U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Data Connection Protocol

RCOM - First Project

Eduardo Correia
up201806433@fe.up.pt

Ana Inês Barros
up201806593@fe.up.pt

November 11, 2020

Contents

Chapter 1

Summary

This project was created for the Computer Networks (RCOM) course unit and it envisioned developing a data connection protocol. This work allowed us to apply many of the concepts that we learned in classes, such as, how protocols, in paticular TCP/IP, are implemented and the network layers model structure. At the same time it provided a practical component of the otherwise theoretical subjects.

Chapter 2

Introduction

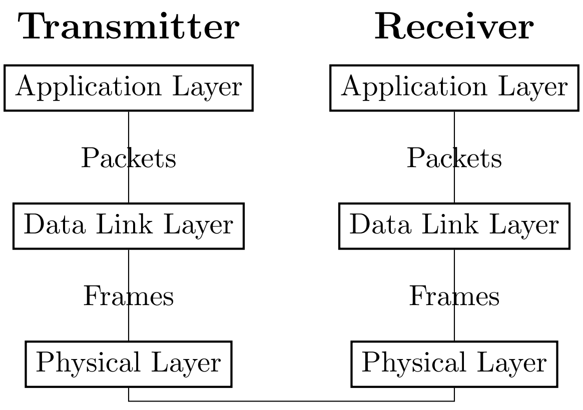
The main goal of this laboratory work was to develop and test a robust data connection protocol. For that purpose, a program that transfer files between two computers, connected by a RS-232 serial port, was developed, following the project's guidelines. This report will detail our implementation of the protocol, as well its specification, following this structure:

- **Architecture:** Functional blocks and interfaces;
- **Code Structure:** APIs, main data structures and functions and their relation with the architecture;
- **Use Cases:** Identification of main use cases of the program and function calls sequences;
- **Validation:** Description of the protocol validation tests and quantified presentation of obtained results;
- **Efficiency:** Description of performance tests, illustrated with graphics;
- **Conclusion:** Summary of the information presented in the previous sections and reflections about the achieved learning objectives.

Chapter 3

Architecture

Following the protocol specification for this project, we implemented an architecture based on layers, such as the one in the TCP/IP model.



This architecture in layers in based on the independence between them.

In the data link layer, there is no processing of the packets that it receives from the application layer, since it is considered to be irrelevant. The application layer only knows how to access the service of the data link protocol but does not understand how it works internally. This makes it possible for each layer to only have access to relevant information to perform the assigned functions.

Chapter 4

Code Structure

4.1 Application

4.1.1 Data Structures

```
typedef struct {
    char port[20];          /* Dispositivo /dev/ttySx, x = 0, 1 */
    int fileDescriptor;     /* Descritor correspondente à porta série */
    enum Status status;     /* TRANSMITTER | RECEIVER */
    char filename[256];     /* Nome do ficheiro */
    int sequence_number;    /* Numero de sequencia do pacote */
    int chunk_size;         /* Tamanho pacote*/
} applicationLayer;
```

4.1.2 llopen

```
int llopen(char* port, enum Status status)
```

– Asks link layer to establish the connection between the receiver and the transmitter.

– Sets the value of the file descriptor of the serial port and the numbering of packets.

– Returns the serial port's file descriptor.

4.1.3 llwrite

```
int llwrite(int fd, unsigned char* buffer, int length)
```

– Asks the data link layer to send the packet passed in the function's arguments to the file descriptor fd.

– Returns the number of bytes sent.

4.1.4 llread

```
int llread(int fd, unsigned char** buffer)
```

– Asks the data link layer to receive a packet from the file discriptor passed in the function's arguments.

– Returns the number of bytes received.

4.1.5 llclose

```
int llclose(int fd)
```

– Asks the data link layer to finish the connection.

– Closes the file descriptor of the file.

– Closes the file descriptor of the serial port.

4.1.6 file_transmission

```
int file_transmission()
```

Transmitter:

– Opens file to send.

– Prepares packets and sends them with llwrite.

Receiver:

– Receives start packet and opens file.

– Receives more data packets using llread and writes data to an array.

– When the end packet is received, writes all the contents of the array to the file.

4.2 Data Link

4.2.1 Data Structures

```
typedef struct {
    unsigned int sequenceNumber;   /* Número de sequência da trama: 0, 1 */
    unsigned int timeout;          /* Valor do temporizador: 1 s */
    unsigned int numTransmissions; /* Número de tentativas em caso de falha*/
} linkLayer;
```

4.2.2 write_info_frame

```
int write_info_frame(int fd, unsigned char* packet, int length)
```

– Prepares an information frame with packet received in the function's arguments.

– Writes frame to the file descriptor fd and sets an alarm.

– Waits to read an acknowlegement message. If the alarm rings, it retries to send the same frame up until a number of times. In case of no response, it gives up tying.

– If it receives an acknowledgement message of type RR, sequence number is updated and the alarm is cancelled.

– If it receives an acknowledgement message of type REJ, it resets the alarm and resends the frame.

– Returns the number of bytes written or -1 in case of timeout.

4.2.3 read_info_frame

```
int read_info_frame(int fd, unsigned char* packet, int length)
```

– Reads byte by byte from the file descriptor fd until it receives an information frame.

– Writes to fd an acknowledgement message. REJ if the received frame has errors, RR in case it is a repeated or new frame without errors.

– Updates the sequence number in case of receiving a new frame.

– Returns the number of bytes received.

4.2.4 establish_connection

```
int establish_connection(char* port, enum Status status)
```

Transmitter:

– Opens the serial port device for reading and writting and sets the new configuration.

– Sends a SET message and sets an alarm.

– Waits until it receives a UA message. While it has not received one, it resends the SET message when the alarm rings up until a number of times.

– Deactivates the alarm and returns upon receiving an UA message.

– Returns 0 if the connection is successfully established and -1 in case of timeout.

Receiver:

– Opens the serial port device for reading and writting and sets the new configuration.

– When it receives a SET message, it sends an UA message.

– Returns 0 when connection is established.

4.2.5 finish_connection

```
int finish_connection(int fd, enum Status status)
```

Transmitter

– Sends a DISC message to file descriptor fd and sets an alarm.

– Waits until it receives a DISC message. While it has not received one, it resends the DISC message when the alarm rings up until a number of times.

– Upon receiving a DISC message, it deactivates the alarm and sends a UA message.

– Returns 0 if successfully finished the connection, -1 in case of timeout.

Receiver

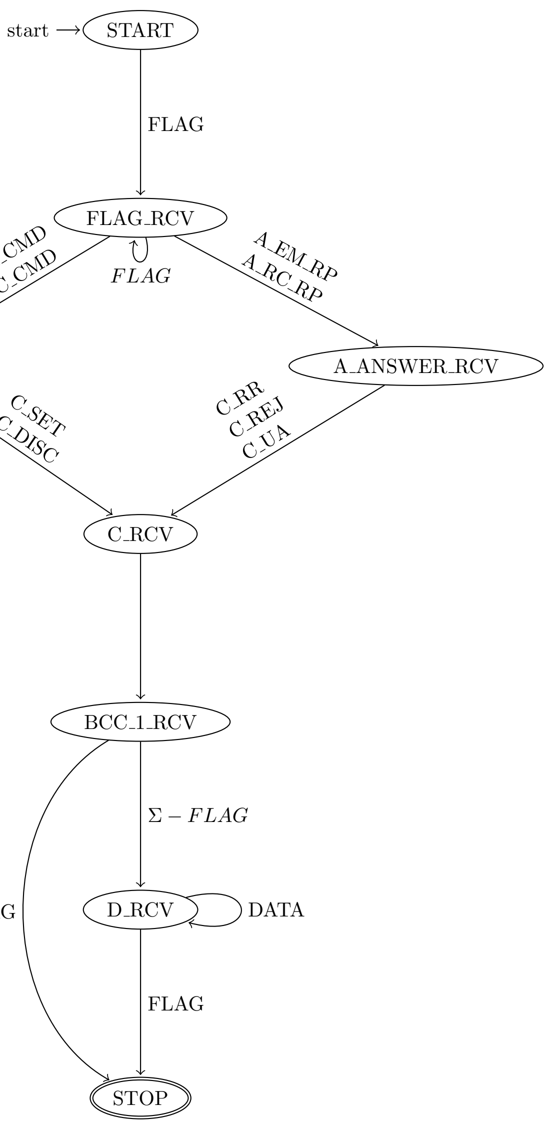
– Waits until it receives a DISC message from file descriptor fd.

– Responds with another DISC message.

– Waits to receive a UA message.

– Returns 0 if successfully finished the connection.

4.3 State Machine



Note: We omitted these transitions from the diagram to keep it simple, but, in each state, the machine returns to `START` if anything is received except for what we're expecting

The struct of the state machine needs to store the current state it is in and the status type (transmitter/receiver).

```
struct state_machine {
    int current_state;
    enum Status status;
};
```

4.3.1 change_state

```
void change_state(struct state_machine* stm, char field)
```

This function updates the state of the state machine accordingly to the field received in the function's arguments.

Chapter 5

Use Cases

Our program works with a variety of configurable values which the user can set when running the program.

- **Serial Port:** Number of the serial port device;

  `/dev/ttySX`

- **Status:** Role in transmission;

  `-c` or `--client`

  `-s` or `--server`

- **Timeout:** Alarm time;

  `-t` *`TIMEOUT`* or `--timeout` *`TIMEOUT`*

- **Number of transmissions:** Maximum number of alarm timeouts;

  `-n` *`NUM_TRANSMISSIONS`* or `--num_transmissions` *`NUM_TRANSMISSIONS`*

- **File to Transfer:** Path of the file to send;

  `-f` *`FILE`* or `--file` *`FILE`*

- **Chunk Size:** Size, in bytes, of the data field in packets;

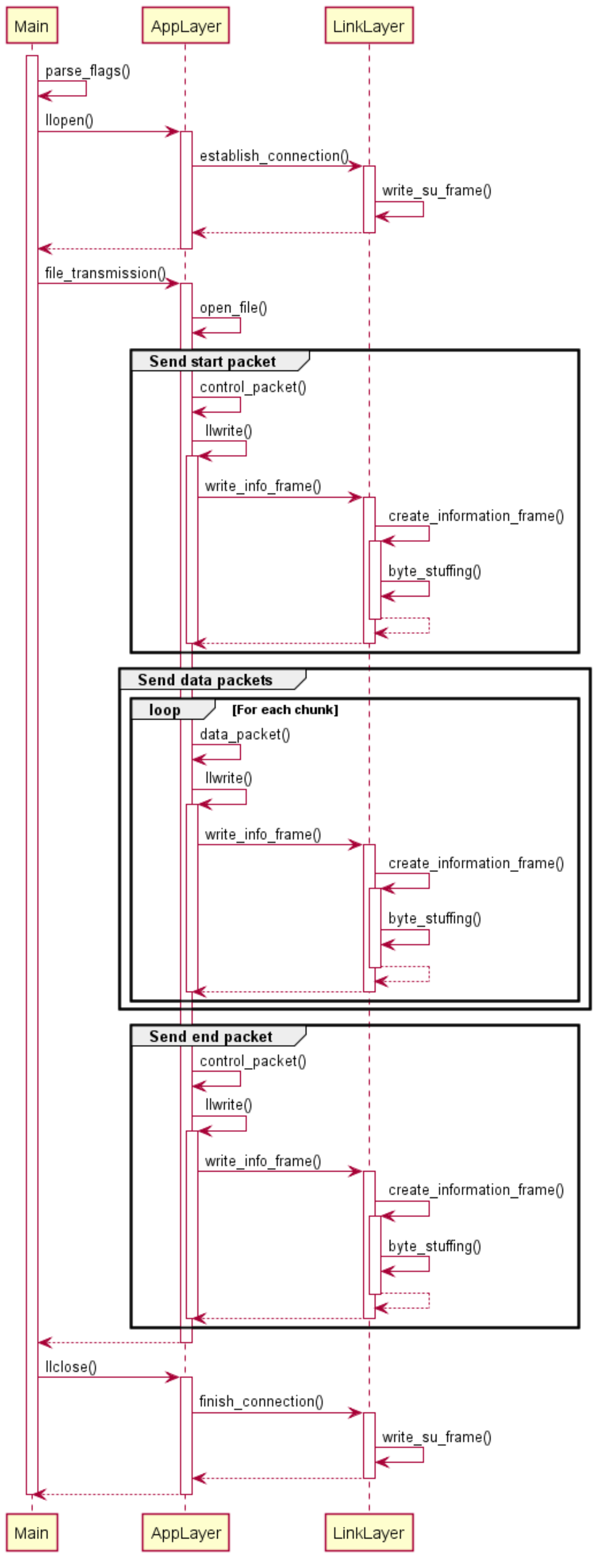
  `--chunk_size` *`CHUNK_SIZE`*

5.1 Transmitter

5.1.1 Usage Example

```
./main -c "/dev/ttyS0" --timeout 5 --num_transmissions 3 --chunk_size 256
↪  --file "../files/pinguim.gif"
```

5.1.2 Function call sequence

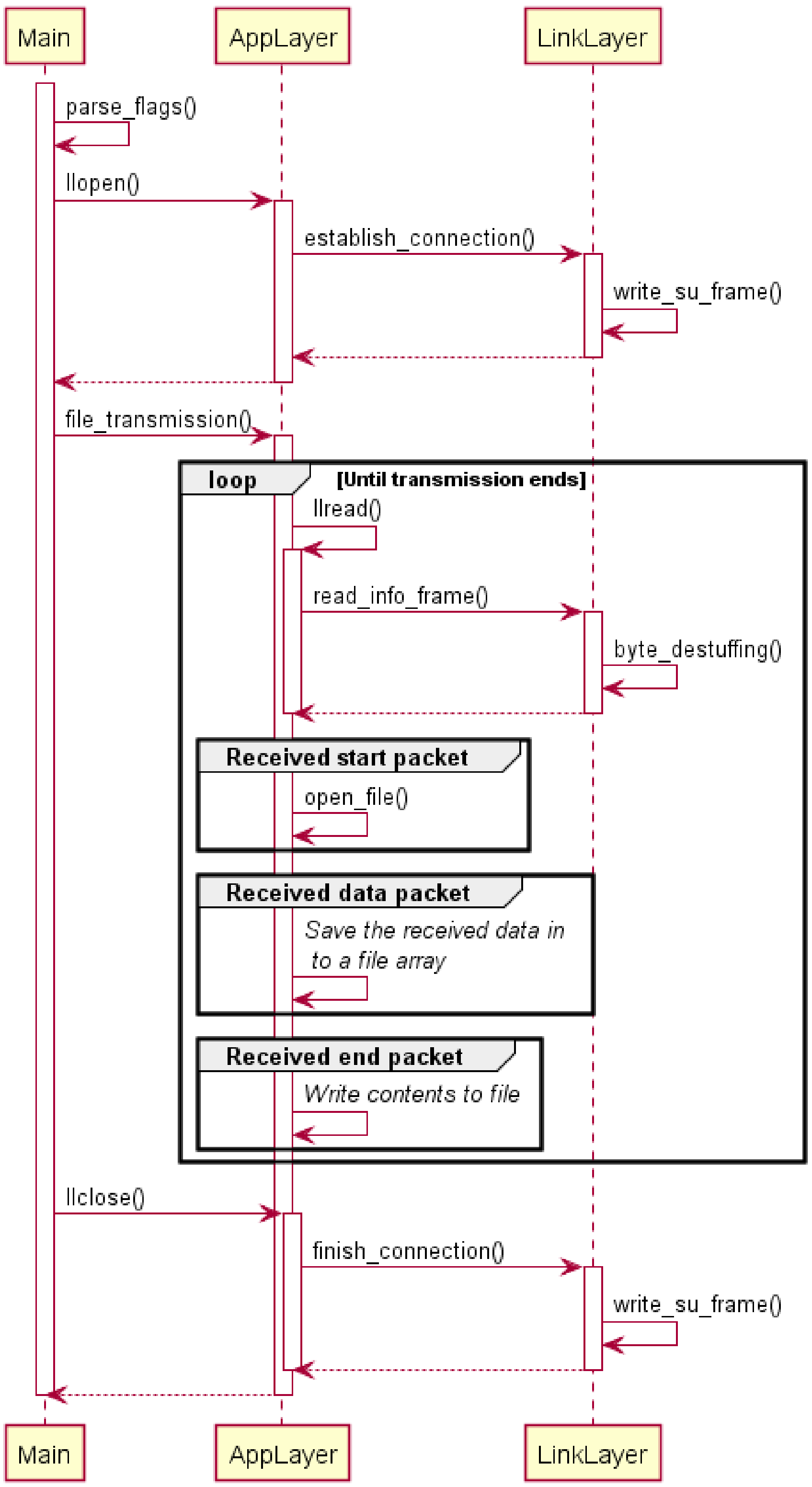


5.2 Receiver

5.2.1 Usage Example

```
./main -c "/dev/ttyS0" --chunk_size 256
```

5.2.2 Function call sequence



Chapter 6

Validation

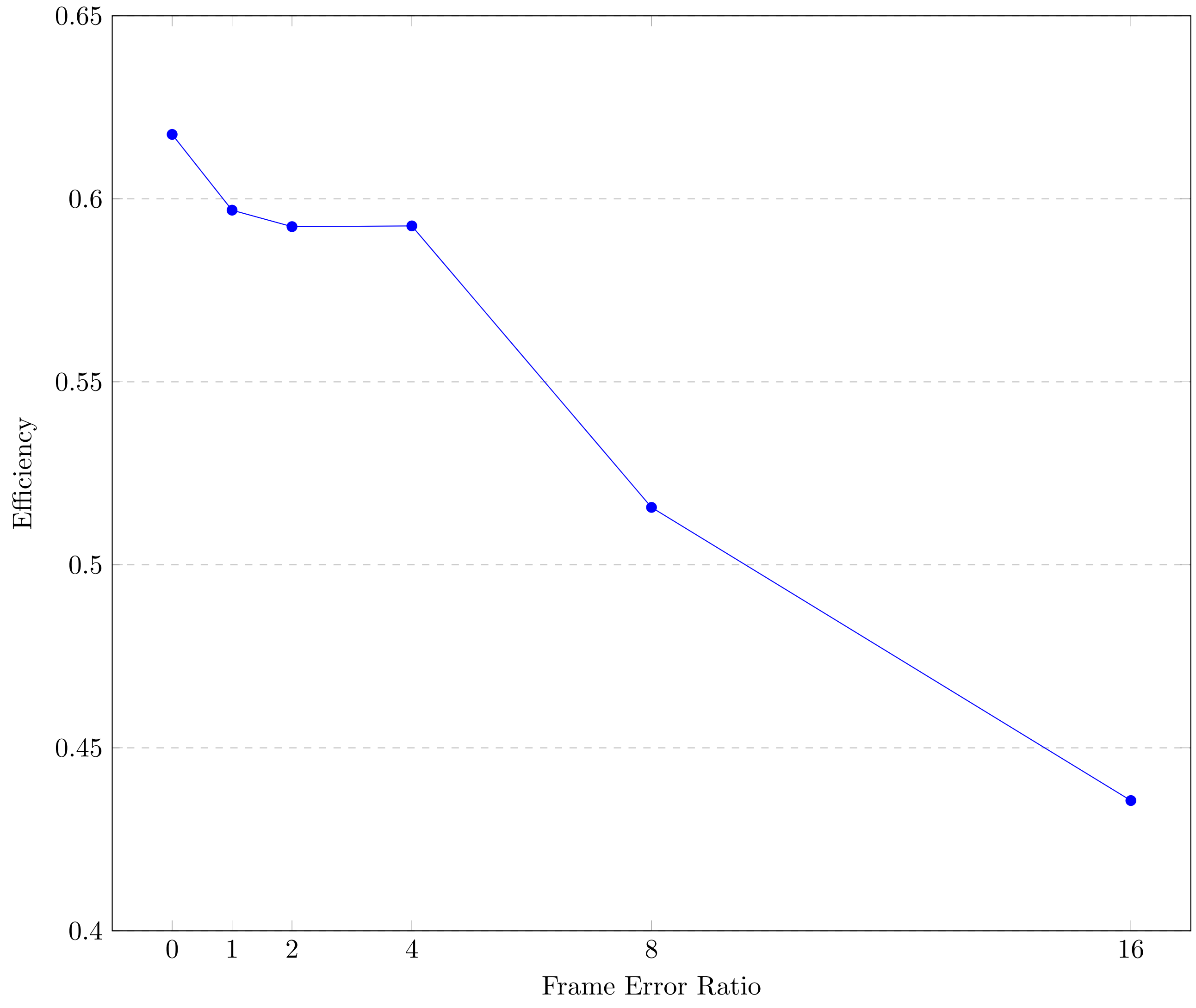
To test that our code was working we tried running the following tests and all were successfully completed.

1. Transmission of the `pinguim.gif` file.
2. Transmission of bigger and smaller files than the `pinguim.gif` file.
3. Disconnecting and reconnecting the serial port during transmission.
4. Causing interference in the serial port with a coin to simulate errors in the frames.
5. Different baudrate values.
6. Different chunk size values.
7. Different T_Prop times.

Chapter 7

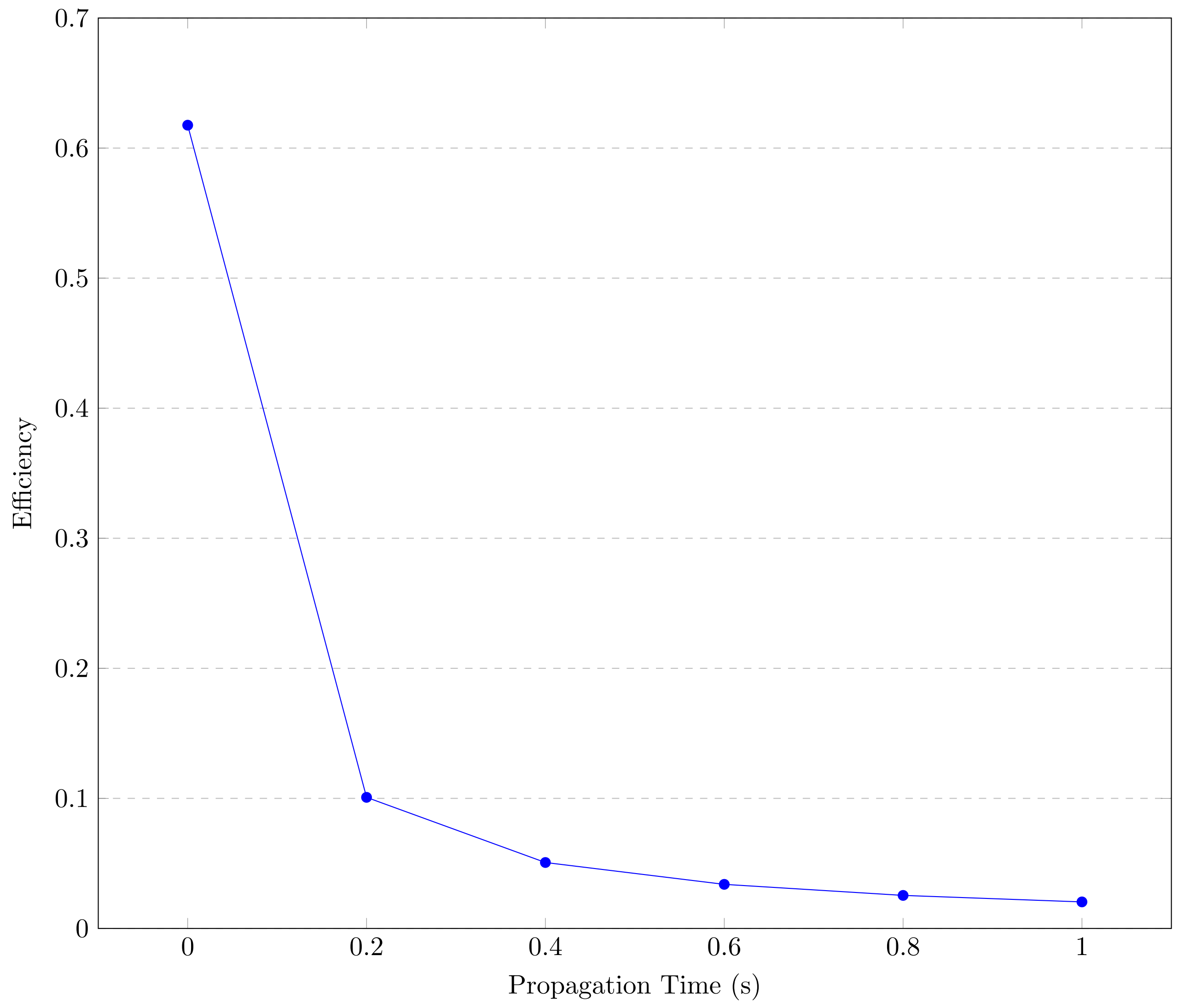
Efficiency

7.1 FER Variation



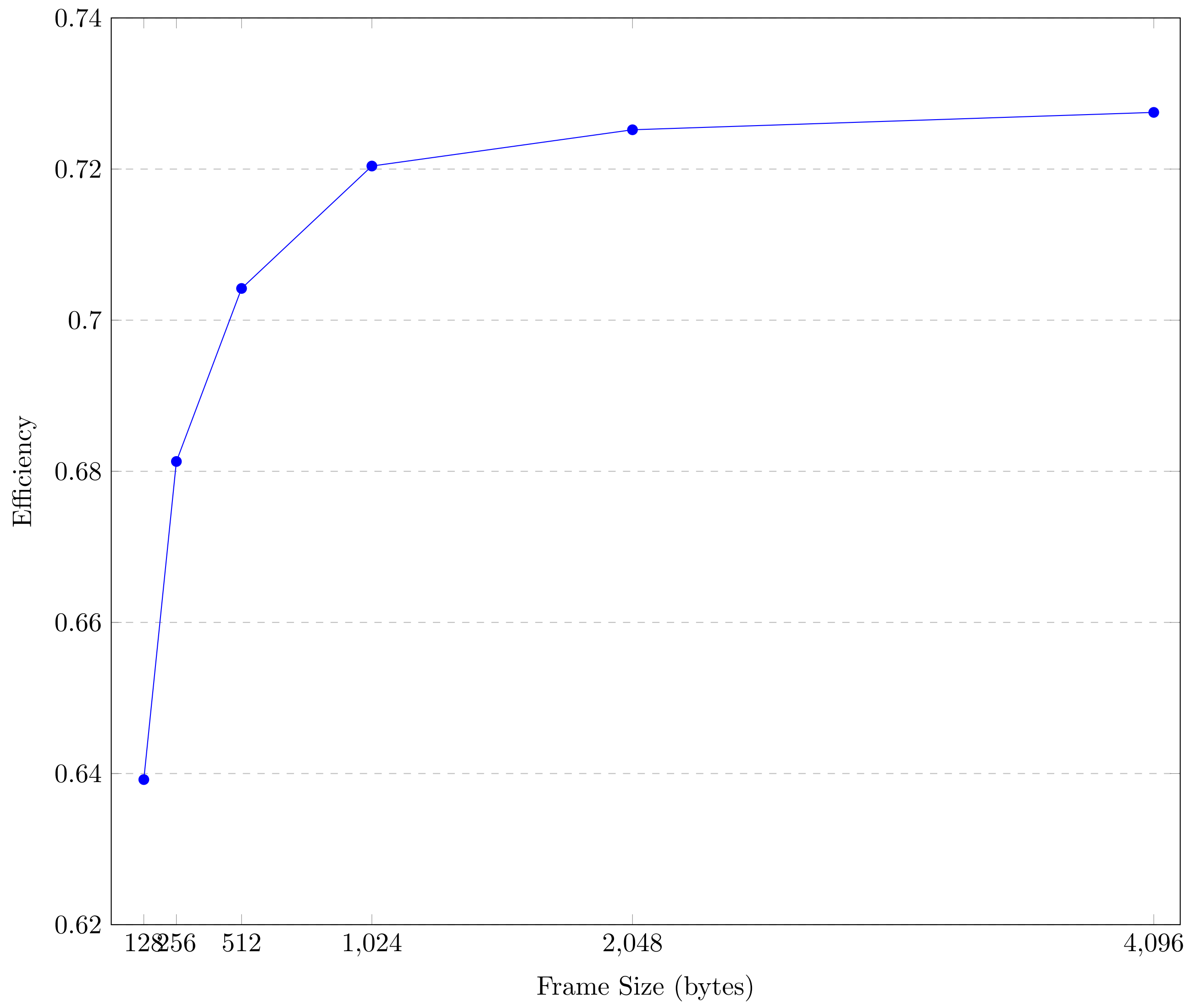
FER	Transmission Time	R (N^{er} Bits / Time)	S (R/C)
0	3.701	23715	0.6176
1	3.828	22922	0.5969
2	3.857	22749	0.5924
4	3.856	22755	0.5926
8	4.431	19802	0.5157
16	5.246	16726	0.4356

7.2 Propagation Time (T_Prop) Variation



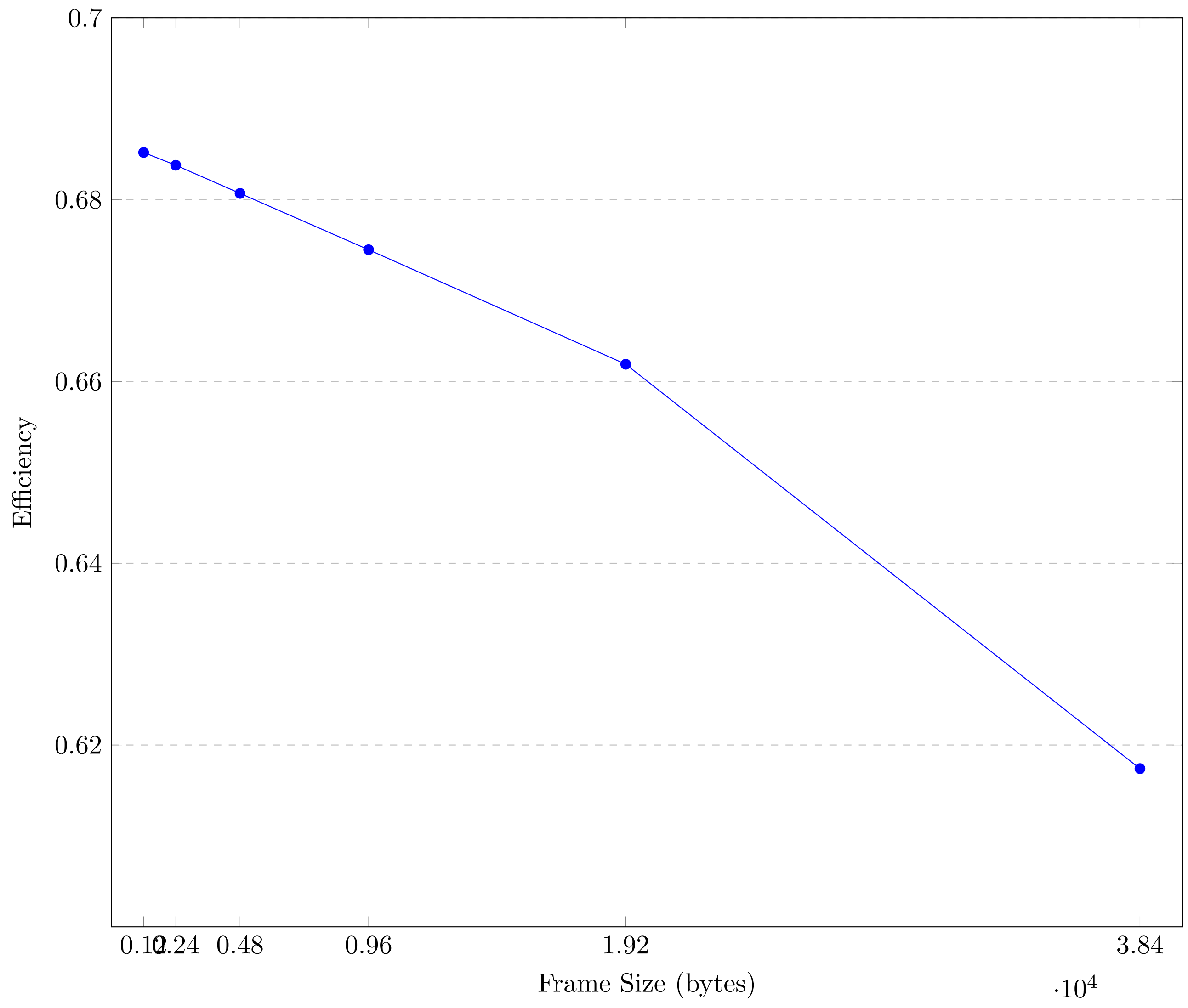
Propagation Time	Transmission Time	R (N^{er} Bits / Time)	S (R/C)
0.0	3.700	23715	0.6176
0.2	22.674	3870	0.1008
0.4	45.072	1947	0.0507
0.6	67.469	1301	0.0339
0.8	89.874	976	0.0254
1.0	112.271	782	0.0204

7.3 Frame Size (Chunk Size) Variation



Frame Size	Transmission Time	R (N^{er} Bits / Time)	S (R/C)
128	3.575	24544	0.6392
256	3.354	26161	0.6813
512	3.245	27040	0.7042
1024	3.172	27662	0.7204
2048	3.151	27846	0.7252
4096	3.141	27935	0.7275

7.4 Connection Capacity (C) Variation



Frame Size	Transmission Time	R (N^{er} Bits / Time)	S (R/C)
1200	106.709	822	0.6852
2400	53.465	1641	0.6838
4800	26.856	3267	0.6807
9600	13.551	6475	0.6745
19200	6.904	12709	0.6619
38400	3.141	23708	0.6174

Chapter 8

Conclusion

To resume what has been said up above, our program followed the project guidelines and is overall efficient. We believe that our protocol was implemented as intended by the course unit.

This project ended up being a good way of strengthen our knowledge about RCOM. Emphasizing the code structure on layer independence and implementing the Stop&Wait protocol gave us a clear vision of what the project goals were. We were very happy with the ending result of the project.

Due to the current pandemic, we weren't able to freely attend the classroom to test our program. This impacted our workflow heavily, since we had to connect with SSH to the laboratory PCs at the same time as other people which raised many problems. One of them being that this project took much more time out of our hands than we expected. However, we still managed to deliver everything on time.

8.1 Note about number of pages

We're aware that it was requested to keep the number of pages of this report up to a maximum of eight. However, we wanted to include the necessary information and illustrations, while still maintaining the content spaced out and structured to allow for easier readability. We hope that the professor understands.

Chapter 9

Attachments

9.1 main.c

```c
#include "app.h"

void parse_flags(int argc, char *argv[]);

int main(int argc, char **argv) {
    app = (applicationLayer *)malloc(sizeof(applicationLayer));
    llink = (linkLayer *)malloc(sizeof(linkLayer));

    // Parse commnad line arguments
    parse_flags(argc, argv);

    // Ask app to establish connection
    int fd = llopen(app->port, app->status);

    if (fd < 0) {
        perror("Failed to establish connection.\n");
        free(app);
        free(llink);
        exit(1);
    }

    //Start file transmission
    file_transmission();

    //End connection
    if(llclose(fd) < 0){
        perror("Failed to close connection.\n");
        free(app);
        free(llink);
        exit(1);
    }

    free(app);
    free(llink);

    return 0;
}
```

```c
void parse_flags(int argc, char** argv) {
    app->chunk_size = MAX_CHUNK_SIZE;

    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], ""))
            continue;

        // Serial Port
        if (!strncmp(argv[i], "/dev/ttyS", 9))
            strcpy(app->port, argv[i]);

        // Status
        else if (!strcmp(argv[i], "-c") || !strcmp(argv[i], "--client"))
            app->status = TRANSMITTER;

        else if (!strcmp(argv[i], "-s") || !strcmp(argv[i], "--server"))
            app->status = RECEIVER;

        // Timeout
        else if (!strcmp(argv[i], "-t") || !strcmp(argv[i], "--timeout"))
            llink->timeout = atoi(argv[i + 1]);

        // Number of transmissions
        else if (!strcmp(argv[i], "-n") || !strcmp(argv[i], "--num_transmissions"))
            llink->numTransmissions = atoi(argv[i + 1]);

        // File to transfer
        else if (!strcmp(argv[i], "-f") || !strcmp(argv[i], "--file"))
            strcpy(app->filename, argv[i + 1]);

        // Chunk size
        else if (!strcmp(argv[i], "--chunk_size"))
            app->chunk_size = atoi(argv[i + 1]);
    }
}
```

9.2 link.c

```c
#include "link.h"
#include <time.h>

struct termios oldtio, newtio;
struct sigaction action;
bool flag = true;
int fd, alarm_counter = 0;

void alarm_handler();

/**
 * Function used for byte stuffing.
 */
int byte_stuffing(unsigned char* packet, int length, unsigned char** frame) {
    *frame = NULL;
    *frame = (unsigned char*) malloc(length * 2);

    int index = 0;
    // Fill the frame, replacing flage and escape occurrences
    for (int c = 0; c < length; c++) {
        if (packet[c] == FLAG) { // Stuff Flag
            (*frame)[index] = ESCAPE;
            (*frame)[++index] = FLAG_STUFF;
        }

        else if (packet[c] == ESCAPE) { // Suff Escape
            (*frame)[index] = ESCAPE;
            (*frame)[++index] = ESCAPE_STUFF;
        }

        else
            (*frame)
                [index] = packet[c]; // Keep value

        index++;
    }

    return index; // Return size of result
}

/**
 * Function for byte destuffing.
 */
int byte_destuffing(unsigned char* packet, int length, unsigned char** frame) {
    *frame = NULL;
    *frame = (unsigned char*) malloc(length);

    int index = 0;
    // Fill the frame, replacing escaped occurrences
    for (int c = 0; c < length; c++) {
        if (packet[c] == ESCAPE)
```

```
            (*frame)[index] = packet[++c] ^ 0x20;
        else
            (*frame)[index] = packet[c];

        index++;
    }

    return index; // Returns size of result
}

/**
 * Creates and sends a supervision/unnumbered frame.
 */
int write_su_frame(int fd, char a, char c) {
    unsigned char buf[5];

    buf[0] = FLAG;  // F
    buf[1] = a;     // A
    buf[2] = c;     // C
    buf[3] = a ^ c; // BCC
    buf[4] = FLAG;  // F

    return write(fd, buf, 5);
}

/**
 * Creates an information frame.
 */
int create_information_frame(unsigned char* packet, int length, unsigned char**
↪   frame) {
    // Calculates BCC2
    unsigned char BCC_2 = packet[0];
    for (int i = 1; i < length; i++)
        BCC_2 ^= packet[i];

    // Byte stufing
    unsigned char *stuffed_bcc, *stuffed_data;
    int bcc_length = byte_stuffing(&BCC_2, 1, &stuffed_bcc);       // Byte-stuff
    ↪   BCC_2
    int new_length = byte_stuffing(packet, length, &stuffed_data); // Byte-stuff
    ↪   packet

    // Allocs memory for frame
    *frame = (unsigned char*) malloc(new_length + 5 + bcc_length);

    // Fills frame
    (*frame)[0] = FLAG;                                     // F
    (*frame)[1] = A_EM_CMD;                                 // A
    (*frame)[2] = llink->sequenceNumber & SEQUENCE_MASK_S; // Sequence number
    (*frame)[3] = A_EM_CMD ^ (*frame)[2];                  // BCC_1
    memcpy(*frame + 4, stuffed_data, new_length);          // Stuffed Packets

    new_length += 4;                                      // Updates length (4 bytes
    ↪   from F,A,Ns,BCC)
    memcpy(*frame + new_length, stuffed_bcc, bcc_length); // BCC_2
    new_length += bcc_length;                            // Updates length (adds
     ↪  bcc 2 length)

    (*frame)[new_length++] = FLAG; // F

    free(stuffed_bcc);
    free(stuffed_data);

    return new_length; // Returns length of result
}

/**
 * Sends and information frame.
 */
int write_info_frame(int fd, unsigned char* packet, int length) {
    // Prepares frame to send
    unsigned char* frame;
    int frame_length = create_information_frame(packet, length, &frame);

    // Start state machine
    struct state_machine stm;
    stm.current_state = START;
    stm.status = TRANSMITTER;

    // Reset Values
    alarm_counter = 0;
    flag = true;

    bool bcc_success = true;
    unsigned char bcc_val;
    char c_val, buffer[1];
    int writtenLen; // Written characters

    // Writing Frame Loop
    while (alarm_counter < llink->numTransmissions) {
        if (flag) {
            alarm(llink->timeout); // Activates alarm
            flag = false;

            if ((writtenLen = write(fd, frame, frame_length)) < 0) { // Writes
             ↪  frame
                perror("Failed to send information frame message.");
                break;
            }
            printf("Sent information frame. \n");
        }

        if (read(fd, buffer, 1) < 0) { // Reads response
            if (errno != EINTR) {      // Check if read was interrupted by alarm
                perror("Failed to read acknowledgement message.");
                break;
            }
        }

        else {
            change_state(&stm, buffer[0]); // Updates state of state machine

            switch (stm.current_state) {
                case A_ANSWER_RCV:
                    bcc_val = buffer[0]; // Stores A value
                    break;

                case C_RCV:
                    bcc_val ^= buffer[0]; // Calculates BCC 1
                    c_val = buffer[0];    // Stores C
                    break;

                case BCC_1_RCV:
                    if (bcc_val != buffer[0]) // Check if BCC is right
                        bcc_success = false;
                    break;

                case STOP:
                    if ((c_val == C_RR || c_val == (C_RR & SEQUENCE_MASK_R)) &&
                     ↪  bcc_success) { // Check if it is RR message
                        printf("Received RR message.\n");

                        alarm(0);                                       //
                         ↪  Deactivate alarm
                        llink->sequenceNumber = ~llink->sequenceNumber; // Update
                         ↪  sequence numebr (0 or 1)
                        free(frame);

                        return writtenLen;
                    }

                    else { // Received REJ frame. Need to resend I Frame (flag =
                     ↪  1).
                        printf("Received REJ message.\n");
                        flag = true;                 // Reset Flag
                        alarm_counter = 0;           // Reset alarm_counter
                        stm.current_state = START; //Restart state machine
                    }
            }
        }
    }

    if (alarm_counter == llink->numTransmissions)
        perror("Failed to receive acknowledgement.\n");

    free(frame);

    return -1;
}
```

```c
/**
 * Receives an information frame
 */
int read_info_frame(int fd, unsigned char** data_field) {
    // Start state machine
    struct state_machine stm;
    stm.status = RECEIVER;
    stm.current_state = START;

    char buffer[1];
    int data_counter = 0, length = 0;
    unsigned char bcc_val, frame[MAX_SIZE];
    bool received_info = false, bcc_success = true, discard_frame = false, change_Ns
     ↪  = false;

    // Reading loop
    while (!received_info) {
        if (read(fd, buffer, 1) < 0) {
            perror("Failed to read.");
            exit(1);
        }

        else {
            // Frame Error (FER)
            int a, b, p = 0;
            a = rand() % 100 + 1;
            b = rand() % 100 + 1;

            change_state(&stm, buffer[0]); // Update state in state machine

            switch (stm.current_state) {
                case A_CMD_RCV:          // Received A
                    bcc_val = buffer[0]; // Store A
                    break;

                case C_I_RCV: // Receive C
                    if (buffer[0] != (llink->sequenceNumber & SEQUENCE_MASK_S))
                        discard_frame = true; // Repeated frame. Discard.
                    else
                        change_Ns = true; // Frame is not repeated. Need to update
                         ↪  sequence number.

                case C_RCV: // Received C
                    bcc_val ^= buffer[0];
                    break;

                case BCC_1_RCV:
                    if (bcc_val != buffer[0] || a <= p) // Check BCC value
                        bcc_success = false;
                    break;

                case D_RCV:                            // Received data or BCC2
                    frame[data_counter++] = buffer[0]; // Store data received
                    break;

                case STOP: // Finished receiving data
                    printf("Received information frame.\n");

                    length = byte_destuffing(frame, data_counter, data_field); //
                     ↪  Destuffing of data

                    // Calculate BCC2
                    bcc_val = (*data_field)[0];

                    for (int i = 1; i < (length - 1); i++)
                        bcc_val ^= (*data_field)[i];

                    if (bcc_val != (*data_field)[length - 1] || b <= p) // Check
                     ↪  BCC2
                        bcc_success = false;

                    int written_len = 0;
                    // Decide if we need to send RR or REJ
                    if (bcc_success) {
                        received_info = true; // Finish loop

                        written_len = write_su_frame(fd, A_RC_RESP, C_RR |
                            ↪  (llink->sequenceNumber && SEQUENCE_MASK_R)); // Write RR
                            ↪  message.
                        printf("Sent RR message.\n");

                        if (change_Ns)
                            llink->sequenceNumber = ~llink->sequenceNumber; //
                             ↪  Update sequence number
                        change_Ns = false;                                // Reset
                         ↪  value
                    } else {
                        written_len = write_su_frame(fd, A_RC_RESP, C_REJ |
                            ↪  (llink->sequenceNumber && SEQUENCE_MASK_R)); // Write
                            ↪  REJ message.
                        printf("Sent REJ message.\n");
                    }

                    if (!bcc_success || discard_frame) { // Need to reset values if
                     ↪  we received repeated frame or if the bcc is wrong
                        // Reset values
                        stm.current_state = START;      // Reset state machine
                        memset(frame, 0, data_counter); // Clean up the frame
                        data_counter = 0;               // Reset frame counter
                        bcc_success = true;
                        discard_frame = false;
                        received_info = false;
                    }

                    if (written_len < 0) { // Verify written errors
                        perror("Failed to send acknowledgement message.");
```

```
                        exit(1);
                    }
                }
            }
        }

        // Propagation Time
        //usleep(0);

        return length;
}

/**
 * Opens serial port device and sets configurations.
 * Sends SET & UA frames.
 */
int establish_connection(char* port, enum Status status) {
    /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
    */
    fd = open(port, O_RDWR | O_NOCTTY);

    if (fd < 0) {
        perror(port);
        exit(-1);
    }

    /* Save current port settings */
    if (tcgetattr(fd, &oldtio) == -1) {
        perror("tcgetattr");
        exit(-1);
    }

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    /* Set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; /* Inter-character timer unused */
    newtio.c_cc[VMIN] = 1;  /* Blocking read until 5 chars received */

    /*
    VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
    leitura do(s) próximo(s) caracter(es)
    */

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
```

```c
        exit(-1);
    }

    printf("New termios structure set\n");

    // Set state machine
    struct state_machine stm;
    stm.status = status;
    stm.current_state = START;

    llink->sequenceNumber = 0; // Initializes sequence number
    char buf[5];
    if (status == TRANSMITTER) { // Transmitter
        // Set alarm handler
        action.sa_handler = &alarm_handler;
        sigemptyset(&action.sa_mask);
        action.sa_flags = 0;

        // Installs co-routine that attends interruption
        if (sigaction(SIGALRM, &action, NULL) < 0) {
            perror("Failed to set SIGALARM handler.\n");
            exit(1);
        }

        while (alarm_counter < llink->numTransmissions) {
            if (flag) {
                alarm(llink->timeout); // Activactes alarm
                flag = false;

                if (write_su_frame(fd, A_EM_CMD, C_SET) < 0) { // Sends SET message
                    perror("Failed to send SET message.");
                    exit(1);
                }
                printf("Sent SET message. \n");
            }

            if (read(fd, buf, 1) < 0) { // Receive UA message
                if (errno != EINTR) {   // Check if read was interrupted by an
                ↪  alarm
                    perror("Failed to read UA message.");
                    exit(1);
                }
            }

            else {
                change_state(&stm, buf[0]);      // Update state of state machine
                if (stm.current_state == STOP) { // Successfully received UA
                ↪  message
                    printf("Received UA message.\n");
                    alarm(0); // Deactivate alarm
                    break;    // Stop reading loop
                }
            }
        }
        if (alarm_counter == llink->numTransmissions) { // Check timeout
            perror("Failed to establish connection.\n");
            exit(1);
        }
    }

    else if (status == RECEIVER) {
        bool receivedSet = false;

        while (!receivedSet) {
            if (read(fd, buf, 1) < 0) { // Receive SET message
                perror("Failed to read SET message.");
                exit(1);
            } else {
                change_state(&stm, buf[0]);    // Update state of state machine.
                if (stm.current_state == STOP) // Sucessfully read SET message
                    receivedSet = true;
            }
        }
        printf("Received SET message.\n");

        if (write_su_frame(fd, A_RC_RESP, C_UA) < 0) { // Send UA message
            perror("Failed to send UA message.");
            exit(1);
        }
        printf("Sent UA message.\n");
    }
    printf("Established connection\n");
    return fd;
}

/**
 * Finishes connection.
 * Sends DISC Frames.
 */
int finish_connection(int fd, enum Status status) {
    // Set up state machine
    struct state_machine stm;
    stm.status = status;
    stm.current_state = START;

    alarm_counter = 0;
    flag = true;
    char buf[1];
    if (status == TRANSMITTER) {
        // Tries to send DISC Message
        while (alarm_counter < llink->numTransmissions) {
            if (flag) {
                alarm(llink->timeout); // Activactes alarm
                flag = false;

                if (write_su_frame(fd, A_EM_CMD, C_DISC) < 0) { // Sends DISC
                 ↪  message.
```

```
                    perror("Failed to send DISC message.");
                    exit(1);
                }
                printf("Sent DISC message. \n");
            }

            if (read(fd, buf, 1) < 0) { // Tries to receive DISC message
                if (errno != EINTR) {   // Check if read was interrupted by alarm.
                    perror("Failed to read DISC message.");
                    exit(1);
                }
            } else {
                change_state(&stm, buf[0]); // Update state of state machine

                if (stm.current_state == STOP) { // Successfully read a DISC
                ↪   message
                    printf("Received DISC message.\n");
                    alarm(0); // Cancel alarm.

                    if (write_su_frame(fd, A_EM_CMD, C_UA) < 0) { // Sends UA
                    ↪   message
                        perror("Failed to send UA message.");
                        exit(1);
                    }
                    printf("Sent UA message. \n");
                    break; // Stop loop.
                }
            }
        }

        if (alarm_counter == llink->numTransmissions) {
            perror("Failed to finish connection.\n");
            exit(1);
        }

    } else if (status == RECEIVER) {
        bool receivedDISC = false;
        while (!receivedDISC) {
            if (read(fd, buf, 1) < 0) { // Receives DISC message
                perror("Failed to read DISC message.");
                exit(1);
            } else {
                change_state(&stm, buf[0]); // Update state machine

                if (stm.current_state == STOP) // Sucessfully read DISC message.
                    receivedDISC = true;
            }
        }
        printf("Received DISC message.\n");

        if (write_su_frame(fd, A_RC_CMD, C_DISC) < 0) { // Sends DISC message.
            perror("Failed to send DISC message.");
            exit(1);
        }
        printf("Sent DISC message.\n");

        bool receivedUA = false;
        while (!receivedUA) {
            if (read(fd, buf, 1) < 0) { // Tries to receive UA message
                perror("Failed to read UA message.");
                exit(1);
            } else {
                change_state(&stm, buf[0]); // Update state machine.

                if (stm.current_state == STOP) // Sucessfully read UA message.
                    receivedUA = true;
            }
        }
        printf("Received UA message.\n");
    }
    printf("Finished connection.\n");
    return 0;
}

/**
 * Handles an alarm signal.
 */
void alarm_handler() {
    printf("Alarm # %d\n", alarm_counter);
    flag = true; // Resets flag so it tries again
    alarm_counter++;
}
```

9.3 app.c

```c
#include "app.h"

#include <math.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/time.h>

FILE* fp;

/**
 * Creates a control packet based on file.
 */
int control_packet(enum Control status, char* filename, int filesize, unsigned
↪  char** packet) {
    int L1 = ceil(log(filesize) / log(2) / 8); // Calculates L1
    size_t L2 = strlen(filename); // Calculates L2

    // Allocs memory for packet
    int packet_size = 3 + L1 + 2 + L2; // Calculates size of packet (C, T1, L1, V1,
    ↪  T2, L2, V2)
    *packet = NULL;
    *packet = (unsigned char*) malloc(packet_size);

    // Creates packet
    int c_ind = 0, t1_ind = 1, l1_ind = 2, v1_ind = 3; // indexes
    (*packet)[c_ind]  = status;      // C
    (*packet)[t1_ind] = T_FILESIZE; // T1
    (*packet)[l1_ind] = L1;          // L1

    // V1
    int c;

    for (c = v1_ind; c < (L1 + v1_ind); c++)
        (*packet)[c] = (filesize >> 8 * (c - v1_ind)) & (0xFF); // V1

    (*packet)[c++] = T_FILENAME; // T2
    (*packet)[c++] = L2;         // L2

    // V2
    int v2_ind = c;

    for (; c < L2 + v2_ind; c++)
        (*packet)[c] = filename[c - v2_ind];

    return packet_size;
}

/**
 * Create a data packet.
 * Returns the length of the created packet.
 */
int data_packet(unsigned char* data_field, int data_size, unsigned char** packet) {
    int L1 = data_size & 0xFF; // Calculates L1
    int L2 = data_size >> 8; // Calculates L2

    // Allocs memory for packet
    int packet_length = 4 + data_size; // Length of result packet will be data size+
    ↪  4 bytes (C, N, L1, L2)
    *packet = NULL;
    *packet = (unsigned char*) malloc(packet_length);

    // Creates packets
    int C_ind = 0, N_ind = 1, L2_ind = 2, L1_ind = 3, data_ind = 4; // indexes
    (*packet)[C_ind] = data;
    (*packet)[N_ind] = app->sequence_number;
    (*packet)[L2_ind] = L2;
    (*packet)[L1_ind] = L1;
    memcpy(*packet + data_ind, data_field, data_size); // Adds data to packet

    return packet_length;
}

/**
 * Opens file with name filename.
 * If transmitter, opens for reading. Otherwise it opens for writing.
 * Returns a struct which contains the size of the file.
 */
struct stat open_file(char* filename) {
    // Open file
    if (app->status == TRANSMITTER)
        fp = fopen(filename, "r"); //Open for reading

    else // RECEIVER
        fp = fopen(filename, "w"); //Open for writing

    if (fp == NULL) {
        perror("Failed to open file.\n");
        exit(1);
    }

    // Stat call to get filesize
    struct stat st;
    if (stat(filename, &st) < 0) {
        perror("Failed stat call.\n");
        exit(1);
    }

    return st;
}

/**
 * Establishes an connection.
 */
```

```c
int llopen(char* port, enum Status status) {
    // Tells link layer to establish a connection
    int fd = establish_connection(port, status);

    //Update struct values
    app->fileDescriptor = fd;
    app->sequence_number = 0;

    return fd;
}

/**
 * Closes file and finishes connection.
 */
int llclose(int fd) {
    // Close file
    if (fclose(fp) < 0) {
        perror("Failed to close file.\n");
        exit(1);
    }

    // Tells link layer to finish connection
    if(finish_connection(fd, app->status) < 0)
        return -1;

    return close(fd);
}

/**
 * Sends packets to link layer.
 */
int llwrite(int fd, unsigned char* buffer, int length) {
    return write_info_frame(fd, buffer, length);
}

/**
 * Receives packets from link layer.
 */
int llread(int fd, unsigned char** buffer) {
    return read_info_frame(fd, buffer);
}

/**
 * Prints a progress bar
 * Showing the current transmission progress
 */
void progress_bar(float progress) {
    int length = 50;

    for (int i = 0; i <= length * progress; i++)
        printf("\u2588"); //

    for (int i = length * progress; i < length; i++)
```

```
        printf("\u2581"); //

    printf("  %.2f%% Complete\n", progress * 100);
}

/**
 * Sends file in case of transmitter.
 * Receives file in case of receiver.
 */
int file_transmission() {
    struct stat st;

    // Transmitter
    if (app->status == TRANSMITTER) {
        st = open_file(app->filename); // Open file to send

        // Creates Start Packet
        unsigned char* packet;
        int packet_size = control_packet(start, app->filename, st.st_size, &packet);

        // Sends Start packet
        if(llwrite(app->fileDescriptor, packet, packet_size) < 0){
            free(packet);
            perror("Failed to send start packet.\n");
            exit(1);
        }
        free(packet);

        unsigned char file_array[st.st_size]; //where image is stored
        if(fread(file_array, 1, st.st_size, fp) < 0){ // Reads from file and stores
        ↪   in file_array
            perror("Failed to read from file.\n");
            exit(1);
        }

        // Calculate the number of chunks in which the file will be split
        int num_chunks = ceil(st.st_size / (double) app->chunk_size);

        // Creates data packets
        size_t length = app->chunk_size;
        int file_index = 0;

        for (int i = 0; i < num_chunks; i++) {
            if(i == num_chunks - 1)  // Last iteration
                length = st.st_size - (length * i) ;

            packet_size = data_packet(file_array + file_index, length, &packet); //
            ↪  Prepares a data packet
            file_index += app->chunk_size;

            //Sends data packet
            if(llwrite(app->fileDescriptor, packet, packet_size) < 0){
                free(packet);
                perror("Failed to send data packet.\n");
                exit(1);
            }

            free(packet);

            // Update sequence number
            app->sequence_number = (app->sequence_number + 1) % 255;
        }

        // Creates End Packet
        packet_size = control_packet(end, app->filename, st.st_size, &packet);


        // Sends End packet
        if(llwrite(app->fileDescriptor, packet, packet_size) < 0){
            free(packet);
            perror("Failed to send end packet.\n");
            exit(1);
        }
        free(packet);
    }

    // Receiver
    else if (app->status == RECEIVER) {
        int L1_index, L2_index;
        int L1, L2;

        unsigned char* file_array;
        int filesize = 0, file_index = 0;

        bool transmission_ended = false;
        while (!transmission_ended) {
            // Receive packet
            unsigned char* buffer;

            if(llread(app->fileDescriptor, &buffer) == 0)
                continue; // Empty packet

            switch (buffer[0]) {
                case start: // Received Start Packet
                    //Parsing Data
                    // L1
                    L1_index = 2;
                    L1 = buffer[L1_index];

                    // L2
                    L2_index = 4 + L1; // Skip 4 bytes (C, T1, L1 and T2) and V1
                     ↪   field (of size L1)
                    L2 = buffer[L2_index];

                    // V1 - File Size
                    int V1_index = L1_index + 1;
                    for (int i = 0; i < L1; i++)
                        filesize += buffer[V1_index + i] * pow(256, i);

                    // V2 - File Name
                    int V2_index = L2_index + 1;
                    char* filename = (char*) malloc(L2);
                    memcpy(filename, buffer + V2_index, L2);

                    // Open file
                    // st = open_file(filename);
                    st = open_file("../files/received.gif");
                    // Create file array where data received will be written to
                    file_array = (unsigned char*) malloc(filesize);

                    free(filename);

                    break;

                case data: // Received Data Packet
                    L1_index = 3;
                    L2_index = 2;
                    L1 = buffer[L1_index]; // L1
                    L2 = buffer[L2_index]; // L2

                    int K = 256 * L2 + L1; // K

                    //Sequence number
                    int N_index = 1;
                    int N = buffer[N_index]; // N - Sequence number

                    if (N > app->sequence_number)
                        break; // Ignore repeated packet

                    app->sequence_number = (app->sequence_number + 1) % 255; //
                     ↪  Update sequence number

                    // Save data to file array
                    int data_index = 4;
                    memcpy(file_array + file_index, buffer + data_index, K);
                    file_index += app->chunk_size;

                    // Update progress bar
                    float progress = file_index / (float) filesize;

                    if (progress > 1)
                        progress = 1;

                    progress_bar(progress);

                    break;

                case end: // Received End Packet
                    // Writes contents of array to file
```

```
                    fwrite(file_array, 1, filesize, fp);
                    // Ends cycle
                    transmission_ended = true;

                    free(file_array);

                    break;
                }
            }
        }

        return 0;
    }
```

9.4 state_machine.c

```
#include "state_machine.h"
#include <stdio.h>

void change_state(struct state_machine* stm, char field) {
    switch (stm->current_state) {
        case START:
            if (field == FLAG)  // Received FLAG
                stm->current_state = FLAG_RCV;
            break;

        case FLAG_RCV:
            if(field == FLAG) // Received FLAG
                break; // Do nothing

            else if (stm->status == RECEIVER) {
                if (field == A_EM_CMD) // Received A command
                    stm->current_state = A_CMD_RCV;

                else if (field == A_EM_RESP) // Received A response
                    stm->current_state = A_ANSWER_RCV;

                else stm->current_state = START; // Received other
            }

            else if (stm->status == TRANSMITTER) {
                if (field == A_RC_CMD)  // Received A command
                    stm->current_state = A_CMD_RCV;

                else if (field == A_RC_RESP) // Received A response
                    stm->current_state = A_ANSWER_RCV;

                else stm->current_state = START; // Received other
            }

            break;

        case A_CMD_RCV:
            if (field == C_SET || field == C_DISC) // Received C (SET or DISC)
                stm->current_state = C_RCV;

            else if (field == NS_1 || field == NS_2) // Received sequence number
                stm->current_state = C_I_RCV;

            else stm->current_state = START; // Received other
            break;

        case A_ANSWER_RCV:
            if (field == C_RR || field == C_REJ || field == C_UA) // Received C (RR
             ↪ or REJ or UA)
                stm->current_state = C_RCV;
            else stm->current_state = START; // Received other
            break;

        case C_I_RCV:
        case C_RCV:
            stm->current_state = BCC_1_RCV;
            break;

        case BCC_1_RCV:
            if (field == FLAG) // Received Flag
                stm->current_state = STOP;
            else stm->current_state = D_RCV; // Otherwise, we're receiving data
            break;

        case D_RCV:
            if (field == FLAG) // Finished receiving DATA
                stm->current_state = STOP;
            break;
    }
}
```

9.5 Makefile

```
# Compiler and linker
CC = gcc

# Flags
CFLAGS = -Wall -g

main:
	@gcc ${CFLAGS} -o main app.c link.c main.c state_machine.c -lm

clean:
	@rm -f main

initialize:
	@sudo socat -d -d PTY,link=/dev/ttyS0,mode=777 PTY,link=/dev/ttyS1,mode=777

client: clean main
	@./main -c "/dev/ttyS0" --timeout 5 --num_transmissions 3 --chunk_size 100
	↪  --file "../files/pinguim.gif"

server: clean main
	@./main -s "/dev/ttyS0" --chunk_size 100
```