QUIC Protocol & Distributed File System Application: Java Implementation

COMP8037 – Major Project Proposal

KB - A00000000
November 9th, 2020

Table of Contents

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.

1.1 Education

1.2 Work Experience

He worked at - as a Junior Information Security Analyst from July 2017 to August 2019 where he had to investigate and re-mediate information security incidents. He also worked at BCIT for the period of May to June 2020 as a student research assistant for Aaron Hunter where he wrote code for learning belief revision operators.

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.

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:

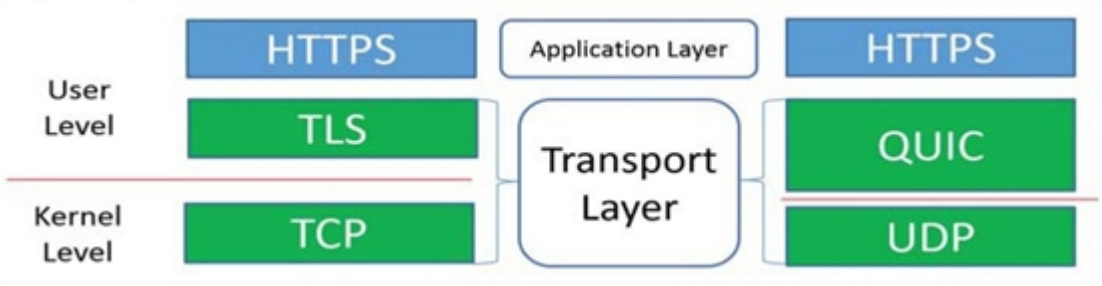

Figure 1: TLS vs QUIC protocol stack (Pandya et al. 27)

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.

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

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.

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 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.
"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."	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.

“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.”		
“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 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 1 keeps sending to Endpoint 2 until it reaches flow control limit set by Endpoint 2.	Endpoint 1 sends 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.

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 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 files to the DFS server at the same time

6 Methodology

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 2: 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 KB is the only developer for this project.

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.

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.

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:

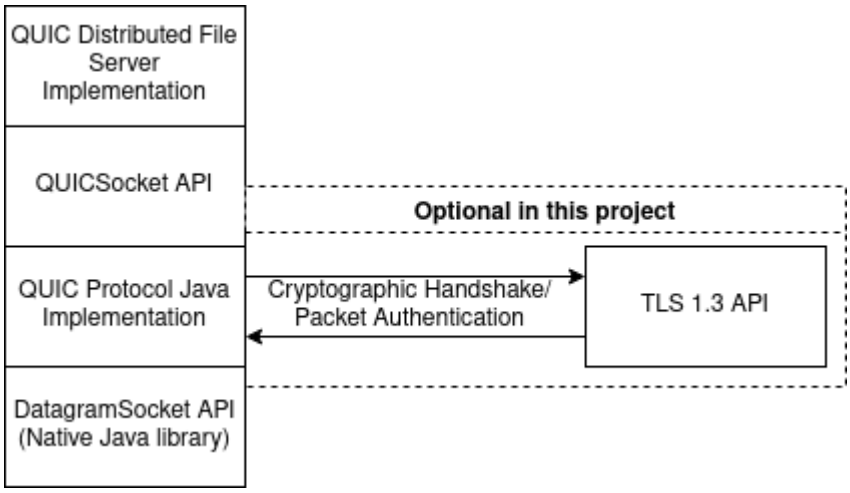

Figure 3: Implementation Stack

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:

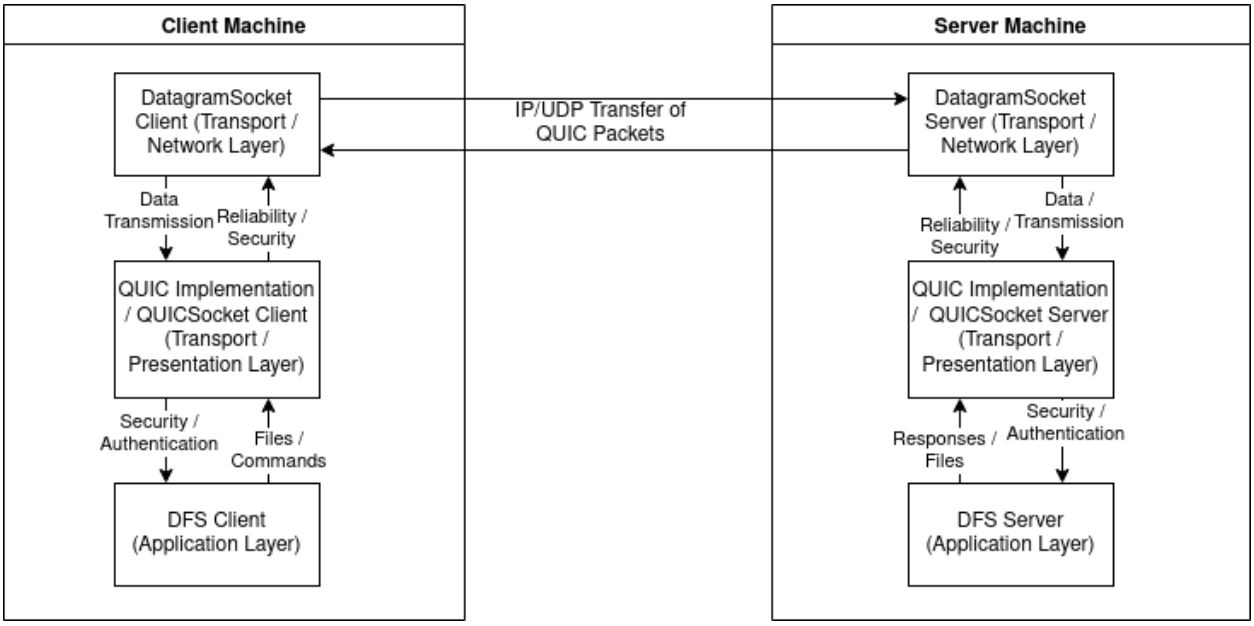

Figure 4: Client-Server Component Interaction

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:

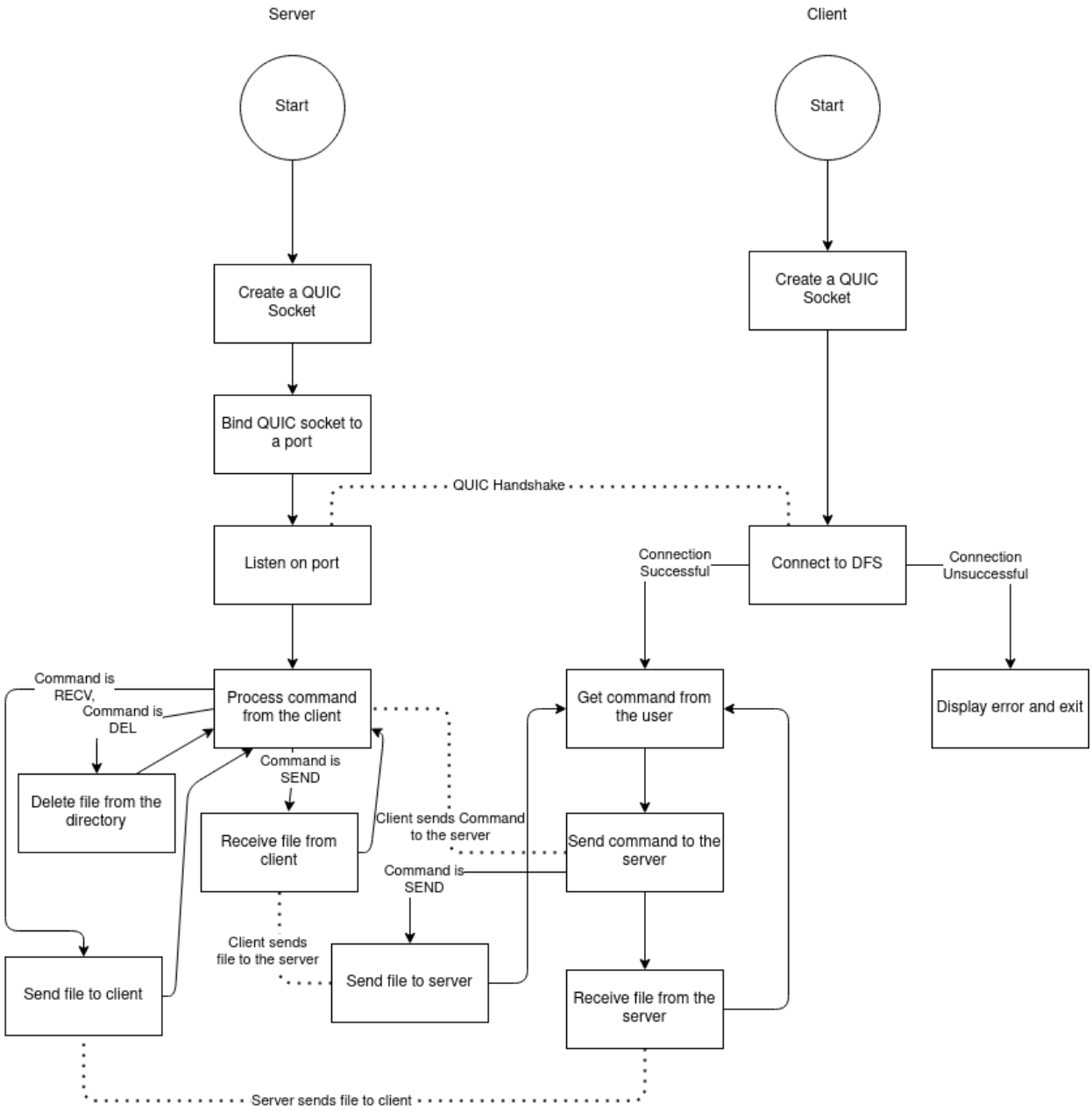

Figure 5: State Diagram

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.

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.

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.

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	10 hours

	sending and getting files from server	
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**.

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

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.

14 Works Cited

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-32.

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.

15 Change Log

- Initial Version (November 7th, 2020)