Building a Simple QUIC Protocol in Python

Made by: Shifaa Khatib , Bashar Nassir , Wassem Shebalny and Anthonios Deeb.

iDs: 324095702 , 212487144 , 323944280,324933993.

Introduction:

In this project, we implemented a simplified version of the QUIC (Quick UDP Internet Connection) protocol in Python. QUIC is an emerging transport protocol developed by Google that aims to improve web performance by reducing latency and improving security. Our project focuses on building a basic QUIC client and server to demonstrate key features of the protocol.

Overview

The project consists of three main components:

1. *QUIC Client*: Responsible for initiating connections to the server, sending requests, and receiving responses.

2. *QUIC Server*: Listens for incoming connections from clients, processes requests, and sends back responses.

3. *Packet Handling*: Defines classes and functions for encoding and decoding QUIC packets, managing sockets, and handling stream payloads.

QUIC Client:

- Establishes a connection to the server using QUIC.

- Sends requests for data over multiple streams.

- Receives responses from the server and handles stream data.

The create_socket method initializes the socket used for communication.

The run method simulates initiating file transfers through multiple streams, sends requests to the server, and handles responses.

The connect method establishes a connection with the server, including sending a client hello message and receiving a server hello message.

The send_packet method sends packets via the QUIC socket, and the receive_packet method retrieves packets from the socket.

The handle_response method is responsible for receiving files from the server, tracking stream data, sending acknowledgments (ACKs), and calculating the time taken to receive files.

Finally, **the printStatistics** method prints various statistics related to the received files, including the number of bytes and packets received in each stream, bandwidth of each stream, total bandwidth, and total number of packets received.

The code entry point ensures that if the script is executed directly, it initializes a QUIC client, connects it to a specified server, runs the client, and closes the socket connection afterward.

QUIC Server:

- Listens for incoming connections from clients.

- Accepts requests and processes them, generating random data for streams.

- Sends back responses containing stream data.

The QUICServer class initializes a server object with attributes like host, port, QUIC socket, a list to store active streams, and an index variable.

The create_socket method initializes the socket used for communication and binds it to the specified host and port.

The accept method waits for incoming connections, receives a client hello message, sends a server hello message in response, and establishes a connection.

The send_packet method sends packets via the QUIC socket to a specified address, and the receive_packet method retrieves packets from the socket.

The handle_client method manages client requests, generates random data (between 1MB and 5MB) for streams, and sends data in chunks via streams.

The generate_random_data method generates random data for each stream, and the send_data method divides the data into chunks and sends it via streams.

The code entry point ensures that if the script is executed directly, it initializes a QUIC server, accepts incoming connections, handles client requests, and closes the socket connection afterward.

Packet Handling:

- Defines classes for encoding and decoding QUIC packets.

- Manages sockets for communication between client and server.

- Handles stream payloads for transferring data over QUIC.

The generate_random_hex function generates a random hexadecimal string of 16 characters, which is utilized for generating connection IDs in the QUIC protocol.

The QUICPacket class represents a QUIC packet, storing attributes such as flags, destination connection ID, packet number, and protected payload. It includes methods for encoding and decoding packets and calculating the size of the packet in bytes.

The QUICStreamPayload class represents payload data transmitted over a QUIC stream, containing attributes like stream ID, offset, length, finished status, and stream data. It also provides methods for encoding and decoding stream payloads.

The QUICLongHeader class represents a QUIC long header, storing flags, destination connection ID, source connection ID, and packet number. It includes methods for encoding and decoding long headers.

The QUICSocket class encapsulates socket-related functionalities for QUIC communication, including socket creation, binding, sending, receiving, and closing. It also provides methods for setting and getting socket-related attributes like address and connection IDs.

The QUICAck class represents an acknowledgment packet in QUIC, storing acknowledgment number and acknowledgment delay. It includes methods for encoding and decoding acknowledgment packets.

Overall, this code serves as a foundational framework for implementing QUIC communication, providing essential functionalities for packet encoding/decoding, socket management, and acknowledgment handling.

Unit Testing:

We conducted unit testing to ensure the correctness and reliability of our implementation. Test cases were designed to cover various scenarios, including packet encoding and decoding, socket functionality, and stream payload handling.

For instance, TestQUICPacket tests the encoding and decoding of QUIC packets, ensuring that the decoded packets match the original ones. TestQUICLongHeader similarly checks the encoding and decoding of QUIC long headers.

TestQUICSocket validates the functionality of the QUIC socket class by testing methods for setting and getting socket attributes, as well as socket creation and closure.

TestQUICStreamPayload focuses on stream payload handling, confirming that encoded payloads can be decoded correctly and match the original payloads.

Finally, TestClientServerInteraction tests the interaction between a client and server by starting the server in a separate thread, creating a client, connecting it to the server, and running the client. This ensures that the client-server interaction functions as expected.

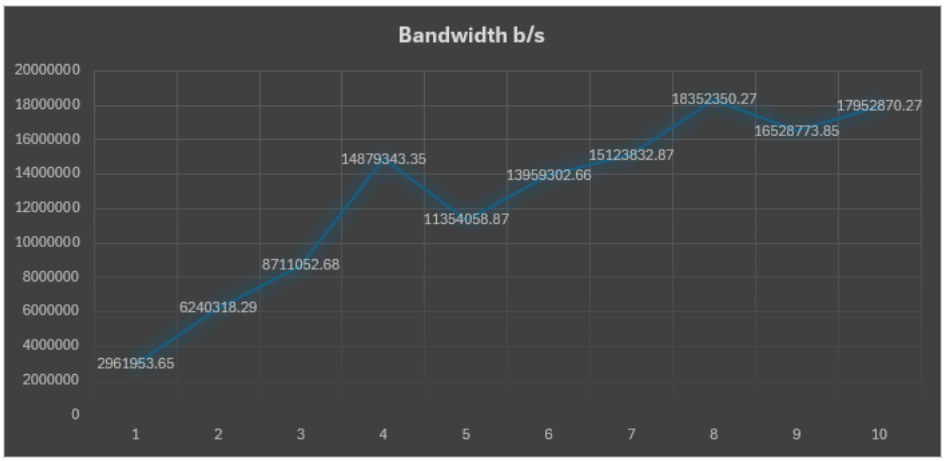
Overall, these unit tests provide comprehensive coverage of key functionalities of the QUIC implementation, aiding in the identification of potential issues and ensuring the robustness of the system.

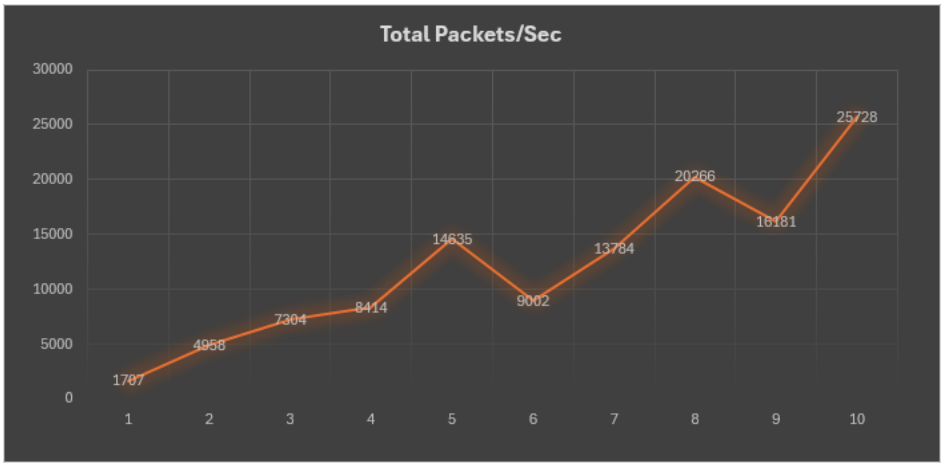
Conclusion;

By building a simplified QUIC protocol in Python, we gained a deeper understanding of network protocols and transport layer technologies. This project demonstrates the fundamental concepts of QUIC and provides a solid foundation for further exploration and development in this area.

After conducting ten iterations of running both the server and client, gradually increasing the number of streams by one with each iteration, and analyzing the gathered statistical data, we have concluded that, on average, both the bytes per second and the total packets received per second increase as the number of streams increases.

Here are the two garphs:





Wireshark Captures:

5	2.888655	127.0.0.1	127.0.0.1	UDP	161	54324 → 8888 Len=119
6	2.888784	127.0.0.1	127.0.0.1	UDP	177	8888 → 54324 Len=135
7	4.716897	127.0.0.1	127.0.0.1	UDP	277	54324 → 8888 Len=235
8	4.725540	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
9	4.725650	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70
10	4.725705	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
11	4.725739	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70
12	4.725784	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
13	4.725812	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70
14	4.725842	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
15	4.725869	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70
16	4.725897	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
17	4.725928	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70
18	4.725953	127.0.0.1	127.0.0.1	UDP	1485	8888 → 54324 Len=1443
19	4.726039	127.0.0.1	127.0.0.1	UDP	112	54324 → 8888 Len=70

Here we can see the start of a packet transaction between the Server and the Client. First three packets are the three-way-handshake packets. The next packet shows the data being sent from the Server to the Client, and its immediately followed by an ACK packet from the Client to the Server. The Server keeps sending the data in chunks until all of it has been sent.

5379	4.860539	127.0.0.1	127.0.0.1	UDP	113	54324 → 8888 Len=71
5380	4.860554	127.0.0.1	127.0.0.1	UDP	423	8888 → 54324 Len=381
5381	4.860575	127.0.0.1	127.0.0.1	UDP	113	54324 → 8888 Len=71

Once the Server finishes sending the data and the Client finishes receiving them, they close the connection, and we can see that by looking at the last 3 packets (FIN packets).

Note: pcap files link [Final Project pcaps]

Server-Client Run Example for 10 streams:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
anthonios@anthonios-virtual-machine:~/Desktop/Final_Project$ python3 Server.py
Server Running
Creating socket...

Accepting connection...

Received Client Hello from ('127.0.0.1', 42582)
Sent Server Hello to ('127.0.0.1', 42582)
Recieved request for 10 streams

Sending files...

Files sent.

anthonios@anthonios-virtual-machine:~/Desktop/Final_Project$
```

```
anthonios@anthonios-virtual-machine:~/Desktop/Final_Project$ python3 Client.py
QUIC client started

Connecting to server...

Creating socket...

Sent Client Hello

Waiting for Server response...

Received Server Hello

Enter the number of streams you want to simulate: 10
Sending Request to ('127.0.0.1', 8888)

Receiving files...

All files received.

Printing statistics...

Number of bytes received in each stream:
Stream 0 received 2873114 bytes
Stream 1 received 3308136 bytes
Stream 2 received 4505860 bytes
Stream 3 received 3400039 bytes
Stream 4 received 3922137 bytes
Stream 5 received 5000928 bytes
Stream 6 received 1933754 bytes
Stream 7 received 3022309 bytes
Stream 8 received 4022337 bytes
Stream 9 received 3235266 bytes
-----------------------------------------------------------------
Number of packets received in each stream:
Stream 0 received 2007 packets
Stream 1 received 2251 packets
Stream 2 received 2277 packets
Stream 3 received 1739 packets
Stream 4 received 2204 packets
Stream 5 received 4627 packets
Stream 6 received 1209 packets
Stream 7 received 1602 packets
Stream 8 received 3213 packets
Stream 9 received 1942 packets
-----------------------------------------------------------------
Bandwidth of each stream:
Stream 0 bandwidth: 5350256.11 B/s, 3737 packets per second
Stream 1 bandwidth: 6160342.74 B/s, 4191 packets per second
Stream 2 bandwidth: 8390712.26 B/s, 4240 packets per second
Stream 3 bandwidth: 6346374.42 B/s, 3238 packets per second
Stream 4 bandwidth: 7303713.03 B/s, 4104 packets per second
Stream 5 bandwidth: 9312608.61 B/s, 8615 packets per second
Stream 6 bandwidth: 3600990.49 B/s, 2251 packets per second
Stream 7 bandwidth: 5628069.09 B/s, 2983 packets per second
Stream 8 bandwidth: 7490293.18 B/s, 5983 packets per second
Stream 9 bandwidth: 6024629.68 B/s, 3617 packets per second
-----------------------------------------------------------------
Total bandwidth: 65607906.7 B/s
-----------------------------------------------------------------
Total packets received: 23071 packets per second
anthonios@anthonios-virtual-machine:~/Desktop/Final_Project$
```

חלק יבש:

שאלה 1: תארו במלים שלכם 5 חסרונות/מגבלות של.TCP

1- TCP משלב בקרת גודש ומנגנוני מסירה אמינים. עם זאת, צימוד זה יכול לגרום לבעיות. לדוגמה, אם חבילה אובדת, TCP מאט את השידור, גם אם מנות אחרות כבר התקבלו בהצלחה.

2-TCP מבטיח שהנתונים מועברים ברצף. אם חבילה אבדה, לא ניתן לשלוח מנות עוקבות עד שהחבילה שאבדה תשודר מחדש ותתקבל. המשמעות היא שמסירת הנתונים מתעכבת עד לשחזור החבילה שאבדה.

3-TCP דורש לחיצת יד תלת כיוונית כדי להגדיר חיבור לפני שהעברת נתונים יכולה להתחיל. בנוסף, אם מאבטחים את החיבור עם TLS, יש צורך בנסיעה הלוך ושוב נוספת כדי להחליף אישורי אבטחה. תהליך הגדרה זה מציג עיכוב לפני העברת נתונים בפועל.

4-ל TCP headers יש שדות בגודל קבוע, המגבילים את כמות הנתונים שהם יכולים לשאת. אפשרויות לפונקציות נוספות מוגבלות על ידי גודל מקסימלי של 40 בתים. מגבלה זו משפיעה על היכולת של TCP להסתגל לדרישות ולטכנולוגיות הרשת המתפתחות.

5- TCP מזהה חיבורים על סמך השילוב של כתובות IP ומספרי יציאות של נקודות הקצה. אם כתובת ה-IP של אחת מנקודות הקצה משתנה במהלך החיבור, מה שיכול לקרות עקב גורמים כמו ניידות מארח או NAT, החיבור הקיים נשבר, מה שמצריך לחיצת יד חדשה כדי ליצור חיבור חדש. זה גורם לאובדן נתונים ותקורה נוספת.

שאלה 2: ציינו 5 תפקידים שפרוטוקול תעבורה צריך למלא.

1- Defining Connection and Data Identifier : פרוטוקול התעבורה צריך ליצור מזהים ייחודיים עבור חיבורים בין נקודות קצה ועבור יחידות נתונים בודדות המוחלפות בתוך חיבורים אלה.

2- Transport Connection Management: לנהל את ההגדרה והפירוק של חיבורים, לשמור על מזהה חיבור ייחודי גלובלי תוך מתן גמישות להגדרת הודעות בקרה חדשות ותמיכה בשינויים בכתובות ה-IP המארח.

3- Reliable Data Delivery: הבטחת מסירה אמינה של יחידות נתונים תוך טיפול בבעיות כגון חסימת ראש קו על ידי הטמעת מנגנוני בקרת זרימת חלונות.

4- Congestion Control: וויסות מספר החבילות בתוך הרשת כדי למנוע עומס ולייעל את ביצועי הרשת.

5- Security: מתן מנגנונים לאבטחת חילופי נתונים, כולל הצפנה לסודיות נתונים ואימות לאימות הזהות והאמינות של הצדדים המתקשרים.

שאלה 3:

תארו את אופן פתיחת הקשר)"לחיצת ידיים"(ב .Quic-כיצד הוא משפר חלק מהחסרונות של TCP שתיארתם בסעיף 1?

ב-QUIC, החיבור נפתח באמצעות לחיצת יד משולבת של תחבורה וקריפטוגרפיה, מה שמפחית משמעותית את השהייה בהשוואה ל-TCP. כך פועלת לחיצת היד של QUIC וכיצד היא משתפרת עם החסרונות של TCP:

1- Combined Handshake Process:

- QUIC משלב את לחיצות יד ההעברה וההצפנה לזמן הלוך ושוב אחד(RTT), ומפחית את הזמן הנדרש ליצירת חיבור מאובטח.

- על ידי שילוב האימות של חילופי מפתחות TLS עם החלפת פרמטרי תעבורה, QUIC מבטל את הצורך בסבבי תקשורת נפרדים, בניגוד ל-TCP.

2-Connection ID Negotiation: תהליך לחיצת היד כולל משא ומתן על מזהי חיבור, המשמשים כמזהים ייחודיים לחיבור. משא ומתן זה מבטיח הגדרה אמינה של חיבור חדש בדומה ל-TCP.

3- RTT Data Support: - QUIC מאפשר ללקוחות לשלוח נתוני יישומים מוצפנים ב-0-RTT בחבילה הראשונה לשרת, תוך שימוש חוזר בפרמטרים שנקבעו מחיבור קודם ובזהות מפתח משותף מראש של TLS 1.3 (PSK).

- תכונה זו מאפשרת העברת נתונים מהירה יותר, שימושית במיוחד עבור תרחישים הדורשים תקשורת מהירה, מבלי לחכות להשלמת תהליך לחיצת היד המלא.

4- Address Validation and Security:

- לחלופין, שרת יכול לאמת כתובת של לקוח על ידי שליחת חבילת נסה מחדש המכילה אסימון אקראי, מה שמשפר את האבטחה במהלך תהליך לחיצת היד.

- הודעות לחיצת יד TLS 1.3, כולל יצירת סוד משותף, מוטמעות בחבילות ראשוניות, מה שמבטיח סודיות ואותנטיות של מנות עתידיות ב-1-RTT.

5-Improved Latency and Efficiency:

- על ידי שילוב של תחבורה ולחיצות יד קרטוגרפיות, QUIC מפחית משמעותית את ההשהיה הנדרשת ליצירת חיבור מאובטח בהשוואה ל-TCP, הדורש לפחות 2 RTTs.

- תהליך לחיצת היד היעיל הזה משפר את היעילות וההיענות הכללית, מועיל במיוחד עבור יישומים בזמן אמת ושיפור חווית המשתמש.

שאלה 4:

תארו בקצרה את מבנה החבילה של .QUIC כיצד הוא משפר חלק מהחסרונות של TCP שתיארתם בסעיף 1?

מבנה החבילה של QUIC שונה מ-TCP בכמה דרכים, ומתייחס לכמה מהחסרונות של TCP:

1. Flexible Packet Header Format:

- ל-QUIC שני סוגים של כותרות מנות: כותרת ארוכה ליצירת חיבור וכותרת קצרה למנות עוקבות לאחר לחיצת היד.

- גמישות זו מאפשרת ל-QUIC לייעל את גודל המנות, ולשפר את היעילות בהשוואה לפורמט הכותרת הקבועה של TCP.

2. Unique Packet Numbering:

- לכל חבילה בחיבור QUIC מוקצה מספר מנות ייחודי, אשר גדל באופן מונוטוני, המציין את סדר השידור של מנות.

- בניגוד ל-TCP, שבו מספור מנות משולב עם שחזור אובדן, מספור המנות המנותק של QUIC מקל על מעקב מדויק יותר אחר מנות ומפשט את מנגנוני שחזור האובדן.

3. Improved Acknowledgment Mechanism:

- מקלטי QUIC מאשרים מנות באמצעות מסגרות ACK, הכוללות את מספר החבילות הגדול ביותר שהתקבלו ומאשרים באופן סלקטיבי מנות שהתקבלו.

- על ידי תמיכה בעד 256 בלוקים של ACK במסגרת אחת, QUIC משפר את העמידות בפני סידור מחדש של מנות והפסדים בהשוואה לטווחי SACK המוגבלים של TCP.

4. Stream Multiplexing and Head-of-Line Blocking Mitigation:

- QUIC משלב ריבוי זרמים בשכבת התחבורה, בדומה ל-HTTP/2, המאפשר זרימות מרובות בו זמנית בתוך חיבור יחיד.

- תכונה זו מפחיתה חסימת ראש קו, כאשר אובדן מנות בזרם אחד יכול לחסום מסירת נתונים בזרמים אחרים, ולשפר את יעילות העברת הנתונים הכוללת.

שאלה 5: מה QUIC עושה כאשר חבילות מגיעות באיחור או לא מגיעות כלל?

כאשר מנות מגיעות מאוחר או אינן מגיעות כלל ב-QUIC, הפרוטוקול משתמש במנגנונים לאיתור אובדן ושחזור:

1- ACK-Based Loss Detection:

- QUIC מסתמך על אישורים (ACKs) מהמקלט כדי לזהות אובדן מנות. כאשר ACK מתקבל, כל המסגרות הנישאות בחבילה המאושרת נחשבות שהתקבלו. אם חבילה אינה מאושרת כאשר מזוהה חבילה שנשלחה מאוחר יותר, ומתקיימים ערכי סף מסוימים, QUIC מחשיב את הframes בחבילה זו לאיבוד.

2- Threshold-Based Loss Detection:

- QUIC משתמש בשני סוגים של ספים לקביעת אובדן מנות:

- מבוסס מספר מנות: אם מספר הרצף של חבילה קטן מהחבילה המאושרת במספר מסוים (נקבע על פי סף הסדר מחדש של מנות), היא מוכרזת כאבדה.

- מבוסס זמן: אם חבילה נשלחה לפחות זמן מסוים לפני החבילה המאושרת, שנקבעה על פי סף זמן ההמתנה, היא מוכרזת אבודה.

3- Probe Timeout (PTO) Mechanism:

- כדי לזהות אובדן של חבילות, QUIC מאתחל טיימר לתקופת פסק זמן הבדיקה (PTO) בכל פעם שנשלחת חבילה הדורשת אישור. כאשר טיימר ה-PTO פג, השולח שולח חבילה חדשה הדורשת אישור כבדיקה, אשר עשויה לכלול שידור חוזר של כמה נתונים כדי להפחית את מספר השידורים החוזרים.

4- Loss Recovery:

- לאחר זיהוי אובדן חבילות, QUIC מכניס את החבילות האבודים למנות יוצאות חדשות, אשר מוקצים להן מספרי מנות חדשים. מנות אלו עוברות מנגנוני שחזור אובדן כדי להבטיח אספקה מהימנה של זרם בתים מסודר, בדומה ל-TCP.

מנגנוני זיהוי האובדן והשחזור של QUIC מאפשרים לו לשמור על העברת נתונים אמינה גם מול אובדן מנות או עיכוב, מה שמבטיח תקשורת חזקה ויעילה על גבי רשתות לא אמינות.

שאלה 6: תארו את בקרת העומס(control congestion) שלQUIC

מנגנון בקרת העומס של QUIC נועד להבטיח העברת נתונים יעילה תוך הימנעות מגודש ברשת. להלן סקירה כללית של האופן שבו QUIC מטפל בבקרת עומס:

1- Decoupling of Congestion Control from Reliability Control:

QUIC מפריד בין בקרת עומס לבקרת אמינות. ניתוק זה מאפשר ל-QUIC לנהל עומס באופן עצמאי מהבטחת אספקת נתונים אמינה.

2- Utilization of Window-Based Congestion Control:

- QUIC משתמש בתוכנית בקרת עומס מבוססת חלון, בדומה ל-TCP. סכימה זו מגבילה את המספר המרבי של בתים שיכולים להיות לשולח במעבר בכל זמן נתון.

3- Persistent Congestion Detection:

- QUIC מזהה עומד מתמשך כדי למנוע הפחתת חלון עומס מיותר. אם שתי מנות הדורשות אישור מוכרזות כאבדות ואף אחת מהמנות שנשלחות ביניהן לא מאושרת, נוצר עומס מתמשך.

4- Sender Pacing:

- שולחי QUIC מקדמים את השליחה שלהם כדי להפחית את הסבירות לגרימת עומס לטווח קצר. הם מבטיחים שהמרווח בין שליחת מנות חורג ממגבלה המחושבת על סמך גורמים כגון ה-RTT הממוצע (smoothed_rtt), גודל חלון העומס וגודל החבילה. קצב זה עוזר לווסת את קצב העברת הנתונים ולמנוע עומס.

בסך הכל, מנגנוני בקרת העומס של QUIC שואפים לשמור על יציבות הרשת, לייעל את התפוקה ולספק נתח הוגן מרוחב הפס(Bandwith) לכל החיבורים תוך התאמה לתנאי הרשת המשתנים.