Data Link Protocol

Computer Networks
Bachelors in Informatics and Computing Engineering

3LEIC03_G6

Tiago Rodrigues up201907021@fe.up.pt
Mário Travassos up201905871@fe.up.pt

December 10, 2021

Summary

This report will cover the first work proposed for the Computer Networks Curricular Unit, with the objective of creating a small application that could transfer data through two computers asynchronously, through a serial port.

The application is capable of transferring files whilst maintaining their integrity, and detect errors in transmission, resolving them if possible.

Introduction

This report is the result of an examination of the practical component, which was the development of a data transfer protocol. A serial port was used to transfer the files in an asynchronous fashion.

The report is organized as follows:

1. Architecture - Functional blocks and interfaces
2. Code Structure - APIs, main code structures and their relation with the architecture
3. Main use cases - Identification and Call Stack Sequence
4. Data link Protocol - Main functional aspects and implementation strategy
5. Application Protocol - Main functional aspects and implementation strategy
6. Validation - Description of the tests conducted
7. Efficiency - Statistical characterization of efficiency
8. Conclusion - Summary of the above descriptions, reflection on the learning objectives

1 Architecture

The application consists of two main layers, one which interacts directly with the file to be sent, and prepares it for the transfer, and another one which is responsible for interfacing with the hardware. These are, respectively, the Application Layer and the Data-Link Layer. A detailed explanation is presented further below.

1.1 Application Layer

This layer is contained within the **rcom-ftp.c** file, and it encompasses everything related to interacting with the file to be sent. Opening, closing, reading its contents and preparing the data to be transfered, and assembling the file from data received are some of its responsibilities. This is the layer through which the user interacts with the application.

1.2 Data-Link Layer

This layer can be found in the **ll.c** file. It is responsible for ensuring a smooth data transmission over the hardware, including opening, closing, writing and reading from the serial port, with the help of the auxiliary functions present in **config.c**, **read.c**, **send.c**, **state.c** and **utils.c**.

2 Code Structure

The code is divided into seven source code files, separated by responsibility (reading from or writing to the serial port), and by layer. Also, each of them has a corresponding header file. Finally, defines are also organized by layer. Defines used in the data link and auxiliary functions of that module are stored in a dedicated header file, aptly named **datalink-defines.h** while defines used in the application layer are saved inside **rcom-ftp.h**.

2.1 Application - rcom-ftp.c

This module contains the entire application layer developed.

Main Functions

- main - Interacts with the user and passes the arguments given to the rest of the program.
- sendFile - Prepares and sends the file requested by the user.
- readFile - Receives data and assembles the file sent by the sender.

Main Data Structures

- fileData - Responsible for holding some of the file's metadata.

2.2 Config - config.c

This module contains the functions required for setting up the serial port for proper file transferring and receiving.

Main Functions

- set_config - Sets up the initial configuration for the serial port.
- reset_config - Restores the serial port to its initial state.

2.3 Link Layer - ll.c

This module contains the interface for the Link Layer of the protocol.

Main Functions

- llopen - Opens the serial port for frame transmission.
- llwrite - Writes a frame to the serial port.
- llread - Reads a frame from the serial port and checks its integrity.
- llclose - Closes the serial port.

2.4 Reading - read.c

This module contains the functions responsible for reading from the serial port.

Main Functions

- readSupervisionFrame - Reads a supervision frame and checks if the information was received correctly.
- readInformationMessage - Reads an information message and stores the data, including the BCC, in a buffer.

2.5 Writing - send.c

This module contains the functions responsible for writing to the serial port.

Main Functions

- writeSupervisionAndRetry - Attempts to write a supervision Frame in 3 attempts. If it succeeds, it stops.
- writeInformationAndRetry - Attempts to write an information Frame in 3 attempts. If successful, it stops.

2.6 State Management - state.c

This module contains the functions responsible for managing the state of the application.

Main Functions

- handle_state - Function responsible for managing the state of the application, according to the data received.

Main Data Structures

- state_t - Enumeration containing the possible states of the application.
- state_machine - Besides the state of the machine, holds some of the main pieces of information from each frame.

2.7 Utilities - utils.c

This module contains some auxiliary functions, used in other modules.

Main Functions

- stuff_data - This functions stuffs the data given.
- unstuff_frame - This functions unstuffs the frame given.

2.8 Constants - datalink-defines.h and rcom-ftp.h

These modules contain some of the more meaningful constants shared throughout the application.

3 Main use cases

The application should be first compiled with the provided Makefile, by simply running `make`. Cleanup may be performed by running `make clean`

The application is run in two different ways, depending on the role we are fulfilling. The basic syntax is as follows: `./rcom-ftp <role> <port> [file]`

Where:

- `role` is either `emitter` or `receiver`, depending on whether we want to transfer a file, or receive a file, respectively
- `port` is the number of the port to be used by that instance of the program
- `file` is the file to be sent. This parameter is only used when `role` is `emitter`

As an example, one can run the receiver as `Eu i./rcom-ftp receiver 10`, to receive a file on the port `/dev/ttyS10`, while the emitter will run `./rcom-ftp emitter 11 file.ext`, to send the file `file.ext` through the port `/dev/ttyS11`.

To run the application and transfer a file, both the receiver and the emitter must be called. The emitter will be trying to connect for a total of 9 tries, making a stop of 1 second between each attempt. If it can't succeed in connecting, it will halt.

If the connection was successful, the application layer will begin dividing the file into data packets, and sending them to the receiver through the **llwrite** function, whilst the receiver reads them with the **llread** function, and assembles them into a file. Finally, both of them should call **llclose** in order to close the channel of transmission.

4 The Data link Layer

This layer is responsible for interacting with the serial port, hence functionally behaving as a layer of abstraction, allowing the application to more easily make the connection required to exchange data. It uses several auxiliary functions, as a way to better structure the code, assigning to each function a single responsibility. The main functions are divided as follows:

llopen

The **llopen** function sets up the communication between the transmitter and receiver. It start by configuring the serial for for proper reading and writing, setting the appropriate flags and setting **VTIME** to 30 and **VMIN** to 0, which ensures the read function won't have to wait for a character to return.

After that, and according to the provided role, it will either send a **SET** command and await for a **UA**, if it is the emitter, or wait for a **SET** command and then send a **UA**, in the case of the transmitter. To do this, they take advantage of the `writeSupervisionAndRetry` and the `readSupervisionFrame` functions, which, respectively, handle writing to the serial port and reading from it, setting the appropriate state.

llwrite

As the name implies, the **llwrite** function writes a given packet of data to the serial port. To do this, it calls the `writeInformationAndRetry` function, which, through the use of `writeInformationFrame`, appends the frame header and trailer, and writes it to the serial port. It retries if the writing is unsuccessful, a total of 3 times, at which point it returns with an error. After successfully writing, it waits to receive a confirmation message, either accepting or rejecting the frame. If it accepts, then the frame is written and **llwrite** halts. Otherwise, if it was rejected, then it re-sends the same frame again, not increasing the attempts made (as writing was successful, but the message got rejected). Finally, if some error occurred, it tries to resend the message until it exhausts all attempts.

llread

The workings of **llread** are similar to those of **llwrite**. First, it tries to read the incoming information message. When successful, it then proceeds to unstuff the incoming frame and check if the data has been corrupted. If it hasn't, then it acknowledges the packet, sending back a **RR** frame, and if sent successfully, returns the size of the information read. When the data appears to be corrupted, it sends a **REJ** frame, telling the emitter to retry sending that information. Finally, if some error occurred, it returns with an error as well, ceasing transmission.

llclose

Finally, this function handles the cease of communication between the two computers. If the caller is the emitter, it will start of by sending a **DISC** frame, telling the receiver it wishes to stop communication. Then, it waits for a **DISC** response and sends a final **UA** frame before terminating. The receiver works the other way around, first waiting for a **DISC** frame to be read, then sending his **DISC**, and finally reading the **UA** before being able to shutdown. Both, in case there is a problem with writing, try again for a total of 9 attempts, before exiting with error.

5 The Application Layer

sendFile

This is the main function of the emitter. It starts by retrieving some metadata from the file, namely the size of the file and the name. This will be useful both to prepare the data, to make it easier to divide into equally sized packets. Afterwards, it generates a starting control packet that contains the metadata from the file, passing it to the receiver so it too knows what file it is receiving, and proceeds to send out the file's contents. Each packet sent can be configured to a desired size. The value we chose as default was a maximum allowed data chunk size of 1024 bytes. All packets (save for the last one, if less than that maximum size of data remains) will be that large. Finally, it re-sends the control packet, but with a flag changed meaning that it will be end of the transmission. At each step, in the event of a malfunction, it returns with an error.

receiveFile

On the other end, the **receiveFile** function is in charge of handling the receiving of files. Its way of working is much like the one on **sendFile**. Firstly, it reads the starting packet containing the metadata on the file, and creates a new file, with those parameters (although the file name will have "received-" prepended to it). Afterwards, it reads the packets containing information, one by one, and assembles the file after each successful packet receive. Finally, it reads the end Packet, which will tell the receiver to stop trying to read any longer. Once again, if at any point the receiver detects an error reading any packet, it will stop with an error.

generateDataPacket

This function creates the packets that will be sent over to the receiver. It starts by appending the correct packet header, and then copies the information over to packet buffer, and returns.

readDataPacket

This function is analogous to the **generateDataPacket** one, and its job is to read a data packet and check for its integrity after transmission. To do this, it starts by checking if the packet is in fact a data packet, by verifying the first bit. Then it makes sure that the data packet received is the packet we're expecting to receive, aborting if that is not the case. Finally, and if all goes well, it retrieves the information contained in the packet and saves it in a buffer.

6 Validation

In order to test the protocol, files of different sizes were sent, ranging from **test1.txt** (128 bytes) and **test10.pdf** (128662771 bytes). Also, the data packet size was varied to check for performance, with sizes going from 8 bytes to 1024 bytes. Finally, we also tested variations in the baudrate, with values going from 2896, all the way up to 11764736. Besides varying the values of certain aspects of the program, the transmission was tested with timeouts on the lab, and with introduction of random errors, by placing a piece of metal that would generate errors in the data transfer sometimes. Upon each test, the integrity of the generated file was verified by comparing the hashes of the original and the received file, using the **md5sum** hashing function.

7 Efficiency

All the tests results are available in **Annex II**

Variation in the File Size

As expected, an increase in the file size increased the execution time linearly, both for the execution time of the emitter and the receiver. This is expected and implies that the system functions well regardless of the input given, which is key in a data transfer protocol.

Variation in the Packet Size

Regarding the Data Packet Size, it is possible to see that with larger packets there is a performance gain to be had, as there is less overhead with adding and decoupling packet headers and frame headers and trailers. But also, for packets that are too large, the possibility of errors being generated and the need for re-transmission can become burdening, and slow down the rest of the program.

Variation in the Baudrate

For the values tested, no significant changes in the execution time of the program were recorded.

8 Conclusion

The task assigned consisted in implementing a data transfer protocol to communicate over a serial port. In this assignment, it was key to create distinct layers, and, more importantly, make sure they work independently of one another, not knowing the details of each other's implementation. This lead to a structure in which the application layer, the "highest level" section of the program, only knew what the data-link layer did, but not how it operated, and the data-link knew only that it would receive data and would have to transmit it. This effectively creates two distinct entities that in nothing influence each other, but work together for the objective of transmitting data.

The project was completed successfully, and it contributed a great deal for deepening our knowledge of how data is transferred in between devices, how eventual errors are handled, and how a file can be sent from one location to another location, possibly quite far away, seemingly by magic.

Annex I - Source Code

config.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

#include "config.h"
#include "datalink-defines.h"

static struct termios old_termio;

int set_config(int port) {

  char serial_port[15];
  sprintf(serial_port, "/dev/ttyS%d", port);

  int serial_port_fd = open(serial_port, O_RDWR | O_NOCTTY);

  if (serial_port_fd < 0) {
    perror(serial_port);
    exit(-1);
  }

  struct termios new_termio;

  if (tcgetattr(serial_port_fd, &old_termio) == -1) {
    perror("tcgetattr");
    exit(-1);
  }

  bzero(&new_termio, sizeof(new_termio));
  new_termio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
  new_termio.c_iflag = IGNPAR;
  new_termio.c_oflag = 0;
  new_termio.c_lflag = 0;
  new_termio.c_cc[VTIME] = 30;
  new_termio.c_cc[VMIN] = 0;
```

```
  tcflush(serial_port_fd, TCIOFLUSH);

  if (tcsetattr(serial_port_fd, TCSANOW, &new_termio) == -1) {
    perror("tcsetattr");
    exit(-1);
  }

  return serial_port_fd;
}

int reset_config(int fd) {

  if (tcsetattr(fd, TCSANOW, &old_termio) == -1) {
    perror("tcsetattr");
    exit(-1);
  }

  return close(fd);
}
```

config.h

```
#ifndef RCOM_CONFIG_H_
#define RCOM_CONFIG_H_

int set_config(int port);

int reset_config(int fd);

#endif // RCOM_CONFIG_H_
```

datalink-defines.h

```
#ifndef DEFINES_H_
#define DEFINES_H_

// Serial port
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */

// Roles
#define RECEIVER 0
#define TRANSMITTER 1
#define UNKNOWN_ROLE -1

#define BIT(shift) 1 << shift

// Frame Size
#define SU_FRAME_SIZE 5
#define I_FRAME_SIZE(s) 7 + s

// Message Sending
#define MAX_DATA_CHUNK_SIZE 1024
#define MAX_FRAME_SIZE MAX_DATA_CHUNK_SIZE * 2 + 7
#define NUM_TRIES 3
#define REJECTED -2

// Message Information
#define FLAG 0x7E
#define ESCAPE 0x7d
#define FLAG_ESCAPE 0x5E
#define ESCAPE_ESCAPE 0x5D
#define A_SEND_CMD 0x03
#define A_SEND_RSP 0x01
#define A_RECV_CMD 0x01
#define A_RECV_RSP 0x03
#define C_SET 0x03
#define C_DISC 0x0B
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
#define C_REJ1 0x81
#define C_S0 0x00
#define C_S1 0x40
#define C_UA 0x07
```

```
#define C_RR(n) (n << 7) | 0x05
#define C_REJ(n) (n << 7) | 0x01
#define C_S(n) (((n) % 2) << 6)
#define BCC1(a, c) a ^ c
#define C_INF(n) (n << 6)

#endif // DEFINES_H_
```

ll.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <termios.h>

#include "ll.h"
#include "datalink-defines.h"
#include "config.h"
#include "read.h"
#include "send.h"
#include "state.h"
#include "utils.h"

static bool role = -1;

int llopen(int port, bool role_p) {

  int ret = -1;
  int fd = -1;

  role = role_p;
  ret = set_config(port);

  if (ret == -1) {
    fprintf(stderr, "Error opening serial port\n");
    exit(-1);
  } else {
    fd = ret;
  }

  if (role == TRANSMITTER) {

    for (int i = 0; i < NUM_TRIES; i++) {

      ret = writeSupervisionAndRetry(fd, A_SEND_CMD, C_SET);

      if (ret != 0) {
        continue;
      }

      ret = readSupervisionFrame(fd);
```

```
      if (ret != SU_FRAME_SIZE || get_ctrl() != C_UA ||
          get_addr() != A_RECV_RSP) {
        continue;
      } else {
        return fd;
      }
    }

    return -1;

  } else if (role == RECEIVER) {
    int ret;
    for (int i = 0; i < NUM_TRIES; i++) {

      ret = readSupervisionFrame(fd);

      if (ret != SU_FRAME_SIZE || get_ctrl() != C_SET ||
          get_addr() != A_SEND_CMD) {
        continue;
      }

      ret = writeSupervisionAndRetry(fd, A_RECV_RSP, C_UA);

      if (ret != 0) {
        continue;
      } else {
        return fd;
      }
    }
    return -1;
  } else {

    return UNKNOWN_ROLE;
  }
}

int llread(int fd, unsigned char *buffer) {
  static int packet = 0;

  int ret = -1;
  unsigned char stuffed_msg[MAX_FRAME_SIZE];
  unsigned char unstuffed_msg[MAX_DATA_CHUNK_SIZE + 7];
  for (int i = 0; i < NUM_TRIES; i++) {
```

```
    ret = readInformationMessage(fd, stuffed_msg);

    if (ret == -1) {
      continue;
    }

    ret = unstuff_frame(stuffed_msg, ret, unstuffed_msg);

    size_t size = ret - 1;

    if (ret == -1) {
      continue;
    }

    unsigned char unstuffed_bcc2 = unstuffed_msg[size];
    unsigned char recv_data_bcc2 = build_BCC2(unstuffed_msg, size);

    if (unstuffed_bcc2 == recv_data_bcc2 && get_ctrl() == C_S(packet)) {
      packet++;

      ret = writeSupervisionAndRetry(fd, A_RECV_RSP, C_RR(packet % 2));

      if (ret != 0) {
        continue;
      } else {
        memcpy(buffer, unstuffed_msg, size);
        return size;
      }
    } else if (unstuffed_bcc2 == recv_data_bcc2) {
      ret = writeSupervisionAndRetry(fd, A_RECV_RSP, C_RR(packet % 2));
      tcflush(fd, TCIFLUSH);
      sleep(1);
      continue;
    } else {
      ret = writeSupervisionAndRetry(fd, A_RECV_RSP, C_REJ(packet));
      tcflush(fd, TCIFLUSH);
      sleep(1);
      continue;
    }
  }

  return -1;
}
```

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

  static int frame_nr = 0;

  int ret = -1;

  for (int i = 0; i < NUM_TRIES; i++) {
    ret = writeInformationAndRetry(fd, A_SEND_CMD, buffer, length, frame_nr);

    if (ret != -1) {
      frame_nr++;
      return ret;
    } else {
      sleep(1);
    }
  }

  return -1;
}

int llclose(int fd) {
  int ret;

  if (role == TRANSMITTER) {

    for (int i = 0; i < NUM_TRIES; i++) {
      ret = writeSupervisionAndRetry(fd, A_SEND_CMD, C_DISC);

      if (ret != 0) {
        continue;
      }

      ret = readSupervisionFrame(fd);

      if (ret != SU_FRAME_SIZE || get_ctrl() != C_DISC ||
          get_addr() != A_RECV_CMD) {
        continue;
      }

      ret = writeSupervisionAndRetry(fd, A_SEND_RSP, C_UA);
      if (ret != 0) {
        continue;
      } else {
```

```
        sleep(1);
        return reset_config(fd);
      }
    }

  } else if (role == RECEIVER) {

    for (int i = 0; i < NUM_TRIES; i++) {

      ret = readSupervisionFrame(fd);

      if (ret != SU_FRAME_SIZE || get_ctrl() != C_DISC ||
          get_addr() != A_SEND_CMD) {
        continue;
      }

      ret = writeSupervisionAndRetry(fd, A_RECV_CMD, C_DISC);

      if (ret != 0) {
        continue;
      }

      ret = readSupervisionFrame(fd);

      if (ret != SU_FRAME_SIZE || get_ctrl() != C_UA ||
          get_addr() != A_SEND_RSP) {
        continue;
      } else {
        return reset_config(fd);
      }
    }

  } else {
    return UNKNOWN_ROLE;
  }

  return -1;
}
```

ll.h

```c
#ifndef _RCOM_LL_H_
#define _RCOM_LL_H_

#include <stdbool.h>

/**
 * @brief Opens the serial port, and sets the role for the program
 *
 * @param port Number of the serial port to be opened
 * @param role_p Role of the program (either RECEIVER or TRANSMITTER)
 *
 * @returns The fd of the connection if successfull, -1 otherwise
 */
int llopen(int port, bool role_p);

/**
 * @brief Read information from the serial port, storing it in buffer
 *
 * @param fd File descriptor of the serial port
 * @param buffer Buffer to store the information read
 *
 * @return Size of the information read if successful, -1 otherwise
 */
int llread(int fd, unsigned char *buffer);

/**
 * @brief Writes to the serial port,
 *
 * @param fd File descriptor of the serial port
 * @param buffer Buffer of information to be written
 * @param length Length of the information to be written
 *
 * @return Size of the information if successful, -1 otherwise
 */
int llwrite(int fd, unsigned char *buffer, unsigned int length);

/**
 * @brief Closes the serial port, restoring the initial state
 *
 * @param fd File descriptor of the serial port
 *
 * @return 0 if successful, -1 otherwise
```

```
 */
int llclose(int fd);

#endif /* _RCOM_LL_H_ */
```

rcom-ftp.c

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>

#include "rcom-ftp.h"
#include "ll.h"

int retrieveFileData(struct fileData *fData, FILE *filePtr, char *fileName) {
  struct stat fileStat;
  if (fstat(fileno(filePtr), &fileStat) != 0) {
    perror("fstat");
    return 1;
  }

  fData->filePtr = filePtr;
  fData->fileName = fileName;
  fData->fileNameSize = (unsigned int)strlen(fileName);
  fData->fileSize = (unsigned int)fileStat.st_size;
  fData->fullPackets = fData->fileSize / MAX_DATA_CHUNK_SIZE;
  fData->leftover = fData->fileSize % MAX_DATA_CHUNK_SIZE;

  return 0;
}

int generateControlPacket(unsigned char *ctrlPacket, struct fileData *fData,
                          int start) {
  if (start != PACKET_CTRL_START && start != PACKET_CTRL_END) {
    return -1;
  }

  // Control field
  ctrlPacket[0] = start;
  // TLV1
  ctrlPacket[1] = CTRL_FILE_SIZE;
  ctrlPacket[2] = FILE_SIZE_BYTES;
  memcpy(ctrlPacket + 3, &(fData->fileSize), FILE_SIZE_BYTES);
  // TLV2
  ctrlPacket[3 + FILE_SIZE_BYTES] = CTRL_FILE_NAME;
  memcpy(ctrlPacket + 4 + FILE_SIZE_BYTES, &(fData->fileNameSize), 1);
  memcpy(ctrlPacket + 5 + FILE_SIZE_BYTES, fData->fileName,
```

```
        fData->fileNameSize);

  return 0;
}

void generateDataPacket(unsigned char *dataPacket, struct fileData *fData,
                        unsigned char *data, unsigned int dataSize, int seqN) {
  unsigned int l1 = dataSize % 256;
  unsigned int l2 = dataSize / 256;
  unsigned char dataPacketHeader[4] = {PACKET_DATA, seqN, l2, l1};
  memcpy(dataPacket, dataPacketHeader, 4);
  memcpy(dataPacket + DATA_PACKET_HEADER_SIZE, data, dataSize);
}

int sendFile(int portfd, char *fileName) {
  FILE *filePtr = fopen(fileName, "r");
  if (filePtr == NULL) {
    perror("fopen");
    return 1;
  }
  printf("\nOpened file.\n");

  struct fileData fData;
  if (retrieveFileData(&fData, filePtr, fileName) != 0) {
    printf("Error retrieving the file's data.\n");
    return 1;
  }

  printf("Successfully retrieved the file's data.\n");

  // Set up start Control Packet
  unsigned int ctrlPacketSize =
      CTRL_PACKET_SIZE(FILE_SIZE_BYTES, fData.fileNameSize);
  unsigned char ctrlPacket[ctrlPacketSize];
  if (generateControlPacket(ctrlPacket, &fData, PACKET_CTRL_START) != 0) {
    printf("Error creating the start control packet.\n");
    return 1;
  }
  printf("Successfully generated the Start Control Packet.\n");

  // Send start Control Packet
  if (llwrite(portfd, ctrlPacket, ctrlPacketSize) != ctrlPacketSize) {
    printf("Error sending start control Packet.\n");
    return -1;
```

```
}
printf("Successfully sent the Start Control Packet.\n");

// Iterate through file, create and send Data Packets
unsigned int packetsSent = 0;
unsigned int allPackets =
    (fData.leftover == 0) ? fData.fullPackets : fData.fullPackets + 1;

printf("\nSending packets...\n");
while (packetsSent < allPackets) {
  // read data
  unsigned int dataSize = (packetsSent < fData.fullPackets)
                              ? MAX_DATA_CHUNK_SIZE
                              : fData.leftover;
  unsigned char data[dataSize];
  for (size_t i = 0; i < dataSize; i++) {
    data[i] = fgetc(filePtr);
  }

  // Create Data Packet
  unsigned int dataPacketSize = DATA_PACKET_SIZE(dataSize);
  unsigned char dataPacket[dataPacketSize];
  generateDataPacket(dataPacket, &fData, data, dataSize, packetsSent % 256);

  if (llwrite(portfd, dataPacket, dataPacketSize) == -1) {
    fprintf(stderr, "Error writing packet %d\n", packetsSent);
    exit(-1);
  }

  packetsSent++;
  printf("Sent %u out of %u packets.\n", packetsSent, allPackets);
}
printf("\nAll %u packets sent.\n\n", allPackets);

// Set up End Control Packet
ctrlPacket[0] = PACKET_CTRL_END;
printf("Successfully generated the End Control Packet.\n");

// Send End Control Packet
if (llwrite(portfd, ctrlPacket, ctrlPacketSize) != ctrlPacketSize) {
  printf("Error sending end control Packet.\n");
  return -1;
}
printf("Successfully wrote the End Control Packet.\n");
```

```
  if (fclose(filePtr) != 0) {
    perror("fclose");
    return 1;
  }
  printf("Successfully closed the file.\n\n");

  return 0;
}

int readStartPacket(int portfd, struct fileData *fData) {
  unsigned char startPacket[MAX_CTRL_PACKET_SIZE];
  int size = -1;
  if ((size = llread(portfd, startPacket)) == -1) {
    fprintf(stderr, "llread failed at reading start packet\n");
    return 1;
  }

  if (startPacket[0] != PACKET_CTRL_START) {
    printf("Did not receive the start packet.\n");
    return 1;
  }

  if (startPacket[1] != 0) {
    printf("Did not receive a correct start packet.\n");
    return 1;
  }

  if (startPacket[2] != FILE_SIZE_BYTES) {
    printf("Did not receive a correct start packet.\n");
    return 1;
  }

  fData->filePtr = NULL;
  memcpy(&(fData->fileSize), startPacket + 3, FILE_SIZE_BYTES);

  if (startPacket[3 + FILE_SIZE_BYTES] != 1) {
    printf("Did not receive a correct start packet.\n");
    return 1;
  }

  fData->fileNameSize = startPacket[4 + FILE_SIZE_BYTES];
  fData->fileName = (char *)malloc(fData->fileNameSize * sizeof(char));
  memcpy(fData->fileName, startPacket + 3 + FILE_SIZE_BYTES + 2,
```

```
        fData->fileNameSize);

  fData->fullPackets = fData->fileSize / MAX_DATA_CHUNK_SIZE;
  fData->leftover = fData->fileSize % MAX_DATA_CHUNK_SIZE;

  return 0;
}

int readDataPacket(int portfd, unsigned char *data, unsigned int dataPacketSize,
                   unsigned int dataSize, unsigned int expPacketNum) {
  unsigned char *dataPacket = (unsigned char *)malloc(dataPacketSize);
  if (llread(portfd, dataPacket) == -1) {
    free(dataPacket);
    printf("llread failed at readDataPacket\n");
    return 1;
  }

  if (dataPacket[0] != PACKET_DATA) {
    free(dataPacket);
    printf("Did not receive a data packet.\n");
    return 1;
  }

  unsigned int n = dataPacket[1];

  //if we don't receive the correct packet
  if(n != expPacketNum){
    free(dataPacket);
    return 1;
  }

  unsigned int l2 = dataPacket[2];
  unsigned int l1 = dataPacket[3];
  unsigned int readSize = 256 * l2 + l1;
  for (size_t p = 0; p < readSize; p++) {
    data[p] = dataPacket[DATA_PACKET_HEADER_SIZE + p];
  }

  free(dataPacket);
  return 0;
}

int readEndPacket(int portfd) {
  unsigned char startPacket[MAX_CTRL_PACKET_SIZE];
```

```
  if (llread(portfd, startPacket) == -1) {
    return 1;
  }

  if (startPacket[0] != PACKET_CTRL_END) {
    printf("Did not receive the end packet.\n");
    return 1;
  }

  return 0;
}

int receiveFile(int portfd) {
  // read start packet
  struct fileData fData;

  if (readStartPacket(portfd, &fData) != 0) {
    printf("Error reading the start packet.\n");
    return 1;
  }
  printf("\nSuccessfully read the start packet and retrieved the file's data.\n");

  // prepare modified fileName
  char newName[9 + fData.fileNameSize];
  sprintf(newName, "%s%s", "received-", fData.fileName);

  // create file
  FILE *fp;
  if (fData.fileNameSize + 9 <= MAX_FILENAME_SIZE) {
    fp = fopen(newName, "w");
  } else { // create file with start packet values if by modifying the name we
           // went above the max filename size
    fp = fopen(fData.fileName, "w");
  }
  printf("Created file.\n");

  printf("\nReceiving packets...\n");
  unsigned int packetsRecvd = 0;
  unsigned int allPackets =
      (fData.leftover == 0) ? fData.fullPackets : fData.fullPackets + 1;
  while (true) {
    unsigned int dataSize = (packetsRecvd < fData.fullPackets)
                                ? MAX_DATA_CHUNK_SIZE
                                : fData.leftover;
```

```
    unsigned int dataPacketSize = DATA_PACKET_SIZE(dataSize);

    unsigned char data[dataSize];

    unsigned int expPacketNum = packetsRecvd % 256;

    if (readDataPacket(portfd, data, dataPacketSize, dataSize, expPacketNum) != 0) {
      printf("Error receiving data packet number %u.\n", packetsRecvd);
      return 1;
    }

    // write data to file
    if (fwrite(data, 1, dataSize, fp) != dataSize) {
      printf("Error writing data packet number %u.\n", packetsRecvd);
      return 1;
    }

    // loop control
    packetsRecvd++;
    printf("Received %u out of %u packets.\n", packetsRecvd, allPackets);
    // break if we have read all packets
    if (packetsRecvd == allPackets)
      break;
  }
  printf("\nAll %u packets received.\n\n", allPackets);

  if (readEndPacket(portfd) != 0) {
    printf("Error reading the end packet.\n");
    return 1;
  }
  printf("Successfully read the End Control Packet.\n");

  // close
  if (fclose(fp) != 0) {
    printf("Error closing the file.\n");
    return 1;
  }
  printf("Successfully closed the file.\n\n");

  return 0;
}

int main(int argc, char *argv[]) {
```

```
if (!(argc == 3 && strcmp(argv[1], "receiver") == 0) &&
    !(argc == 4 && strcmp(argv[1], "emitter") == 0)) {
  fprintf(stderr,
          "Usage:\t./rcom-ftp Role SerialPortNum [File]\n\tex: \t./rcom-ftp "
          "emitter 11 pinguim.gif\n\t\t./rcom-ftp receiver 10\n");
  exit(-1);
}

bool role = -1;
int port = -1;
if (strcmp(argv[1], "emitter") == 0) {
  role = TRANSMITTER;
} else if (strcmp(argv[1], "receiver") == 0) {
  role = RECEIVER;
}

if (isdigit(argv[2][0]) && atoi(argv[2]) >= 0) {
  port = atoi(argv[2]);
} else {
  fprintf(stderr, "Invalid port specified\n");
  exit(-1);
}
int fd = -1;
if ((fd = llopen(port, role)) == -1) {
  fprintf(stderr, "Error opening port \n");
  exit(-1);
}

if (role == TRANSMITTER) {
  if (sendFile(fd, argv[3]) != 0) {
    return 1;
  }
} else if (role == RECEIVER) {
  if (receiveFile(fd) != 0) {
    return 1;
  }
}

if (llclose(fd) == -1) {
  fprintf(stderr, "Error closing port \n");
  exit(-1);
}

return 0;
```

```
}
```

rcom-ftp.h

```c
#ifndef _RCOM_FTP_H_
#define _RCOM_FTP_H_

#include <stdbool.h>

// Roles
#define RECEIVER false
#define TRANSMITTER true
#define UNKNOWN_ROLE -1

// Packet defines
#define MAX_DATA_CHUNK_SIZE 1024
#define DATA_PACKET_HEADER_SIZE 4
#define DATA_PACKET_SIZE(dSize) dSize + DATA_PACKET_HEADER_SIZE
#define CTRL_PACKET_SIZE(l1, l2) 5 + l1 + l2
#define FILE_SIZE_BYTES 4     // 4 bytes handles file size up to around 4gb
#define MAX_FILENAME_SIZE 255 // 255 bytes in most popular file systems
#define MAX_CTRL_PACKET_SIZE 5 + FILE_SIZE_BYTES + MAX_FILENAME_SIZE

#define PACKET_DATA 1
#define PACKET_CTRL_START 2
#define PACKET_CTRL_END 3
#define CTRL_FILE_SIZE 0
#define CTRL_FILE_NAME 1

struct fileData {
  FILE *filePtr;             // Pointer to the file
  char *fileName;            // Name of the file
  unsigned int fileNameSize; // Size of the file name in bytes
  unsigned int fileSize;     // Size of the file in bytes
  unsigned int fullPackets;  // Number of full packets we can send
  unsigned int leftover;     // Size of the last non-full packet in bytes
};

/**
 * @brief Retrieves file data
 *
 * @param fData         Struct holding the file's data
 * @param filePTR       Pointer to the file
 * @param fileName      Name of the file
 * @return 0 on success, -1 otherwise
 */
```

```
int retrieveFileData(struct fileData *fData, FILE *filePtr, char *fileName);

/**
 * @brief Generates a control packet from a file's data
 *
 * @param ctrlPacket    Pointer to the packet where we're storing data
 * @param fData         Struct holding the file's data
 * @param start         Which Control Packet to generate
 * @return 0 on success, -1 otherwise
 */
int generateControlPacket(unsigned char *ctrlPacket, struct fileData *fData,
                          int start);

/**
 * @brief Generates a data packet from a file's content
 *
 * @param dataPacket    Pointer to the packet where we're storing data
 * @param fData         Misc filedata
 * @param data          Data we're storing in the packet
 * @param dataSize      Size of the data in the packet
 * @param seqN          Sequence number
 */
void generateDataPacket(unsigned char *dataPacket, struct fileData *fData,
                        unsigned char *data, unsigned int dataSize, int seqN);

/**
 * @brief Creates packets from file and sends them to the receiver
 *
 * @param portfd    File Descriptor of the serial port
 * @param fileName  Name of the file to send
 */
int sendFile(int portfd, char *fileName);

/**
 * @brief Receives the start packet from the transmitter and stores the file
 * information in fileData Struct
 *
 * @param portfd    File Descriptor of the serial port
 * @param fData     Pointer to the struct which will hold the file's data
 * @return 0 on success, 1 otherwise
 */
int readStartPacket(int portfd, struct fileData *fData);

/**
```

```
 * @brief Receives packets from the transmitter and decodes them
 *
 * @param portfd        File Descriptor of the serial port
 * @param data          Buffer where we'll store the data to be written to the
 * file
 * @param dataSize      Size of the data we're reading
 * @param expPacketNum  Number of the packet we're expecting to receive
 * @return 0 on success, 1 on error, 2 upon receiving a repeated packet
 */
int readDataPacket(int portfd, unsigned char *data, unsigned int dataPacketSize, unsigned
                   unsigned int expPacketNum);

/**
 * @brief Receives packets from the transmitter and decodes them
 *
 * @param portfd    File Descriptor of the serial port
 */
int readEndPacket(int portfd);

/**
 * @brief Receives packets from the transmitter and decodes them
 *
 * @param portfd    File Descriptor of the serial port
 */
int receiveFile(int portfd);

// REMOVE LATER!!!
int gimmeStartPacket(struct fileData *fData);
int gimmeDataPacket(unsigned char *data, unsigned int dataSize,
                    unsigned int expPacketNum);

#endif /* _RCOM_FTP_H_ */
```

read.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include "read.h"
#include "datalink-defines.h"
#include "state.h"

int readSupervisionFrame(int fd) {

  unsigned char buf;

  int ret = -1;

  set_state(START);
  time_t start_time = time(NULL), end_time = time(NULL);

  while (difftime(end_time, start_time) < 3 && get_state() != STOP) {

    ret = read(fd, &buf, 1);

    if (ret == -1) {
      perror("read");
      exit(-1);
    }
    if (ret == 0) {
      end_time = time(NULL);
    } else {
      end_time = time(&start_time);
    }

    handleState(buf);
  }

  return difftime(end_time, start_time) < 3 ? SU_FRAME_SIZE : -1;
}

int readInformationFrameResponse(int fd) {

  int ret = readSupervisionFrame(fd);
```

```
  if (ret != SU_FRAME_SIZE) {
    return -1;
  }

  unsigned char ctrl = get_ctrl();

  if (ctrl == C_RR0 || ctrl == C_RR1) {
    return (ctrl >> 7);
  }

  if (ctrl == C_REJ0 || ctrl == C_REJ1) {
    return REJECTED;
  }

  else {
    return -1;
  }
}

int readInformationMessage(int fd, unsigned char *stuffed_msg) {
  unsigned char byte;

  int idx = 0;
  int ret = -1;

  set_state(START);
  time_t start_time = time(NULL), end_time = time(NULL);
  while (difftime(end_time, start_time) < 3 && get_state() != STOP) {

    ret = read(fd, &byte, 1);

    if (ret == -1) {
      perror("read");
      exit(-1);
    }
    if (ret == 0) {
      end_time = time(NULL);
    } else {
      end_time = time(&start_time);
    }

    if (get_state() == DATA_RCV) {
      stuffed_msg[idx++] = byte;
    }
```

```
    handleState(byte);
  }

  return difftime(end_time, start_time) < 3 ? idx - 1 : -1;
}
```

read.h

```c
#ifndef _RCOM_READ_H_
#define _RCOM_READ_H_

/**
 * @brief Reads a supervision Frame, altering the state based on the information
 * received. Times out after 3 seconds of not receiving anything..
 *
 * @param fd File descriptor of the serial port
 *
 * @return The size of a Control frame is succeeded, -1 otherwise
 */
int readSupervisionFrame(int fd);

/**
 * @brief Reads a response to an Information Frame,
 *
 * @param fd File Descriptor of the serial port
 *
 * @return If the I Frame was accepted, returns the RR bit. If it was rejected,
 * it returns REJECTED. otherwise, it returns -1.
 */
int readInformationFrameResponse(int fd);

/**
 * @brief Reads an Information Frame, altering the stated accordingly and saving
 * the information part of the packet
 *
 * @param fd File Descriptor of the serial port
 * @param stuffed_msg Buffer to store the data part of the frame received
 *
 * @return size of the stuffed message (-1 as to not count for the FLAG, which
 * is also included), or -1 otherwise
 */
int readInformationMessage(int fd, unsigned char *stuffed_msg);

#endif /* _RCOM_READ_H_ */
```

send.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "send.h"
#include "read.h"
#include "datalink-defines.h"
#include "utils.h"

int writeInformationFrame(int fd, unsigned char addr, unsigned char *info_ptr,
                          size_t info_size, int msg_nr) {

  if (info_size <= 0)
    return -1;

  unsigned char stuffed_info[info_size * 2];
  int stuffed_size = stuff_data(info_ptr, info_size, stuffed_info);

  int ret = -1;
  unsigned char frame[I_FRAME_SIZE(stuffed_size)];

  frame[0] = FLAG;
  frame[1] = addr;
  frame[2] = C_S(msg_nr);
  frame[3] = BCC1(addr, C_S(msg_nr));

  memcpy(frame + 4, stuffed_info, stuffed_size);

  unsigned char original_bcc2 = build_BCC2(info_ptr, info_size);
  unsigned short stuffed_bcc2;

  if (original_bcc2 == FLAG) {
    stuffed_bcc2 = (ESCAPE << 8) | FLAG_ESCAPE;
  } else if (original_bcc2 == ESCAPE) {
    stuffed_bcc2 = (ESCAPE << 8) | ESCAPE_ESCAPE;
  } else {
    frame[stuffed_size + 4] = original_bcc2;
    frame[stuffed_size + 5] = FLAG;

    ret = write(fd, frame, I_FRAME_SIZE(stuffed_size) - 1);

    return ret == (I_FRAME_SIZE(stuffed_size) - 1) ? 0 : -1;
```

```
  }

  frame[stuffed_size + 4] = ((stuffed_bcc2 & 0xFF00) >> 8);
  frame[stuffed_size + 5] = (stuffed_bcc2 & 0x00FF);
  frame[stuffed_size + 6] = FLAG;

  ret = write(fd, frame, I_FRAME_SIZE(stuffed_size));

  return ret == I_FRAME_SIZE(stuffed_size) ? 0 : -1;
}

int writeInformationAndRetry(int fd, unsigned char addr,
                             unsigned char *info_ptr, size_t info_size,
                             int msg_nr) {

  int current_attempt = 0;
  int ret = -1;
  do {
    current_attempt++;

    ret = writeInformationFrame(fd, addr, info_ptr, info_size, msg_nr);

    if (ret != 0) {
      continue;
    }

    ret = readInformationFrameResponse(fd);

    if (ret == -1) {
      continue;
    } else if (ret == REJECTED) {
      current_attempt = 0;
      continue;
    } else {
      if (ret == ((msg_nr + 1) % 2)) {
        return info_size;
      } else {
        continue;
      }
    }

  } while (current_attempt < NUM_TRIES);

  return -1;
```

```
}

int writeSupervisionFrame(int fd, unsigned char msg_addr,
                          unsigned char msg_ctrl) {
  unsigned char buf[SU_FRAME_SIZE];

  buf[0] = FLAG;
  buf[1] = msg_addr;
  buf[2] = msg_ctrl;
  buf[3] = BCC1(msg_addr, msg_ctrl);
  buf[4] = FLAG;

  return write(fd, &buf, SU_FRAME_SIZE);
}

int writeSupervisionAndRetry(int fd, unsigned char msg_addr,
                             unsigned char msg_ctrl) {
  int ret = 0;

  for (int i = 0; i < NUM_TRIES; i++) {
    ret = writeSupervisionFrame(fd, msg_addr, msg_ctrl);
    if (ret != SU_FRAME_SIZE) {
      sleep(1);
    } else {
      return 0;
    }
  }

  return -1;
}
```

send.h

```
#ifndef _RCOM_SEND_H_
#define _RCOM_SEND_H_

#include <stddef.h>

/**
 * @brief Writes an Information Frame to the Serial Port
 *
 * @param fd File Descriptor of the serial port
 * @param addr Address of the sender of the frame
 * @param info_ptr Information to be written to the serial port
 * @param info_size Size of the information to be sent
 * @param msg_nr Number of the packet that will be sent
 *
 * @return 0 if the entire frame was written, -1 otherwise
 */
int writeInformationFrame(int fd, unsigned char addr, unsigned char *info_ptr,
                          size_t info_size, int msg_nr);

/**
 * @brief Writes an Information Frame to the Serial Port, retrying 3 times if
 * failed
 *
 * @param fd File Descriptor of the serial port
 * @param addr Address of the sender of the frame
 * @param info_ptr Information to be written to the serial port
 * @param info_size Size of the information to be sent
 * @param msg_nr Number of the packet that will be sent
 *
 * @return The size of the packet if the entire frame was written and accepted,
 * -1 otherwise
 */
int writeInformationAndRetry(int fd, unsigned char addr,
                             unsigned char *info_ptr, size_t info_size,
                             int msg_nr);
/**
 * @brief Writes a Supervision Frame to the Serial Port
 *
 * @param fd File Descriptor of the serial port
 * @param msg_addr Address of the sender of the frame
 * @param msg_ctrl Control message to be sent
 *
```

```
 * @return The size of the frame if the entire frame was written,
 * -1 otherwise
 */
int writeSupervisionFrame(int fd, unsigned char msg_addr,
                          unsigned char msg_ctrl);

/**
 * @brief Writes a Supervision Frame to the Serial Port, and retries if failing
 * to write everything
 *
 * @param fd File Descriptor of the serial port
 * @param msg_addr Address of the sender of the frame
 * @param msg_ctrl Control message to be sent
 *
 * @return 0 if successfull, -1 otherwise
 */
int writeSupervisionAndRetry(int fd, unsigned char msg_addr,
                             unsigned char msg_ctrl);

#endif /* _RCOM_SEND_H_ */
```

state.c

```c
#include <stdio.h>

#include "state.h"
#include "datalink-defines.h"

static state_machine state;

state_t get_state() { return state.current_st; }

unsigned char get_addr() { return state.addr; }

unsigned char get_ctrl() { return state.ctrl; }

void set_state(state_t st) { state.current_st = st; }

void set_addr(unsigned char addr) { state.addr = addr; }

void set_ctrl(unsigned char ctrl) { state.ctrl = ctrl; }

static void handle_start(unsigned char byte) {
  switch (byte) {
  case FLAG:
    set_state(FLAG_RCV);
    break;
  default:
    break;
  }
}

static void handle_flag_rcv(unsigned char byte) {
  switch (byte) {
  case A_SEND_CMD:
  case A_RECV_CMD:
    set_state(A_RCV);
    set_addr(byte);
    break;
  case FLAG:
    break;
  default:
    set_state(START);
    break;
  }
```

```
}

static void handle_a_rcv(unsigned char byte) {
  switch (byte) {
  case C_SET:
  case C_UA:
  case C_DISC:
  case C_RR0:
  case C_RR1:
  case C_REJ0:
  case C_REJ1:
    set_state(C_RCV);
    set_ctrl(byte);
    break;
  case C_S0:
  case C_S1:
    set_state(I_MSG);
    set_ctrl(byte);
    break;
  case FLAG:
    set_state(FLAG_RCV);
    break;
  default:
    set_state(START);
    break;
  }
}

static void handle_c_recv(unsigned char byte) {
  switch (byte) {
  case FLAG:
    set_state(FLAG_RCV);
    break;
  default:
    if (byte == (BCC1(get_addr(), get_ctrl()))) {
      set_state(STOP);
    } else {
      set_state(START);
    }
    break;
  }
}

static void handle_bcc_ok(unsigned char byte) {
```

```
  switch (byte) {
  case FLAG:
    set_state(STOP);
    break;
  default:
    set_state(START);
    break;
  }
}

static void handle_i_msg(unsigned char byte) {
  switch (byte) {
  case FLAG:
    set_state(FLAG_RCV);
    break;
  default:
    if (byte == (BCC1(get_addr(), get_ctrl()))) {
      set_state(DATA_RCV);
    } else {
      set_state(START);
    }
    break;
  }
}

static void handle_data_rcv(unsigned char byte) {
  switch (byte) {
  case FLAG:
    set_state(STOP);
    break;
  default:
    break;
  }
}

void handleState(unsigned char byte) {

  switch (get_state()) {
  case START:
    handle_start(byte);
    break;
  case FLAG_RCV:
    handle_flag_rcv(byte);
    break;
```

```
  case A_RCV:
    handle_a_rcv(byte);
    break;
  case C_RCV:
    handle_c_recv(byte);
    break;
  case BCC_OK:
    handle_bcc_ok(byte);
    break;
  case I_MSG:
    handle_i_msg(byte);
    break;
  case DATA_RCV:
    handle_data_rcv(byte);
    break;
  case STOP:
    break;
  }
}
```

state.h

```c
#ifndef STATE_H_
#define STATE_H_

typedef enum {
  START,    // Start of the input
  FLAG_RCV, // Received a flag
  A_RCV,    // Received Sender or Receiver Address
  C_RCV,    // Received ctrl byte that is not from an I frame
  I_MSG,    // Switch state for I_MSG
  BCC_OK,   // Receives the BCC for a supervision message
  DATA_RCV, // Receiving data, "ignore" these bytes for now
  STOP      // Success state
} state_t;

typedef struct {
  state_t current_st;
  unsigned char addr;
  unsigned char ctrl;
} state_machine;

/**
 * @brief Handles the state of the program, according to the input given
 *
 * @param byte Byte received from serial port
 */
void handleState(unsigned char byte);

/**
 * @brief Retrieves the current state of the program
 *
 * @return The current state of the program
 */
state_t get_state();

/**
 * @brief Sets the current state of the program
 *
 * @param state New state of the program
 */
void set_state(state_t state);

/**
```

```
 * @brief Retrieves the address of the last message sent
 *
 * @return The address of the last message sent
 */
unsigned char get_addr();

/**
 * @brief Sets the current address of the program
 *
 * @param addr New address of the program
 */
void set_addr(unsigned char addr);

/**
 * @brief Retrieves the current state of the program
 *
 * @return The current state of the program
 */
unsigned char get_ctrl();

/**
 * @brief Sets the current control message of the program
 *
 * @param ctrl New control of the program
 */
void set_ctrl(unsigned char ctrl);

#endif // STATE_H_
```

utils.c

```c
#include <stdio.h>

#include "datalink-defines.h"
#include "utils.h"

unsigned char build_BCC2(unsigned char *data, size_t size) {
  unsigned char bcc2 = data[0];

  for (size_t p = 1; p < size; p++) {
    bcc2 ^= data[p];
  }

  return bcc2;
}

int unstuff_frame(unsigned char *stuffed_msg, size_t size,
                  unsigned char *unstuffed_msg) {

  int stuffed_idx = 0, unstuffed_idx = 0;

  for (; stuffed_idx < size; stuffed_idx++) {

    if (stuffed_msg[stuffed_idx] == ESCAPE) {
      stuffed_idx++;

      if (stuffed_msg[stuffed_idx] == ESCAPE_ESCAPE) {
        unstuffed_msg[unstuffed_idx++] = ESCAPE;
      } else if (stuffed_msg[stuffed_idx] == FLAG_ESCAPE) {
        unstuffed_msg[unstuffed_idx++] = FLAG;
      } else {
        return -1;
      }
    } else {
      unstuffed_msg[unstuffed_idx++] = stuffed_msg[stuffed_idx];
    }
  }

  return unstuffed_idx;
}

int stuff_data(unsigned char *data, size_t data_size,
               unsigned char *stuffed_data) {
```

```
  int data_idx = 0, stuffed_idx = 0;

  for (; data_idx < data_size; data_idx++) {
    if (data[data_idx] == FLAG) {
      stuffed_data[stuffed_idx++] = ESCAPE;
      stuffed_data[stuffed_idx++] = FLAG_ESCAPE;
    } else if (data[data_idx] == ESCAPE) {
      stuffed_data[stuffed_idx++] = ESCAPE;
      stuffed_data[stuffed_idx++] = ESCAPE_ESCAPE;
    } else {
      stuffed_data[stuffed_idx++] = data[data_idx];
    }
  }

  return stuffed_idx;
}
```

utils.h

```c
#ifndef RCOM_UTILS_H_
#define RCOM_UTILS_H_

#include <stddef.h>

/**
 * @brief builds the BCC2 block, based on the original input data
 *
 * @param data Data to be verified
 * @param Size of the input data
 *
 * @return The BCC of the data given
 */
unsigned char build_BCC2(unsigned char *data, size_t size);

/**
 * @brief Unstuffs the information received
 *
 * @param stuffed_msg Messaged to be unstuffed
 * @param size Size of the message to be unstuffed
 * @param unstuffed_msg Buffer to hold the actual message
 *
 * @return Size of the unstuffed information if successfull, -1 otherwise
 */
int unstuff_frame(unsigned char *stuffed_msg, size_t size,
                  unsigned char *unstuffed_msg);

/**
 * @brief Stuffs the information received
 *
 * @param data Messaged to be stuffed
 * @param data_size Size of the message to be unstuffed
 * @param stuffed_data Buffer to hold the stuffed message
 *
 * @return Size of the stuffed information if successfull, -1 otherwise
 */
int stuff_data(unsigned char *data, size_t data_size,
               unsigned char *stuffed_data);

#endif // RCOM_UTILS_H_
```

Annex II - Tests

Variation in the File Size

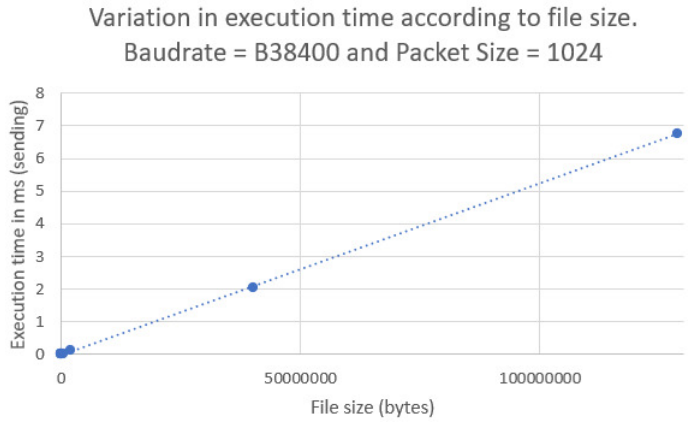

Figure 1: Execution time of the emitter according to File Size

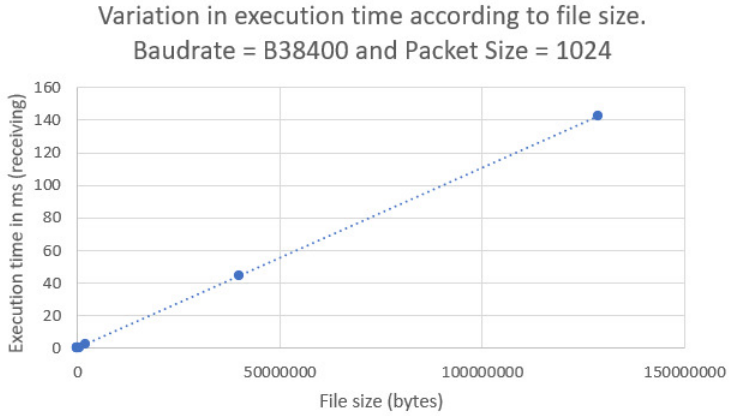

Figure 2: Execution time of the receiver according to File Size

Variation in the Packet Size

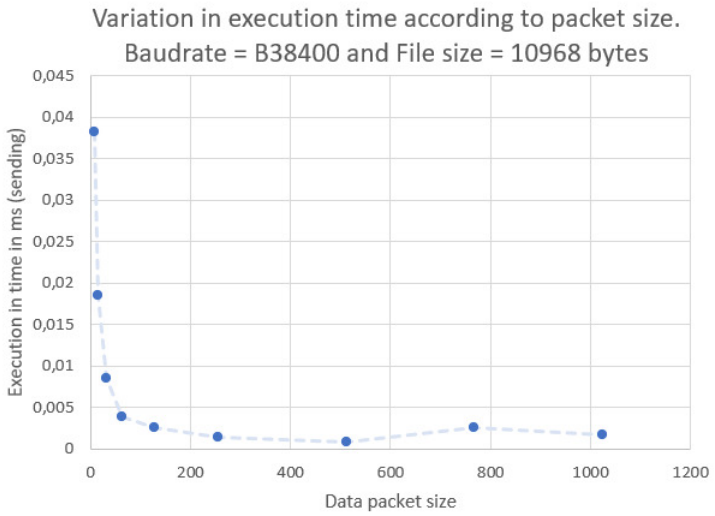

Figure 3: Execution time of the emitter according to Packet Size

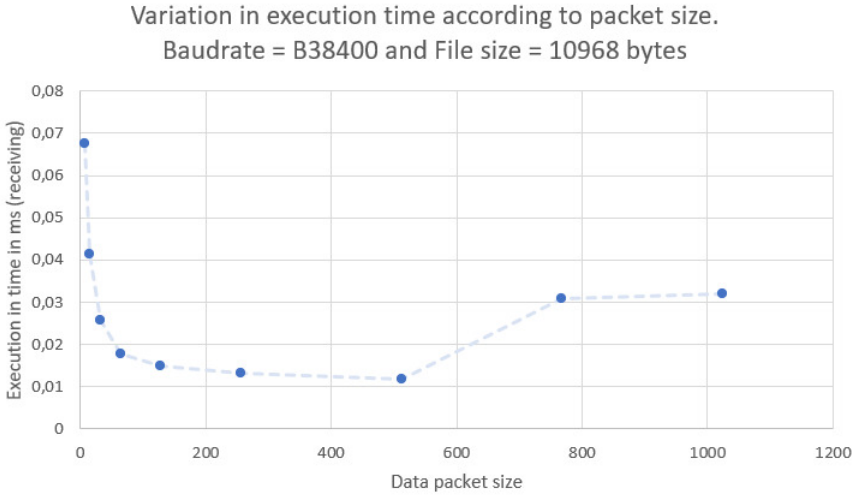

Figure 4: Execution time of the receiver according to Packet Size

Variation in the Baudrate

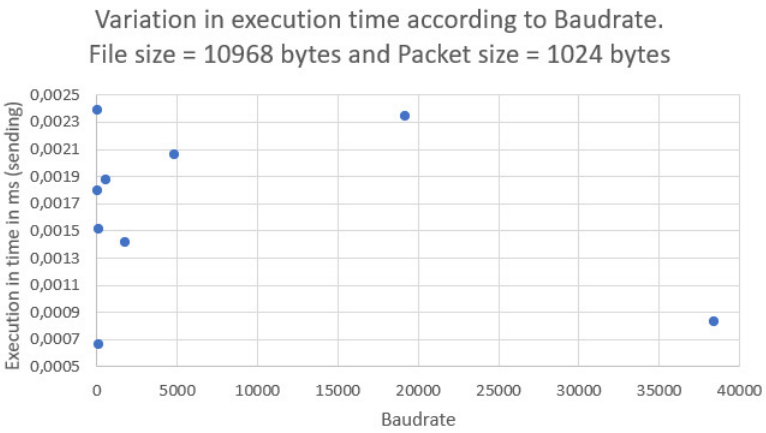

Figure 5: Execution time of the emitter according to Baudrate

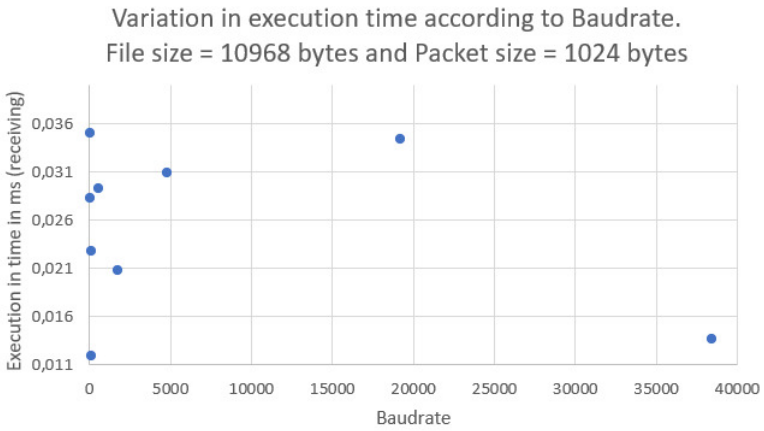

Figure 6: Execution time of the receiver according to Baudrate