פרויקט גמר
QUIC

קורס רשתות תקשורת
תשפ"ד

מדעי המחשב, אוניברסיטת אריאל

ליאור טרכטמן

רועי שמילוביץ'

עדיאל הלוי

סמואל לזריאנו

תוכן עניינים

אם משהו לא טוב בכיווץ, הכל הועלה גם לגיט בקישור הבא:

https://github.com/SamuraiPolix/Networking-Final-Project.git

וההקלטות של הWireshark למקרה ולא יצליח לעלות למודל, יהיו או בדרייב או בגיט למעלה:

https://drive.google.com/file/d/1nLo7O6AFyo8N9iJpn2I_OwAmxZ_Pi890/view?usp=sharing

חלק "יבש"

שאלה 1: 5 חסרונות/מגבלות של TCP

(התשובה שלנו לשאלה מבוססת על המאמר A Quick Look at QUIC בסעיף TCP's 2.2 Problems)

TCP - Transmission Control Protocol הוא פרוטוקול תקשורת אמין ונפוץ, אך יש לו כמה חסרונות ומגבלות:

1. **הצימוד הבעייתי בין מנגנון בקרת הגודש (CC) שלו לבין מסירה אמינה:** החיסרון בא לידי ביטוי בצימוד כיוון שזה מאלץ את TCP להשהות את התקדמות חלון הגודש שלו כאשר מתרחש אובדן מנות, שכן הפרוטוקול ממתין לשידור מחדש של החבילה האבודה ואישורה. במהלך תקופה זו, למרות שייתכן כי מנות אחרות כבר יצאו מהרשת ונמסרו למקלט, החלון נותר עומד על כנו, ומונע שידור של מנות חדשות. זה גורם לשימוש לא יעיל במשאבי הרשת, מכיוון שמספר החבילות הלא מאושרות כבר לא משקף במדויק את הקיבולת של הרשת, מה שמוביל לעיכובים פוטנציאליים ולתפוקה מופחתת.
2. **חסימת ראש התור (HLB):** ל-TCP חשוב לשמור על שליחת נתונים לפי סדר אז אם חבילה בודדת נאבדת, כל החבילות הבאות נחסמות מלהימסר לאפליקציה עד שהחבילה החסרה תועבר בהצלחה מחדש. זה יוצר חתיכת פקק וגורם לאפליקציה להמתין לחבילה שאבדה, גם אם מנות אחרות כבר הגיעו, היא פשוט לאת קבל אותם והן יצטרכו להישלח שוב פעם. זה משבש את זרימת הנתונים, מה שמוביל להפחתת הביצועים ולהגברת השהייה בתקשורת.
3. **הקמת חיבור ארוכה:** תהליך לחיצת היד של TCP דיי ארוך, עם 3 לחיצות ידיים לפני שאפשר להתחיל להעביר מידע כלשהו. מעבר לזה, אם צריכים לאבטח את החיבור באמצעות TLS, צריכים לעשות עוד דרך הלוך חזור כדי להחליף אישורי אבטחה ומפתחות, מה שרק מוסיף עוד תהליך (בהתחשב גם בפאקטות הקמת חיבור שנאבדות בדרך). כל הסיפור הזה יכול להוביל לזמני המתנה משמעותיים לפני שבכלל התחלנו להעביר נתונים, שלא לדבר על רשת לא יציבה עם הרבה לקוחות ועומס.
4. **גודל Header קבוע:** הגודל הקבוע של שדות הכותרות של TCP הוא בעיה בכמה מקרים. אחד, הוא פשוט תופס הרבה מקום קריטי שיכול להתאים לPayload במקום לנתוני חיבור שאפשר פשוט לזכור בהתחלה וזהו. שנית, שדות מספר הרצף SEQ ושדות האישור ACK מוגבלים כל אחד ל-4 בתים, בזמן שגודל חלון בקרת הזרימה מוגבל ל-2 בתים בלבד. העובדה שהשדות האלה קטנים ושהגודל שלהם קבוע הוא בעייתי כי ברגע שמגיעים לרשת מאוד מהירה, ככה שהמהירות תעלה, יש מצב שנגיע ללמחסור בSEQ וגודל חלון בקרת זרימה יהיה מוגבל. זה מגביל את ביצועי הTCP ברשתות ממש מהירות.
5. **כתובת IP משתנה:** TCP רגיש לשינויים בכתובות IP במהלך חיי החיבור כי הוא מסתמך על שילוב של כתובות IP ומספרי יציאות כדי לזהות חיבורים באופן ייחודי ואז ברגע שכתובת IP משתנה (לא תמיד באשמת הלקוח), החיבור נשבר, ואז לא רק שמאבדים את המידע שהיינו באמצע להעביר וצריכים לחזור עד לאותה הנקודה שוב פעם, אלא גם צריכים ליצור מחדש את התקשורת, להגדיר חיבור חדש, שוב פעם עם 3 לחיצות יד. זה חסרון משמעותי במיוחד בסביבות רשת דינמיות.

שאלה 2: 5 תפקידים שפרוטוקול תעבורה צריך למלא

פרוטוקול תעבורה ממלא תפקידים חשובים בניהול תקשורת בין מחשבים ברשת. להלן חמישה תפקידים מרכזיים שפרוטוקול תעבורה צריך למלא:

1. **העברת נתונים אמינה**: פרוטוקול תעבורה לצריך לדאוג שכל המידע שנשלח יגיע כמו שצריך ליעד שלו, כולל תיקון בעיות כמו אובדן מידע ותקלות.
2. **ניהול קצבי שליחה (Flow Control)**: הוא שולט על קצב שליחת המידע בין המחשבים, כדי שלא יווצר עומס על המחשב המקבל, במיוחד אם הוא לא יכול לעמוד בקצב המשלוח.
3. **הבטחת סדר הנתונים**: במקרים בהם הסדר חשוב, הפרוטוקול מבטיח שהמידע יגיע לפי הסדר שבו נשלח, ואם יש בעיה, הוא מסדר את זה.
4. **ניהול הקמת וסיום חיבורים (Connection Management)**: הפרוטוקול אחראי על פתיחה וסגירה של חיבורים בין מחשבים, וגם על התחזוקה שלהם בזמן העברת המידע.
5. **בקרת עומס (Congestion Control)**: פרוטוקול תעבורה צריך להיות מסוגל לזהות עומסים ברשת ולהתאים את קצב העברת הנתונים בהתאם, כדי למנוע מצב של עומס יתר, שיכול להוביל לאובדן חבילות ולירידה בביצועים הכוללים של הרשת.

שאלה 3: אופן פתיחת הקשר ב-Quic וכיצד הוא משפר חלק מהחסרונות של TCP

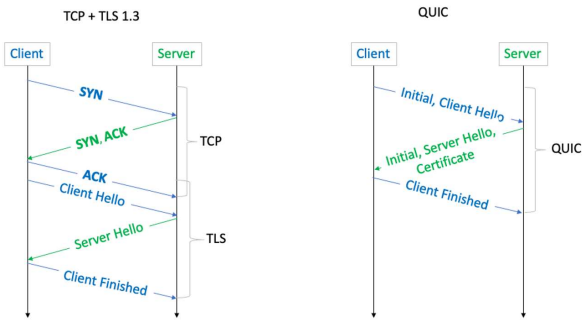
(התשובה שלנו לשאלה מבוססת על המאמר [A Quick Look at QUIC](#) בסעיף 3.2 QUIC Handshake)

תהליך לחיצת היד ב-QUIC חוסכת שלבים של TCP על ידי שילוב בקשת החיבור והחלפת מפתחות TLS 1.3 בבת אחת. התהליך מתחיל בזה שהלקוח שולח חבילה ראשונית הכוללת את מזהה החיבור שהוא בחר לעצמו (Source Connection ID), הודעת לחיצת יד (מכירים כSYN בTCP), יחד עם גרסה נתמכת של הפרוטוקול ומידע על ההגדרות שלו ומפתח מוצפן TLS 1.3. אחרי החבילה הזאת השרת בעצם גם קובע Connection ID משלו (יתכן שיהיו כמה כאלה לאורך החיבור) ומגדיר פרמטרים של תקשורת מאובטחת. השרת מגיב עם חבילה (מכירים כSYN ACK בTCP) שכוללת גם את מזהה חיבור היעד שלו. יכול להיות שהשרת ינפיק חבילת בדיקה ללקוח שתכין קוד מסוים כדי לאמת את כתובת הלקוח - הלקוח חייב לכהכליל את הקוד הזה בחבילה ראשונה חדשה שהוא ישלח כדי להמשיך בלחיצת יד. ככה בחצי מהחבילות של TCP, גם יצרנו חיבור וגם מפתח TLS משותף לאבטחת החיבור. יתרון חזק של QUIC בשימוש ב"מזהה חיבור" הוא התמיכה בRTT-0, שמאפשרת ללקוח לשלוח נתונים מוצפנים בלי צורך להקים שוב חיבור, על ידי שימוש שימוש בפרמטרים שנקבעו לפני זה, אם כי נתונים אלו חשופים להתקפות חוזרות.

תהליך פתיחת הקשר הזה ב-QUIC מטפל במספר חסרונות של TCP:

1. **עיכוב עקב הקמת חיבור:** QUIC משפר את עיכוב הגדרת החיבור של TCP על ידי שילוב של תהליך לחיצת היד עם החלפת מפתחות TLS 1.3 בסיבוב אחד, כפי שהסברנו למעלה - זה *מקטין את ההמתנה* בהקמת הקשר.
2. **מתאים יותר לשימוש בזמן אמת:** מעבר לכך שזמן הקמת החיבור נחסך (נקודה קודמת), גם ברגע שהוקם קשר, למשך זמן מסוים, *אין צורך להקים שוב פעם קשר*. אלא השרת זוכר את מזהה החיבור של הלקוח ואת הקוד המוצפן שלהם אז הלקוח יכול לשלוח מידע ישר מבלי להקים שוב חיבור.
3. **פחות עומס על הרשת:** שוב, בזמן שTCP דורש שלוש לחיצות יד כדי להקים חיבור, QUIC מקצר את התהליך ל-2 לחיצות ידיים בלבד, ובמקרים מסוימים (לדוגמא חיבור חוזר בין לקוח ללקוח) אפשר להתחיל לשלוח נתונים כבר בזמן פתיחת הקשר, מה *שמפחית את העומס הכללי*, בעיקר כשמדובר בשרת שהרבה לקוחות פונים אליו ובתדירות גבוהה.\
4. **קשר שלא תלוי בשינוי ה-IP של הלקוח:** QUIC מתגבר על הבעיה של TCP עם שינויים בכתובת IP על ידי זה שהוא משתמש בבמזהה חיבור (בלחיצת היד) שנשאר קבוע גם אם כתובת ה-IP הבסיסית משתנה. זה מאפשר לחיבורי QUIC להימשך גם אחרי ותוך כדי שינוי כתובת, בלי להרוס ולשבש העברת נתונים.

[1]TCP and QUIC handshakes



[1] Figure 2 in https://blog.apnic.net/2022/11/03/comparing-tcp-and-quic/

שאלה 4: מבנה החבילה של QUIC וכיצד הוא משפר חלק מהחסרונות של TCP

(התשובה שלנו לשאלה מבוססת על המאמר A Quick Look at QUIC בסעיף QUIC 4 PACKET)

QUIC נועד להיות גמיש, יעיל ומאובטח, והוא מבוסס על UDP על מתי להיות יותר מהיר וגמיש. כל חבילה בפרוטוקול QUIC מכילה כמה חלקים חשובים שהגיעו כדי לייעל את את היתרונות של TCP ולתקן חסרונות שלו.

1. כותרת Header:

מספר גרסה: מציין את גרסת QUIC בשימוש, מה שמאפשר לעדכן את הפרוטוקול בקלות.

מזהה חיבור: מספר ייחודי לכל חיבור, שמאפשר לנתב את החבילות נכון גם כשהכתובת של המכשיר משתנה (למשל, כשעוברים בין רשתות).

סוג החבילה: מציין את סוג החבילה (כמו Handshake), כדי לנהל נכון את שלבי החיבור

*ל-QUIC יש 2 סוגי Header-ים:

- ראש חבילה ארוך: משמש ליצירת חיבורים (נותנים בלחיצת היד את כל המידע שצריכים ואז כל צד שומר את המידע הזה אצלו תוך כדי ואין צורך להעביר אותו שוב בכל פאקטה שיוצאת)
- ראש חבילה קצר: עבור חבילות מידע שלא קשורות ללחיצת יד, מאפשר יותר מקום בחבילה להעברת מידע שלא קשור בהכרחת לשמירת החיבור.

כל חבילה ב-QUIC כוללת Frame אחד או יותר, לא כל הFrame-ים בהכרח מאותו הסוג, כל עוד הן מתאימות בגודל החבילה.

2. גוף החבילה Payload:

מספר רצף Seq: מספר שמאפשר לנמען לדעת את סדר החבילות ולסדר אותן נכון, גם אם הן מגיעות בסדר שגוי.

Frame-ים: יחידות נתונים קטנות בתוך החבילה, כל אחת יכולה לייצג סוג שונה של מידע (כמו נתוני אפליקציה או אישור קבלה).

מידע קריפטוגרפי: החבילה כולה מוצפנת כדי לשמור על סודיות ושלמות המידע

3. שדות נוספים

checksum: לבדוק שהחבילה תקינה ולהגן מפני שגיאות.

שדות אופציונליים: תוספות אפשריות לניהול מתקדם יותר של החיבור.

יתרונות לעומת TCP:

1. בניגוד ל-TCP שמשתמשת בHeader בגודל קבוע, QUIC מקצה לכל מנה את הגודל המתאים לפי איזו סוג חבילה היא (ראש גדול או קטן) ובכך מאפשר ניצול טוב יותר של רוחב הפס ופחות "בזבוז" מקום ונתונים על דברים שלא צריכים להעביר כל פעם מחדש.
2. בניגוד ל-TCP שמחכה לאישורי ACK על מנות שנשלחו, QUIC לא רק שמשאפשר לשלוח בכמה זרמים במקביל (ושומר Stream ID בHeader), אלא הוא גם מאשר מנות על ידי דיווח על מספר החבילות הגדול ביותר שהתקבל או/ו שימוש באישור ACK סלקטיבי ב Header כדי לכסות את כל החבילות הקודמות. אז הוא בעצם לא צריך לשלוח ACK לכל חבילה, יש לQUIC תמיכה בעד 256 בלוקים של ACK

למסגרת בהשוואה לשלושת טווחי האישור הסלקטיביים של TCP. מגנון הACK הזה משפר יפה מאוד את היכולת של QUIC לטפל בסידור מחדש של מנות ואובדן ביעילות ולחסוך זמני המתנה.

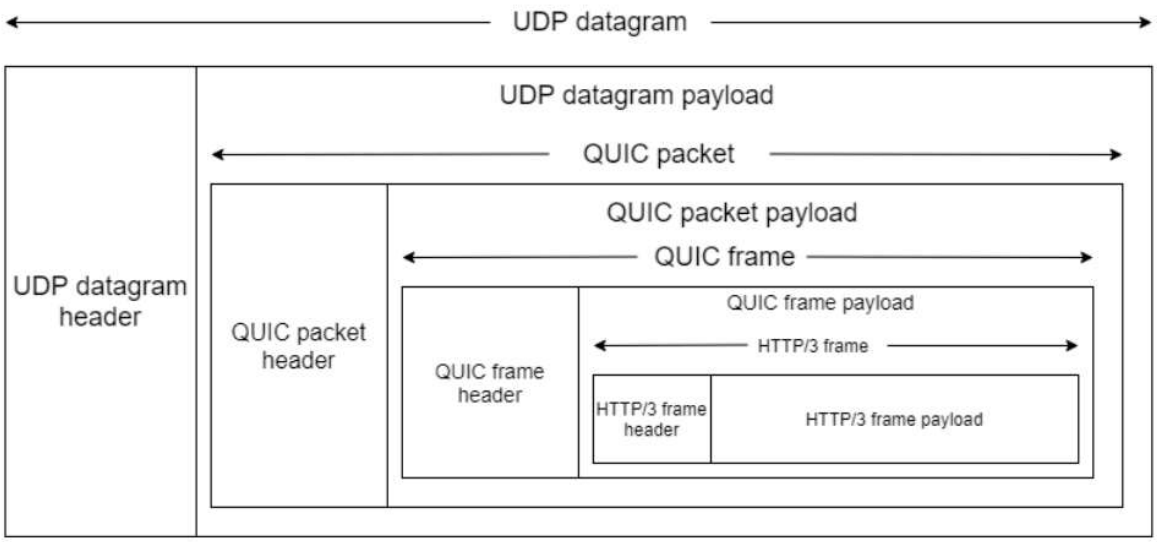
3. אבטחה משולבת:
   - כל חבילה מוצפנת לחלוטין, מה שנותן אבטחה טובה יותר בלי צורך בשכבות נוספות כמו TLS.
4. גמישות לעתיד:
   - מבנה החבילה של QUIC מאפשר להוסיף תכונות חדשות בלי לפגוע בגרסאות הקודמות, כך שהפרוטוקול יכול להתפתח ולהשתפר עם הזמן.

הסבר על ה Connection ID בחבילה ואיך השתמשנו בו במימוש שלנו:

לפי 3.1 במאמר A Quick Look at QUIC ולפי 5.1.1 ב RFC 9000, בחבילה אמורים להיות Source ID ו Destination ID. מי שיוזם את לחיצת היד (הלקוח), מתחיל עם Connection ID = 0 והוא נכנס ל Source ID בחבילה, השרת שמקבל את החבילה, קובע את ה Connection ID שלו להיות 1.

מאחר ואנחנו לא נדרשים לממש עוד תכונות של QUIC, כמו ריבוי משתמשים, מעבר למה שבחרנו בחלק הרטוב, החלטנו להשאיר את שני ה Connection ID, אבל תכלס הם תמיד 0 (לקוח) או 1 (שרת) והקדשנו בייט יחיד לאורך שלהם. אם היינו נדרשים לממש חיבור של מספר משתמשים במקביל, היינו מגדילים את הconnection id length של שניהם ומאפשרים עוד מספרי זיהוי, יוצרים בשרת Dict שמקשר לכל כתובת ופורט, מספר/י Connection ID (ללקוח אחד יכולים להיות כמה מספרים). ברגע שהשרת היה מקבל פאקטת Initial & Client Hello - SYN, הוא היה בודק במילון אם הID כבר תפוס, אם כן, הוא היה שולח פאקטת פקודה RETIRE_CONNECTION_ID ולא ממשיך את לחיצת היד. יכולנו גם להגריל מספר ID רנדומלי לכל צד, אך העדפנו להידבק ל RFC 9000 ולא להיכנס לעומק יותר מהנדרש.

כל החבילה הכוללת:



[2] QUIC inside UDP structure

[2] https://docs.netscaler.com/en-us/citrix-adc/current-release/system/http3-over-quic-protocol.html
Long and short headers: https://datatracker.ietf.org/doc/html/rfc8999

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

ראינו בשאלה 2 את התכונות שצריכות להיות לפרוטוקול תעבורה, אז ל-QUIC, כמו בכל פרוטוקול תעבורה אמין, קיימים מנגנונים לטיפול במצבים שבהם חבילות מגיעות באיחור או לא מגיעות בכלל. אם נשווה את QUIC לפרוטוקולים אחרים כמו TCP אנחנו נראה גם שהוא עושה זאת בצורה יעילה וגמישה הרבה יותר.

מה QUIC עושה כאשר חבילות מגיעות באיחור או לא מגיעות כלל:

1. **מנגנון אוטומטי של משלוח מחדש**:
   - אם חבילה לא מגיעה ליעד בתוך זמן סביר, QUIC שולח מחדש את החבילה. זה עובד על מספרי רצף (Packet Numbers), או SEQ כפי שמכירים מפרוטוקלים אחרים, המאפשרים למערכת לזהות חבילות חסרות ולשלוח אותן מחדש.
2. **מצטבר ודינמי (Acknowledgment (ACK**:
   - QUIC משתמש בפריימים מיוחדים הנקראים ACK Frames כדי להודיע על קבלת חבילות. אם הנמען מזהה חבילה חסרה, הוא שולח ACK Frame המציין את הטווחים של החבילות שהתקבלו ואת אלו שלא.
   - אם חבילה מגיעה באיחור, QUIC מסדר אותה ביחד עם החבילות האחרות שכבר הגיעו. זה עוזר למנוע מצב שבו כל התקשורת נעצרת בגלל חבילה אחת שהתעכבה. בנוסף, אם חבילה נשלחת מחדש ומגיעה באיחור יחד עם החבילה המקורית, QUIC מזהה את הכפילות ומתעלם ממנה כדי לא לשלוח את הנתונים שוב ושוב.
3. **ריבוי זרמים (Multiplexing Streams)**:
   - ב-QUIC, כל חיבור יכול להכיל מספר זרמים במקביל (flows). אם חבילה באחד מהזרמים לא מגיעה או מתעכבת, זה לא משפיע על הזרמים האחרים, מה שמונע את בעיית ה-Head-of-Line Blocking שקיימת ב-TCP. ככה, העיכוב או איבוד של חבילות מסוימות לא מעכב את כל תהליך התקשורת, מה שמשפר את ביצועי הרשת.
4. **ניהול זמן השהיה (RTT) אדפטיבי**:
   - QUIC משתמש בטיימרים כדי לבדוק אם יש בעיות בקצב ההעברה או בחיבור. אם חבילה לא מגיעה בזמן שצריך QUIC יכול לשלוח אותה מחדש או לעשות התאמות אחרות כדי לשמור על חיבור תקין.
5. **מנגנוני בקרה ובקרת עומס (Congestion Control)**:
   - כשQUIC מזהה שיש חבילות שאבדו, הוא יכול להוריד את קצב ההעברה כדי למנוע הצפה של הרשת ולעזור לחבילות הבאות להגיע בשלום. אחרי שהבעיה נפתרת והחבילות מתחילות להגיע כמו שצריך QUIC מעלה שוב את הקצב. זה עוזר לשמור על קצב העברה גבוה גם בתנאים של רשתות לא יציבות ולדאוג לא להעמיס יותר מדי.

שאלה 6: בקרת העומס (Congestion Control) של QUIC.

בקרת העומס (Congestion Control) של QUIC מיועדת לנהל את קצב שליחת הנתונים ברשת בצורה חכמה, כדי למנוע עומסים, לשפר את ביצועי הרשת, ולהבטיח שהנתונים יגיעו בצורה אמינה.

איך זה עובד:

1. **מבוסס על TCP Cubic**:
   - QUIC משתמש באלגוריתם שנקרא Cubic TCP, שמוכר ומשומש גם ב-TCP. האלגוריתם עוזר לקבוע את קצב שליחת הנתונים לפי מצב הרשת.
   - Cubic TCP עובד בצורה של "ניסוי וטעיה": הוא מגביר בהדרגה את קצב השליחה עד שמתגלה עומס,ואז מוריד את הקצב ומתחיל שוב את התהליך.
2. **ניהול מופרד של זרמים**:
   - בכל חיבור של QUIC אפשר לנהל כמה זרמים במקביל. כל זרם מנוהל בנפרד מבחינת עומס, מה שמונע בעיות כמו עיכובים במשלוח נתונים (Head of line blocking) ונותן שליטה מדויקת יותר בקצב השליחה
3. **תגובה מהירה לשינויים**:
   - בניגוד ל-TCP, שבו התגובה לעומס יכולה להיות איטית, QUIC מגיב מהר יותר לשינויים בתנאי הרשת. הוא עושה זאת על ידי מדידת זמני השהיה RTT ושימוש במנגנונים משופרים שמאפשרים לזהות עומס ולהגיב לו במהירות.
4. **איסוף נתונים ומדידת RTT**:
   - QUIC אוסף נתונים על זמני השהיה ועל תגובות מהנמען בכל שליחה של חבילה ובכל קבלה של ACK, כך שהוא יודע כל הזמן מה מצב הרשת ויכול להתאים את עצמו בהתאם.
5. **מנגנון שליטה בקצב (Rate Control)**:
   - QUIC קובע כמה נתונים לשלוח בכל רגע נתון לפי מצב הרשת. כשמתגלה עומס, הוא מוריד את קצב השליחה כדי למנוע אובדן חבילות ולהתאים את עצמו לרשת.

האלגוריתם של Cubic ומנגנוני הניהול המתקדמים עוזרים ל-QUIC לנצל את רוחב הפס בצורה הטובה ביותר, גם כשיש עומסים ברשת. הוא יודע להגיב מהר לשינויים ברשת, מה שעוזר לו להימנע מעומסים ו"להתאושש" במהירות מאובדן חבילות. זה שהוא מנהל זרמים נפרדים גם מאפשר לשמור על ביצועים טובים גם כשחלק מהנתונים מתעכבים או נאבדים ובכללי בקרת העומס של QUIC עוזרת לשמור על אמינות הרשת, להפחית המתנה ולספק חוויית משתמש חלקה יותר, במיוחד בסביבות רשת מורכבות

חלק "רטוב"

בחרנו לממש ריבוי זרימות (flow) ב QUIC, נסביר פה את תהליך החשיבה וכתיבת הקוד.

כפי שראינו בשאלה 1, חסרון משמעותי של TCP הוא חסימת ראש התור (Head of Line Blocking), זה קורה כאשר חבילה אחת נאבדת וזה מונע את ההעברה של כל שאר החבילות עד שהחבילה שנאבדה "משוחזרת" ונשלחת שוב, זאת מכיוון שTCP שומר על אמינות מירבית אך עובד על זרם יחיד - דבר שפוגע בהעברה של חבילות שגם לא בהכרח קשורות לחבילה הזאת ויוצר עיכוב רציני ברשת. היתרון המרכזי אותו בחרנו לממש הוא שימוש בזרמים רבים כך שאובדן חבילה, יפגע רק בזרם בו היא נמצאת ובכך לא יחסום את מסירת החבילות בכל שאר הזרמים - דבר שישפר את הביצועים כי לא כל החבילות עכשיו צריכות "לחכות בתור" שהחבילה שנאבדה תישלח, אלא רק "תור אחד" בזרם בו החבילה ההיא הייתה.

על מנת להקל על המשתמש, "להסתיר" את איך שQUIC עובד ולחסוך בקוד, השתמשנו בפונקציות משותפות ב api.py (השראה ממטלה 2 במערכות הפעלה 🙂)

המימוש שלנו:

יצרנו פונקציה פשוטה (דומה לפונקציה שקיבלנו מכם במטלה 3) שתיצור קובץ עם מידע רנדומלי כדי שנוכל לדמות שליחה של מספר קבצים בזרמים שונים. את כל פונקציות העזר שעוסקות ביצירת הקבצים, מחיקה שלהם וכו', הכנסנו לתוך `generate_data.py`

על מנת לשלוח את הפאקטות בזרמים שונים, השתמשנו ב-Threading, שפשוט יריץ את ה sendto במקביל עבור כל קובץ בזרם שלה. בעיקרון, רק הלקוח צריך לפצל לזרמים עם Threading, השרת משתמש בsocket יחיד ולכן הוא יכול פשוט לקבל בו את כל המידע, הוא פשוט צריך לנווט את כל הנתונים שמגיעים אליו ל"זרם" המתאים שהוא שומר.

מאחר ואנחנו משתמשים ב-Threading ועלולים לגשת ולגעת במשתנים משותפים בכמה תהליכים במקביל, השתמשנו ב Threads lock כדי למנוע unexpected behaviour אם שניים או יותר תהליכים, מתעסקים עם אותו האיזור בזכרון במקביל, כך בכל פעם המנעול נמצא אצל Thread יחיד והוא היחיד שנוגע במקום הזה.

לאחר כתיבת הקוד, ניסינו לקחת על עצמנו גם להוסיף אמינות, על ידי שמירת כל הפאקטות שנשלחו מהלקוח במילון ומחיקה שלהן מהמילון ברגע שהלקוח מקבל ACK עליהן (עשינו Thread נוסף שפשוט מקבל כל הזמן חבילות ומוחק מהמילון של מי שמחכה לACK) אך זה פגע בביצועים ושינה ממש את הנתונים שרצינו לראות ולהדפיס בריבוי זרימות, וזה מעבר למה שהתבקשנו לעשות אז החלטנו לוותר על זה. בחרנו להשאיר את החלקים האלה של הקוד כדי לאפשר לעצמנו המשך משם.

מעבר לכך, ניסינו במהלך כל כתיבת הקוד, להשאיר מקום להתרחבות הקוד לפרוטוקול QUIC המלא (גם אם אנחנו נדרשים לממש QUIC מנוון עם ריבוי זרימות בלבד), עשינו את זה על ידי שימוש בheader המלא של QUIC, עם אפשרות ל Short Header ו- Long Header בpack-ינג של החבילה, כאשר ה Long Header הוא רק ללחיצת יד ובכל השאר זה Short Header כדי לחסוך ב: Overhead, גודל פאקטה כללי, לאפשר יותר מידע אמיתי בכל שליחה לנצל באופן מירבי של רוחב הפס. כמובן שאם היינו משאירים רק את הדברים הדרושים לנו ב-Header, היינו מקבלים מספר קטן יותר של חבילות בכל זרם מאחר והיה יותר מקום בכל חבילה לslice יותר גדול מהקובץ, אך בחרנו לאפשר את הפיתוח העתידי של הקוד.

נציין גם שאנחנו מתעלמים לחלוטין מכל packet loss שיכול להיות בבדיקות שלנו, ברור לנו שאנחנו עובדים מעל פרוטוקול שהוא לא אמין ויש צורך להוסיף בדיקות הגעה שלמה של החבילות, אך בחרנו להתמקד בריבוי זרימות.

לכן, עם המצב הנוכחי והדרישות של הפרויקט כרגע, כנראה ש-TCP יהיה יותר טוב מכל הבחינות, אך עם מימוש מלא ונכון של QUIC, ראינו לאורך כל הפרויקט שיש לו עדיפות על TCP.

- בשרת ובלקוח השתמשנו בשבלונה ממטלה 2, בשביל הarg parsing

server.py

מאחר ולא מימשנו אמינות בקוד שלנו, אין יותר מדי קוד בשרת, חוץ מלקרוא כל הזמן מהsocket

לפני ההגשה החלטנו להעביר גם את הדפסת הנתונים מהשרת ללקוח, זאת מכיוון שראינו שהרבה מהחבילות הולכות לאיבוד בדרך ורצינו הדפסת נתונים נכונה יותר לריבוי זרימות.

אז הקוד של השרת באמת נראה מנוון. אך אם נחליט להוסיף את ההדפסה ולשפצר קצת את שאריות הקוד שהשארנו מהACK, נצטרך גם להשתמש ב Threading כדי לקבל חבילות במקביל לשליחה של ACK-ים.

client.py

פה נמצא רוב הבשר.

כפי שציינו למעלה, השתמשנו ב-Threading כדי להריץ את כל הstreams במקביל ובThreading.Lock על מנת לשמור על הsocket ולוודא שאנחנו לא בטעות מתנגשים בקריאה או בכתיבה עליו.

גם כאן, השארנו את הקוד שהוספנו בשביל ACK כדי שנוכל להמשיך ולפתח את זה לכיוון הQUIC המלא

בסוף הריצה הדפסנו הודעה שכוללת את הסטטיסטיקה הנדרשת לכל זרם בנפרדת וביחד לכולם.

api.py

פה מימשנו מחלקה של `QuicPacket` שמתאימה לראש הארוך.

כדי לתפוס את כמות הביטים הנכונה לכל משתנה בפאקטה שנשלחת - עשינו pack ו- unpack (מהספרייה struct) לפי גודל המשתנה שאנחנו רוצים (כי פייתון שפה יותר דינמית אז לא באמת הגדרנו משתנים עם גדלים בשום מקום).

```
'''
Packing and unpacking:
!: represents network byte order (big-endian)
B: unsigned char (1 byte - 8 bits)
H: unsigned short (usually 2 bytes - 16 bits)
I: unsigned int (usually 4 bytes - 32 bits)

We do this because that our way to control the sizing of the fields in the packet in python, to minimize overhead
'''
```

הביט הראשון מסמל האם זה ראש ארוך או קצר ולכן לפיו אנחנו הולכים לפונקציה הפרטית המתאימה שתפרק או תרכיב את החבילה לפני/אחרי שליחה/קבלה

```python
def pack(self):
    # Checks header form to determine if it is a long header or a short header
    if self.header_form == LONG_HEADER:
        return self.__pack_long()
    else:
        return self.__pack_short()

def unpack(self, data):
    # Checks header form to determine if it is a long header or a short header
    if data[0] == LONG_HEADER:
        self.__unpack_long(data)
    else:
        self.__unpack_short(data)
```

לפי RFC 8999 חלק 5.1, זה הראש הארוך שמימשנו

```python
def __init__(self, source_id, destination_id, payload, stream_id, pos_in_stream):
    # Long Header as suggested in RFC 8999 Section 5.1
    self.version_specific_bits = 0          # not used - Version-Specific Bits (7 bits)
    self.version = 0                        # not used - Version (32 bits)
    self.destination_connection_id_length = 1  # not used - Destination Connection ID Length (8 bits)
    self.destination_connection_id = destination_id      # Destination Connection ID (0..2040 bits, we set it to 1 bit to allow only 2 co
    self.source_connection_id_length = 1      # not used - Source Connection ID Length (8 bits)
    self.source_connection_id = source_id           # Source Connection ID (0..2040 bits, we set it to 1 bit to allow only 2 connection I
    # self.version_specific_data = 0          # not used - Version-Specific Data (..)
    self.stream_id = stream_id
    self.pos_in_stream = pos_in_stream
    self.payload_length = len(payload)       # used instead of above comment, Payload Length (0..2^16-1, 16 bits)
    self.payload = payload                   # Payload (0..2^16-1)
    if not isinstance(payload, bytes):
        self.non_binary_payload = payload
    else:
        self.non_binary_payload = self.payload.decode("utf-8")
    self.__set_packet_type()
```

ברוב אנחנו לא משתמשים אבל שמרנו כדי לאפשר התרחבות וגם את חלקם אנחנו בכלל לא דוחסים לפאקטה, אלה הם פה לנוחיות שהתנהלות עם החבילות

UnitTesting.py

קובץ UnitTesting שנועד לבדוק את הפונקציות שכתבנו לאורך כל הפרוייקט.

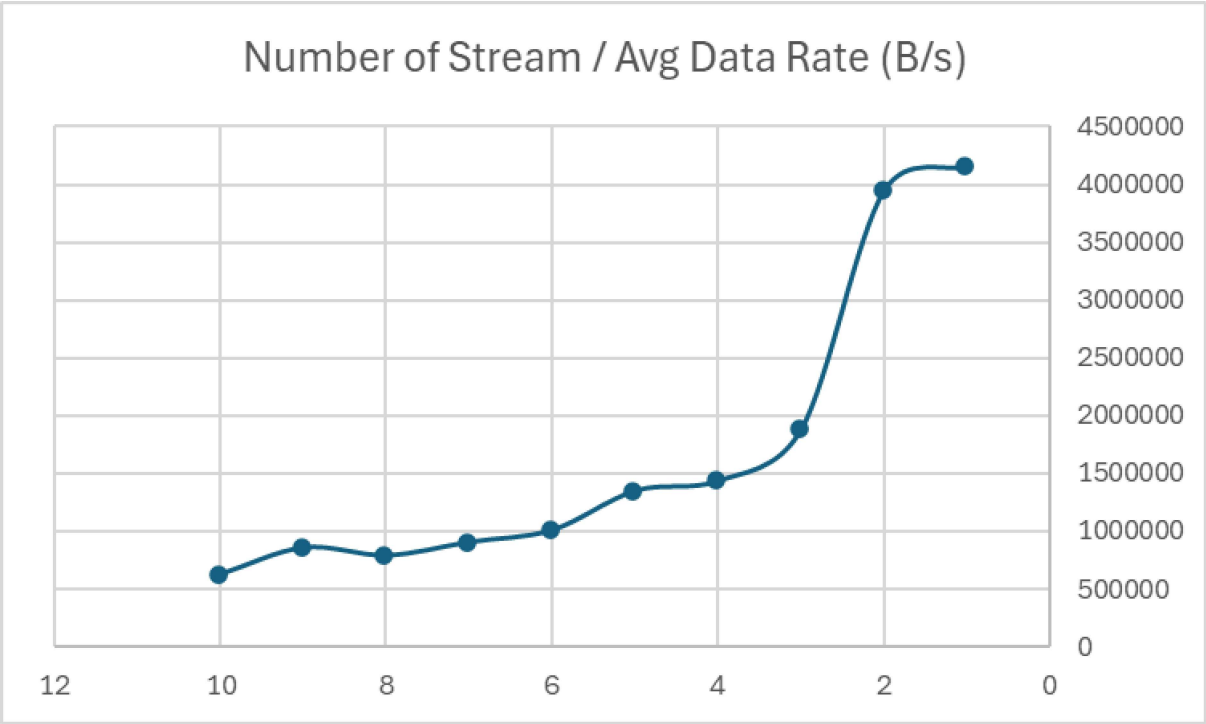
התמקדנו בעיקר בבדיקת מבנה הפאקטה שיצרנו, הpack ו-unpack-ינג שלה

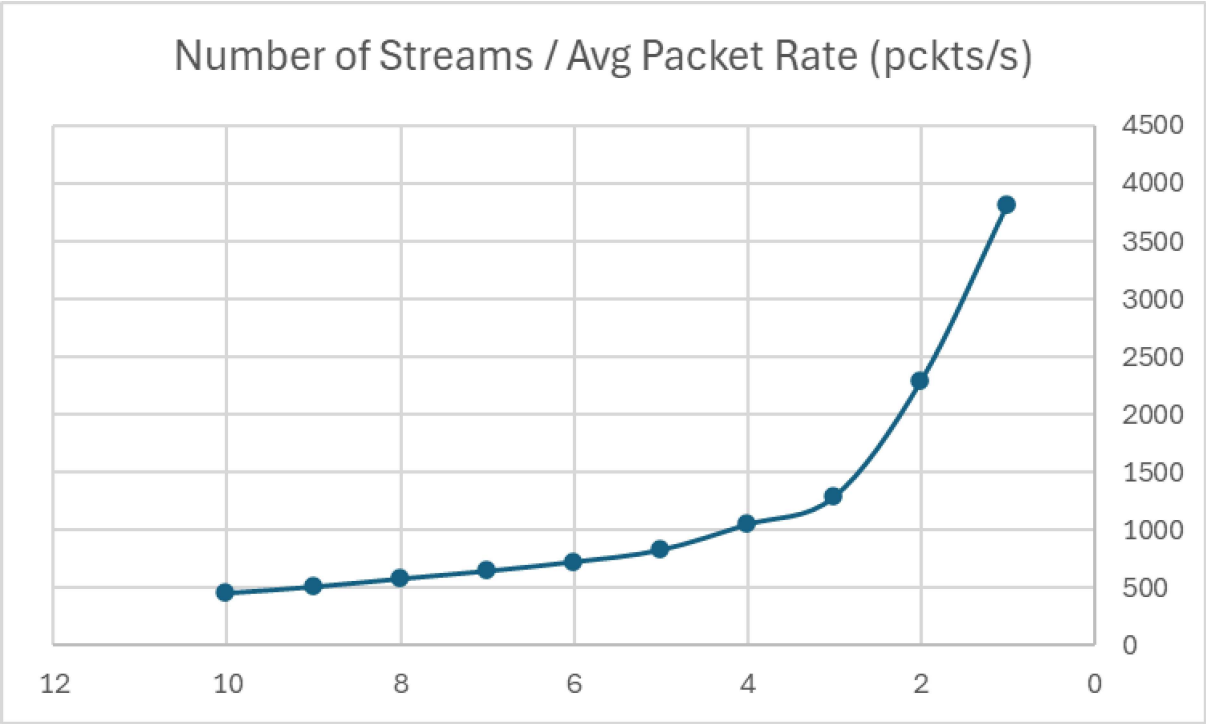
main.py

קובץ שהכנו כדי שנוכל להריץ אותו והוא יריץ את הניסויים שביקשנו, בסוף מקבלים קובץ client_stats.txt, כאשר כל שורה מייצגת ניסוי בפורמט הבא:

NumOfStreams,AvgDataRate,AvgPacketRate

הרצנו ניסויים על מספר זרימות שמשתנה בין 1 ל-10. נציג כאן את הגרפים של הנתונים הכוללים (ממוצע הזרמים, סעיפים ד ה) ונסביר בקצרה את המגמה ואת הסיבות לה.





(הסיבה להבדל בין הגרפים היא לדעתנו זה שאנחנו מגרילים גודל פאקטה שונה בכל ריצה, אז אין עקביות כל כך בין מספר הפאקטות למספר הביטים שנשלחו בשניה (כי גודל החבילה משתנה בין ריצות)

מסקנה מהגרפים:

אנחנו רואים שעם זרם אחד, נשלח הרבה יותר מידע בשניה, כי הוא היה הזרם היחיד שרץ והיה לו את כל הרוחב פס לרשותו (כמו רכב על כביש מהר עם כל הנתיבים לרשותו). ברגע שאנחנו מפצלים לכמה זרמים (נותנים לכל רכב כמות מוגבלת של נתיבים), אנחנו רואים שהקצב הכללי של כולם יורד (עליה ל-2 היא לא ירידה משמעותית, אבל ל-3 משום מה זה היה קצת קשוח). הקצב יורד כי עכשיו הרוחב פס נתפס על ידי יותר מזרם אחד - אז במקרים בהם יש איבוד חבילות גבוה וטיפול בחבילות

בעיתיות, זה יהיה מעולה ויעזור לנו מאוד כי איבוד חבילה יפגע רק בזרם יחיד, אבל במקרה שלנו, בו לא התייחסנו לאיבוד חבילות בכלל, אנחנו רואים רק את החסרון שבזרמים מרובים - חלוקת רוחב פס לכמה מנצלים והורדת המהירות של כל מנצל. לכן, נרצה להשתמש בריבוי זרימות רק אם יש רשת אמינה לפחות בקצת, שכוללת בדיקת איבוד חבילות וכו'.

Avg Packet Rate	Avg Data Rate	Number of Streams
3817.544309	4164570.684	1
2288.054342	3954218.527	2
1291.6575	1881896.233	3
1054.63579	1436840.039	4
833.7639327	1351391.853	5
730.8690462	1011125.543	6
651.7965271	906509.2141	7
585.2681455	790754.869	8
515.8433014	859218.5349	9
460.668339	629064.2275	10

* הוספנו לכאן שיהיה, ה client.py מוציא בסוף הריצה את הנתונים האלה לתוך קובץ client_stats.txt. הטבלה הזאת יצאה בסוף הריצה של main.py (שעשה בעצם את הבדיקה לכל הניסוי הזה)