Computer Networks - 1 Project Report

Diogo Martins 202207944
Rubem Neto 202207086

8, November 2024

1 Outline

This project report presents the design and implementation of a custom, two-layered protocol to facilitate reliable file transfer between computers over RS-232 serial connections. Developed as part of a Computer Networks course, the project's objective was to create a protocol capable of managing typical challenges in serial communication, such as error detection, byte stuffing, and efficient flow control, ensuring reliable data transfer even under less-than-ideal network conditions.

Testing demonstrated that the protocol effectively handles varying baud rates, Frame Error Ratios (FER), packet sizes, and simulated transmission distances, achieving reliable file transfer across diverse conditions. The study highlights the significant effects of baud rate, FER, and packet size on protocol efficiency, with results aligning well with the theoretical expectations of Stop-and-Wait protocol models.

2 Introduction

2.1 Objectives

In this project, our goal is to implement a two-layered communication protocol and transmit a data file between two computers using this protocol. All of the communication was made using the serial port.

The final application must transfer the files satisfactorily and efficiently, implementing all of the proper error handling, syncing, and framing of the information.

The structure of this report is defined by:

- Architecture - A description of the functional blocks and interfaces.
- Code Structure - APIs, main structures, main functions, and their relationship in the architecture.
- Main Use Cases - Identification of main use cases and main sequences of function calls.
- Logical Link Protocol - Description of main aspects of the protocol and explanation of choices and implementation strategies.
- Application Protocol - Description of main aspects of the protocol and explanation of choices and implementation strategies.
- Validation - Description of tests performed.
- Data Link Protocol Efficiency - Statistical Analysis of the protocol's efficiency and comparison with a theoretical characterization of a Stop and Wait Protocol.
- Conclusions - Summary of the information presented along the report and reflection on the objectives.
- Appendix - Source Code and Graphs used.

3 Architecture

3.1 Layers

The data connection protocol developed is divided into two different layers that communicate with each other, one at a higher level than the other. The Application Layer invokes the functions implemented by the Data Link Layer, which thus offers an API to it.

3.1.1 Application Layer

The high-level layer, is the one closest to the user and the one that interacts with the Data Link Layer and the computers' files. It has the responsibility of reading the file, splitting it into multiple fragments, and sending them to the Data Link Layer. On the other side, it has to read the fragments from the Data Link Layer and put them all together again to build the original file.

3.1.2 Data Link Layer

The low-level layer, is the one that communicates with the serial port driver. It is responsible to begin and terminate the connection, sending and receiving formatted frames, as well as validating them through state machines. This layer must catch any error on the transmission and send appropriate error messages.

3.2 Interfaces

The program is executed in two different terminals, one in transmitter mode and the other in the receiver. Running the program, we must specify in which mode we want to execute it, as well as pass the file to transmit and the location of the serial port. We then have a different path of commands in the source code for each mode.

4 Code Structure

4.1 Link Layer

The Link Layer is responsible for handling all of the communication between devices at a lower level, interacting almost directly with the serial port. It also uses several auxiliary state machines to determine the validity of received and sent messages.

4.1.1 Structures

In this project, the use of C structures and C enumerations played a role in helping the implementation of the required functions.

```
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char *serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

- LinkLayerRole: To determine whether the application was running on transmitter mode or receiver mode.
- LinkLayer: To store all of the information related to the configuration of the protocol.
  - serialPort: Serial Port File Descriptor.
  - role: Role on the communication.
  - baudRate: The Baud Rate that will be used in the protocol.
  - nRetransmissions: Number of times the transmitter will send a message until it stops sending.

  - timeout: Amount of time, in seconds, that the transmitter will wait until re-sending a message.

4.1.2 link_layer.c

- llopen(LinkLayer connectionParameters): This function opens the connection for the transmitter and receiver.
- llwrite(const unsigned char *buf, int bufSize): Sends information using the serial port connection.
- llread(unsigned char *packet): Reads information using the serial port connection.
- llclose(int showStatistics): Closes the connection between the receiver and transmitter.
- sendData(const unsigned char *buf, int bufSize): Helper function that sends the buffer bytes.
- sendMessage(int A,int C): Helper function that sends control messages.

4.1.3 state_machine.c

- void state_machine_sendSET(unsigned char byte, int *state, bool tx): Validates received a message after a control message was sent.
- void state_machine_control_packet(int *state, unsigned char byte, int frame_nr): Validates a message sent from the receiver to the transmitter as a response of a sent information buffer.
- void state_machine_writes(int *state, unsigned char byte, bool bcc2_checked, int frame_ns): Validates the information frame (where the information buffer is present)
- void state_machine_close(int *state, unsigned char byte): Validates if a message is a valid DISC message, that closes the communication.

4.2 Application Layer

In this layer, the high-level operations are done, for example, splitting the file into different packets, sending one by one (using the link layer) and using the link layer for the communication.

4.2.1 Structures

Similarly to the previous section, the use of structures helped on the implementation of the protocol.

- ctrlPacket: Helps build and manipulate the control packet.
- dataPacket: Helps build and manipulate the data packet.

```
typedef struct
{
    unsigned char type;
    unsigned char file_name[UCHAR_MAX +
        1];
    size_t   file_size ;
} ctrlPacket;

typedef struct
{
    int sequence_number;
    size_t payload_length;
    unsigned char
        packet[MAX_PAYLOAD_SIZE];
} dataPacket;
```

4.2.2 application_layer.c

- applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename): Starting point of the application, where the connection is opened and the transfer of the file is made.
- application_layer_rx_protocol(const char *filename): Performs the operation of receiving the file and storing it.

- application_layer_tx_protocol(const char *filename): Performs the operation of reading and sending the file.

- build_control_packet(unsigned char *packet, crtlPacker ctrl_packet_info): Creates a control packet that defines the start and end of the transmission and some metadata.

- build_data_packet(unsigned char *packet, dataPacket data_packet): Creates the packets which have the file data.

5 Main Use Cases

As stated before, the program can be executed in two different modes, each one with a different behavior and objective.

5.1 Transmitter

1. Starts by opening the connection through llopen(), which exchanges control frames between the transmitter and the receiver. In this case, the transmitter sends a SET frame and waits for the receiver's UA response.

2. Through application_layer_tx_protocol(), this mode opens the file and sends the start control packet (including file size and file name) reasoning that it is ready to send data packets.

3. It then starts continuously reading fragments of the file and sending them over with llwrite(). These fragments are sent within a frame, containing a start and end flag, a frame identification, and two BCC's to perform validation. The fragment bytes are also stuffed to maintain the data integrity during the transmission.

4. Throughout the execution the program must care for errors in the transmission and adapt its behavior to solve them.

5. After sending all the fragments, a final control packet is sent (similar to the initial), and the connection is closed via llclose(), which exchanges disconnecting messages between roles.

5.2 Receiver

1. It starts by running llopen(), waiting for the SET message from the transmitter. On reception, a UA message is sent as a successful response.

2. The application_layer_rx_protocol() is then executed. Through llread(), this mode processes all information received, validating and, when needed, destuffing it. The first frame received must be a control packet containing the file name and size.

3. After that, and until an end packet is received, data packets containing file fragments must be read and deconstructed to write them into the file.

4. The connection is closed via llclose() after the end packet is transmitted between roles, signaling that the entire file was exchanged.

6 Logical Link Protocol

6.1 llopen

This method configures the serial port and, by sending and receiving messages, initializes the connection between the two devices.

6.1.1 Transmitter (tx)

Sends a SET message and waits until it receives a valid UA message. If no message is received in a time period (timeout), it sends the frame again and reset the waiting timer. There is a limit to the number of re-transmissions and if that limit is achieved, the program ends.

6.1.2 Receiver (rx)

Receives a SET message and sends a UA message. If no SET message is received, the program waits indefinitely until a valid message is received.

6.2 llwrite

Creates an information frame. The frame is sent until a valid response is received. If the message is a reject message, the frame is sent again. If no response is received in the timeout period, the message is re-sent. After sending a "number of re-transmissions" times and no response is received, the program ends.

If the message is successful, the program updates its frame number.

6.2.1 Stuffing

In the message buffer, there could be bytes with the same value as the flag and that could cause problems. For that, instead of sending the "flag" byte, the message is sanitized and its value is replaced by the values (0x7f and 0x5e).

```
if (buf[index_buf] == F_FLAG)
{
    send[index_send] = ESC;
    index_send++;
    bufSize++;
    send[index_send] = REPLACED;
}
else
{
    send[index_send] = buf[index_buf];
}
bcc2 = BCC(bcc2, buf[index_buf]);
index_buf++;
index_send++;
```

This code snippet shows how the stuffing is done while creating an information frame.

6.3 llread

This method reads the message and de-stuffs all of the appropriate bytes. Before a response is sent, the program checks for errors in the message. Those errors can be:

- BCC is not valid - That means that the right message was not received as expected, then the program ignores and waits for a new frame.
- BCC2 is not valid - This happens when the information on that frame contains errors, therefore the message cannot be processed. In this scenario, a rejection message is sent.

If the message is valid, it sends a receiver-ready response and updates the frame number.

6.4 llclose

This function closes the connection between the devices and shows statistics of the connection.

6.4.1 Transmitter (tx)

Sends DISC message, receives DISC message, sends UX message, and closes the connection. Works exactly like in the llopen and llwrite regarding the timeout and number of transmissions.

6.4.2 Receiver (rx)

Receives DISC, sends DISC, and closes the connection. If no valid message is received, it keeps waiting indefinitely.

7 Application Protocol

7.1 application_layer_tx_protocol

This function is called upon program execution in transmitter mode after the connection is opened. Firstly, a control packet is built and sent over to start the data transmission. After that, the file to send is read in chunks, and each one is sent within a data packet until the end of the file is reached. If no error occurs, the end control packet is sent to handshake at the end of the transmission.

7.2 application_layer_rx_protocol

This function is executed if the program is loaded in receiver mode. This module's task is to keep receiving packets from the transmitter and handle them based on their type. It expects the first packet to be an initial control one and the following ones to be of data type. It reads data packets and writes their content into the new file descriptor. If the end control packet is received, the handshake occurs, the transmission ends and the file is closed. While receiving data packets, this function must check the order as they arrive and validate it.

7.3 Utils

At this layer, other functions were defined to keep the code clean.

- build_control_packet (to format all the control packets)
- build_data_packet (to format all the data packets)
- calculate_file_size (to calculate the file size)
- formatScreen (to have a more user-friendly interface)

8 Validation

The application was tested with various arguments:

- Baud Rates of 9600, 19200, 38400, 57600 and 115200.
- Frame to Error Ratios (FER) of 10%, 20%, 30%, 45%, 50%.
- Packet Sizes of 400B, 600B, 1000B, 1500B and 2000B.
- Simulated distances (assuming 5µs/km) of 1000km, 500km, 10000km, 50000km and 100000km
- Disconnecting the communication for some time.
- Breaking the communication only on the transmitter and only in the receiver

All of the tests were successful.

9 Data Link Protocol Efficiency

9.1 Procedure

In order to measure the transfer speed of all bytes sent, we first initiated a timestamp when the connection was established and ended it when the connection was closed. During this time interval, we count the amount of bytes sent over and also the number of corrupted frames. After this measurement and following all the arguments mentioned in Validation to make comparisons, we can define link layer efficiency as follows:

$$E = \frac{Baudrate_r}{Baudrate_s}$$

9.2 Relationship between baud rate and efficiency

To test this relation, we ran the program with each baud rate mentioned in the Validation section, by changing the appropriate flag in the serial connection. As visible in **Fig. 1**, we can state that for bigger baud rate values, we have a slightly lower data transfer efficiency. This happens because, although bytes are transmitted faster, more resources are consumed to ensure accurate transmission, especially in noisy environments where higher baud rates require retransmissions.

9.3 Relationship between FER and efficiency

This relationship was built by manually changing the validation algorithm of a data frame, rejecting a frame on purpose to provoke an error message and, therefore, a re-transmission. As expected, in **Fig. 2**, we can see that more rejected frames seed a less efficient transmission.

9.4 Relationship between FER and time taken

Manipulating the validation algorithm as before, we were able to build the relation exhibited in **Fig. 3**, which shows that transmissions with higher error rates last more time. This is naturally expected in the Stop-and-Wait protocol, once an error occurs a re-transmission and therefore, more time.

9.5 Relationship between packet size and efficiency

To measure this, we changed the number of bytes read to build each fragment of the source file, at the Application Layer level. As graphically stated in **Fig. 4**, higher packet sizes mean a slight increase in the transmission efficiency. With lower packet sizes come higher transmission times, which means lower efficiency. Optimally, the connection should be made with a lower packet size, once it means more validation between layers and further reliability in the protocol.

9.6 Relationship between simulated distance and efficiency

We did this by adding a delay of 5 microseconds per km for each packet. Checking **Fig. 5**, we see that efficiency stays more or less constant until 10^4, dropping after it. This behavior is expected, as we are dependent on the Stop-and-Wait protocol, which uses a native delay of 10ms per packet.

10 Conclusions

Based on the goal of the project, there is no doubt that the achieved protocol performed all of the required tasks. Using the tests performed as a reference, the project had good efficiency. Also, it kept a steady performance when facing errors and different distances.

This project was a good learning experience since it permitted the grasp of different processes and concepts in the implementation of a communication protocol, such as byte stuffing and error handling.

11 Appendix

11.1 Source Code

11.1.1 link_layer.c

```
    // Link layer protocol implementation

#include "../include/link_layer.h"
#include "../include/serial_port.h"
#include "constants.h"
#include "statistics.h"
#include "state_machines.h"
#include <stdio.h>
#include <signal.h>
#include <stdbool.h>
#include <unistd.h>

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

int alarmEnabled = false;
int alarmCount = 0;
LinkLayer cp;
int file_descriptor = -1;

int frame_ns = 0;
int frame_nr = 1;

int counter = 0;
bool once = true;

void alarmHandler(int signal)
{
    alarmEnabled = false;
    alarmCount++;
}

int sendMessage(int A, int C)
{
    unsigned char buf[5] = {0};

    buf[0] = F_FLAG;
    buf[1] = A;
    buf[2] = C;
    buf[3] = BCC(A, C);
    buf[4] = F_FLAG;
```

```
    return writeBytesSerialPort(buf, 5);
}

int sendData(const unsigned char *buf, int bufSize)
{
    unsigned char send[BUF_SIZE + 1] = {0};
    unsigned char bcc2 = 0x00;

    send[0] = F_FLAG;
    send[1] = A_TX;
    send[2] = frame_ns == 1 ? C_FRAME1 : C_FRAME0;
    send[3] = BCC(send[1], send[2]);

    int index_buf = 0;
    int index_send = 4;

    while (index_send < bufSize + 4) // will this work?
    {
        if (buf[index_buf] == F_FLAG)
        {
            send[index_send] = ESC;
            index_send++;
            bufSize++;
            send[index_send] = REPLACED;
        }
        else
        {
            send[index_send] = buf[index_buf];
        }
        bcc2 = BCC(bcc2, buf[index_buf]);
        index_buf++;
        index_send++;
    }
    send[index_send] = bcc2;
    index_send++;
    send[index_send] = F_FLAG;

    return writeBytesSerialPort(send, index_send + 2);
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    cp = connectionParameters;
     file_descriptor  = openSerialPort(connectionParameters.serialPort,
```

```
                                    connectionParameters.baudRate);
int state = 0;
unsigned char byte = 0x00;
int tries = cp.nRetransmissions;
if ( file_descriptor < 0)
    return -1;

if (connectionParameters.role == LlTx)
{
    (void)signal(SIGALRM, alarmHandler);
    while ( tries != 0 && state != STOP)
    {
        while (alarmEnabled == true && state != STOP)
        {
            if (readByteSerialPort(&byte) > 0)
                state_machine_sendSET(byte, &state, true);
        }

        if (alarmEnabled == false)
        {
            alarmEnabled = true;
            alarm(connectionParameters.timeout);
            sendMessage(A_TX, C_SET);
        }
        tries--;
    }
    if (state != STOP)
    {
        printf("Timed out after %i tries!\n", cp.nRetransmissions);
        return -1;
    }
    else
        return 0;
}
else
{
    state = 0;
    byte = 0x00;

    while (state != STOP)
    {
        if (readByteSerialPort(&byte) > 0)
            state_machine_sendSET(byte, &state, false);
    }
    sendMessage(A_RC, C_UA);
}
return 0;
```

```
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    // F A C BCC1 D1 ... Dn BCC2 F
    if (once == true)
    {
        stat_start_timer();
    }
    once = false;

    int state = 0;
    int tries = cp.nRetransmissions;

    unsigned char byte = 0x00;

    // send i-ns and receive rr-nr
    (void)signal(SIGALRM, alarmHandler);

    alarmEnabled = false;

    while (state != STOP && tries != 0)
    {

        if (alarmEnabled == false)
        {
            sendData(buf, bufSize);
            alarm(cp.timeout);
            alarmEnabled = true;
        }

        while (alarmEnabled == true && state != STOP)
        {
            if (readByteSerialPort(&byte) > 0)
                state_machine_control_packet(&state, byte, frame_nr);

            if (state == RESEND)
            {
                stat_add_bad_frame();
                // stat_set_bits_received (bufSize + 4);

                printf("Error. Resending...\n");
                state = START;
                tries = cp.nRetransmissions + 1;
```

```
                break;
            }
        }

        tries--;
        alarmEnabled = false;
    }
    if (state != STOP)
    {
        printf("Timed out after %d tries!\n", cp.nRetransmissions);
        return -1;
    }
    else
    {
        stat_add_good_frame();
        stat_set_bits_received (bufSize + 5);

        double t_total = stat_get_t_total();
        double t_frame = (double)bufSize / (double)cp.baudRate;
        double frame_efficiency = t_frame / t_total;
        stat_add_total_efficiency (frame_efficiency);

        frame_ns = (frame_ns == 0 ? 1 : 0);
        frame_nr = (frame_nr == 0 ? 1 : 0);
    }
    return 0;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    if (once == true)
    {
        stat_start_timer();
    }
    once = false;

    int i = 0;
    int state = 0;
    unsigned char byte = 0x00;

    unsigned char bcc2 = 0x00;

    bool bcc_checked = false;
```

```
while (state != STOP)
{
    if (readByteSerialPort(&byte) > 0)
    {
        if (state == BCC2_CHECK)
        {
            packet[i - 1] = '\0';
            bcc_checked = bcc2 == 0;
            if (bcc_checked == false)
            {
                stat_add_bad_frame();
                // stat_set_bits_received (i + 4);
                sendMessage(A_TX, (frame_ns == 0 ? C_REJ0 : C_REJ1));
                state = START;
                printf("Error!\n");
                bcc2 = 0x00;
                i = 0;
            }
        }
        if (state == BCC_OK)
        {
            if (byte != ESC && byte != F_FLAG)
            {
                packet[i] = byte;
                i++;
                bcc2 = BCC(bcc2, byte);
            }
        }
        if (state == DATA_STUFFED)
        {
            if (byte == REPLACED)
            {
                packet[i] = F_FLAG;
                i++;
                bcc2 = BCC(bcc2, F_FLAG);
            }
            else if (byte == ESC)
            {
                packet[i] = ESC;
                i++;
                bcc2 = BCC(bcc2, ESC);
            }
            else
            {
                packet[i] = ESC;
                i++;
                bcc2 = BCC(bcc2, ESC);
```

```
                packet[i] = byte;
                i++;
                bcc2 = BCC(bcc2, byte);
            }
        }

        if (state == A_RCV)
        {
            if ((frame_ns == 0 && byte == 0x80) || (frame_ns == 1 && byte == 0x00))
            {
                frame_ns = (frame_ns == 0 ? 1 : 0);
                frame_nr = (frame_nr == 0 ? 1 : 0);
                sendMessage(A_TX, (frame_ns == 0 ? C_RR1 : C_RR0));
                return -1;
            }
        }

        state_machine_writes(&state, byte, bcc_checked, frame_ns);
    }
}

    sendMessage(A_TX, (frame_ns == 0 ? C_RR1 : C_RR0));
     stat_set_bits_received (i + 4);
    stat_add_good_frame();
    frame_ns = (frame_ns == 0 ? 1 : 0);
    frame_nr = (frame_nr == 0 ? 1 : 0);
    return 0;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{

    double time_taken = stat_get_time();
    double fer = stat_get_fer();
    double bad_frames = stat_get_bad_frames();
    double good_frames = stat_get_good_frames();
    double baud_rate = stat_get_bitrate(time_taken);
    double efficiency = baud_rate / cp.baudRate;
    double a = ((1 / efficiency) - 1) / 2;

    (void)signal(SIGALRM, alarmHandler);
    alarmEnabled = false;

    int state = 0;
```

```
unsigned char byte = 0x00;
if ( file_descriptor  < 0)
    return -1;

if (cp.role == LlTx)
{
    while (cp.nRetransmissions != 0 && state != STOP)
    {

        if (alarmEnabled == false)
        {
            sendMessage(A_TX, C_DISC);
            alarm(1);
            alarmEnabled = true;
        }

        while (alarmEnabled == true && state != STOP)
        {
            if (readByteSerialPort(&byte) > 0)
                state_machine_close(&state, byte);
        }
        cp.nRetransmissions--;
        alarmEnabled = false;
    }
    if (state != STOP)
        printf("Timed out after %d tries!\n", cp.nRetransmissions);
    else
        sendMessage(A_RC, C_UA);
}
else
{
    state = 0;
    byte = 0x00;

    while (state != STOP)
    {
        if (readByteSerialPort(&byte) > 0)
            state_machine_close(&state, byte);
    }
    state = 0;

    sendMessage(A_TX, C_DISC);
    while (state != STOP)
    {
        if (readByteSerialPort(&byte) > 0)
            state_machine_sendSET(byte, &state, true);
    }
```

```
    }

    printf("Closed!\n");

    if (showStatistics)
    {
        printf(" Statistics : \n");
        printf("|- Transfer time: %f\n", time_taken);
        printf("|- Bits sent: %f\n", baud_rate * time_taken);
        printf("|- Measured Baud-rate: %f\n", baud_rate);
        printf("|- Measured Baud-rate: %d\n", cp.baudRate);
        printf("|- FER: %f\n", fer);
        printf("|- Max frame Size: %d\n", MAX_PAYLOAD_SIZE);
        printf("|-  N  bad frames: %f\n", bad_frames);
        printf("|-  N  good frames: %f\n", good_frames);
        printf("|- Efficiency: %f\n", efficiency);
        printf("|- a: %f\n", a);
        printf("%f,%f,%f,%d,%f,%d,%f,%f,%f,%f\n",time_taken,baud_rate*time_taken,baud_rate,cp.baudRate,fer
    }

    int clstat = closeSerialPort();
    return clstat;
}
```

11.1.2 state_machines.c

```
    #include "state_machines.h"

void state_machine_sendSET(unsigned char byte, int *state, bool tx)
{
    switch (*state)
    {
    case START:
        if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = *state;
            break;
        }
    case FLAG_RCV:
        if (byte == F_FLAG)
        {
            *state = *state;
```

```
            break;
        }
        else if (byte == (tx ? A_RC : A_TX))
        {
            *state = A_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case A_RCV:
        if (byte == (tx ? C_UA : C_SET))
        {
            *state = C_RCV;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case C_RCV:
        if (byte == (tx ? (A_RC ^ C_UA) : (A_TX ^ C_SET)))
        {
            *state = BCC_OK;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case BCC_OK:
        if (byte == F_FLAG)
        {
```

```
            *state = STOP;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    default:
        break;
    }
}

void state_machine_control_packet(int *state, unsigned char byte, int frame_nr)
{
    switch (*state)
    {
    case START:
        if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = *state;
            break;
        }
    case FLAG_RCV:
        if (byte == F_FLAG)
        {
            *state = *state;
            break;
        }
        else if (byte == A_TX)
        {
            *state = A_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case A_RCV:
        if (byte == (frame_nr == 0 ? C_RR0 : C_RR1))
        {
```

```
            *state = C_RCV;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else if (byte == (frame_nr == 0 ? C_REJ1 : C_REJ0))
        {
            *state = RESEND;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case C_RCV:
        if (byte == (frame_nr == 0 ? BCC(A_TX, C_RR0) : BCC(A_TX, C_RR1)))
        {
            *state = BCC_OK;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case BCC_OK:
        if (byte == F_FLAG)
        {
            *state = STOP;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    default:
        break;
```

```
    }
}

void state_machine_writes(int *state, unsigned char byte, bool bcc2_checked,int frame_ns)
{
    switch (*state)
    {
    case START:
        if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = *state;
            break;
        }
    case FLAG_RCV:
        if (byte == F_FLAG)
        {
            *state = *state;
            break;
        }
        else if (byte == A_TX)
        {
            *state = A_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case A_RCV:
        if (byte == (frame_ns == 0 ? C_FRAME0 : C_FRAME1))
        {
            *state = C_RCV;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
```

```
            *state = START;
            break;
        }
    case C_RCV:
        if (byte == (frame_ns == 0 ? BCC(A_TX, C_FRAME0) : BCC(A_TX, C_FRAME1)))
        {
            *state = BCC_OK;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case BCC_OK:
        if (byte == ESC)
        {
            *state = DATA_STUFFED;
            break;
        }
        else if (byte != F_FLAG)
        {
            *state = *state;
            break;
        }
        else
        {
            *state = BCC2_CHECK;
            break;
        }
    case DATA_STUFFED:
        if (byte == ESC)
        {
            *state = *state;
            break;
        }
        else
        {
            *state = BCC_OK;
            break;
        }
    case BCC2_CHECK:
```

```
            if (bcc2_checked)
            {
                *state = STOP;
            }
            else
            {
                *state = START;
            }
        default:
            break;
        }
}

void state_machine_close(int *state, unsigned char byte)
{
    switch (*state)
    {
    case START:
        if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = *state;
            break;
        }
    case FLAG_RCV:
        if (byte == F_FLAG)
        {
            *state = *state;
            break;
        }
        else if (byte == A_TX)
        {
            *state = A_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case A_RCV:
        if (byte == C_DISC)
        {
```

```
            *state = C_RCV;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case C_RCV:
        if (byte == (A_TX ^ C_DISC))
        {
            *state = BCC_OK;
            break;
        }
        else if (byte == F_FLAG)
        {
            *state = FLAG_RCV;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    case BCC_OK:
        if (byte == F_FLAG)
        {
            *state = STOP;
            break;
        }
        else
        {
            *state = START;
            break;
        }
    default:
        break;
    }
}
```

11.1.3 application_layer.c

```
    // Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
#include "constants.h"

typedef struct
{
    unsigned char type;
    unsigned char file_name[UCHAR_MAX + 1];
    size_t    file_size ;
} ctrlPacket;

typedef struct
{
    int  sequence_number;
    size_t  payload_length;
    unsigned char packet[MAX_PAYLOAD_SIZE];
} dataPacket;

void progress_bar(double percentage) {
    int  smallest_div  = 2;
    int  num_bars = (int)percentage/smallest_div;
    printf("[");
    for(int  i  = 0; i  < num_bars - 1;i++) {
        printf("=");
    }
    printf(">");
    for(int  i  = 0; i  < (100/smallest_div) - num_bars; i++) {
        printf(".");
    }
    printf("]  ");
}

void formatScreen(bool connectionOpened, double percentage, bool tx)
{
    printf("Opening connection...\n");
    if  (connectionOpened)
        printf("Connection opened!\n");

    progress_bar(percentage);
    if  (tx)
```

```
        printf("%0.0f%% Sent\n", percentage);

    else
        printf("%0.0f%% Received\n", percentage);
}

size_t calculate_file_size (const char *filename)
{
    FILE *fp = fopen(filename, "r");
    fseek(fp, 0L, SEEK_END);
    size_t res = ftell(fp);
    fclose(fp);
    return res;
}

size_t build_data_packet(unsigned char *packet, dataPacket data_packet)
{

    unsigned char l1 = data_packet.payload_length / 256;
    unsigned char l2 = data_packet.payload_length % 256;

    packet[0] = DATA;
    packet[1] = data_packet.sequence_number;
    packet[2] = l1;
    packet[3] = l2;
    memcpy(packet + 4, data_packet.packet, data_packet.payload_length);

    return data_packet.payload_length + 4;
}

size_t build_control_packet(unsigned char *packet, ctrlPacket ctrl_packet_info)
{
    packet[0] = ctrl_packet_info.type;

    packet[1] = PACKET_CONTROL_FILESIZE;

    size_t filesize = ctrl_packet_info. file_size ;

    packet[2] = sizeof( filesize );
    packet[3] = filesize ;
    packet[4] = filesize >> 8;
    packet[5] = filesize >> 16;

    packet[6] = PACKET_CONTROL_FILENAME;

    size_t filename_length = strlen((const char *) ctrl_packet_info.file_name);
```

```
    packet[7] = filename_length;

    memcpy(packet + 8, ctrl_packet_info.file_name, filename_length + 1);

    return filename_length + 8;
}

void application_layer_tx_protocol (const char *filename)
{
    unsigned char initial_ctrl_packet [MAX_PAYLOAD_SIZE];

    ctrlPacket ctrl_packet_info = {
        .type = CTRL_START,
        . file_size = calculate_file_size (filename),
    };

    strcpy((char *) ctrl_packet_info .file_name, filename);

    size_t ctrl_packet_size = build_control_packet( initial_ctrl_packet , ctrl_packet_info );

    if ( llwrite ( initial_ctrl_packet , ctrl_packet_size ) != 0)
    {
        printf("Error writing control packet.\n");
        exit(1);
    }

    FILE *ptr = fopen(filename, "rb");

    dataPacket data_packet;
    int sequence_number = 0;
    unsigned char packet[MAX_PAYLOAD_SIZE + 4];

    size_t num_bytes_read;
    double num_bytes_written;

    while ((num_bytes_read = fread(data_packet.packet, sizeof(char), MAX_PAYLOAD_SIZE,
        ptr)) > 0)
    {
        data_packet.sequence_number = sequence_number;
        data_packet.payload_length = num_bytes_read;

        size_t packet_size = build_data_packet(packet, data_packet);

        size_t total_file_size = 0;
         total_file_size |= initial_ctrl_packet [3];
         total_file_size |= initial_ctrl_packet [4] << 8;
         total_file_size |= initial_ctrl_packet [5];
```

```
            if (llwrite(packet, packet_size) != 0)
            {
                printf("File corrupted. Exiting\n");
                exit(1);
            }

            num_bytes_written += num_bytes_read;

            system("clear");
            formatScreen(true, num_bytes_written / total_file_size * 100, true);
        }

        unsigned char final_ctrl_packet[MAX_PAYLOAD_SIZE];
        ctrl_packet_info.type = CTRL_FINISH;
        ctrl_packet_size = build_control_packet(final_ctrl_packet, ctrl_packet_info);

        if (llwrite(final_ctrl_packet, ctrl_packet_size) != 0)
        {
            printf("Error writing control packet.\n");
            exit(1);
        }
    }

    void application_layer_rx_protocol(const char *filename)
    {
        unsigned char initial_ctrl_packet[MAX_PAYLOAD_SIZE];
        llread(initial_ctrl_packet);

        if (initial_ctrl_packet[0] != CTRL_START)
        {
            printf("Initial control packet not found.\n");
            exit(1);
        }

        size_t total_file_size = 0;
        total_file_size |= initial_ctrl_packet[3];
        total_file_size |= initial_ctrl_packet[4] << 8;
        total_file_size |= initial_ctrl_packet[5];

        FILE *ptr = fopen(filename, "wb+");

        dataPacket data_packet;
        unsigned char packet[MAX_PAYLOAD_SIZE];

        double bytes_written = 0;
```

```
    while (TRUE)
    {
        llread(data_packet.packet);

        if (data_packet.packet[0] != DATA && data_packet.packet[0] == CTRL_FINISH)
            return;

        data_packet.payload_length = data_packet.packet[2] * 256 + data_packet.packet[3];

        memcpy(packet, data_packet.packet + 4, data_packet.payload_length);
        fwrite(packet, 1, data_packet.payload_length, ptr);
        bytes_written += data_packet.payload_length;

        system("clear");
        formatScreen(true, bytes_written / total_file_size * 100, false);
    }
}

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{

    LinkLayerRole r = strcmp(role, "tx") == 0 ? LlTx : LlRx;
    LinkLayer linkLayer;

    memset(linkLayer.serialPort, '\0', 50);
    strcpy(linkLayer.serialPort, serialPort);

    linkLayer.role = r;
    linkLayer.baudRate = baudRate;
    linkLayer.nRetransmissions = nTries;
    linkLayer.timeout = timeout;

    llopen(linkLayer);

    if (r == LlTx)
        application_layer_tx_protocol (filename);
    else
        application_layer_rx_protocol (filename);

    printf("Closing...\n");
    system("clear");
    llclose (1);
};
```

11.1.4 statistics.c

```
#include "statistics.h"

static struct timeval start;
static struct timeval start_frame;
static long received_bits = 0;
static long bad_frames;
static long good_frames;
static double sum_efficiency;

void stat_start_timer()
{
  gettimeofday(&start, NULL);
}

void stat_start_frame_timer()
{
  gettimeofday(&start_frame, NULL);
}

double stat_get_t_total()
{
  struct timeval finish_frame;
  gettimeofday(&finish_frame, NULL);

  return (finish_frame.tv_sec - start_frame.tv_sec) + (finish_frame.tv_usec -
      start_frame.tv_usec) / 1000000.0;
}

void stat_add_total_efficiency(double frame_efficiency)
{
  sum_efficiency = sum_efficiency + frame_efficiency;
}

double stat_get_time()
{
  struct timeval finish;
  gettimeofday(&finish, NULL);

  return (finish.tv_sec - start.tv_sec) + (finish.tv_usec - start.tv_usec) / 1000000.0;
}

void stat_set_bits_received(int bytes)
{
  received_bits += bytes * 8;
}

long stat_get_bits_received()
```

```
{
  return  received_bits ;
}

double  stat_get_bitrate (double time)
{
  return  received_bits  / time;
}

void stat_add_bad_frame()
{
  bad_frames++;
}

long stat_get_bad_frames()
{
  return bad_frames;
}

void stat_add_good_frame()
{
  good_frames++;
}

long stat_get_good_frames()
{
  return good_frames;
}

double  stat_get_fer ()
{
  return bad_frames / ((double)(bad_frames + good_frames));
}
```

11.2 Graphs

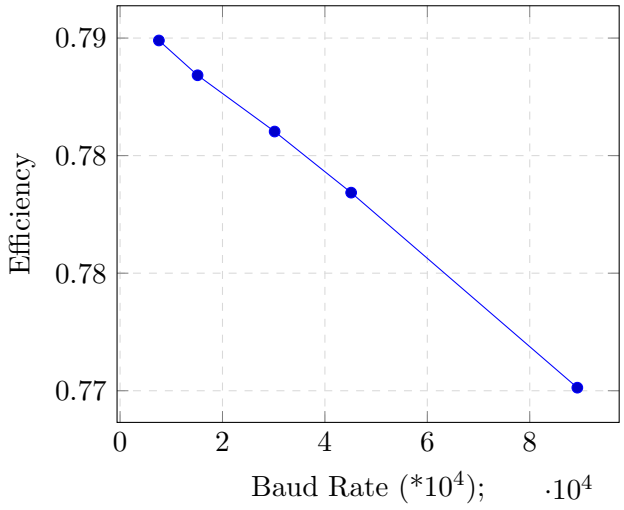

Figure 1: Relationship between Baud Rate and Efficiency.

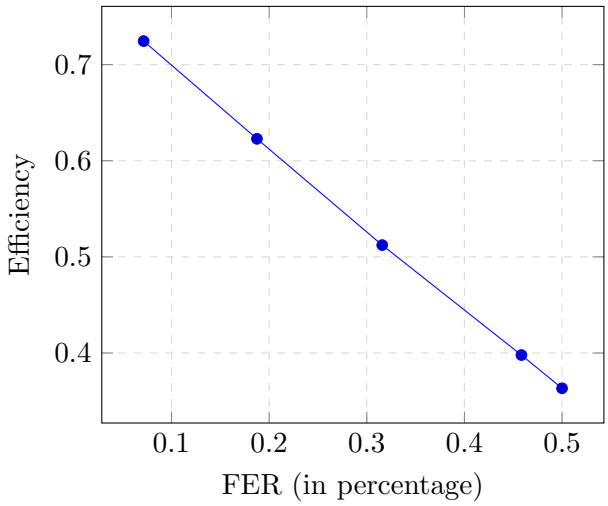

Figure 2: Relationship between FER and Efficiency.

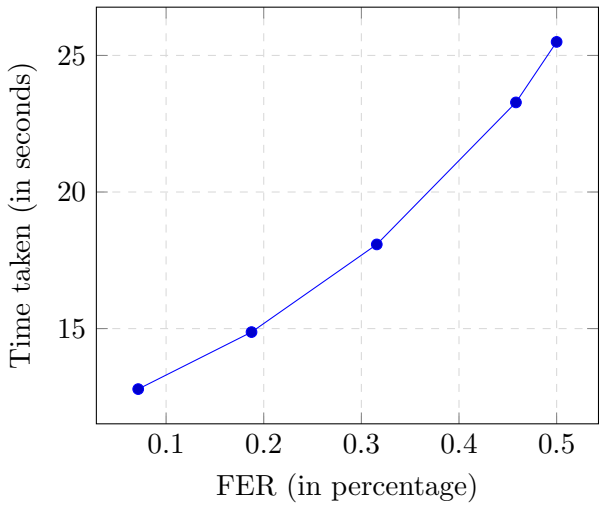

Figure 3: Relationship between FER and Time Taken.

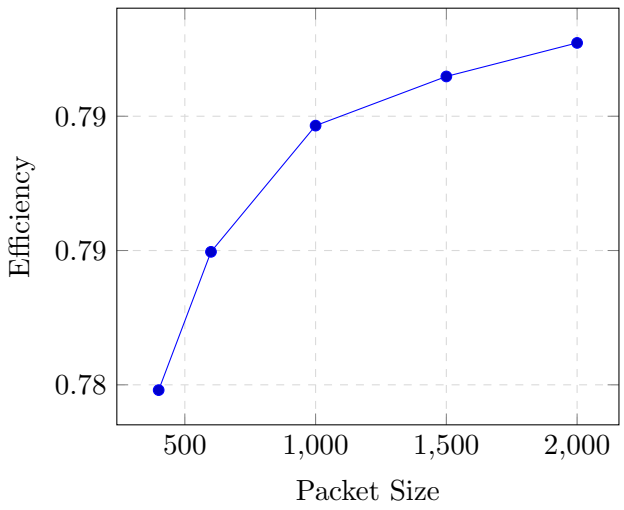

Figure 4: Relationship between Packet Size and Efficiency.

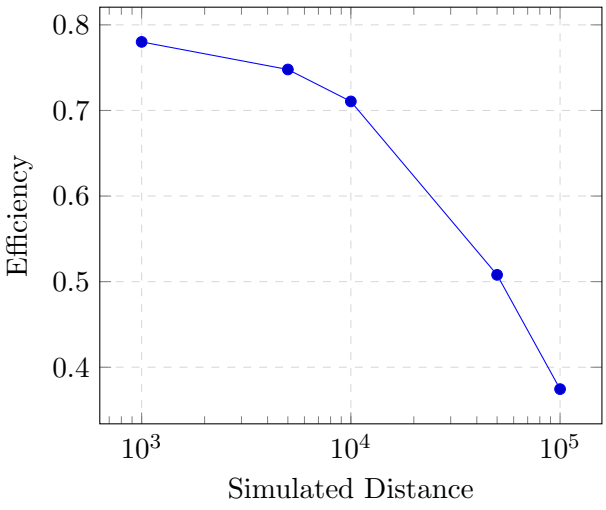

Figure 5: Relationship between Simulated Distance and Efficiency.