**מטלה 3 – רשתות תקשורת**

מגישים: 314779745, 207486473

# הוראות הפעלה לתכנות:

בתיקייה קיימות 6 תכנות ולכולן קובץ make.

ל-make ישנן 2 פקודות עיקריות: make all (ברירת המחדל) ו-make tests.

make all: תקמפל את התכנות הנדרשות במטלה: TCP\_Receiver, TCP\_Sender, RUDP\_Receiver, RUDP\_Sender.

מלבד הדגלים והפונקציונליות הנדרשת במטלה הוספנו עוד שני דגלים:

עבור ה-senders: הדגל -auto n ישלח את הקובץ n פעמים אוטומטית ויסגור, במקום לקבל קלט מהמשתמש.

עבור ה-receivers: הדגל -format ידכא את כל ההדפסות מלבד סיכום המידע בפורמט csv.

כמו כן, אם לא מסופק הדגל -ip הוא מקבל ערך ברירת מחדל של 127.0.0.1 (localhost).

make tests: תקמפל שתי תכנות עזר: rudp\_sender\_test, rudp\_receiver\_test.

התכנות מממשות ממשק בסיסי של שליחת הודעות על פני RUDP כאשר ההודעה “close” מסמנת סגירת קשר.

הערה חשובה: עבור תכנות RUDP עלול להיות באג בחישוב checksum כאשר ל-gcc מוגדר דגל -O3, אם קיימת בעיה ניתן להוריד את רמת האופטימיזציה או להוריד את הדגל לחלוטין.

הקלטות:

בתיקיית Wireshark captures קיימות 2 הקלטות עיקריות:

RUDP segmentation and retransmission: הקלטה של הרצאת RUDP ושליחת הודעה פשוטה עם דוגמה לסגמנטציה ול-retransmission.

TCP Wireshark capture: הקלטה של הרצת TCP\_Receiver ו-TCP\_Sender בה ניתן לראות דוגמאות לדרכי הפעולה של TCP.

בנוסף בתוך התיקייה קיימת תיקיית Scenario captures בה הקלטה של כל אחד מהמקרים שאנחנו מתבקשים להריץ (לא אותן ריצות שמופיעות בחלק C מכיוון שהן דרך סקריפט שכתבנו שהריץ אותן אוטומטית).

בהמשך הקובץ קיימים הסברים לרוב קבצי הקוד (הנדרשים) ולהקלטות המצורפות עם הסבר מפורט על איך פועל פרוטוקול RUDP שלנו.

# חלק א:

בחלק זה כתבנו 2 תוכניות, Sender ו-Receiver כאשר שתיהן משתמשות ב-Socket שמבוסס על פרוטוקול TCP.

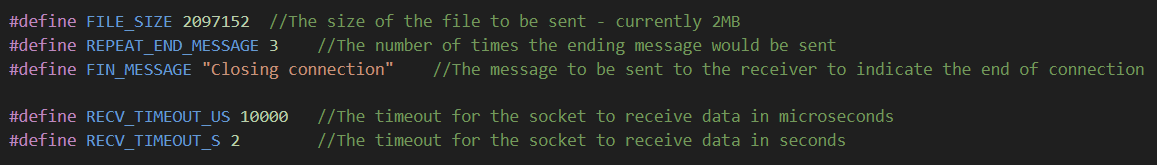
התוכנית TCP\_Sender שולחת ראשית את גודל הקובץ אותו מתכננת לשלוח (כדי שה-Receiver יוכל להקצות זיכרון ל-Buffer שלו כנדרש), לאחר מכן שולחת את הקובץ עצמו. ולבסוף שואלת את המשתמש אם לשלוח את הקובץ שוב. כאשר המשתמש בוחר שלא, התוכנית שולחת הודעת שמצהירה על סיום ההתקשרות עם ה-Receiver.

התוכנית TCP\_Receiver קולטת את הפאקטה הראשונה שמכילה את גודל הקלט ומקצה Buffer בגודל המתאים. לאחר מכן, מקבלת את הקובץ. יתכן שהקובץ ישלח במס' Segments, לכן התוכנית סופרת את מס' הבייטים שהתקבלו וממשיכה "להאזין" עד שהתקבלו מס' הבייטים שהשולח דיווח שישלח.

אם כל הקובץ עבר בשלמותו, התוכנית תהיה מוכנה לקלוט את הקובץ הבא. כך תמשיך עד שתתקבל הודעת סיום, ואז ה-Receiver ייסגר (וישחרר את הזיכרון שהוקצה במהלך התוכנית).

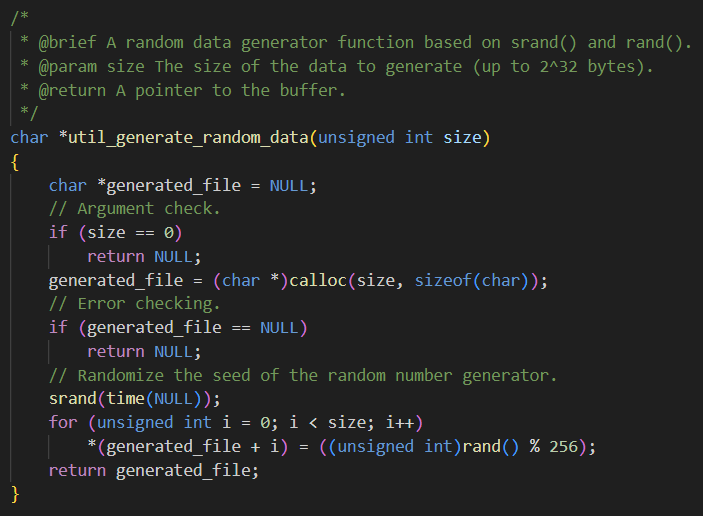
## הסבר קוד TCP\_Sender.c:

ראשית נגדיר את כל הקבועים בהם נשתמש במהלך התוכנית:

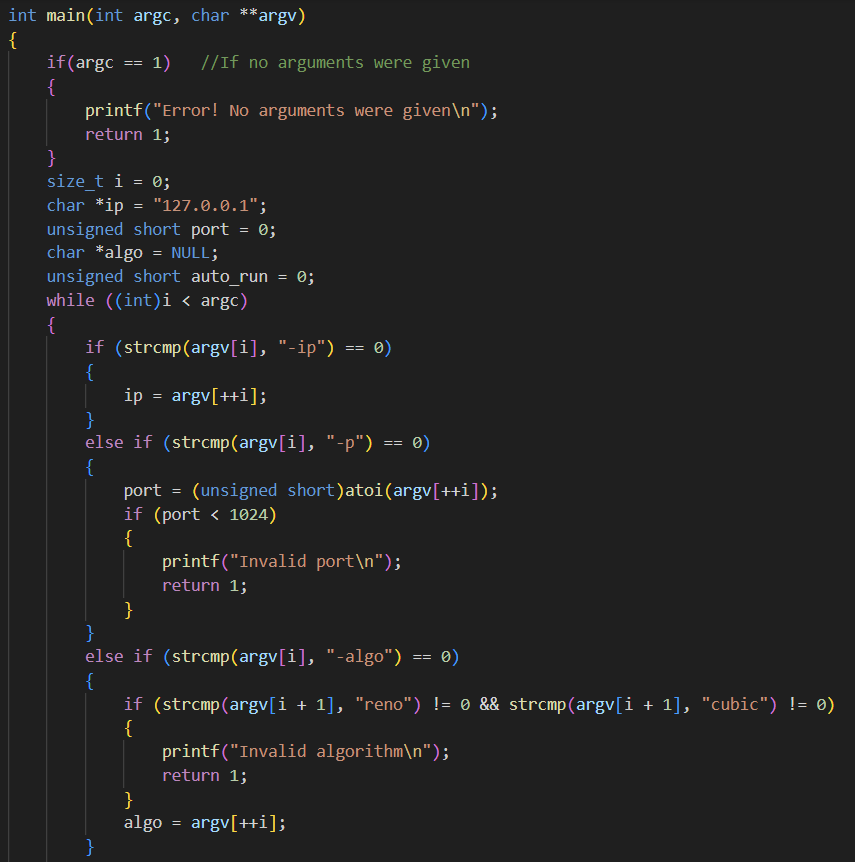


נשים לב שאת הודעת סיום ההתקשרות אנו שולחים מס' פעמים, זאת בשביל להתגבר על מקרה של Packet Loss, כך שאנו מקטינים את הסיכוי שה-Receiver לא יקבל את הודעת הסגירה וישאר פתוח.

נגדיר גם קבועים שישמשו אותנו להגדרת Timeout ל-Socket, את הודעת סיום התקשורת וקבוע של גודל הקובץ שנשלח (כרגע עומד על MB2).

פונקציה נתונה שמחזירה כתובת למערך שמכיל תווים רנודמליים בגודל נתון.

נשתמש בפונקציה הזו כדי לייצר את הפלט שנשלח ל-Receiver.



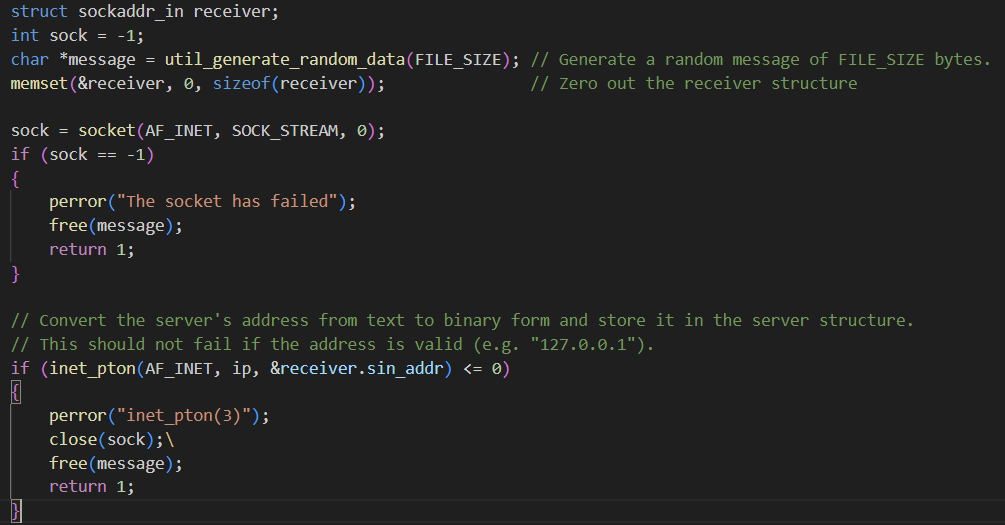
A computer screen with text

Description automatically generated

נבצע Parsing לארגומנטים שנקבל מהמשתמש ונחזיר שגיאה אם הפורט או האלגוריתם לא חוקיים או לא נתונים.

הגדרנו ערך דיפולטיבי ל-IP שיהיה כתובת ה-IP העצמית של המחשב (localhost).

הוספנו בקוד שלנו אופציה להעביר ארגומנט של מס' הפעמים שהשולח ישלח את הפלט על מנת לבצע בדיקה אוטומטית.

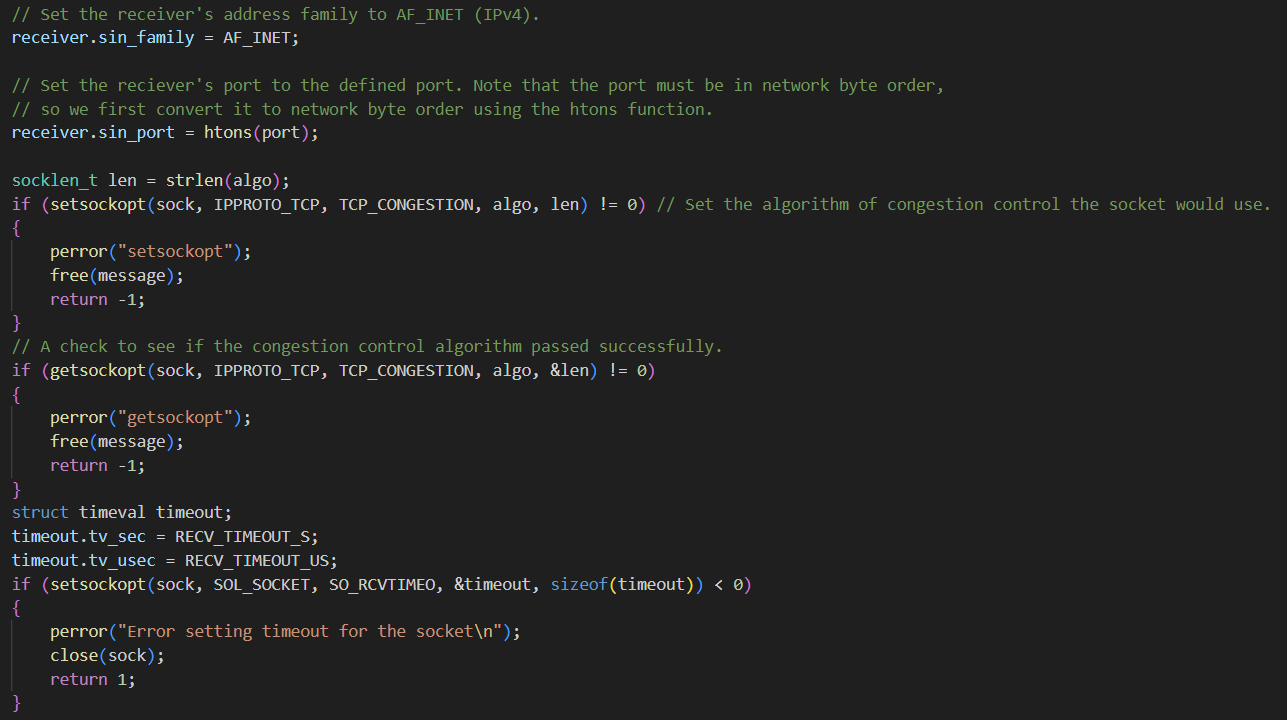


נגדיר socket חדש שמבוסס על IPv4 (AF\_INET) ופרוטוקול TCP (SOCK\_STREAM)

בנוסף נגדיר sockaddr\_in שיכיל את כתובת ה-IP של ה-Receiver. נראה בפונקציות הפתיחה שכתובת זו תהיה כתובת IPv4.

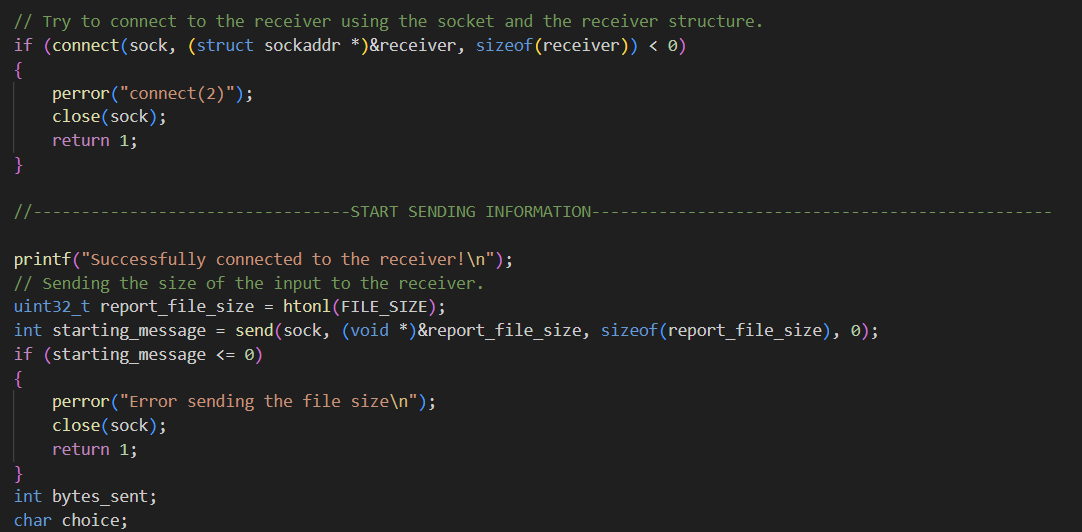
ונגדיר מצביע להודעה שייצרנו.

ניתן לראות שבכל המקומות בתוכנית בהם יתכן שתיהיה שגיאה, אנו בודקים זאת ומסיימים את הריצה במידה ואכן הייתה שגיאה.

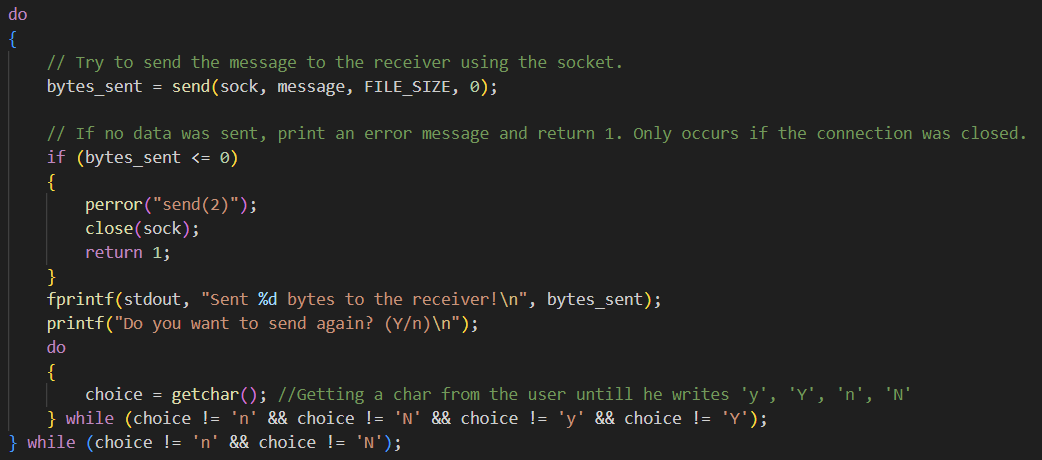


נמשיך בהגדרת ה-Receiver, נגדיר את הפורט שלו ואת סוג הכתובת שלו כ-IPv4.

באמצעות פונקציית setsockopt נגדיר את אלגוריתם ה-congestion control של ה-socket ואת זמן ה-Timeout של ה-socket.

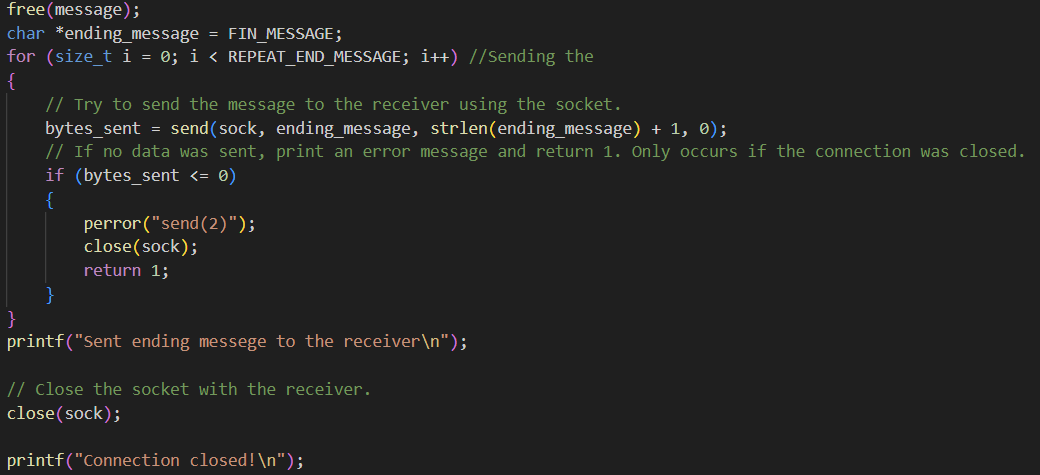


נתחבר ל- Reciver באמצעות ה-receiver sockaddr\_in שיצרנו שמכיל את כתובת ה-Receiver, ונשלח את ההודעה הראשונה שתדווח ל-Receiver על גודל הקובץ אותו נשלח.



נשלח את הקובץ ונשאל את המשתמש האם לשלוח את הקובץ שוב.

נשלח את הקובץ עד שהמשתמש ילחץ על n/N.

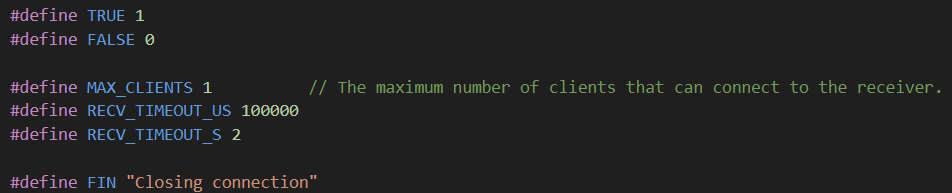
אם יצאנו מהלולאה, המשתמש ביקש לסיים את השליחה. כלומר, ניתן לסיים את התוכנית.

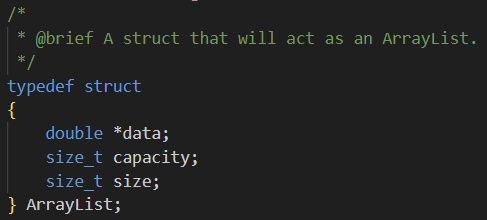
נשחרר את הזיכרון שבו השתמשנו במהלך הריצה ונשלח הודעת סיום (מס' פעמים קבוע, כדי להתגבר על Packet Loss).

לאחר מכן נסגור את ה-socket.

## הסבר קוד TCP\_Receiver.c:

ראשית נגדיר את כל הקבועים בהם נשתמש במהלך הריצה:

ה-Receiver מכיר את הודעת הסיום כדי שידע איזה הודעה מסמנת את סיום התקשורת.

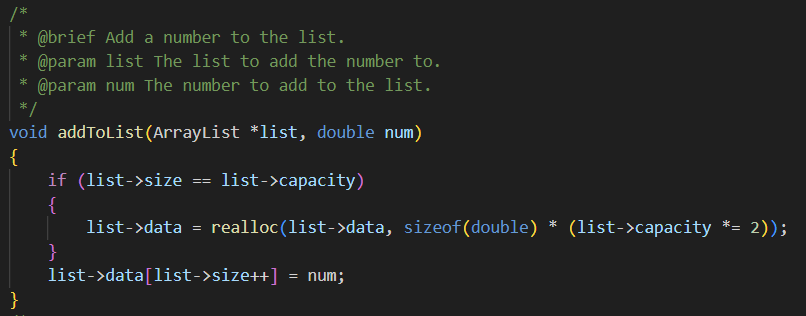


ע"מ לשמור את ערכי הזמן והמהירות של התוכנית, יצרנו struct של מערך דינאמי.

\*data – מצביע לערך המספרי ששמור במערך

Capacity – כמה איברים ניתן להכניס למערך

Size – כמה איברים שמורים כרגע במערך.

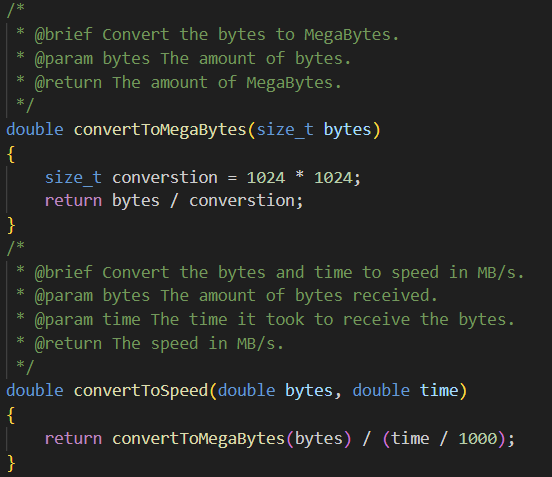


פונקציית הוספת איבר חדש למערך.

המערך מגדיל את עצמו בדיוק כמו מערך דינאמי ב-Java.

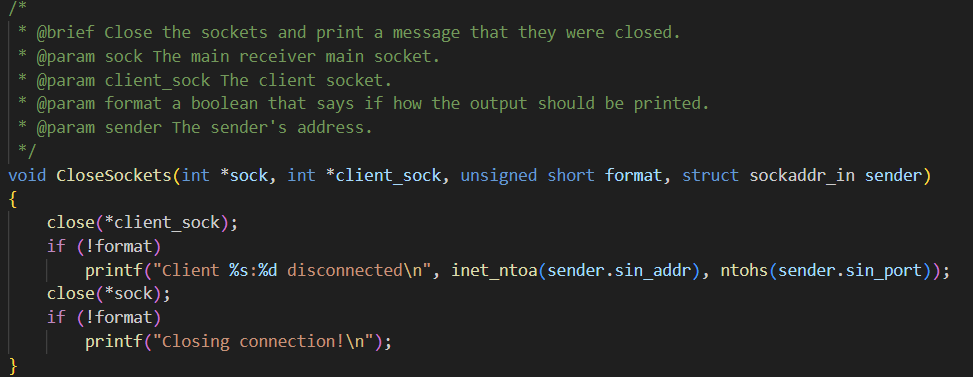
אם כמות האיברים שכרגע במערך זהה לגודל המערך, נגדיל את גודל המערך פי 2.

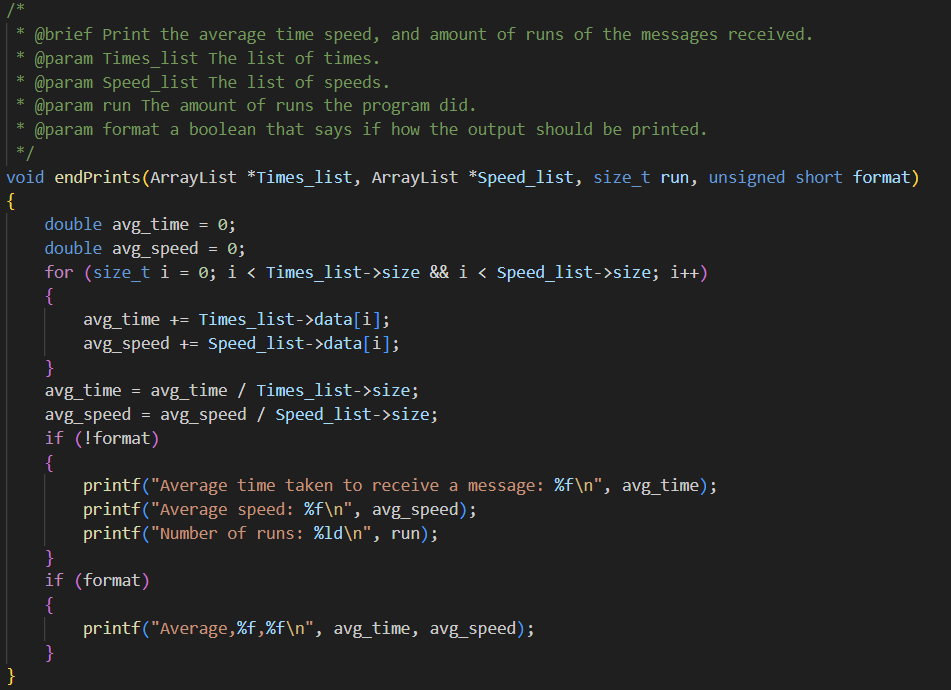
לבסוף, נוסיף את האיבר החדש למקום האחרון.



פונקציה שממירה מבייטים ל-MB

ופונקציה שמקבלת זמן וכמות בייטים ומחזירה את המהירות שבה נשלחו אותם בייטים.

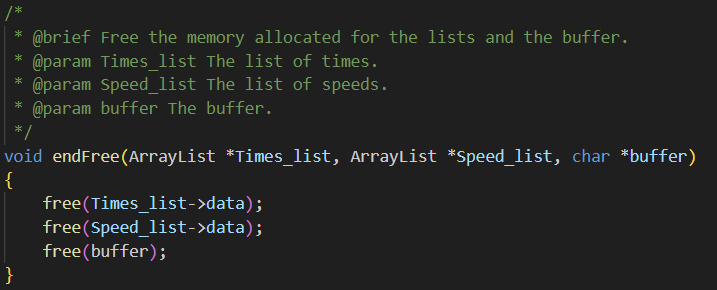
כדי לפשט את הקוד שב-main, הוצאנו את סגירת ה-sockets לפונקציה נפרדת.



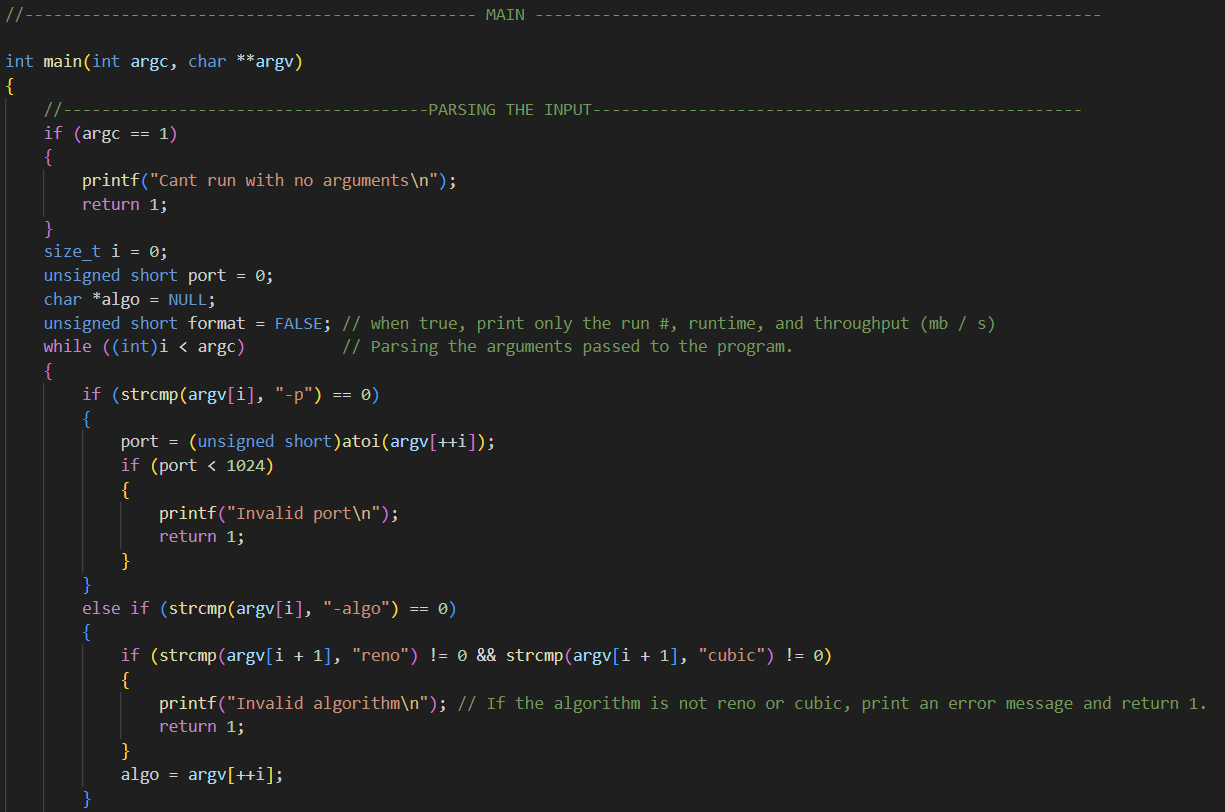
מאותה סיבה גם הוצאנו את כל חישובי והדפסות סיום הריצה לפונ' נפרדת.

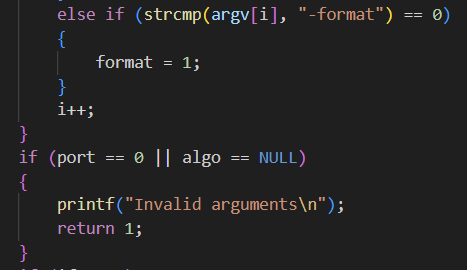
פונ' endPrints מבצעת חישובי ממוצעי זמן ומהירות הריצה הכולל של כל הריצות שבוצעו.

לבסוף מדפיסה את הממוצעים בהתאם למצב הפעלת התוכנית (format דלוק או כבוי).

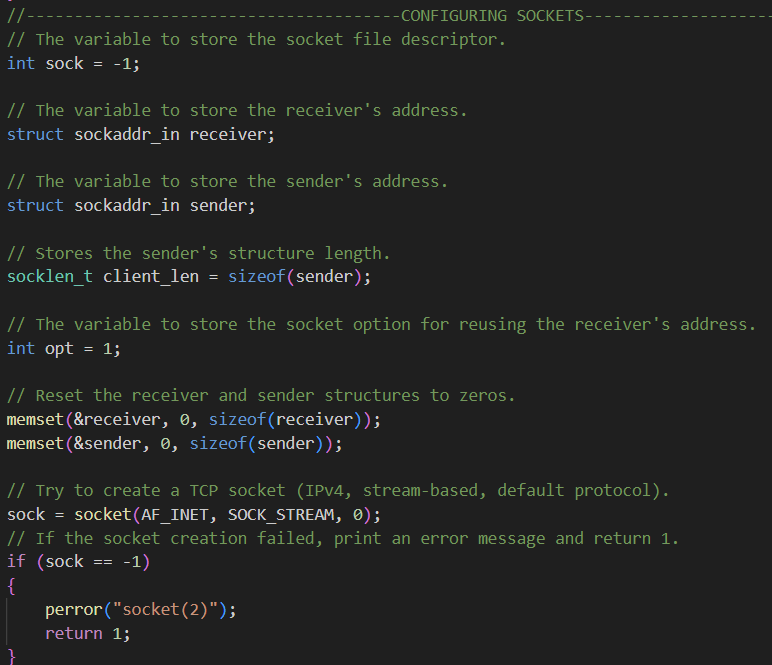
אותו דבר גם על שחרור הזיכרון במידה והייתה שגיאה בריצה או בסיום הריצה.

**כעת נעבור ל-main:**

כמו שביצענו ב-Sender, נבצע Parsing לארגומנטים של התוכנית.

הגדרנו ארגומנט format שגורם לתכנה להדפיס מידע מצומצם בפורמט CSV לטובת בדיקות אוטומטיות.

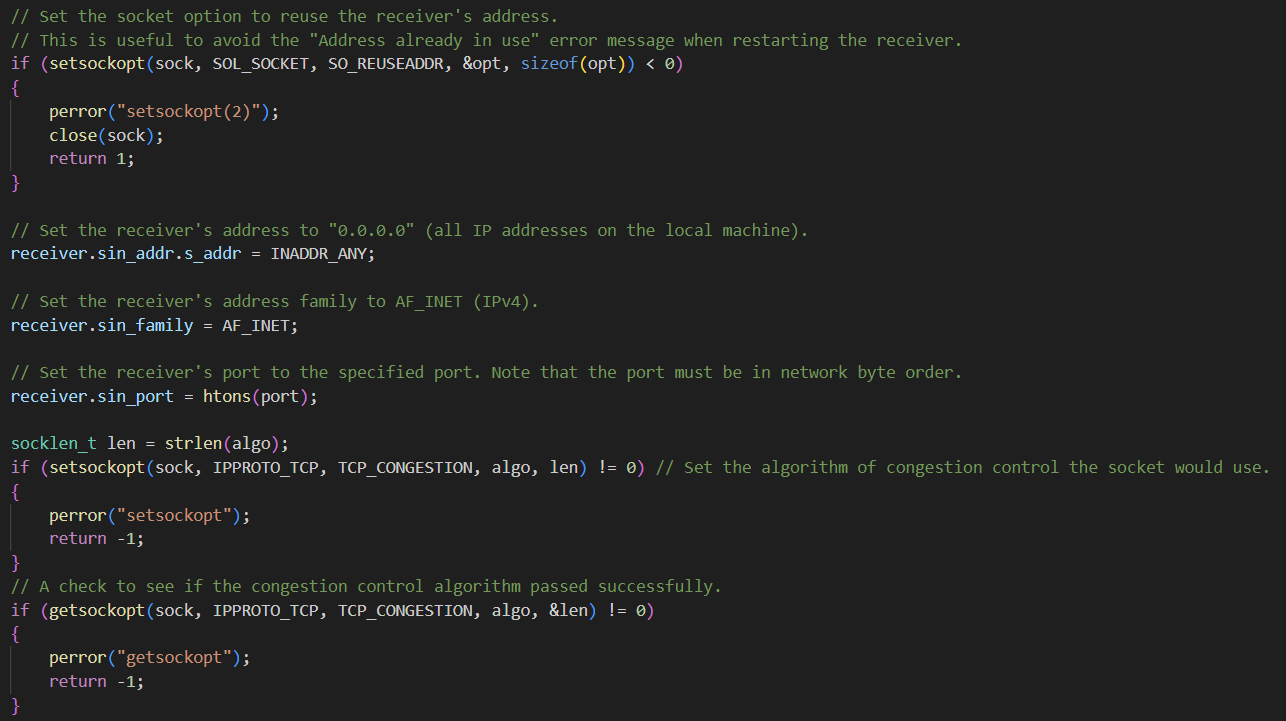
אם לא התקבל פורט או אלגוריתם TCP Congestion, נדפיס שגיאה ונסיים את הריצה.



נגדיר Socket וכתובות sender ו-receiver.

נאפס את 2 הכתובות. ונגדיר את ה-Socket שישתמש בפרוטוקול TCP ויבוסס על IPv4.

נשתמש במשתנה opt בהמשך ע"מ להגדיר שיהיה אפשר להשתמש באותה כתובת עבור ה-receiver מס' פעמים.

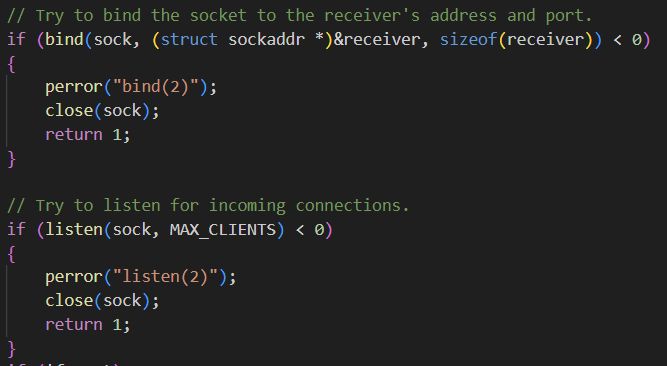


נגדיר שיהיה אפשר להשתמש בכתובת ה-IP של ה-receiver מס' פעמים, זאת כדי להימנע משגיאת

“Address already in use”.

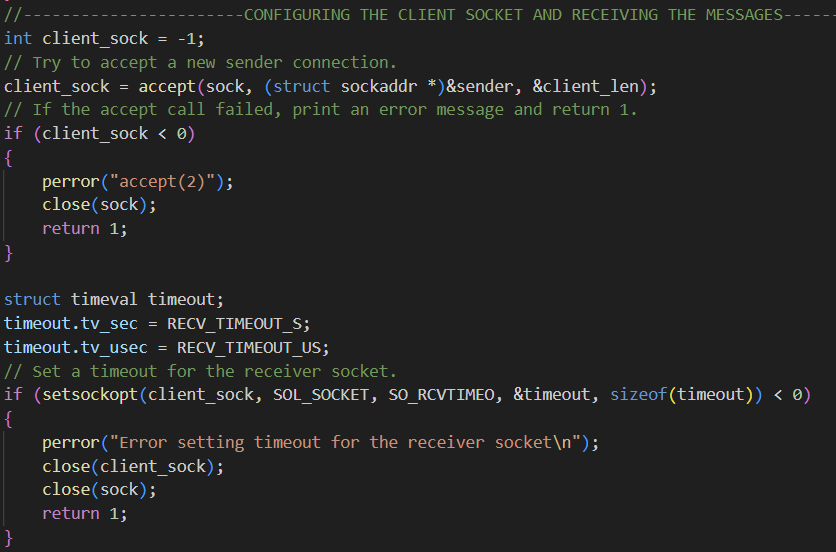
לאחר מכן נגדיר את ה-receiver שיהיה תואם לשולח (פורט תואם ו-IPv4) ונגדיר שנסכים לקבל הודעות מכל IP (INADDR\_ANY).

נגדיר את אלגוריתם ה-Congestion control של ה-socket לאלגוריתם שקיבלנו כארגומנט מהמשתנה (algo) באמצעות פונקציית setsockopt ולאחר מכן נבדוק האם ה-Socket אכן קיבל את הפקודה.



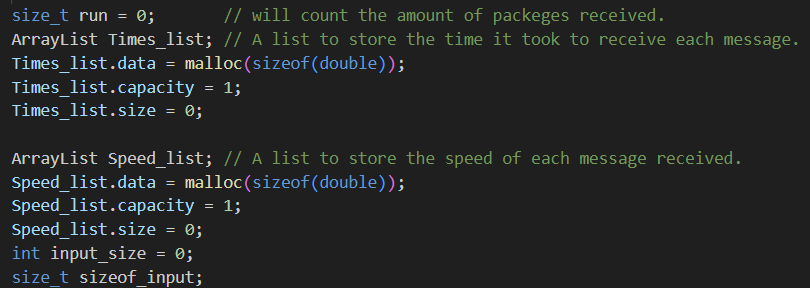
לאחר שסיימנו להגדיר את ה-socket, נבצע עליו bind עם כתובת ה-IP והפורט הנתונים (אותם הכנסנו ל-receiver בשורות מעל).

ולאחר מכן "נאזין" לחיבורים חדשים באמצעות פונקציית ה-listen.



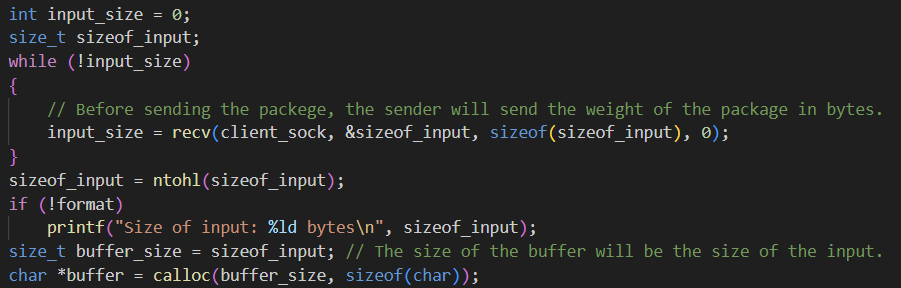
נגדיר Socket חדש של החיבור החדש (שהתקבל בפונקציית accept).

ונגדיר ל-Socket זה Timeout שהוגדר ברשימת הקבועים.



נגדיר משתנה שיכיל את כמות הקבצים שהתקבלו (רק כאשר נקבל את כל הקובץ נגדיל את המשתנה)

נגדיר גם 2 מערכים דינאמיים שיכילו את הזמנים והמהירויות של כל ריצה.

נמתין לקבל את החבילה הראשונה שהיא תכיל את גודל הקובץ שישלח.

נבצע Parsing למידע שקיבלנו, על מנת להשתמש בו בהמשך.

לאחר שקיבלנו את גודל הקובץ, נקצה מקום בזיכרון ל-Buffer בגודל buffer\_size.

A computer screen shot of a program

Description automatically generated

נגדיר מעין משתנה "בוליאני" שיסמן אם התקבלה הודעת סיום מה-Sender.

כל עוד לא התקבלה הודעת סיום, נחכה לקבל הודעות חדשות.

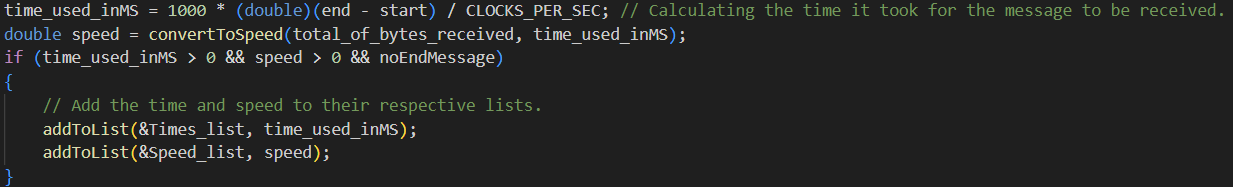
bytes\_received – מכיל את כמות הבייטים שהתקבלו ב-recv יחיד.

total\_of\_bytes\_received – מכיל את סך הבייטים שהתקבלו עד לקבלת כל הקובץ.

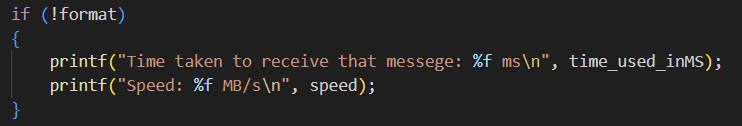
נפסיק לקבל חבילות חדשות מה-Sender אם סך הבייטים שקיבלנו גדול שווה לגודל הקובץ (שנתון לנו) או שהתקבלה הודעת סיום.

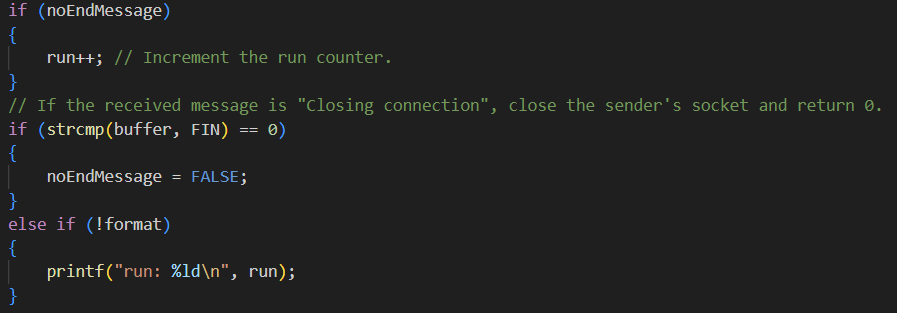
כמו כן, אם קיבלנו 0 בייטים, ה-Sender התנתק. נגדיר מצב זה בתור קבלת הודעת סיום.

נשים לב שגודל הקובץ לא משתנה, לכן נכון להשתמש ב-sizeof\_input שקלטנו בתחילת הריצה, עבור כל הקבצים שנקבל.

בנוסף, מכיוון שלא נדרש לשמור את המידע (ע"פ הנחיות המטלה), אנו דורסים בכל recv את המידע הקודם שהיה ב-buffer.

נחשב את הזמן שלקח ל-Receiver לקבל את הקובץ, נחשב את המהירות. ונכניס את 2 הערכים למערכים המתאימים.

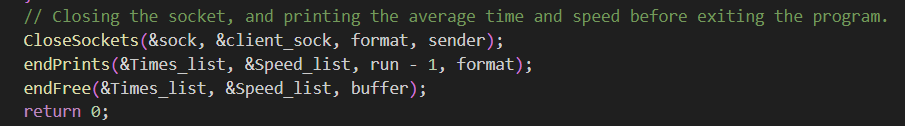
נדפיס את הזמן שלקח לקבל את הקובץ הנוכחי ואת המהירות.



לאחר שקיבלנו קובץ, נבצע את הפעולות הבאות:

נגדיל את מונה הריצות שלנו ב-1.

נבדוק אם אם התקבלה הודעת סיום.



לאחר שהתקבלה הודעת סיום, נסגור את ה-Sockets, נדפיס סטטיסטיקות ונשחרר את הזיכרון שהקצנו במהלך הריצה.

# חלק ב:

בחלק זה כתבנו 3 תוכניות, RUDP\_Sender, RUDP\_Receiver, RUDP\_API

כאשר ה-Sender וה-Receiver משתמשות ב-Socket שמבוסס על פרוטוקול RUDP שממומש ב-RUDP\_API.

התוכנית RUDP\_Sender שולחת ראשית את גודל הקובץ אותו מתכננת לשלוח (כדי שה-Receiver יוכל להקצות זיכרון ל-Buffer שלו כנדרש), לאחר מכן שולחת את הקובץ עצמו. ולבסוף שואלת את המשתמש אם לשלוח את הקובץ שוב. כאשר המשתמש בוחר שלא, התוכנית קוראת לפונקציית סגירת ה-Socket שמהלכה נשלחת הודעת FIN ל-Receiver.

התוכנית RUDP\_Receiver קולטת את הפאקטה הראשונה שמכילה את גודל הקלט ומקצה Buffer בגודל המתאים. לאחר מכן, מקבלת את הקובץ. ב-RUDP, יתכן שהקובץ ישלח במס' מקטעים אבל פונקציית ה-rudp\_recv מעבירה את התוכן ל-buffer רק כאשר כל התוכן התקבל (מס' הבייטים שהתקבלו שווה לגודל הקובץ). התוכנית מאזינה ל-Sender עד שהוא שולח הודעת FIN.

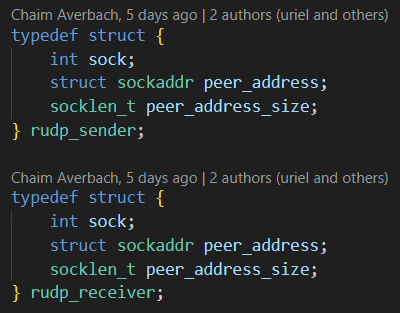
כאשר תתקבל הודעת FIN, ה-Receiver יחזיר הודעת FIN ACK ל-Sender ולאחר מכן יסגר, ישחרר את הזיכרון שהוקצה במהלך התוכנית וידפיס סטטיסטיקות.

חשוב: לפעמים כאשר ב-Makefile מוסיפים את ה-flag -O3 שמגדיר לקומפיילר לבצע אופטימיזציה, קיימות בעיות בחישוב checksum. אם נתקלים בבעיות אפשר למחוק את ה-flag או להוריד אותו לרמת אופטימיזציה נמוכה יותר.

## הסבר קוד rudp.h:

rudp.h הוא קובץ header שמגדיר את הפונקציות והמבנים הנדרשים לעבודה עם RUDP.

ראשית נגדיר את המבנים המייצגים את כל אחד מסוגי ה-socket:

יש לנו מבנה לכל אחד מסוגי הסוקטים מכיוון שכל פונקציה (שנראה בהמשך) בנויה לפעול רק על אחד מהם.

יחד עם זאת, המבנה עצמו זהה –

int sock – מזהה של ה-UDP socket עליו מתלבש ה-RUDP socket.

sockaddr peer\_address – מבנה המכיל את כתובת ה-peer איתו נתקשר.

socklen\_t peer\_address\_size – אורך השדה peer\_address – נדרש עבור פונקציות של UDP.

A screenshot of a computer program

Description automatically generatedלאחר מכן קיימות הצהרות של שתי הפונקציות לפתיחת ה-sockets עם הסברים לגביהן והערות רלוונטיות.

נראה את מימושן בהמשך.

A screenshot of a computer program

Description automatically generatedלאחר מכן הצהרות פונקציות הסגירה, שימו לב שפונקציה rudp\_close\_sender מבצעת את הסגירה בתקשורת (הודעות FIN).

A computer screen with text on it

Description automatically generatedלאחר מכן ההודעות האחראיות על התקשורת.

שימו לב במיוחד לערכי ההחזרה המיוחדים של rudp\_recv.

A screen shot of a computer program

Description automatically generated

כעת אנחנו מגיעים להצהרה על ה-header של הפרוטוקול.

ראשית אנו מגדירים bit masks לשדה flags.

לכל אחד יש משמעות מיוחדת שתוסבר בהמשך.

לאחר מכן אנחנו מגדירים את ה-header עצמו – זה header לסגמנט יחיד של RUDP:

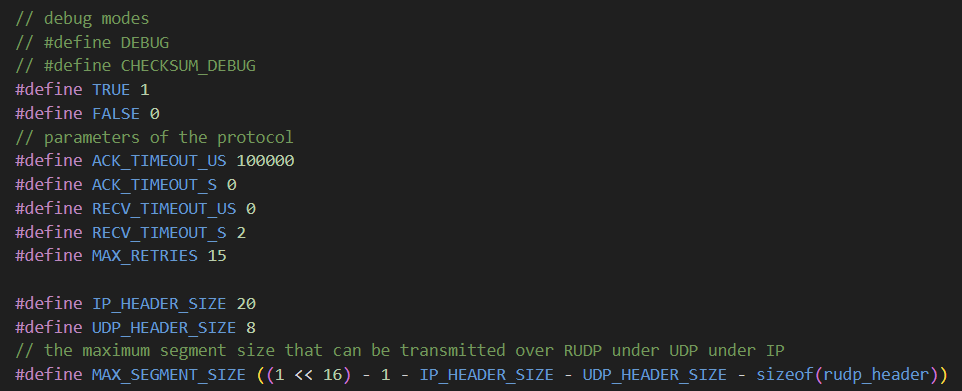
unsigned short len – אורך המידע בסגמנט (לא כולל ה-header).

char flags – בית אחד המכיל את הדגלים הרלוונטים לסגמנט.

unsigned short checksum – שני בתים המכילים checksum לסגמנט.

unsigned short segment\_num – מספר הסגמנט בהודעה – עבור הודעות שהתחלקו למס' סגמנטים.

## הסבר קוד RUDP\_API.c:

ראשית נגדיר את הקבועים שבהם נשתמש בתוכנית.

בנוסף להגדרת קבועים לזמני ה-Timeout של הודעת ה-ACK והודעות אחרות, הגדרנו גם את הקבועים הבאים:

MAX\_RETRIES – כמות הפעמים שננסה לשלוח פאקטה במידה ולא קיבלנו ACK, לפני שנגדיר את השליחה ככישלון.

IP\_HEADER\_SIZE – גודל ה-Header של פאקטה שמשתמשת בפרוטוקול ה-IP.

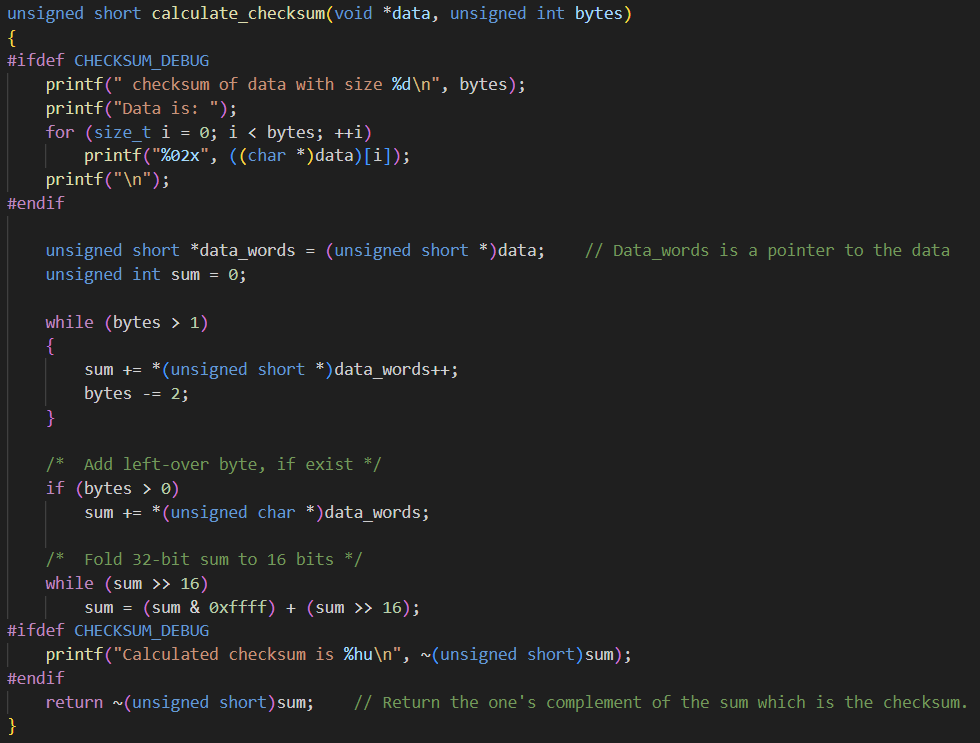
UDP\_HEADER\_SIZE – גודל ה-Header של פאקטת UDP.

MAX\_SEGMENT\_SIZE – הגודל המקסימלי של פאקטה יחידה הוא:

שכן, שדה ה-length ב-header של ה-IPv4 הוא באורך 16 ביטים, כך שהאורך המקסימלי של פאקטה יחידה הוא: .

מתוך מס' זה, נוריד את כל ה-Headerים שאנו מוסיפים להודעה – שתופסים מקום באורך ההודעה: IP Header, UDP Header, RUDP Header. שכן, RUDP משתמש בפרוטוקולי IP ו-UDP ועליהם מוסיף Header חדש מעצמו.

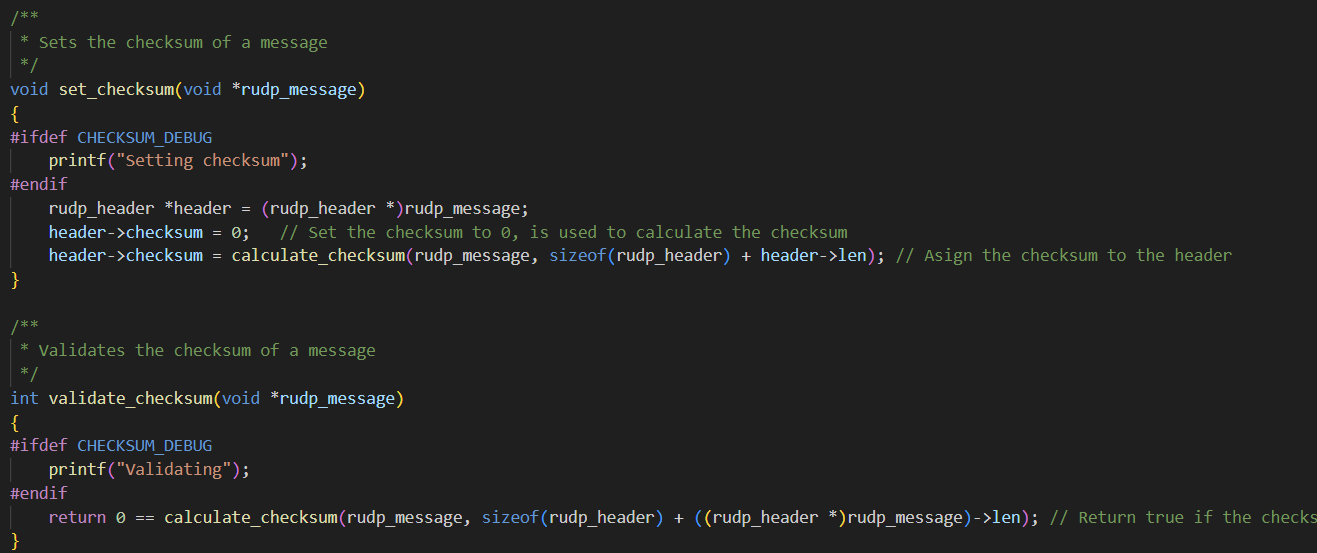
בנוסף הוספנו 2 הגדרות להודעות debug – DEBUG להודעות כלליות ו-CHECKSUM\_DEBUG להודעות בנושא checksum.

פונקציית חישוב ה-Checksum.

הפונקציה מבצעת חישוב של Checksum ה-data לפי החישוב שלמדנו בהרצאה.

\*data – מצביע למידע שעליו נדרש לבצע את חישוב ה-Checksum

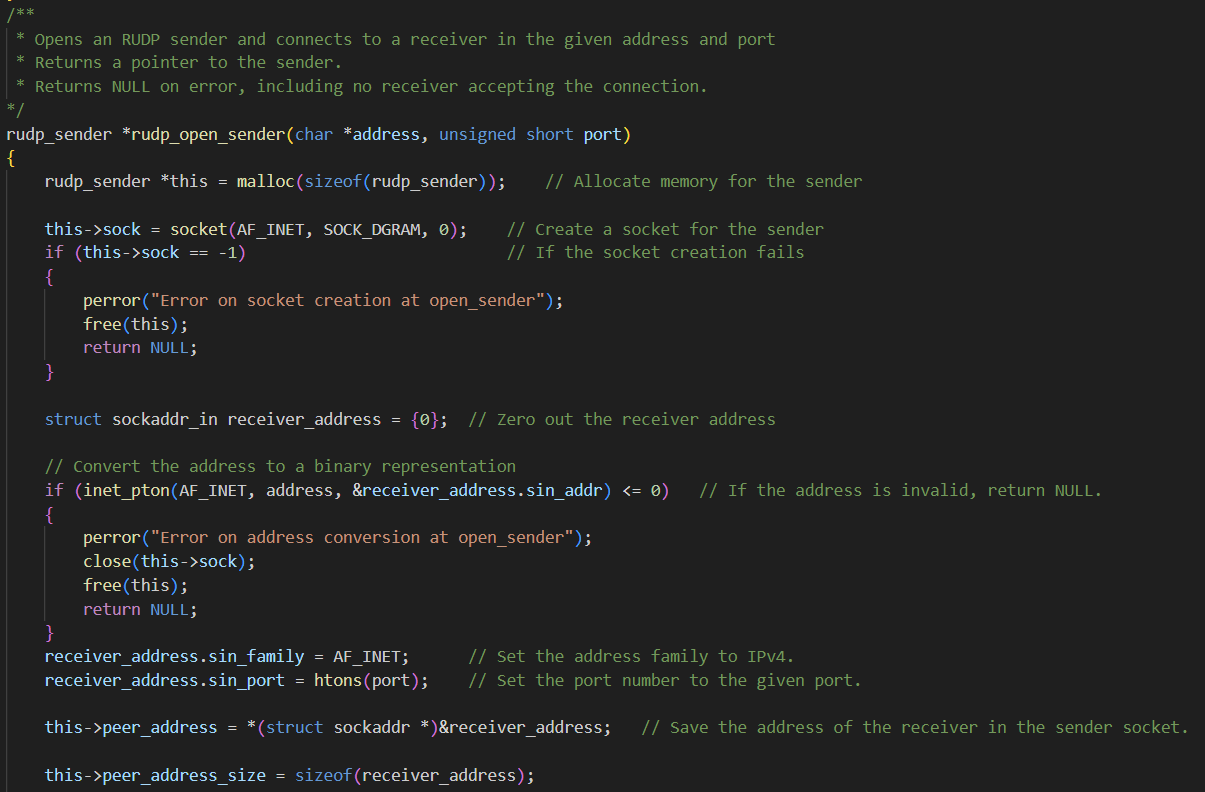
bytes – גודל ה-data בבייטים.

****

set\_checksum – מקבלת מצביע להודעת RUDP ומגדירה ל-header שלה את ערך ה-checksum הנדרש.

נשים לב שהגדרנו קודם את ערך ה-checksum ל-0, על מנת שלא יפריע לחישוב.

validate\_checksum - מקבלת הודעת RUDP, מחשבת את ה-checksum שלה ומחזירה “True” אם הוא שווה ל-0 כנדרש.



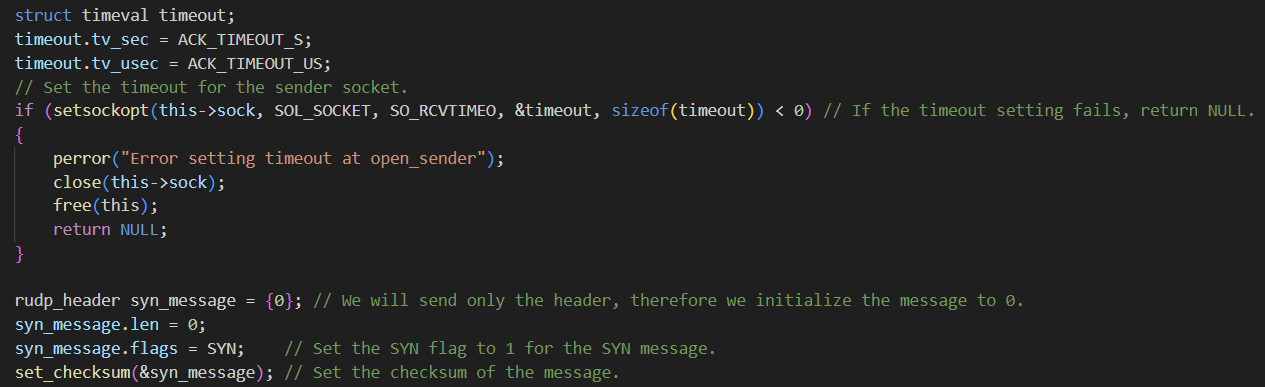
פונקציה לפתיחת RUDP Sender socket.

הפונקציה מקבלת כתובת ופורט ומחזירה מצביע ל-socket של ה-sender.

ראשית נקצה מקום ונגדיר את ה-socket כ-UDP שמבוסס על IPv4 (ועליו "נלביש" את עקרונות ה-RUDP).

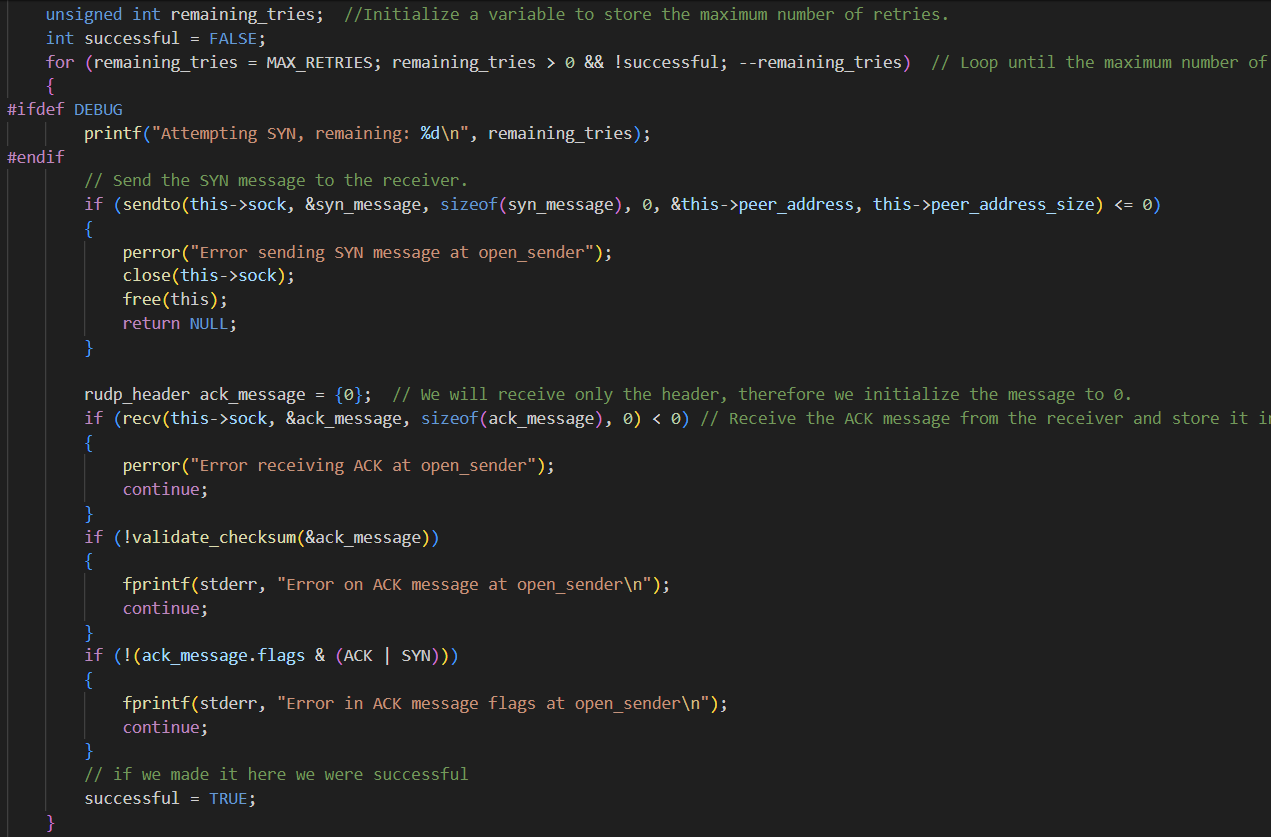
לאחר מכן נקצה את הכתובת הנתונה לתוך ערך הכתובת של ה-receiver, נגדיר כתובת זו ככתובת IPv4 ב-receiver ונקצה את הפורט הנתון ל-receiver.

בנוסף, נשמור את הכתובת ה-receiver בשדה ה-peer\_address של השולח.



נגדיר Receive timeout ל-Sender socket (זמנים שהוגדרו בתור קבועים)

נכין הודעת SYN, בה רק ה-SYN flag דלוק. הודעה זו נשלח בהמשך ל-Receiver כדי להתחיל תקשורת.

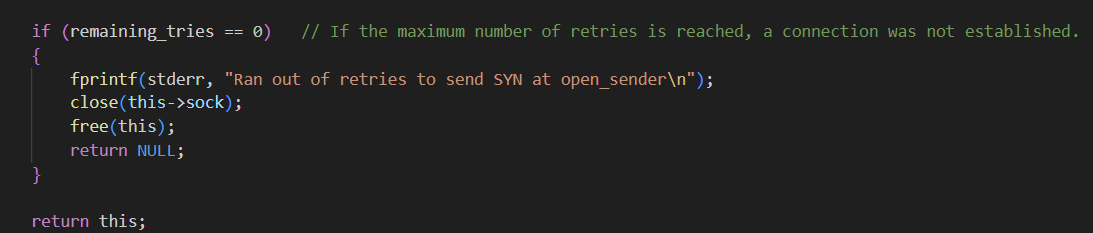


נשלח הודעת SYN ל-Receiver עד אשר נקבל הודעת SYN ACK בחזרה או עד שנגיע לכמות הפעמים המקסימלית שהגדרנו לניסיון תחילת התקשרות.

בקוד ניתן לראות שאנחנו "מכינים" הודעת ACK, מקצים את ההודעה שקיבלנו מה-Receiver לכתובת של הודעת ה-ACK, בודקים את אמיתות ה-Checksum שלה ומוודאים שזו אכן הודעת SYN ACK.

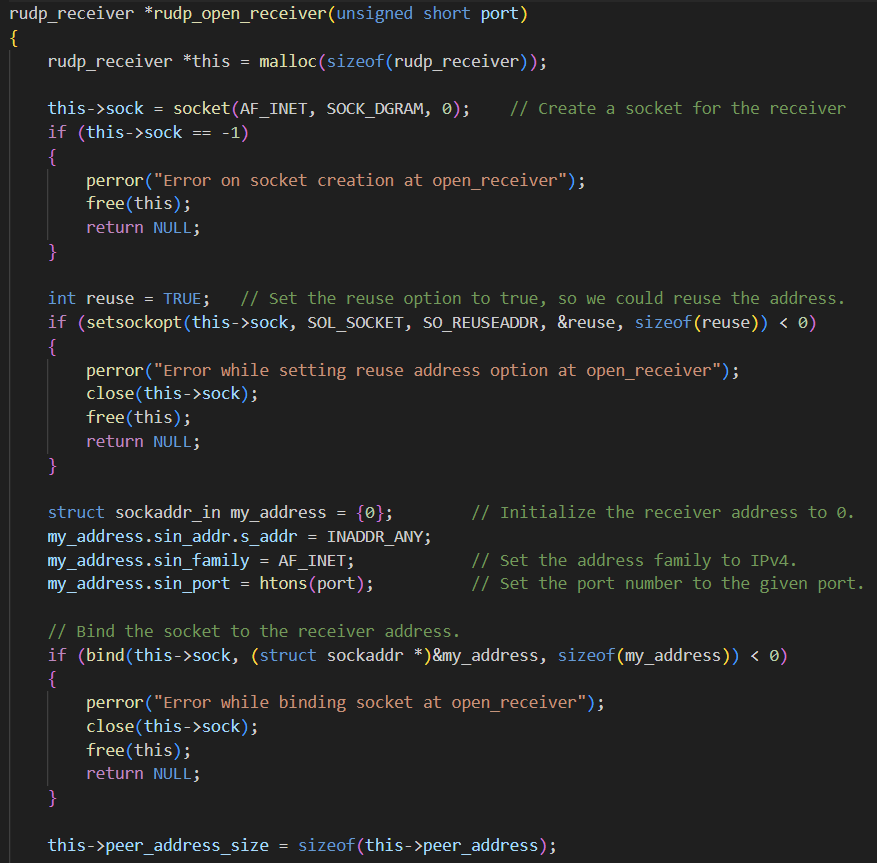
אם אחד התנאים לא מתקיים, נדפיס שגיאה ונחזור לתחילת הלולאה.

במידה וכל התנאים מתקיימים, הרי שזו הודעה תקפה, נשנה את משתנה ה-successful ל-True, מה שיסמן ללולאה לסיים את הריצה.



אם סיימנו את נסיונות שליחת הודעת ה-SYN ללא הצלחה, כלומר ניסינו MAX\_TRIES פעמים ובאף אחת מהפעמים לא קיבלנו הודעת SYN ACK (עם Checksum תקין כמובן), נדפיס שגיאה ונסגור את Socket השולח.

לבסוף נחזיר מצביע ל-Socket השולח. אם הייתה שגיאה במהלך הריצה של הפונקציה, החזרנו NULL על מנת לסמן זאת.



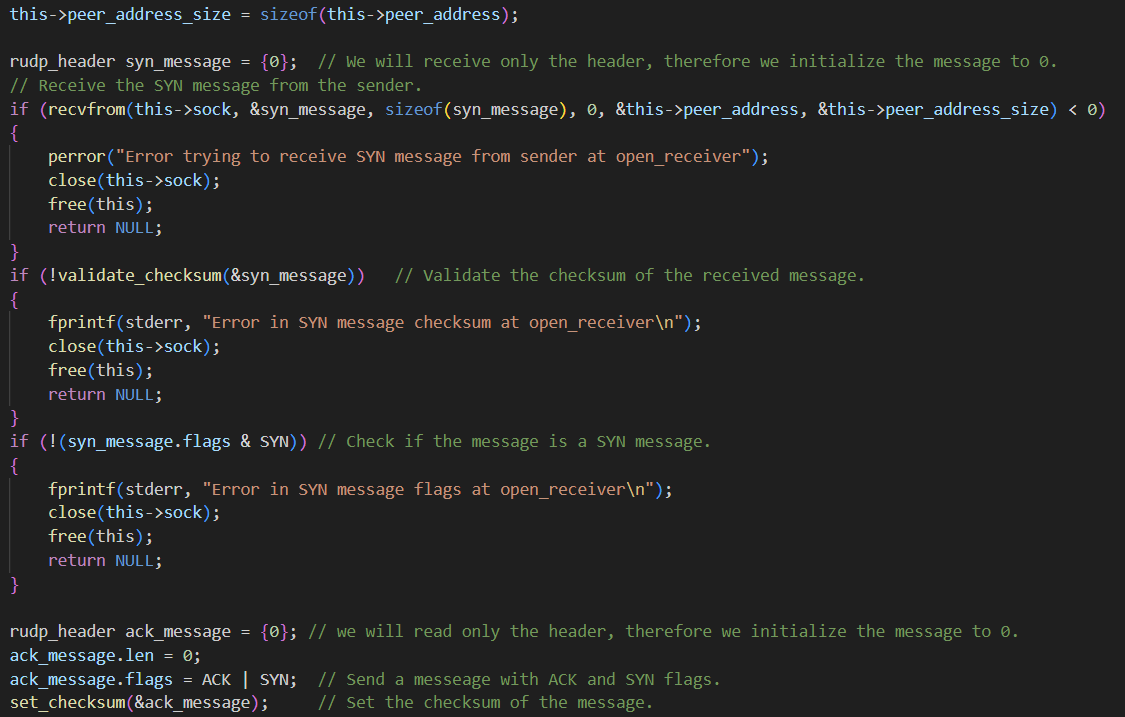
פונקציה לפתיחת RUDP Receiver socket.

הפונקציה מקבלת פורט ופותחת Socket שמוקצה לפורט הזה.

ראשית, נקצה מקום ל-Socket ונגדיר אותו בתור UDP Socket מבוסס על IPv4. לאחר מכן נגדיר שניתן להשתמש בכתובת של ה-Socket יותר מפעם אחת (בשביל למנוע שגיאות של “Address already in use”)

לאחר מכן, כמו שביצענו ב-TCP Receiver socket, נגדיר sockaddr\_in מסוג IPv4, שיכול לקבל תקשורת מכל כתובת, ומאותחל לפורט הנתון.

נבצע bind לאותו sockaddr\_in ל-socket שיצרנו, כך נגדיר ל-socket מאיזה פורט להאזין בהמשך ולאיזה סוג כתובת לצפות (IPv4).



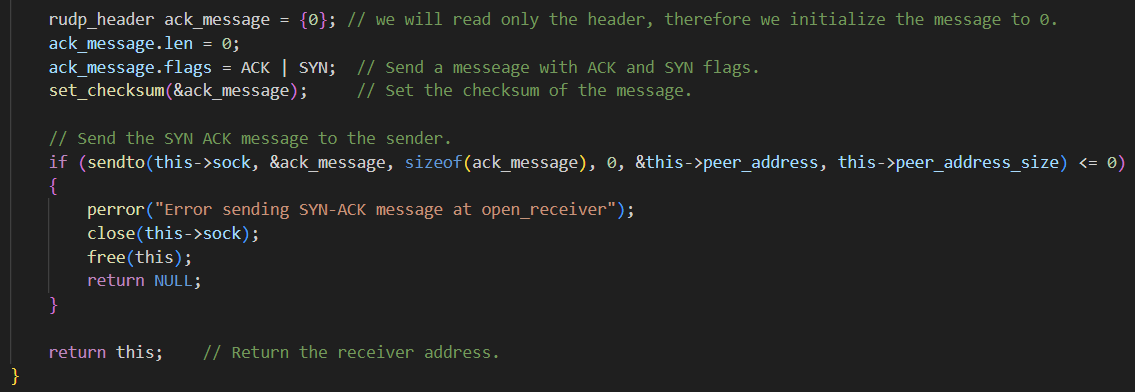
נכין הודעת SYN חדשה, הודעה זו ריקה, שכן בהודעות SYN/ACK/FIN אנו מקבלים רק את ה-Header.

נמתין לקבלת הודעה מהשולח, אותה נכתוב לכתובת של ההודעה שאותה הכנו.

לאחר מכן, נבדוק האם ההודעה היא הודעת SYN תקינה:

אם ה-Checksum שלה לא תקין – נחזיר שגיאה.

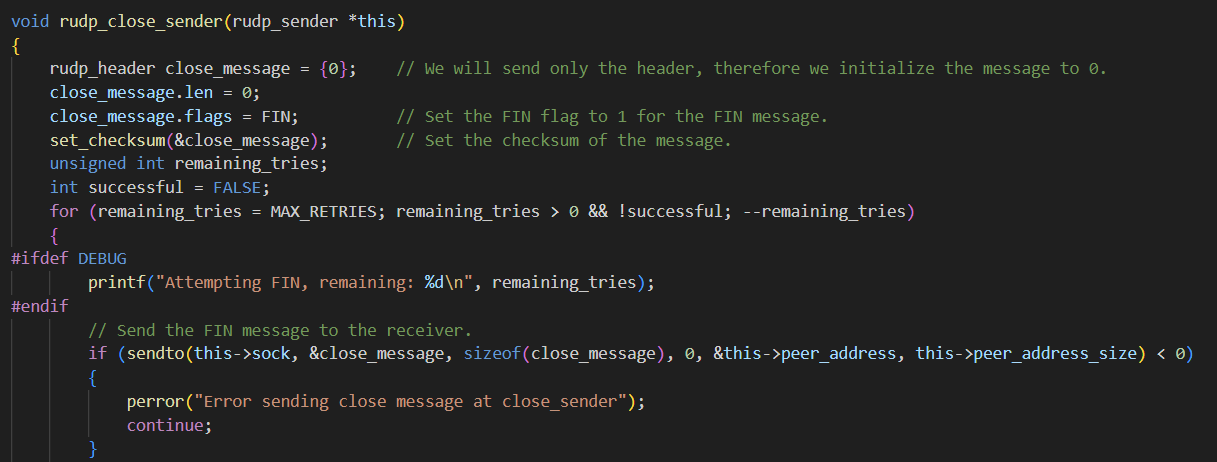
נבדוק את ערך flag ה-SYN שלה ב-header, אם אינו 1 – נחזיר שגיאה (נבצע את הבדיקה באמצעות שליפה של ערך SYN עם bitwise-and).



לאחר שהתקבלה הודעת SYN, אנחנו רוצים להחזיר הודעת SYN ACK.

נכין הודעת SYN ACK בה נדליק את ה-SYN and ACK flags (בעזרת bitwise-or ביניהם). ולהודעה זו נגדיר Checksum כדי שתהיה הודעה תקינה.

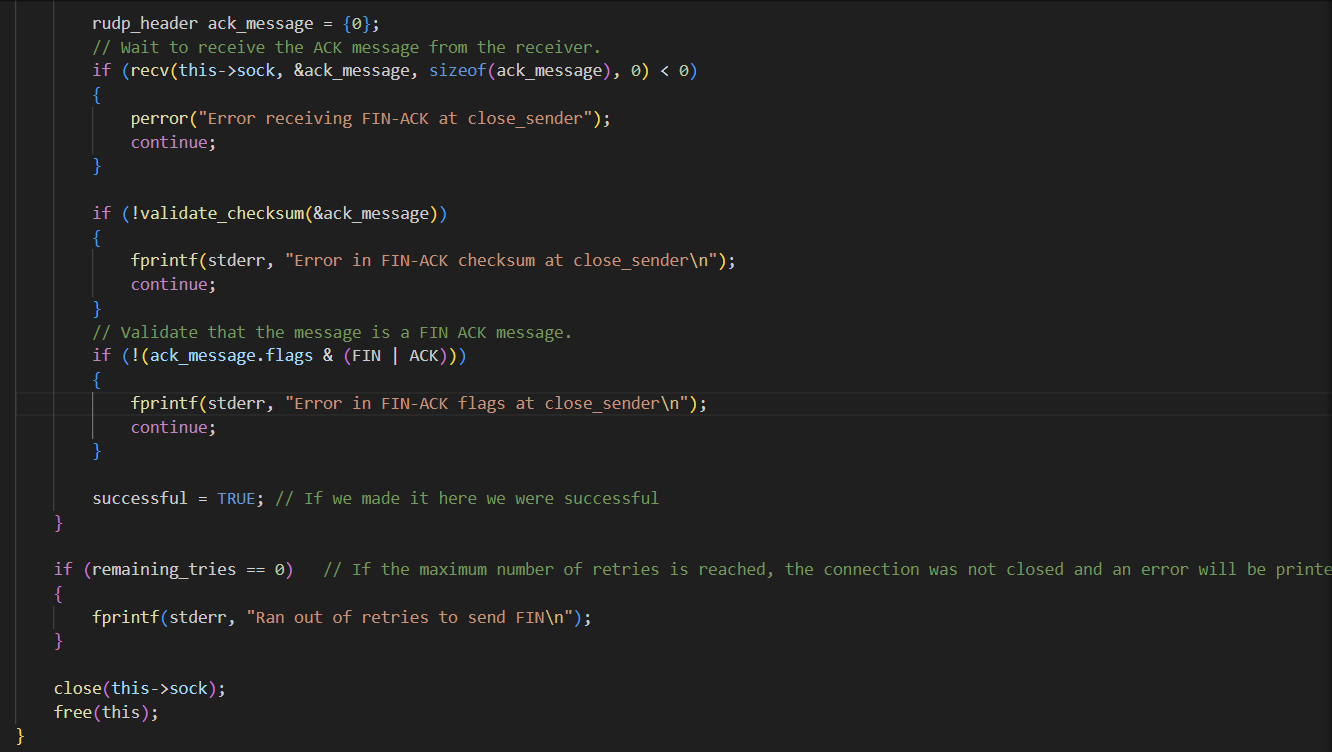
לאחר מכן נשלח את ההודעה ל-Sender ובמידה ולא הייתה שגיאה, נחזיר את מצביע ל-RUDP Receiver socket.



פונקציה זו מקבלת RUDP Sender, שולחת הודעת FIN ל-Receiver אליו הוא קשור וסוגרת אותו.

ראשית הפונקציה מכינה הודעת FIN, ע"י הגדרת RUDP Header בו ה-FIN flag דלוק. לאחר מכן נגדיר על הודעה זו Checksum כמובן.

נשלח את הודעת ה-FIN כמות הפעמים שהוגדרה (MAX\_RETRIES) או עד אשר נקבל FIN ACK ה-Receiver.

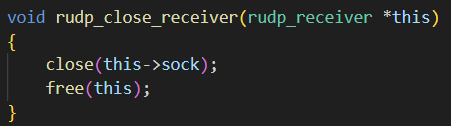


נקצה מקום להודעת ACK, אליה נכניס את ההודעה שנקבל מה-Receiver.

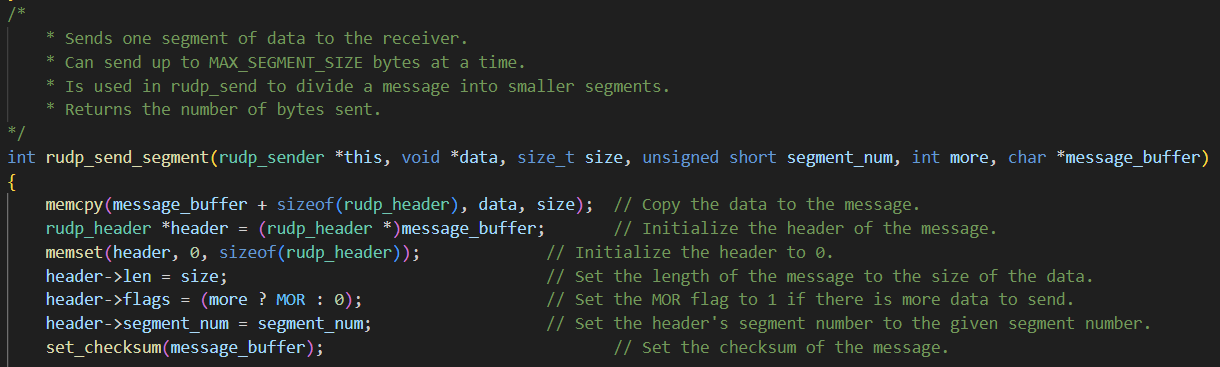
כאשר תתקבל הודעה, נבדוק את תקינות ה-Checksum שלה ואם היא אכן הודעת FIN ACK. במידה ושני התנאים מתקיימים, נגדיר ניסיון זה כהצלחה. במידה ולא, נדפיס שגיאה וננסה שוב לשלוח הודעת FIN.

אם יצאנו מהלולאה כאשר הגענו למקסימום ניסיונות שליחת ההודעה, נדפיס שגיאה אך בכל זאת נמשיך לסגירה.

לבסוף נסגור את ה-Socket ונשחרר את הקצאת המקום.

פונקציה זו מקבלת socket מסוג rudp\_receiver, סוגרת אותו ומשחררת את המקום אליו הוא הוקצה.

זאת בהנחה שהתקשורת נסגרה ע"י ה-sender.



פונקציה זו מקבלת סגמנט יחיד של מידע (מתוך הקובץ) ושולחת אותו ל-Receiver.

לפני כן, הפונקציה מצמידה ל-Header של הסגמנט אם יש עוד סגמנטים מאותו קובץ (ארגומנט ה-more), את מס' הסגמנט (ארגומנט ה-segment\_num) ואת גודל הסגמנט (ארגומנט ה-size).

נשים לב שהפונקציה מצפה לקבל גם message\_buffer שהוא באפר שהקוצה לה מראש על מנת שלא תצטרך להקצות באפר חדש לסגמנט.

לאחר מכן, הפונקציה מגדירה checksum להודעה, לפני שליחתה.

A screen shot of a computer program

Description automatically generated

ננסה לשלוח את הסגמנט MAX\_RETRIES פעמים או עד אשר ההודעה תעבור בהצלחה (הגדרת "עברה בהצלחה" בתמונה הבאה).

A screenshot of a computer program

Description automatically generated

על כל שליחת הודעה נרצה לקבל ACK, לכן נקצה מקום להודעת ה-ACK, נקבל הודעה ונבדוק אם flag ה-ACK שלה אינו דלוק ואם ה-segment\_num שלה אינו תואם ל-segment\_num של ההודעה ששלחנו. זאת מכיוון שלכל הודעה אנו מצפים לקבל ACK שתואם לאותה הודעה. (אם ה-ACK לא תואם נתעלם ממנו.)

במידה ושני התנאים האלו אינם מתקיימים (וה-checksum תקין), הרי שזו הודעת ACK תקינה לסגמנט הנוכחי. נגדיר את השליחה כשליחה מוצלחת ולא נשלח סגמנט זה יותר (נצא מלולאת ה-for).

אם יצאנו מהלולאה בסיום הנסיונות, הרי שלא הצלחנו לשלוח הודעה. נחזיר -1, שיסמן לנו שגיאה.

אם הצלחנו לשלוח את הסגמנט בהצלחה, נחזיר את גודלו. יועיל לנו בעתיד לחישוב אם כל הקובץ עבר בשלמותו.

A screenshot of a computer program

Description automatically generated

פונקציה זו מקבלת Sender socket, מידע, וגודל המידע. מפצלת את המידע לסגמנטים ושולחת כל סגמנט ל-Receiver.

ראשית נקצה מקום בזיכרון להודעה שאותה נשלח (מקום להודעה + Header) עבור rudp\_send\_segment.

לאחר מכן נמיר את המצביע ל-data ל-char\*, שכן הפונקציה מקבלת void\*.

נשים לב שבשליחה אחת ניתן לשלוח מקסימום MAX\_SEGMENT\_SIZE בייטים. כך שבמידה וגודל הקובץ עולה על גודל זה, נפצל אותו למספר סגמנטים ונשלח את הסגמנטים (באמצעות הפונקציה הקודמת) עד שכמות הבייטים שנשלח יהיו שווים לגודל הקובץ.

לבסוף נשחרר את הזיכרון שהקצנו להודעה ונחזיר את גודל הקובץ שנשלח.

A computer screen with text

Description automatically generated

פונקציה זו מקבלת Receiver socket, buffer וגודל ה-buffer ונועדה ע"מ לקבל מידע דרך ה-Receiver socket.

ראשית נגדיר זמני Timeout ל-Socket, נמיר את המצביע למידע ל-char\* ונקצה מקום לקבלת סגמנט אחד של מידע.

נגדיר את משתנה ה-boolean ל-True, כדי שהלולאה בתמונה הבאה תתקיים.

בנוסף נגדיר שה-Timeout של ה-Socket הוא אינסופי, שכן בהתחלה אנחנו רוצים להמתין זמן לא מוגבל עד לקבלת ההודעה הראשונה.

A screen shot of a computer program

Description automatically generated

כל עוד נדרש לקבל עוד סגמנטים, נהיה בתוך לולאת ה-while.

בלולאה נקבל מידע בגודל מקסימלי של MAX\_SEGMENT\_SIZE מה-Sender.

לאחר שנקבל את ההודעה הראשונה נשנה את Timeout ה-Socket ל-Timeout שאינו אינסופי, שכן אנו רוצים שבמידה ולא קיבלנו את המשך המידע במשך זמן מסוים, ה-Socket יסגר.

A screenshot of a computer program

Description automatically generated

עבור כל הודעה נוודא שהיא תקינה (גודל תקין, Checksum תקין, מס' בייטים שהתקבלו חיובי ממש).

במידה וכל התנאים מתקיימים וההודעה אכן תקינה, נבדוק את flag ה-More ב-Header שלה ונגדיר לפיו את משתנה ה-more המקומי שיגדיר אם נדרש לקבל עוד מידע (עוד סבב של הלולאת while).

לאחר מכן, נכין הודעת ACK עם ה-segment\_size של ההודעה שהתקבלה (כלומר, זה ה-ACK שמתאים ספציפית להודעה הזו).

בנוסף, אם ההודעה היא הודעת FIN או SYN, נוסיף ל-Header של הודעת ה-ACK שלנו גם את flag ה-SYN או ה-FIN כדי להחזיר הודעה מתאימה.

A screen shot of a computer program

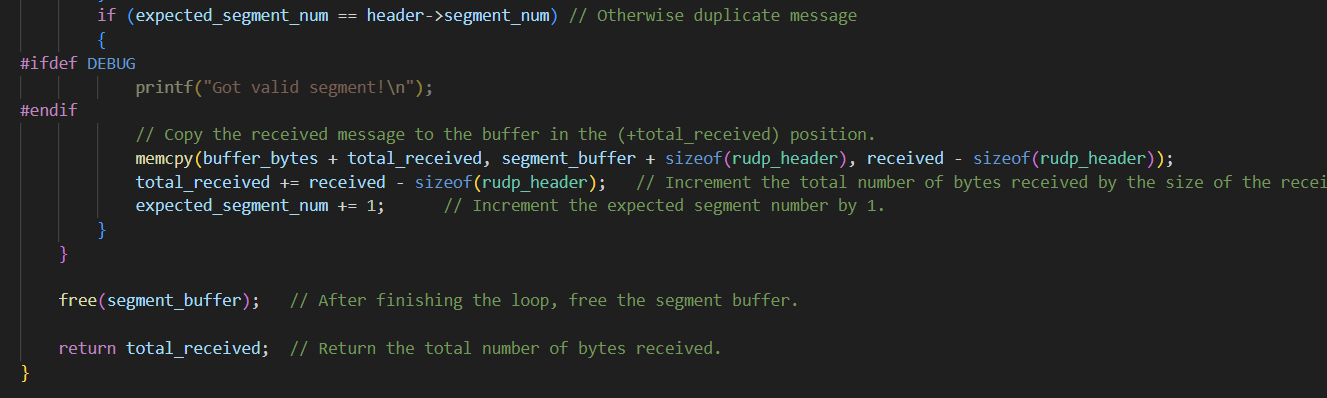
Description automatically generated

לפני שליחת ההודעה נגדיר לה את שדה ה-Checksum.

ננסה לשלוח את הודעת ה-ACK ל-Sender, MAX\_RETRIES פעמים או עד שההודעה תישלח בצורה תקינה (במידה ושליחת ההודעה תיכשל, נדלג על השורה ב-successful = TRUE והלולאה תמשיך לאיטרציה נוספת).

במידה ושלחנו הודעת FIN ACK, נשחרר את הזיכרון שהוקצה ל-buffer ונחזיר -1 שמסמן סיום תקשורת מצד ה-Receiver Socket.

במידה וכמות הבייטים שהתקבלו גדולה מגודל ה-buffer שהוקצה, נדפיס שגיאה קריטית ונסיים את ריצת התוכנית.



הבדיקה האחרונה לפונקציה זו היא אם ה-expected\_segment\_num שווה ל-segment\_num של ההודעה שקיבלנו. כלומר ההודעה שקיבלנו היא הבאה לה אנחנו מצפים.

אם זו אכן ההודעה לה אנחנו מצפים, נעתיק את הסגמנט ל-buffer, נגדיל את סוכם הבייטים שהתקבלו ונשנה את מס' הסגמנט שאנחנו מצפים לקבל לסגמנט הבא.

לבסוף נשחרר את הזיכרון שהקצנו ל-segment buffer ונחזיר את כמות הבייטים שקיבלנו.

## הסבר קוד RUDP\_Sender.c:

מכיוון שרוב הקוד של תוכנית זו זהה לקוד של TCP\_Sender אותו כבר הסברנו. נסביר רק את החלקים ששונים ביניהם.

A screen shot of a computer code

Description automatically generatedלאחר קבלת הארגומנטים וביצוע Parsing על כל אחד מהם, נפתח RUDP Sender socket (בו מבוצע גם ההתחברות ל-Receiver וה-Handshake)

לאחר מכן, נצהיר על גודל הקובץ אותו אנו מתכננים לשלוח ל-Receiver.

A computer screen with text on it

Description automatically generatedב-RUDP, פונקציית ה-rudp\_send מבצעת את הסגמנטציה בתוכה, לכן בהפעלה אחת של הפונקציה כל הקובץ נשלח.

נשלח את הקובץ כמות הפעמים שהמשתמש יבקש.

A screen shot of a computer code

Description automatically generatedלבסוף, נשחרר את הזיכרון שהקצאנו להודעה ונבצע סגירה של ה-Socket (שתשלח גם הודעת FIN ל-Receiver).

## הסבר קוד RUDP\_Receiver.c:

גם בתוכנית זו רוב התוכנית זהה ל-TCP\_Receiver שעליו כבר כתבנו, לכן נראה את השינויים העיקריים:

A computer screen with white text

Description automatically generatedלאחר ביצוע Parsing לארגומנטים שהתקבלו לתוכנית, נפתח את ה-RUDP Receiver socket על הפורט שהתקבל ונתחיל להאזין לתקשורת בפורט זה.

נמשיך ברגע שתסתיים לחיצת הידיים.

A computer screen with text on it

Description automatically generatedנמתין לקבלת ההודעה הראשונה שכוללת הצהרה על גודל הקובץ שישלח.

לאחר שהתקבל הגודל, נמיר אותו ל-size\_t ע"י הקצאתו למשתנה buffer\_size.

A screen shot of a computer program

Description automatically generated

נקצה מקום בגודל buffer\_size ל-buffer ונמתין לקבלת הודעות כל עוד לא התקבלת הודעת סיום.

נשים לב שפונקציית rudp\_recv מבצעת את עיבוד הסגמנטים בתוכה ובסיומה מחזירה את גודל כל הקובץ (ומבצעת השמה של המידע ל-buffer).

עבור כל קובץ שנקבל נבדוק אם קיבלנו -2, סימן שהייתה שגיאה.

אם קיבלנו -1, זו הודעת FIN.

אם קיבלנו 0, יתכן שזו הייתה הודעת SYN לא רלוונטית, לא נכליל אותה בחישוב הזמנים והמהירויות ונמשיך בקבלת שאר הקובץ.

A screen shot of a computer program

Description automatically generatedנמשיך בבדיקת שגיאות ונוודא שגודל הקובץ שקיבלנו זהה לגודל הקובץ שהוצהר בתחילת הריצה.

לאחר מכן נבצע חישובי זמנים ומהירויות.

אם יצאנו מהלולאה, התקבלה הודעת סיום. נסגור את ה-RUDP Receiver socket (הודעת ה-FIN ACK נשלחת לפני כן בתור תגובה להודעת ה-FIN של ה-Sender).

נדפיס סטטיסטיקות ונשחרר את הזיכרון שהקצנו במהלך הריצה.

# הסבר הקלטות

## הסבר אופן פעולת RUDP

בחלק זה נתייחס להקלטה “RUDP segmentation and retransmission” ונסביר את אופן הפעולה הכללי של פרוטוקול RUDP, הקלטה זו נוצרה באמצעות תוכנות **rudp\_sender\_test** ו-**rudp\_receiver\_test** שבנינו על מנת לבדוק את הפרוטוקול, התוכנות מבצעות שליחה פשוטה של מחרוזות. לצורך ההדגמה הורדנו את MAX\_SEGMENT\_SIZE ל-10 (על מנת שתתבצע סגמנטציה) והגדרנו packet loss של 10% בעזרת tc על מנת להדגים retransmission.

ראשית נראה את כל התקשורת:

A screen shot of a computer

Description automatically generated

שימו לב שכל הודעה באורך 8 היא הודעה של header בלבד (SYN/FIN/ACK).

ראשית שתי ההודעות הראשונות הן לחיצת הידיים:

A screen shot of a computer

Description automatically generatedA black screen with white text

Description automatically generated

בהודעה הימנית (SYN) ניתן לראות שלאחר 4 ספרות ראשונות (אורך החבילה – 0 – בשני בתים) נמצאות שתי הספרות של ה-flag שהן 01 בהתאמה ל-flag SYN בלבד. ניתן לראות מהפורטים שההודעה נשלחה מה-sender ל-receiver (פורט ה-receiver הוגדר להיות 1025).

בהודעה השמאלית (SYN-ACK) ניתן לראות באותן ספרות של ה-flags 03 בהתאמה ל- SYN | ACK (00000001 | 0000010 = 00000011 = 0x03).

בנוסף ניתן לראות בכל אחת מההודעות checksum (לדוגמא 0xfeff) ו-segment\_num (0 בשניהם).

לאחר לחיצת הידיים ההודעה הבאה היא הסגמנט הראשון:

A black screen with white text

Description automatically generated

ניתן לראות שבשדה len קיים 10 (0x0a00 בהקסה ב-little-endian) מכיוון שזה האורך המקסימלי, שדה flags הוא 8 מכיוון שזה הערך של flag MOR שמסמן שזה אינו הסגמנט האחרון בהודעה, ה-checksum הוא איזשהו מספר שתלוי בכל ההודעה, ו-segment\_num הוא 0 (זה הסגמנט הראשון בהודעה). את שאר המידע נראה מצד ימין של החלון:



ניתן לראות את החלק הראשון של ההודעה ששלחנו “Hello thro”, נראה את שאר ההודעה בסגמנט הבא.

לאחר מכן קיימת הודעת ACK מה-receiver:

A screen shot of a computer

Description automatically generated

ניתן לראות שהיא דומה להודעת SYN-ACK רק שהשדה flags מכיל רק 2 (רק דגל ACK), שדה segment\_num מכיל 0 מכיוון שזה ACK לסגמנט הראשון.

לאחר שקיבלנו ACK על ההודעה הראשונה מנגנון stop-and-wait עובר להודעה השנייה.

רואים שההודעה השנייה נשלחה 4 פעמים (בתמונה הראשונה), ניתן להסיק שזאת מכיוון שבשלושת הפעמים הראשונות נאבד ה-ACK, נראה את אחת מהן:

A screen shot of a computer

Description automatically generated

ניתן לראות כאן דברים דומים להודעה הראשונה – אין דגלים והאורך הוא 10, אך בניגוד ניתן לראות ששני הבתים האחרונים של ה-header (0100) מכילים segment\_num 1 (ב-little endian) מכיוון שזה הסגמנט השני ושדה flags הוא 0 מכיוון שה-flag MOR כבוי כי זה הסגמנט האחרון.

נראה את תוכן ההודעה בצד ימין של החלון:



ניתן לראות כאן את סוף ההודעה: “ugh RUDP!” ובסוף null-terminator (00 בבית האחרון).

כמו שאמרנו, ניתן לראות שאותה הודעה נשלחה 4 פעמים, עד שלבסוף התקבלה הודעת ACK:

A black screen with white text

Description automatically generated

ניתן לראות שבדומה ל-ACK הראשון ה-len הוא 0 ושדה flags הוא 2, אך כאן בדומה להודעה שזה ה-ACK שלה ב-segment\_num יש 1 (0x0100 ב-little-endian).

לבסוף קיימות שתי הודעות סגירה:

A black screen with white text

Description automatically generated A black screen with white text

Description automatically generated

ניתן לראות שהן דומות מאוד להודעות הפתיחה מלבד ששדה flags הוא 4 ב-FIN (צד ימין) ו-6 ב-FIN-ACK בהתאמה למה שהיינו מצפים מאיך שהשדות מוגדרים – שדה FIN הוא 4 ו-bitwise-or שלו עם שדה ACK שהוא 2 ייתן 6.

לאחר שראינו את התקשורת נגלה את פלט התוכנות:

A screenshot of a computer

Description automatically generated

ניתן לראות שלאחר פתיחת הקשר ה-sender ביקש הודעה ונתנו לו את ההודעה “Hello through RUDP!”, הוא ניסה לשלוח אותה, כולל נסיונות חוזרים לאחר שהוא לא קיבל ACK 3 פעמים (כמו שראינו בהקלטה) ולבסוף ה-receiver קיבל את ההודעה והציג אותה בשלמותה (לאחר שפרוטוקול RUDP חיבר בחזרה את הסגמנטים).

לאחר מכן נתנו ל-sender את הפקודה “close” שגרמה לו לסגור את את התקשורת ושתי התוכנות דיווחו על סגירה.

## הרצת RUDP\_Sender ו-RUDP\_Receiver

כעת, נראה פעולה תקינה של הפרוטוקול:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

בטרמינל השמאלי ניתן לראות פתיחת תקשורת של ה-RUDP Receiver, ההודעה הראשונה שהתקבלה היא גודל הקובץ אליו ה-Receiver נדרש לצפות.

לאחר מכן, עבור כל פעם שישלח הקובץ, יודפס מס' הפעמים שהקובץ נשלח (מתחיל מ-0), הזמן שלקח לקבלת הקובץ ומהירות השליחה.

כאשר ה-Sender מסיים את התקשורת, נדפיס סיכום של הריצה, ממוצעי מהירות, זמן ומס' ריצות כולל.

בטרמינל הימני ניתן לראות את ריצת ה-RUDP Sender. נשים לב שאם המשתמש לא מגדיר IP לתוכנית, קיים IP דיפולטיבי (כמו שניתן לראות בתמונה, למרות שלא נתנו IP התוכנית עדיין רצה).

התוכנית תשלח את הקובץ ותשאל את המשתמש האם לשלוח שוב.

התוכנית תמשיך לשלוח את הקובץ כל עוד המשתמש מקליד ‘y’.

כאשר המשתמש יקליד ‘n’, תופעל פונקציית סגירת ה-Sender socket שבמהלכה תישלח הודעת FIN ל-Receiver ושני ה-Sockets יסיימו את התקשורת ביניהם.

## הסבר הקלטות TCP:

בחלק זה נתייחס להקלטה TCP Wireshark capture”" שמצורפת (הקלטה יחידה לכל החלק הנ"ל).

ראשית נראה את הפעילות התקינה של התוכנה:

נפעיל את ה-TCP\_Receiver להאזנה בפורט 4000 בשימוש באלגוריתם reno.

הוספנו מוד format שבו יש פחות הדפסות. בהרצה הנוכחית המוד הזה כבוי.

לאחר מכן, נפעיל את ה-TCP\_Sender על אותו פורט עם אלגוריתם TCP Congestion זהה (נשים לב שבקוד של ה-TCP\_Sender הגדרנו IP דיפולטיבי במידה והמשתמש אינו מספק IP, לכן הצלחנו לתקשר עם ה-Receiver למרות שלא סופק IP כארגומנט).

נשלח את אותה חבילה בגודל 2MB חמישה פעמים ולבסוף נסגור את התקשורת ע"י סיום שליחת הקובץ.

בטרמינל של ה-TCP\_Receiver ניתן לראות ב-live את קבלת החבילות, את הזמן שלקח לכל חבילה להגיע, את המהירות ואת מס' החבילה (כאשר כל חבילה היא קובץ שלם, לא סגמנט מתוך קובץ).

כאשר ה-Sender מסיים את התקשורת, נדפיס סיכום של הריצה וממוצעי מהירות, זמן ומס' ריצות כולל.

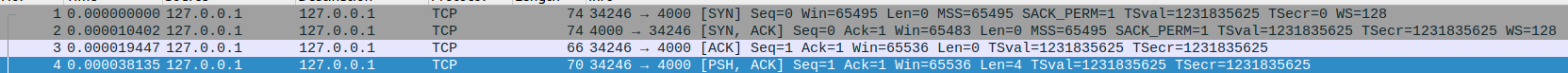
A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

לאחר תחילת ההתקשרות בין ה-Sender ל-Receiver, ה-Sender שולח חבילה ראשונה שכוללת הצהרה על גודל הקובץ אותו היא תעביר בהמשך. זאת לטובת הקצאת זיכרון מתאים ב-buffer של ה-Receiver.



A screenshot of a computer

Description automatically generatedנשים לב שה-Data שנשלח הוא 2MB ביצוג שלהם בבייטים ב-Hexadecimal.

לאחר מכן תתחיל השליחה של הקובץ במס' סגמנטים מה-Sender ל-Receiver. לא נעלה תמונה של זה כי כל מה שרואים זה רצף של הודעות PSH והודעות ACK עם מידע אקראי.

המשתמש יבחר כמה פעמים לשלוח את הקובץ. ה-Receiver ימשיך להאזין עד אשר תתקבל הודעת סיום התקשורת.

בסיום הריצה של ה-Sender נשלחת חבילה שכוללת את המלל “Closing connection” כדי להצהיר ל-Receiver על סיום התקשורת ביניהם (כדי שה-Receiver לא ימתין לקבלת חבילות נוספות).

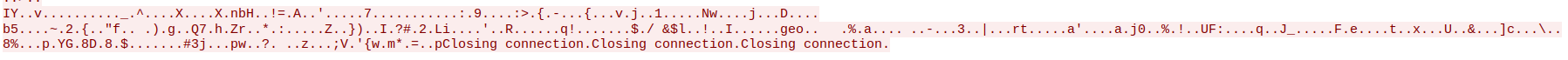
ניתן לראות ב-Wireshark את החבילה הזאת לפני סיום התקשורת (לפני ה-3 Way Handshake של TCP).

כמו כן, כל המלל שנמצא לפני ה-Closing connection הן שכבות של הפרוטוקולים שעטפו את החבילה (TCP, IP וכו').

A screenshot of a computer screen

Description automatically generated

ה-Sender ישלח את הודעת הסיום כמות מסוימת של פעמים (שמוגדרת כקבוע בקוד), ניתן לראות זאת בצורה ברורה ב-TCP Stream:



נזכיר שההודעה הזו נשלחת מס' פעמים כדי להתגבר על Packet Loss.

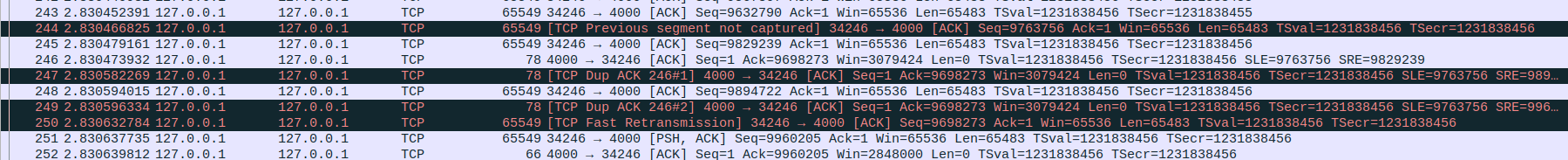
לאחר שהודעות הסגירה ישלחו, ה-Sender ישלח הודעת FIN (הודעת FIN של TCP) ושני ה-Sockets יבצעו לחיצת יד משולשת לפני סגירה:

A screenshot of a computer

Description automatically generated

בתמונה ניתן לראות את לחיצת היד המשולשת ואת הודעת ה-FIN הראשונה שבעצם מתחילה את סגירת התקשורת (כפי שניתן לראות ב-Expert Info).

כעת, נגלה שבמהלך כל השליחה היה Packet Loss ונראה איך זה השפיע על שליחת הנתונים ואיך ה-TCP התגבר עליו:

נביט בחלק מעניין מאמצע שליחת אחד הקבצים:

נאפיין חבילה לפי 3 הספרות הראשונות ב-Seq number שלה.

נשים לב שבשורה 243 נשלחה החבילה ה-963. בשורה 244, Wireshark מתריע שלא התקבלה חבילה מכיוון שבשורה זו התקבלה חבילה 976 (Wireshark מזהה את הדילוג ולכן מתריע). למרות זאת בשורה 245 ממשיכה השליחה לחבילה 982.

בשורה 246 ה-Receiver שולח ACK על כל החבילות עד 969, כלומר החבילה הבאה שה-Receiver מצפה לקבל היא 969 (שזו החבילה שקיבלנו התרעה שלא התקבלה מקודם).

בשורה 247 אנחנו שולחים ACK חוזר על כך שקיבלנו את כל החבילות על 969. זהו ACK על החבילה שקיבלנו בשורה 244, שאינה 969.

בשורה 248, ה-Sender עדיין ממשיך לשלוח לנו את החבילה הבאה (289), שכן היא נמצאת בחלון שלו.

בשורה 249, ה-Receiver שוב שולח ACK, בהתאמה לחבילה 982. מס' ה-ACK הוא של חבילה 969 כדי להתריע ל-Sender על כך שה-Receiver עדיין מצפה לקבל את חבילה 969.

לבסוף, מאחר שהתקבלו שלושה ACK-ים על אותה חבילה. בשורה 250 מתבצע TCP Fast Retransmission בו ה-Sender שולח את חבילה 969.

בשורה 251, ה-Sender כבר מתקדם לחבילה הבאה ושולח את 996.

בשורה 252, ה-Receiver שולח ACK על חבילה 969. ניתן לראות שמס' ה-ACK הוא 996, שכן את כל החבילות בין 969 ל-996 ה-Receiver קיבל לפני ה-fast retransmission ועדיין זוכר אותן (חבילה 996 נשלחה מה-Sender במקביל ל-ACK שנשלח מה-Reciever).

# חלק ג' – מחקר

עבור חלק זה יצרנו סקריפט פייתון – Measurements.py שאחראי על הגדרת packet loss, הרצת תוכניות ואיסוף הנתונים, ספציפית בעזרת דגלי auto ו-format שיצרנו למטרה זו. הסקריפט מצורף בקבצים שהוגשו.

לא נוסיף הסבר לסקריפט Measurements.py כאן, קיימות בו הערות לעיונכם.

לאחר ההרצה אנחנו מקבלים את התוצאה הבאה:

A screenshot of a computer screen

Description automatically generated

ניתן לראות בכל טבלה באיזה פרוטוקול השתמשנו (TCP/RUDP) ועבור TCP באיזה אלגוריתם השתמש כל צד. כאשר R מסמל Receiver ו-S עבור Sender.

נענה על השאלות שמועלות במטלה:

1. ניתן לראות ש-cubic נתן תוצאות טובות ב-packet loss נמוך וגבוה, במיוחד לפי רוחב הפס שהוא הגיע אליו – בערך פי 2 ללא איבוד ובערך פי 1.25 עם איבוד גבוה.

כמו כן ניתן לראות שבשילוב פרוטוקולים שונים מתקבלת תוצאה טובה יותר באיבוד נמוך אך הפער מצטמצם באיבוד גבוה, ב-5% וב-10% ניתן לראות שהמרכיב המשמעותי הוא הפרוטוקול של השולח דווקא.

1. מימוש RUDP שלנו משיג ביצועים קרובים ל-TCP באיבוד נמוך (בחלק מהמקרים) אך כשהאיבוד עולה הוא נשאר מאחורה, כנראה מכיוון שלא מימשנו flow control מתקדם ולכן כל חבילה שהולכת לאיבוד מכריחה המתנה של ה-timeout.

נעדיף RUDP במקרים של איבוד נמוך מכיוון שהוא מהיר יותר בשליחה בודדת, אך מכיוון שמנגנון flow control שלנו פשוט, כאשר מגיעים לאיבוד גבוה עדיף TCP.

# שאלות תיאורטיות

1. על מנת ש-SSThreshold גבוה יועיל לנו ראשית צריך רשת אמינה, אחרת יתחילו איבודים לפני שנגיע אליו ולא נרוויח מההגדלה. שנית צריך קשר ארוך על מנת שגודל החלון באמת יגיע ל-SSThreshold בזמן הקשר. לבסוף ההגדלה תועיל לנו בצורה המירבית בקשר עם RTT גדול, מכיוון שאם נגיע למצב שה-transmission delay הכולל של החבילות בחלון גדול מה-RTT לא נרוויח יותר ביצועים (נחכה ל-transmission ולא ל-ackים), ב-RTT גדול החלון יוכל לגדול יותר ולהרוויח יותר ביצועים.

על כן התשובה הנכונה היא 1. בקשר ארוך על גבי רשת אמינה עם RTT גדול.

1. על מנת לחשב את התפוקה במהלך הקשר (התפוקה הממוצעת) נחשב את סך המידע שעובר בקשר ואת סך הזמן של הקשר.

ראשית עבור סכום המידע (בבתים), ב-RTT הראשון נשלח 1MSS, בשני 2MSS וכן הלאה בכפולות של 2 עד , נכתוב זאת בנוסחה:

כעת עבור הזמן הכולל (בשניות), נרצה לספור את כמות ה-RTT שעוברים עד שגודל החלון מגיע ל-S, מכיוון שהחלון גדל פי 2 בכל RTT מספר הפעמים שהוא יוכפל יהיה בערך ולכן

על מנת לחשב את התפוקה הכוללת נחלק ביניהם:

ולכן התשובה הנכונה היא א'.

1. X שלנו הוא 3.

על מנת להבטיח תפוקה מקסימלית, החלון צריך להיות גדול מספיק כך שהשולח לא "יפצח גרעינים" בין הזמן שהוא מסיים לשלוח את המידע בחלון הנוכחי עד שהוא מקבל ack על תחילת החלון. כלומר סכום ה-transmission delays של החבילות בחלון מקסימלי שווה לפחות ל-RTT.

נחשב כל אחד בעזרת הנוסחה

ראשית RTT שווה לפעמיים הזמן שלוקח למידע לעבור בין התחנות, לפי הנתונים לגבי מרחק התחנות וקצב ההתפשטות:

שנית, נחשב transmission delay עבור חבילה בודדת. לפי הנתונים לגבי גודל החבילה וקצב התקשורת, לאחר שנחלק את קצב התקשורת ב-8 על מנת להמיר מ-Gbps ל-GB/s נקבל:

עבור תפוקה מקסימלית נרצה לבחור N כך ש- כלומר לאחר העברת אגפים: מכיוון ש-N שלם נדרוש וזה צריך להיות גודל החלון (לכל הפחות) על מנת להבטיח תפוקה מקסימלית.