90, -27, -26, 114, -23, -118, 63,
  -77, 101, -81, 55, -31, -50, -79, 101, -11, -94, -57, -93, -55, 12, -4, -71, 82, -41, -47, -77, 43, 5, 122, -48, 22, 9, -32, 122,
 -34, 97, 79, -93, -73, -86, -122, -49, -125, -31, 49, 60, -36, 86, 80, 81, 68, 66, -39, 72, 48, -41, -56, 110, -118, 87, 69, 101, -4,
  117, 30, 33, 80, -119, 101, 46, -52, -7, -104, -37, 69, -49, 81, 35, 3, 66, -14, -20, 12, 33, 20, -63, 59, -72, -99, 48, -18, -68,
 98, -52, 54, -43, -56, -109, -95, -51, 37, 5, 50, -3, 98, -54, -124, 6, -108, 96, 101, 64, -35, 44, 114, 48, 83, 109, -75, 63, -14,
 65, 6, 75, -66, -68, 25, -91, -118, 60, 55, 82, 41, 12, 109, -51, 24, 75, 122, 97, 0, -107, 21, -52, 36, 80, 98, 29, -4, 41, -103,
 -96, 20, 12, 48, -37, -96, 100, 65, -105, -51, 119, 119, 76, -22, 77, -125, -43, -107, -58, -16, -120, -68, 75, -20, -30, 19, -54,
 -108, -44, 95, 67, -35, 53, -41, 30, 114, 1, -16, -49, 107, 109, 60, -89, -13, -15, -46, 99, -9]


}
```

26	Code for Endpoint 1 to send a Handshake Packet to Endpoint 2 with Source Connection ID being more than 21 bytes:	**PASS**

```
/// Test 26
byte[] srcConnectionID = new byte[21];
try {
    SecureRandom.getInstanceStrong().nextBytes(srcConnectionID);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
HandshakePacket hp = new HandshakePacket(qs.getDestConnectionID(), srcConnectionID, packetNum: 0, frames: null);
hp.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));
qs.sendSinglePacket(hp);
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1will make it automatically start to receive and process packets. Endpoint 2 discards the Handshake Packet received from Endpoint 1:

```
New Client Connected: ID l5HFSXbRAttllUeulXps9zEE6Wo=

DISCARDED PACKET (due to Source Connection ID being more than 20 bytes):

Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): l5HFSXbRAttllUeulXps9zEE6Wo=
Source Connection ID Length: 21
Source Connection ID (Base64): pphkyUop1uqgCclD3CckcOfMuuF6
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 505
Packet Number: 3
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [46, 35, 38, 48, 17, 23, -104, 96, 89, 124, -43, 51, -33, -29, -117, -11, -115, -38, 52, -88, -61, -108, 80,
 -44, -103, 124, -4, 36, -119, -15, 18, 69, -36, 33, -95, -110, 72, 29, 0, 14, 31, -85, 109, -17, -2, -109, -17, 3, -58, 48, -62, 81,
 -79, 77, 91, -8, 16, -96, 105, 40, -76, -33, -115, 22, -35, -34, 66, -6, -35, -4, -78, -121, 65, -79, -40, 12, -102, 54, -18, -32,
 -72, -31, -99, -120, 48, -111, 78, -104, -12, -71, -84, 7, -59, -120, -3, -113, -7, 48, 32, 124, 54, 4, -32, 123, 111, -121, -6, -99,
  29, 63, 113, 25, 1, -10, -63, 109, -128, 118, 55, -38, 113, -58, 110, 31, 78, -72, -44, 112, 82, 71, 36, -50, 27, -79, -15, -12, 41,
  -116, -26, 54, -87, 1, 71, 56, -112, -122, -20, -31, -94, 96, -62, -56, -72, 99, 79, 82, 52, 116, 105, -80, 54, -123, 99, -57, -79,
 -121, -83, -15, 114, 35, 53, -15, 107, 90, -51, -49, -62, 75, -14, -50, 105, -62, -110, 68, -66, -77, 118, -42, 115, 106, 4, -122,
 41, 122, -75, 89, 48, -31, -50, 38, -28, 103, -32, -77, -124, -98, -20, -109, -75, 67, -36, -54, 24, 95, 84, -91, -75, 126, -60, -78,
  -59, -118, -63, 126, -42, 67, -67, -125, -43, -31, -95, -2, -24, -45, -99, 22, 67, 16, -84, -110, -101, -40, -103, 8, 80, 41, -109,
 -71, 113, 31, -1, -79, -112, 103, -65, -118, 9, 28, -1, -54, 125, 30, -94, 75, 104, -124, -74, -1, -1, 114, -10, -22, -103, 89, 61,
 -94, 35, -70, -41, 39, 0, 46, 85, -106, -38, 44, -118, -117, 113, -28, 21, 96, -63, 48, 81, -23, -71, -53, -125, 37, 50, -75, -44,
 75, 112, -83, -121, -101, -120, 107, -67, -96, 7, -85, 25, 96, -49, -84, -71, 107, 27, -116, -9, -127, -106, -68, -22, 54, 17, 21,
 -51, 60, 52, 29, 94, 124, -25, -82, -30, 61, 112, -27, -27, -49, 69, -42, -66, -30, -115, -21, 43, -51, 39, 11, -41, -72, 87, -72,
 14, -6, -120, -80, -115, -121, 93, 122, -58, 49, 96, -90, -97, 75, -6, 114, -24, 12, -93, -116, 52, 87, -75, -39, 82, -36, -25, -67,
 -71, -61, 73, -51, -23, -78, -23, -49, 46, 16, 26, 73, -13, -41, -27, 59, 64, -33, 8, -98, -111, -107, -22, 114, -20, 112, -44, -1,
 -96, -99, -34, -40, -37, -66, -105, 49, -91, -85, -70, -51, 7, 28, -58, -67, 98, -64, -123, -100, 37, 3, -109, -108, 61, -110, 86,
 -103, -111, 60, 10, -49, -47, -76, 52, -106, 8, -72, 20, 37, 126, 31, -48, 93, -58, 108, -96, -38, 34, 38, -30, -103, 9, -47, -67,
 30, 58, 102, -48, -42, 106, -35, 88, 126, 44, 64, 63, 68, 40, -34, 100, -127, -113, 50, 88, -49, -60, 113, -64, -21, -3, -117, -101,
 85, 49, 122]


}
```

27	Code for Endpoint 1 to send a Handshake Packet to Endpoint 2 with Reserve Bits not being 0 (set to “10”):	**PASS**

```
/// Test 27
HandshakePacket hp = new HandshakePacket(qs.getDestConnectionID(), qs.getSourceConnectionID(), packetNum: 0, frames: null);
hp.addFrame(new CryptoFrame( offset: 0, ServerEECertCVFIN()));
hp.setReservedBits(new int[]{ 1, 0 });
qs.sendSinglePacket(hp);
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Handshake packet from Endpoint 1 with Reserved Bits not zeroed out and closes connection due to error:

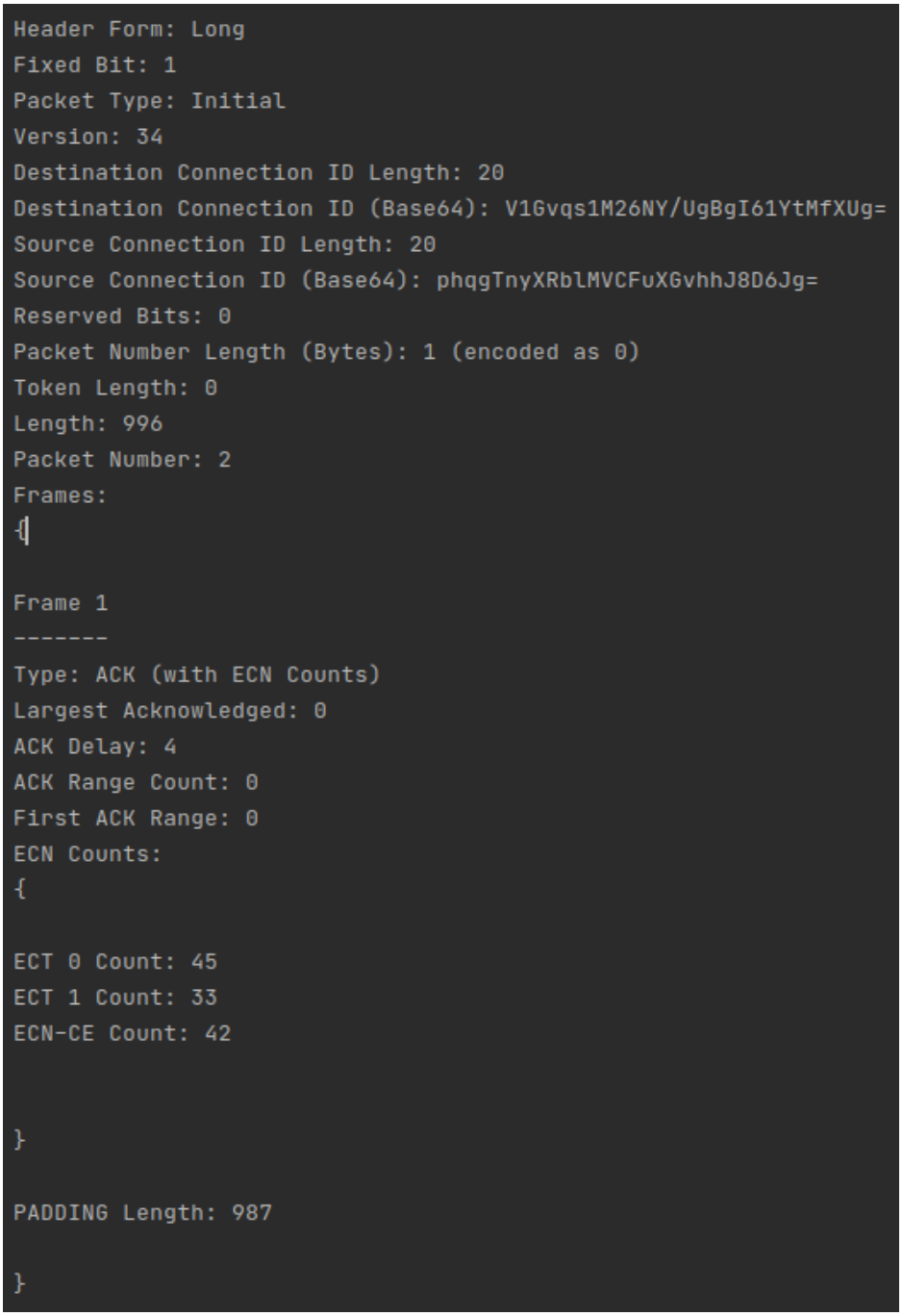
```
New Client Connected: ID gYob5+bY7FJU9zTK30mBkYWf9Jo=
Header Form: Long
Fixed Bit: 1
Packet Type: Handshake
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): gYob5+bY7FJU9zTK30mBkYWf9Jo=
Source Connection ID Length: 20
Source Connection ID (Base64): CIJeD469xWqncOs7wpu9eGuUwkI=
Reserved Bits: 2
Packet Number Length (Bytes): 1 (encoded as 0)
Length: 505
Packet Number: 3
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [115, 49, -27, -13, 61, -22, -87, -59, 95, 21, 14, 40, 97, -121, 33, -37, 1, -35, 47, 8, 53, 96, -45, 117, -47,
  22, 12, -60, -60, 87, 115, -124, -32, -41, 32, 118, 122, 75, -88, 39, -9, -87, -83, 47, -27, 56, -68, 20, -12, -55, -80, -28, 26,
 -38, -73, 10, -107, 94, -63, -55, 49, 8, -95, 101, -7, 95, -124, 8, -19, 40, 122, 16, 4, 13, 36, 122, -100, 82, 18, -64, -114, 91,
 19, 17, 0, -14, -106, 16, 33, 30, -87, -104, -18, -30, 106, -67, 45, -119, -115, -41, 110, 90, 124, 118, 5, -76, 124, 51, -47, 23,
 -124, -83, 86, -81, -127, 75, 110, 16, -7, 82, 50, -36, -32, -76, -29, 114, -3, 122, -110, 31, -105, 49, 122, 86, 108, 92, 21, -72,
 98, 21, 111, 104, -86, 73, 64, 118, -77, 101, -67, 126, -38, 69, -84, -63, -31, 85, 69, -110, -55, -124, 85, -76, 121, 106, -66, 21,
 82, -84, -34, -116, -127, 22, 124, -49, -86, 47, -70, -125, 108, 39, 103, -81, 22, 71, 10, -93, 57, -119, -107, 11, 35, -121, 29,
 -113, 33, -39, -121, -96, -106, 32, 87, 109, 96, 31, -34, 31, -85, -85, -7, -109, 107, 121, 110, 105, -38, -31, -126, -44, -57, 92,
 -74, -40, -79, 45, 30, 20, 48, -32, -97, 76, -117, 90, -121, -119, 2, 124, -114, -47, 74, 51, 53, 55, -121, -90, -116, 6, -55, -88,
 6, 25, -75, 30, -50, -66, -98, 54, 58, -59, -20, 89, -13, -70, 118, -41, -113, -87, -68, 52, 27, 43, 105, 123, 93, 3, -117, -91,
 -124, -126, -115, 37, 85, -36, 45, 99, 85, 104, 120, -81, 92, 64, -42, 74, 55, 107, -27, -117, 45, -120, -49, -84, 26, -66, -6, 80,
 -84, -36, 7, -39, 118, -102, 91, -119, -20, -20, 121, -49, 74, -12, 31, 38, 111, 18, -95, 127, 107, -20, 95, 7, 89, -128, -124, -125,
  2, -22, -8, -98, 96, 29, 114, 95, 79, -17, 2, -1, 52, 28, -6, -69, -45, 31, -74, -51, -123, -45, -102, 6, 96, -82, 122, 65, 122,
 -99, 36, -48, -23, -74, -72, -125, -107, -115, 25, -30, 59, -71, 61, -48, -87, -72, 127, -73, 30, -109, -30, -59, -23, -112, -59, 37,
  119, -53, -73, -3, 117, 60, -111, 10, -110, 40, -26, 49, -89, 106, 63, 86, 68, 29, 125, 25, 35, 12, -108, 106, -64, -27, -75, 16,
 49, 125, 6, -19, -10, 1, -81, 70, -111, 67, -123, -77, -78, -3, -14, -7, 4, 121, 36, -46, -27, 97, 7, 101, -1, -104, 122, -72, -101,
 -121, 15, -80, -103, -115, 112, -7, -43, -56, 13, 47, -94, -8, 117, 58, -76, 35, 82, 9, -94, 31, -25, -94, -43, 87, 60, 121, 59, -22,
  -84, -46, 95, 73, -35, -60, 90, 21, -121, -84, -80, -110, -7, -46, 96, 111, 102, -12, 23, -31, 42, -23, -83, 117, -115, 6]


}

Client ID gYob5+bY7FJU9zTK30mBkYWf9Jo= Disconnected with Error Code 0x0A (PROTOCOL_VIOLATION): Reserved Bits not zero.
```

Endpoint 1 receives connection close from Endpoint 2 with PROTOCOL_VIOLATION error code (0x0A):

	`` Header Form: Short Fixed Bit: 1 Destination Connection ID (Base64): CIJeD469xWqncOs7wpu9eGuUwkI= Reserved Bits: 0 Spin Bit: 0 Key Phase: 0 Packet Number Length (Bytes): 1 (encoded as 0) Packet Number: 1 Frames: { Frame 1 ------- Type: CONNECTION_CLOSE (Application Specific) Error Code: 10 Reason Phrase Length: 22 Reason Phrase: Reserved Bits not zero } Closed Connection to Server gYob5+bY7FJU9zTK30mBkYWf9Jo= with Error Code 0x0A (PROTOCOL_VIOLATION): Reserved Bits not zero. ```	
28	Code for Endpoint 1 (Server) to send a Initial Packet to Endpoint 2 (Client) with Token Length not being 0 (set to 1): ``` /// Test 28 InitialPacket ip = new InitialPacket(client.getDestConnectionID(), client.getSourceConnectionID(), new byte[]{45}, packetNum: 0, frames: null); ip.addFrame(new CryptoFrame(offset: 0, ServerEECertCVFIN())); client.sendSinglePacket(ip); ``` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Handshake packet from Endpoint 1 with Reserved Bits not zeroed out and closes connection due to error:	**PASS**

```
Header Form: Long
Fixed Bit: 1
Packet Type: Initial
Version: 34
Destination Connection ID Length: 20
Destination Connection ID (Base64): EKGDDI4Rd/xfUIUT3N8WTECFbco=
Source Connection ID Length: 20
Source Connection ID (Base64): 2yANJ6LHrFwImtkPkhx2H0HHgzI=
Reserved Bits: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Token Length: 1
Token (Base64): LQ==
Length: 1095
Packet Number: 1
Frames:
{

Frame 1
-------
Type: CRYPTO
Offset: 0
Length: 500
Crypto Data (Base 64): [-116, -16, 29, 35, 15, 122, 124, 65, 114, -101, 83, 121, -100, -99, 32, 78, -59, -122, 2, -87, 73, 82, -88,
 49, -19, 82, -102, 126, -104, -61, -83, 24, -51, -77, 11, 92, -48, -61, 93, -75, 49, 125, 32, 41, 106, -118, 85, -113, -86, 117, 10,
 -28, 95, 55, 20, 94, -47, 33, -30, 2, -76, -78, 54, 21, 66, -47, 99, -56, 27, -58, -96, -48, 57, -119, 85, 109, 120, -19, 4, 117,
 -29, 88, -65, -87, 126, 32, 86, 20, 90, 105, -117, -90, 15, 95, -21, -119, 119, 34, -78, -96, -69, -125, 43, -13, -52, -52, -90, -22,
  87, 55, -100, 87, 117, 38, 14, 93, -76, 48, 107, -33, 125, 13, -53, 126, -26, 25, -124, -25, -97, 45, -96, 76, -69, 78, 52, 75, -68,
  44, -116, -106, 24, 52, -9, -97, -87, -30, 16, 84, 107, 72, 124, -54, 113, 5, -11, 29, 44, 13, 14, 1, 78, -89, -72, -13, 88, -31,
 -6, 21, 95, -105, 20, -76, -24, -18, -37, -25, -2, -24, 63, 11, -103, 13, 97, 22, -67, -68, 103, 22, 4, 96, 96, 74, -77, -83, -19,
 -86, -103, 99, 110, 103, 19, 51, -56, -51, 35, 89, -92, 70, 12, -73, 102, -35, -115, -16, -32, 33, -120, 71, 63, 16, 56, 27, 8, -36,
 78, -24, -77, 17, -100, -93, -121, 77, 20, 113, -102, -124, 27, -42, -34, -85, -46, 27, -94, -42, 14, -3, -99, -2, 76, 76, -55, 1,
 -80, -123, -2, -5, -46, 104, 37, -106, 84, -120, -14, -12, -74, -82, 27, 22, 3, -9, 18, -39, -115, 20, 102, -108, 63, 106, -12, -107,
  -127, -110, -37, 7, 28, -26, -4, -15, 45, -14, -101, 72, 92, -78, 62, -37, 5, -113, 83, 51, 55, -120, -58, 94, -63, 93, 19, -25, 84,
  -121, -5, -66, -24, 10, -3, -86, 81, 26, 71, -33, 116, 125, -105, 6, -71, -69, -52, -123, 117, 117, 99, 74, 5, 22, -27, -83, 64,
 115, 62, 84, -77, -112, -104, 12, 125, 31, 119, 99, 49, 55, -47, 117, 113, 41, 66, 0, 20, 0, 3, -18, 112, -125, -27, 8, -64, -95,
 -30, -74, 56, -17, 125, 50, -54, -5, -101, 121, -49, 107, 118, 36, -13, 95, 102, 65, -3, 123, 102, -96, 88, -59, 13, 47, -127, -76,
 -59, 23, -12, -98, -89, 89, 21, 16, -73, 77, -101, -121, 73, -82, 50, 84, -8, 1, -80, -26, 35, -12, 38, -89, 103, -63, 93, -38, -84,
 -45, 11, 89, 119, -33, 49, -94, -107, 32, 36, -53, -109, 54, 73, 125, 109, -51, -3, -49, -48, -76, 8, 16, 50, 112, -21, 55, -99, 126,
  -91, 123, 0, 77, 107, -90, 77, -99, 82, 20, 40, 36, -65, 104, -100, -101, 64, 95, 48, -32, -14, 26, -112, -27, -46, -7, 66, -86,
 -78, 122, 80, -52, -122, 119, -33, 59, -34, -91, 42, -82, 80, -24, -95, -38, -119, 48, 68, 51]

PADDING Length: 590

}

Closed Connection to Server 2yANJ6LHrFwImtkPkhx2H0HHgzI= with Error Code 0x0A (PROTOCOL_VIOLATION): Token Length more than 0.
```

29	While ECN counts are not used (as Congestion Control is not implemented; see “Implication of Implementation” section for more information), the protocol implementation is able to parse packets that include these ECN Counts including setting the type of ACK frame properly. For example, here is code for Endpoint 1 sending an Initial Packet with an ACK frame that includes ECN Counts to Endpoint 2: `/// Test 29` `InitialPacket ackPacket = new InitialPacket(qs.getDestConnectionID(),qs.getSourceConnectionID(),new byte[]{}, packetNum: 0, frames: null);` `ackPacket.addFrame(new AckFrame( largestAcknowledged: 0, ackDelay: 4, firstAckRange: 0, new ECNCounts( ect0Count: 45, ect1Count: 33, ecnCeCount: 42)));` `qs.sendSinglePacket(ackPacket);` Here is what Endpoint 2 receives:	**PASS**

	 As you can see, the type is shown as “ACK (with ECN Counts)” and ECN counts are shown correctly. Thus, JQUIC is able to send and parse ACK frames with ECN counts correctly.	
30	Code for Endpoint 1 to send a Short (1-RTT) Packet with ACK Frame whose computation results in a negative number:	**PASS**

```
/// Test 30
ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);
sp.addFrame(new AckFrame( largestAcknowledged: 0, ackDelay: 4, firstAckRange: 1));
qs.sendSinglePacket(sp);
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Short packet from Endpoint 1 with ACK frame that results in negative packet number and closes connection due to error:

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): VZiwBn8oFzF0miypRbgKyLUxZ5s=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: ACK
Largest Acknowledged: 0
ACK Delay: 4
ACK Range Count: 0
First ACK Range: 1


}

New Client Connected: ID VZiwBn8oFzF0miypRbgKyLUxZ5s=
Client ID VZiwBn8oFzF0miypRbgKyLUxZ5s= Disconnected with Error Code 0x07 (FRAME_ENCODING_ERROR): ACK number below 0.
```

Endpoint 1 receives connection close from Endpoint 2 with FRAME_ENCODING error code (0x07):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): 8qVBFcUp0J0QtbLVAtG1Q/+Vjrs=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 7
Reason Phrase Length: 18
Reason Phrase: ACK number below 0


}

Closed Connection to Server VZiwBn8oFzF0miypRbgKyLUxZ5s= with Error Code 0x07 (FRAME_ENCODING_ERROR): ACK number below 0.
```

31	Code for Endpoint 1 to open send-only stream with Endpoint 2 (by sending some data to Endpoint 2): `/// Test 31` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` Code for Endpoint 2 to send RESET_STREAM frame to send-only stream on Endpoint 1: `/// Test 31` `QUICConnection.QUICStream quicStream = qs.waitForStream();` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new ResetStreamFrame(quicStream.getStreamID(), applicationProtocolErrorCode: 1, finalSize: 0));` `qs.sendSinglePacket(sp);` Endpoint 1 receives RESET_STREAM frame to its send-only stream and closes connection with STREAM_STATE_ERROR error code (0x05): `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): iuWkJCcwXZ7qZk99C/9ovzUZeZI=` `Reserved Bits: 0` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 2` `Frames:` `{` `Frame 1` `-------` `Type: RESET_STREAM` `StreamID: 3` `Application Protocol Error Code: 1` `Final Size: 0` `}` `Client ID iuWkJCcwXZ7qZk99C/9ovzUZeZI= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): Reset frame sent to send-only stream.` Endpoint 2 receives CONNECTION_CLOSE from Endpoint 1 with STREAM_STATE_ERROR error code (0x05):	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): G7kAyeA3J3E8hgaEAMXUhQtstyQ=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 5
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 36
Reason Phrase: Reset frame sent to send-only stream


}

Closed Connection to Server iuWkJCcwXZ7qZk99C/9ovzUZeZI= with Error Code 0x05 (STREAM_STATE_ERROR): Reset frame sent to send-only
 stream.
```

32	Code for Endpoint 1 to send STOP_SENDING frame to non-existent locally initiated stream (ID 1): `/// Test 32` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StopSendingFrame( streamID: 1, applicationProtocolErrorCode: 1));` `qs.sendSinglePacket(sp);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Short packet from Endpoint 2 with STOP_SENDING frame for non-existent locally initiated stream (ID 1) and closes connection due to error: `New Client Connected: ID /L6dXvqAfmtqQ9+r6tqxpKcoAJE=` `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): /L6dXvqAfmtqQ9+r6tqxpKcoAJE=` `Reserved Bits: 0` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 1` `Frames:` `{` `Frame 1` `-------` `Type: STOP_SENDING` `StreamID: 1` `Application Protocol Error Code: 1` `}` `Client ID /L6dXvqAfmtqQ9+r6tqxpKcoAJE= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): STOP_SENDING frame to non-existent locally-initiated stream.` Endpoint 1 receives CONNECTION_CLOSE from Endpoint 2 with STREAM_STATE_ERROR error code (0x05):	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): FYH7T0j5jkFsHdrmAjQu53QdcCM=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 59
Reason Phrase: STOP_SENDING frame to non-existent locally-initiated stream


}

Closed Connection to Server /L6dXvqAfmtqQ9+r6tqxpKcoAJE= with Error Code 0x05 (STREAM_STATE_ERROR): STOP_SENDING frame to non-existent
  locally-initiated stream.
```

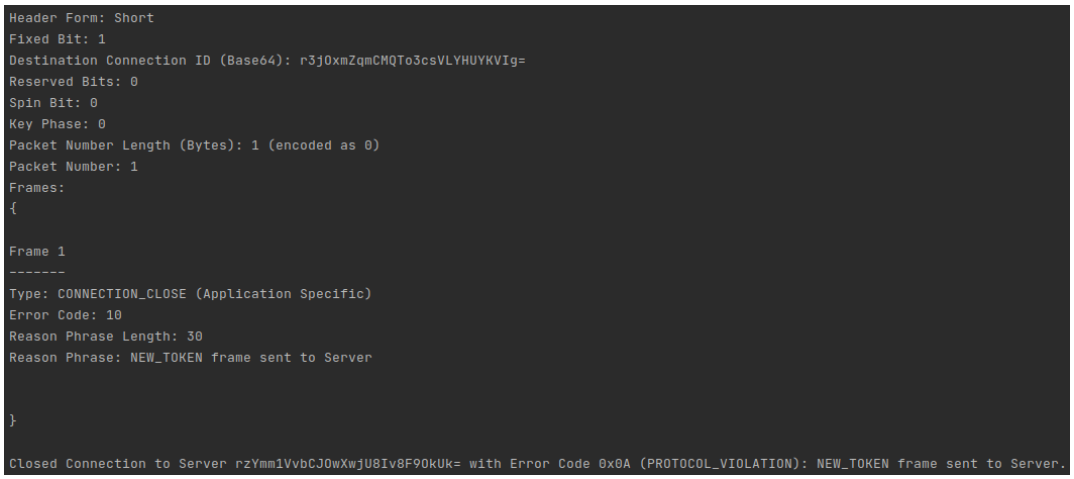
33	Code for Endpoint 1 to open send-only stream with Endpoint 2 (by sending some data to Endpoint 2) and send STOP_SENDING frame to Endpoint 2: `/// Test 33` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` `ShortPacket sp2 = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp2.addFrame(new StopSendingFrame(quicStream.getStreamID(), applicationProtocolErrorCode: 1));` `client.sendSinglePacket(sp2);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives the STOP_SENDING frame from Endpoint 1 and closes connection due to error:	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): w6n8e5YLi46onxsk/tYJUhJhW9U=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 4
Frames:
{

Frame 1
-------
Type: STOP_SENDING
StreamID: 3
Application Protocol Error Code: 1


}

Closed Connection to Server 0fU1xV0qk2zTf/0ODjIQ25y3COY= with Error Code 0x05 (STREAM_STATE_ERROR): Stop Sending frame sent to
  receive-only stream.
```

Endpoint 1 receives CONNECTION_CLOSE from Endpoint 2 with STREAM_STATE_ERROR error code (0x05):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): 0fU1xV0qk2zTf/0ODjIQ25y3COY=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 3
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 46
Reason Phrase: Stop Sending frame sent to receive-only stream


}

Client ID 0fU1xV0qk2zTf/0ODjIQ25y3COY= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): Stop Sending frame sent to receive-only
  stream.
```

34	Code for Endpoint 1 (Client) to send Short Packet with NEW_TOKEN frame to Endpoint 2 (Server): `/// Test 34` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new NewTokenFrame(new byte[]{45, 44}));` `qs.sendSinglePacket(sp);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives the Short Packet with NEW_TOKEN frame from Endpoint 1 and closes connection due to error:	**PASS**

	 Endpoint 1 receives CONNECTION_CLOSE from Endpoint 2 with PROTOCOL_VIOLATION error code (0x0A): 	
35	Code for Endpoint 1 (Client) to send STREAM frame to non-existent locally initiated stream (ID 1) on Endpoint 2 (Server):	**PASS**

```
/// Test 35
ShortPacket sp = new ShortPacket(qs.getDestConnectionID(),  packetNum: 0,  frames: null, (byte) 0, (byte) 0);
sp.addFrame(new StreamFrame( streamID: 1,  offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,
            fin: false, "hello".getBytes(StandardCharsets.UTF_8)));
qs.sendSinglePacket(sp);
```

There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Short packet from Endpoint 1 with STREAM frame for non-existent locally initiated stream (ID 1) and closes connection due to error:

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): m1vzWlsXEzVGl+YppI210yZ/LKY=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: STREAM (LEN Bit Set)
StreamID: 1
Length: 5
Fin Bit: Not Set
Data: [104, 101, 108, 108, 111]


}

Client ID m1vzWlsXEzVGl+YppI210yZ/LKY= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): STREAM frame to non-existent
 locally-initiated stream.
```

Endpoint 1 receives CONNECTION_CLOSE from Endpoint 2 with STREAM_STATE_ERROR error code (0x05):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): nWr+M3M0J4k40XHhutQrA5DqoJ4=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 53
Reason Phrase: STREAM frame to non-existent locally-initiated stream


}

Closed Connection to Server m1vzWlsXEzVGl+YppI210yZ/LKY= with Error Code 0x05 (STREAM_STATE_ERROR): STREAM frame to non-existent
 locally-initiated stream.
```

36	Code for Endpoint 1 to open send-only stream with Endpoint 2 (by sending some data to Endpoint 2): `/// Test 31, 36` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` Code for Endpoint 2 to send STREAM frame to send-only stream on Endpoint 1: `/// Test 36` `QUICConnection.QUICStream quicStream = qs.waitForStream();` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `qs.sendSinglePacket(sp);` Endpoint 1 receives STREAM frame to its send-only stream and closes connection with STREAM_STATE_ERROR error code (0x05): `Header Form: Short` `Fixed Bit: 1` `Destination Connection ID (Base64): B6QTQ7uRzzph+5xAzX0Ff1gtITc=` `Reserved Bits: 0` `Spin Bit: 0` `Key Phase: 0` `Packet Number Length (Bytes): 1 (encoded as 0)` `Packet Number: 2` `Frames:` `{` `Frame 1` `-------` `Type: STREAM (LEN Bit Set)` `StreamID: 3` `Length: 5` `Fin Bit: Not Set` `Data: [104, 101, 108, 108, 111]` `}` `Client ID B6QTQ7uRzzph+5xAzX0Ff1gtITc= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): STREAM frame sent to send-only stream.` Endpoint 2 receives CONNECTION_CLOSE from Endpoint 1 with STREAM_STATE_ERROR error code (0x05):	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): DBxk/iob1ksY4NLFHd2w2VVUZP4=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 5
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 37
Reason Phrase: STREAM frame sent to send-only stream


}

Closed Connection to Server B6QTQ7uRzzph+5xAzX0Ff1gtITc= with Error Code 0x05 (STREAM_STATE_ERROR): STREAM frame sent to send-only
 stream.
```

37	Code for Endpoint 1 to open send-only stream with Endpoint 2 (by sending some data to Endpoint 2): `/// Test 37` `QUICConnection.QUICStream quicStream = qs.waitForStream();` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamDataBlockedFrame(quicStream.getStreamID(), maximumStreamData: 100));` `qs.sendSinglePacket(sp);` Code for Endpoint 2 to send STREAM_DATA_BLOCKED frame to send-only stream on Endpoint 1: `/// Test 31, 36, 37` `QUICConnection.QUICStream quicStream = client.createStream( bidirectional: false);` `ShortPacket sp = new ShortPacket(client.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new StreamFrame(quicStream.getStreamID(), offset: 0, "hello".getBytes(StandardCharsets.UTF_8).length,` `fin: false, "hello".getBytes(StandardCharsets.UTF_8)));` `client.sendSinglePacket(sp);` Endpoint 1 receives STREAM_DATA_BLOCKED frame to its send-only stream and closes connection with STREAM_STATE_ERROR error code (0x05):	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): 9tt9gihVpXaVCsfheyFV7BfWSkk=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 2
Frames:
{

Frame 1
-------
Type: STREAM_DATA_BLOCKED
StreamID: 3
Maximum Stream Data: 100


}

Client ID 9tt9gihVpXaVCsfheyFV7BfWSkk= Disconnected with Error Code 0x05 (STREAM_STATE_ERROR): STREAM_DATA_BLOCKED frame sent to
 send-only stream.
```

Endpoint 2 receives CONNECTION_CLOSE from Endpoint 2 with STREAM_STATE_ERROR error code (0x05):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): LtbuGsuk43vl58569omcvEhM0IQ=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 5
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 5
Reason Phrase Length: 50
Reason Phrase: STREAM_DATA_BLOCKED frame sent to send-only stream


}

Closed Connection to Server 9tt9gihVpXaVCsfheyFV7BfWSkk= with Error Code 0x05 (STREAM_STATE_ERROR): STREAM_DATA_BLOCKED frame sent to
 send-only stream.
```

38	Code for Endpoint 1 (Client) to send Short Packet with HANDSHAKE_DONE frame to Endpoint 2 (Server): `/// Test 38` `ShortPacket sp = new ShortPacket(qs.getDestConnectionID(), packetNum: 0, frames: null, (byte) 0, (byte) 0);` `sp.addFrame(new HandshakeDoneFrame());` `qs.sendSinglePacket(sp);` There is no unique testing code for Endpoint 2 as just having a connection with Endpoint 1 will make it automatically start to receive and process packets. Endpoint 2 receives Short packet from Endpoint 1 with HANDSHAKE_DONE frame and closes connection due to error:	**PASS**

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): pfF6TQQJYZ0BUpHHlzBK36fgi50=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: HANDSHAKE_DONE


}

New Client Connected: ID pfF6TQQJYZ0BUpHHlzBK36fgi50=
Client ID pfF6TQQJYZ0BUpHHlzBK36fgi50= Disconnected with Error Code 0x0A (PROTOCOL_VIOLATION): HANDSHAKE_DONE frame sent to Server.
```

Endpoint 1 receives CONNECTION_CLOSE from Endpoint 2 with PROTOCOL_VIOLATION error code (0x0A):

```
Header Form: Short
Fixed Bit: 1
Destination Connection ID (Base64): WsTc+4ciVdSPYPl/WgsWWeS2N08=
Reserved Bits: 0
Spin Bit: 0
Key Phase: 0
Packet Number Length (Bytes): 1 (encoded as 0)
Packet Number: 1
Frames:
{

Frame 1
-------
Type: CONNECTION_CLOSE (Application Specific)
Error Code: 10
Reason Phrase Length: 35
Reason Phrase: HANDSHAKE_DONE frame sent to Server


}

Closed Connection to Server pfF6TQQJYZ0BUpHHlzBK36fgi50= with Error Code 0x0A (PROTOCOL_VIOLATION): HANDSHAKE_DONE frame sent to Server.
```

Final Verdict: 38 of 38 Manual Tests passed.

2.6.2 Regression/Automated Tests

There are 88 asserts captured in 39 tests and broken down into five different test files:

- **FrameToBytesTest:** Tests for each frame type to see if they can be converted to and from byte array. This is important as this is the process that takes place when the frame is sent and received. Here is an example of tests for AckFrame:

```java
@Test
public void AckFrameAndRangesToBytes() {
    AckFrame af = new AckFrame( largestAcknowledged: 5, ackDelay: 5, firstAckRange: 5);
    Assert.assertEquals((new AckFrame(af.getBytes(), frame_offset: 0)).toString(), af.toString());
    AckRange ar = new AckRange( gap: 4, ackRangeLength: 5);
    AckRange ar2 = new AckRange( gap: 10, ackRangeLength: 6);
    ArrayList<AckRange> ackRanges = new ArrayList<>(Arrays.asList(ar, ar2));
    af = new AckFrame( largestAcknowledged: 543, ackDelay: 5, firstAckRange: 5, ackRanges);
    Assert.assertEquals((new AckFrame(af.getBytes(), frame_offset: 0)).toString(), af.toString());
    ECNCounts ecnc = new ECNCounts( ect0Count: 45, ect1Count: 33, ecnCeCount: 32);   Boya, Today • Done
    af = new AckFrame( largestAcknowledged: 543, ackDelay: 5, firstAckRange: 5, ecnc);
    Assert.assertEquals((new AckFrame(af.getBytes(), frame_offset: 0)).toString(), af.toString());
    af = new AckFrame( largestAcknowledged: 543, ackDelay: 5, firstAckRange: 5, ackRanges, ecnc);
    Assert.assertEquals((new AckFrame(af.getBytes(), frame_offset: 0)).toString(), af.toString());
}
```

- **HelperFuncsTest**: Tests for static functions in class HelperFuncs including for conversion to and from Variable Length Integer, custom exponent function, getting bits from a byte function and getting packet number length in bytes. Some of these will be discussed further in the subsection below, but here is a simple “pow” (exponent) function test:

```java
@Test
public void testPow() {
    Assert.assertThat(pow((byte) 0, power: 0), CoreMatchers.is( value: 1L));
    Assert.assertThat(pow((byte) 0, power: 1), CoreMatchers.is( value: 0L));
    Assert.assertThat(pow((byte) 1, power: 0), CoreMatchers.is( value: 1L));
    Assert.assertThat(pow((byte) 1, power: 1), CoreMatchers.is( value: 1L));
    Assert.assertThat(pow((byte) 2, power: 1), CoreMatchers.is( value: 2L));
    Assert.assertThat(pow((byte) 2, power: 2), CoreMatchers.is( value: 4L));
    Assert.assertThat(pow((byte) 2, power: 3), CoreMatchers.is( value: 8L));
}
```

- **PacketToBytesTest**: Similar to “FrameToBytesTest”, theses tests are for each implemented packet type to see if they can be converted to and from byte array. Again, this is important as this is the process that takes place when the packet is sent and received. For example, here is the test for Handshake packet:

```java
@Test
public void testHandshakePacketToBytes() {
    byte[] destConnectionID = new byte[CONNECTION_ID_LENGTH];
    try {
        SecureRandom.getInstanceStrong().nextBytes(destConnectionID);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
Boya, Today • Done Regression/Automated Tests
    byte[] srcConnectionID = new byte[CONNECTION_ID_LENGTH];
    try {
        SecureRandom.getInstanceStrong().nextBytes(srcConnectionID);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    ArrayList<Frame> handshakeFrames = new ArrayList<>();
    handshakeFrames.add(new CryptoFrame( offset: 0, ServerEECertCVFIN()));

    HandshakePacket hp = new HandshakePacket(destConnectionID, srcConnectionID,  packetNum: 45, handshakeFrames);

    Assert.assertEquals((new HandshakePacket(hp.getBytes(),  packet_offset: 0)).toString(), hp.toString());
}
```

- **ParserTest**: this tests whether PacketParser and FrameParser can properly parse UDP datagrams with coalesced packets similar to the Manual Test 18, but this one is done in an automated way:

```java
@Test
public void parserTest() {

    byte[] destConnectionID = new byte[CONNECTION_ID_LENGTH];
    try {
        SecureRandom.getInstanceStrong().nextBytes(destConnectionID);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    byte[] sourceConnectionID = new byte[CONNECTION_ID_LENGTH];
    try {
        SecureRandom.getInstanceStrong().nextBytes(sourceConnectionID);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    ArrayList<Frame> initialFrames = new ArrayList<>();
    initialFrames.add(new CryptoFrame( offset: 0, ServerHello()));
    initialFrames.add(new CryptoFrame( offset: 0, ServerEECertCVFIN()));
    ArrayList<Frame> handshakeFrames = new ArrayList<>();
    handshakeFrames.add(new CryptoFrame( offset: 0, ServerEECertCVFIN()));

    InitialPacket initialPacket = new InitialPacket(destConnectionID, sourceConnectionID, new byte[]{}, packetNum: 0, initialFrames);
    HandshakePacket handshakePacket = new HandshakePacket(destConnectionID, sourceConnectionID, packetNum: 0, handshakeFrames);

    byte[] totalPayload = new byte[initialPacket.getBytes().length + handshakePacket.getBytes().length];
    System.arraycopy(initialPacket.getBytes(), i: 0, totalPayload, i1: 0, initialPacket.getBytes().length);
    System.arraycopy(handshakePacket.getBytes(), i: 0, totalPayload, initialPacket.getBytes().length, handshakePacket.getBytes().length);

    ArrayList<Packet> retrievedPackets = PacketParser.parsePackets(totalPayload);
    Assert.assertEquals(retrievedPackets.size(), actual: 2);
    Assert.assertEquals(retrievedPackets.get(0).toString(), initialPacket.toString());
    Assert.assertEquals(retrievedPackets.get(1).toString(), handshakePacket.toString());
}
```

- **TLSMockTest:** This just tests the TLSMock function to see if they produce the correct lengths as that currently determines what type of Crypto data it is during the handshake:

```java
@Test
public void lengthTest() {
    Assert.assertEquals( expected: 307, TLSMock.ClientHello().length);
    Assert.assertEquals( expected: 96, TLSMock.ServerHello().length);
    Assert.assertEquals( expected: 500, TLSMock.ServerEECertCVFIN().length);
    Assert.assertEquals( expected: 100, TLSMock.ClientFIN().length);
}
```

2.6.2.1 Variable-Length Integer Tests

QUIC protocol has a feature called “Variable Length Integer” for many of the fields in its frames and packets to decrease the size of its packet by encoding the byte length of the field in 2 most significant bits. To do this in JQUIC implementation,

there are two different functions to convert longs to and from variable length integer: “convertNumtoVarLenInt” and “convertVarLenIntToNum”. These two functions are tested using automated tests by testing the limit for each byte length to see whether it is encoded and decoded correctly. Here are those limits as shown in the protocol draft (Iyengar and Thomson):

2Bit	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 4: Summary of Integer Encodings

And here are the tests for “convertNumToVarLenInt”:

```
@Test
public void testConvertNumToVarLenInt() {
    Assert.assertThat(convertNumToVarLenInt(0).length, CoreMatchers.is( value: 1));
    Assert.assertThat(convertNumToVarLenInt(0), CoreMatchers.is(new byte[]{0}));
    Assert.assertThat(convertNumToVarLenInt(1).length, CoreMatchers.is( value: 1));
    Assert.assertThat(convertNumToVarLenInt(1), CoreMatchers.is(new byte[]{1}));
    Assert.assertThat(convertNumToVarLenInt(63).length, CoreMatchers.is( value: 1));
    Assert.assertThat(convertNumToVarLenInt(63), CoreMatchers.is(new byte[]{63}));
    Assert.assertThat(convertNumToVarLenInt(64).length, CoreMatchers.is( value: 2));
    Assert.assertThat(convertNumToVarLenInt(64), CoreMatchers.is(new byte[]{64,64}));
    Assert.assertThat(convertNumToVarLenInt(16383).length, CoreMatchers.is( value: 2));
    Assert.assertThat(convertNumToVarLenInt(16383), CoreMatchers.is(new byte[]{127,-1}));
    Assert.assertThat(convertNumToVarLenInt(16384).length, CoreMatchers.is( value: 4));
    Assert.assertThat(convertNumToVarLenInt(16384), CoreMatchers.is(new byte[]{-128,0,64,0}));
    Assert.assertThat(convertNumToVarLenInt(1073741823).length, CoreMatchers.is( value: 4));
    Assert.assertThat(convertNumToVarLenInt(1073741823), CoreMatchers.is(new byte[]{-65,-1,-1,-1}));
    Assert.assertThat(convertNumToVarLenInt(1073741824).length, CoreMatchers.is( value: 8));
    Assert.assertThat(convertNumToVarLenInt(1073741824), CoreMatchers.is(new byte[]{-64,0,0,0,64,0,0,0}));
    Assert.assertThat(convertNumToVarLenInt(4611686018427387903L).length, CoreMatchers.is( value: 8));
    Assert.assertThat(convertNumToVarLenInt(4611686018427387903L), CoreMatchers.is(new byte[]{-1,-1,-1,-1,-1,-1,-1,-1}));
}
```

As you can see, each range gets tested. This is also true for non-exception throwing tests for “convertVarLenIntToNum”:

```
@Test
public void testConvertVarLenIntToNum() {
    Assert.assertThat(convertVarLenIntToNum(new byte[]{0}), CoreMatchers.is( value: 0L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{1}), CoreMatchers.is( value: 1L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{63}), CoreMatchers.is( value: 63L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{64,64}), CoreMatchers.is( value: 64L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{127,-1}), CoreMatchers.is( value: 16383L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{-128,0,64,0}), CoreMatchers.is( value: 16384L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{-65,-1,-1,-1}), CoreMatchers.is( value: 1073741823L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{-64,0,0,0,64,0,0,0}), CoreMatchers.is( value: 1073741824L));
    Assert.assertThat(convertVarLenIntToNum(new byte[]{-1,-1,-1,-1,-1,-1,-1,-1}), CoreMatchers.is( value: 4611686018427387903L));
}
```

Whether those functions throw exceptions when the parameters given to them are out of bounds of Integer Encoding ranges shown above is also tested:

```
@Test(expected = IllegalArgumentException.class)
public void testConvertNumToVarLenIntThrowsIllegalArgumentExceptionForLowerLimit() { convertNumToVarLenInt(-1); }

@Test(expected = IllegalArgumentException.class)
public void testConvertNumToVarLenIntThrowsIllegalArgumentExceptionForUpperLimit() {
    convertNumToVarLenInt(4611686018427387904L);
}
```

```
@Test(expected = IllegalArgumentException.class)
public void testConvertVarLenIntToNumThrowsIllegalArgumentExceptionForArrayLength() {
    convertVarLenIntToNum(new byte[]{-1,-1,-1,-1,-1,-1,-1,-1,-1});
}

@Test(expected = IllegalArgumentException.class)
public void testConvertVarLenIntToNumThrowsIllegalArgumentExceptionForEmptyArray() {
    convertVarLenIntToNum(new byte[]{});
}
```

By doing these four tests above, the following requirements from the draft are satisfied:

- “The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the offset and data length - cannot exceed 2^62-1”
- “[Maximum Streams] value cannot exceed 2^60, as it is not possible to encode stream IDs larger than 2^62-1”
- “A stream ID is a 62-bit integer (0 to 2^62-1) that is unique for all streams on a connection”
- “The largest offset delivered on a stream - the sum of the offset and data length - cannot exceed 2^62-1. “

This is because you cannot possibly encode number smaller than 0 and larger than 2^62 - 1 in a QUIC packet or frame and an exception is thrown if that range is exceeded as verified by the tests.

All the tests shown here are located in “HelperFuncsTest” file.

2.6.2.2 GetPacketNumberLengthInBytes Tests

Packet number is also encoded (but as discussed in “Implications of Implementation”, not in a fully correct way in this current implementation). It is encoded in the two least significant bytes of the the first bytes for those packets that have actually contain the packet number (like Initial, Handshake and Short (1-RTT) packets). To encode that length, a function called “GetPacketNumberLengthInBytes” which takes in a number and return the amount of bytes required to store that value in the packet minus one. This is tested in “HelperFuncsTest” file. Here are the tests for this function:

```
@Test
public void testGetPacketNumberLengthInBytes() {
    Assert.assertThat(getPacketNumberLengthInBytes(0), CoreMatchers.is( value: 0));
    Assert.assertThat(getPacketNumberLengthInBytes(1), CoreMatchers.is( value: 0));
    Assert.assertThat(getPacketNumberLengthInBytes(255), CoreMatchers.is( value: 0));
    Assert.assertThat(getPacketNumberLengthInBytes(256), CoreMatchers.is( value: 1));
    Assert.assertThat(getPacketNumberLengthInBytes(65535), CoreMatchers.is( value: 1));
    Assert.assertThat(getPacketNumberLengthInBytes(65536), CoreMatchers.is( value: 2));
    Assert.assertThat(getPacketNumberLengthInBytes(16777215), CoreMatchers.is( value: 2));
    Assert.assertThat(getPacketNumberLengthInBytes(16777216), CoreMatchers.is( value: 3));
    Assert.assertThat(getPacketNumberLengthInBytes(4294967295L), CoreMatchers.is( value: 3));
}
```

As you can see, it tests each range from 0 to 3 bytes. There are also tests for upper and lower limits for packet number (0 to 2^32-1):

```
@Test(expected = IllegalArgumentException.class)
public void testGetPacketNumberLengthInBytesForUpperLimit() {
    getPacketNumberLengthInBytes(4294967296L);
}

@Test(expected = IllegalArgumentException.class)
public void testGetPacketNumberLengthInBytesForLowerLimit() { getPacketNumberLengthInBytes(-1); }
```

While this does satisfy the requirement that “If the packet number for sending reaches 2^62 - 1, the sender MUST close the connection without sending a

CONNECTION_CLOSE frame or any further packets.", it unfortunately does so because the current implementation has a much lower limit which again will be explained in "Implications of Implementation".

2.6.2.3 GetBit Tests

GetBit is a small function for getting bits at a specific position from a byte (using Little-Endian sequence). Here are the tests for this:

```
@Test
public void testGetBit() {
    Assert.assertThat(getBit((byte) 0,  pos: 0), CoreMatchers.is( value: 0));
    Assert.assertThat(getBit((byte) 1,  pos: 0), CoreMatchers.is( value: 1));
    Assert.assertThat(getBit((byte) 2,  pos: 0), CoreMatchers.is( value: 0));
    Assert.assertThat(getBit((byte) 2,  pos: 1), CoreMatchers.is( value: 1));
    Assert.assertThat(getBit((byte) 3,  pos: 0), CoreMatchers.is( value: 1));
}
```

All the tests shown here are located in "HelperFuncsTest" file.

2.6.2.4 Regression/Automated Test Results

All the regression/automated tests passed successfully:

```
✔ Tests passed: 39 of 39 tests – 75 ms

> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
Deprecated Gradle features were used in this build, making it incompatible with Gradle 7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/6.7/userguide/command_line_interface.html#sec:command_line_warnings
BUILD SUCCESSFUL in 608ms
3 actionable tasks: 2 executed, 1 up-to-date
3:15:50 p.m.: Task execution finished ':test --tests *'.
```

2.6.3 Acceptance Tests

Acceptance tests are done specifically for QUIC Distributed File Server(DFS) Application to manually test its functionality and to confirm it works correctly. The following three machines will be used for this test:

- **UPSILON** (IP 192.168.1.78): Fedora Linux machine that will be used to run the client for QUIC DFS; it has OpenJDK 11 installed and configured to be used
- **OMICRON** (IP 192.168.1.72): Fedora Linux machine used to run the server for QUIC DFS; it has OpenJDK 11 installed and configured to be used
- **TAU** (IP 192.168.1.79)**:** Fedora Linux VM used to run the second client for QUIC DFS; it has OpenJDK 11 installed and configured to be used

Here are the acceptance tests:

No.	Test Description	Passing Criteria
1	Running client on **UPSILON** with no arguments (Full command: ‘./gradlew -q --console=plain runClient’)	Client shows a “no arguments” error, usage and help message and then exits
2	Running server on **OMICRON** with no arguments (Full command: ‘./gradlew -q --console=plain runServer’)	Server shows a “no arguments” error and usage message and then exits
3	Running client on **UPSILON** with arguments “192.168.1.72 1024”(Full command: ‘./gradlew -q --console=plain runClient --args “192.168.1.72 1025”’) with no server running on **OMICRON**	Client times out after trying to connect for 10 seconds
4	Running server on **OMICRON** with arguments “1024 files” (Full Command: ‘./gradlew -q --console=plain runServer --args "1024 files"’).	Server outputs a message that the server has been started on UDP port 1024
5	Running client on **UPSILON** with arguments similar to Acceptance Test 3 while server on **OMICRON** is already	Client on **UPSILON** connects to server on **OMICRON.** Assume

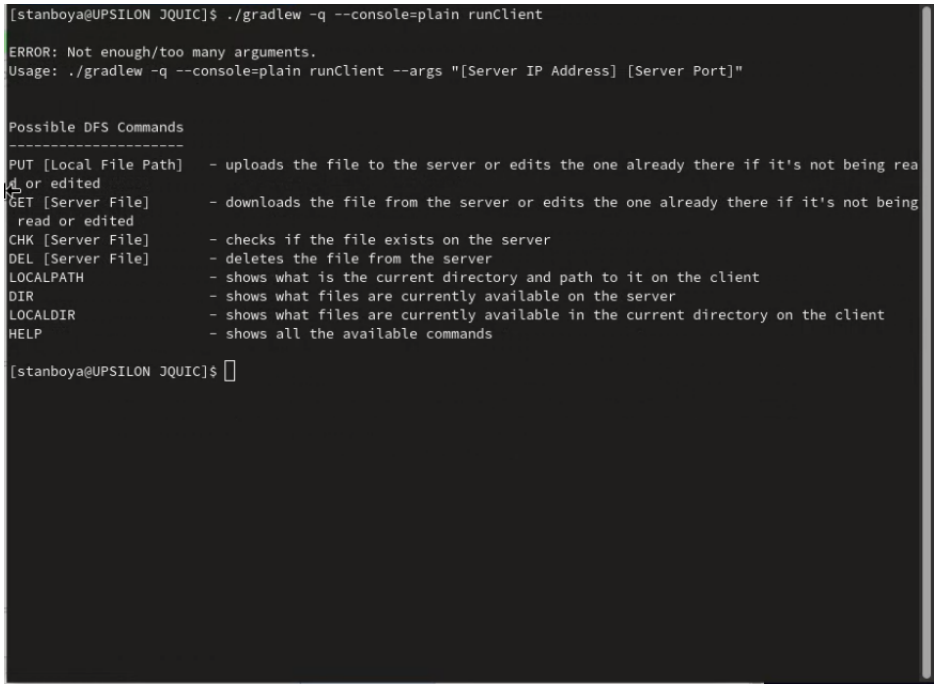
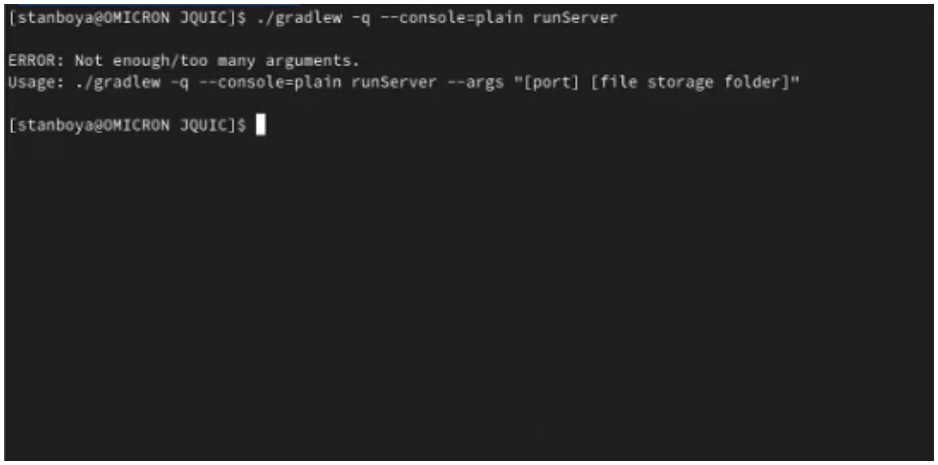
	running with arguments similar to Acceptance Test 4.	the client is connected to the server for the next tests unless stated otherwise.
6	Trying command “PUT test.txt” in client on **UPSILON** with “test.txt” not existing on **UPSILON**	Client on **UPSILON** prints out file error message and asks for another command
7	Trying command “CHK test.txt” in client on **UPSILON** with “test.txt” being a small text file not present on **OMICRON**	Client on **UPSILON** outputs that the file “test.txt” is not available on the server on **OMICRON** and asks for another command
8	Trying command “PUT test.txt” in client on **UPSILON** with “test.txt” being a small text file on **UPSILON**	Client on **UPSILON** sends the file “test.txt” to **OMICRON** and asks for another command; file “test.txt” can be seen on **OMICRON** in “files” folder
9	Trying command “CHK test.txt” in client on **UPSILON** with “test.txt” being a small text file now present on **OMICRON** from the previous test	Client on **UPSILON** outputs that the file “test.txt” is available on the server on **OMICRON**
10	Trying command “PUT midSizeTextFile.txt” in client on **UPSILON** with “midSizeTextFile.txt” being around 1MB text file (Moby Dick from gutenburg.org: http://www.gutenberg.org/ebooks/2701) on **UPSILON**	Client on **UPSILON** sends the file “midSizeTextFile.txt” to **OMICRON,** outputs success message and asks for another command; file “midSizeTextFile.txt” can be seen on **OMICRON** in “files” folder
11	Trying command “DIR” in client on **UPSILON**	Client on **UPSILON** shows the files being available in the storage folder for the server on **OMICRON,** specifically “test.txt” and “midSizeTextFile.txt”
12	Trying command “DEL test.txt” in client on **UPSILON**	Client on **UPSILON** outputs a message saying that the file has been successfully deleted in Server on **OMICRON;** File “test.txt” is not in the storage

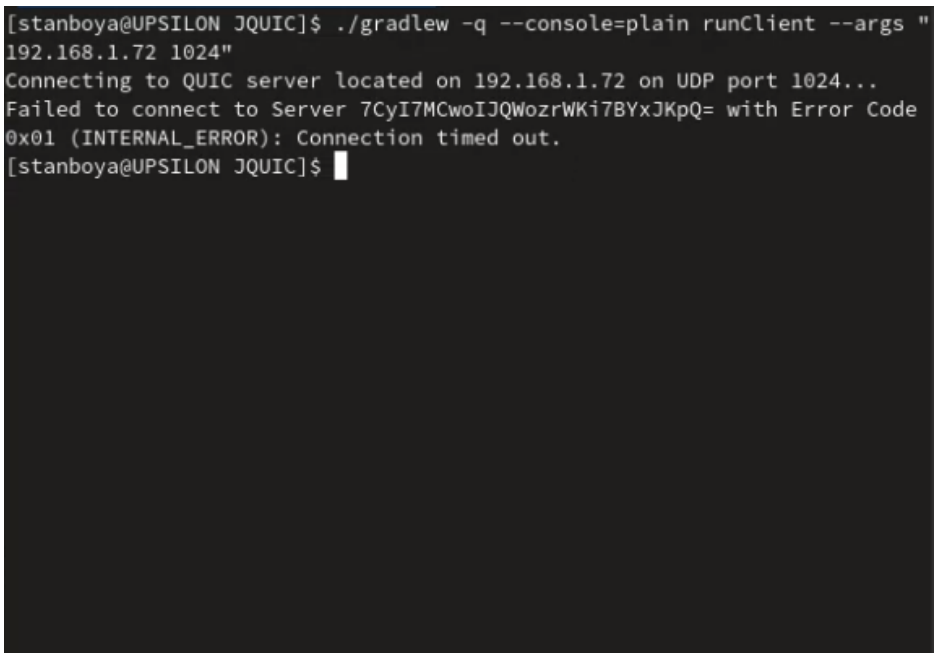
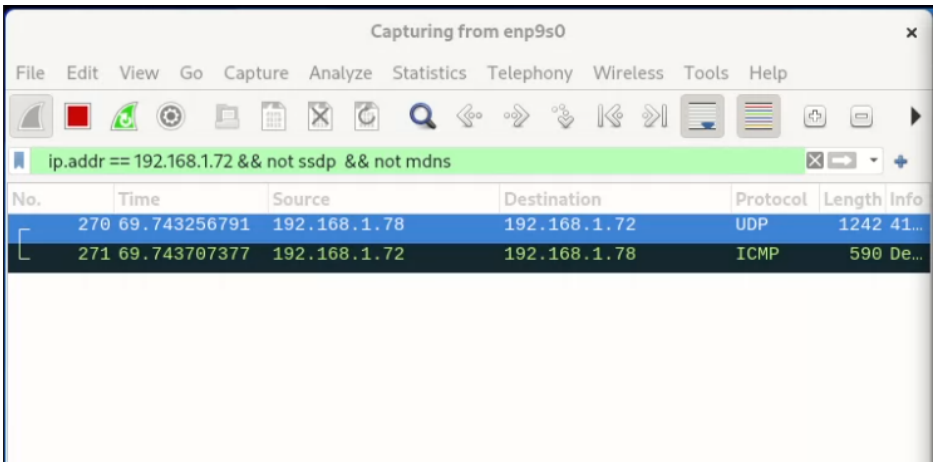
		folder for the Server on **OMICRON**
13	Trying command “GET test.txt” in client on **UPSILON** with “test.txt” being a small text file in the storage folder for the server on **OMICRON;** Note that for this test, “test.txt” is absent on **UPSILON** and not in storage folder for server in **OMICRON**	Client on **UPSILON** outputs a failure message saying that the file was not successfully downloaded from the server as it does not exist there and asks for another command
14	Trying command “GET midSizeTextFile.txt” in client on **UPSILON** with “midSizeTextFile.txt” being around 1MB text file in the storage folder for the server on **OMICRON;** Note that for this test, “midSizeTextFile.txt” is absent on **UPSILON**	Client on **UPSILON** outputs a success message saying that the file is successfully downloaded from the server and asks for another command; the file “midSizeTextFile.txt” is present in the directory the DFS client application was ran from on **UPSILON**
15	Trying command “HELP” in client on **UPSILON**	The help message is output and application asks for another command on **UPSILON**
16	Client **TAU** tries to connect to **OMICRON** while **UPSILON** is already connected to **OMICRON**	**TAU** successfully connects to **OMICRON**; Assume that both **TAU** and **UPSILON** are connected to **OMICRON** for the next tests unless stated otherwise
17	Client **UPSILON** tries to upload a file “midSizeTextFile.txt” to **OMICRON** using PUT command similar to Acceptance Test 10; Client **TAU** tries the same command while **UPSILON** is still uploading; Note that both **UPSILON** and **TAU** have the file “midSizeTextFile.txt” ready to upload	**UPSILON** successfully uploads the file, but **TAU** fails and outputs a message saying that the file that it tried to upload is currently being edited by another client
18	Client **UPSILON** tries to upload a file “midSizeTextFile.txt” to **OMICRON** using PUT command similar to Acceptance Test 10; Client **TAU** tries the same command, but with a different file called “test.txt” while **UPSILON** is still uploading; Note that **UPSILON** has the file “midSizeTextFile.txt” ready to upload	Both **UPSILON** and **TAU** are able to successfully upload the files to the server

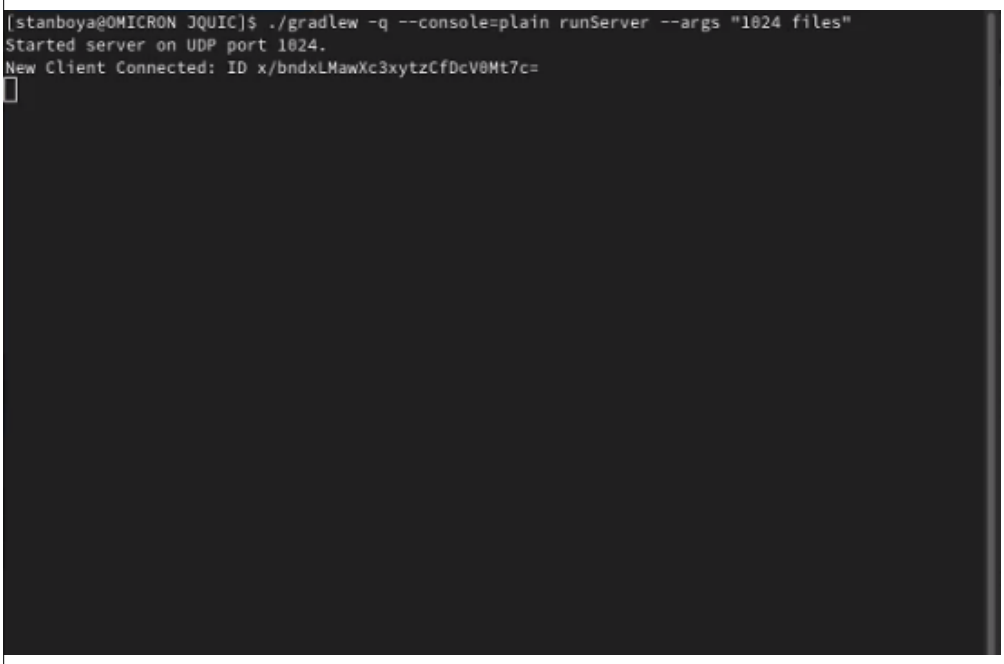
	and **TAU** has the file “test.txt”	
19	Client **UPSILON** tries to upload a file “midSizeTextFile.txt” to **OMICRON** using PUT command similar to Acceptance Test 10; Client **TAU** tries to get file “midSizeTextFile.txt” from **OMICRON** similar to Acceptance Test 13 while **UPSILON** is still uploading; Note that **UPSILON** has the file “midSizeTextFile.txt” ready to upload and **OMICRON** has that file in the storage folder for the server so it would just be a file update from **UPSILON**	**UPSILON** successfully uploads the file, but **TAU** fails to get the file from the server on **OMICRON** and outputs a message saying that the file that it tried to download is currently being edited by another client
20	Clients **UPSILON** and **TAU** try to download the file “midSizeTextFile.txt” from the server on **OMICRON** at the same time with both running the command similar to Acceptance Test 13; Note that **OMICRON** has the file “midSizeTextFile.txt” in the storage folder for the server	**UPSILON** successfully downloads the file, but **TAU** fails to get the file from the server on **OMICRON** and outputs a message saying that the file that it tried to download is currently being edited by another client
21	Client **UPSILON** tries to get a file “midSizeTextFile.txt” from the server on **OMICRON** using GET command similar to Acceptance Test 13; Client **TAU** tries to delete that same file from OMICRON using DEL command similar to Acceptance Test 12; Note that **OMICRON** has the file “midSizeTextFile.txt” in the storage folder for the server	**UPSILON** successfully downloads the file, but **TAU** fails to delete the file from the server on **OMICRON** and outputs a message saying that the file that it tried to delete is currently being edited by another client
22	Trying command “EXIT” in client on **TAU**	**TAU** disconnects from **OMICRON**, but **UPSILON** still stays connected and can execute commands (tested by running command ‘DIR’ and getting output back on **UPSILON** similar to Acceptance Test 11)
23	Running server on **OMICRON** with arguments “1024 files” similar to Acceptance Test 4 and then another server on **OMICRON** with same arguments	The first server starts correctly similar to result in Acceptance Test 4, but the second server outputs “port already in use” error and exits
24	Client **UPSILON** tries to download a file “midSizeTextFile.txt”	Both **UPSILON** and **TAU** are able

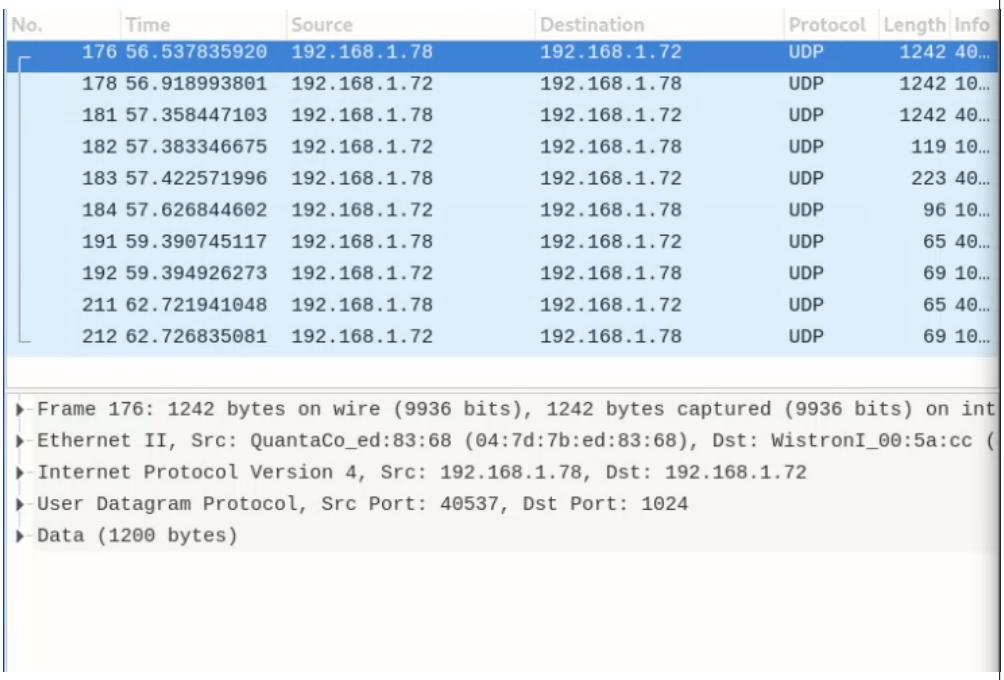
	from **OMICRON** using GET command similar to Acceptance Test 14; Client **TAU** tries the same command, but with a different file called “test.txt” while **UPSILON** is still downloading; Note that server on **OMICRON** has both of those files	to successfully download the files to the server
25	Trying command “LOCALDIR” in client on **UPSILON**	All the contents of the directory that the program was run in are shown on **UPSILON**
26	Trying command “LOCALPATH” in client on **UPSILON**	The absolute path of the directory that the program was run in are shown on **UPSILON**
27	Closing client on **UPSILON** instantly without graceful shutdown (using CTRL+C command in terminal) while it is connected to server on **OMICRON**; Trying to connect to server **OMICRON** from **UPSILON** again and then trying command “DIR” and then exiting again; assume that **OMICRON** and **UPSILON** are connected at the start of this test similar to Acceptance Test 5	A timeout error for **UPSILON** connection is shown on **OMICRON**; **UPSILON** is able to again connect to the server on **OMICRON,** execute the command and exit gracefully
28	Closing server on **OMICRON** instantly without graceful shutdown (using CTRL+C command in terminal) while client on **UPSILON** is still connected to it; assume that **OMICRON** and **UPSILON** are connected at the start of this test similar to Acceptance Test 5	**UPSILON** gets a timeout error and exits

2.6.3.1 Acceptance Test Results

No.	Results (Screenshots, Packet Captures, Etc.)	Verdict
1	**UPSILON** (Error, usage and help message shown; program exits): 	**PASS**
2	**OMICRON** (Error message shown; program exits): 	**PASS**

3	**UPSILON** (Client tries to connect, but times out as server is not available at 192.168.1.72 on UDP port 1024):   	**PASS**
4	**OMICRON** (Server starts on UDP port 1024):	**PASS**

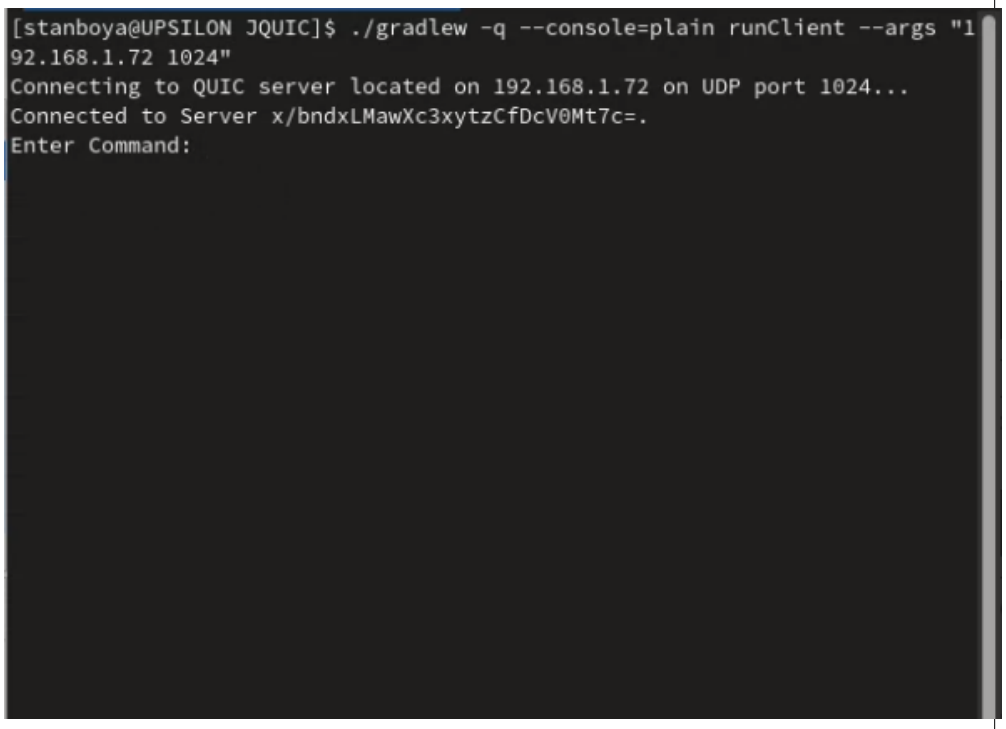
		
5	**OMICRON** (Server started on UDP port 1024 similar to Acceptance Test 4): 	**PASS**



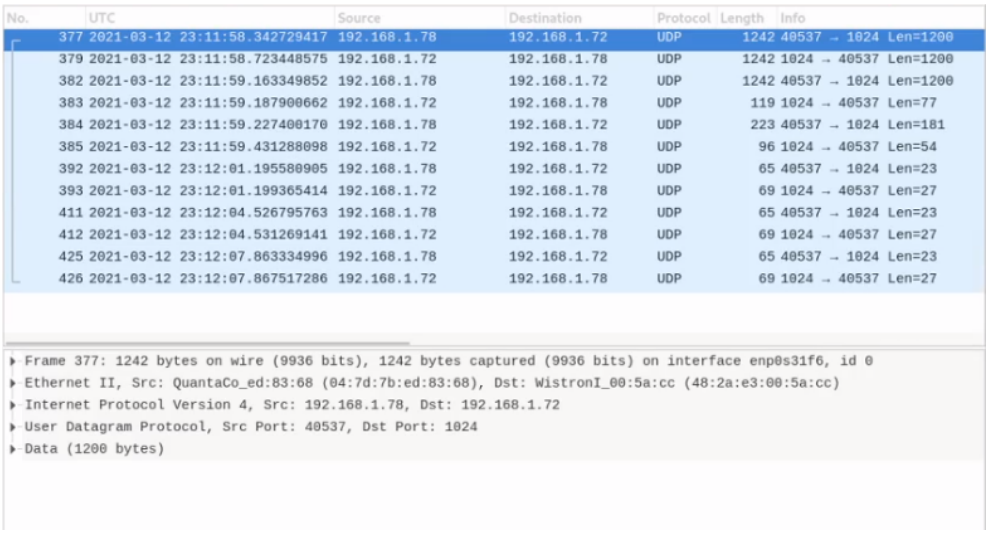
No.	Time	Source	Destination	Protocol	Length	Info
176	56.537835920	192.168.1.78	192.168.1.72	UDP	1242	40…
178	56.918993801	192.168.1.72	192.168.1.78	UDP	1242	10…
181	57.358447103	192.168.1.78	192.168.1.72	UDP	1242	40…
182	57.383346675	192.168.1.72	192.168.1.78	UDP	119	10…
183	57.422571996	192.168.1.78	192.168.1.72	UDP	223	40…
184	57.626844602	192.168.1.72	192.168.1.78	UDP	96	10…
191	59.390745117	192.168.1.78	192.168.1.72	UDP	65	40…
192	59.394926273	192.168.1.72	192.168.1.78	UDP	69	10…
211	62.721941048	192.168.1.78	192.168.1.72	UDP	65	40…
212	62.726835081	192.168.1.72	192.168.1.78	UDP	69	10…

```
▸ Frame 176: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on int
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 40537, Dst Port: 1024
▸ Data (1200 bytes)
```

UPSILON (Client connected to server and asks for command input):



```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server x/bndxLMawXc3xytzCfDcV0Mt7c=.
Enter Command:
```

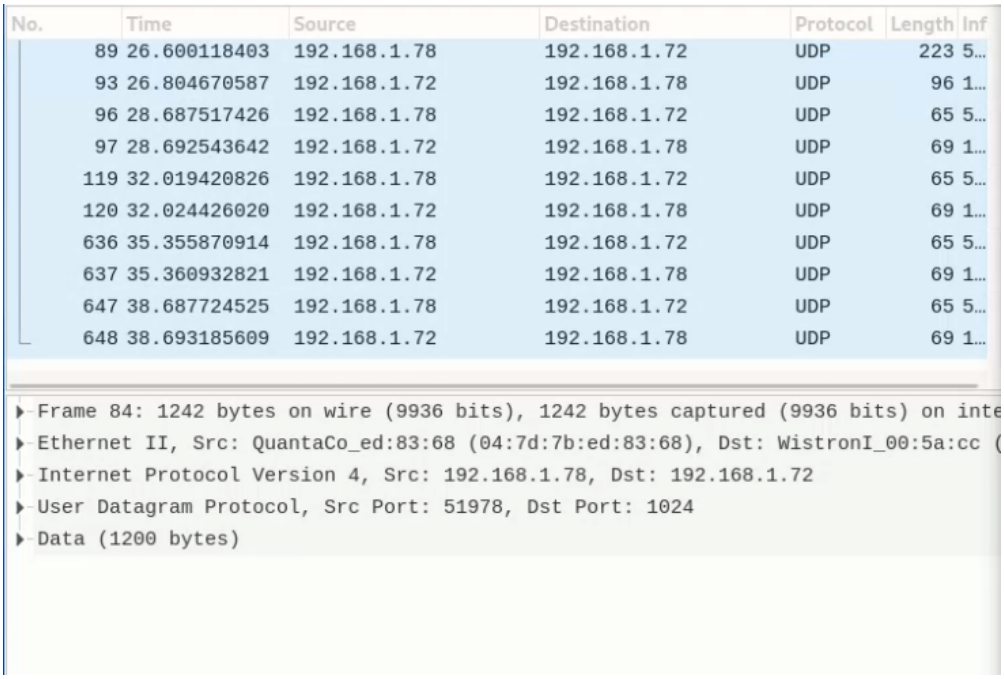


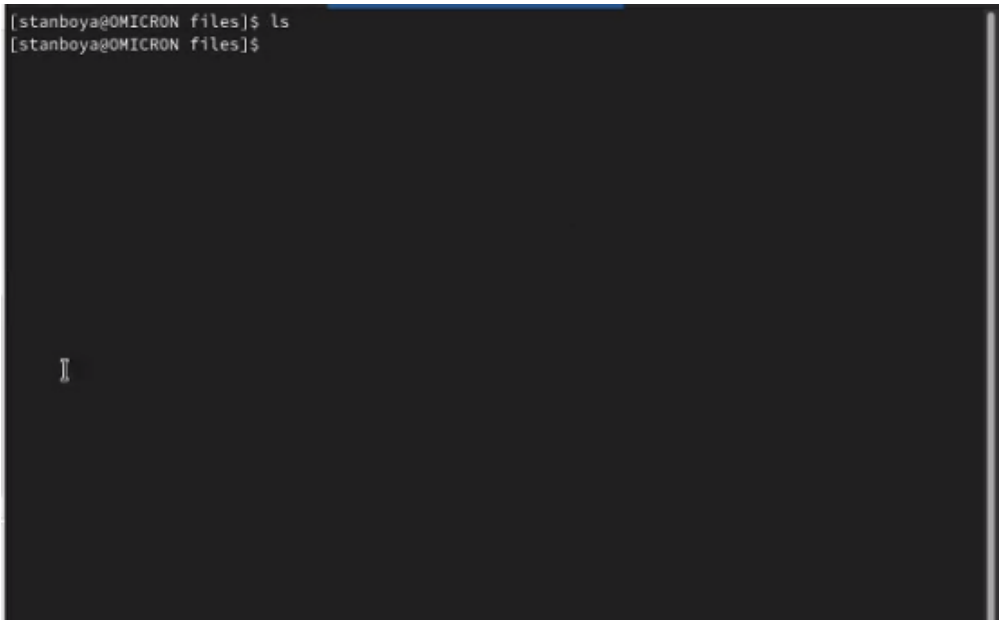
No.	UTC	Source	Destination	Protocol	Length	Info
377	2021-03-12 23:11:58.342729417	192.168.1.78	192.168.1.72	UDP	1242	40537 → 1024 Len=1200
379	2021-03-12 23:11:58.723448575	192.168.1.72	192.168.1.78	UDP	1242	1024 → 40537 Len=1200
382	2021-03-12 23:11:59.163349852	192.168.1.78	192.168.1.72	UDP	1242	40537 → 1024 Len=1200
383	2021-03-12 23:11:59.187900662	192.168.1.72	192.168.1.78	UDP	119	1024 → 40537 Len=77
384	2021-03-12 23:11:59.227400170	192.168.1.78	192.168.1.72	UDP	223	40537 → 1024 Len=181
385	2021-03-12 23:11:59.431288098	192.168.1.72	192.168.1.78	UDP	96	1024 → 40537 Len=54
392	2021-03-12 23:12:01.195580905	192.168.1.78	192.168.1.72	UDP	65	40537 → 1024 Len=23
393	2021-03-12 23:12:01.199365414	192.168.1.72	192.168.1.78	UDP	69	1024 → 40537 Len=27
411	2021-03-12 23:12:04.526795763	192.168.1.78	192.168.1.72	UDP	65	40537 → 1024 Len=23
412	2021-03-12 23:12:04.531269141	192.168.1.72	192.168.1.78	UDP	69	1024 → 40537 Len=27
425	2021-03-12 23:12:07.863334996	192.168.1.78	192.168.1.72	UDP	65	40537 → 1024 Len=23
426	2021-03-12 23:12:07.867517286	192.168.1.72	192.168.1.78	UDP	69	1024 → 40537 Len=27

6 **UPSILON** (Client tries to put the file on the server, but cannot since the file does not exist):

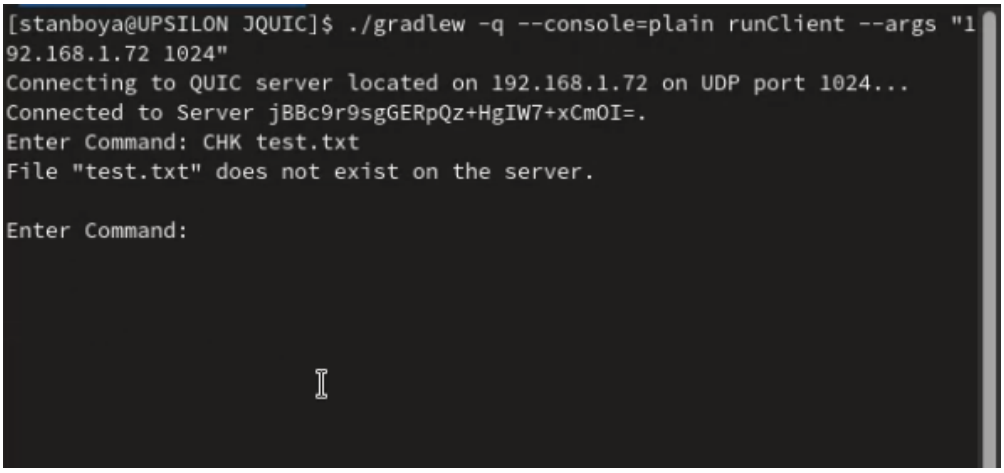
PASS

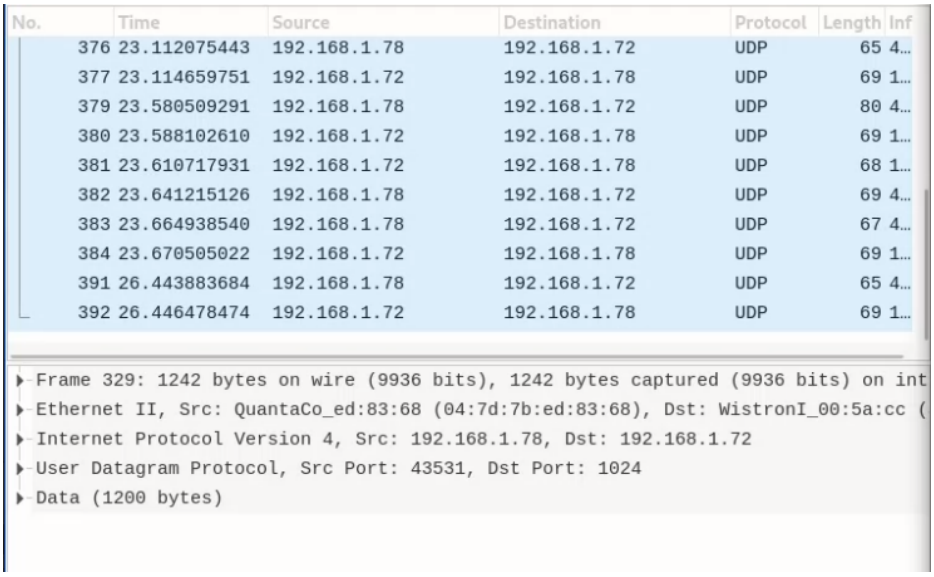
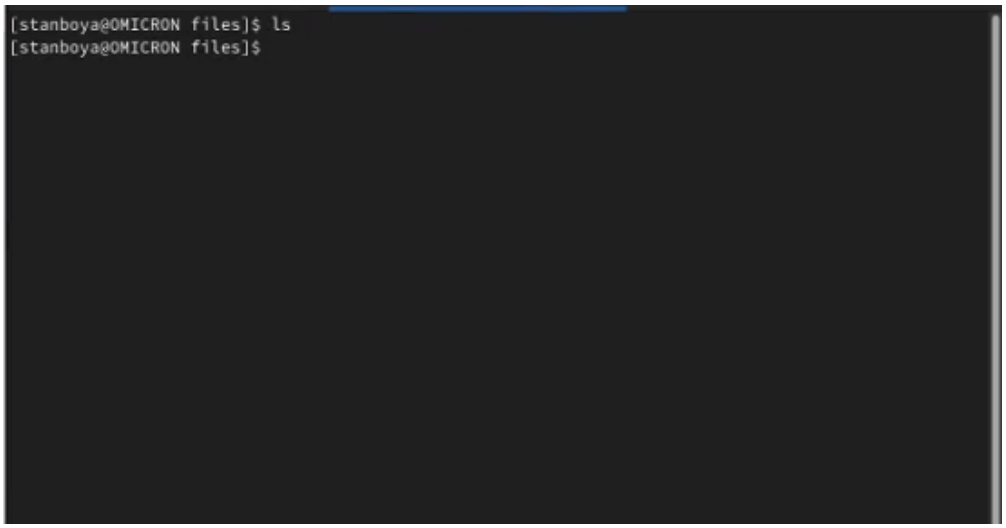
```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server TX1OBrfvNeQlpfbMvzqunuSdV7k=.
Enter Command: PUT test.txt
The following file does not exist: "test.txt". Please try a different file
.

Enter Command:
```

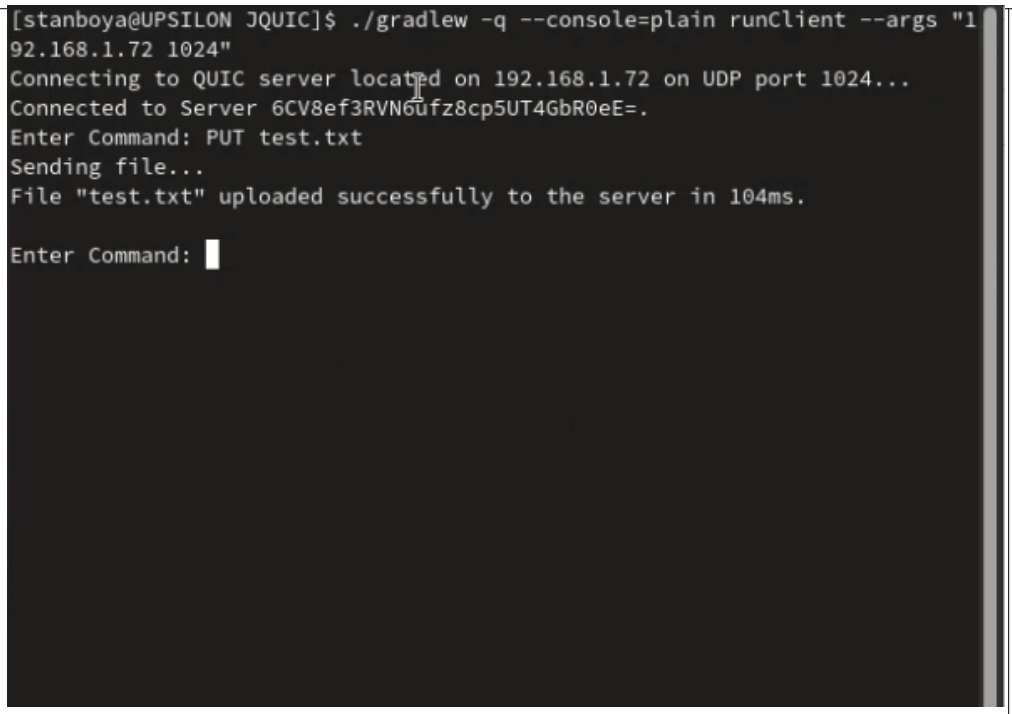
	 The output for **OMICRON** is similar to results for Acceptance Test 5.	
7	**OMICRON** (empty storage folder):	**PASS**

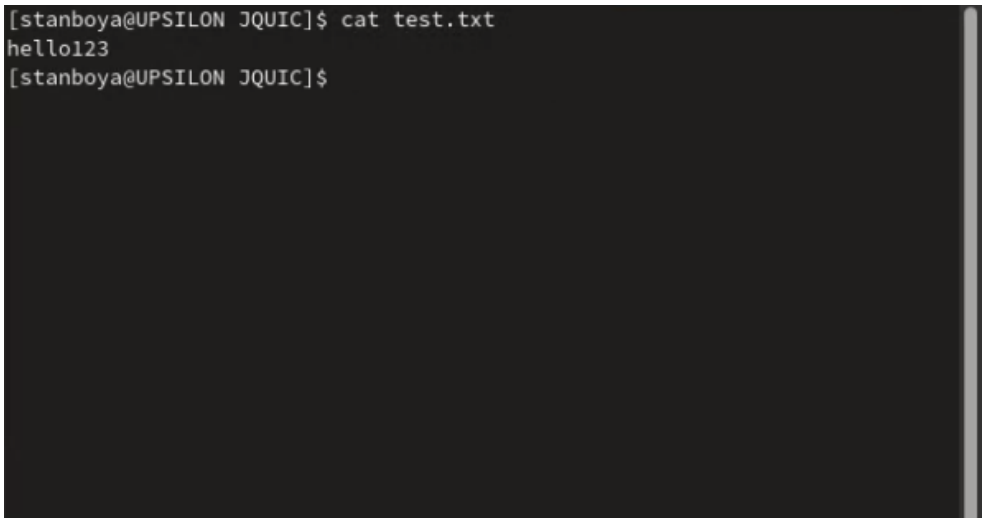


UPSILON (client checks for file “test.txt” on the server and does not find it):



	 The other output for **OMICRON** is similar to results for Acceptance Test 5.	
8	**OMICRON** (empty storage folder before **UPSILON** uploads the file):  **UPSILON** (client successfully uploads file “test.txt” to the server):	**PASS**


[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server 6CV8ef3RVN6ufz8cp5UT4GbR0eE=.
Enter Command: PUT test.txt
Sending file...
File "test.txt" uploaded successfully to the server in 104ms.
Enter Command:

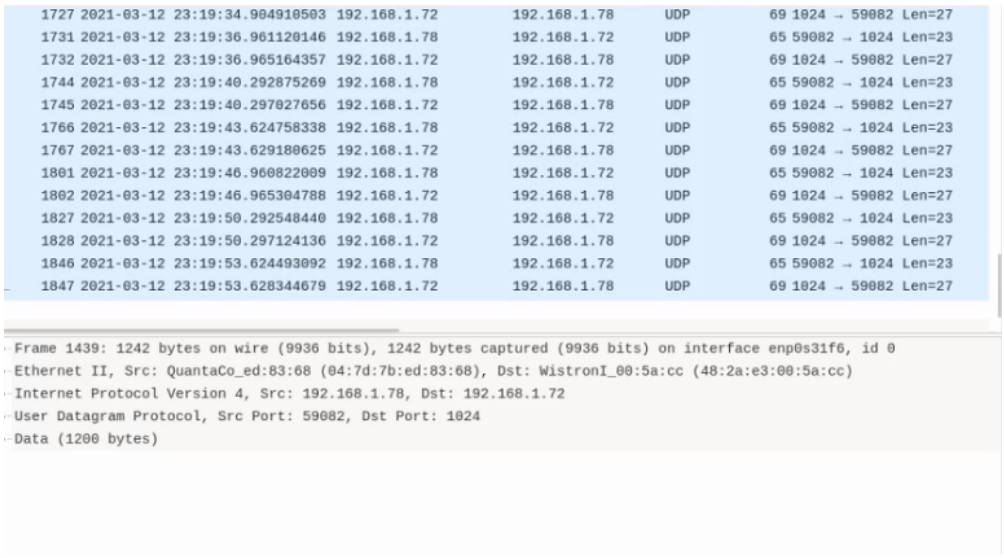
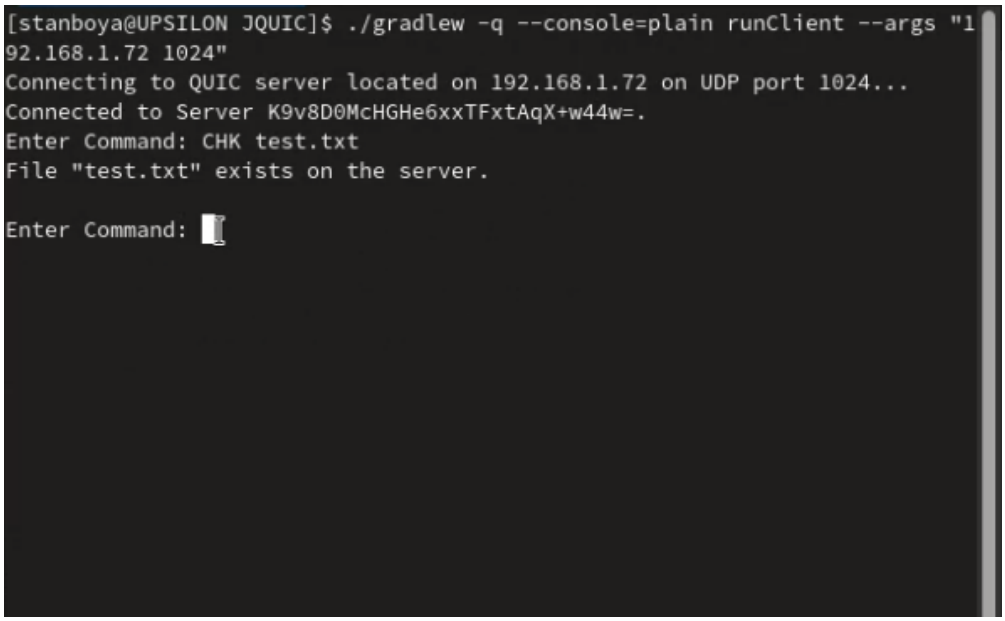

[stanboya@UPSILON JQUIC]$ cat test.txt
hello123
[stanboya@UPSILON JQUIC]$

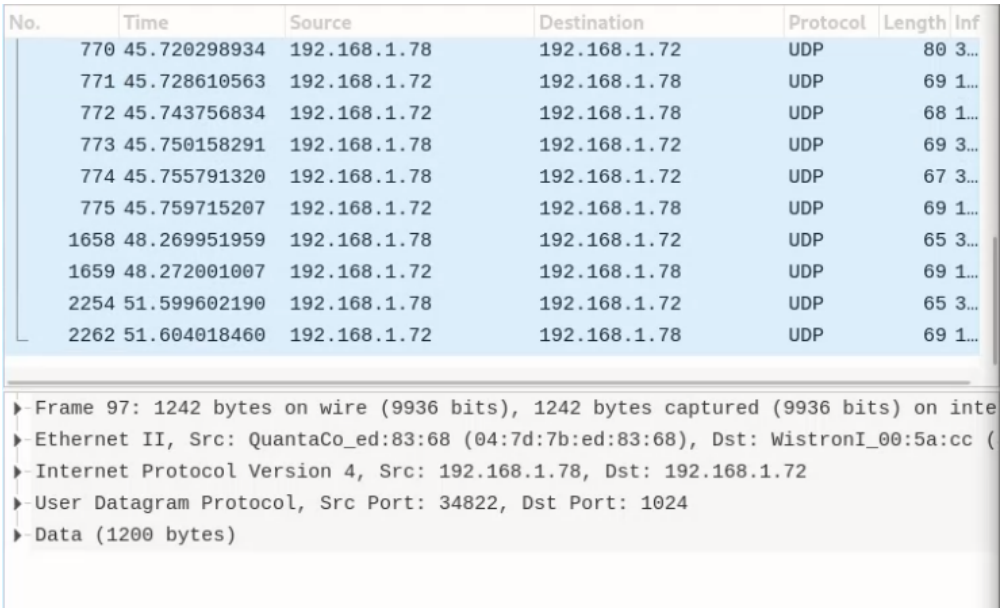
No.	Time	Source	Destination	Protocol	Length	Inf
560	141.506613124	192.168.1.72	192.168.1.78	UDP	67	1…
561	141.510593254	192.168.1.78	192.168.1.72	UDP	69	5…
562	141.551383928	192.168.1.78	192.168.1.72	UDP	67	5…
563	141.555020500	192.168.1.72	192.168.1.78	UDP	69	1…
567	143.611006848	192.168.1.78	192.168.1.72	UDP	65	5…
568	143.615283268	192.168.1.72	192.168.1.78	UDP	69	1…
578	146.942748377	192.168.1.78	192.168.1.72	UDP	65	5…
579	146.947289147	192.168.1.72	192.168.1.78	UDP	69	1…
589	150.274664592	192.168.1.78	192.168.1.72	UDP	65	5…
590	150.279423577	192.168.1.72	192.168.1.78	UDP	69	1…

```
▸ Frame 345: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on int
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 59082, Dst Port: 1024
▸ Data (1200 bytes)
```

OMICRON (storage folder with file “test.txt” that UPSILON uploaded):

```
[stanboya@OMICRON files]$ ls
test.txt
[stanboya@OMICRON files]$ cat test.txt
hello123
[stanboya@OMICRON files]$
```

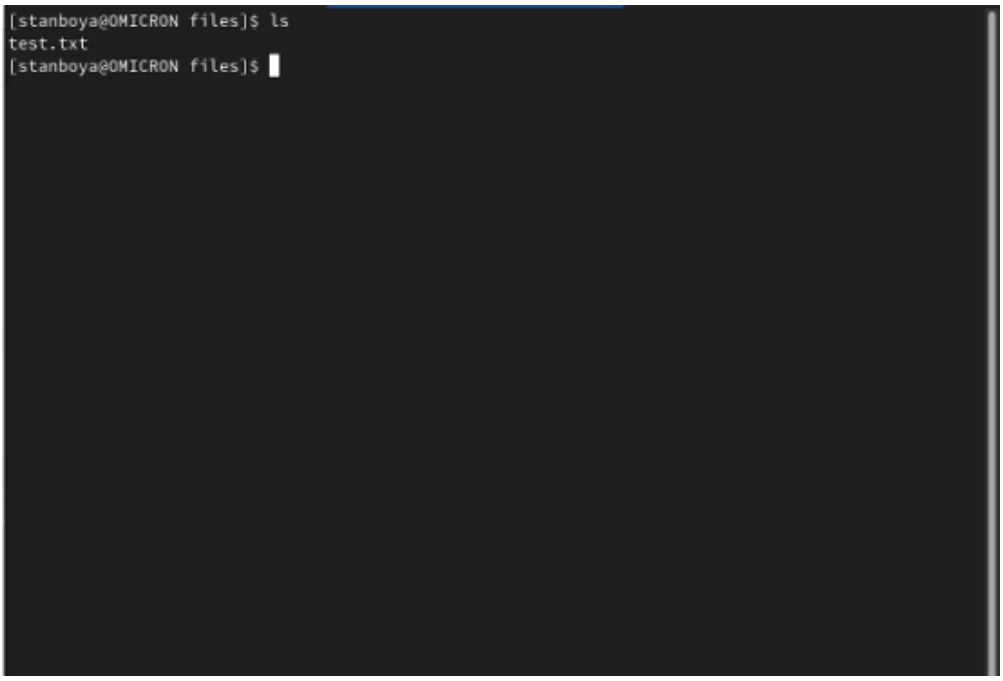
	 The other output for **OMICRON** is similar to results for Acceptance Test 5.	
9	**UPSILON** (client checks for file “test.txt” on the server and finds it since it was uploaded in the previous test): 	**PASS**



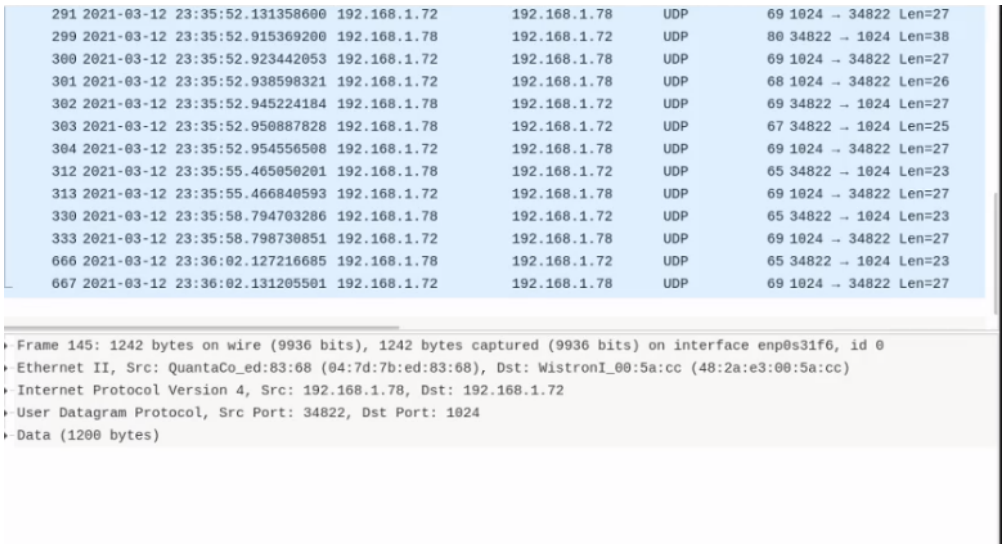
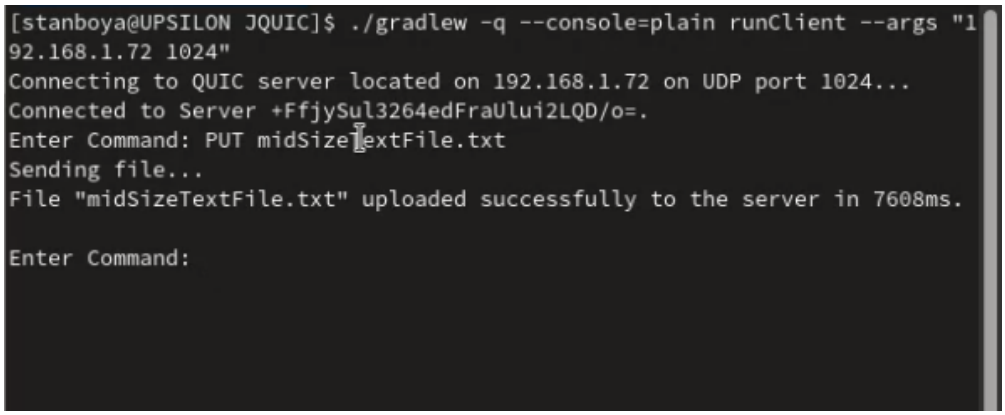
No.	Time	Source	Destination	Protocol	Length	Inf
770	45.720298934	192.168.1.78	192.168.1.72	UDP	80	3…
771	45.728610563	192.168.1.72	192.168.1.78	UDP	69	1…
772	45.743756834	192.168.1.72	192.168.1.78	UDP	68	1…
773	45.750158291	192.168.1.78	192.168.1.72	UDP	69	3…
774	45.755791320	192.168.1.78	192.168.1.72	UDP	67	3…
775	45.759715207	192.168.1.72	192.168.1.78	UDP	69	1…
1658	48.269951959	192.168.1.78	192.168.1.72	UDP	65	3…
1659	48.272001007	192.168.1.72	192.168.1.78	UDP	69	1…
2254	51.599602190	192.168.1.78	192.168.1.72	UDP	65	3…
2262	51.604018460	192.168.1.72	192.168.1.78	UDP	69	1…

```
▸ Frame 97: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on inte
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 34822, Dst Port: 1024
▸ Data (1200 bytes)
```

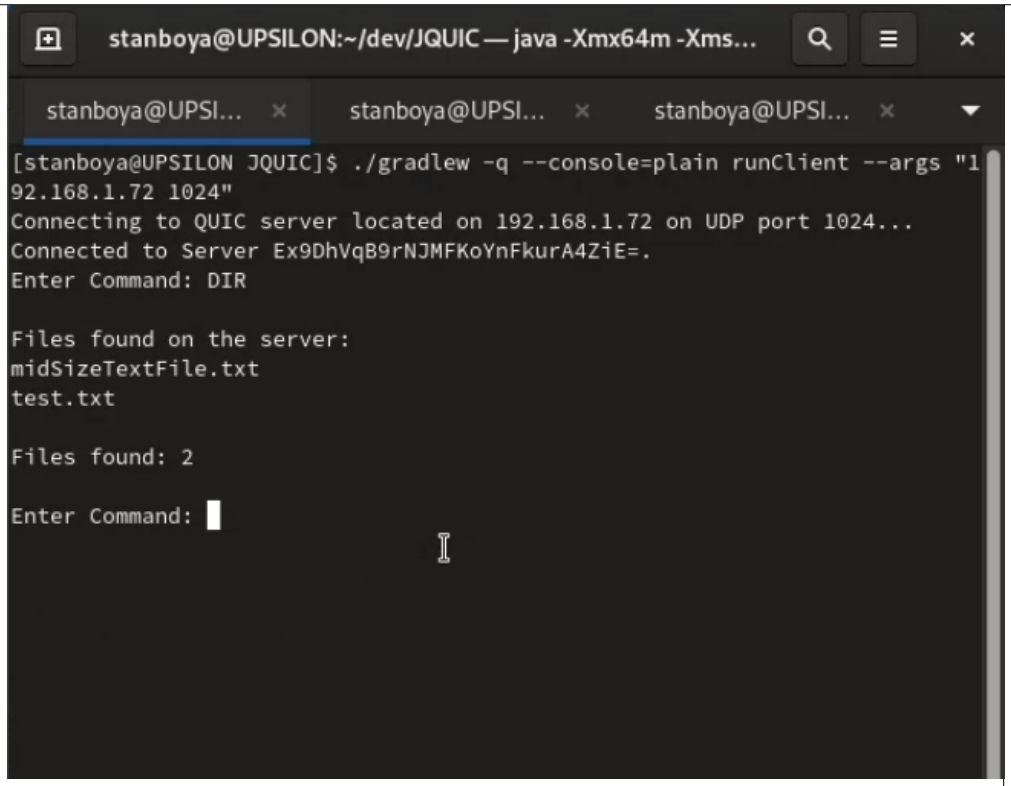
OMICRON (storage folder with file “test.txt” that UPSILON uploaded):

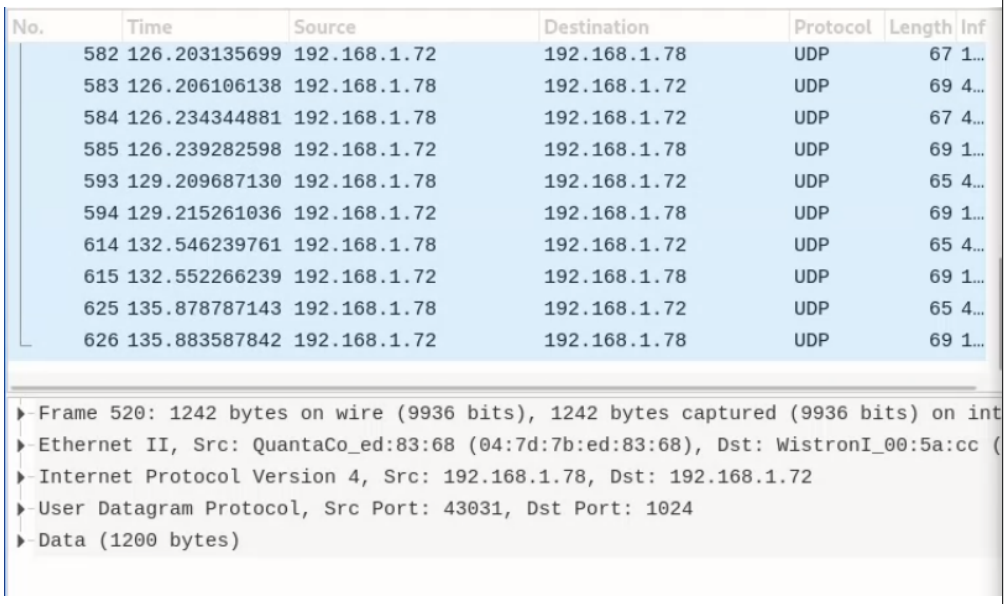


```
[stanboya@OMICRON files]$ ls
test.txt
[stanboya@OMICRON files]$
```

	 The other output for **OMICRON** is similar to results for Acceptance Test 5.	
10	**OMICRON** (storage folder before “midSizeTextFile.txt” is uploaded):  **UPSILON** (client successfully uploads file “midSizeTextFile.txt” to the server): 	**PASS**

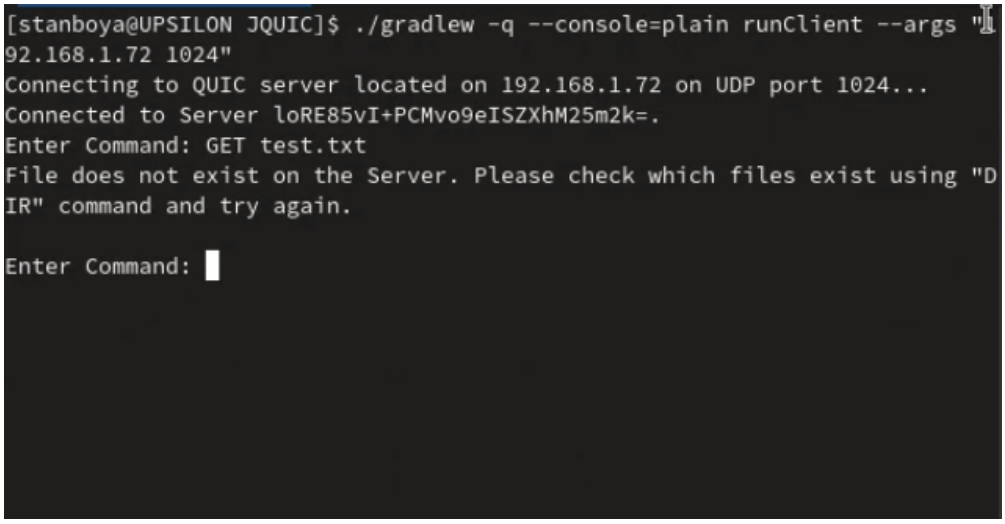
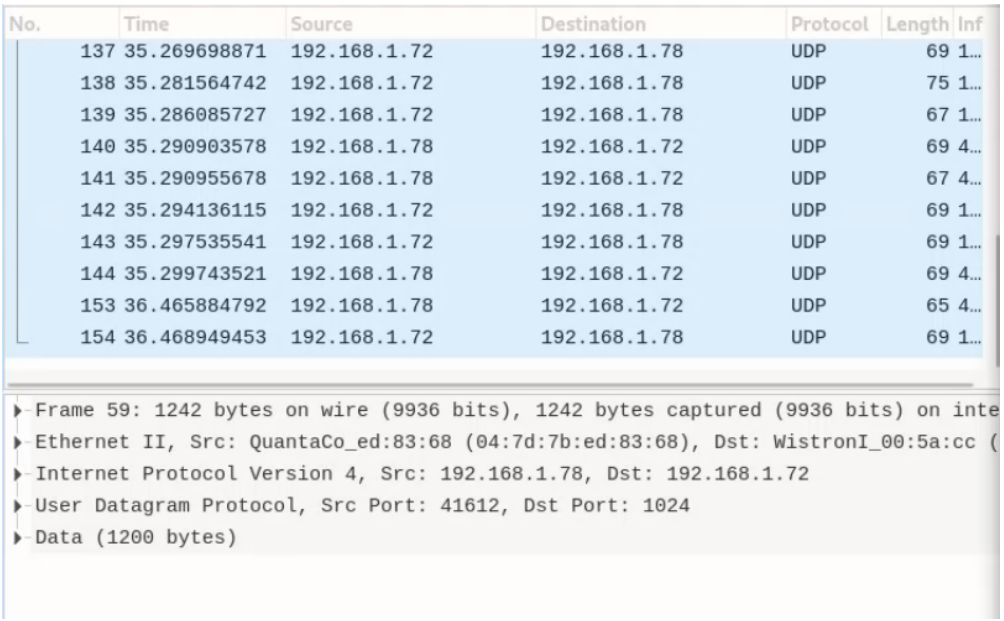
	3287 70.041381941 192.168.1.72 192.168.1.78 UDP 71 1… 3288 70.045471238 192.168.1.72 192.168.1.78 UDP 68 1… 3289 70.049596630 192.168.1.72 192.168.1.78 UDP 71 1… 3290 70.053768574 192.168.1.72 192.168.1.78 UDP 71 1… 3291 70.055148514 192.168.1.78 192.168.1.72 UDP 71 3… 3292 70.057981565 192.168.1.72 192.168.1.78 UDP 71 1… 3293 70.060205167 192.168.1.78 192.168.1.72 UDP 1073 3… 3294 70.065353659 192.168.1.72 192.168.1.78 UDP 71 1… 3295 70.076737187 192.168.1.78 192.168.1.72 UDP 71 3… 3296 70.077899872 192.168.1.72 192.168.1.78 UDP 71 1… Frame 416: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on int Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72 User Datagram Protocol, Src Port: 38708, Dst Port: 1024 Data (1200 bytes) **OMICRON** (storage folder after “midSizeText.txt” is uploaded): [stanboya@OMICRON files]$ ls midSizeTextFile.txt test.txt No. UTC Source Destination Protocol Length Info 2946 2021-03-12 23:42:29.542692661 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2947 2021-03-12 23:42:29.545876266 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2948 2021-03-12 23:42:29.547155262 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2949 2021-03-12 23:42:29.549912366 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2950 2021-03-12 23:42:29.550315899 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2951 2021-03-12 23:42:29.553042867 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2952 2021-03-12 23:42:29.554448010 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2953 2021-03-12 23:42:29.557153263 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2954 2021-03-12 23:42:29.558563669 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2955 2021-03-12 23:42:29.561271407 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2956 2021-03-12 23:42:29.562684530 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 2957 2021-03-12 23:42:29.565383073 192.168.1.78 192.168.1.72 UDP 1073 38708 → 1024 Len=1031 2958 2021-03-12 23:42:29.566822240 192.168.1.72 192.168.1.78 UDP 71 1024 → 38708 Len=29 The other output for **OMICRON** is similar to results for Acceptance Test 5.	
11	**OMICRON** (storage folder during this test): [stanboya@OMICRON files]$ ls midSizeTextFile.txt test.txt **UPSILON** (successfully retrieves the directory contents of DFS storage folder on **OMICRON**):	**PASS**





The other output for OMICRON is similar to results for Acceptance Test 5.

12	**OMICRON** (storage folder during this test): `[stanboya@OMICRON files]$ ls` `midSizeTextFile.txt  test.txt` **UPSILON** (successfully deletes file in storage on **OMICRON**): `[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1` `92.168.1.72 1024"` `Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...` `Connected to Server TbTeTaJNFyWrKOW3jToIXWpAs5o=.` `Enter Command: DEL test.txt` `File "test.txt" was successfully deleted from the server.` `Enter Command:` No. Time Source Destination Protocol Length Inf 173 45.151301475 192.168.1.78 192.168.1.72 UDP 80 5… 174 45.156379502 192.168.1.72 192.168.1.78 UDP 69 1… 175 45.168015946 192.168.1.72 192.168.1.78 UDP 68 1… 176 45.171846880 192.168.1.78 192.168.1.72 UDP 69 5… 177 45.172490510 192.168.1.72 192.168.1.78 UDP 67 1… 178 45.188313131 192.168.1.78 192.168.1.72 UDP 69 5… 179 45.191930384 192.168.1.78 192.168.1.72 UDP 67 5… 180 45.195849709 192.168.1.72 192.168.1.78 UDP 69 1… 185 47.139263896 192.168.1.78 192.168.1.72 UDP 65 5… 186 47.144062021 192.168.1.72 192.168.1.78 UDP 69 1… ▸ Frame 67: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on inte ▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72 ▸ User Datagram Protocol, Src Port: 51828, Dst Port: 1024 ▸ Data (1200 bytes) **OMICRON** (storage folder during this test): `[stanboya@OMICRON files]$ ls` `midSizeTextFile.txt` The other output for **OMICRON** is similar to results for Acceptance Test 5.	**PASS**
13	**OMICRON** (storage folder during this test): `[stanboya@OMICRON files]$ ls` `midSizeTextFile.txt`	**PASS**

	UPSILON (Tries to get the file from the server, but gets an error since it does not exist and asks for another command):   The other output for **OMICRON** is similar to results for Acceptance Test 5.	
14	**OMICRON** (storage folder during this test):  **UPSILON** (before getting file “midSizeTextFile.txt” from the server):	**PASS**

```
[stanboya@UPSILON JQUIC]$ ls
build             files    gradlew      settings.gradle.kts  test.txt
build.gradle.kts  gradle   gradlew.bat  src
```

UPSILON (successfully gets the file “midSizeTextFile.txt” from the server):

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server JCSw6NOwX2x/kMuqNIHKvVmtAOo=.
Enter Command: GET midSizeTextFile.txt
Receiving file...
File "midSizeTextFile.txt" downloaded successfully from the server in 6666
ms.

Enter Command:
```

No.	Time	Source	Destination	Protocol	Length	Inf
3029	62.946975004	192.168.1.72	192.168.1.78	UDP	1073	1…
3030	62.947174618	192.168.1.78	192.168.1.72	UDP	71	3…
3031	62.951115345	192.168.1.72	192.168.1.78	UDP	1073	1…
3032	62.951429496	192.168.1.78	192.168.1.72	UDP	71	3…
3033	62.954560808	192.168.1.78	192.168.1.72	UDP	71	3…
3034	62.955655855	192.168.1.72	192.168.1.78	UDP	1073	1…
3035	62.958716507	192.168.1.78	192.168.1.72	UDP	71	3…
3036	62.959751415	192.168.1.72	192.168.1.78	UDP	1073	1…
3037	62.962884929	192.168.1.72	192.168.1.78	UDP	1073	1…
3038	62.962905561	192.168.1.78	192.168.1.72	UDP	71	3…

```
▸ Frame 99: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on inte
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 34176, Dst Port: 1024
▸ Data (1200 bytes)
```

UPSILON (after getting file “midSizeTextFile.txt” from the server):

```
[stanboya@UPSILON JQUIC]$ ls
build             files    gradlew      midSizeTextFile.txt  src
build.gradle.kts  gradle   gradlew.bat  settings.gradle.kts  test.txt
```

The other output for **OMICRON** is similar to results for Acceptance Test 5.

15	**UPSILON** (printing out the help message due to “HELP” command):	**PASS**

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "
192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server TNgtYo4VwTGeQvq7ySybgG0emP0=.
Enter Command: HELP

Possible DFS Commands
---------------------
PUT [Local File Path]   - uploads the file to the server or edits the one
 already there if it's not being read or edited
GET [Server File]       - downloads the file from the server or edits the
 one already there if it's not being read or edited
CHK [Server File]       - checks if the file exists on the server
DEL [Server File]       - deletes the file from the server
LOCALPATH               - shows what is the current directory and path to
 it on the client
DIR                     - shows what files are currently available on the
 server
LOCALDIR                - shows what files are currently available in the
 current directory on the client
HELP                    - shows all the available commands


Enter Command:
```

The output for **OMICRON** is similar to results for Acceptance Test 5.

16 | **OMICRON** (both clients from **TAU** and **UPSILON** successfully connect to the server: | **PASS**

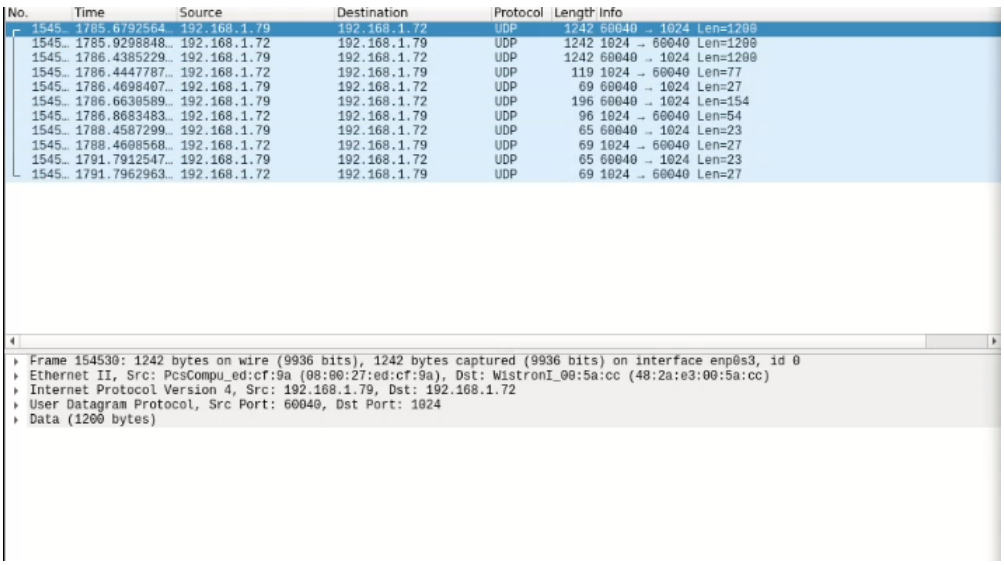
```
[stanboya@OMICRON JQUIC]$ ./gradlew -q --console=plain runServer --args "1024 files"
Started server on UDP port 1024.
New Client Connected: ID 7ZXTv0Bip6zvuuxRqXetg1zM9Tc=
New Client Connected: ID JhZuguigtMyG/dALpjHT/BokQJ4=
```

No.	UTC	Source	Destination	Protocol	Length	Info
3174	2021-03-13 02:24:04.044796983	192.168.1.72	192.168.1.79	UDP	96	1024 → 60040 Len=54
3177	2021-03-13 02:24:05.635589279	192.168.1.79	192.168.1.72	UDP	65	60040 → 1024 Len=23
3178	2021-03-13 02:24:05.637334134	192.168.1.72	192.168.1.79	UDP	69	1024 → 60040 Len=27
3194	2021-03-13 02:24:06.068286204	192.168.1.78	192.168.1.72	UDP	65	33016 → 1024 Len=23
3195	2021-03-13 02:24:06.073144059	192.168.1.72	192.168.1.78	UDP	69	1024 → 33016 Len=27
3221	2021-03-13 02:24:08.968032556	192.168.1.79	192.168.1.72	UDP	65	60040 → 1024 Len=23
3222	2021-03-13 02:24:08.972754541	192.168.1.72	192.168.1.79	UDP	69	1024 → 60040 Len=27
3224	2021-03-13 02:24:09.404422704	192.168.1.78	192.168.1.72	UDP	65	33016 → 1024 Len=23
3225	2021-03-13 02:24:09.408822584	192.168.1.72	192.168.1.78	UDP	69	1024 → 33016 Len=27
3238	2021-03-13 02:24:12.302865093	192.168.1.79	192.168.1.72	UDP	65	60040 → 1024 Len=23
3239	2021-03-13 02:24:12.304664569	192.168.1.72	192.168.1.79	UDP	69	1024 → 60040 Len=27
3244	2021-03-13 02:24:12.737069501	192.168.1.78	192.168.1.72	UDP	65	33016 → 1024 Len=23
3245	2021-03-13 02:24:12.741739355	192.168.1.72	192.168.1.78	UDP	69	1024 → 33016 Len=27

```
▸ Frame 2837: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on interface enp0s31f6, id 0
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (48:2a:e3:00:5a:cc)
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 33016, Dst Port: 1024
▸ Data (1200 bytes)
```

TAU (client successfully connects to server on **OMICRON**):

```
[tau@localhost JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server JhZuguigtMyG/dALpjHT/BokQJ4=.
Enter Command:
```



No.	Time	Source	Destination	Protocol	Length	Info
1545…	1785.6792564…	192.168.1.79	192.168.1.72	UDP	1242	60040 → 1024 Len=1200
1545…	1785.9298848…	192.168.1.72	192.168.1.79	UDP	1242	1024 → 60040 Len=1200
1545…	1786.4385229…	192.168.1.79	192.168.1.72	UDP	1242	60040 → 1024 Len=1200
1545…	1786.4447787…	192.168.1.72	192.168.1.79	UDP	119	1024 → 60040 Len=77
1545…	1786.4698407…	192.168.1.79	192.168.1.72	UDP	69	60040 → 1024 Len=27
1545…	1786.6630589…	192.168.1.79	192.168.1.72	UDP	196	60040 → 1024 Len=154
1545…	1786.8683483…	192.168.1.72	192.168.1.79	UDP	96	1024 → 60040 Len=54
1545…	1788.4587299…	192.168.1.79	192.168.1.72	UDP	65	60040 → 1024 Len=23
1545…	1788.4608568…	192.168.1.72	192.168.1.79	UDP	69	1024 → 60040 Len=27
1545…	1791.7912547…	192.168.1.79	192.168.1.72	UDP	65	60040 → 1024 Len=23
1545…	1791.7962963…	192.168.1.72	192.168.1.79	UDP	69	1024 → 60040 Len=27

```
▸ Frame 154530: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_ed:cf:9a (08:00:27:ed:cf:9a), Dst: WistronI_00:5a:cc (48:2a:e3:00:5a:cc)
▸ Internet Protocol Version 4, Src: 192.168.1.79, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 60040, Dst Port: 1024
▸ Data (1200 bytes)
```

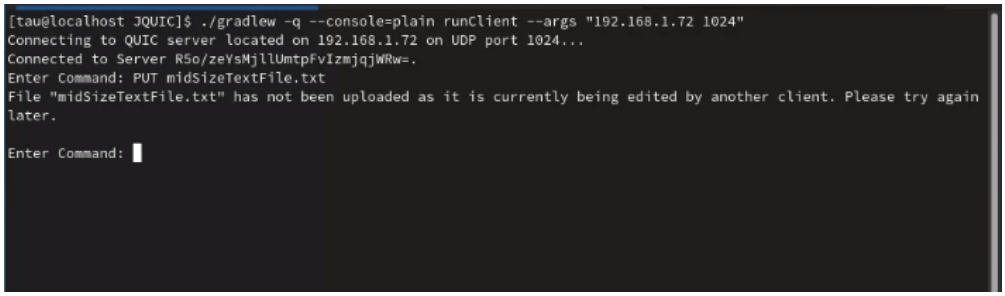
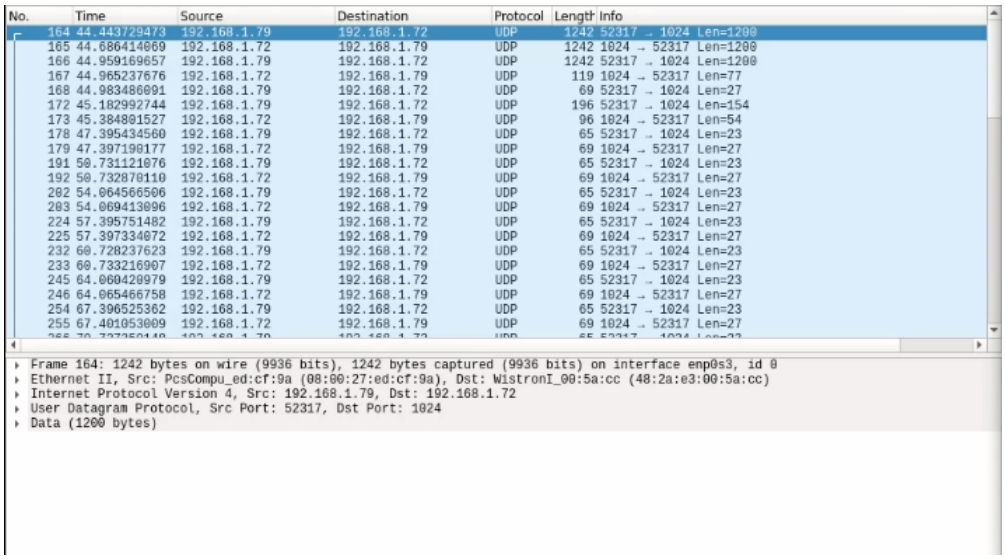
The output for **UPSILON** is similar to results for Acceptance Test 5.

17 **UPSILON** (The file “midSizeTextFile.txt” is still successfully uploaded): **PASS**

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "19
2.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server jfEudLIXzsDCTFmFn8sbu/ObR44=.
Enter Command: PUT midSizeTextFile.txt
Sending file...
File "midSizeTextFile.txt" uploaded successfully to the server in 8760ms.

Enter Command:
```

No.	Time	Source	Destination	Protocol	Length	Inf
3164	88.321113258	192.168.1.78	192.168.1.72	UDP	1073	4…
3165	88.321829080	192.168.1.72	192.168.1.78	UDP	71	1…
3167	88.325266602	192.168.1.78	192.168.1.72	UDP	1073	4…
3168	88.326028890	192.168.1.72	192.168.1.78	UDP	71	1…
3169	88.328428989	192.168.1.78	192.168.1.72	UDP	1073	4…
3170	88.330444060	192.168.1.72	192.168.1.78	UDP	71	1…
3171	88.357240071	192.168.1.78	192.168.1.72	UDP	71	4…
3172	88.358170852	192.168.1.72	192.168.1.78	UDP	71	1…
3189	91.024348367	192.168.1.78	192.168.1.72	UDP	66	4…
3190	91.025906843	192.168.1.72	192.168.1.78	UDP	71	1…

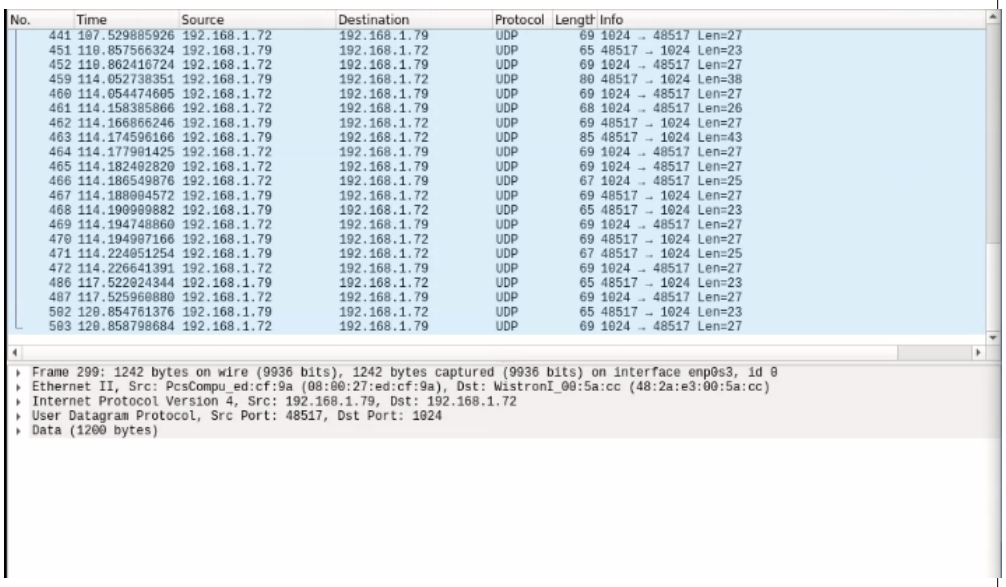
TAU (Client fails to upload the file “midSizeTextFile.txt” as **UPSILON** is uploading it

	at the same time):   The output for **OMICRON** is similar to results for Acceptance Test 16.	
18	**OMICRON** (storage folder before **TAU** and **UPSILON** upload to it) (not shown on video):	**PASS**

```
[stanboya@OMICRON files]$ ls
[stanboya@OMICRON files]$
```

TAU (File “test.txt” successfully uploaded to the server while client on **UPSILON** is also uploading):

```
[tau@localhost JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server L41ioih9HpTqQA1P8BTPwh5hm7A=.
Enter Command: PUT test.txt
Sending file...
File "test.txt" uploaded successfully to the server in 144ms.

Enter Command:
```

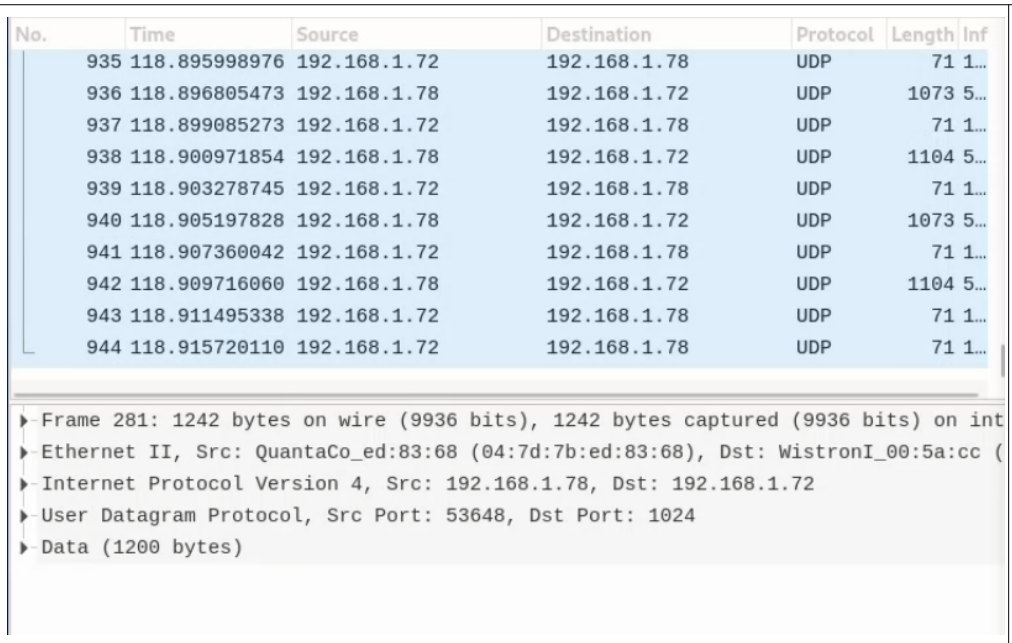


No.	Time	Source	Destination	Protocol	Length	Info
441	107.529885926	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
451	110.857566324	192.168.1.79	192.168.1.72	UDP	65	48517 → 1024 Len=23
452	110.862416724	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
459	114.052738351	192.168.1.79	192.168.1.72	UDP	80	48517 → 1024 Len=38
460	114.054474605	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
461	114.158385866	192.168.1.72	192.168.1.79	UDP	68	1024 → 48517 Len=26
462	114.166866246	192.168.1.79	192.168.1.72	UDP	69	48517 → 1024 Len=27
463	114.174596166	192.168.1.79	192.168.1.72	UDP	85	48517 → 1024 Len=43
464	114.177901425	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
465	114.182402820	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
466	114.186549876	192.168.1.72	192.168.1.79	UDP	67	1024 → 48517 Len=25
467	114.188004572	192.168.1.79	192.168.1.72	UDP	69	48517 → 1024 Len=27
468	114.190909882	192.168.1.79	192.168.1.72	UDP	65	48517 → 1024 Len=23
469	114.194748860	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
470	114.194907166	192.168.1.79	192.168.1.72	UDP	69	48517 → 1024 Len=27
471	114.224051254	192.168.1.79	192.168.1.72	UDP	67	48517 → 1024 Len=25
472	114.226641391	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
486	117.522024344	192.168.1.79	192.168.1.72	UDP	65	48517 → 1024 Len=23
487	117.525960880	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27
502	120.854761376	192.168.1.79	192.168.1.72	UDP	65	48517 → 1024 Len=23
503	120.858798684	192.168.1.72	192.168.1.79	UDP	69	1024 → 48517 Len=27

```
▸ Frame 209: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_ed:cf:9a (08:00:27:ed:cf:9a), Dst: WistronI_00:5a:cc (48:2a:e3:00:5a:cc)
▸ Internet Protocol Version 4, Src: 192.168.1.79, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 48517, Dst Port: 1024
▸ Data (1200 bytes)
```

UPSILON (File “midSizeTextFile.txt” successfully uploaded to server):

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "
192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server FZbgA8oSCpkRBfRImGGe70nHZpY=.
Enter Command: PUT midSizeTextFile.txt
Sending file...
File "midSizeTextFile.txt" uploaded successfully to the server in 7657ms.

Enter Command:
```



No.	Time	Source	Destination	Protocol	Length	Inf
935	118.895998976	192.168.1.72	192.168.1.78	UDP	71	1…
936	118.896805473	192.168.1.78	192.168.1.72	UDP	1073	5…
937	118.899085273	192.168.1.72	192.168.1.78	UDP	71	1…
938	118.900971854	192.168.1.78	192.168.1.72	UDP	1104	5…
939	118.903278745	192.168.1.72	192.168.1.78	UDP	71	1…
940	118.905197828	192.168.1.78	192.168.1.72	UDP	1073	5…
941	118.907360042	192.168.1.72	192.168.1.78	UDP	71	1…
942	118.909716060	192.168.1.78	192.168.1.72	UDP	1104	5…
943	118.911495338	192.168.1.72	192.168.1.78	UDP	71	1…
944	118.915720110	192.168.1.72	192.168.1.78	UDP	71	1…

OMICRON (storage folder after **TAU** and **UPSILON** upload to it) (not shown on video):

```
[stanboya@OMICRON files]$ ls
midSizeTextFile.txt  test.txt
```

The output for **OMICRON** is similar to results for Acceptance Test 16.

19 **UPSILON** (client successfully uploads file “midSizeTextFile.txt” to the server): **PASS**

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server QXsYyx8uAmQJfxVZNY8QvXmxP9w=.
Enter Command: PUT midSizeTextFile.txt
Sending file...
File "midSizeTextFile.txt" uploaded successfully to the server in 6528ms.

Enter Command:
```

No.	Time	Source	Destination	Protocol	Length	Inf
2742	213.639327419	192.168.1.78	192.168.1.72	UDP	1073	5…
2743	213.640289551	192.168.1.72	192.168.1.78	UDP	71	1…
2744	213.643449236	192.168.1.78	192.168.1.72	UDP	1073	5…
2745	213.644376648	192.168.1.72	192.168.1.78	UDP	71	1…
2746	213.647559227	192.168.1.78	192.168.1.72	UDP	1073	5…
2747	213.648460870	192.168.1.72	192.168.1.78	UDP	71	1…
2748	213.651659583	192.168.1.78	192.168.1.72	UDP	1073	5…
2749	213.652542975	192.168.1.72	192.168.1.78	UDP	71	1…
2750	213.655783548	192.168.1.78	192.168.1.72	UDP	1073	5…
2751	213.656710369	192.168.1.72	192.168.1.78	UDP	71	1…

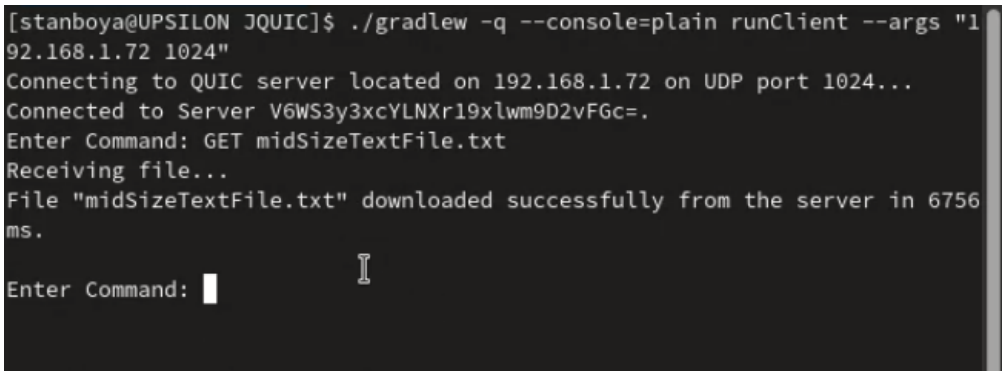
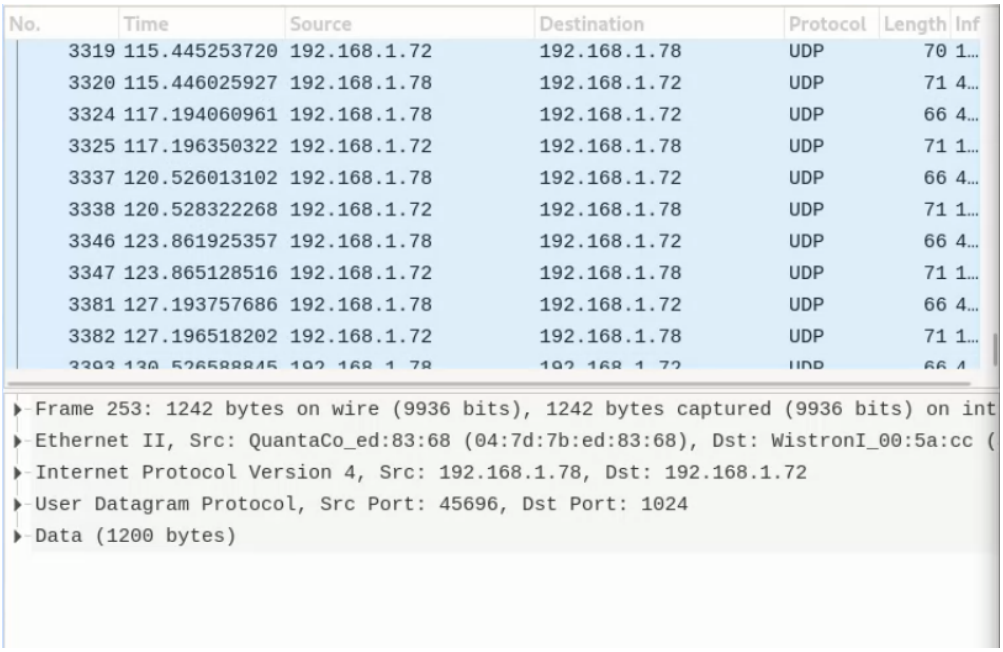
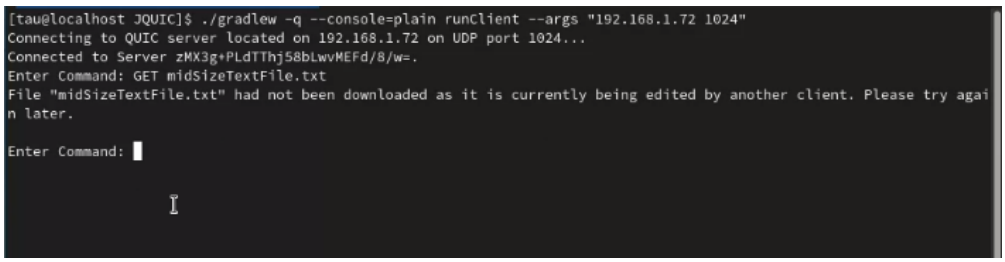
TAU (client fails to get file “midSizeTextFile.txt” from the server as it is being edited by **UPSILON**):

```
[tau@localhost JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server RaD/2n4SXz0kytUYX8mpOdVJ+ZY=.
Enter Command: GET midSizeTextFile.txt
File "midSizeTextFile.txt" had not been downloaded as it is currently being edited by another client. Please try agai
n later.

Enter Command:
```

No.	Time	Source	Destination	Protocol	Length	Info
1074	174.115947454	192.168.1.79	192.168.1.72	UDP	1242	42002 → 1024 Len=1200
1076	174.416583489	192.168.1.72	192.168.1.79	UDP	1242	1024 → 42002 Len=1200
1077	174.816003840	192.168.1.79	192.168.1.72	UDP	1242	42002 → 1024 Len=1200
1081	174.824282995	192.168.1.72	192.168.1.79	UDP	119	1024 → 42002 Len=77
1082	174.839657817	192.168.1.79	192.168.1.72	UDP	223	42002 → 1024 Len=181
1083	175.045147770	192.168.1.72	192.168.1.79	UDP	96	1024 → 42002 Len=54
1087	177.064034299	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1088	177.068916826	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1114	180.399973431	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1115	180.404160426	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1124	183.731714396	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1125	183.737014328	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1137	187.063645514	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1138	187.068587629	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1151	190.395808770	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1152	190.400437983	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1164	193.728364347	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1165	193.732557322	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1181	197.064146724	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23
1182	197.068462034	192.168.1.72	192.168.1.79	UDP	69	1024 → 42002 Len=27
1196	200.395706358	192.168.1.79	192.168.1.72	UDP	65	42002 → 1024 Len=23

```
▸ Frame 1074: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_ed:cf:9a (08:00:27:ed:cf:9a), Dst: WistronI_00:5a:cc (48:2a:e3:00:5a:cc)
▸ Internet Protocol Version 4, Src: 192.168.1.79, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 42002, Dst Port: 1024
▸ Data (1200 bytes)
```

The output for **OMICRON** is similar to results for Acceptance Test 16.

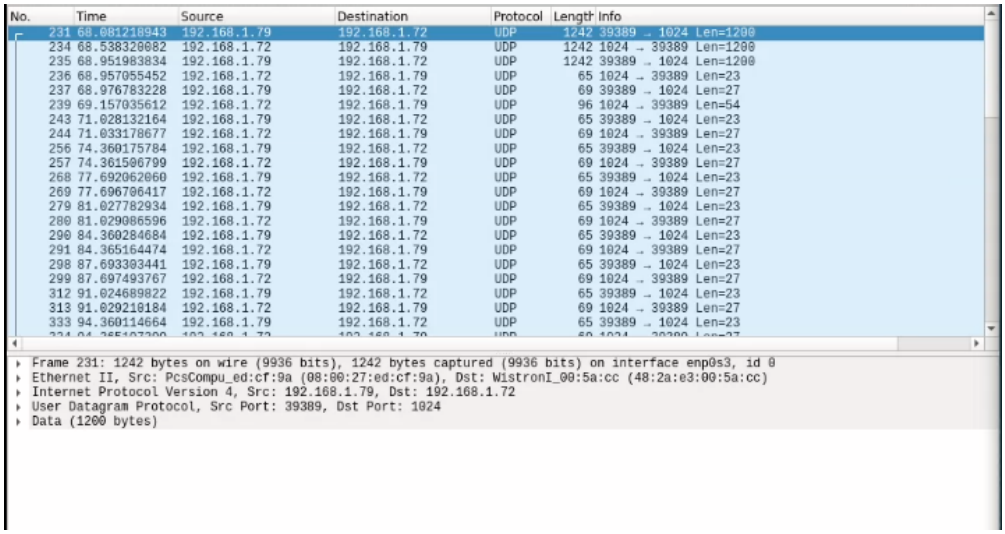
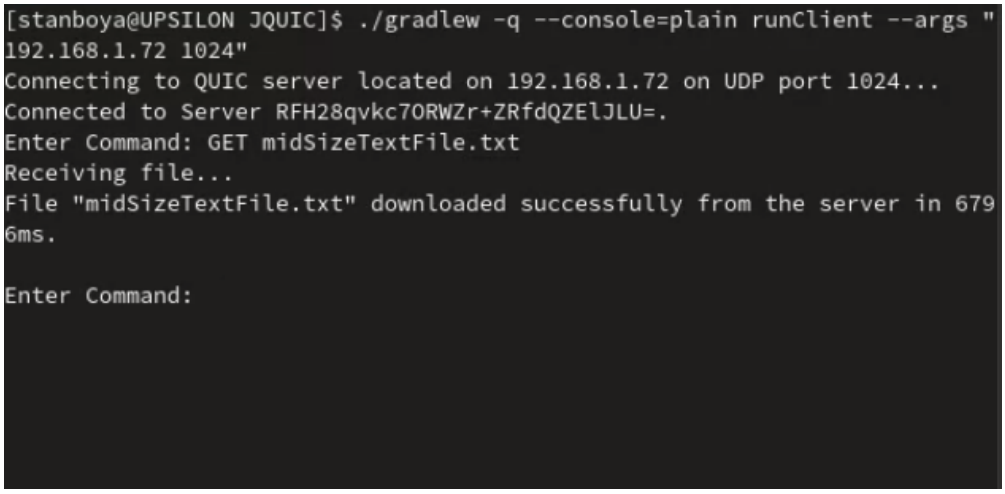
20	**UPSILON** (client successfully downloads file “midSizeTextFile.txt” from the server):   **TAU** (client fails to get file “midSizeTextFile.txt” from the server as it is being read by UPSILON): 	**PASS**

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "1
92.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server V6WS3y3xcYLNXr19xlwm9D2vFGc=.
Enter Command: GET midSizeTextFile.txt
Receiving file...
File "midSizeTextFile.txt" downloaded successfully from the server in 6756
ms.

Enter Command:
```

No.	Time	Source	Destination	Protocol	Length	Inf
3319	115.445253720	192.168.1.72	192.168.1.78	UDP	70	1…
3320	115.446025927	192.168.1.78	192.168.1.72	UDP	71	4…
3324	117.194060961	192.168.1.78	192.168.1.72	UDP	66	4…
3325	117.196350322	192.168.1.72	192.168.1.78	UDP	71	1…
3337	120.526013102	192.168.1.78	192.168.1.72	UDP	66	4…
3338	120.528322268	192.168.1.72	192.168.1.78	UDP	71	1…
3346	123.861925357	192.168.1.78	192.168.1.72	UDP	66	4…
3347	123.865128516	192.168.1.72	192.168.1.78	UDP	71	1…
3381	127.193757686	192.168.1.78	192.168.1.72	UDP	66	4…
3382	127.196518202	192.168.1.72	192.168.1.78	UDP	71	1…
3393	130.526588845	192.168.1.78	192.168.1.72	UDP	66	4

```
▸ Frame 253: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on int
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 45696, Dst Port: 1024
▸ Data (1200 bytes)
```

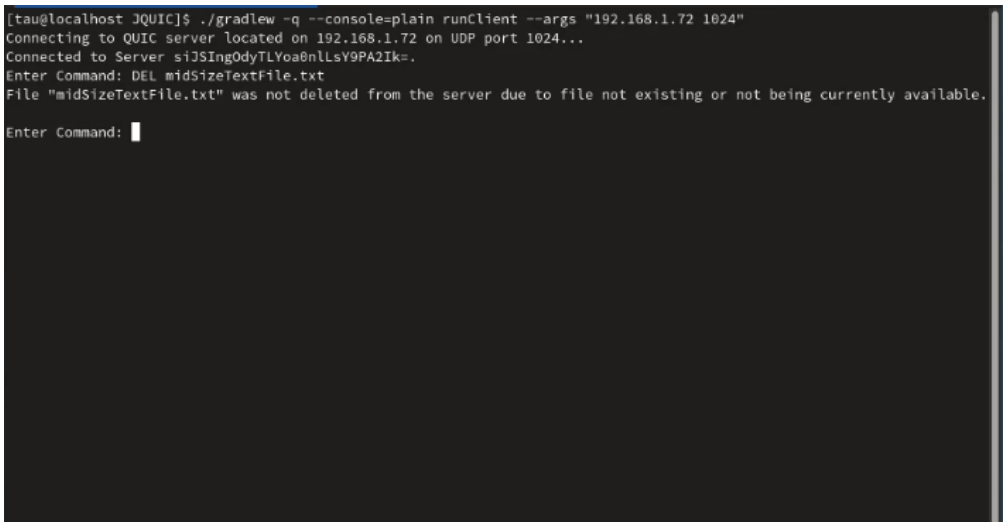
```
[tau@localhost JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server zMX3g+PLdTThj58bLwvMEFd/8/w=.
Enter Command: GET midSizeTextFile.txt
File "midSizeTextFile.txt" had not been downloaded as it is currently being edited by another client. Please try agai
n later.

Enter Command:
```

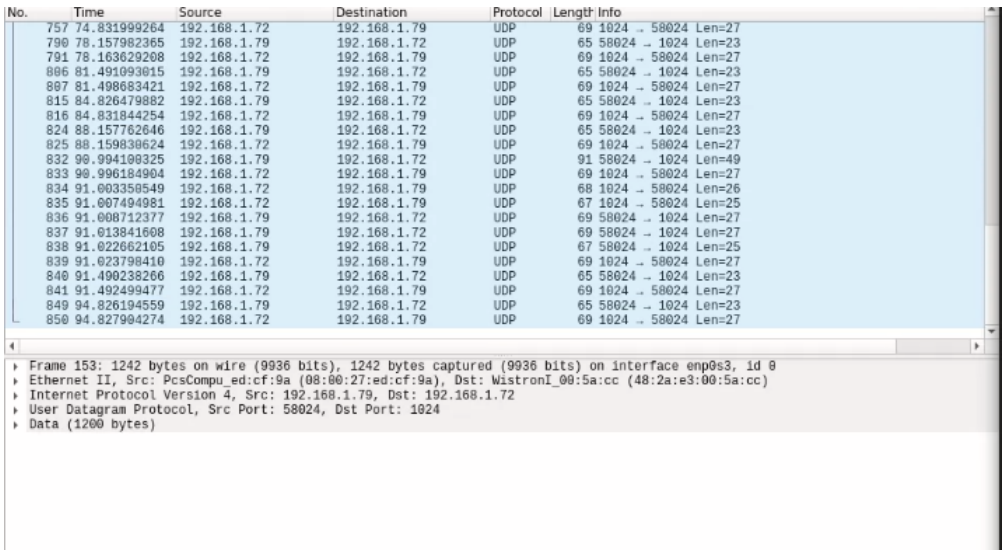
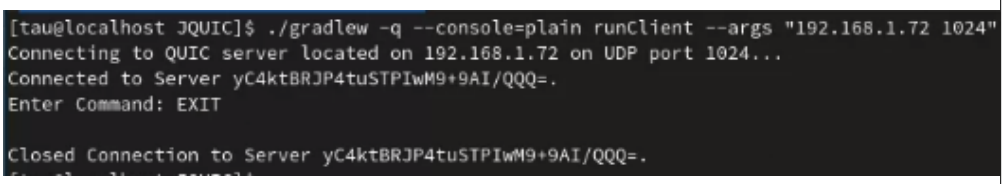
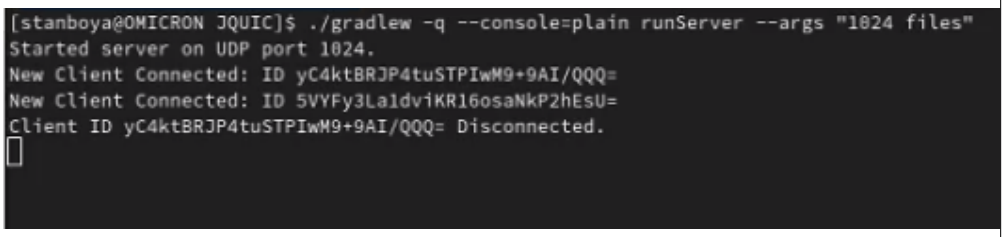
	 The output for **OMICRON** is similar to results for Acceptance Test 16.	
21	**UPSILON** (client successfully downloads file “midSizeTextFile.txt” from the server): 	**PASS**

No.	Time	Source	Destination	Protocol	Length	Inf
863	93.981391367	192.168.1.72	192.168.1.78	UDP	68	1…
864	93.983777603	192.168.1.78	192.168.1.72	UDP	69	3…
865	93.985695298	192.168.1.72	192.168.1.78	UDP	1069	1…
866	93.986913533	192.168.1.78	192.168.1.72	UDP	68	3…
867	93.989858226	192.168.1.72	192.168.1.78	UDP	1099	1…
868	93.991033913	192.168.1.78	192.168.1.72	UDP	69	3…
869	93.992962700	192.168.1.72	192.168.1.78	UDP	1069	1…
870	93.995172737	192.168.1.78	192.168.1.72	UDP	69	3…
871	93.996977236	192.168.1.72	192.168.1.78	UDP	69	1…
872	93.999321298	192.168.1.78	192.168.1.72	UDP	69	3…

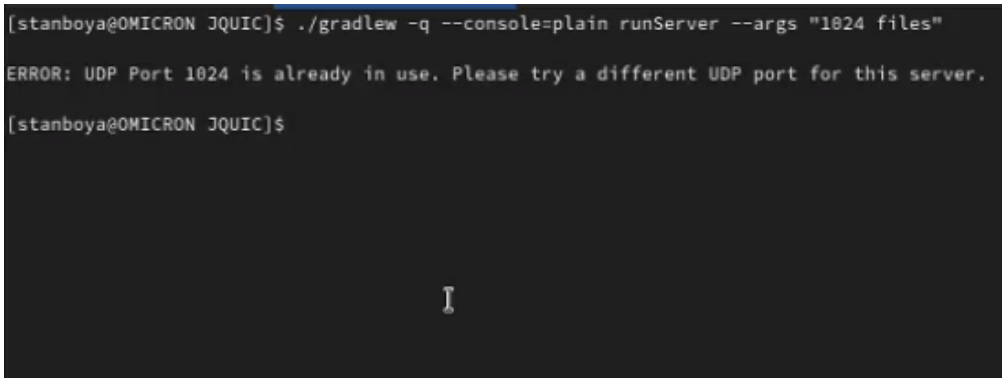
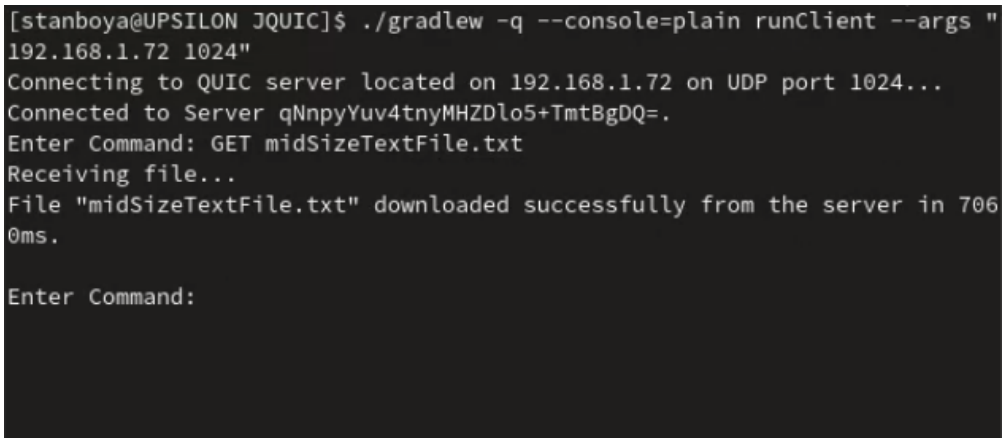
```
▸ Frame 185: 1242 bytes on wire (9936 bits), 1242 bytes captured (9936 bits) on int
▸ Ethernet II, Src: QuantaCo_ed:83:68 (04:7d:7b:ed:83:68), Dst: WistronI_00:5a:cc (
▸ Internet Protocol Version 4, Src: 192.168.1.78, Dst: 192.168.1.72
▸ User Datagram Protocol, Src Port: 39325, Dst Port: 1024
▸ Data (1200 bytes)
```

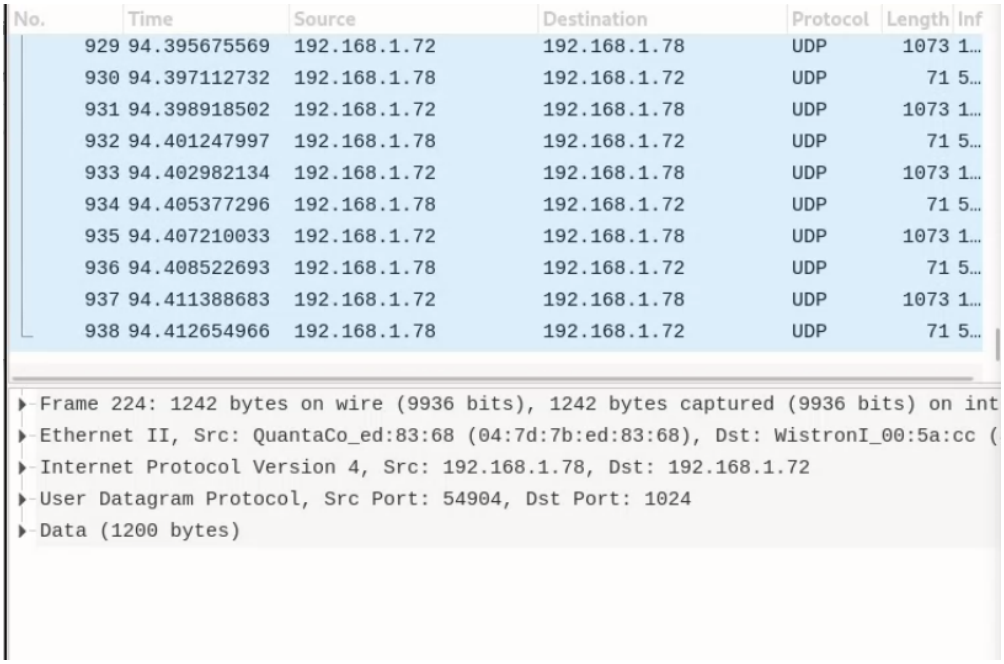
TAU (client fails to delete file “midSizeTextFile.txt” on the server as it is being read by UPSILON):



	 The output for **OMICRON** is similar to results for Acceptance Test 16.	
22	**TAU** (disconnects from the server on **OMICRON**):  **OMICRON** (server shows that **TAU** disconnected from it):  **UPSILON** (client is still able to send commands to server on **OMICRON** and receive back output for them):	**PASS**

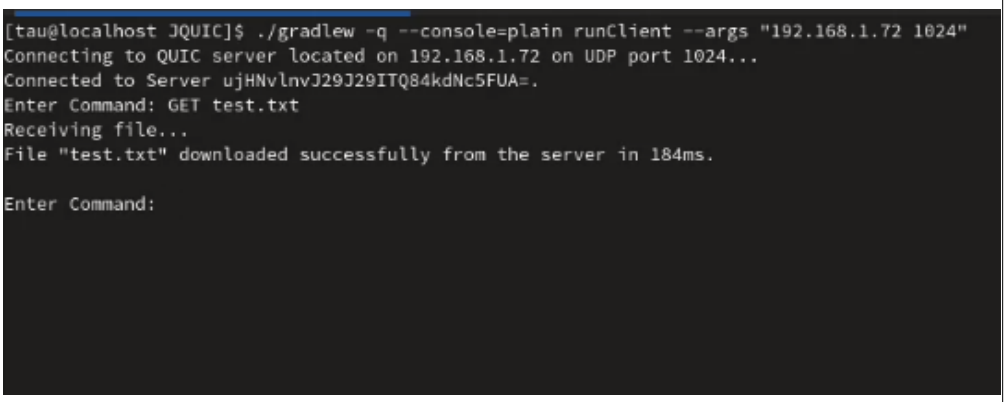
		
23	**OMICRON** (first server starts normally):  **OMICRON** (second server fails to start as the UDP port 1024 is already taken):	**PASS**

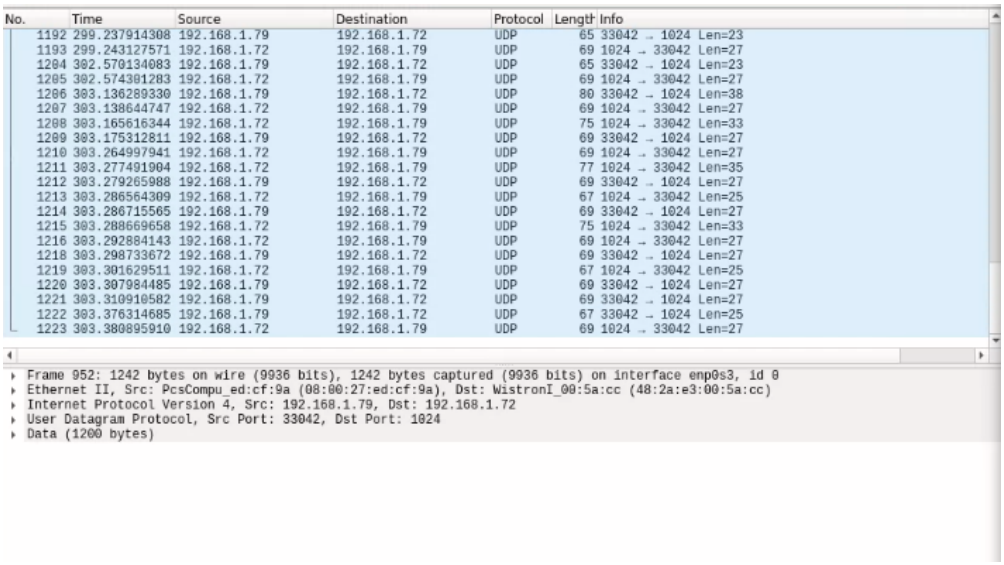
		
24	**UPSILON** (The file “midSizeTextFile.txt” is still successfully downloaded): 	**PASS**



No.	Time	Source	Destination	Protocol	Length	Inf
929	94.395675569	192.168.1.72	192.168.1.78	UDP	1073	1…
930	94.397112732	192.168.1.78	192.168.1.72	UDP	71	5…
931	94.398918502	192.168.1.72	192.168.1.78	UDP	1073	1…
932	94.401247997	192.168.1.78	192.168.1.72	UDP	71	5…
933	94.402982134	192.168.1.72	192.168.1.78	UDP	1073	1…
934	94.405377296	192.168.1.78	192.168.1.72	UDP	71	5…
935	94.407210033	192.168.1.72	192.168.1.78	UDP	1073	1…
936	94.408522693	192.168.1.78	192.168.1.72	UDP	71	5…
937	94.411388683	192.168.1.72	192.168.1.78	UDP	1073	1…
938	94.412654966	192.168.1.78	192.168.1.72	UDP	71	5…

TAU (File “test.txt” successfully downloaded from the server while client on UPSILON is also downloading):



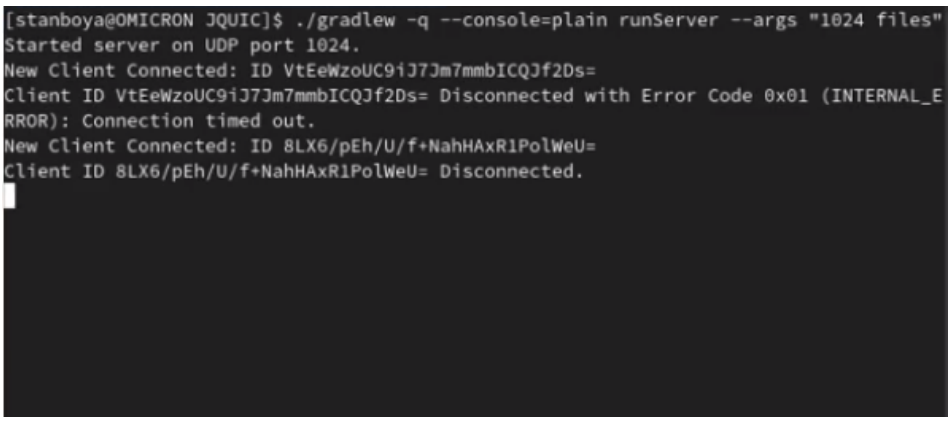


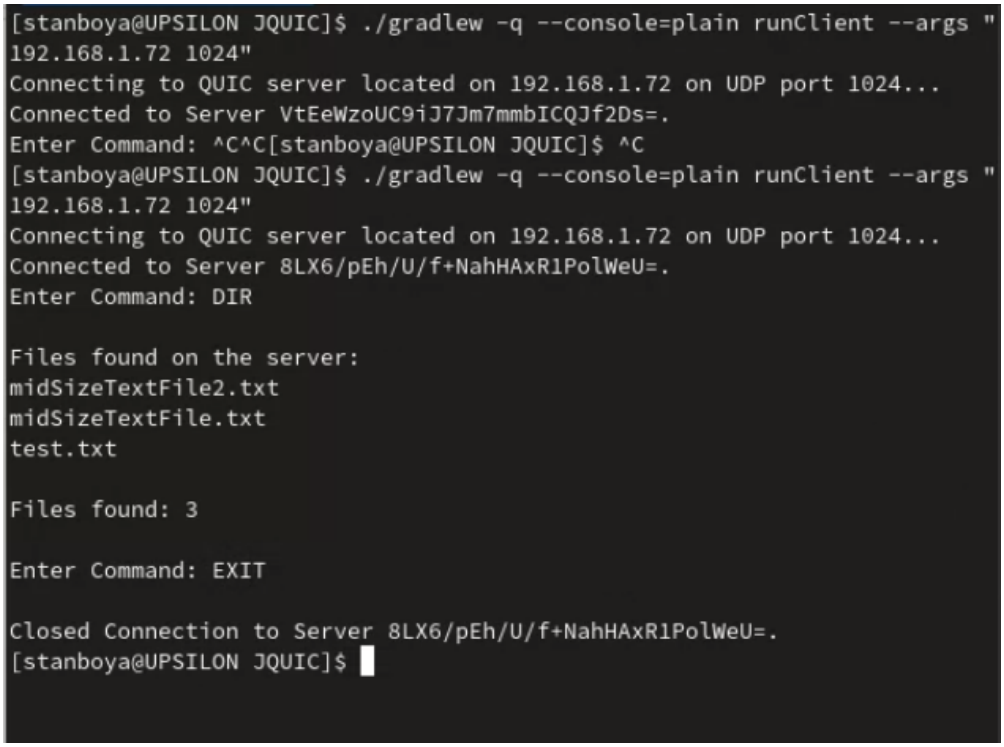
No.	Time	Source	Destination	Protocol	Length	Info
1192	299.237914308	192.168.1.79	192.168.1.72	UDP	65	33042 → 1024 Len=23
1193	299.243127571	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27
1204	302.570134083	192.168.1.79	192.168.1.72	UDP	65	33042 → 1024 Len=23
1205	302.574381283	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27
1206	303.136289330	192.168.1.79	192.168.1.72	UDP	80	33042 → 1024 Len=38
1207	303.138644747	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27
1208	303.165616344	192.168.1.72	192.168.1.79	UDP	75	1024 → 33042 Len=33
1209	303.175312811	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1210	303.264997941	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27
1211	303.277491904	192.168.1.72	192.168.1.79	UDP	77	1024 → 33042 Len=35
1212	303.279265988	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1213	303.286564309	192.168.1.72	192.168.1.79	UDP	67	1024 → 33042 Len=25
1214	303.286715565	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1215	303.288669658	192.168.1.72	192.168.1.79	UDP	75	1024 → 33042 Len=33
1216	303.292884143	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27
1218	303.298733672	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1219	303.301629511	192.168.1.72	192.168.1.79	UDP	67	1024 → 33042 Len=25
1220	303.307984485	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1221	303.310910582	192.168.1.79	192.168.1.72	UDP	69	33042 → 1024 Len=27
1222	303.376314685	192.168.1.79	192.168.1.72	UDP	67	33042 → 1024 Len=25
1223	303.380895910	192.168.1.72	192.168.1.79	UDP	69	1024 → 33042 Len=27

The output for **OMICRON** is similar to results for Acceptance Test 16.

25	**UPSILON** (printing out all the contents of the directory that the program was run in):	**PASS**

```
[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "
192.168.1.72 1024"
Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...
Connected to Server puVrgY3Q3o82TXWabqlVV1+72lA=.
Enter Command: LOCALDIR

Files in current directory:
midSizeTextFile.txt
.gradle
gradle
gradlew
src
settings.gradle.kts
build
test.txt
.gitignore
build.gradle.kts
.git
gradlew.bat
files

Files/Folders found: 13

Enter Command:
```

	The output for **OMICRON** is similar to results for Acceptance Test 5.	
26	**UPSILON** (printing out the absolute path of the directory that the program was run in):  The output for **OMICRON** is similar to results for Acceptance Test 5.	**PASS**
27	**OMICRON** (server shows a time out error after client immediately shuts down and then the client from **UPSILON** is able to connect again and exit gracefully afterwards):  	**PASS**

	UPSILON (Client instantly shut down by using CTRL+C command and then is able to connect again, run the command DIR and exit):  [stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024" Connecting to QUIC server located on 192.168.1.72 on UDP port 1024... Connected to Server VtEeWzoUC9iJ7Jm7mmbICQJf2Ds=. Enter Command: ^C^C[stanboya@UPSILON JQUIC]$ ^C [stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024" Connecting to QUIC server located on 192.168.1.72 on UDP port 1024... Connected to Server 8LX6/pEh/U/f+NahHAxR1PolWeU=. Enter Command: DIR Files found on the server: midSizeTextFile2.txt midSizeTextFile.txt test.txt Files found: 3 Enter Command: EXIT Closed Connection to Server 8LX6/pEh/U/f+NahHAxR1PolWeU=. [stanboya@UPSILON JQUIC]$	
28	**OMICRON** (server instantly shut down by using CTRL+C command):  [stanboya@OMICRON JQUIC]$ ./gradlew -q --console=plain runServer --args "1024 files" Started server on UDP port 1024. New Client Connected: ID e1GW4ebY4s7w8XhzBbMbTMd8NZo= ^C^C[stanboya@OMICRON JQUIC]$ ^C [stanboya@OMICRON JQUIC]$	**PASS**

	UPSILON (client shows time out error and exits): `[stanboya@UPSILON JQUIC]$ ./gradlew -q --console=plain runClient --args "192.168.1.72 1024"` `Connecting to QUIC server located on 192.168.1.72 on UDP port 1024...` `Connected to Server e1GW4ebY4s7w8XhzBbMbTMd8NZo=.` `Enter Command: Closed Connection to Server e1GW4ebY4s7w8XhzBbMbTMd8NZo= with Error Code 0x01 (INTERNAL_ERROR): Connection timed out.` `[stanboya@UPSILON JQUIC]$`	

Final Verdict: 28 of 28 Acceptance Tests passed.

2.7 Implications of Implementation

In the following section, I will describe the performance, stability, scalability as well as other aspects of the implementation such as which parts of the draft document were and were not implemented or only partially implemented and what consequences this will have on this implementation.

2.7.1 QUIC Version Used For Implementation

QUIC version 34 is used for this implementation which can be found here: https://www.ietf.org/archive/id/draft-ietf-quic-transport-34.html. This is because it was the latest version at the time of the implementation and there have been no QUIC versions after it so far.

2.7.2 QUIC Features Added from Draft Requirements

Here are the features that have been added in this implementation from the draft requirements:

- QUIC Streams and Stream States
- Flow Control include stream-specific, connection and stream limit
- Connection IDs (basic implementation)
- Handshake (though not cryptographic, only a mock-up)
- Version Negotiation
- Connection termination including Idle Timeout, Immediate Close and Stateless Reset
- Error Handling
- Coalescing packets
- Packet Numbers (including proper packet number spaces), but with one exception (talked about in next section)
- All the frame types (though not all of them are currently used)
- ACK's including proper ACK calculation from ACK frames (though construction of ACK frames is currently only with just largest acknowledged, no ranges)
- Loss Control (though not fully)
- Proper padding for datagrams containing InitialPacket
- All packet types except for 0-RTT packet (though RetryPacket is currently not used)
- Pinging so that the timeout does not happen
- Very basic version of Address Validation (only happens upon receiving handshake packets and limits before validating the address are currently not enforced)
- Majority of Transport Error Codes
- Variable Integer Encoding and Decoding
- Support for 1-RTT with stream frames during the handshake (though not currently used)

2.7.3 QUIC Features not Added from Draft Requirements

Here are the features that were not added from draft requirements and a short explanation of why they are not added:

- **Packet Encryption**: while QUIC requires encryption in order to be a full-fledged implementation, it was not added; it was already mentioned that it will not be added in

the Scope section in the proposal, but the reason it was not added is because QUIC requires TLS 1.3 while JDK11 does support TLS 1.3 for its SSLSocket library, it does not have a decent TLS 1.3 API to use (no way to access ClientHello or ServerHello calls for example) which would be required to add this feature; error codes for cryptographic errors have also not been added

- **Congestion Control**: this one is the most regrettable as theoretically, it would improve the performance of this implementation substantially; for the document that contains the requirements for congestion control called “quic-recovery”, it was said that “the features from this document might not be fully implemented” and while loss control was indeed implemented (at least in its basic implementation), congestion control was skipped; the reason it was decided not to implement this feature is that a large feature like congestion control requires very extensive testing with a large scale environment and multiple nodes through which all the data would flow through which the student did not really have access to at this time and poorly implemented congestion control can really harm the implementation even more so than if it was not there at all; in a lot of ways, congestion control for QUIC would be a project on its own, but will definitely be added in the future for this implementation
- **Address Validation (full implementation):** normally in QUIC, endpoints limit sending to non-validated addresses to three times the amount that the other side sent to them, but this limit is not enforced in the current implementation; the reason that address validation limit is not enforced (only validation through handshake) is that it requires many different ways to validate as for example, usage of Retry Packets which requires implementation of Retry Integrity Tag which requires implementation of TLS 1.3, but since not all methods of validations are there, it was decided not to do it
- **Connection Migration**: it was said in the scope that this feature will not be added as it would too difficult for the student to test; similar to Congestion Control, a feature like Connection Migration could probably be a project on its own
- **0-RTT packets**: These are packets that can be sent very early own in the handshake, but this requires that the client already has the specific cryptographic keys from the server; since this implementation does not really have keys (due to absence of TLS 1.3), these packets have no reasonable mock-up way to be implemented
- **Transport Parameters**: since TLS is currently just a mock-up, it was decided not to add dynamic transport parameters; transport parameters determine initial and important values for the connection (like for example initial maximum data limit to be sent); all the current Transport Parameters are hard-coded in “GlobalConstants.java” file
- **Full Packet Encoding and Decoding**: in full implementation, packet encoding and decoding requires certain functions that reduce the possible size of packet during transfer from 8 bytes to 4 bytes; however, this current implementation does not have that resulting in packets only having maximum size of 4 bytes (2^32 packet numbers)

instead of 8 bytes minus two bits (2^62 packet numbers); the reason for this is that due to lack of congestion control which causes a large amount of packets to be sent quickly, this encoding and decoding could result in incorrect packet numbers being read as says in the following statement in the draft: “A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received” (Iyengar and Thomson). As such, it was decided not to implement this feature as it could further destabilize this implementation, but will be implemented once congestion control is added

- **More complex connection ID implementation**: connections should have multiple multiple connection IDs, but this was not added as it was found to be somewhat pointless in this current implementation as connection migration and retry packets were not added; Connection IDs are also not persistent between connections, meaning that IDs for the same client will change after connection has ended and new one started; furthermore, only Destination Connection ID’s are used for connection identification and only by the server, the client determines whether it’s the right server by IP address and UDP port

Certain other smaller features and requirements have not been added (such as “stream ID that is used out of order results in all streams of that type with lower-numbered stream IDs also being opened” (Iyengar and Thomson)), but these have little impact on major inner workings of the protocol and are generally irrelevant in current implementation due to how it is structured.

2.7.4 Performance

While this project wasn’t necessarily aimed at performance, it was definitely considered especially when doing file transfers. Some performance were done such as decreasing the sending time (at cost of some stability, more on this later), slowing down the file reading (which improved the performance as client could not overwhelm the server with data as much and vice versa) and other small improvements which decreased the time to send a 1MB file from 15 minutes to around 7 seconds as can be seen in Acceptance Test results. While this result is not that great (getting the same file from gutenberg.org took around 1 second), you must keep in mind that this is an unfinished protocol implementation with no congestion control while developers of HTTP server that gutenberg.org runs on (most likely Apache) had years of development and performance improvement upkeep. Hopefully, future development for this implementation will also get to this type of performance or even better.

2.7.5 Stability

The implementation has been found to have some stability problems especially when client is running on a better machine than the server. A lot of time, this results in server sending a stateless reset token to the client as it is not able to read the UDP packets fast enough resulting in it reading only parts of some packets though this problem happened the other way as well. The two “duck tape” solutions in this implementation was to send UDP packets on a timer (currently every 4 ms) as that slows it down enough that it works in majority of cases and also to put some sleep statements in DFS Client and Server to make sure that the file sizes and booleans are received correctly and do not overlap each other (though this is more of a application implementation problem than the protocol itself). The UDP sending timer could also be increased to improve the stability with around 10ms being when crashes almost never happen but it was found that the performance suffers at that point in terms of slower file transfers. While flow control also helps, the real solution here is to implement congestion control as this would allow the client and server to tell when the other side is not able to cope with the amount of traffic coming in by detecting loss. This is exactly what will be focused on in future releases for this implementation.

2.7.6 Scalability

While the user did not have access to a large-scale testing environment, the DFS server worked fine with two DFS clients, being able to receive different files from them and send different files to them at the same time as seen in Acceptance Tests. This is because server is almost completely multi-threaded with different threads performing their own functions for the protocol and each DFS client being served in their own thread. Again, adding congestion control to this implementation would fix a lot of stability problems. It has also been found that even if one DFS client suffers a significant, unrecoverable error, server is still able to serve other clients and does not necessarily crash (though it can have memory problems) since all the clients are served on different threads so only one client thread would suffer a crash (though usually, the connection still gets closed properly even with a crash) and others would continue working which makes it so that the client can just connect again even if something goes wrong. Though one problem with that is that DFS Server deploys file locks to make sure multiple clients can’t read from and write to the same file at the same time which means that if one client crashes while uploading a file, that file lock could potentially still continue, resulting in some files becoming unusable unless the server is restarted.

2.7.7 Out-of-Order Stream Data

In the draft protocol, it says “QUIC makes no specific allowances for delivery of stream data out of order. However, implementations MAY choose to offer the ability to deliver data out of

order to a receiving application" (Iyengar and Thomson) which this implementation does as an extra feature in "quicStreamRecv" thread for each QUICStream. Basically, it tracks the size of data and offset from each STREAM frame, and inputs them in a map with key being the offset and value being the data in byte array form. Then, if previous data has been already been put into the stream to be read by the application, the program checks whether the distance between offset and previous offset is equal to size of the data and if does, it gets put into the stream to be read by the application as well. This makes sure that all the data is delivered and put into the stream in order.

2.7.8 Repositories for JQUIC and DFS

While in the proposal, it was mentioned that QUIC implementation (JQUIC in this case) and DFS will be two different repositories, it was decided to have them together with DFS being an example application for JQUIC because DFS was determined to currently be too small to have its own repository. Once more features are added to DFS, it will probably become a project on its own.

2.7.9 Multi-Threading-Friendly Data Structures Used

Rather than using semaphores, majority of communication between threads is done using LinkedBlockingQueue and ConcurrentMap which are data structures from Java Native Libraries. The reason for this is that the student wanted to have as little blocking between threads as possible and these two data structures allowed this as they are thread-friendly with LinkedBlockingQueue having capability to poll for certain amount of time.

PipedInputStream and PipedOutputStream were used to transport data between application and QUIC stream sending and receiving threads. They were found to be the most optimal solution as they have infinite buffer and are fairly quick.

Some atomic variables were also used such as AtomicLong for packet numbers and AtomicBoolean for many different features like determining when QUIC connection is closing so all the controller threads that are supporting it could shut down gracefully.

2.7.10 Connection Server Closing

While QUICConnection closes fine, QUICConnectionServer does not close UDPReceivingPacketController and ServerPacketController which could have major

repercussions when another QUICConnectionServer is created in same application. The reason they are not closed is because they are needed for QUICConnections that were created by that QUICConnectionServer to continue functioning as those contollers sort the packets to those QUICConnections by Connection ID. The student did not find an easy solution to this so far, but will try to fix this issue in the future.

2.8 Innovation

QUIC protocol in itself is a new protocol that was introduced publicly in 2013 (Weiss) which is pretty new for a transport layer protocol so any implementations for it is fairly innovative in itself. But there is also another component to this: protocol server implementation in a programming language that it has not been implemented yet. QUIC protocol itself is not only new, but has features that are innovative in nature such as the ones mentioned in "Problem Statement and Background" section such as its 0-RTT approach. The student will open-source the new QUIC Java API (but not the Distributed File System Application) after the report is submitted and will continue to improve on it based on the drafts to make sure everyone can benefit from his work and create their own Java applications that use QUIC as the transport layer protocol.

While there are multiple implementations of QUIC protocol in multiple different languages including Java client implementation called "kwik" ("Quicwg/Base-Drafts"), there hasn't been one for Java server implementation (at least not a well known one officially recorded by the working draft authors on their GitHub Wiki). Java Programming Language is one of the most used programming languages in the world based on the well-known TIOBE index ("TIOBE Index | TIOBE - The Software Quality Company") so not having a proper Java server implementation for a new transport-layer protocol seems like a problem that needs to be solved due to the protocol's potential as a new standard for reliable transport layer protocol as well as HTTP3.

There is also no well-known Distributed File System application written with QUIC as its transport layer so this is another innovative component to this project as well. It is a good way of looking at how easy it would be to implement QUIC in a more general networking application, that's not using HTTP protocol like Distributed File System.

Finally, in the implementation, some JDK11 features are used such as "StringUtils.repeat" function for printing, meaning that the student uses fairly new Java features in the implementation. TLS 1.3 will also be added to the implementation in the future which is a new TLS standard.

2.9 Complexity

As was written in the proposal for this project, “complexity for this project lies in the study and difficulty of implementation of the cutting-edge transport layer network protocol that is still only being drafted…”. While this is still true, there were other complex parts that were found during the development of this project. One major difficulty was getting all the threads to work together in an asynchronous way as well as determining which parts of protocol belong to each different controller thread. The other complex part was determining how an application would interact with these controllers through the API. For example, how does a QUIC endpoint wait for a stream from the other side? It was determined that the best way for these interactions to happen is asynchronously using data structures such as LinkedBlockingQueue and ConcurrentMap as well as PipedInputStream and PipedOutputStream as talked in “Implications of Implementation”.

For a diploma student, this maybe an almost impossible challenge, as they seldom do development with multiple different threads like this, especially in the Java courses. They also do not learn about flow control and do not implement reliable transport layer network protocols from scratch, especially newer ones and especially in programming languages other than C and C++.

The real problem to solve here is that the QUIC network protocol is still in draft and so, will be revised while the student develops it which would make the student constantly update the code based on the new draft changes. It is also about implement QUIC server API in a language that does not have an implementation like this, especially in a multi-threaded way with all components of the QUIC protocol working asynchronously.

The student must have specialized knowledge in networking applications and protocols to complete this application. They must understand networking concepts such as transport layer handshake and flow control.

Before this project began, the student lacked knowledge in networking stream implementations which QUIC uses. He lacked knowledge in QUIC transport layer protocol in general so he needed to research the protocol itself so that he could implement it. He also learned more about Java programming language, especially the thread-friendly data structures and streams, as he has forgotten it a little since he learned it in the diploma program many years ago. He also gained lots of valuable experience on working on a larger Java project with many classes and threads.

2.10 Research in New Technologies

Through this project, the student learned a lot about Java streams and data structures which were very important for multi-threading. In fact, there was a lot of research into best ways to do multi-threading in Java. There was also a lot of consideration for QUIC Streams and how their API should interact with the application. Some JDK 11 features were also used such as “StringUtils.repeat” function.

The student also researched a lot into Java Socket and ServerSocket APIs to create an API similar to them for this project.

In general, many parts of QUIC protocol had to be extensively researched and understood in order to implement this project. This includes flow control, QUIC streams, handshake, connection closing, connection IDs. It also took some time to understand how to transform these concepts into code including what design and tasks of different controller threads are and what class structure should be used for QUIC packets and frames.

The student also learned about how to use Gradle Build Tool to build and run larger Java projects as well as how to use JUnit to test them. Also, he learned about IDE called Intellij IDEA which he has not used before this project

2.11 Future Enhancements

First, as talked about in the proposal, after the committee approves this report, the student will open-source the JQUIC project so that other users online could help me develop it further into a full-fledged implementation and later, use it for their own projects. Here is the list of features/enhancements that I would like to add:

- Clean-up and re-organization of some parts of JQUIC to make it easier to develop on it further
- Congestion Control
- Proper Packet Number Encoding and Decoding
- Encryption and Decryption
- Transport Parameters
- Retry Packet
- Address Validation
- Connection Migration
- 0-RTT packet
- Other QUIC features that are added in future draft versions

After this point, the student will try to get it to work with other QUIC servers and clients for interoperability. DFS Application will also be worked on but in a closed source way.

2.12 Timeline and Milestones

2.12.1 Expected Versus Actual Timeline

Here is the table for the expected versus actual duration for each task (note that the milestone tasks were basically extra time given to tasks of each milestone so they are not part of actual duration):

ID	Task Name	Expected Duration	Actual Duration
1	*QUIC Java Implementation*: Set up basic structure	30 hours	35 hours
2	*QUIC Java Implementation*: Set up QUIC handshake	30 hours	35 hours
3	*QUIC Java Implementation*: Set up QUIC packet sending / receiving	30 hours	40 hours
4	*QUIC Java Implementation*: Set up QUIC frame sending / receiving	30 hours	10 hours
5	*QUIC Java Implementation*: Set up QUIC flow control	30 hours	25 hours
6	*QUIC Java Implementation*: Set up full error handling	30 hours	35 hours
7	*QUIC Java Implementation*: Regression / Manual Testing	30 hours	30 hours
8	**Milestone 1: Finish the QUIC Java Implementation**	10 hours	-
9	*Java QUICSocket API*: Set up QUICSocket API	20 hours	35 hours
10	*Java QUICSocket API*: Add multi-threading support	20 hours	25 hours
11	*Java QUICSocket API*: Regression / Manual Testing	20 hours	10 hours
12	**Milestone 2: Finish the QUICConnection API**	10 hours	-
-	*QUIC DFS*: Set up I/O for client and server	-	10 hours
13	*QUIC DFS*: Set up basic connection between client and server	5 hours	7 hours

14	*QUIC DFS*: Set up sending GET and SEND commands and sending and getting files from server	10 hours	20 hours
15	*QUIC DFS*: Set up sending DEL commands and deleting files from server	10 hours	10 hours
16	*QUIC DFS*: Set up multi-threading so that multiple clients can connect and execute commands at the same time	10 hours	10 hours
17	*QUIC DFS*: Acceptance Testing	10 hours	15 hours
18	**Milestone 3**: **Finish the QUIC DFS**	5 hours	-
19	*Report*: Write the report	20 hours	25 hours
20	*Report*: Revise the report	10 hours	10 hours
21	**Milestone 4: Complete the Report**	5 hours	-
-	**TOTAL**	375 hours	387 hours

2.12.2 Milestone Completion Dates

Here is the table for each milestone and the date it was completed:

Milestone	**Completion Date**
Finish the QUIC Java Implementation	February 24th, 2021 (with some small changes and bug fixes afterwards)
Finish the QUICConnection API	February 24th, 2021 (with some small changes and bug fixes afterwards)
Finish the QUIC DFS	March 2nd, 2021 (with some small changes and bug fixes afterwards)
Complete the Project Report	March 19th, 2021 (not counting possible edits based on Supervisor and Committee comments)

The reason that the first two milestones are completed at the same time is because it was decided to develop them along one another so when the QUIC implementation was finished, the API implementation was pretty much done as well.

3 Conclusion

The student has a background in security and networking software development and this project was mostly focused on the networking software part as project was about creating a base for a full implementation of QUIC protocol. From this project, the student learned a lot about asynchronous multi-threading, streams (both in Java and networking), flow control, as well as how to set up and work on a large networking project made out of multiple classes and threads. The student also learned how to translate network protocol draft document requirements into implementation which is very important since most real-world projects are made from specific requirements that the developer has to fulfill. The student also got to practice multiple types of testing (Manual Testing, Automated Testing and Acceptance Testing). The student also got to practice separating application implementation from protocol implementation which helped him understand more about software and architecture design.

3.1 Lessons Learned

The student got a lot of experience in network programming and software architecture design. He learned how hard it is develop and test a reliable transport layer protocol, how difficult it can be to design a proper API for it and that it is better to be able to test something rather than to have as many features as possible. Also, he learned that separating the network protocol into different controllers will make it easier to manage especially if it is something as complex as network protocol with many different parts interacting with each other at all times.

3.2 Closing Remarks

Throughout this project, the student felt somewhat lost at points and unfocused. He wasn't quite sure which protocol features to flesh out and which ones to leave out. He also had trouble designing the controllers as it was somewhat hard for him to separate out specific functions of the protocol since everything seemed so entangled together. And yet, what came out of this project isn't too bad and with some restructuring, could serve as a decent base for a more full-fledged implementation. So in that way, this project was successful and achieved its objectives (with some

stability problems) though the student felt somewhat more ambitious when he was writing the proposal. He will open-source and continue working on this implementation along with other developers who want to help and develop it into a full-fledged version of QUIC it is meant to be so that it could someday be used as an API for many applications.

4 Appendix

4.1 Terminology

Here is terminology for all the abbreviations that are found in this report:

- **DFS**: Distributed File Server; built on top of QUIC implementation for this project as an example application to demonstrate the API as well as the protocol itself
- **JQUIC**: the main name for this project which includes QUIC protocol implementation in Java, an API to interact with this implementation as well as the example DFS application
- **1-RTT Packet**: 1-RTT stands for “1 Return Trip Time”, but for packets, it is another name for ShortPacket and vice versa

4.2 Approved Proposal

Note: the proposal is in slightly smaller font that the rest of the text to save some space.

4.2.1 Student Background

The student’s name is KB. He is currently a student at BCIT. He specializes in development of networking and security applications.

4.2.1.1 Education

4.2.1.2 Work Experience

4.2.2 Project Description

The aim of this project would be to create a Java server and client implementation and API based on the latest draft of Quick UDP Internet Connections (QUIC) protocol (Iyengar and Thomson) and then to create a Distributed File Server (DFS) application based on this Java QUIC API. Latest IETF draft standard of QUIC that is available at the time will be used to do the Java API implementation and will be updated along with the new drafts that come out which brings an extra challenge to this project. The implementation of DFS application would be fairly standard with a multi-threaded central server, that clients can connect to simultaneously, to edit, delete and store files on as it is mostly just a demonstration application for the API.

4.2.3 Problem Statement and Background

Java Programming Language is one of the most used programming languages in the world based on the well-known TIOBE index ("TIOBE Index | TIOBE - The Software Quality Company"). The current commonly used transport layer protocol, Transport Control Protocol (TCP), already has a lot of implementations in Java including the official Socket API implementation by Oracle (Socket (Java Platform SE 7)), while **Quick UDP Internet Connections** (QUIC) protocol only has one called "kwik" and it's only an API for client-side ("Quicwg/Base-Drafts") which is a problem for the reasons mentioned below. QUIC is a new "multiplexed and secure general-purpose transport protocol", that provides "stream multiplexing, stream and connection-level flow control, low-latency connection establishment, connection migration and resilience to NAT rebinding, authenticated and encrypted header and payload"(Iyengar and Thomson). This protocol was first unveiled publicly by Google in June 2013 (Weiss) and is now in the stage of rapid development (Kakhki et al. 86) with multiple drafts of the protocol submitted to Internet Engineering Task Force (IETF) each year. QUIC is actually built on top of IP/UDP stack (Pandya et al. 27) so it is a part of application layer, but implements a majority of transport layer features for reliability like congestion control (Kakhki et al. 86) since UDP is unreliable

and does not have them as per the original UDP specification (Postel). The diagram below shows where QUIC fits in as part of the network stack:

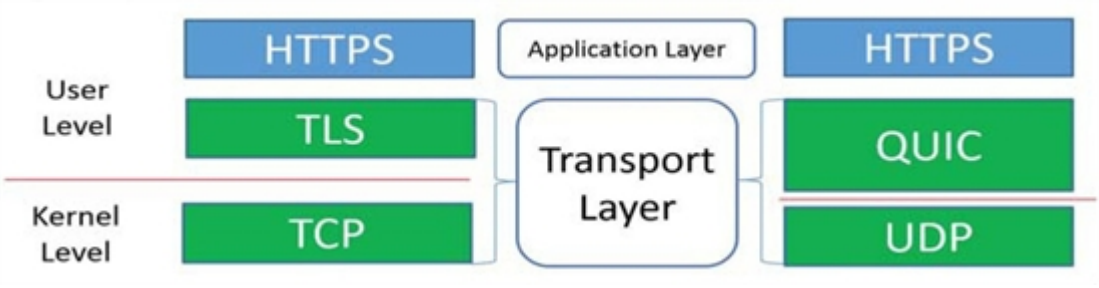


Figure 2: TLS vs QUIC protocol stack (Pandya et al. 27)

The reason to implement this new protocol is that QUIC has numerous advantages over Transport Control Protocol (TCP), which is the current standard protocol for reliable connections (Joshi and Krishnamurthy), due to both its modernity and its application layer implementation as mentioned before. For example, **Transport layer congestion control** is a very important feature of the TCP/IP network stack as it allows for "both fair and high utilization of Internet links shared by multiple flows" (Kakhki et al. 86). QUIC improves on TCP's congestion control by implementing "...better estimation of connection Round-trip Time (RTTs) and [detecting] and [recovering] from loss more efficiently" (Kakhki et al. 87). One of the largest benefits of QUIC over TCP is the much faster connection establishment due to ability to establish connections without the three-way handshake when the client has connected to the server before (Kakhki et al. 87). There are other beneficial features to QUIC such as fixing "head of line" blocking problem which is present in current IP+TCP+HTTP/2 stack. "Head of line" blocking happens because "HTTP/2 allows multiple objects to be fetched over the same connection, using multiple streams within a single flow" (Kakhki et al. 87). TCP causes all streams to stall when a loss occurs in even a single stream. In contrast, QUIC allows other streams to continue exchanging packets (Kakhki et al. 87). QUIC also has "forward error correction and improved privacy and flow compared to TCP" (Kakhki et al. 87). QUIC also has connection IDs which is useful for maintaining connection when changing networks like for example, changing from WiFi to mobile data network (Pandya et al. 27). This can be very useful when for example, driving around the city and automatically connecting to multiple different public WiFi networks.

In terms of popularity, QUIC is already "enabled by default in Google's Chrome browser and underlying Chromium open source browser code" (Mulks). Furthermore, HTTP/3, the next version of HTTP standard uses QUIC as its transport layer protocol and in fact was previously known as "HTTP over QUIC" (Ghedini). This solidifies QUIC as an upcoming reliable transport layer protocol standard that might overtake TCP in the future in terms of popularity.

The above reasons show that there are reasons to use QUIC over TCP which include its advantages outlined above as well as its upcoming usage as main transport-layer protocol for HTTP/3 stack which will no doubt be used as a standard protocol for majority of API and web browser communications in the future once more browsers support it so it makes sense to create a QUIC Implementation,

Client/Server API and a test application like Distributed File Server in Java to server as foundation for an HTTP/3 implementation or other applications.

However, there are some drawbacks to QUIC. While "QUIC outperforms TCP+HTTPS [stack] in nearly every scenario" according to the tests done by both Google and the paper authors (Kakhki et al. 87), QUIC has problems with out-of-order packet delivery which it interprets as loss and so, "performs significantly worse than TCP in many scenarios"(Kakhki et al. 87). QUIC also suffers performance problems on mobile devices due to its "application-layer packet processing and encryption" (Kakhki et al. 87) which can be a huge setback due to popularity of mobile devices with more than 50% of worldwide traffic being mobile phones ("Mobile Vs. Desktop Usage (Latest 2019 Data)"). These drawbacks might be rectified as the protocol is further developed and having more implementations for the protocol will only serve to further help the protocol be developed as it will find potential problems with the protocol in practice.

4.2.4 Scope and Depth

The main features and structure of QUIC protocol are defined in 4 different draft documents which are constantly being updated: **quic-transport**, **quic-tls**, **quic-recovery** and **quic-invariants**. Some features of QUIC protocol will be implemented based on the most recent draft or release candidate of the protocol based on those four documents. Here is what those documents describe (as well as how much of that document will be implemented):

- **quic-transport**: describes the core of the QUIC protocol (Iyengar and Thomson); all the features from this document will be implemented, but might be simplified or mocked depending on their dependence on features in **quic-tls** and **quic-recovery**
- **quic-tls**: describes how the Transport Layer Security (TLS) is used to secure QUIC (Thompson and Turner); the features from this document will probably not be fully implemented or not be implemented at all
- **quic-recovery**: describes loss detection and congestion control mechanism of the QUIC protocol (Iyengar and Swett); the features from this document might not be fully implemented
- **quic-invariants**: describes properties that are unchanged between the draft versions for QUIC Protocol (Thompson); all features of this document will be implemented

TLS 1.3 for QUIC cryptographic handshake (Iyengar and Thomson) will probably be a mock-up due to Java not having proper library for low-level TLS 1.3 calls. Connection migration will also probably not be included as it would be very difficult for the student to test it in a reliable manner. Beyond the features listed in these four documents (and only some of them as described above), no other additional features will be added for QUIC protocol implementation.

In terms of the QUIC Distributed File Server Application, it will have the following features:

- Built with IP/QUIC Stack (not full implementation, some features are mocked or are absent as mentioned above)

- Ability for users to create and delete files on the server
- Locally-hosted
- Multi-threaded server (multiple clients can connect to the server and execute tasks at the same time)

Most of the time spent in this project will be used to implement the QUIC protocol. DFS application in this case is more of demonstration application for QUIC Java API so its features will be fairly sparse. Here are the features that are currently out-of-scope for QUIC DFS (some of these might be implemented if any time is left over):

- End-to-End Encryption (due to cryptographic handshake not being implemented)
- Graphical User Interface (GUI)
- File auditing
- Web Interface (using HTTP/3)
- Any other feature that is not mentioned in the above feature list for DFS

4.2.5 Test Plan

The testing for this project consists of a combination of regression, manual and acceptance testing. Regression testing will be done using JUnit unit testing framework and will be mostly used for the protocol implementation to verify that all of the protocol features and requirements are implemented correctly according to the draft. Acceptance testing will be mostly focused on the Distributed File Server portion of this project.

4.2.5.1 Regression & Manual Testing

Regression and Manual Testing will be done by first outlining all the draft requirements and translating them into test cases. Here are some examples of regression and manual testing that will need to be done to satisfy the draft requirements (Iyengar and Thomson):

Draft Requirement	**Test**	**Passing Criteria**
“An endpoint could receive data for a stream at the same stream offset multiple times. Data that has already been received can be discarded. The	Endpoint 1 changes the data in the STREAM frame and sends at the same stream offset as another STREAM frame to Endpoint 2 while	Endpoint 1 receives back a CONNECTION_CLOSE frame with error type PROTOCOL_VIOLATION

data at a given offset MUST NOT change if it is sent multiple times; an endpoint MAY treat receipt of different data at the same offset within a stream as a connection error of type PROTOCOL_VIOLATION.”	in “Send” state.	from Endpoint 2.
“Sending the first STREAM or STREAM_DATA_BLOCKED frame causes a sending part of a stream to enter the "Send" state.” “The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) is created when the first STREAM, STREAM_DATA_BLOCKED, or RESET_STREAM frame is received for that stream… The initial state for the receiving part of a stream is "Recv".”	Endpoint 1 sends a mock STREAM frame to Endpoint 2 when it is in “Ready” state.	Endpoint 1 enters the “Send” state. Endpoint 2 enters the “Recv” state.
“After the application indicates that all stream data has been sent and a STREAM frame containing the FIN bit is sent, the sending part of the stream enters the "Data Sent" state.” “When a STREAM frame with a FIN bit is received, the final size of the stream is known… The receiving part of the stream then enters the "Size Known" state.”	Endpoint 1 sends a STREAM frame with FIN bit set to Endpoint 2.	Endpoint 1 enters the “Data Sent” state. Endpoint 2 enters the “Size Known” state.
“From any of the "Ready", "Send", or "Data Sent" states, an application can signal that it wishes to abandon transmission of stream data… the endpoint sends a RESET_STREAM frame, which causes the stream to enter the "Reset Sent” state.” “Receiving a RESET_STREAM frame in the "Recv" or "Size Known" states causes the stream to enter the "Reset Recvd" state.”	Endpoint 1 sends a RESET_STREAM frame while in “Send” state to Endpoint 2.	Endpoint 1 enters the “Reset Sent” state. Endpoint 2 enters the “Reset Recvd” state.
“If a sender has sent data up to the	Endpoint 1 keeps sending to	Endpoint 1 sends

limit, it will be unable to send new data and is considered blocked. A sender SHOULD send a STREAM_DATA_BLOCKED or DATA_BLOCKED frame to indicate to the receiver that it has data to write but is blocked by flow control limits.”	Endpoint 2 until it reaches flow control limit set by Endpoint 2.	STREAM_DATA_BLOCKED frame to Endpoint 2.
“If a RESET_STREAM or STREAM frame is received indicating a change in the final size for the stream, an endpoint SHOULD respond with a FINAL_SIZE_ERROR error“	Endpoint 1 sends the final size to Endpoint 2, but sends data beyond this final size.	Endpoint 2 responds with a CONNECTION_CLOSE frame with error type FINAL_SIZE_ERROR to Endpoint 1.
“If a max_streams transport parameter or a MAX_STREAMS frame is received with a value greater than 2^60, this would allow a maximum stream ID that cannot be expressed as a variable-length integer… If either is received, the connection MUST be closed immediately with a connection error of type TRANSPORT_PARAMETER_ERROR if the offending value was received in a transport parameter or of type FRAME_ENCODING_ERROR if it was received in a frame…“	Endpoint 2 sends a QUIC packet with transport parameter with value of 2^60 + 1 during the handshake to Endpoint 1.	Endpoint 1 responds with a CONNECTION_CLOSE frame with error type TRANSPORT_PARAMETER_ERROR to Endpoint 2.
“A server MUST discard an Initial packet that is carried in a UDP datagram with a payload that is smaller than the smallest allowed maximum datagram size of 1200 bytes.**“**	Endpoint 1 sends an Initial packet with payload smaller than 1200 bytes to Endpoint 2.	Endpoint 2 discards the Initial packet sent by Endpoint 1.
“An endpoint that receives a STOP_SENDING frame MUST send a RESET_STREAM frame if the stream is in the Ready or Send state.**“**	Endpoint 1 receives STOP_SENDING frame from Endpoint 2 while in “Send” state.	Endpoint 1 sends RESET_STREAM frame to Endpoint 2.
“A sender MUST ignore any MAX_STREAM_DATA or MAX_DATA frames that do not increase flow control limits.”	Endpoint 2 sends MAX_STREAM_DATA frame to Endpoint 1 does not increase flow control limit.	Endpoint 1 discards the MAX_STREAM_DATA frame received from Endpoint 2

Most of the regression/manual tests will be done to satisfy the “MUST” and “MUST NOT” requirements found in the draft. More different tests will be added as the project progresses and features are implemented.

4.2.5.2 Acceptance Testing

Here are some examples of acceptance testing that will need to be done in order to satisfy the requirements of the DFS application:

- Client is able to connect to the DFS server
- Client is able to upload file to DFS server
- Client is able to check which files already exist on the DFS server
- Client is able to download file from DFS server
- Client is able to delete file on the DFS server
- Client is able to replace file on the DFS server
- Multiple clients can connect to the DFS server at the same time
- Multiple clients can upload and download files to the DFS server at the same time

4.2.6 Methodology

4.2.6.1 Methodology

This project is technically two different projects (QUIC protocol implementation plus API and the Distributed File Server) and different methodologies will be used.

For QUIC protocol implementation, a waterfall approach will be used as different parts of the protocol build upon each other to conform to the draft so waterfall works best for this as it is about building large projects up front. However, even with waterfall approach, the implementation will still be divided into multiple sub-projects based on the features that the QUIC protocol has according to the draft document (Iyengar and Thomson).

Figure 3: Waterfall Methodology

For QUIC DFS, a more Agile approach will be used where each requirement will be implemented in short 2-3 day sprints as this program builds on top of QUIC protocol using the API and has a fairly small basic implementation so it does not require a lot of code up front. The implementation for both

the protocol implementation and DFS will be executed serially rather than in parallel (like it usually is for Agile methodology) as Konstantin is the only developer for this project.

4.2.6.2 Approach

As mentioned earlier, the project is technically two different sub-projects so it will be done in four different phases:

1. First phase will be the Java QUIC protocol API implementation. It will include the features of the draft requirements (Iyengar and Thomson) mentioned in the "Scope and Depth" section of this proposal. The implementation will be sufficient enough to implement Distributed File Server client and server based on it. This API will be similar to Java Socket API used for protocols like TCP and UDP and will be built on top of UDP.
2. The basic QUIC Distributed File Server will be implemented based on the acceptance testing requirements shown in the "Test Plan" section of this proposal. Both client and server will be implemented at the same time and use the same QUIC protocol API base that will be developed in step 1.
3. Extra features like the proper cryptographic handshake and congestion control will be added to the QUIC protocol implementation as well as QUIC DFS extra features like encryption, web interface and file auditing will be added if there is enough time.
4. Final stage of the project will be about creating the report as well as ironing out any bugs left over in QUIC Protocol and QUIC DFS. QUIC protocol will be open-sourced when the report and the code are handed over to the committee.

4.2.6.3 Technologies

For project management, the following technologies will be used:

- **GitHub Project Boards**: this is an online service on GitHub used for organization of tasks and issues that allows the users to add and move tasks through several stages until completion as the codebase is being developed. There will be two project boards: one for QUIC API implementation and the other for QUIC DFS.
- **Focus To-Do**: a Chrome extension/Android app that allows the user to add tasks to be done by certain dates and to complete Pomodoros (25-55 minute work periods) based on those tasks; this allows the user to pace himself and check and adjust time estimates for each task and make sure everything is done according to the schedule

For software design, the following tools and technologies will be used:

- **Draw.io**: a tool created by “diagrams.net” and used to create designs and diagrams for various purposes; in this case, it will be used to create and adjust system/software architecture diagrams (including the ones in this proposal)

For software development, the following tools and technologies will be used:

- **Java Programming Language**: a class-based, object-oriented programming language that is very popular (“TIOBE Index | TIOBE - The Software Quality Company”) with a large developer ecosystem and support
- **OpenJDK 11 (Open Java Development Kit 11)**: a free and open-source Java programming language implementation (licensed under the GNU-GPLv2) that both the QUIC protocol API and DFS application will be built upon. Its native library, called DatagramSocket, will be used for UDP sending/receiving QUIC protocol is built on top of UDP protocol according to the specification as discussed in “Problem Statement and Background” section.
- **IntelliJ IDEA Ultimate Edition**: an Integrated Development Environment (IDE) for JVM with support for Java, Kotlin and other JVM languages that includes a debugger, profiling tools and git integration. It will be used to write all the code for this project
- **Gradle**: a build automation tool that will be used to quickly build the project
- **Git**: version control tool which will be used for this project
- **GitHub**: a hosting website for software development and version control. Will be used for online repository storage for the project for quicker and more organized deployment on multiple machines which is especially important for this project as it has a heavy networking component and requires testing from multiple machines.
- **JUnit**: unit testing framework for Java programming language; will be used in regression testing in this project.

4.2.7 System/Software Architecture Diagram

In terms of the System architecture, the following shows how the program will be built in terms of implementation stack:

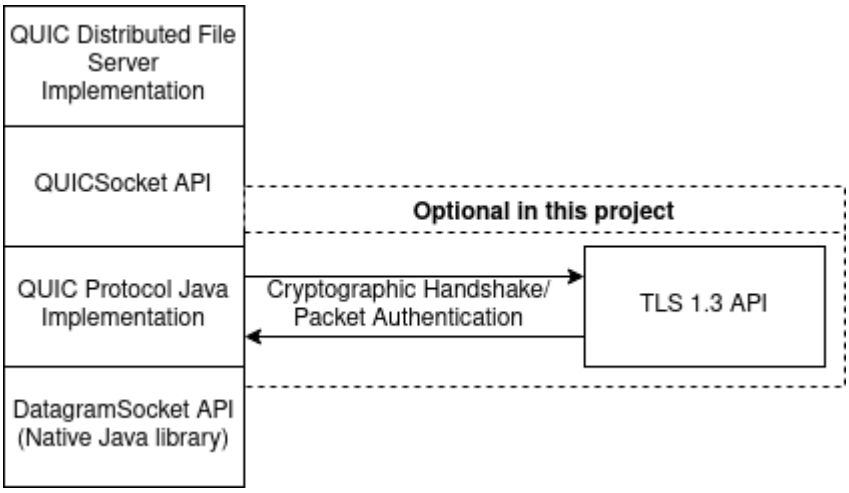


Figure 4: Implementation Stack

Here is what the architecture diagram looks like in terms of DFS server-client interactions and how different components of the stack interact with each other:

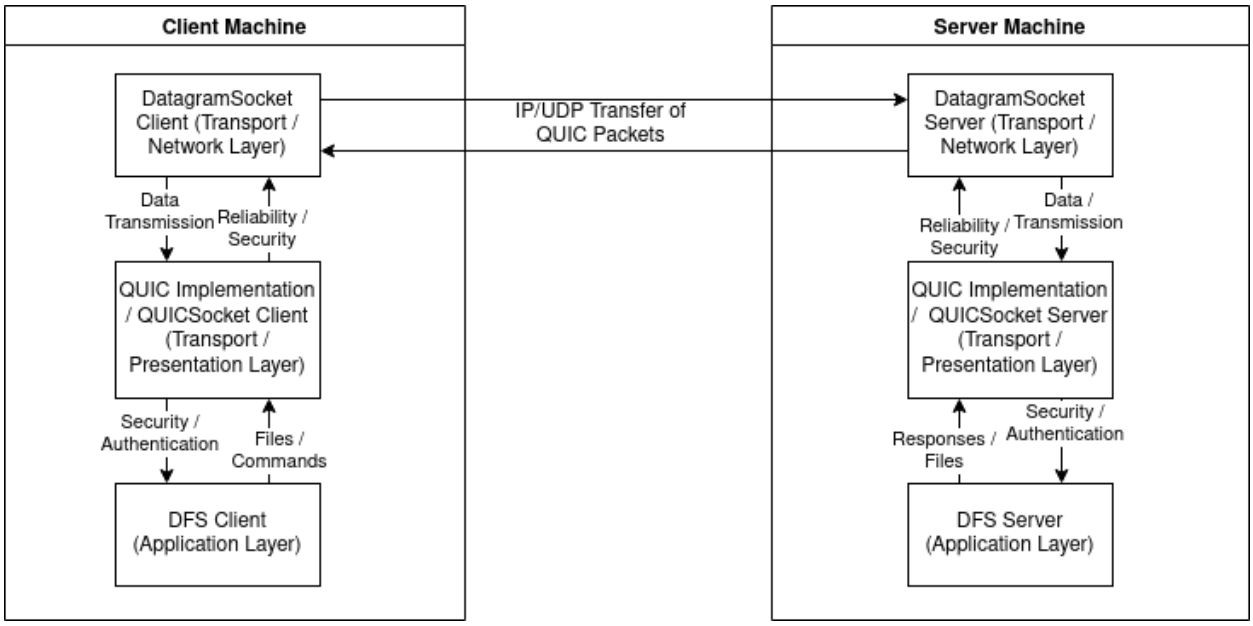


Figure 5: Client-Server Component Interaction

Note that some protocols like Ethernet are omitted in the above diagram as they are mostly abstracted by DatagramSocket API and are not directly specified.

Here is also a state diagram of the interaction between QUIC DFS client and server:

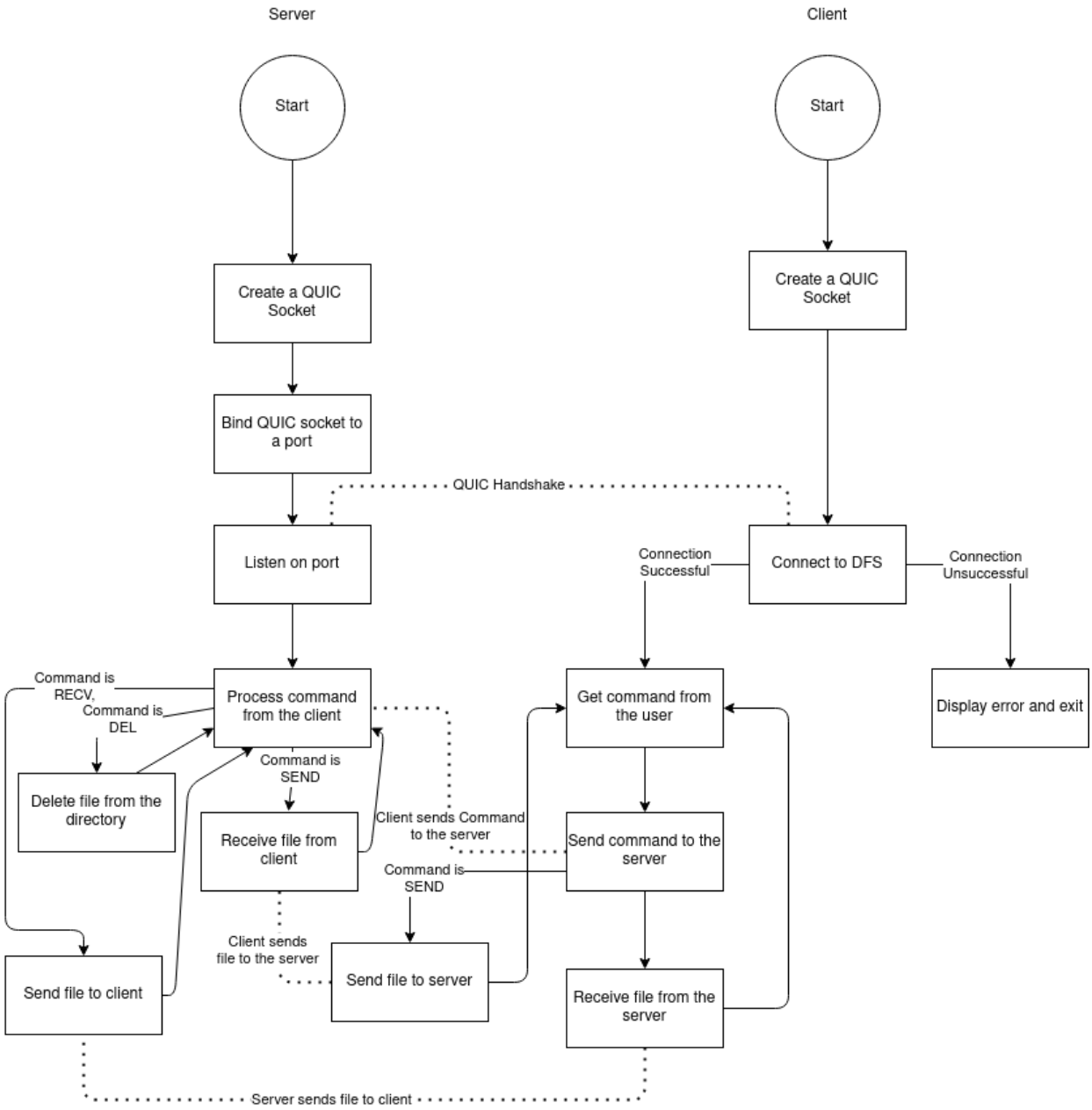


Figure 6: State Diagram

4.2.8 Innovation

QUIC protocol in itself is a new protocol that was introduced publicly in 2013 (Weiss) which is pretty new for a transport layer protocol so any implementations for it is fairly innovative in itself. But there is also another component to this: protocol server implementation in a programming language that it has

not been implemented yet. QUIC protocol itself is not only new, but has features that are innovative in nature such as the ones mentioned in “Problem Statement and Background” section such as its 0-RTT approach. The student will open-source the new QUIC Java API (but not the Distributed File System Application) after the report is submitted and will continue to improve on it based on the drafts to make sure everyone can benefit from his work and create their own Java applications that use QUIC as the transport layer protocol.

While there are multiple implementations of QUIC protocol in multiple different languages including Java client implementation called “kwik” (“Quicwg/Base-Drafts”), there hasn’t been one for Java server implementation (at least not a well known one officially recorded by the working draft authors on their GitHub Wiki). Java Programming Language is one of the most used programming languages in the world based on the well-known TIOBE index (“TIOBE Index | TIOBE - The Software Quality Company”) so not having a proper Java server implementation for a new transport-layer protocol seems like a problem that needs to be solved due to the protocol’s potential as a new standard for reliable transport layer protocol as discussed above.

There is also no well-known Distributed File System application written with QUIC as its transport layer so this is another innovative component to this project as well. It is a good way of looking at how easy it would be to implement QUIC in a more general networking application, that’s not using HTTP protocol like Distributed File System.

4.2.9 Complexity

Complexity for this project lies in the study and difficulty of implementation of the cutting-edge transport layer network protocol that is still only being drafted, meaning that there are constant changes to the protocol which the developer will have to work with or around. Furthermore, the developer will need to understand the advanced parts of congestion control, flow control, streaming. QUIC also has a lot more different states than TCP which the developer will have to keep mindful of and implement.

For a diploma student, this might present an insurmountable challenge as protocol features like congestion and flow control are seldom the focus in the diploma program and diploma students never have to implement entire reliable transport layer network protocols from scratch, especially newer ones. Also, students in diploma program for data communications option are usually supposed to write all the code in C/C++ so they may not have the full knowledge of how to implement the network protocols in other languages like Java.

The real problem to solve here is that the QUIC network protocol is still in draft and so, will be revised while the student develops it which would make the student constantly update the code based on the new draft changes. It is also about implement QUIC server API in a language that does not have an implementation like this.

The student must have specialized knowledge in networking applications and protocols to complete this application. They must understand networking concepts such as congestion, three-way handshake and flow control.

The student currently lacks knowledge in networking stream implementations which QUIC uses as well as TLS 1.3 which QUIC uses for encryption (Iyengar and Thomson). He lacks knowledge in QUIC transport layer protocol in general so they will need to research the protocol itself so that they could implement it. He also needs to do more studying of Java programming language as he has forgotten it a little since he learned it in the diploma program.

4.2.10 Technical Challenges

This project is technically challenging because network protocols are generally hard to implement (especially new ones with advanced features) and we have never implement a full transport layer protocol in class and the student has not done it before himself either. Transport layer network protocol implementations usually include congestion and flow algorithms which are pretty complex, even for networking student, especially this is a new protocol with more modern, complex congestion and flow algorithms.

The other part, as mentioned before, is that QUIC is still in development so it will be changing over time which makes it technically challenging to make sure to update the protocol with each new draft that comes out especially as the implementation API is getting developed. The student might have to change the protocol implementation even while they are implementing the distributed file system. Only in the last 4 weeks of the project will the student lock to a specific version of the draft for the report, but they will most likely keep working on it after the major project is completed for BCIT to make sure it is up to date even when the QUIC protocol gets a release candidate.

4.2.11 Development Schedule and Milestones

Below table details the time line which will be used to build the project:

ID	Task Name	Duration
1	*QUIC Java Implementation*: Set up basic structure	30 hours
2	*QUIC Java Implementation*: Set up QUIC handshake	30 hours
3	*QUIC Java Implementation*: Set up QUIC packet sending / receiving	30 hours
4	*QUIC Java Implementation*: Set up QUIC frame sending / receiving	30 hours

5	*QUIC Java Implementation*: Set up QUIC flow control	30 hours
6	*QUIC Java Implementation*: Set up full error handling	30 hours
7	*QUIC Java Implementation*: Regression / Manual Testing	30 hours
8	**Milestone 1: Finish the QUIC Java Implementation**	10 hours
9	*Java QUICSocket API*: Set up QUICSocket API	20 hours
10	*Java QUICSocket API*: Add multi-threading support	20 hours
11	*Java QUICSocket API*: Regression / Manual Testing	20 hours
12	**Milestone 2: Finish the QUICSocket API**	10 hours
13	*QUIC DFS*: Set up basic connection between client and server	5 hours
14	*QUIC DFS*: Set up sending GET and SEND commands and sending and getting files from server	10 hours
15	*QUIC DFS*: Set up sending DEL commands and deleting files from server	10 hours
16	*QUIC DFS*: Set up multi-threading so that multiple clients can connect and execute	10 hours
17	*QUIC DFS*: Acceptance Testing	10 hours
18	**Milestone 3: Finish the QUIC DFS**	5 hours
19	*Report*: Write the report	20 hours
20	*Report*: Revise the report	10 hours
21	**Milestone 4: Complete the Report**	5 hours
-	**TOTAL**	375 hours

Assuming 15 weeks worth of project work with 5 work days per week, 375 hours will add up to around **5 hours of major project work per work day**.

4.2.12 Deliverables

The deliverables in this project are:

- **QUIC Java Server/Client Implementation**: Implementation of QUIC Java Server/Client Implementation built on top of DatagramSocket API
- **QUICSocket Application Programming Interface (API)**: Implementation of QUICSocket API so that applications can be built on top of the QUIC Java Server/Client Implementation (this will be packaged with that implementation)
- **QUIC Distributed File System (DFS) Application**: Implementation of QUIC DFS Server/Client built on top of QUICSocket API
- **Report**: Major Project report

4.2.13 Conclusion and Expertise Development

Even in its narrowed scope, this project is still pretty ambitious as it is about implementing a new transport layer protocol that is still in draft. The student will get to learn more about implementing a modern transport layer protocol, including streams, frames, flow control, congestion control (possibly) and handshake which is perfect for letting the student gain more experience in network programming, his chosen specialization. This will also allow the student to gain knowledge of QUIC protocol in particular which, as mentioned in "Problem Statement and Background" section, is a new transport layer protocol that potentially could rival TCP in popularity as it is used as part of the upcoming HTTP/3 protocol which Chrome Browser already enables (Mulks). The Java QUIC implementation and API will also be open-sourced after the report is completed and approved by the committee and continued to be developed which could turn it into a full-fledged implementation that will be used by Java applications which could be very helpful to the software development community.

4.3 Project Supervisor Approvals

Student's supervisor for this project, Aman Abdulla, has approved both the project and the report. He told the student in a written email that he notified Michal Dziubek about it and that the student can submit this project formally to the committee. As the email from him is marked as confidential, it was not added here.

5 References

1. “QUIC: A UDP-Based Multiplexed and Secure Transport.” Edited by Janardhan Iyengar and Martin Thomson, *IETF Tools*, Internet Engineering Task Force (IETF), 20 Oct. 2020, tools.ietf.org/html/draft-ietf-quic-transport-34.
2. Kakhki, Arash Molavi, et al. “Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols.” *Communications of the ACM*, vol. 62, no. 7, July 2019, pp. 86–94. *EBSCOhost*, doi:10.1145/3330336.
3. Weiss, Todd R. “Google Unveils Quic Network Protocol, Dart Developer Tool.” *EWeek*, July 2013, p. 5. *EBSCOhost*, search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=a9h&AN=91658230.
4. “Quicwg/Base-Drafts.” *GitHub*, github.com/quicwg/base-drafts/wiki/Implementations. Accessed 29 Sept. 2020.
5. Joshi, James, and Prashant Krishnamurthy. “Network Security.” *Information Assurance*, 2008, pp. 19–38., doi:10.1016/b978-012373566-9.50004-5.
6. “TIOBE Index | TIOBE - The Software Quality Company.” *Tiobe.Com*, 2018, www.tiobe.com/tiobe-index/.
7. Mulks, Luke. “QUIC in the Wild.” *Brave Browser*, Brave Software, Inc., 28 Jan. 2020, brave.com/quic-in-the-wild/.
8. Ghedini, Alessandro. “HTTP/3: the Past, the Present, and the Future.” *The Cloudflare Blog*, Cloudflare, Inc., 4 Dec. 2019, blog.cloudflare.com/http3-the-past-present-and-future/.
9. Pandya, Vraj, and Stefan Andrei. “About the Design of QUIC Firefox Transport Protocol.” *BRAIN: Broad Research in Artificial Intelligence & Neuroscience*, vol. 8, no. 2, July 2017, pp. 26–32. *EBSCOhost*, search.ebscohost.com/login.aspx?direct=true&AuthType=ip,sso&db=a9h&AN=124762859.
10. “Mobile Vs. Desktop Usage (Latest 2019 Data).” *BroadbandSearch.Net*, 2019, www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics.
11. Pandya, Vraj, and Stefan Andrei. “TLS vs QUIC protocol stack”. Diagram. *BRAIN: Broad Research in Artificial Intelligence & Neuroscience*, vol. 8, no. 2, July 2017, pp. 27.
12. Postel, Jon. “User Datagram Protocol.” *IETF Tools*, Information Sciences Institute, 28 Aug. 1980, tools.ietf.org/html/rfc768.
13. “Using TLS to Secure QUIC.” Edited by Martin Thompson and Sean Turner, *IETF Tools*, Internet Engineering Task Force (IETF), 20 Oct. 2020, tools.ietf.org/html/draft-ietf-quic-tls-32.

14. “QUIC Loss Detection and Congestion Control.” Edited by Janardhan Iyengar and Ian Swett, *IETF Tools*, Internet Engineering Task Force (IETF), 20 Oct. 2020, tools.ietf.org/html/draft-ietf-quic-recovery-32.
15. Thompson, Martin. “Version-Independent Properties of QUIC.” *IETF Tools*, Internet Engineering Task Force (IETF), 25 Sept. 2020, tools.ietf.org/html/draft-ietf-quic-invariants-11.
16. “Socket (Java Platform SE 7)” *Oracle*, docs.oracle.com/javase/7/docs/api/java/net/Socket.html. Accessed 7 Nov. 2020.

6 Change Log

- **Revision 1** (April 23rd, 2021)
  - Updated based on committee feedback
    - Edited description for the System Architecture Diagram to introduce the diagram (Pages 10-11)
    - Added small clarification in JQUIC State Diagram general description about where to find more information about the different controllers (Page 16)
    - Added more description for all of the JQUIC State Diagrams (Pages 16-32)
    - Added description for the Class Diagram with small explanation for each class category (Pages 38-39)
- **Initial Version** (March 19th, 2021)