ספר פרויקט

044167

Distributed Stateful Applications on Programmable Switches

NSSL

Project #7062

מנחה: ליאור זינו

מגישות:

סיון ורט	318883469
יובל בן חיים	209120864

תוכן עניינים

תקציר

ככל שהשימוש ברשתות תקשורת הולך וגובר עם השנים, איתו גדלה גם כמות המידע המועבר, וכך נוצרו עומסים ברשת. מכאן, שנוצרה דרישה למעבר על התעבורה ברשת וניתוחה באופן מהיר, דינאמי ואפקטיבי עוד בשלב ה-data plane, זאת על מנת לייעל את השימוש ברשתות ולהקל על המשתמשים בהם. ישנה חשיבות גדולה בניתוח מוקדם של המידע ויצירת איזון של העומסים ברשת כדי להביא לחסכון בזמן של העברת המידע, ובפרט בהקשרים של שרתים ונקודות קצה רבות הדורשות תיאום רב ביניהם.

מטרת הפרויקט, להציע את השימוש באלגוריתם QPipe אשר מבצע ניטור ועיבוד של התעבורה ברשת באופן דינאמי על מנת לאזן את מעבר המידע ברשת ולייעל את ניהולה.

האלגוריתם דורש זיכרון אך מציע פתרון לניהול הרשת תוך כדי חיסכון בזיכרון הנשמר. כלומר, באמצעות QPipe יחול שיפור ביחס בין גודל הזיכרון שבשימוש לאחוז הדיוק באיסוף הסטטיסטיקות על התעבורה.

בנוסף, בפרויקט מוצג יישום בו השימוש באלגוריתם יכול לסייע לאיזון העומסים ברשת. היישום כולל host שמעביר חבילות ברשת דרך מתג לשתי נקודות קצה שונות. בעזרת האלגוריתם החבילות ממוינות ומחולקות בין שתי נקודות הקצה השונות באופן מאוזן.

מבוא

עם עליית השימוש ברשתות תקשורת, ניכרים עומסים בצמתים מרכזיים ברשת. לאור זאת, עלתה הדרישה לאיזון התעבורה ברשת וביזור עומסים, כדי לייעל את הרשתות ולהקל את השימוש בהן. פתרון אפשרי לבעיה הינו מעבר על המידע ברשת, וניתוח שלו באופן מהיר היאפשר איסוף מידע על התעבורה ברשת.

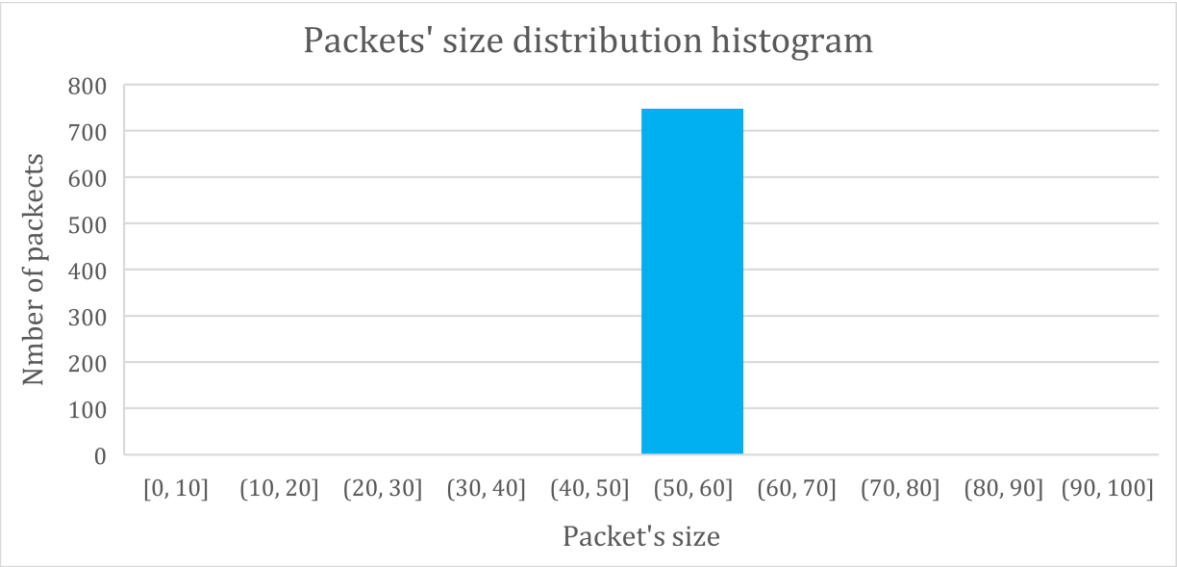
למשל, כאשר מדובר בשרתים גדולים יכול להיווצר עומס על השרתים ונקודות הקצה, שיכול לגרום לעיכוב חבילות ואובדן שלהן. בעזרת ניתוח התעבורה ואיזון העומסים ברשת, ניתן לנהל את הרשת באופן בו המידע יגיע ליעד בצורה יעילה יותר ואף בזמן קצר יותר, וכך למנוע צווארי בקבוק ברשת. איסוף וניטור מידע סטטיסטי חיוני של תעבורת הרשת יכול לשפר את יעילותה ואת אופן הניהול שלה.

בפרויקט זה, ממומש אלגוריתם הנקרא QPipe, אלגוריתם מסוג ה-Streaming Algorithms, אשר מטרתו לנטר, לאסוף ולעבד את המידע במתג עצמו. יצירת הסטטיסטיקות על המידע המועבר ברשת באופן דינאמי בזמן ה-data plane, תוביל לכך שזמן התגובה לזיהוי וטיפול באנומליות ברשת יתקצר משמעותית.

נציין כי לשם כך, נדרש זיכרון רב אשר הינו משאב מוגבל מפני שלמתגים יש רק עשרות בודדות של מגה ביטים.

כיום, הגישה הרווחת היא שימוש ב-sampling-based solutions, זוהי גישה לפיה לא נדרשים ניתוח וחקירה של הדגימות על מנת לקבוע פתרון אלא על פי הדגימות הרנדומליות של המידע נקבע מבנה (עץ או גרף למשל) מהם מוחלט הפתרון המתאים. גישה זו נפוצה מפני שהיא נוחה למימוש, אבל היא דורשת זיכרון רב על מנת להגיע לדיוק עבור streams גדולים. למשל, שימוש בהיסטוגרמה אשר מתארת חלוקה של הפקטות ברשת לפי גודל הפקטה. כל עמודה בהיסטוגרמה תייצג טווח גדלים, ורגיסטר בנתב ישמש כמונה עבור כמות הפקטות המסווגות לטווח זה. מידת הדיוק של המידע שנאסף תיקבע לפי הטווחים של חלוקת ההיסטוגרמה, ותדרוש יותר זיכרון על מנת להגיע לדיוק טוב. בנוסף, שימוש בהיסטוגרמה יכול להוות "בזבוז" של זיכרון, שכן ייתכן שישנם הרבה מונים ריקים, כלומר זיכרון "מבוזבז" (כי אף חבילה לא תהיה בגודל של טווח מסוים של ההיסטוגרמה).

לדוגמה, שימוש בהיסטוגרמה בעלת 10 עמודות המייצגות 10 מונים בזיכרון, כך שכל מונה שומר את מספר הפקטות אשר גודלן בטווח מסוים המצוין מטה. ברשת מסוימת נקלטה תעבורה אשר גודל כל הפקטות הוא בין 50 ל-60, ולכן תתקבל ההיסטוגרמה הבאה:

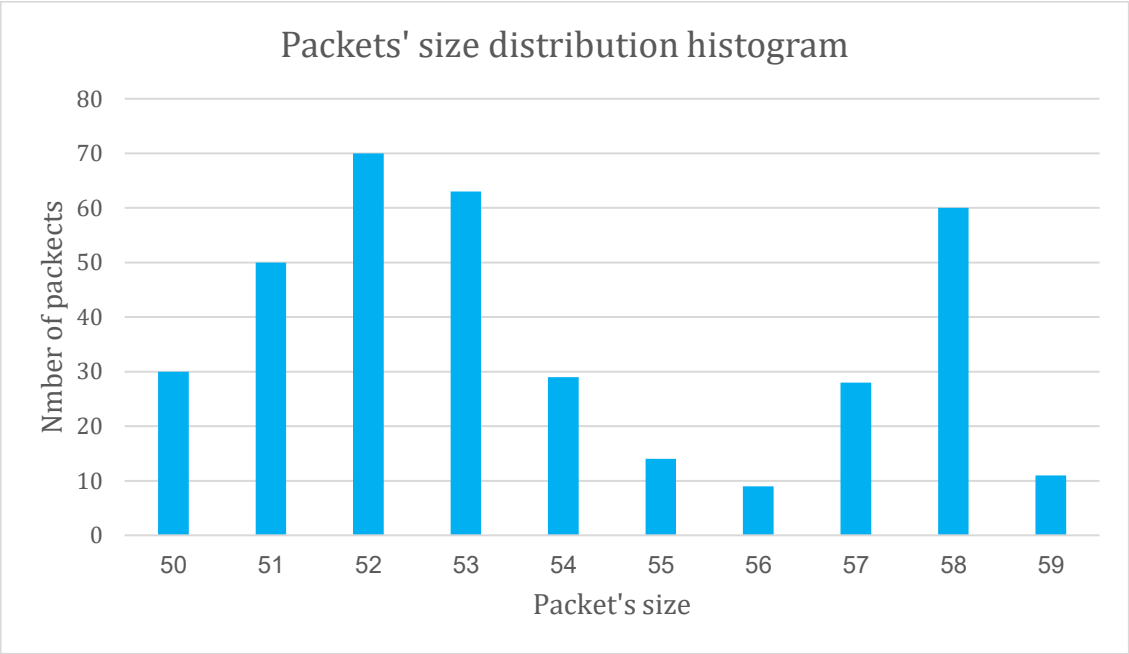


איור 1 : היסטוגרמת התפלגות גדלי חבילות

ניתן לראות שישנם הרבה מונים "מבוזבזים", כלומר זיכרון לא מנוצל, ומעבר לכך הדיוק בגודל הפקטות יהיה נמוך מאוד שכן הסקלה של העמודות לא מתאימה.

בפועל, היה רצוי לקבל היסטוגרמה בה כל עמודה מייצגת טווח מצומצם יותר, אשר תואם את הטווח של גודל הפקטות שעוברות ברשת. באופן זה, יינתן מידע מדויק יותר על אודות ההתפלגות של הגדלים השונים, אך לא ניתן לדעת הטווח הרצוי מראש.

ההיסטוגרמה הרצויה :



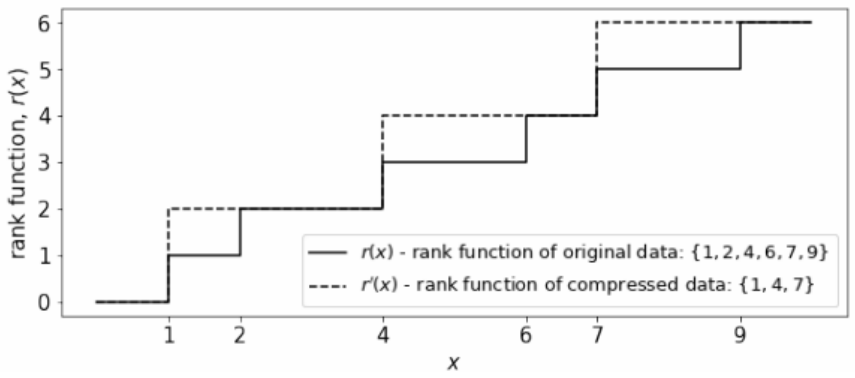
איור 2 : היסטוגרמת פילוג גדלי חבילות עם טווח מצומצם

השימוש ב-QPipe נועד על מנת לשפר את היחס בין גודל הזיכרון שבשימוש, לאחוז הדיוק באיסוף הסטטיסטיקות על התעבורה ברשת.

דוגמה המסבירה את ה-quantile sketching:

ברשת מסוימת התקבל המידע הבא: 1, 2, 4, 6, 7, 9. לכל ערך data יש דרגה משלו שמסמנת את מיקומו ביחס לסט המידע המקורי, כפי שניתן לראות באיור מטה בגרף הרציף. על מנת לצמצם את כמות הזיכרון, יש לחתוך את סט המידע לחצי. בדוגמה הוסר כל המידע במיקום הזוגי, ובמקביל הוגדל המשקל של כל דרגה פי 2, כלומר הדרגות שכעת המידע יכול לקבל הינן 2 4 ו-6.

לכל ערך מהמידע החתוך תותאם דרגה המתאימה. נשים לב שעבור חלק מהמידע, מתקבלת הדרגה אשר הייתה במקור, למשל עבור 2, 6 ו-9, ועבור חלק מהמידע קיבלנו שגיאה שהיא לכל היותר 1. כלומר, בחצי מכמות המונים (שקול לחצי מכמות הזיכרון) קיבלנו שגיאה קטנה יחסית עבור הדרגה.



איור 3: תוצאות דרגת quantile sketching

רקע תיאורטי

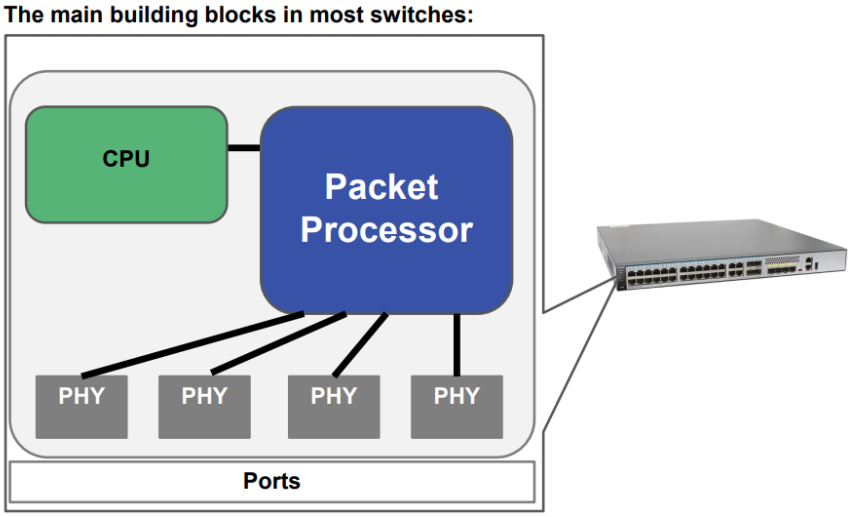
ארכיטקטורה

מתג הינו רכיב רשתי המחבר בין צמתים שונים ברשת, בין מכשירי קצה (כגון מחשבים ושרתים), ובין מרכיבי רשת בסיסיים (Hub, מתגים נוספים וכו'). המתג שייך לשכבה השנייה במודל השכבות OSI, שכבת ה-MAC (נקראת גם Link Layer). תפקיד שכבה זו הינו להעביר את החבילה בתוך הרשת המקומית, בין מחשבים שכנים. העברת החבילה נעשית על סמך כתובת MAC ייחודית לכל רכיב רשתי (אשר צרובה בכרטיס הרשת).

עבור כל חבילה שמתקבלת, המתג מבצע את פעולת ה-Forward (העברה הלאה) אל ה-Port המתאים לפי טבלאות מיתוג.

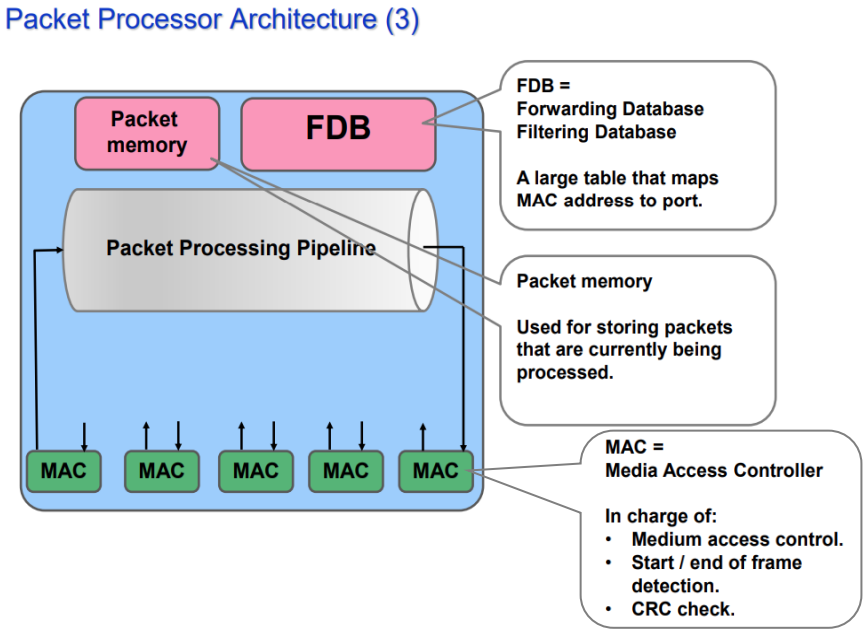
מבנה המתג:

מורכב מ-CPU, לרוב x86 או ARM כתלות ביצרן.



איור 4: מבנה המתג

כאשר כך נראה המימוש הפנימי של ה-packet processor :



איור 5 : מבנה ה-Packet Processor

שכבת ה-phy אחראית על המרה של המידע מאות אנלוגי לדיגיטלי ולהפך, והיא חלק מהשכבה הפיזית. רכיב ה-packet processor מעבד את הפקטות שנשלחות על ידי ה-pipeline ומנתב אותן ברשת.

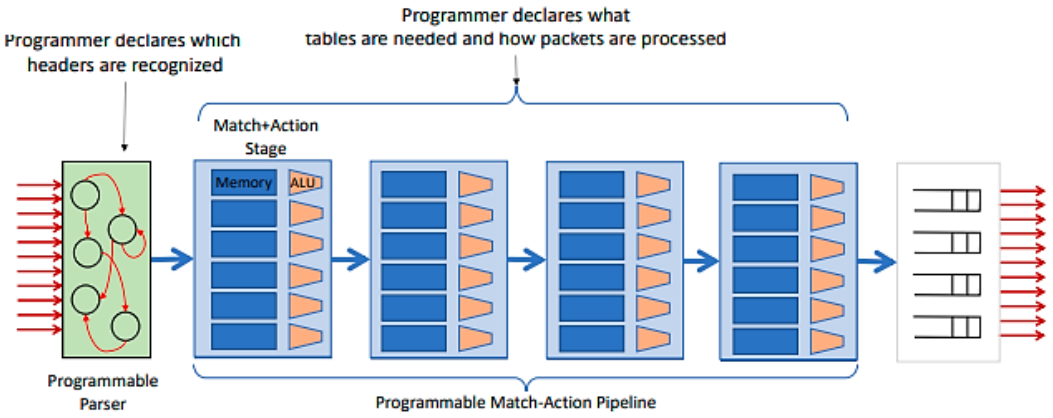
באמצעות ה-control plane ו-data plane מבצעים את הניתוב :

Data plane – אחראי על עיבוד המידע של הפקטות הכולל בין היתר העברות נתונים, העתקות מידע, פעולות שמתבצעות בעת שליחה וקבלה של מידע.

Control plane – אחראי על תהליכים שמתרחשים פעם אחת, למשל setup, ותהליכים שמתרחשים בעליית הרשת ובעת קינפוג המערכת.

בפרויקט נעשה שימוש בארכיטקטורת PISA (Protocol Independent Switch Architecture) אשר מבצעת Forwarding לחבילות באמצעות Pipeline. הארכיטקטורה ממושמשת במתגים חכמים הניתנים לתכנות, במטרה לאפשר ניתוב עצמאי מפרוטוקול בעזרת איסוף מידע על הפקטות שעוברות ברשת. כמו כן, הארכיטקטורה אינה תלויה בחומרה, ובעלת יכולות פרסור וחישוב על פקטות ברשת. באמצעות כך המתג יכול לאסוף מידע, למשל מספר הפקטות שעוברות בזמן מסוים, ולקבל החלטות ניתוב לאחר עיבודו ב-controller, וכן, ליצור סטטיסטיקות על התעבורה.

מימוש הארכיטקטורה:



איור 6 : ארכיטקטורת PISA

הפקטות שמתקבלות נכנסות מצד שמאל באיור הארכיטקטורה, ומשם עוברות דרך parser. לאחר מכן, עוברות דרך יחידות עיבוד ובסוף ה-pipeline יש יחידות שאוספות את המידע (ומאחדות מחדש על פי הצורך).

פקטה מתקבלת למתג כרצף של ביטים, ה-parser אחראי להבין מה מייצג הרצף. ה-parser ממומש כמכונת מצבים ומחלק את הפקטה לחלקים, מתוכם ייבחר המידע הרלוונטי עליו יתבצע חילוץ, על מנת לקבל את המידע הדרוש. חבילות זורמות באופן קבוע, ולכן מתקבל latency משוער וקבוע במערכת. ה-header יכול להשתנות (בהתאם לפעולה שרוצים לבצע).

הארכיטקטורה היא מסוג RMT (reconfigurable match action tables), ובפרט ניתן לתכנת את ה-parser. ה-headers מתוך הפקטות עוברים תהליך עיבוד דרך יחידות match-action. בשלב זה, המתג יכול לשנות, להוסיף, לעדכן ולבצע פעולות חישוב על המידע שהוצא מה-parser. בסוף התהליך, מאחדים חזרה את המידע המעובד לפקטה באמצעות ה-de-parser, והפקטה החדשה נשלחת הלאה לגורם הבא.

בכל שלב ניתן לבצע מספר פעולות במקביל. ניתן לשמור עבור מצב מסוים זיכרון מקומי, כך שכל הפקטות שעוברות במתג יוכלו לגשת למידע המקומי. בנוסף, ניתן להשתמש בזיכרון זה על מנת לבצע חישובים, באופן דינאמי תוך כדי כניסת הפקטות. אין סריאליזציה בקבלת הפקטות, הן יכולות "לראות" מצב זהה (למשל של ערכי משתנים) וייתכן שיתקבל data race, ויראו ערכים שונים בין הפקטות. אך, קיימות פעולות אטומיות שיכולות להבטיח שאם דרוש ביצוע פעולות במקביל, הן יתבצעו באופן נכון.

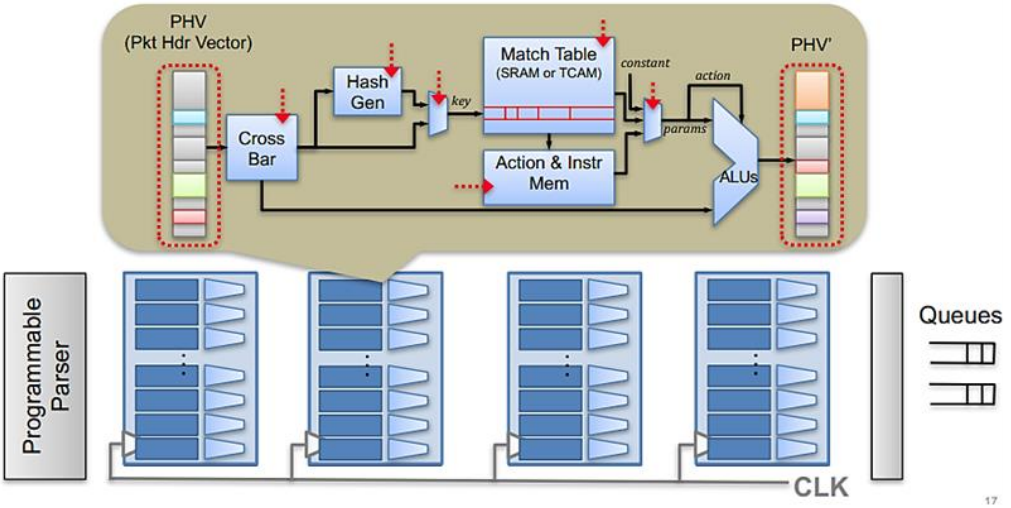
ה-state שמור בתוך ה-match table. ניתן לקחת את הזיכרון, לקרוא אותו, לבצע פעולה אטומית כלשהי ולכתוב חזרה לזיכרון כך שהפקטה הבאה שתכנס תוכל לראות את המצב המעודכן.

בארכיטקטורה של PISA יחידת העיבוד הבסיסית נקראת match-action. יחידה זו מבצעת match על מפתח ספציפי לחיפוש, וכתוצאה מכך מחליטה לאיזה action לגשת ואותו היא מבצעת. זו טבלה אשר מגדירה לכל מפתח (מבצעת match), ערך action ספציפי. בעזרת מפתח ניתן לגשת לטבלה לרשומה המתאימה ובתוכה לבצע את ה-action המתאים.

Action הוא ביצוע פעולה אריתמטית, חיבור חיסור וכו'.

נוכל להגדיר ולקבוע את הערכים בטבלה בהתאם לפעולות שנרצה לבצע. בנוסף, רשומות הטבלה הן דינמיות ומשתנות בזמן הריצה.

Cross bar הוא מעין mux שמסתכל על ה-PHV (packet header vector, וקטור אשר כולל את ה-headers וה-metadata לאורך ה-pipeline) ולוקח משם את המידע הדרוש לו. יש רגיסטרים שנכנסים לטבלה (PHV) ויוצאים ממנה ('PHV).



איור 7: Match-Action stage

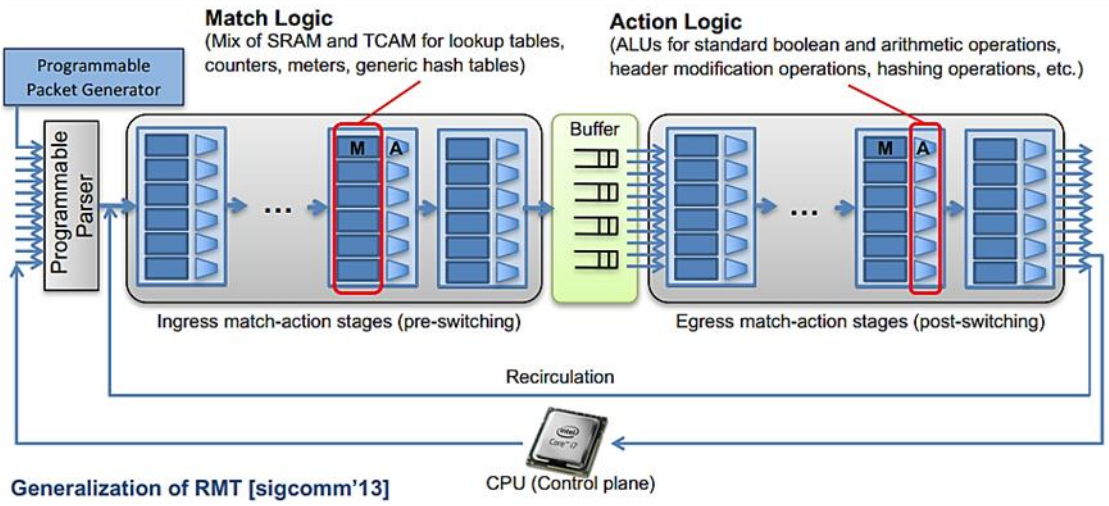
ה-header and metadata bass זו הכניסה לתוך המצב. מתוכה ניתן לקחת ביטים מסוימים, ולהשתמש בהם על מנת לבנות מפתח (בארכיטקטורה זו משתמשים ב-3 שדות שונים על מנת להרכיב את המפתח). כאשר משרשרים את שלושת השדות ששייכים ל-header ול-metadata מקבלים את המפתח.

ה-control הוא הגורם אשר מאכלס את הטבלה של match-action. לאחר קבלת המפתח ניתן לבצע חיפוש על מנת למצוא את רשומת המידע המתאימה, כך שמתקבלות שתי תוצאות:

1. Miss – המפתח לא נמצא, ולכן יש להשתמש בפעולת ברירת המחדל.
2. Hit – חיפוש המפתח החזיר תוצאה, וכעת ניתן לגשת לרשומה המתאימה.

המידע נכנס ל-mux, עובר דרך selector כאשר הביטים עוברים דרכו, ולאחר מכן מורץ ה-action. ה-action יכול לקבל מידע, וגם להשתמש בשדות אחרים שמוגדרים על ה-bus (כמו

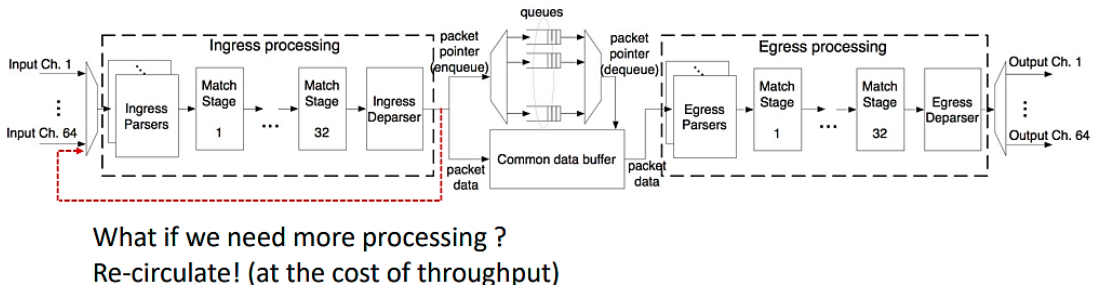
פרמטרים נוספים) או לתת פרמטרים שה-control מגדיר. התוצאה נכתבת ל-bus של המוצא, וכך המצב הבא יוכל לקרוא אותה. בנוסף, ניתן לכתוב את ה-action ולהחליט על הלוגיקה כי חלק זה ניתן לתכנות במתג.



איור 8 : Match-Action unit

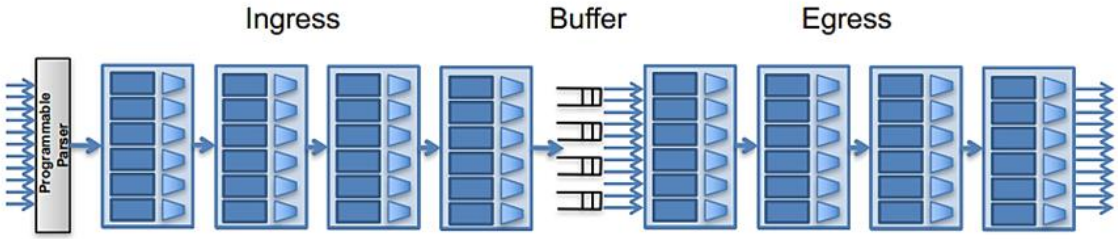
הפקטה עוברת בין המצבים ומפעילה פעולות שרלוונטיות למצב בו נמצאת. על מנת שהארכיטקטורה תעבוד, יש לבטא את התהליך (ה-Flow) כמעבר בין המצבים, והחלטה מהי הפעולה שיש לבצע בכל מצב לפי המידע שהתקבל בו. ההחלטה מה לעשות נקבעת לפי המידע שעובר במצב. המידע יוצר תגובת match, ולפיו נקבעת הפעולה action שתתבצע, דבר שיכול לגרום לשינוי המידע והמשך לשינוי ה-flow של המידע.

למתג שתי יחידות עיבוד:

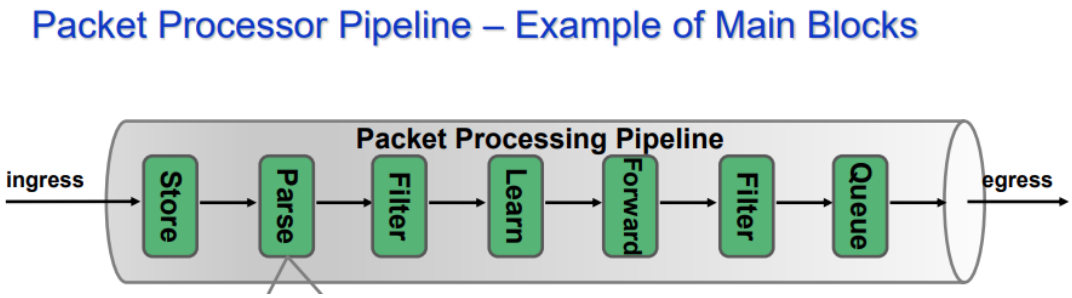


איור 9 : Ingress and Egress processing

PISA עובדת בצורת שני control-ים מרכזיים: ingress ו-egress, ביניהם יש את ה-buffer המכיל את התורים של הפקטות. בשלב הראשון, פקטה נכנסת לשלב ה-ingress ובשלב השני, נכנסת ל-egress. מתג מקבל פקטות מפורט ספציפי, וצריך להחליט לאן לשלוח אותם. פעולה זו מתבצעת ב-ingress. כל control מכיל יחידות של match-action.



איור 10: Ingress and egress controls



איור 11: Packet Processor Pipeline

ייתכן שיידרש עיבוד לפני הכניסה לתורים מפני שלכל פורט יש תור משלו. מצב בו נחסמת שליחת פקטות לפורט מסוים בגלל שפורטים אחרים עמוסים אינו רצוי. העיבוד מתרחש בסוף מכיוון שייתכן שיהיה צורך במידע שמחושב רק בסוף התהליך.

אם תהליך העיבוד של המידע לא הסתיים ודרוש תהליך עיבוד נוסף (מעבר לעיבוד שמתבצע ב-ingress וב-egress), וכן ה-pipeline הסתיים, אז יבוצע שימוש ב-recirculation (רסירקולציה) לפקטה, אשר תאפשר את הכנסת הפקטה מחדש לתחילת ה-pipeline ולבצע המשך עיבוד עליה. תוספת זו פוגעת ב-throughput של המערכת, אך מאפשרת לבצע פעולות עיבוד מורכבות יותר.

אלגוריתם QPipe

תיאור

מטרת האלגוריתם היא להחזיר מערך המורכב שכבות אשר כל שכבה קטנה פי 2 מהשכבה שלפניה, בעזרתו ניתן לבצע קוונטיזציה של stream עבור התעבורה ברשת.

שלבים באלגוריתם:

1. דגימה – האלגוריתם מבצע דגימה של החבילות בהסתברות מסוימת.
2. דחיסה – אם החבילה נדגמה:
    a. אם השכבה אינה מלאה, ערך החבילה נכנס לשכבה הנוכחית.
    b. אחרת, השכבה מלאה:
        i. מוצא את הערך המינימלי הראשון במערך.
        ii. מוצא את הערך המינימלי השני במערך.
        iii. בוחר באופן רנדומלי את אחד מהערכים ומעלה אותו לשכבה הבאה.

פירוט השלבים:

1. אחת מכל K חבילות העוברות ברשת נדגמת באלגוריתם באופן הסתברותי, זאת על מנת לקבל מדגם מייצג באופן טוב יותר של אופי הרשת. לרוב ברשתות ובמיוחד במתגים, כמות התעבורה שעוברת הינה גדולה מאוד, ולכן רצוי לבחור K אשר ישקף את אופי התעבורה.
2. ממלאים את השכבה הראשונה של המערך מהחבילות שנדגמו. לאחר שהשכבה מלאה, מתחיל שלב הדחיסה.
3. על מנת למצוא את שני הערכים המינימליים בשכבה, נדרש לסרוק את כל השכבה פעם אחת. משתמשים ב-"worker packets",שהן חבילות העוברות ברשת ויכולות להעביר מידע על ידי ביצוע של רסירקולציה (שליחה מחדש בתוך המתג). ישנם רגיסטרים השומרים את ערך האיבר המינימלי הראשון והאינדקס שלו אשר מכונה beta, וכן רגיסטרים השומרים את ערך האיבר המינימלי השני והאינדקס שלו אשר מכונה gamma. מאתחלים רגיסטר אשר מכונה filter_index לראש השכבה ומקדמים אותו עבור כל worker packet באחד. הרגיסטר filter_index בודק אם ערך האיבר הנוכחי קטן או שווה לערך השמור ברגיסטר beta. אם כן, האיבר החדש תופס את מקומו של beta והערך ששמור ב-beta עובר להיות הערך המינימלי השני ונשמר ב-gamma. ההחלפה מתבצעת בהתאם גם באינדקסים. אם לא, ממשיכים לאיבר הבא במערך, כך שלאחר מעבר על כל השכבה,beta מכיל את האיבר המינימלי ו-gamma מכיל את האיבר המינימלי השני.
4. נבחר באופן רנדומלי איבר אחד מהשניים המינימליים על מנת להעלות אותו לשכבה הבאה. את האיבר אשר לא מעלים לשכבה הבאה מוחקים. כמו כן, ערך האיבר המקסימלי מבין השניים יישמר ברגיסטר המכונה theta על מנת שבחיפוש הבא המתבצע באופן דומה לנקודה הקודמת ייבדק התנאי כך שיתקבלו שני האיברים המינימליים בשכבה אשר גדולים מ-theta.

על מנת למחוק את האיברים מהשכבה ולצמצם את החיפוש, מתבצעת החלפה של האיברים הראשונים בשכבה עם האיברים המינימליים שנמצאו, כך שבמעבר הבא יהיו 2 איברים פחות לבדוק.

5. כאשר השכבה מלאה, עוברים לשכבה הבאה עד אשר מגיעים לסוף המערך ומתקבל מערך דחוס בגודל שנקבע מראש.

דוגמת דחיסה

דוגמת הרצה עבור דחיסה של שתי שכבות בהנחה שהתקבלו החבילות בעלות הערכים הבאים בשכבה הראשונה:

4	1	16	55	63	81	44	11	23	77	3	22

כעת יש למצוא את שני האיברים המינימליים במערך:

4	1	16	55	63	81	44	11	23	77	3	22

נבצע החלפה של שני האיברים המינימליים עם שני האיברים הראשונים. החלפה ראשונה:

1	4	16	55	63	81	44	11	23	77	3	22

החלפה שנייה:

1	3	16	55	63	81	44	11	23	77	4	22

באופן רנדומלי אחד מהם יעלה לשכבה השנייה והשני יימחק:

1	3	16	55	63	81	44	11	23	77	4	22

3					

השכבה השנייה אינה מלאה, ולכן יש למצוא את שני האיברים המינימליים הגדולים מ-3 בשכבה הראשונה ולהעלות אחד מהם לשכבה השנייה:

1	3	16	55	63	81	44	11	23	77	4	22

3					

לאחר ההחלפה וההעלאה:

1	3	4	11	63	81	44	55	23	77	16	22

3	4				

מציאת המינימליים:

1	3	4	11	63	81	44	55	23	77	16	22

3	4				

לאחר ההחלפה וההעלאה:

1	3	4	11	16	22	44	55	23	77	63	81

3	4	16			

מציאת המינימליים:

1	3	4	11	16	22	44	55	23	77	63	81

3	4	16			

לאחר ההחלפה וההעלאה:

1	3	4	11	16	22	23	44	55	77	63	81

3	4	16	44		

1	3	4	11	16	22	23	44	55	77	63	81

3	4	16	44		

לאחר ההחלפה וההעלאה:

1	3	4	11	16	22	23	44	55	63	77	81

3	4	16	44	55	

מציאת המינימליים:

1	3	4	11	16	22	23	44	55	63	77	81

3	4	16	44	55	

לאחר ההחלפה וההעלאה:

1	3	4	11	16	22	23	44	55	63	77	81

3	4	16	44	55	77

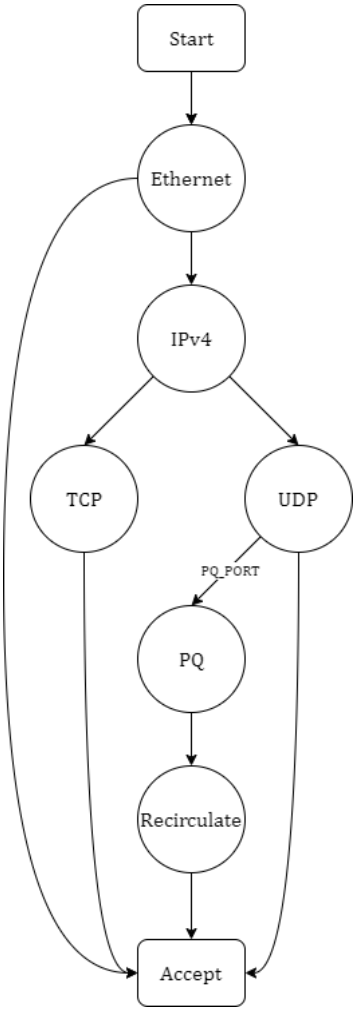
כעת השכבה מלאה ושלב הדחיסה נגמר.

מבנה הקוד

הקוד נכתב בשפה P4, שהיא שפת תכנות לשליטה באופן העיבוד והעברה של חבילות בהתקני רשת, כגון נתבים, מתגים והתקני רשת הניתנים לתכנות. זו שפה אשר מיועדת עבור תכנות ל-data plane. השפה עובדת בצורת pipeline, היא מאפשרת למתכנת לגשת לרגיסטרים בהתקן הרשת ולשלוט בזיכרון, לבצע פעולות חישוב, ועל פיהן לנתב את החבילות ברשת. השפה אינה תלויה בפרוטוקול מסוים או במידע על אודות החומרה ומקמפלת את התוכנית עבור המכשיר אליו משויכת התוכנית, ולכן היא דינמית ומאפשרת לעבד בצורה רחבה את החבילות.

Parser:

הפרסר הינו מכונת מצבים סופית אשר בודק את הבתים המגיעים כחבילה ומחלץ header-ים מהם לפי שכבות ופרוטוקולים. לכל שלב בתהליך הפרסור יש אפשרות לקבל את החבילה, לדחות אותה או לבצע transition, כלומר להמשיך לפרסר את ה-header הבא. תחילה הפרסר בודק את ה-header עבור Ethernet ואם מדובר בחבילת IP, הוא מפרסר את ה-header שלה ובודק באיזה פרוטוקול מדובר, TCP או UDP. לבסוף, הפרסר בודק האם החבילה עברה רסירקולציה, כלומר האם מדובר בחבילה חדשה מהרשת או חבילה שנשלחה מתוך הרשת שוב.



איור 12 : מכונת מצבים סופית של הפרסר

Headers:

בחלק קוד זה מוגדרים כל השדות הרלוונטיים לכל header אשר מפורסר מהחבילה. למשל עבור ה-header של ה-IP נמצא שדות כמו גרסה, פרוטוקול, TTL וכו'.

בנוסף, ישנו Struct עבור ה-metadata אשר דרוש עבור החישובים באלגוריתם. כל שדה מוגדר תחילה לפי כמות הביטים שהוא דורש, ולאחר מכן מוגדר השם באופן הבא:

```
struct meta_t {
    bit<16> recirc_flag;
    bit<32> sample;
    bit<1>  sample_01;

    bit<1> recirced;

    bit<32> head;
    bit<32> head_n;
    bit<32> tail;
    bit<32> tail_n;
    bit<32> len;
    bit<32> item_num;
    bit<32> left_bound;
    bit<32> right_bound;
    bit<32> array_to_operate;
    bit<32> busy;
    bit<32> option_type;

    bit<32> theta;
    bit<32> beta;
    bit<32> gamma;
    bit<32> filter_index;
    bit<32> filter_index_n;
    bit<32> filter_item;
    bit<32> delete_index;
    bit<32> delete_index_n;
    bit<32> filter_item_2;
    bit<32> old_beta;
    bit<32> old_beta_index;
    bit<32> max_v;
    bit<32> index_beta;
    bit<32> index_gamma;
    bit<32> to_delete_num;
    bit<32> to_delete_num_n;
    bit<32> head_v;
    bit<32> coin;
    bit<32> picked_value;
    bit<32> value;
    bit<32> a_value;
}
```

שדות חשובים:

`sample_01` מייצג האם החבילה עולה לשכבה הבאה או נמחקת.

`recirced` מייצג האם מדובר בחבילה חדשה או חבילה שעברה רסירקולציה.

theta, beta, gamma מייצגים את הערכים המינימליים שנמצאו במעבר על השכבה לקראת ההעלאה לשכבה הבאה.

בנוסף, יש שדות המייצגים את אורך המערכים והאינדקסים עליהם עוברים.

Control:

מקבל חבילה ומבצע את הפעולות האפשריות באמצעות match-action. ה-control מכיל Tables ו-Actions ואחראי על העדכון שלהם. בדומה לפונקציות, כל Control אחראי על ביצוע פעולה קטנה למשל השוואה בין שני רגיסטרים והשמת הערך ברגיסטר אחר.

דוגמא ל-Control המבצע העלאה ב-1 של רגיסטר tail המייצג את הזנב של כל שכבה באלגוריתם:

```
control inc_tail (inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    // inc_tail
    apply {
        tail_register.read(meta.meta.tail,
                           (bit<32>)meta.meta.array_to_operate);
        if (meta.meta.tail == meta.meta.right_bound) {
            meta.meta.tail_n = meta.meta.left_bound;
        }
        else {
            meta.meta.tail_n = meta.meta.tail + 1;
        }
        tail_register.write((bit<32>)meta.meta.array_to_operate,
                            (bit<32>)meta.meta.tail_n);
    }
}
```

תחילה, מתבצעת קריאה של ערך מהרגיסטר tail בשכבה בה נמצא האלגוריתם לתוך שדה meta אשר עוזר לביצוע החישובים. לאחר מכן מעלים את הערך ב-1 אם הוא לא בקצה המערך, אחרת, הערך יהיה הגבול השמאלי. לבסוף כותבים חזרה את הערך לרגיסטר במקום המתאים.

Control מרכזי שישנו בקוד הוא ingress שאחראי על הניתוב ובפועל מנהל את כל האלגוריתם. הוא מורכב משלושה שלבים מרכזיים:

1. ביצוע דגימה של החבילות המגיעות, ומילוי המערך בחבילות הנדגמות.
2. איתור שני המספרים המינימליים במערך.
3. מחיקה של אחד המספרים והעלאת המספר השני לשכבה הבאה.

כל החישובים לצורך האלגוריתם מתבצעים באמצעות קריאה וכתיבה לרגיסטרים ושימוש ב-metadata.

לאחר שמתבצעים כל החישובים, החבילה מורכבת מחדש באמצעות ה-egress pipeline, מחשבים את ה-checkSum של החבילה, ה-de-parser מרכיב את ה-header-ים המתאימים ומתבצע ניתוב או רסירקולציה בהתאם.

Match-Action:

בנוסף לקוד האלגוריתם ב-P4 ישנו גם קובץ json המתאר את הפעולות match-action עבור המתג. הקובץ כולל את ה-table entries עבור האלגוריתם או היישום.

עבור כל רשומה מצוינת הטבלה אליה הרשומה שייכת, השדה עליו צריכה להתבצע פעולת ה-match, פעולת ה-action המתאימה, ופרמטרים נוספים שרלוונטיים עבור פעולת ה-action.

לדוגמה:

```
"table_entries": [
  {
    "table": "ingress.ipv4_route",
    "default_action": true,
    "action_name": "ingress.drop",
    "action_params": { }
  },
  {
    "table": "ingress.select_route",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
    },
    "action_name": "ingress.set_select",
    "action_params": {
      "direction": 0
    }
  },
```

הרשומה הראשונה מייצגת פעולת ברירת מחדל עבור טבלת ipv4_route.

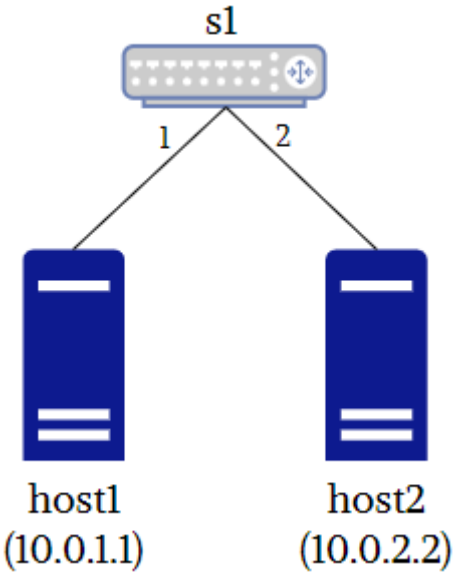
הרשומה השנייה שייכת לטבלת select_route ומייצגת פעולת match עבור כתובת ה-IP של היעד. אם מדובר ב-host 1 אז ה-direction הנבחר הוא 0 אשר מתקבל כפרמטר עבור פעולת set_select שהיא פעולת ה-action המתאימה לטבלה.

מוגדר control flow כך שהקומפיילר ייקח את קוד ה-P4 ויקמפל אותו ליעד המתאים ב-switch. על מנת לעדכן את הלוגיקה עצמה מחדש, ניתן לבצע זאת על ידי כתיבה וקמפול מחדש. לעומת זאת, ה-Flow של הקוד יכול להשתנות בזמן הריצה באופן דינמי, למשל עדכון הטבלה על ידי הוספת IP שלא הופיע עד כה.

טופולוגיה:

קובץ json גם משמש לתיאור הטופולוגיה של הרשת עבור ה-mininet. ניתן להגדיר בו את הטופולוגיה של הרשת ולשנות אותה בצורה דינמית מבלי להתערב בקוד האלגוריתם, כלומר האלגוריתם יעבוד עבור כל טופולוגיה של רשת תקינה.

הטופולוגיה שנבחרה עבור האלגוריתם:



איור 13: טופולוגיה של האלגוריתם

קובץ ה-json המבטא טופולוגיה זו:

```
"hosts": {
        "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
               "commands":["route add default gw 10.0.1.10 dev eth0",
                           "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
        "h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
               "commands":["route add default gw 10.0.2.20 dev eth0",
                           "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]}
    },
    "switches": {
        "s1": { "runtime_json" : "pod-topo/s1-runtime.json" }
    },
    "links": [
        ["h1", "s1-p1"], ["h2", "s1-p2"]
    ]
```

בהתאמה לאיור הטופולוגיה, ניתן לראות שני hosts עם כתובות IP ו-MAC מוגדרות, אשר מחוברים בקישורים למתג s1 בפורטים המתאימים.

Python Scripts:

קיימים מספר קבצי הרצה של python אשר משתמשים ב-scapy על מנת לייצר תעבורה ברשת ולהעביר אותה בין תחנות הקצה.

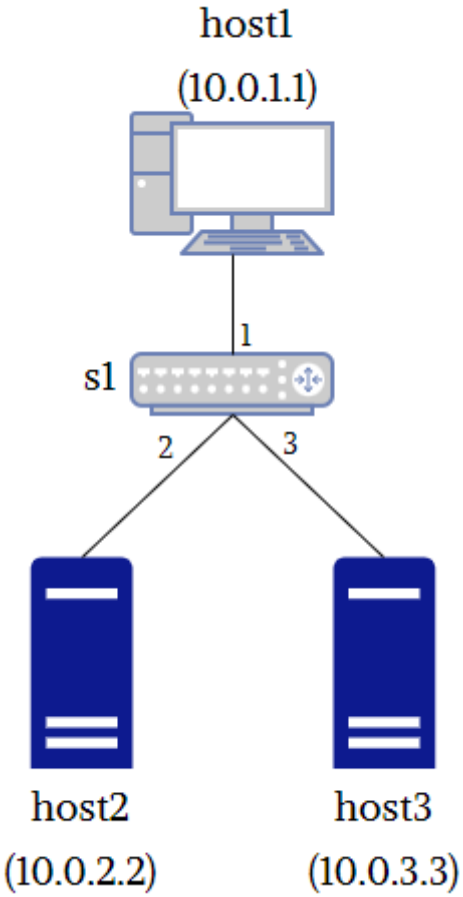
הקובץ send.py אחראי על שליחת הפקטות ומקבל מספר פרמטרים : כתובת IP של היעד, הודעה, פרוטוקול וזמן שליחה.

הקובץ receive.py מאזין ל-interface ומקבל בו חבילות.

יישום

נכתב ב-P4 יישום בסיסי המשלב את האלגוריתם ובעזרתו מקבל החלטות ניתוב. היישום מנתב חבילות ברשת ומפעיל את האלגוריתם. לאחר מכן, מקבל ממנו את החציון עבור stream החבילות שהתקבל עד כה. חבילות הגדולות מהחציון מנותבות לשרת אחד, ואילו חבילות הקטנות מהחציון מנותבות על ידי המתג לשרת אחר.

הטופולוגיה של היישום:



איור 14: טופולוגיה של היישום

מקרה כזה מתאים לביזור עומסים על שרת מסוים, למשל חברות אשר מקבלות פניות מלקוחות רבים כמו Google, ירצו להשתמש במספר רב של שרתים. הלקוחות שירצו להתחבר לאתר מכירים את כתובת ה-IP שלו ופונים לשרת המרכזי. באמצעות שימוש באלגוריתם QPipe וניטור דפוסי התנהגות מסוימים, המתג יכול לקבל החלטה לנתב את החבילות לשרת נוסף אשר מטפל בפניות לאתר, ובכך להקל את העומס על השרת המרכזי באופן השקוף ללקוחות.

מקרה נוסף אשר היישום יכול לעזור בו הינו ניתוב חבילות החשודות כזדוניות על סמך פרטים מסוימים ב-header לשרת נוסף אשר יהווה DMZ בין רשת פנימית לחיצונית. השרת יבדוק את תוכן החבילה ויעביר אותה הלאה רק אם החליט שהיא תקינה, אחרת יזרוק אותה.

דוגמת הרצה

התוכנה מורצת באמצעות mininet. טווח הערכים המוגרלים לחבילות הינו 300-700.

תחילה יש לאתחל את המתג עם פעולות ה-Match-Action המתאימות:

```
p4@p4:/media/sf_Project_A/A_Qpipe16/exercises/QPipeRouting$ make
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files build/qpipeRouting.p4.p4info.txt -o build/qpipeRouting.json qpipeRouting.p4 --Wdisable
[--Wwarn=invalid] warning: no user metadata fields tagged with @field_list(0)
sudo python3 ../../utils/run_exercise.py -t pod-topo/topology.json -j build/qpipeRouting.json -b simple_switch_grpc
Reading topology file.
Building mininet topology.
Configuring switch s1 using P4Runtime with file pod-topo/s1-runtime.json
 - Using P4Info file build/qpipeRouting.p4.p4info.txt...
 - Connecting to P4Runtime server on 127.0.0.1:50051 (bmv2)...
 - Setting pipeline config (build/qpipeRouting.json)...
 - Inserting 7 table entries...
 - ingress.ipv4_route: (default action) => ingress.drop()
 - ingress.select_route: hdr.ipv4.dstAddr=['10.0.1.1', 32] => ingress.set_select(direction=0)
 - ingress.select_route: hdr.ipv4.dstAddr=['10.0.2.2', 32] => ingress.set_select(direction=1)
 - ingress.select_route: hdr.ipv4.dstAddr=['10.0.3.3', 32] => ingress.set_select(direction=1)
 - ingress.ipv4_route: meta.meta.selected=0 => ingress.set_egress(egress_spec=1, dmac=08:00:00:00:01:11)
 - ingress.ipv4_route: meta.meta.selected=1 => ingress.set_egress(egress_spec=2, dmac=08:00:00:00:02:22)
 - ingress.ipv4_route: meta.meta.selected=2 => ingress.set_egress(egress_spec=3, dmac=08:00:00:00:03:33)
s1 -> gRPC port: 50051
**********
h1
default interface: eth0 10.0.1.1        08:00:00:00:01:11
**********
**********
h2
default interface: eth0 10.0.2.2        08:00:00:00:02:22
**********
**********
h3
default interface: eth0 10.0.2.2        08:00:00:00:03:33
**********
Starting mininet CLI
```

איור 15: אתחול המתג

ניתן לראות שהרשומות בטבלה כוללות בחירת הכיוון של החבילה, האם היא יוצאת מ-host1 אל השרתים host2, host3 או ההפך. לאחר בחירת הכיוון, בודקים אם ערך החבילה הנבדק גדול או קטן מהחציון ובהתאם לכך מנתבים לשרת המתאים.

host1 שולח חבילות ל-host2:

```
"Node: h1"
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

איור 16: host1 – command prompt

host2 ו-host3 מקשיבים וניתן לראות שרק host2 מקבל חבילות שכן האלגוריתם רץ למציאת החציון:

איור 17: host2, host3 – command prompt

החציון שהתקבל הינו 505, וכעת גם host3 מתחיל לקבל חבילות בהתאם לניתוב:

איור 18: host2, host3 – command prompt

חבילות הגדולות מהחציון מועברות ל-host2, וחבילות הקטנות ממנו עוברות ל-host3.

ביצוע הפרויקט

סביבת עבודה

הפרויקט התבצע בסביבת עבודה של virtual machine, שבה סביבת Ubuntu גרסה 20.04 אשר תומכת בשפה-P4. לצורך הרצת האלגוריתם והיישום יש צורך במתג חכם כדוגמת Barefoot (Nvidia) או Tofino (Intel) ורכיבי רשת פיזיים, או לחילופין במודל המחקה את המתג ורכיבי רשת נוספים.

אופן הרצה

יש להוריד את הקוד של האלגוריתם מ-GitHub:

https://github.com/SivanVe/A_Qpipe16.git

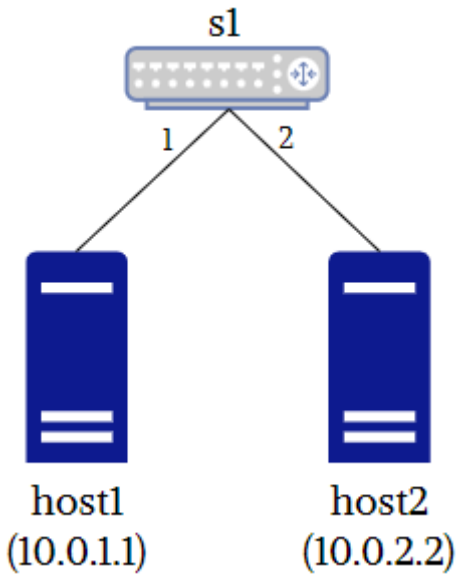
לאחר מכן, לקמפל את הקוד בטרמינל על ידי הפקודה:

```
make
```

כעת נפתח CLI של mininet המדמה רשת וירטואלית לפי הנתונים שנכתבו בקבצי ה-json, בה ניתן להעביר פקטות בין רכיבי רשת.

הטופולוגיה של הרשת:

הטופולוגיה כוללת 2 host-ים אשר מתקשרים ביניהם דרך מתג יחיד.



איור 19: הטופולוגיה של האלגוריתם

mininet:

```
p4@p4:/media/sf_Project_A/A_Qpipe16/exercises/QPipe$ make
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files build/qpipe.p4.p4info.txt -o build/qpipe.json qpipe.p4
Wdisable
[--Wwarn=invalid] warning: no user metadata fields tagged with @field_list(0)
sudo python3 ../../utils/run_exercise.py -t pod-topo/topology.json -j build/qpipe.json -b sim
e_switch_grpc
Reading topology file.
Building mininet topology.
Configuring switch s1 using P4Runtime with file pod-topo/s1-runtime.json
 - Using P4Info file build/qpipe.p4.p4info.txt...
 - Connecting to P4Runtime server on 127.0.0.1:50051 (bmv2)...
 - Setting pipeline config (build/qpipe.json)...
 - Inserting 3 table entries...
 - ingress.ipv4_route: (default action) => ingress.drop()
 - ingress.ipv4_route: hdr.ipv4.dstAddr=['10.0.1.1', 32] => ingress.set_egress(egress_spec=1,
mac=08:00:00:00:01:11)
 - ingress.ipv4_route: hdr.ipv4.dstAddr=['10.0.2.2', 32] => ingress.set_egress(egress_spec=2,
mac=08:00:00:00:02:22)
s1 -> gRPC port: 50051
**********
h1
default interface: eth0 10.0.1.1        08:00:00:00:01:11
**********
**********
h2
default interface: eth0 10.0.2.2        08:00:00:00:02:22
**********
Starting mininet CLI
```

```
======================================================================
Welcome to the BMV2 Mininet CLI!
======================================================================
Your P4 program is installed into the BMV2 software switch
and your initial runtime configuration is loaded. You can interact
with the network using the mininet CLI below.

To inspect or change the switch configuration, connect to
its CLI from your host operating system using this command:
  simple_switch_CLI --thrift-port <switch thrift port>

To view a switch log, run this command from your host OS:
  tail -f /media/sf_Project_A/A_Qpipe16/exercises/QPipe/logs/<switchname>.log

To view the switch output pcap, check the pcap files in /media/sf_Project_A/A_Qpipe16/exercis
/QPipe/pcaps:
 for example run:  sudo tcpdump -xxx -r s1-eth1.pcap

To view the P4Runtime requests sent to the switch, check the
corresponding txt file in /media/sf_Project_A/A_Qpipe16/exercises/QPipe/logs:
 for example run:  cat /media/sf_Project_A/A_Qpipe16/exercises/QPipe/logs/s1-p4runtime-reques
.txt

mininet> xterm h1 h2
```

איור 20: אתחול המתג

ניתן לראות באיורים 19 ו-20 את הטופולוגיה וחוקי הניתוב בין שני ה-hosts, וכן את פתיחת ה-xterm אשר ישמשו את ה-host-ים על ידי הפקודה:

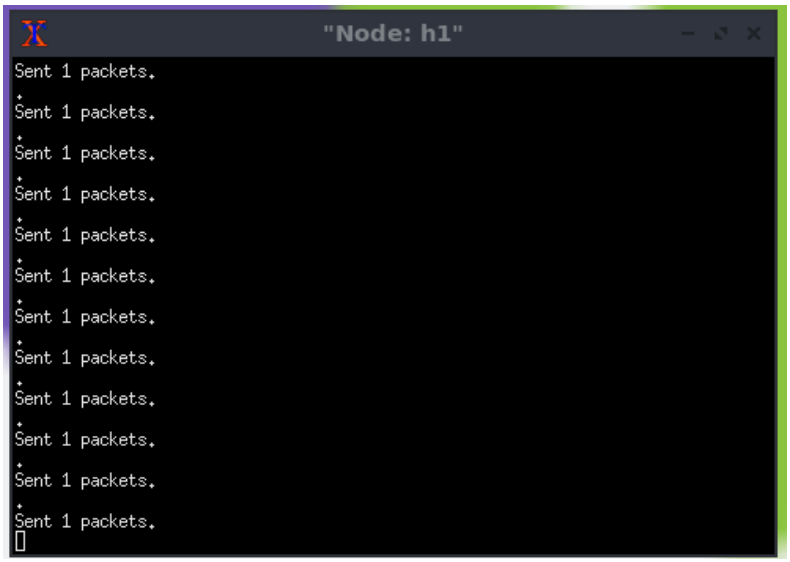
```
xterm h1 h2
```

h1 ישלח הודעות ל-h2, לכן תחילה יש לאפשר ל-h2 להקשיב לרשת על ידי הפקודה:

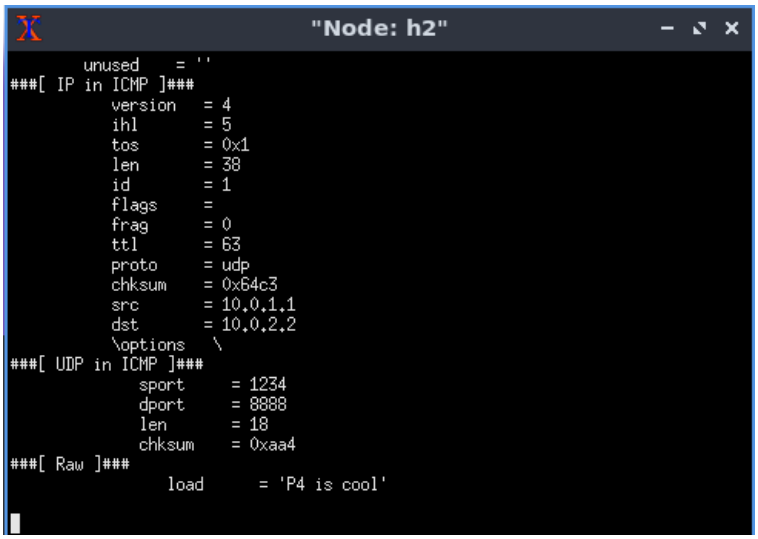
```
python3 ./receive.py
```

לאחר מכן, ניתן לשלוח הודעות מ-h1 על ידי הפקודה (ניתן לשנות את הפרמטרים):

```
python3 ./send.py --p=UDP --des=10.0.2.2 --m="your message" --dur=500
```



איור 21: host1 – command prompt



איור 22: host2– command prompt

בדוגמה לעיל הוגדר בטרמינל של h1 לשלוח פקטות בפרוטוקול UDP, עם הודעה מסוימת. אכן ניתן לראות שהן הגיעו ליעד בטרמינל של h2 עם פרטי מידע על הפקטות. באופן זה, ניתן לייצר תעבורה.

לכל פקטה מוגדרים הנתונים: אורך, id, TTL, פרוטוקול, תוכן ההודעה שמועברת וכו'. בנוסף, מוגדר לכל פקטה ערך מסוים (value) באופן רנדומלי, עליו בוצע הניתוח הסטטיסטי של האלגוריתם.

לפני הרצת האלגוריתם יש לאתחל את הרגיסטרים באמצעות הפקודה:

```
python3 tools/runtime_CLI.py --thrift-port 9090 < reg_init
```

ניתן לשנות את גדלי השכבות על ידי שינוי אתחול הרגיסטרים.

על מנת לראות את תוכן הרגיסטרים יש להריץ את הפקודה:

```
python3 tools/runtime_CLI.py --thrift-port 9090 register_read
beta_ing_register
```

בדוגמה הזאת קוראים את התוכן של הרגיסטר המכיל את הערך המינימלי בשכבה.

תוצאות ומסקנות

הבדיקות שהתבצעו נועדו לבדוק את טיב הקוונטיזציה של QPipe לעומת היסטוגרמה, עבור גדלים שונים של זיכרון, וכן עבור stream-ים שונים של תעבורה. לצורך הסימולציה, הוגדר ערך value גנרי ורנדומלי ב-header של החבילה, אשר יכול לייצג את הגודל עליו רוצים לבצע קוונטיזציה, למשל גודל החבילה, שכיחות פרוטוקול מסוים וכדומה. כמו כן, הוגדרה טופולוגיה בסיסית בין שתי תחנות קצה (host) ומתג המקשר ביניהם, והן מעבירות תעבורה ברשת. לכל חבילה הוגדר ערך value פנימי אשר מוגרל באופן רנדומלי וניתן להחליט על הטווח של הערך.

הערך value מוגרל באופן רנדומלי מתוך טווח מסוים שניתן לקבוע מראש עבור ה-stream הנוכחי. עבור כל הרצה, חושב הממוצע והחציון של ערכי ה-value של ה-stream הנוכחי. כמו כן, התבצע שימוש גם בהיסטוגרמה, על מנת לחשב את ערכי הממוצע והחציון. וכן, הורץ האלגוריתם QPipe אשר מחזיר מערך, ממנו חושבו גם כן הממוצע והחציון. האלגוריתם הורץ מספר רב של פעמים והתבצע מיצוע על השגיאות שנבדקו.

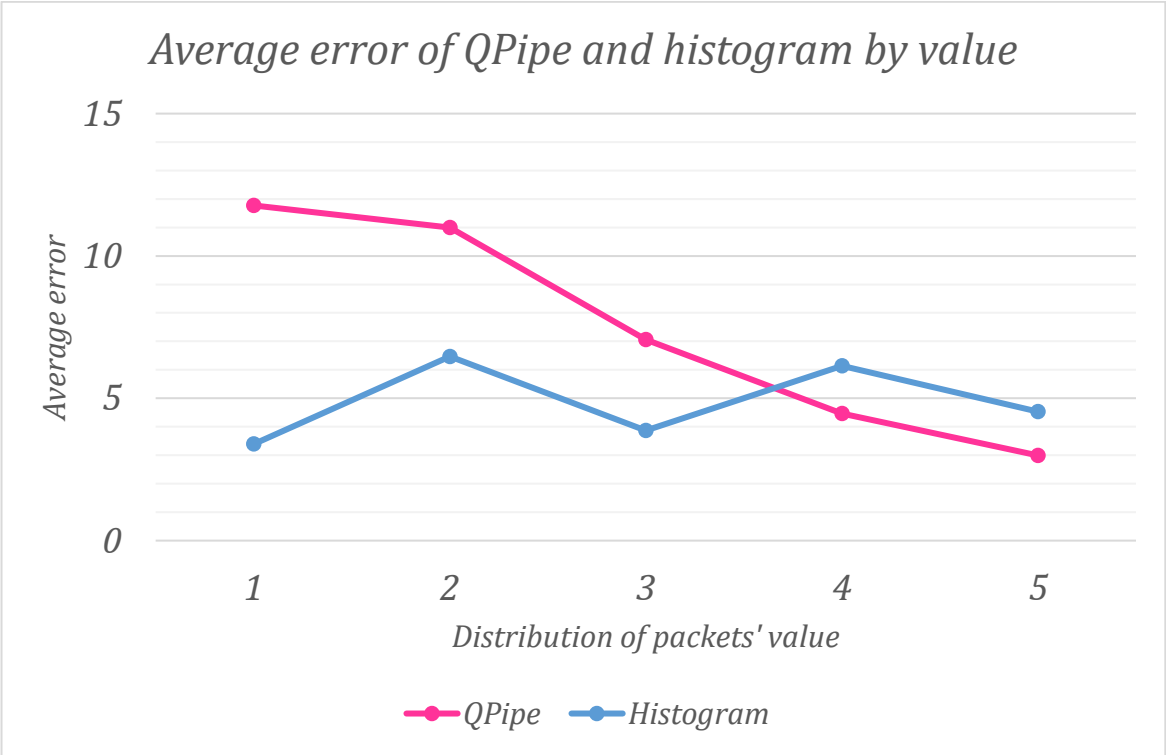
תחילה, נקבעו טווחי ערך ה-value כך שהערכים מתקבלים בין 0-1000. לפי ההנחה שככל שהתעבורה בעלת מאפיינים דומים יותר עבור הערך הנמדד, תתקבל שגיאה קטנה יותר עבור אלגוריתם QPipe.

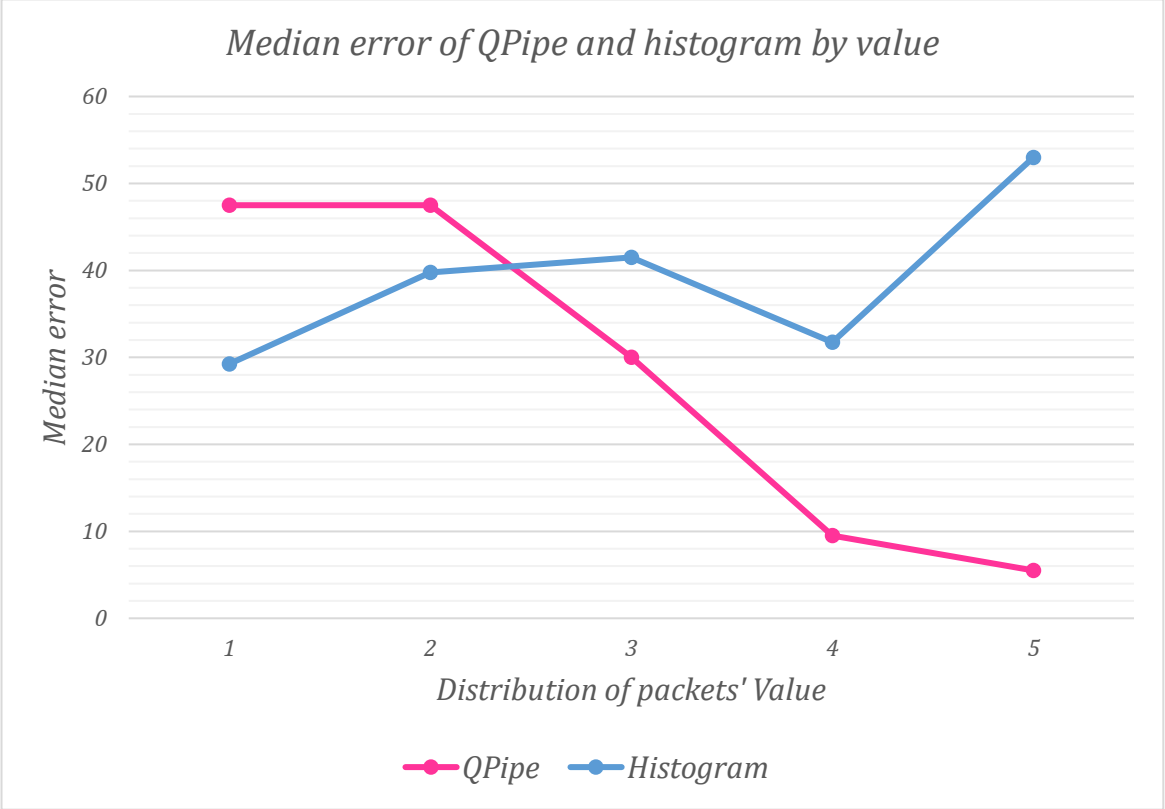
נבדקו 5 טווחים שונים:

1. 0-1000
2. 100-900
3. 200-800
4. 300-700
5. 400-600

כך שהנקודה הראשונה מייצגת את התעבורה המגוונת ביותר, עד לנקודה האחרונה המייצגת את התעבורה האחידה ביותר.

תוצאות הרצת האלגוריתם QPipe בהשוואה ליצירת היסטוגרמה עבור זיכרון של 8 תאים וטווחי תעבורה משתנים (כפי שפורט לעיל) מוצגות בגרפים הבאים:





איור 23: גרפי חציון וממוצע של השגיאה

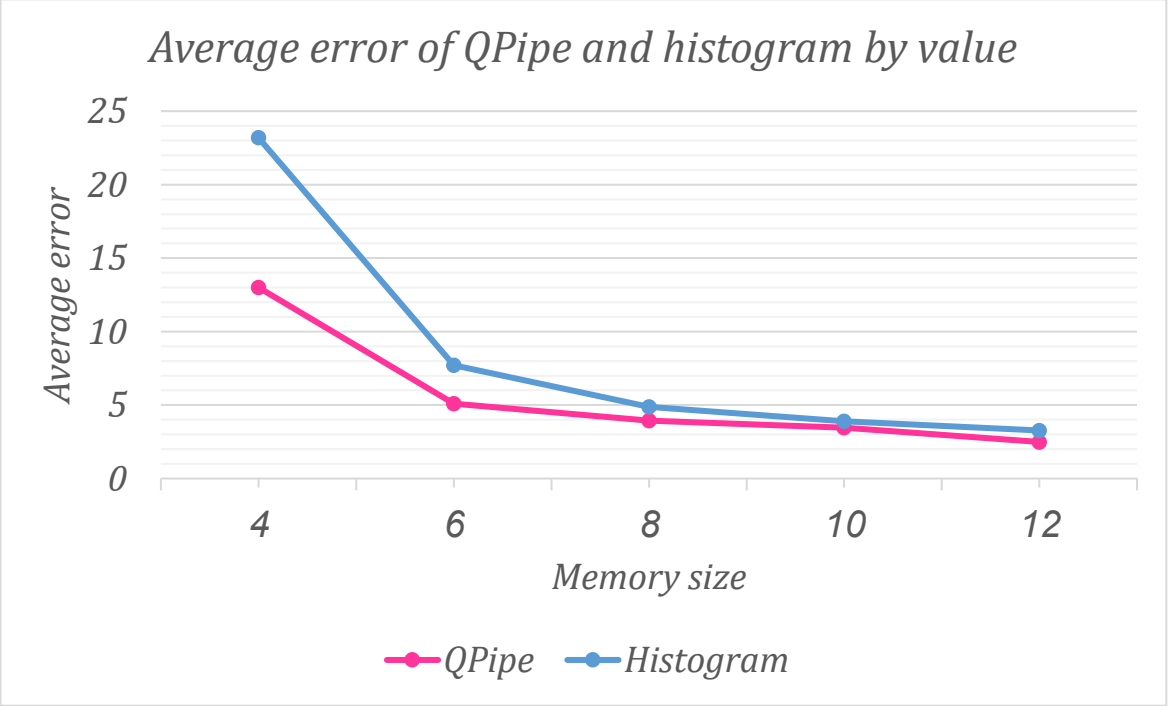
ניתן לראות עבור ה-QPipe שישנה מגמה יורדת בשגיאה הן בחציון והן בממוצע, ככל שטווח ערכי ה-value של התעבורה מצטמצם, כלומר התעבורה נהיית יותר הומוגנית. עבור ההיסטוגרמה, אין מגמה אחידה אך השגיאות נעות סביב טווח דומה עבור כל סוגי התעבורה.

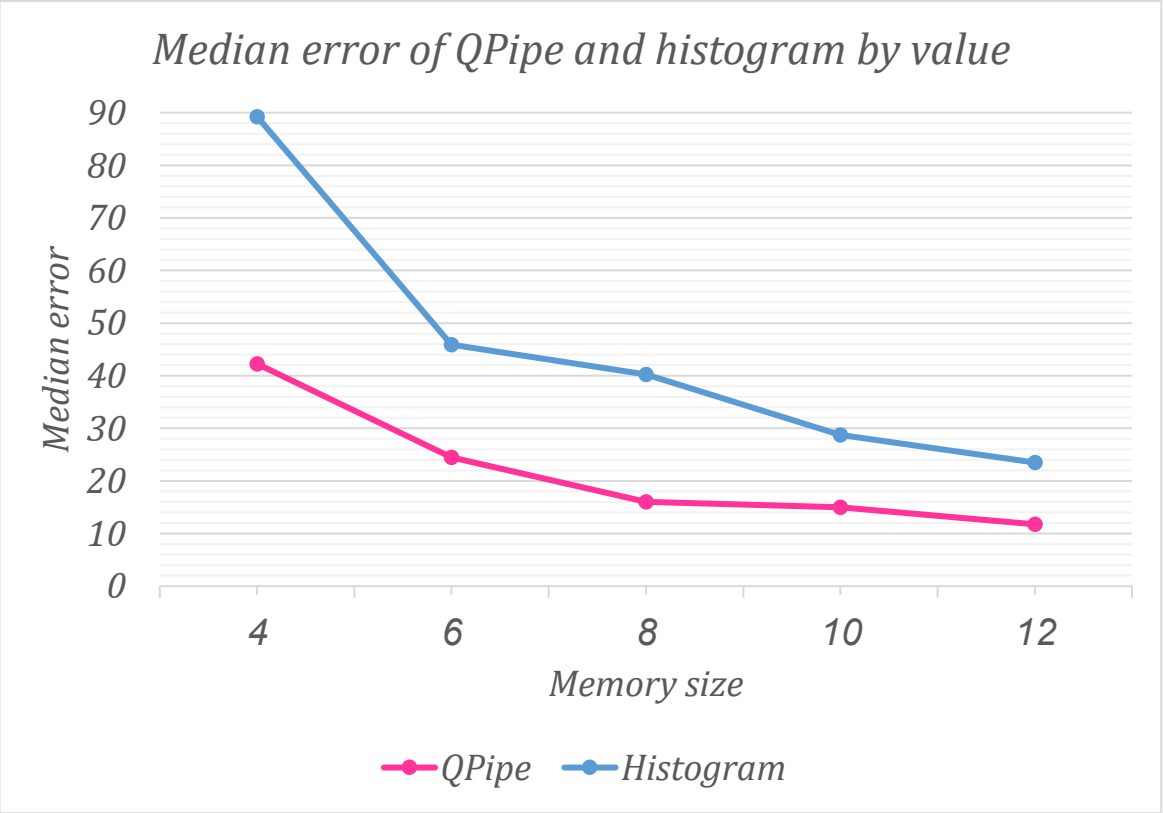
נשים לב כי שגיאת הממוצע קטנה יותר משגיאת החציון עבור שני האלגוריתמים. ככל שהטווח הצטמצם השגיאה עבור QPipe יורדת, אך עבור ההיסטוגרמה השגיאה נשארת יחסית קבועה.

ניתן לראות כי שימוש ב-QPipe למציאת החציון מניב תוצאות טובות יותר, כלומר שגיאה נמוכה יותר, מאשר החציון של ההיסטוגרמה. החל מטווח 300-700 ניתן לראות שהאלגוריתם מביא לתוצאה טובה יותר מההיסטוגרמה. בנוסף, השיפור בגרף החציון ב-QPipe הינו משמעותי יותר מהשיפור הנראה בגרף הממוצע.

נסיק כי השימוש באלגוריתם QPipe עדיף כאשר טווח הערכים בתעבורה אינו רחב מאוד, ואילו שימוש בהיסטוגרמה יתאים לתעבורה בעלת טווח רחב יותר, זאת בהתאם להנחה.

תוצאות הרצת האלגוריתם QPipe בהשוואה ליצירת היסטוגרמה עבור זיכרון משתנה וטווח תעבורה קבוע (ערכי value בין 300-700) מוצגות בגרפים הבאים :





איור 24 : גרפי חציון וממוצע של השגיאה

בגרפים אלו טווח ערכי ה-value הינו קבוע בין 300-700, אך גודל הזיכרון משתנה מ4 ל-12.

ניתן לראות כי ישנה מגמה יורדת עבור השגיאה של שני האלגוריתמים בשני הגרפים ככל שמגדילים את גודל הזיכרון. נשים לב כי השיפור עבור שגיאת הממוצע קטן יותר מאשר השיפור בשגיאת החציון. כמו כן, השגיאה הנובעת משימוש באלגוריתם QPipe הינה נמוכה יותר מהשגיאה המתקבלת מההיסטוגרמה בשני הגרפים לכל גודל זיכרון.

נסיק כי בתעבורה בעלת מאפיינים דומים, השימוש באלגוריתם QPipe יביא לשגיאה נמוכה יותר בחישוב הממוצע והחציון, לכן עדיף להשתמש בו.

לסיכום, ככל שאופי התעבורה דומה והומוגני יותר, דבר אשר נפוץ ברשתות מסוימות, השגיאות אשר מתקבלות מ-QPipe הינן קטנות יותר מהשגיאות המתקבלות מההיסטוגרמה. מתוצאות אלו, ניתן ללמוד, שהשימוש באלגוריתם זה יביא לייעול ניהול הרשת על ידי ניטור של התעבורה ברשת והחלטה דינמית של הניתוב בעקבותיו.

ניתן לבצע איסוף מידע וסטטיסטיקות על החבילות ברשת במגוון דרכים, ובכך להשיג מידע מקדים על אודות התעבורה ברשת כדי לייעל את הניתוב. כידוע, הזיכרון ברכיבים הינו מוגבל, ולכן מהווה פקטור משמעותי, לצד זאת, נרצה להגיע לדיוק מירבי בניטור על החבילות עם זיכרון מועט. נראה מהגרפים שעבור גדלי זיכרון קטנים, שגיאת ה-QPipe קטנה באופן משמעותי מהשגיאה המתקבלת מההיסטוגרמה. הגדלת הזיכרון מובילה להקטנה של השגיאה בשני האלגוריתמים, ובפרט להתכנסות לערכי שגיאה דומים.

בנוסף, מציאת חציון הינה פעולה פשוטה יותר לביצוע מאשר חישוב ממוצע , ולכן ישנה עדיפות לשימוש בחציון עבור חישובים אוחרים לשם ניתוב החבילות ברשת באופן יעיל, על מנת לא להעמיס על המערכת הכוללת.

ביבליוגרפיה

[1] N. Ivkin et al. QPipe: Quantiles Sketch Fully in the Data Plane
https://xinjin.github.io/files/CoNEXT19_QPipe.pdf

[2] $P4_{16}$ Language Specification:
https://p4.org/p4-spec/docs/P4-16-v1.2.4.html

[3] Barefoot Networks, Programmable Data Plane at Terabit Speeds
https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf