**החלק ה״יבש״ בפרויקט**

**5 חסרונות/מגבלות של TCP**

1. חוסר יעילות שנובע משימוש באותו אלגוריתם עבור Congestion Control וגם Reliable Delivery

TCP שואף לשלוט במדויק במספר הפקטות שנמצאות בתוך רשת ומסתמן שהוא משתמש באותו האלגוריתם חלון עבור flow control וגם reliable delivery שזה לא אופטימלי - לדוגמא, אם פקטה הולכת לאיבוד אז חלון שלם של פקטות ״ייזרק״ אפילו אם חלק מהם כבר הגיעו ליעד בהצלחה.

2. HoL - Head of Line Blocking

TCP מוודא שהדאטא מדולבר בסדר מסוים, אך אם פקטה אחת הולכת לאיבוד, אז כל הפקטות שאחריה יחכו עד שהפקטה שהלכה לאיבוד תשלח שוב ותתקבל ובכך להאט את התפוקה של ההעברה.

3. חיבור ראשוני איטי

לפני שTCP יכול לשלוח דאטא, חייבת להתקיים לחיצת יד משולשת (3-way handshake), תהליך שלוקח זמן. הדילי הזה יכול להוות בעיה, במיוחד שדרוש חיבור מהיר. בנוסף, פרוטוקולים מתקדמים של TCP יכולים לממש גם תהליכי אימות ואבטחה לפני החיבור, מה שאפילו יאט את החיבור הראשוני עוד יותר.

(לדוגמא בעת שימוש באינטרנט כנראה שנצטרך גם לבצע לחיצת יד TLS על מנת להבטיח חיבור מאובטח)

4. Header קבוע בגודלו

לTCP יש Header בגודל קבוע, מה שמגביל כמה עם כמה דאטא הוא יכול לעבוד בצורה יעילה, במיוחד במהירויות גבוהות.

5. שינוי כתובת IP

חיבור TCP מאופיין ע״י כתובת IP ומס׳ הפורט הרלוונטי. אם כתובת הIP תשתנה בעת חיבור, נצטרך להתחבר מחדש - מה שגורר handshake חדש וכפי שאמרנו ב3. זה תהליך שהוא איטי.

**5 תפקידים שפרוטוקול תקשורת צריך למלא**

1. Reliability: הפרוטוקול צריך להבטיח שהנתונים אכן מועברים ויוכל לשדר מחדש נתונים שאבדו.

2. Efficiency: הזמן שלוקח לעיבוד והעברת נתונים צריך להיות מהיר יחסית.

3. Scalability: הפרוטוקול צריך לתמוך בגדלי רשת שונים, במיוחד בכמות גדולה.

3. Compatibility: הפרוטוקול צריך לעבוד על פלטפורמות רבות ללא בעיה דבר שהוא חשוב במיוחד בעידן של היום.

4. Secure: הפרוטוקול צריך להיות עמיד מפני גישה לא רצויה ולהגן על הנתונים שלו באופן מסוים (הצפנה).

5. Adaptive: הפרוטוקול אמור להיות מסוגל להתאים את עצמו לתנאי הרשת על מנת למקסם את הביצועים שלו.

**לחיצת היד כפי שהיא ממומשת בQUIC**

QUIC משלבת את הלחיצת היד התעבורתיות והקריפטוגרפיות (לדוגמא TLS) אל תוך תהליך יחיד שלוקח 1-RTT בלבד. בנוסף, לחיצת היד של QUIC משלבת הצפנה (דרך TLS 1.3 ) דרך הפרמטרים התעבורתיים. התהליך הזה הרבה יותר מהיר בהשוואה לTCP שעושה את תהליך לחיצת היד ב2-RTT. בQUIC, הפקטה הראשונית מנהלת ״משא ומתן״ על מזהי החיבור ובנוסף היא גם מהווה לחיצת יד של TLS על מנת לאבטח פקטות עתידיות. השרת יכול לאמת את הכתובת הלוקח עם פקטה נוספת אך זה לא חובה.

בנוסף, QUIC גם תומך ב0-RTT transmission, מה שמאפשר ללקוחות לשלוח מידע מוצפן כבר בפקטה הראשונה בתנאי שהלקוח תקשר לאחרונה עם השרת.

A diagram of a server and client

Description automatically generated with medium confidence

(Taken from https://commons.wikimedia.org/wiki/File:Tcp-vs-quic-handshake.svg)

**מבנה הפקטה בQUIC**

פורמט הפקטה:

בTCP גודל הheader הוא קבוע. אך בQUIC ישנם שני סוגים שונים של headers. במהלך יצירת החיבור, QUIC משתמש בheader גדול כדי לכלול מידע הכרחי. לאחר יצירת החיבור, QUIC עובר לתבנית header קצרה יותר שמכילה רק את המידע ההכרחי ובכך יעיל יותר. בנוסף, לכל פקטה בQUIC ישנה מזהה ייחודי שמאפשר לשמור על סדר השידור ועוזר בתהליך שחזור הפקטה במקרה והלכה לאיבוד.

A diagram of a network

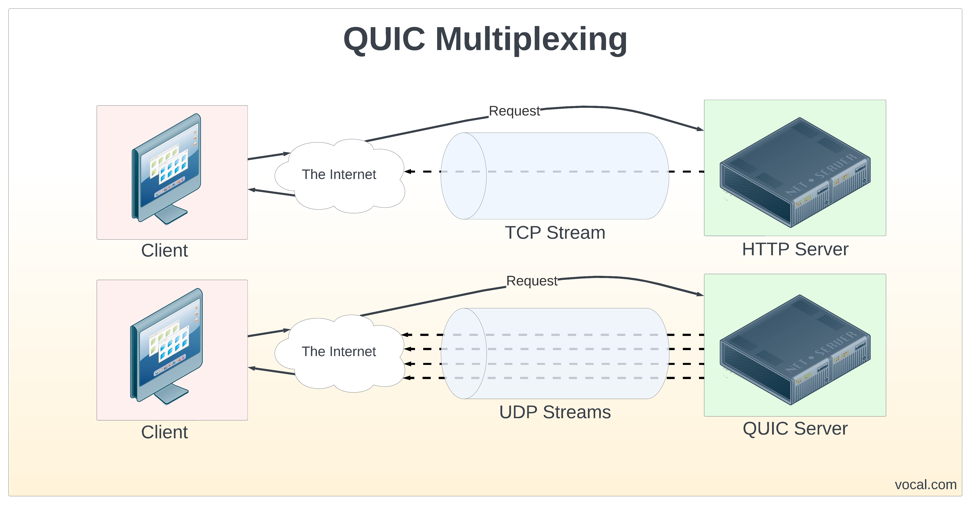
Description automatically generated

(Taken from https://www.f5.com/content/dam/f5-com/nginx-import/primer-QUIC-networking-encryption\_stream-anatomy.png)

זרימות:

QUIC משלב ריבוי זרימות אל תוך השכבה התעבורתית בדומה לHTTP/2. בQUIC לכל חיבור יכול להיות מספר זרמים בו-זמנית, המזוהים ע״י stream ids ייחודיים. כל זרם כזה מתנהג בדומה לחיבור TCP כך שזרימת המידע הינה מסודרת. כל מידע בתוך זרם מפוצל לפריימים וכל פריים מזוהה לפי שילוב של הid והoffset שניתן לו - מה שעוזר למנגנוני loss detection של הפרוטוקול לזהות מידע שאבד ולהשלים אותו במידת הצורך.

בנוסף, נשים לב שהשימוש בזרימות מנטרלת את בעיית הHead-of-Line (HOL) הקיימת בTCP כפי שציינו ב1. במקרה של איבוד פקטות (packet loss) רק הזרם שבו הייתה אבדה יתעכב ואילו שאר הזרימות יזרמו כרגיל.



(Taken from https://vocal.com/wp-content/uploads/2024/02/QUIC-Multiplexing-2048x1066.png)

תמיכה בunreliable datagram delivery:

ישנן אפליקציות, במיוחד אלו שמשתמשות במידע בזמן אמת, המעדיפות לשדר מידע מאובטח באופן לא אמין בתמורה למהירות. אפליקציות מסוג זה, יעדיפו להשתמש בUDP או בDTLS. QUIC גם תומך בשידור מידע מאובטח אך לא אמין באמצעות Datagram Frames. בניגוד לפריימים רגילים, אלו פריימים שלא ישודרו פעם נוספת במקרה וילכו לאיבוד. בהשוואה לUDP או DTLS, QUIC מבצעת לחיצת יד המאפשרת שידור מאובטח אך לא אמין.

**מה QUIC עושה כאשר פקטות מגיעות באיחור או לא מגיעות בכלל?**

QUIC משתמש בזיהוי אובדן מבוסס ACK בכך שכל פקטה מפוצלת למספר של פריימים וכשפקטה היא ACK, הוא מאשר את קבלת הפקטה כולה וכל הפריימים שלה מתקבלים. אם פקטה לא מקבלת ACK אך פקטה מאוחרת יותר תקבל ACK, הפקטה המוקדמת והפריימים שלה ילכו לאיבוד.

QUIC משתמש ב2 דרכים על מנת לקבוע איבוד פקטות:

1. לפי המספר שלהן (לכל פקטה יש מזהה ייחודי)
2. עבר זמן מסוים ולא התקבל ACK עבור פריים מסוים

הדרכים הללו מאפשרות לסידור מחדש של הפקטות ובכך הפרוטוקול מונע שידור מחדש ללא צורך ממשי בכך.

וכשפקטה הולכת לאיבוד, הפריימים שלה ישודרו מחדש בדומה לTCP - באופן אמין ומסודר.

**הCongestion Control של QUIC**

QUIC מיישם CC באופן דומה לTCP. כפי שציינו קודם, QUIC ימספר כל פקטה וישתמש בoffset בframes על מנת להבטיח אמינות. בנוגע לCC, QUIC משתמש בCC מבוסס חלון, בדומה לTCP. לQUIC אין מימוש ספציפי של אלגוריתם CC מבוסס חלון, ומשאיר את המימוש בצד הלקוח - מה שהופך את התהליך לגמיש יותר.

כדי למנוע הקטנת חלון לא רצויה, QUIC יצמצם את החלון שלו אך ורק אם יזהה עומס. עומס מזוהה אם שתי פקטות אובדות ואין פקטות ביניהן המקבלות ACK. המרווח זמן בין הפקטות האבודות הללו מחושב בשילוב הRTT הנוכחי כדי לקבוע עומס.

מצד הלקוח, הלקוח ימתן את כמות השליחות שלו ע״י שליחת הפקטות שלו במרווחים המבוססים על הRTT הממוצע ובכך יופחת הסיכוי לעומס.

**החלק ה״רטוב״ בפרויקט - מימוש ריבוי זרימות**

**מבוא**

החלטנו לממש את חלק 1 ­­— ריבוי זרימות כאשר ממשנו שתי תוכניות פייתון: client.py וserver.py. החלטנו שבשרת יהיו מאוחסנים קבצים רבים בפורמט .txt בגדלים שונים וכאשר לקוח מתחבר לשרת, השרת יעביר את ללקוח את כל הקבצי הטקסט שמאוחסנים בו. למרות שאנחנו לא מממשים ריבוי זרימות כפי שהן בQUIC באופן ישיר, הגישה של השרת לטיפול בהעברות קבצים במקביל מדמה את הרעיון של זרמים ב-QUIC.

**צד השרת ומימוש ריבוי הזרימות**

השרת יוצר סוקט על פורט מסויים ואז מאזין לחיבורים של לקוחות. עם חיבור של לקוח השרת ישלח לו את כל הקבצים שזמינים להורדה. השרת תומך בעד 10 לקוחות במקביל.

לגבי ריבוי זרימות, נזכיר שאנחנו מנסים לדמות זרימות כפי שמתואר במאמר; כל קובץ שישלח ללקוח מהווה זרם, כאשר עבור כל זרם השרת יקצהthread בודד מה שמדמה את הרעיון של מספר זרימות תחת חיבור אחד כפי שהן ממומשות בQUIC.

השימוש בthreads שונים מייעל את הביצועים בצד השרת ומנטרל את בעיית הHoL אך גם מביא אותנו לבעיה הבאה: הזרמים פועלים במקביל אחד לשני וגורמים למידע להישלח ללקוח בצורה לא מסודרת. נסביר כיצד פתרנו את זה בצד הלקוח בפסקות הבאות.

חשוב לדעת שתקשורת לקוח-שרת מוגבלת בדרך כלל על ידי רוחב הפס של הרשת. העברת מספר קבצים במקביל על פני threads שונים אינה מגדילה את רוחב הפס (המחשבים המודרניים יודעים למקסם את רוחב הפס..) ומה שקורה בפועל בפרויקט שלנו זה שהשרת מחלק את רוחב הפס הזמין בין threads הקיימים.

**צד הלקוח**

צד הלקוח הוא פשוט יחסית, הלקוח מתחבר לשרת בעזרת כתובת ופורט ומבקש את כל הקבצים שמאוחסנים בו.

כפי שציינו מקודם, ישנה בעיה שצצה בעת שימוש במספר threads בצד השרת עבור כל קובץ: המידע שהלקוח יקבל אינו מסודר; ייתכן שהוא יקבל חבילה שעליה יש מידע ששייך לקובץ מס׳ 1 אחרי זה חבילה עם מידע ששייך לקובץ מס׳ 3 ואחרי זה שוב חבילה שקשורה לקובץ מס׳ 1.

כדי לפתור את זה, השרת שולח כל פריים בצורה הבאה:

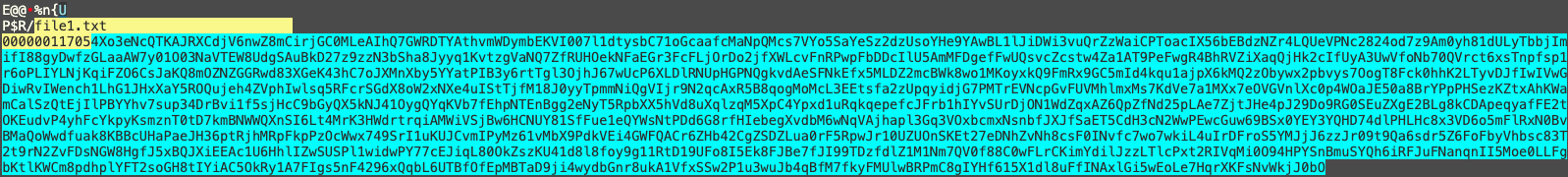
file\_name | frame\_id |frame\_size| file\_data

כאשר החלק הירוק הוא הheader והחלק הסגול הוא הdata.

גודל הheader הוא קבוע בגודל 43 בייטים. הלקוח תחילה ישלוף בדיוק 43 בייטים מהסוקט שלו, ואז ישלוף שוב מידע בגודל frame\_size מהסוקט frame\_size) יוגרל בין 1000 ו2000 בייטים עבור כל קובץ בצד השרת .(

ולבסוף, הלקוח יסדר את המידע שהוא קיבל בצד שלו באמצעות הfile\_name והframe\_id. גישה זו דומה לאופן שבו QUIC מנהלת את המידע בתוך הזרמים שלה; באמצעות הרכבה נכונה בצד הלקוח בהתבסס על frame\_id.

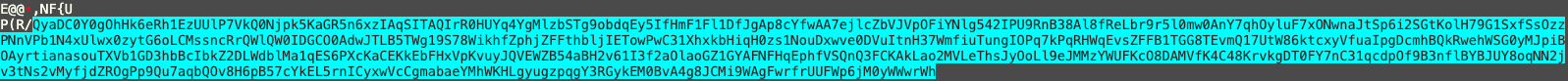
**הפקטה הראשונה**

****

(הפקטה נלקחה מההקלטה 3\_files\_test, מס׳ פקטה 1)

כבר בפקטה הראשונה, ה43 בייטים הראשונים מהווים את הheader (מסומן בצהוב בתמונה למעלה). הclient יקרא תחילה 43 בייטים בדיוק מהסוקט, ואז ישלוף שוב מהסוקט מידע בגודל 1705 וידע לקשר אותו לfile1.txt.

לרוב בהקלטות, עבור כל פקטה, הheader יהיה בתחילת המידע. לפעמים יהיו מקרים שנראה כאילו המידע אכן מפוצל וללא header, לדוגמא:



(הפקטה נלקחה מההקלטה 3\_files\_test, מס׳ פקטה 100)

אין לנו שליטה במתי הפקטה תתחלק, בגלל שאנחנו משתשמים בTCP כבסיס.

אך בפועל זה לא משפיע על הלקוח - הheader כבר נשלח ומיד אחריו יגיע המידע נוסף הזה.

**כיצד ריבוי זרימות מתבטא ברשת?**

A screen shot of a computer screen

Description automatically generated

(הפקטה נלקחה מההקלטה 3\_files\_test, מס׳ פקטה 99)

כאן כבר ניתן לראות את השימוש בריבוי זרימות - על גבי פקטה אחת, נמצא מידע ששייך למס׳ קבצים שונים, באופן לא מסודר. הלקוח יודע איך לסדר את הפריימים האלה אצלו ולבסוף לייצר את הקובץ הדרוש ובכך אנחנו מרוויחים ניצול מקסימלי של כל פקטה.

**שימוש בתקשורת בזמן 0-RTT**

השרת שלנו מחקה תקשורת 0-RTT על ידי שליחת נתונים ביחד כבר בחבילה הראשונה, ללא צורך בלחיצת יד. המימוש הזה ממזער את הזמן שלוקח להתחיל בהעברת נתונים ודומה מאוד למימוש שקורה בפועל בפרוטוקול QUIC.

**ניסויי על מספר זרימות בין 1 ל-10**

ראשית, לצורך אחידות הניסוי, יצרנו 10 קבצים זהים בגודל 4mb שאותם נעביר בפרוטוקול שלנו. תחילה, נציג את הנתונים שאספנו:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | # of streams (files) |
| 9528417 | 8889883 | 9125150 | 9015263 | 8798714 | 8737882 | 8633710 | 8206837 | 8604580 | 8294456 | d |
| 5648 | 6155 | 6363 | 6042 | 6573 | 6853 | 6298 | 6430 | 5560 | 5454 | e |

**הגרף של e כפונקציה של מספר זרימות**

**סיכום תוצאות הניסוי - האם יש קשר בין מספר הזרימות למספר הפקטות הממוצע לשנייה?**

מספר הפקטות הממוצע לשנייה, ניתן לראות תחילה שככול שנוספים זרמים, קצב מספר הפקטות הממוצע לשנייה משתפר. אחריו, כאשר מגיעים למספר זרימות בגודל 5-7 נראה שקצב הרשת מגיע למישור ולבסוף, הוספת זרמים נוספים (8-10) גורמות לדעיכה במספר הפקטות הממוצע לשנייה.

לסיכום, מסתמן שהוספת עד כ-5 זרימות משפרת את מספר הפקטות הממוצע לשנייה יביא אותנו לתוצאות האופטימליות ביותר. לאחר מכן, הוספת מס׳ קטן של זרימות נוספות לא תשפיע אך ככול שנוסיף יותר ויותר זרימות כך ביצועי הרשת ידעכו, ככול הנראה בגלל שהמערכת כבר הגיעה לגבולה ב5-7~ זרימות וכל זרימה לאחר מכן תגרום לעומס וoverhead.

**הגרף של d כפונקציה של מספר זרימות**

**סיכום תוצאות הניסוי - האם יש קשר בין מספר הזרימות למספר הבייטים הממוצע לשנייה?**

נראה שהיו נפילות ספציפיות כאשר הוספנו זרימה נוספת בהינתן 2 זרימות ו8 זרימות. קשה לקבוע מה הסיבה לנפילות אלו וסביר להניח שמדובר בנפילות מקריות בגלל תנאי רשת, כיוון שלא ניתן להניח איזשהו דפוס. סך הכל, מה שכן ניתן לראות בניסוי המצומצם שלנו זה שככול שמוסיפים יותר זרימות, קצב המספר הבייטים הממוצע לשנייה עולה - כך ביצועי הרשת יהיו טובים יותר (נוכל להעביר יותר מידע בכל שנייה). נצטרך לעשות ניסוי רחב יותר עם מספר גדול יותר של זרימות על מנת לקבוע מה הוא מספר הזרימות האופטימלי.

**לקינוח: אי-יעילות ידועות בפרויקט שלנו**

* אי-יעילות בגודל הheader:בכל פקטה, אנו שולחים עימה סוג של header שמציין את שם הקובץ, גודל החבילה שהוגרל ע״י השרת ועוד. חלק מהפרטים אלו הם קבועים ולכן ניתן לפתור את האי-יעילות הזאת בקלות באמצעות שליחת שם הקובץ וגודל החבילה רק בפקטה הראשונה ואילו בפקטות הבאות להשתמש בheader קטן יותר שמציין רק את מזהה הקובץ והמידע בפקטה - כפי שקורה בפועל בפרוטוקול QUIC.
* כמות מרובה של קבצים: כאשר יהיו לנו כמות גדולה של קבצים, השרת יקצה לכל קובץ thread משלו מה שיצא מכלל שליטה באופן מהיר מאוד וכמובן שזה לא אופטימלי.
* שימוש בTCP כפרוטוקול: כמובן שלא התבקשנו לממש QUIC לגמרי אך חשוב לציין שהשימוש בTCP מגביל אותנו מאוד בהמון מובנים וגם בפועל QUIC בנוי מעל UDP.

**נספחים**

מרבית התשובות שלנו בחלק היבש מתבססות על המאמר "A Quick Look at QUIC"

https://web.cs.ucla.edu/~lixia/papers/UnderstandQUIC.pdf

https://en.wikipedia.org/wiki/QUIC

https://blog.cloudflare.com/the-road-to-quic

https://www.chromium.org/quic/

ChatGPT