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 ב-Makefile.

הקלטות:

בתיקיית Wireshark captures קיימות 2 הקלטות עיקריות:

RUDP segmentation and retransmission: הקלטה של הרצת 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:

```c
#define FILE_SIZE 2097152  //The size of the file to be sent - currently 2MB
#define REPEAT_END_MESSAGE 3    //The number of times the ending message would be sent
#define FIN_MESSAGE "Closing connection"    //The message to be sent to the receiver to indicate the end of connection

#define RECV_TIMEOUT_US 10000   //The timeout for the socket to receive data in microseconds
#define RECV_TIMEOUT_S 2        //The timeout for the socket to receive data in seconds
```

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 2MB).

פונקציה נתונה שמחזירה כתובת למערך שמכיל תווים רנודמליים בגודל נתון.

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

```c
/*
 * @brief A random data generator function based on srand() and rand().
 * @param size The size of the data to generate (up to 2^32 bytes).
 * @return A pointer to the buffer.
 */
char *util_generate_random_data(unsigned int size)
{
    char *generated_file = NULL;
    // Argument check.
    if (size == 0)
        return NULL;
    generated_file = (char *)calloc(size, sizeof(char));
    // Error checking.
    if (generated_file == NULL)
        return NULL;
    // Randomize the seed of the random number generator.
    srand(time(NULL));
    for (unsigned int i = 0; i < size; i++)
        *(generated_file + i) = ((unsigned int)rand() % 256);
    return generated_file;
}
```

```c
int main(int argc, char **argv)
{
    if(argc == 1)   //If no arguments were given
    {
        printf("Error! No arguments were given\n");
        return 1;
    }
    size_t i = 0;
    char *ip = "127.0.0.1";
    unsigned short port = 0;
    char *algo = NULL;
    unsigned short auto_run = 0;
    while ((int)i < argc)
    {
        if (strcmp(argv[i], "-ip") == 0)
        {
            ip = argv[++i];
        }
        else if (strcmp(argv[i], "-p") == 0)
        {
            port = (unsigned short)atoi(argv[++i]);
            if (port < 1024)
            {
                printf("Invalid port\n");
                return 1;
            }
        }
        else if (strcmp(argv[i], "-algo") == 0)
        {
            if (strcmp(argv[i + 1], "reno") != 0 && strcmp(argv[i + 1], "cubic") != 0)
            {
                printf("Invalid algorithm\n");
                return 1;
            }
            algo = argv[++i];
        }
```

```c
        else if (strcmp(argv[i], "-auto") == 0)
        {
            auto_run = (unsigned short)atoi(argv[++i]);
        }
        i++;
    }
    if (port == 0 || algo == NULL)
    {
        printf("Invalid arguments\n");
        return 1;
    }
```

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 (INET_AF) 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.

```c
struct sockaddr_in receiver;
int sock = -1;
char *message = util_generate_random_data(FILE_SIZE); // Generate a random message of FILE_SIZE bytes.
memset(&receiver, 0, sizeof(receiver));               // Zero out the receiver structure

sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1)
{
    perror("The socket has failed");
    free(message);
    return 1;
}

// Convert the server's address from text to binary form and store it in the server structure.
// This should not fail if the address is valid (e.g. "127.0.0.1").
if (inet_pton(AF_INET, ip, &receiver.sin_addr) <= 0)
{
    perror("inet_pton(3)");
    close(sock);\
    free(message);
    return 1;
}
```

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

```c
// Set the receiver's address family to AF_INET (IPv4).
receiver.sin_family = AF_INET;

// Set the reciever's port to the defined port. Note that the port must be in network byte order,
// so we first convert it to network byte order using the htons function.
receiver.sin_port = htons(port);

socklen_t len = strlen(algo);
if (setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, len) != 0) // Set the algorithm of congestion control the socket would use.
{
    perror("setsockopt");
    free(message);
    return -1;
}
// A check to see if the congestion control algorithm passed successfully.
if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, &len) != 0)
{
    perror("getsockopt");
    free(message);
    return -1;
}
struct timeval timeout;
timeout.tv_sec = RECV_TIMEOUT_S;
timeout.tv_usec = RECV_TIMEOUT_US;
if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0)
{
    perror("Error setting timeout for the socket\n");
    close(sock);
    return 1;
}
```

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.

```c
// Try to connect to the receiver using the socket and the receiver structure.
if (connect(sock, (struct sockaddr *)&receiver, sizeof(receiver)) < 0)
{
    perror("connect(2)");
    close(sock);
    return 1;
}

//-------------------------------START SENDING INFORMATION--------------------------------------------

printf("Successfully connected to the receiver!\n");
// Sending the size of the input to the receiver.
uint32_t report_file_size = htonl(FILE_SIZE);
int starting_message = send(sock, (void *)&report_file_size, sizeof(report_file_size), 0);
if (starting_message <= 0)
{
    perror("Error sending the file size\n");
    close(sock);
    return 1;
}
```

נתחבר ל-Reciver באמצעות ה-sockaddr_in receiver 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.

```c
do
{
    // Try to send the message to the receiver using the socket.
    bytes_sent = send(sock, message, FILE_SIZE, 0);

    // If no data was sent, print an error message and return 1. Only occurs if the connection was closed.
    if (bytes_sent <= 0)
    {
        perror("send(2)");
        close(sock);
        return 1;
    }
    fprintf(stdout, "Sent %d bytes to the receiver!\n", bytes_sent);
    printf("Do you want to send again? (Y/n)\n");
    do
    {
        choice = getchar(); //Getting a char from the user untill he writes 'y', 'Y', 'n', 'N'
    } while (choice != 'n' && choice != 'N' && choice != 'y' && choice != 'Y');
} while (choice != 'n' && choice != 'N');
```

```c
free(message);
char *ending_message = FIN_MESSAGE;
for (size_t i = 0; i < REPEAT_END_MESSAGE; i++) //Sending the
{
    // Try to send the message to the receiver using the socket.
    bytes_sent = send(sock, ending_message, strlen(ending_message) + 1, 0);
    // If no data was sent, print an error message and return 1. Only occurs if the connection was closed.
    if (bytes_sent <= 0)
    {
        perror("send(2)");
        close(sock);
        return 1;
    }
}
printf("Sent ending messege to the receiver\n");

// Close the socket with the receiver.
close(sock);

printf("Connection closed!\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:

```c
#define TRUE 1
#define FALSE 0

#define MAX_CLIENTS 1          // The maximum number of clients that can connect to the receiver.
#define RECV_TIMEOUT_US 100000
#define RECV_TIMEOUT_S 2

#define FIN "Closing connection"
```

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

```c
/*
 * @brief A struct that will act as an ArrayList.
 */
typedef struct
{
    double *data;
    size_t capacity;
    size_t size;
} ArrayList;
```

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.

```c
/*
 * @brief Add a number to the list.
 * @param list The list to add the number to.
 * @param num The number to add to the list.
 */
void addToList(ArrayList *list, double num)
{
    if (list->size == list->capacity)
    {
        list->data = realloc(list->data, sizeof(double) * (list->capacity *= 2));
    }
    list->data[list->size++] = num;
}
```

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.

```c
/*
 * @brief Convert the bytes to MegaBytes.
 * @param bytes The amount of bytes.
 * @return The amount of MegaBytes.
 */
double convertToMegaBytes(size_t bytes)
{
    size_t converstion = 1024 * 1024;
    return bytes / converstion;
}
/*
 * @brief Convert the bytes and time to speed in MB/s.
 * @param bytes The amount of bytes received.
 * @param time The time it took to receive the bytes.
 * @return The speed in MB/s.
 */
double convertToSpeed(double bytes, double time)
{
    return convertToMegaBytes(bytes) / (time / 1000);
}
```

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.

```c
/*
 * @brief Close the sockets and print a message that they were closed.
 * @param sock The main receiver main socket.
 * @param client_sock The client socket.
 * @param format a boolean that says if how the output should be printed.
 * @param sender The sender's address.
 */
void CloseSockets(int *sock, int *client_sock, unsigned short format, struct sockaddr_in sender)
{
    close(*client_sock);
    if (!format)
        printf("Client %s:%d disconnected\n", inet_ntoa(sender.sin_addr), ntohs(sender.sin_port));
    close(*sock);
    if (!format)
        printf("Closing connection!\n");
}
```

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).

```c
/*
 * @brief Print the average time speed, and amount of runs of the messages received.
 * @param Times_list The list of times.
 * @param Speed_list The list of speeds.
 * @param run The amount of runs the program did.
 * @param format a boolean that says if how the output should be printed.
 */
void endPrints(ArrayList *Times_list, ArrayList *Speed_list, size_t run, unsigned short format)
{
    double avg_time = 0;
    double avg_speed = 0;
    for (size_t i = 0; i < Times_list->size && i < Speed_list->size; i++)
    {
        avg_time += Times_list->data[i];
        avg_speed += Speed_list->data[i];
    }
    avg_time = avg_time / Times_list->size;
    avg_speed = avg_speed / Speed_list->size;
    if (!format)
    {
        printf("Average time taken to receive a message: %f\n", avg_time);
        printf("Average speed: %f\n", avg_speed);
        printf("Number of runs: %ld\n", run);
    }
    if (format)
    {
        printf("Average,%f,%f\n", avg_time, avg_speed);
    }
}
```

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

```c
/*
 * @brief Free the memory allocated for the lists and the buffer.
 * @param Times_list The list of times.
 * @param Speed_list The list of speeds.
 * @param buffer The buffer.
 */
void endFree(ArrayList *Times_list, ArrayList *Speed_list, char *buffer)
{
    free(Times_list->data);
    free(Speed_list->data);
    free(buffer);
}
```

כעת נעבור ל-main:

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

```c
//------------------------------------------------ MAIN ------------------------------------------------------

int main(int argc, char **argv)
{
    //-------------------------------------PARSING THE INPUT-----------------------------------------------
    if (argc == 1)
    {
        printf("Cant run with no arguments\n");
        return 1;
    }
    size_t i = 0;
    unsigned short port = 0;
    char *algo = NULL;
    unsigned short format = FALSE; // when true, print only the run #, runtime, and throughput (mb / s)
    while ((int)i < argc)          // Parsing the arguments passed to the program.
    {
        if (strcmp(argv[i], "-p") == 0)
        {
            port = (unsigned short)atoi(argv[++i]);
            if (port < 1024)
            {
                printf("Invalid port\n");
                return 1;
            }
        }
        else if (strcmp(argv[i], "-algo") == 0)
        {
            if (strcmp(argv[i + 1], "reno") != 0 && strcmp(argv[i + 1], "cubic") != 0)
            {
                printf("Invalid algorithm\n"); // If the algorithm is not reno or cubic, print an error message and return 1.
                return 1;
            }
            algo = argv[++i];
        }
```

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.

```
    else if (strcmp(argv[i], "-format") == 0)
    {
        format = 1;
    }
    i++;
}
if (port == 0 || algo == NULL)
{
    printf("Invalid arguments\n");
    return 1;
}
```

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.

```
//-----------------------------------CONFIGURING SOCKETS-------------------
// The variable to store the socket file descriptor.
int sock = -1;

// The variable to store the receiver's address.
struct sockaddr_in receiver;

// The variable to store the sender's address.
struct sockaddr_in sender;

// Stores the sender's structure length.
socklen_t client_len = sizeof(sender);

// The variable to store the socket option for reusing the receiver's address.
int opt = 1;

// Reset the receiver and sender structures to zeros.
memset(&receiver, 0, sizeof(receiver));
memset(&sender, 0, sizeof(sender));

// Try to create a TCP socket (IPv4, stream-based, default protocol).
sock = socket(AF_INET, SOCK_STREAM, 0);
// If the socket creation failed, print an error message and return 1.
if (sock == -1)
{
    perror("socket(2)");
    return 1;
}
```

```
// Set the socket option to reuse the receiver's address.
// This is useful to avoid the "Address already in use" error message when restarting the receiver.
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0)
{
    perror("setsockopt(2)");
    close(sock);
    return 1;
}

// Set the receiver's address to "0.0.0.0" (all IP addresses on the local machine).
receiver.sin_addr.s_addr = INADDR_ANY;

// Set the receiver's address family to AF_INET (IPv4).
receiver.sin_family = AF_INET;

// Set the receiver's port to the specified port. Note that the port must be in network byte order.
receiver.sin_port = htons(port);

socklen_t len = strlen(algo);
if (setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, len) != 0) // Set the algorithm of congestion control the socket would use.
{
    perror("setsockopt");
    return -1;
}
// A check to see if the congestion control algorithm passed successfully.
if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, &len) != 0)
{
    perror("getsockopt");
    return -1;
}
```

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.

```
// Try to bind the socket to the receiver's address and port.
if (bind(sock, (struct sockaddr *)&receiver, sizeof(receiver)) < 0)
{
    perror("bind(2)");
    close(sock);
    return 1;
}

// Try to listen for incoming connections.
if (listen(sock, MAX_CLIENTS) < 0)
{
    perror("listen(2)");
    close(sock);
    return 1;
}
```

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

ונגדיר ל-Socket זה Timeout שהוגדר ברשימת הקבועים.

```
//----------------------CONFIGURING THE CLIENT SOCKET AND RECEIVING THE MESSAGES-----
int client_sock = -1;
// Try to accept a new sender connection.
client_sock = accept(sock, (struct sockaddr *)&sender, &client_len);
// If the accept call failed, print an error message and return 1.
if (client_sock < 0)
{
    perror("accept(2)");
    close(sock);
    return 1;
}

struct timeval timeout;
timeout.tv_sec = RECV_TIMEOUT_S;
timeout.tv_usec = RECV_TIMEOUT_US;
// Set a timeout for the receiver socket.
if (setsockopt(client_sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0)
{
    perror("Error setting timeout for the receiver socket\n");
    close(client_sock);
    close(sock);
    return 1;
}
```

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.

```
size_t run = 0;       // will count the amount of packeges received.
ArrayList Times_list; // A list to store the time it took to receive each message.
Times_list.data = malloc(sizeof(double));
Times_list.capacity = 1;
Times_list.size = 0;

ArrayList Speed_list; // A list to store the speed of each message received.
Speed_list.data = malloc(sizeof(double));
Speed_list.capacity = 1;
Speed_list.size = 0;
```

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.

```
int input_size = 0;
size_t sizeof_input;
while (!input_size)
{
    // Before sending the packege, the sender will send the weight of the package in bytes.
    input_size = recv(client_sock, &sizeof_input, sizeof(sizeof_input), 0);
}
sizeof_input = ntohl(sizeof_input);
if (!format)
    printf("Size of input: %ld bytes\n", sizeof_input);
size_t buffer_size = sizeof_input; // The size of the buffer will be the size of the input.
char *buffer = calloc(buffer_size, sizeof(char));
```

```
unsigned short noEndMessage = TRUE; // Indicator if the end message was received.
// The receiver's main loop.
while (noEndMessage)
{
    // A variable to store the amout of bytes received in each recv().
    int bytes_received;
    size_t total_of_bytes_received = 0;    // A variable to count the total amount of bytes received.
    clock_t start, end;
    double time_used_inMS;
    start = clock();
    while (total_of_bytes_received < sizeof_input && strcmp(buffer, FIN) != 0)
    {
        // Receive a message from the sender and store it in the buffer.
        bytes_received = recv(client_sock, buffer, buffer_size, 0);
        // Because there is no need to save the information in the buffer, in each run we will override the data in the buffer.
        // If we were to save the data, we would use the following line: recv(clent_sock, buffer + total_of_bytes_received, buffer_

        // If the message receiving failed, print an error message and return 1.
        if (bytes_received < 0)
        {
            perror("Reached timeout");
            CloseSockets(&sock, &client_sock, format, sender);
            endFree(&Times_list, &Speed_list, buffer);
            return 1;
        }
        if (bytes_received == 0) // If the sender disconnected, break the loop.
        {
            noEndMessage = FALSE;
            break;
        }
        total_of_bytes_received += bytes_received;
    }
    end = clock();
```

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.

```
time_used_inMS = 1000 * (double)(end - start) / CLOCKS_PER_SEC; // Calculating the time it took for the message to be received.
double speed = convertToSpeed(total_of_bytes_received, time_used_inMS);
if (time_used_inMS > 0 && speed > 0 && noEndMessage)
{
    // Add the time and speed to their respective lists.
    addToList(&Times_list, time_used_inMS);
    addToList(&Speed_list, speed);
}
```

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

```
if (!format)
{
    printf("Time taken to receive that messege: %f ms\n", time_used_inMS);
    printf("Speed: %f MB/s\n", speed);
}
```

נדפיס את הזמן שלקח לקבל את הקובץ הנוכחי ואת המהירות.

```
if (noEndMessage)
{
    run++; // Increment the run counter.
}
// If the received message is "Closing connection", close the sender's socket and return 0.
if (strcmp(buffer, FIN) == 0)
{
    noEndMessage = FALSE;
}
```

לאחר שקיבלנו קובץ, נבצע את הפעולות הבאות:

נגדיל את מונה הריצות שלנו ב-1.

נבדוק אם אם התקבלה הודעת סיום.

```
// Closing the socket, and printing the average time and speed before exiting the program.
CloseSockets(&sock, &client_sock, format, sender);
endPrints(&Times_list, &Speed_list, run - 1, format);
endFree(&Times_list, &Speed_list, buffer);
return 0;
```

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-O3 flag- 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:

```
Chaim Averbach, 5 days ago | 2 authors (uriel and others)
typedef struct {
    int sock;
    struct sockaddr peer_address;
    socklen_t peer_address_size;
} rudp_sender;

Chaim Averbach, 5 days ago | 2 authors (uriel and others)
typedef struct {
    int sock;
    struct sockaddr peer_address;
    socklen_t peer_address_size;
} rudp_receiver;
```

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.

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

נראה את מימושן בהמשך.

```c
/**
 * @brief Opens an RUDP receiver and waits for a connection from a sender.
 * Listens on the given port. and accepts the first connection.
 * Make sure to call rudp_close_receiver on the result to close and free
 * @param port the port the receiver will listen to
 * @return a pointer to the receiver or NULL on error
 * @note The default maximum number of clients is 1.
 */
rudp_receiver *rudp_open_receiver(unsigned short port);

/**
 * @brief Opens an RUDP sender and connects to a receiver in the given address and port
 * Make sure to call rudp_close_sender on the result to close and free
 * @param address the IP address to which the connection will be opened
 * @param port the port to which the connection will be opened
 * @return a pointer to the sender or NULL on error
 */
rudp_sender *rudp_open_sender(char *address, unsigned short port);
```

לאחר מכן הצהרות פונקציות הסגירה, שימו לב שפונקציה rudp_close_sender מבצעת את הסגירה בתקשורת (הודעות FIN).

```c
/**
 * @brief Closes the RUDP sender, Send a FIN to the associated receiver and closes the connection
*/
void rudp_close_sender(rudp_sender *sender);

/**
 * @brief Closes the RUDP receiver, Sends a FIN to the associated sender and closes the connection
*/
void rudp_close_receiver(rudp_receiver *receiver);
```

לאחר מכן ההודעות האחראיות על התקשורת.

שימו לב במיוחד לערכי ההחזרה המיוחדים של rudp_recv.

```c
/**
 * @brief Sends "size" bytes stored at "data" through the sender ("this")
 * @return the number of bytes sent,
 * @return -1 on error
 */
int rudp_send(rudp_sender *sender, void *data, size_t size);

/**
 * @brief Waits for a message from the sender, writes the message up to "size" bytes into "buffer"
 * @return the number of bytes received,
 * @return -1 on close message received,
 * @return -2 on error
 */
int rudp_recv(rudp_receiver *receiver, void *buffer, size_t size);
```

```c
// ENUM for bit masks in the flag field of rudp_header
enum rudp_header_flags {
    SYN = 1<<0,
    ACK = 1<<1,
    FIN = 1<<2,
    MOR = 1<<3,
};
// The struct of the header of the RUDP protocol
You, 27 seconds ago | 2 authors (You and others)
typedef struct {
    unsigned short len;         // size of the segment not including header
    char flags;                 // various flags of the segment
    unsigned short checksum;    // checksum of the whole segment (including header)
    unsigned short segment_num; // the number of the segment for segmented messages
} rudp_header;
```

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.

```
// debug modes
// #define DEBUG
// #define CHECKSUM_DEBUG
#define TRUE 1
#define FALSE 0
// parameters of the protocol
#define ACK_TIMEOUT_US 100000
#define ACK_TIMEOUT_S 0
#define RECV_TIMEOUT_US 0
#define RECV_TIMEOUT_S 2
#define MAX_RETRIES 15

#define IP_HEADER_SIZE 20
#define UDP_HEADER_SIZE 8
// the maximum segment size that can be transmitted over RUDP under UDP under IP
#define MAX_SEGMENT_SIZE ((1 << 16) - 1 - IP_HEADER_SIZE - UDP_HEADER_SIZE - sizeof(rudp_header))
```

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: $2^{16} - 1 - IP_{Header} - UPD_{Header} - RUDP_{Header}$

שכן, שדה ה-length ב-header של ה-IPv4 הוא באורך 16 ביטים, כך שהאורך המקסימלי של פאקטה יחידה הוא: $2^{16} - 1$.

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.

```
unsigned short calculate_checksum(void *data, unsigned int bytes)
{
#ifdef CHECKSUM_DEBUG
    printf(" checksum of data with size %d\n", bytes);
    printf("Data is: ");
    for (size_t i = 0; i < bytes; ++i)
        printf("%02x", ((char *)data)[i]);
    printf("\n");
#endif

    unsigned short *data_words = (unsigned short *)data;    // Data_words is a pointer to the data
    unsigned int sum = 0;

    while (bytes > 1)
    {
        sum += *(unsigned short *)data_words++;
        bytes -= 2;
    }

    /*  Add left-over byte, if exist */
    if (bytes > 0)
        sum += *(unsigned char *)data_words;

    /*  Fold 32-bit sum to 16 bits */
    while (sum >> 16)
        sum = (sum & 0xffff) + (sum >> 16);
#ifdef CHECKSUM_DEBUG
    printf("Calculated checksum is %hu\n", ~(unsigned short)sum);
#endif
    return ~(unsigned short)sum;    // Return the one's complement of the sum which is the 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.

```c
/**
 * Sets the checksum of a message
 */
void set_checksum(void *rudp_message)
{
#ifdef CHECKSUM_DEBUG
    printf("Setting checksum");
#endif
    rudp_header *header = (rudp_header *)rudp_message;
    header->checksum = 0;   // Set the checksum to 0, is used to calculate the checksum
    header->checksum = calculate_checksum(rudp_message, sizeof(rudp_header) + header->len); // Asign the checksum to the header
}

/**
 * Validates the checksum of a message
 */
int validate_checksum(void *rudp_message)
{
#ifdef CHECKSUM_DEBUG
    printf("Validating");
#endif
    return 0 == calculate_checksum(rudp_message, sizeof(rudp_header) + ((rudp_header *)rudp_message)->len); // Return true if the checks
}
```

```c
/**
 * Opens an RUDP sender and connects to a receiver in the given address and port
 * Returns a pointer to the sender.
 * Returns NULL on error, including no receiver accepting the connection.
*/
rudp_sender *rudp_open_sender(char *address, unsigned short port)
{
    rudp_sender *this = malloc(sizeof(rudp_sender));    // Allocate memory for the sender

    this->sock = socket(AF_INET, SOCK_DGRAM, 0);    // Create a socket for the sender
    if (this->sock == -1)                           // If the socket creation fails
    {
        perror("Error on socket creation at open_sender");
        free(this);
        return NULL;
    }

    struct sockaddr_in receiver_address = {0};  // Zero out the receiver address

    // Convert the address to a binary representation
    if (inet_pton(AF_INET, address, &receiver_address.sin_addr) <= 0)   // If the address is invalid, return NULL.
    {
        perror("Error on address conversion at open_sender");
        close(this->sock);
        free(this);
        return NULL;
    }
    receiver_address.sin_family = AF_INET;      // Set the address family to IPv4.
    receiver_address.sin_port = htons(port);    // Set the port number to the given port.

    this->peer_address = *(struct sockaddr *)&receiver_address;   // Save the address of the receiver in the sender socket.

    this->peer_address_size = sizeof(receiver_address);
```

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.

```
struct timeval timeout;
timeout.tv_sec = ACK_TIMEOUT_S;
timeout.tv_usec = ACK_TIMEOUT_US;
// Set the timeout for the sender socket.
if (setsockopt(this->sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0) // If the timeout setting fails, return NULL.
{
    perror("Error setting timeout at open_sender");
    close(this->sock);
    free(this);
    return NULL;
}

rudp_header syn_message = {0}; // We will send only the header, therefore we initialize the message to 0.
syn_message.len = 0;
syn_message.flags = SYN;    // Set the SYN flag to 1 for the SYN message.
set_checksum(&syn_message); // Set the checksum of the message.
```

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.

```
    unsigned int remaining_tries;  //Initialize a variable to store the maximum number of retries.
    int successful = FALSE;
    for (remaining_tries = MAX_RETRIES; remaining_tries > 0 && !successful; --remaining_tries)  // Loop until the maximum number of
    {
#ifdef DEBUG
        printf("Attempting SYN, remaining: %d\n", remaining_tries);
#endif
        // Send the SYN message to the receiver.
        if (sendto(this->sock, &syn_message, sizeof(syn_message), 0, &this->peer_address, this->peer_address_size) <= 0)
        {
            perror("Error sending SYN message at open_sender");
            close(this->sock);
            free(this);
            return NULL;
        }

        rudp_header ack_message = {0};  // We will receive only the header, therefore we initialize the message to 0.
        if (recv(this->sock, &ack_message, sizeof(ack_message), 0) < 0) // Receive the ACK message from the receiver and store it i
        {
            perror("Error receiving ACK at open_sender");
            continue;
        }
        if (!validate_checksum(&ack_message))
        {
            fprintf(stderr, "Error on ACK message at open_sender\n");
            continue;
        }
        if (!(ack_message.flags & (ACK | SYN)))
        {
            fprintf(stderr, "Error in ACK message flags at open_sender\n");
            continue;
        }
        // if we made it here we were successful
        successful = TRUE;
    }
```

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.

```
if (remaining_tries == 0)   // If the maximum number of retries is reached, a connection was not established.
{
    fprintf(stderr, "Ran out of retries to send SYN at open_sender\n");
    close(this->sock);
    free(this);
    return NULL;
}

return this;
```

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.

```
rudp_receiver *rudp_open_receiver(unsigned short port)
{
    rudp_receiver *this = malloc(sizeof(rudp_receiver));

    this->sock = socket(AF_INET, SOCK_DGRAM, 0);    // Create a socket for the receiver
    if (this->sock == -1)
    {
        perror("Error on socket creation at open_receiver");
        free(this);
        return NULL;
    }

    int reuse = TRUE;   // Set the reuse option to true, so we could reuse the address.
    if (setsockopt(this->sock, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) < 0)
    {
        perror("Error while setting reuse address option at open_receiver");
        close(this->sock);
        free(this);
        return NULL;
    }

    struct sockaddr_in my_address = {0};        // Initialize the receiver address to 0.
    my_address.sin_addr.s_addr = INADDR_ANY;
    my_address.sin_family = AF_INET;            // Set the address family to IPv4.
    my_address.sin_port = htons(port);          // Set the port number to the given port.

    // Bind the socket to the receiver address.
    if (bind(this->sock, (struct sockaddr *)&my_address, sizeof(my_address)) < 0)
    {
        perror("Error while binding socket at open_receiver");
        close(this->sock);
        free(this);
        return NULL;
    }
```

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).

```
this->peer_address_size = sizeof(this->peer_address);

rudp_header syn_message = {0};  // We will receive only the header, therefore we initialize the message to 0.
// Receive the SYN message from the sender.
if (recvfrom(this->sock, &syn_message, sizeof(syn_message), 0, &this->peer_address, &this->peer_address_size) < 0)
{
    perror("Error trying to receive SYN message from sender at open_receiver");
    close(this->sock);
    free(this);
    return NULL;
}
if (!validate_checksum(&syn_message))   // Validate the checksum of the received message.
{
    fprintf(stderr, "Error in SYN message checksum at open_receiver\n");
    close(this->sock);
    free(this);
    return NULL;
}
if (!(syn_message.flags & SYN)) // Check if the message is a SYN message.
{
    fprintf(stderr, "Error in SYN message flags at open_receiver\n");
    close(this->sock);
    free(this);
    return NULL;
}
```

נכין הודעת SYN חדשה, הודעה זו ריקה, שכן בהודעות FIN/ACK/SYN 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).

```
    rudp_header ack_message = {0}; // we will read only the header, therefore we initialize the message to 0.
    ack_message.len = 0;
    ack_message.flags = ACK | SYN;  // Send a messeage with ACK and SYN flags.
    set_checksum(&ack_message);     // Set the checksum of the message.

    // Send the SYN ACK message to the sender.
    if (sendto(this->sock, &ack_message, sizeof(ack_message), 0, &this->peer_address, this->peer_address_size) <= 0)
    {
        perror("Error sending SYN-ACK message at open_receiver");
        close(this->sock);
        free(this);
        return NULL;
    }

    return this;    // Return the receiver address.
}
```

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.

```c
void rudp_close_sender(rudp_sender *this)
{
    rudp_header close_message = {0};    // We will send only the header, therefore we initialize the message to 0.
    close_message.len = 0;
    close_message.flags = FIN;          // Set the FIN flag to 1 for the FIN message.
    set_checksum(&close_message);       // Set the checksum of the message.
    unsigned int remaining_tries;
    int successful = FALSE;
    for (remaining_tries = MAX_RETRIES; remaining_tries > 0 && !successful; --remaining_tries)
    {
#ifdef DEBUG
        printf("Attempting FIN, remaining: %d\n", remaining_tries);
#endif
        // Send the FIN message to the receiver.
        if (sendto(this->sock, &close_message, sizeof(close_message), 0, &this->peer_address, this->peer_address_size) < 0)
        {
            perror("Error sending close message at close_sender");
            continue;
        }
```

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.

```c
        rudp_header ack_message = {0};
        // Wait to receive the ACK message from the receiver.
        if (recv(this->sock, &ack_message, sizeof(ack_message), 0) < 0)
        {
            perror("Error receiving FIN-ACK at close_sender");
            continue;
        }

        if (!validate_checksum(&ack_message))
        {
            fprintf(stderr, "Error in FIN-ACK checksum at close_sender\n");
            continue;
        }
        // Validate that the message is a FIN ACK message.
        if (!(ack_message.flags & (FIN | ACK)))
        {
            fprintf(stderr, "Error in FIN-ACK flags at close_sender\n");
            continue;
        }

        successful = TRUE; // If we made it here we were successful
    }

    if (remaining_tries == 0)   // If the maximum number of retries is reached, the connection was not closed and an error will be printe
    {
        fprintf(stderr, "Ran out of retries to send FIN\n");
    }

    close(this->sock);
    free(this);
}
```

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.

```
void rudp_close_receiver(rudp_receiver *this)
{
    close(this->sock);
    free(this);
}
```

```
/*
    * Sends one segment of data to the receiver.
    * Can send up to MAX_SEGMENT_SIZE bytes at a time.
    * Is used in rudp_send to divide a message into smaller segments.
    * Returns the number of bytes sent.
*/
int rudp_send_segment(rudp_sender *this, void *data, size_t size, unsigned short segment_num, int more, char *message_buffer)
{
    memcpy(message_buffer + sizeof(rudp_header), data, size);  // Copy the data to the message.
    rudp_header *header = (rudp_header *)message_buffer;       // Initialize the header of the message.
    memset(header, 0, sizeof(rudp_header));                // Initialize the header to 0.
    header->len = size;                                    // Set the length of the message to the size of the data.
    header->flags = (more ? MOR : 0);                      // Set the MOR flag to 1 if there is more data to send.
    header->segment_num = segment_num;                     // Set the header's segment number to the given segment number.
    set_checksum(message_buffer);                              // Set the checksum of the message.
```

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.

```
    size_t message_size = size + sizeof(rudp_header);

    unsigned int remaining_tries;
    int successful = FALSE;
    for (remaining_tries = MAX_RETRIES; remaining_tries > 0 && !successful; --remaining_tries)
    {
#ifdef DEBUG
        printf("Attempting send, remaining: %d\n", remaining_tries);
#endif
        // Send the message to the receiver.
        int sent = sendto(this->sock, message_buffer, message_size, 0, &this->peer_address, this->peer_address_size);
        if (sent < 0)
        {
#ifdef DEBUG
            fprintf(stderr, "message_size = %zu\n", message_size);
#endif
            perror("Error sending message at send_segment");
            continue;
        }
        if ((size_t) sent != message_size)
        {
            fprintf(stderr, "Didn't send whole message at send_segment\n");
            continue;
        }
```

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

```
        rudp_header ack_message;    // Initialize the ack_message to store the received ACK message.
        if (recv(this->sock, &ack_message, sizeof(ack_message), 0) < 0)
        {
            perror("Error receiving ACK at send_segment");
            continue;
        }
        if (!validate_checksum(&ack_message))
        {
            fprintf(stderr, "Error in ACK checksum at send_segment\n");
            continue;
        }

        if (!(ack_message.flags & ACK)) // Validate that the message is an ACK message.

        {
            fprintf(stderr, "Error in ACK message flags at send_segment\n");
            continue;
        }

        if (ack_message.segment_num != segment_num) // Validate that the segment number of the ACK message is the same as the segment nu
        {
            fprintf(stderr, "Error: received ACK for different segment");
            continue;
        }

        // if we made it here we were successful
        successful = TRUE;
    }
    if (remaining_tries == 0)   // If the maximum number of retries is reached, the message was not sent and an error will be printed.
        return -1;
    return size;    // Return the size of the message in bytes.
}
```

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.

```c
int rudp_send(rudp_sender *this, void *data, size_t size)
{
    char *message_buffer = malloc(sizeof(rudp_header) + size); // Preallocate memory for the messages that will be sent (+space for he
    char *data_bytes = (char *)data;    // Convert the pointer of the data to char*.
    size_t total_sent = 0;
    unsigned short segment_num = 0;
    while (total_sent < size)   // Loop until the entire message is sent.
    {
        size_t segment_size = size - total_sent;    // Set the segment size to the remaining size of the message.
        int more = FALSE;
        // If the segment size is greater than the maximum segment size, set the segment size to the maximum segment size and set the
        // This is done to divide the message into smaller segments.
        if (segment_size > MAX_SEGMENT_SIZE)
        {
            segment_size = MAX_SEGMENT_SIZE;
            more = TRUE;
        }
        // Send the segment to the receiver.
        int bytes_sent = rudp_send_segment(this, data_bytes + total_sent, segment_size, segment_num, more, message_buffer);
        // If the segment was not sent, return -1.
        if (bytes_sent < 0)
        {
            free(message_buffer);
            return -1;  // Error, send_segment will print the details of the error.
        }

        total_sent += bytes_sent;   // Add the number of bytes sent to the total number of bytes sent.
        segment_num += 1;           // Increment the segment number.
    }

    free(message_buffer);
    return total_sent;              // Return the total number of bytes sent.
}
```

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.

```c
int rudp_recv(rudp_receiver *this, void *buffer, size_t size)
{
    // Set no timeout for first message
    struct timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = 0;
    if (setsockopt(this->sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0)
    {
        perror("Error setting timeout at open_receiver");
        close(this->sock);
        free(this);
        return -2;
    }

    char *buffer_bytes = (char *)buffer;    // Convert the pointer of the buffer to char*.
    char *segment_buffer = malloc(sizeof(rudp_header) + MAX_SEGMENT_SIZE);  // Allocate memory for the buffer which will contain the mes
    int more = TRUE;
    size_t total_received = 0;
    unsigned short expected_segment_num = 0;
```

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.

```
    while (more)
    {
        // Receive the segment from the sender.
        int received = recv(this->sock, segment_buffer, sizeof(rudp_header) + MAX_SEGMENT_SIZE, 0);

        if (expected_segment_num == 0) // Only after first message
        {
            // Set timeout for subsequent messages
            timeout.tv_sec = RECV_TIMEOUT_S;
            timeout.tv_usec = RECV_TIMEOUT_US;
            if (setsockopt(this->sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0)
            {
                perror("Error setting timeout at open_receiver");
                free(segment_buffer);
                return -2;
            }
        }

#ifdef DEBUG
        printf("received = %d\n", received);
#endif

        if (received < 0)   // An error occurred while receiving the message.
        {
            perror("Error receiving at recv");
            free(segment_buffer);
            return -2;
        }

        if ((unsigned int)received < sizeof(rudp_header))   // If the received message is less than the size of the header, An error occ
        {
            fprintf(stderr, "Error at recv: Received too few bytes\n");
            free(segment_buffer);
            return -2;
        }
```

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.

```
        if (!validate_checksum(segment_buffer)) // Validate the checksum of the received message and return an error if the checksum is
        {
            fprintf(stderr, "Error in message checksum at recv\n");
            free(segment_buffer);
            return -2;
        }

        // Initialize the header of the message. It will point to the header of the segment buffer because the header is the first part
        rudp_header *header = (rudp_header *)segment_buffer;
        more = header->flags & MOR;     // Extract the MOR flag into the more variable

        rudp_header ack_message = {0};  //Initialize an ACK message to send to the sender (only the header matters so we initialize the
        ack_message.len = 0;
        ack_message.flags = ACK;        // Set the ACK flag to 1 for the ACK message.
        ack_message.segment_num = header->segment_num;  // Set the segment number of the ACK message to the segment number of the receiv
        if (header->flags & FIN)        // If the received message is a FIN message, set the FIN flag of the ACK message to 1.
        {
#ifdef DEBUG
            printf("Received FIN message!\n");
#endif
            ack_message.flags |= FIN;   // Set the FIN flag of the ACK message to 1 (set the flag to the OR result with FIN).
        }
        if (header->flags & SYN)        // If the received message is a SYN message, set the SYN flag of the ACK message to 1.
        {
#ifdef DEBUG
            printf("Received extra SYN message!\n");
#endif
            ack_message.flags |= SYN;
        }
```

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.

```
        set_checksum(&ack_message);
        unsigned int remaining_tries;
        int successful = FALSE;
        for (remaining_tries = MAX_RETRIES; remaining_tries > 0 && !successful; --remaining_tries)  // Loop until the maximum number of
        {
            // Send the ACK(-FIN/SYN) message to the sender.
            if (sendto(this->sock, &ack_message, sizeof(ack_message), 0, &this->peer_address, this->peer_address_size) < 0)
            {
                perror("Error sending ACK message at recv");
                continue;
            }
            successful = TRUE; // If we made it here we were successful
#ifdef DEBUG
            printf("sent ack for segment number %hu\n", expected_segment_num);
#endif
        }

        if (header->flags & FIN)    // After sending the FIN-ACK message, return -1 (which represents end of connection), receiver shoul
        {
            free(segment_buffer);
            return -1;
        }

        if (total_received > size)  // If the total number of bytes received is greater than the size of the buffer, print an error and
        {
            free(segment_buffer);
            fprintf(stderr, "Fatal error: total_received > size: %zu > %zu", total_received, size);
            exit(1);
        }
```

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.

```
        if (expected_segment_num == header->segment_num) // Otherwise duplicate message
        {
#ifdef DEBUG
            printf("Got valid segment!\n");
#endif
            // Copy the received message to the buffer in the (+total_received) position.
            memcpy(buffer_bytes + total_received, segment_buffer + sizeof(rudp_header), received - sizeof(rudp_header));
            total_received += received - sizeof(rudp_header);   // Increment the total number of bytes received by the size of the recei
            expected_segment_num += 1;      // Increment the expected segment number by 1.
        }
    }

    free(segment_buffer);   // After finishing the loop, free the segment buffer.

    return total_received;  // Return the total number of bytes received.
}
```

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.

לאחר 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.

```c
rudp_sender *sender = rudp_open_sender(ip, port);
if (!sender)
{
    printf("Failed to connect to open sender\n");
    return 1;
}
printf("Successfully connected to the receiver!\n");

char *message = util_generate_random_data(FILE_SIZE);

// Reporting the size of the message.
uint32_t report_file_size = htonl(FILE_SIZE);
int starting_message = rudp_send(sender, (char *)&report_file_size, sizeof(report_file_size));
if (starting_message <= 0)
{
    perror("rudp_send failed\n");
    rudp_close_sender(sender);
    return 1;
}
```

ב-RUDP, פונקציית ה-rudp_send מבצעת את הסגמנטציה בתוכה, לכן בהפעלה אחת של הפונקציה כל הקובץ נשלח.

נשלח את הקובץ כמות הפעמים שהמשתמש יבקש.

```c
do
{
    // Try to send the message to the receiver using the socket.
    bytes_sent = rudp_send(sender, message, FILE_SIZE);
    // If no data was sent, print an error message and return 1. Only occurs if the connection was closed.
    if (bytes_sent <= 0)
    {
        perror("rudp_send failed\n");
        rudp_close_sender(sender);
        return 1;
    }

    fprintf(stdout, "Sent %d bytes to the receiver!\n", bytes_sent);
    printf("Do you want to send again? (Y/n)\n");
    do
    {
        choice = getchar(); //Getting a char from the user untill he writes 'y', 'Y', 'n', 'N'
    } while (choice != 'n' && choice != 'N' && choice != 'y' && choice != 'Y');

} while (choice != 'n' && choice != 'N');
```

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

```c
free(message);

rudp_close_sender(sender);

printf("Connection closed!\n");

// Return 0 to indicate that the client ran successfully.
return 0;
```

הסבר קוד RUDP_Receiver.c

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

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

נמשיך ברגע שתסתיים לחיצת הידיים.

```
rudp_receiver *receiver = rudp_open_receiver(port);

if (!receiver)
{
    fprintf(stderr, "Failed to open receiver");
    return 1;
}
```

נמתין לקבלת ההודעה הראשונה שכוללת הצהרה על גודל הקובץ שישלח.

לאחר שהתקבל הגודל, נמיר אותו ל-size_t ע"י הקצאתו למשתנה buffer_size.

```
int input_size = 0;
size_t sizeof_input;
while (!input_size)
{
    // Before sending the packege, the sender will send the weight of the package in bytes.
    input_size = rudp_recv(receiver, &sizeof_input, sizeof(sizeof_input));
}
sizeof_input = ntohl(sizeof_input);
if(!format) printf("Size of input: %ld bytes\n", sizeof_input);
size_t buffer_size = sizeof_input;
```

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.

```
char *buffer = calloc(buffer_size, sizeof(char));
unsigned short noEndMessage = TRUE; // Indicator if the end message was received.
if(format) printf("Time (ms),Speed (MB/s)\n");
// The receiver's main loop.
while (noEndMessage)
{
    int bytes_received = 0;
    // Create a buffer to store the received message.
    clock_t start, end;
    double time_used_inMS;
    start = clock();
    bytes_received = rudp_recv(receiver, buffer, buffer_size);  // RUDP passes the entire message to the buffer all at once.
    end = clock();
    // If the message receiving failed, print an error message and return 1.
    if (bytes_received == -2)
    {
        fprintf(stderr, "Failed to receive from sender\n");
        rudp_close_receiver(receiver);
        endFree(&Times_list, &Speed_list, buffer);
        return 1;
    }
    // If the received message is "Closing connection", close the sender's socket and return 0.
    if (bytes_received == -1)
    {
        noEndMessage = FALSE;
        continue;
    }
    if (bytes_received == 0)
        continue;
```

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.

```
        if((size_t)bytes_received != sizeof_input)
        {
            printf("Error! Received message size is not the same as the input size\n");
            rudp_close_receiver(receiver);
            endFree(&Times_list, &Speed_list, buffer);
            return 1;
        }
        time_used_inMS = 1000 * (double)(end - start) / CLOCKS_PER_SEC; // Calculating the time it took for the message to be received.
        double speed = convertToSpeed(bytes_received, time_used_inMS);
        addToList(&Times_list, time_used_inMS);
        addToList(&Speed_list, speed);
        if (!format)
        {
            printf("Time taken to receive that messege: %f ms\n", time_used_inMS);
            printf("Speed: %f MB/s\n", speed);
        }
        if (format)
        {
            printf("%ld,%f,%f\n", run, time_used_inMS, (double)convertToMegaBytes(bytes_received) / (time_used_inMS / 1000));
        }
        if (!format)
            printf("Received %d bytes from the sender\n", bytes_received);
        run++;
    }
    if (!format)
        printf("Sender finished!\n");
    rudp_close_receiver(receiver);
    endPrints(&Times_list, &Speed_list, run, format);
    endFree(&Times_list, &Speed_list, buffer);
    return 0;
}
```

נמשיך 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:

31	333.462491803	127.0.0.1	127.0.0.1	UDP	50 34727 → 1025 Len=8
32	333.462558184	127.0.0.1	127.0.0.1	UDP	50 1025 → 34727 Len=8
33	337.796979127	127.0.0.1	127.0.0.1	UDP	60 34727 → 1025 Len=18
34	337.797006426	127.0.0.1	127.0.0.1	UDP	50 1025 → 34727 Len=8
35	337.797014984	127.0.0.1	127.0.0.1	UDP	60 34727 → 1025 Len=18
36	337.901990793	127.0.0.1	127.0.0.1	UDP	60 34727 → 1025 Len=18
37	338.004846559	127.0.0.1	127.0.0.1	UDP	60 34727 → 1025 Len=18
38	338.111707546	127.0.0.1	127.0.0.1	UDP	60 34727 → 1025 Len=18
39	338.111765947	127.0.0.1	127.0.0.1	UDP	50 1025 → 34727 Len=8
40	341.044938616	127.0.0.1	127.0.0.1	UDP	50 34727 → 1025 Len=8
41	341.044965493	127.0.0.1	127.0.0.1	UDP	50 1025 → 34727 Len=8

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

ראשית שתי ההודעות הראשונות הן לחיצת הידיים:

```
▸ User Datagram Protocol, Src Port: 1025, Dst Port: 34727
▾ Data (8 bytes)
    Data: 00000300fcff0000
    [Length: 8]
```

```
▸ User Datagram Protocol, Src Port: 34727, Dst Port: 1025
▾ Data (8 bytes)
    Data: 00000100feff0000
    [Length: 8]
```

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:

```
▸ User Datagram Protocol, Src Port: 34727, Dst Port: 1025
▾ Data (18 bytes)
    Data: 0a000800e335000048656c6c6f207468726f
    [Length: 18]
```

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:

```
........ ..-.....5
··Hello  thro
```

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

לאחר מכן קיימת הודעת ACK מה-receiver:

```
▸ User Datagram Protocol, Src Port: 1025, Dst Port: 34727
▾ Data (8 bytes)
    Data: 00000200fdff0000
    [Length: 8]
```

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:

```
▸ User Datagram Protocol, Src Port: 34727, Dst Port: 1025
▾ Data (18 bytes)
    Data: 0a0000005fd2010075676820525544502100
    [Length: 18]
```

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:

```
00 01 87 a7 04 01 00 1a  fe 2d 0a 00 00 00 5f d2   ........ .-....._
01 00 75 67 68 20 52 55  44 50 21 00               ··ugh RU DP!·
```

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:

```
▸ User Datagram Protocol, Src Port: 1025, Dst Port: 34727
▾ Data (8 bytes)
    Data: 00000200fcff0100
    [Length: 8]
```

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:

```
▸ User Datagram Protocol, Src Port: 1025, Dst Port: 34727
▾ Data (8 bytes)
    Data: 00000600f9ff0000
    [Length: 8]
```

```
▸ User Datagram Protocol, Src Port: 34727, Dst Port: 1025
▾ Data (8 bytes)
    Data: 00000400fbff0000
    [Length: 8]
```

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:

```
chaim@chaim-averbach:~/ComputerNetworking_Ex3$ ./rudp_receiver_test
Created receiver!
Attempting to receive message...
Received: "Hello through RUDP!"
Sender closed the connection
chaim@chaim-averbach:~/ComputerNetworking_Ex3$
```

```
chaim@chaim-averbach:~/ComputerNetworking_Ex3$ ./rudp_sender_test
Error receiving ACK at open_sender: Resource temporarily unavailable
Created sender!
Starting sending loop, send "close" to finish.
Enter message: Hello through RUDP!
Attempting to send message: "Hello through RUDP!"...
Error receiving ACK at send_segment: Resource temporarily unavailable
Error receiving ACK at send_segment: Resource temporarily unavailable
Error receiving ACK at send_segment: Resource temporarily unavailable
Sent message.
Enter message: close
Closed correctly
chaim@chaim-averbach:~/ComputerNetworking_Ex3$
```

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:

```
uriel@LinuxVM:~/VS Code Programs/Computer Networking - Ex3$ ./RUDP_Receiver -p 4000
Port: 4000
Format: OFF
Size of input: 2097152 bytes
run: 0
Time taken to receive that messege: 3.674000 ms
Speed: 544.365814 MB/s
Received 2097152 bytes from the sender
run: 1
Time taken to receive that messege: 0.797000 ms
Speed: 2509.410289 MB/s
Received 2097152 bytes from the sender
run: 2
Time taken to receive that messege: 0.700000 ms
Speed: 2857.142857 MB/s
Received 2097152 bytes from the sender
run: 3
Time taken to receive that messege: 0.886000 ms
Speed: 2257.336343 MB/s
Received 2097152 bytes from the sender
run: 4
Time taken to receive that messege: 0.905000 ms
Speed: 2209.944751 MB/s
Received 2097152 bytes from the sender
Sender finished!
---------------------------------------
Runs summary:
Run 1# data: Time: 3.67 ms, Speed: 544.37 MB/s
Run 2# data: Time: 0.80 ms, Speed: 2509.41 MB/s
Run 3# data: Time: 0.70 ms, Speed: 2857.14 MB/s
Run 4# data: Time: 0.89 ms, Speed: 2257.34 MB/s
Run 5# data: Time: 0.91 ms, Speed: 2209.94 MB/s
Average time: 1.392400
Average bandwidth: 2075.640011
Number of runs: 5
```

```
uriel@LinuxVM:~/VS Code Programs/Computer Networking - Ex3$ ./RUDP_Sender -p 4000
IP: 127.0.0.1
Port: 4000
Connecting to 127.0.0.1:4000...
Successfully connected to the receiver!
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
n
Connection closed!
```

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.

```
uriel@LinuxVM:~/VS Code Programs/Computer Networking - Ex3$ ./TCP_Sender -p 4000 -algo reno
IP: 127.0.0.1
Port: 4000
Algorithm: reno
Connecting to 127.0.0.1:4000...
Successfully connected to the receiver!
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
y
Sent 2097152 bytes to the receiver!
Do you want to send again? (Y/n)
n
Sent ending messege to the receiver
Connection closed!
```

```
uriel@LinuxVM:~/VS Code Programs/Computer Networking - Ex3$ ./TCP_Receiver -p 4000 -algo reno
Port: 4000
Algorithm: reno
Format: OFF
Listening for incoming connections on port 4000...
Client 127.0.0.1:58366 connected
Size of input: 2097152 bytes
Time taken to receive that messege: 0.451000 ms
Speed: 4434.589800 MB/s
Received 2097152 bytes from the sender 127.0.0.1:58366
run: 1
Time taken to receive that messege: 0.744000 ms
Speed: 2688.172043 MB/s
Received 2097152 bytes from the sender 127.0.0.1:58366
run: 2
Time taken to receive that messege: 0.561000 ms
Speed: 3565.062389 MB/s
Received 2097152 bytes from the sender 127.0.0.1:58366
run: 3
Time taken to receive that messege: 0.709000 ms
Speed: 2820.874471 MB/s
Received 2097152 bytes from the sender 127.0.0.1:58366
run: 4
Time taken to receive that messege: 0.747000 ms
Speed: 2677.376171 MB/s
Received 2097152 bytes from the sender 127.0.0.1:58366
run: 5
Time taken to receive that messege: 0.047000 ms
Speed: 0.000000 MB/s
Received 57 bytes from the sender 127.0.0.1:58366
Client 127.0.0.1:58366 disconnected
Closing connection!
-----------------------------------
Runs summary:
Run 1# data: Time: 0.45 ms, Speed: 4434.59 MB/s
Run 2# data: Time: 0.74 ms, Speed: 2688.17 MB/s
Run 3# data: Time: 0.56 ms, Speed: 3565.06 MB/s
Run 4# data: Time: 0.71 ms, Speed: 2820.87 MB/s
Run 5# data: Time: 0.75 ms, Speed: 2677.38 MB/s
Average time: 0.642400
Average bandwidth: 3237.214975
Number of runs: 5
```

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.

1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	34246 → 4000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1231835625 TSecr=0 WS=128
2	0.000010402	127.0.0.1	127.0.0.1	TCP	74	4000 → 34246 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=1231835625 TSecr=1231835625 WS=128
3	0.000019447	127.0.0.1	127.0.0.1	TCP	66	34246 → 4000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1231835625 TSecr=1231835625
4	0.000038135	127.0.0.1	127.0.0.1	TCP	70	34246 → 4000 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=1231835625 TSecr=1231835625

נשים לב שה-Data שנשלח הוא 2MB ביצוג שלהם בבייטים ב-Hexadecimal.

```
▸ Frame 4: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 34246, Dst Port: 4000, Seq: 1, Ack: 1, Len: 4
▾ Data (4 bytes)
    Data: 00200000
    [Length: 4]
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 38 bd ac 40 00 40 06  7f 11 7f 00 00 01 7f 00   ·8··@·@· ........
0020  00 01 85 c6 0f a0 b0 0d  72 e9 2a e2 5c 8b 80 18   ........ r·*·\···
0030  02 00 fe 2c 00 00 01 01  08 0a 49 6c 51 e9 49 6c   ···,···· ··IlQ·Il
0040  51 e9 00 20 00 00                                  Q··  ··
```

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').

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 47 be a2 40 00 40 06  7e 0c 7f 00 00 01 7f 00   ·G··@·@· ~·......
0020  00 01 85 c6 0f a0 b0 cd  73 13 2a e2 5c 8b 80 18   ........ s·*·\···
0030  02 00 fe 3b 00 00 01 01  08 0a 49 6c 66 28 49 6c   ···;···· ··Ilf(Il
0040  66 28 43 6c 6f 73 69 6e  67 20 63 6f 6e 6e 65 63   f(Closin g connec
0050  74 69 6f 6e 00                                     tion·
```

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

```
1Y..v..........._^....X....X.nbH..!=.A..'.....7..........:.9....:>.{.-...{...v.j..1.....Nw....j...D....
b5....~.2.{..'f.. .).g..Q7.h.Zr..*.:.....Z..})..I.7#.2.Li....'..R......q!.......$./ 4$l..!..I......geo..   .%.a.... ..-...3..|...rt.....a'....a.j0..%.!..UF:....q..J_.....F.e....t..x...U..&...]c...\..
8%...p.YG.8D.8.$.......#3j...pw..?. ..z...;V.'{w.m".=..pClosing connection.Closing connection.Closing connection.
```

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:

```
324 5.182455198 127.0.0.1   127.0.0.1   TCP   66 34246 → 4000 [FIN, ACK] Seq=12582974 Ack=1 Win=65536 Len=0 TSval=1231840808 TSecr=1231840808
325 5.182628928 127.0.0.1   127.0.0.1   TCP   66 4000 → 34246 [FIN, ACK] Seq=1 Ack=12582975 Win=3145728 Len=0 TSval=1231840808 TSecr=1231840808
326 5.182640417 127.0.0.1   127.0.0.1   TCP   66 34246 → 4000 [ACK] Seq=12582975 Ack=2 Win=65536 Len=0 TSval=1231840808 TSecr=1231840808
```

```
▾ Flags: 0x011 (FIN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Nonce: Not set
    .... 0... .... = Congestion Window Reduced (CWR): Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
  ▸ .... .... ...1 = Fin: Set
  ▾ [TCP Flags: ·····A···F]
    ▾ [Expert Info (Note/Sequence): This frame initiates the connection closing]
        [This frame initiates the connection closing]
        [Severity level: Note]
        [Group: Sequence]
```

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:

```
243 2.830452391 127.0.0.1   127.0.0.1   TCP   65549 34246 → 4000 [ACK] Seq=9632790 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838455
244 2.830466825 127.0.0.1   127.0.0.1   TCP   65549 [TCP Previous segment not captured] 34246 → 4000 [ACK] Seq=9763756 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838456
245 2.830479161 127.0.0.1   127.0.0.1   TCP   65549 34246 → 4000 [ACK] Seq=9829239 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838456
246 2.830473932 127.0.0.1   127.0.0.1   TCP      78 4000 → 34246 [ACK] Seq=1 Ack=9698273 Win=3079424 Len=0 TSval=1231838456 TSecr=1231838456 SLE=9763756 SRE=9829239
247 2.830582269 127.0.0.1   127.0.0.1   TCP      78 [TCP Dup ACK 246#1] 4000 → 34246 [ACK] Seq=1 Ack=9698273 Win=3079424 Len=0 TSval=1231838456 TSecr=1231838456 SLE=9763756 SRE=989…
248 2.830594015 127.0.0.1   127.0.0.1   TCP   65549 34246 → 4000 [ACK] Seq=9894722 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838456
249 2.830596334 127.0.0.1   127.0.0.1   TCP      78 [TCP Dup ACK 246#2] 4000 → 34246 [ACK] Seq=1 Ack=9698273 Win=3079424 Len=0 TSval=1231838456 TSecr=1231838456 SLE=9763756 SRE=996…
250 2.830632784 127.0.0.1   127.0.0.1   TCP   65549 [TCP Fast Retransmission] 34246 → 4000 [ACK] Seq=9698273 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838456
251 2.830637735 127.0.0.1   127.0.0.1   TCP   65549 34246 → 4000 [PSH, ACK] Seq=9960205 Ack=1 Win=65536 Len=65483 TSval=1231838456 TSecr=1231838456
252 2.830639812 127.0.0.1   127.0.0.1   TCP      66 4000 → 34246 [ACK] Seq=1 Ack=9960205 Win=2848000 Len=0 TSval=1231838456 TSecr=1231838456
```

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:

```
Times in ms:
                            0%        2%        5%       10%
RUDP                  0.590100  3.868050  0.921300  2.061800
TCP R: reno S: reno   1.199062  1.185267  0.624588  1.092688
TCP R: reno S: cubic  0.348526  0.154850  1.671529  0.862563
TCP R: cubic S: reno  0.303833  0.879000  0.984647  1.118389
TCP R: cubic S: cubic 0.397937  0.375933  0.505400  0.868000

Bandwidths in MB/s:
                                 0%            2%           5%          10%
RUDP                    5768.824078    965.950429  2407.332452  1888.227393
TCP R: reno S: reno     4713.702308   4640.007837  4067.302809  2186.731111
TCP R: reno S: cubic   11710.868724  13669.421547  3985.437492  2702.375634
TCP R: cubic S: reno   10247.147979   8167.674029  2910.451694  2374.188380
TCP R: cubic S: cubic   8882.608808   8947.484037  5095.145059  2635.584308
```

ניתן לראות בכל טבלה באיזה פרוטוקול השתמשנו (RUDP/TCP) 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.
2) 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.
3) 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-ack'ים), 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.
2. 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 $S \cdot MSS$, נכתוב זאת בנוסחה:

$$\sum Bytes = MSS \cdot (1 + 2 + 4 + \cdots + S) = MSS \cdot \sum_{i=0}^{log_2 S} 2^i = MSS \cdot \left(2^{log_2 S+1} - 1\right) = MSS \cdot \left(2 \cdot 2^{log_2 S} - 1\right)$$
$$= MSS \cdot (2S - 1) \approx 2S \cdot MSS$$

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 $log_2 S$ ולכן $\sum time \approx RTT \cdot log_2 S$

על מנת לחשב את התפוקה הכוללת נחלק ביניהם:

$$\text{Throughput} = \frac{\sum \text{Bytes}}{\sum \text{time}} \approx \frac{2\text{S} \cdot \text{MSS}}{\text{RTT} \cdot \log_2 S}$$

ולכן התשובה הנכונה היא א'.

3. 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 $t = s/v$
ראשית RTT שווה לפעמיים הזמן שלוקח למידע לעבור בין התחנות, לפי הנתונים לגבי מרחק התחנות וקצב ההתפשטות: $RTT = 2 \cdot \frac{10^3 m}{2 \cdot 10^8 m/s} = 10^{-5} s$
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: $t_T = \frac{3 \cdot 10^3 B}{10^9 B/s} = 3 \cdot 10^{-6} s$
עבור תפוקה מקסימלית נרצה לבחור N כך ש-$N \cdot t_T \geq RTT$ כלומר $N \cdot 3 \cdot 10^{-6} \geq 10^{-5}$ לאחר העברת אגפים:
$N \geq \frac{1}{3} \cdot \frac{10^{-5}}{10^{-6}} = \frac{10}{3}$ מכיוון ש-N שלם נדרוש $N \geq 4$ וזה צריך להיות גודל החלון (לכל הפחות) על מנת להבטיח תפוקה מקסימלית.