First Laboratory Project Report

Student:
Bernardo Jose Ponce Figueiredo de Brito — up202301442
Vlad Ryan Plavosin — up202301426
Department of Informatics Engineering
University of Porto

October 24, 2023

Contents

1 Introduction

In the ever-evolving landscape of computer networking, the ability to securely and seamlessly transfer data between devices stands as a fundamental cornerstone. The foundation of this capability often hinges on innovative applications that harness the power of technology to facilitate these exchanges. In this report, we delve into the journey of conceiving, developing, and executing a file transfer application—a quintessential project within the realm of Computer Networks.

The central objective of this project was to engineer an application capable of transmitting files asynchronously between two computers. While the premise may sound straightforward, the real-world implications of such an endeavor are far-reaching. By employing a serial port as the conduit, the application grapples with the intricacies of data transmission, network protocols, and the management of potential errors.

Our exploration takes us through the intricacies of this application's architecture, highlighting its capabilities and the underlying principles that enable its functionality. We delve into its capacity for asynchronous data transfer, the art of communication via a serial

port, and, crucially, its prowess in error handling and data recovery—two facets pivotal to ensuring the unswerving reliability of data exchange.

This report serves as both a testament to the potential unlocked within the field of computer networking and as a guide for those embarking on similar projects. Whether you are a student striving to comprehend the intricate world of network communication or a professional seeking insights into robust data transmission, this narrative invites you to journey with us through the realm of computer networks and the development of an application poised to shape this landscape.

2 Architecture

The developed application consists of two distinct layers: the Link Layer and the Application Layer. Let's examine each of them in detail.

2.1 Link layer

This layer encompasses the necessary functions to initiate and terminate the connection between the two computers through the serial port, as well as handling read and write operations. Additionally, it includes the function for message destuffing, which perform removal of escape characters to protect messages containing data bytes identical to message delimiters (flags).

2.2 Application layer

The Application Layer spans across several files, specifically application.c, transmitter.c, and receiver.c, and serves as an intermediary between the user and the Data-Link Layer. It receives user arguments and manages everything related to file opening, reading, and writing.

This division into layers facilitates modularity and organization of the application, enabling better code maintenance and extensibility. The Link Layer handles connectivity and data transmission, while the Application Layer deals with user interactions and file processing.

3 Code structure

The program code is organized into seven different files, based on the layers they operate in (Application Layer or Link Layer), the role they play (Transmitter or Receiver), and the functionalities they implement. Each of these files also has an associated header file. In addition, the program includes two header files that define directives.

3.1 Link_layer.c

Functions llopen(): Opens a serial port and performs the exchange of SET and UA frames.

llwrite(): Handles data frame transmission with byte stuffing for reliable communication.

llread(): Handles incoming I-frames, performs destuffing, and responds with RR or REJ

based on data integrity. It also copies valid data to the buffer **llclose()**:

Terminates serial communication with DISC and UA frames.

messageDestuffing(): Removes escape characters from a byte sequence and returns the resulting message size.

BCC2(): Calculates and returns the BCC-2 (Block Check Character 2) for a given byte sequence and size.

3.2 Message.c

Functions

messageStuffing(): Performs byte ”stuffing” on a byte sequence, adding escape characters as needed and returning the size of the stuffed message.

prepareSupervisionMessage(): Prepares and sends supervision messages with address and control codes, handling response types.

prepareDataMessage(): Prepares data messages, calculates control codes, BCCs, and performs byte stuffing. It manages data transmission and acknowledgment reception.

sendMessage(): Sends a message, handles response types, and manages acknowledgment reception with retry attempts.

readMessage(): Reads a message, handles response types, and manages acknowledgment reception with a specified timeout.

3.3 State.c

Functions

getState(), getLastResponse(), and getRole() : Return the current state, the most recent received response, and the application’s role, respectively.

setStateMachineRole(): Sets the application’s role (Transmitter/Receiver).

configStateMachine(): Adjusts the state machine’s behavior according to the desired response type.

updateState(): Processes the received bytes and manages the state transitions of the state machine, handling different states such as START, FLAGRCV, ARCV, CRCV, WAITINGDATA, BCCOK, and STOP.

FlagRCVstateHandler(): Determines state transitions based on the received byte, considering the current mode and role. It can transition to the ARCV state or START state.

ARCVstateHandler(): Handles byte reception in the ARCV state, considering the current mode and role to transition to the CRCV state or START state based on the received byte.

CRCVstateHandler(): Handles bytes in the CRCV state, transitioning to WAITINGDATA, BCCOK, or START state based on BCC (Block Check Character) and the current mode.

WaitingDatastateHandler(): Manages state transitions when receiving bytes in the WAITINGDATA state. It transitions to the STOP state upon receiving a FLAG (MSGFLAG) or remains in the WAITINGDATA state if other bytes are received.

3.4 Transmitter.c

Functions

transmitterApplication(): Handles the transmission of a file over a serial connection. It reads the file and sends it in packets, including start and end packets.

sendControlPacket(): Prepares and sends a control packet, including information about file size and file name, to be transmitted over the serial connection.

3.5 Receiver.c

Functions

receiverApplication(): Receives and parses packets from the serial connection to store in the specified file path. It continues until an end packet is received or the maximum number of failed attempts is reached.

parsePacket(): Handles the parsing of received packets, including start, end, and data packets, and manages the associated file operations. It also checks for unmarked packets.

3.6 Application layer.c

Functions

applicationLayer(): Controls the application, including link layer setup, role-specific application execution, data exchange, and connection closure.

3.7 Main.c

Functions

Main(): Function reads command-line arguments and initiates the link-layer protocol application, displaying the specified configuration.

4 Use cases

To start the program, compile and run the application using the following command:

/gcc -o main ¡ all files name ¿

Receiver: ./main /dev/ttyS0 rx pinguin

Transmitter: ./main /dev/ttyS0 tx pinguin.gif

The receiver should be started first. Otherwise, the transmitter's application will send messages for 9 seconds (3 attempts with 3 seconds of waiting for a response each), and after that time, if the receiver's application hasn't yet been initiated, it will terminate the program.

5 Logical Link Protocol

The Logical Link Protocol is an essential component in data transmission over a serial connection. It defines a set of functions and procedures that ensure reliable serial communication between a transmitter and a receiver. The protocol manages aspects such as opening and closing the connection, data transmission, and error detection.

llopen

The llopen function is responsible for establishing a serial connection. To do this, it opens the serial port and configures communication parameters, such as the transmission speed and data format. The function also calls readMessage() to receive the initial message (SET or UA) and prepareSupervisionMessage() to send a response (UA or SET), depending on the assigned role (Transmitter or Receiver).

llwrite

The llwrite function is responsible for transmitting data frames (I-frames) over the serial connection. It performs byte stuffing on these I-frames before transmission. To achieve this, the function utilizes the prepareDataMessage() to construct a data frame containing the bytes to be sent. The function then attempts to send this frame, and in the case of successful transmission, it returns the number of bytes transmitted. However, if transmission encounters issues, the function will make repeated attempts after a certain number of tries until successful transmission is achieved.

llread

The llread function is primarily responsible for receiving data frames (I-frames) over the serial connection. This function initiates by receiving data frames in the COMMANDDATA format through the readMessage function. It then proceeds with byte destuffing, a process carried out by the messageDestuffing function, to remove escape characters and reconstruct the original frame, storing the destuffed data in the unstuffedMessage buffer. Following this,

the function verifies data integrity by calculating and comparing the Block Check Character 2 (BCC2) using the BCC2(). If the frame is valid, based on a matching BCC2, the function updates the sequence number, prepares a supervision message using prepareSupervisionMessage(), and copies the data to the user-provided buffer. However, in the event of errors like duplicate packets or BCC2 discrepancies, the function dispatches appropriate supervision messages and performs serial connection flushing using tcflush.

llclose

The llclose function serves the critical role of closing the serial connection in a controlled manner. The behavior of this function depends on the role assigned to it, which can be either transmitter (LlTx) or receiver (LlRx). When operating as a receiver, the function reads the COMMANDDISC message through the readMessage(), indicating the desire to disconnect.
It responds by transmitting a supervision message prepared via prepareSupervisionMessage(), acknowledging the disconnection request with an UA. Conversely, when functioning as a transmitter, the llclose function initiates the disconnection process. It sends a COMMANDDISC message using prepareSupervisionMessage() and waits for the UA response. Afterward, it ensures that the serial port's configuration is returned to its original settings and closes the serial connection before exiting. These functionalities encapsulate the overall operation of the llclose function, facilitating the proper termination of the serial communication link.

6 Conclusion

The project implements a custom data link layer protocol, which handles data transmission over a virtual serial port. It involves concepts like frame encoding and decoding, error detection, and flow control. The code snippets provided represent various components of this protocol, including functions for opening and closing the connection, sending and receiving data frames, handling control packets, and more.

The transmitter application reads data from a file and transmits it to the receiver using the custom protocol. The receiver application receives the data, checks for errors, and saves it to a destination file. The project also includes options for specifying the serial port, baud rate, number of retries, and timeout parameters.

It's worth noting that this project appears to be a hands-on implementation of a simple data link layer protocol for educational purposes or specific applications where custom communication is required.

Overall, the project demonstrates the practical application of data link layer concepts in a real-world scenario, emphasizing the implementation of a reliable data communication protocol over a virtual communication channel.

7 Code Sample

Link layer.c

```
1 // Link layer protocol implementation
2
3 #include "include/link_layer.h"
```

```
4
5 struct termios oldtio;
6
7           unsigned char BCC2(unsigned char * data, int dataSize, int startingByte) {
8           unsigned char bcc = data[startingByte];
9
10                  for(int i = startingByte + 1; i < dataSize; i++)
11                  bcc = bcc ^ data[i];
12
13           return bcc;
14           }
15
16           int messageDestuffing(unsigned char * buffer, int startingByte, int lenght, unsigned char * ,→ destuffedMessage) {
17           int messageSize = 0;
18
19                  for (int i = 0; i < startingByte; i++) {
20                  destuffedMessage[messageSize++] = buffer[i];
21                  }
22
23                        for (int i = startingByte; i < lenght; i++) {
24                        if (buffer[i] == ESCAPE) {
25                        destuffedMessage[messageSize++] = buffer[i + 1] ^ 0x20;
26                        i++;
27                        }
28                        else {
29                        destuffedMessage[messageSize++] = buffer[i];
30                        }
31                        }
32
33           return messageSize;
34           }
35
36                  int llopen(LinkLayer connectionParameters) {
37                  int fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
38                  if (fd < 0) {
39                  perror("Error opening port");
40                  return -1;
41                  }
42
43                 struct termios newtio;
44
45                  if (tcgetattr(fd, &oldtio) == -1) {
46                  perror("tcgetattr failed");
47                  return -1;
48                  }
49
50           memset(&newtio, 0, sizeof(newtio));
51           newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
52           newtio.c_iflag = IGNPAR;
53           newtio.c_oflag = 0;
54
55           /* set input mode (non-canonical, no echo,...) */
56           newtio.c_lflag = 0;
57
58            newtio.c_cc[VTIME]= 1;          /* inter-character timer unused */
59            newtio.c_cc[VMIN]  = 0; /* blocking read until 5 chars received */
60
61                tcflush(fd, TCIOFLUSH);
62
63                  if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
64                  perror("tcsetattr failed to set new termios struct");
65                  return -1;
66                  }
67
```

```
68              printf("New termios structure set\n");
69
70             (void)signal(SIGALRM, alarm_handler);
71
72                  int ok;
73                  if(connectionParameters.role == LlRx){
74                  setStateMachineRole(LlRx);
75                  unsigned char message[5];
76                  if (readMessage(fd, message, COMMAND_SET) < 0) return -1;
77                  ok = prepareSupervisionMessage(fd, MSG_A_RECV_RESPONSE, MSG_CTRL_UA, NO_RESPONSE);
78                  if(ok == -1)
79                  return ok;
80                  }
81                  else if(connectionParameters.role == LlTx){
82                  setStateMachineRole(LlTx);
83                  ok = prepareSupervisionMessage(fd, MSG_A_TRANS_COMMAND, MSG_CTRL_SET, RESPONSE_UA);
84                  if(ok == -1)
85                  return ok;
86                  }
87                  else
88                  return -1;
89
90      return fd;
91      }
92
93      int llwrite(int fd, unsigned char * buffer, int lenght) {
94      static int packet = 0;
95
96      int ret;
97      int numTries = 0;
98
99                  while (numTries < TIMEOUT) {
100                 numTries++;
101                 if ((ret = prepareDataMessage(fd, buffer, lenght, packet)) > -1) {
102                 packet = (packet + 1) % 2;
103                 return ret;
104                 }
105                 fprintf(stderr, "sendDataMessage failed\n");
106                 }
107
108     return -1; 109 }
110
111             int llread(int fd, unsigned char * buffer) {
112             static int packet = 0;
113             unsigned char stuffedMessage[MAX_BUFFER_SIZE], unstuffedMessage[MAX_PACKET_SIZE + 7];
114             int numBytesRead;
115             if ((numBytesRead = readMessage(fd, stuffedMessage, COMMAND_DATA)) < 0) {
116             fprintf(stderr, "Read operation failed\n");
117             return -1;
118             }
119             int res = messageDestuffing(stuffedMessage, 1, numBytesRead - 1, unstuffedMessage);
120
121       unsigned char receivedBCC2 = unstuffedMessage[res - 1];
122       unsigned char receivedDataBCC2 = BCC2(unstuffedMessage, res - 1, 4);
123
124             if (receivedBCC2 == receivedDataBCC2 && unstuffedMessage[2] == MSG_CTRL_S(packet)) {
125             packet = (packet + 1) % 2;
126             if (prepareSupervisionMessage(fd, MSG_A_RECV_RESPONSE, MSG_CTRL_RR(packet), NO_RESPONSE) < 0) return
              ,→ -1;
127             memcpy(buffer, &unstuffedMessage[4], res-5);
128             return res - 5;
129             }
130             else if (receivedBCC2 == receivedDataBCC2) {
```

```
                prepareSupervisionMessage(fd, MSG_A_RECV_RESPONSE, MSG_CTRL_RR(packet), NO_RESPONSE);
                fprintf(stderr, "Duplicate Packet!\n");
                tcflush(fd, TCIFLUSH);
                return -1;
                } else {
                prepareSupervisionMessage(fd, MSG_A_RECV_RESPONSE, MSG_CTRL_REJ(packet), NO_RESPONSE);
                fprintf(stderr, "Error in BCC2, sent REJ!\n");
                tcflush(fd, TCIFLUSH);
                return -1;
                }
                }

int llclose(int fd) {
        switch (getRole()) {
                        case LlRx:
                        printf("Closing Reciever\n");
                        unsigned char message[5];
                        if (readMessage(fd, message, COMMAND_DISC) < 0) return -1;
                        prepareSupervisionMessage(fd, MSG_A_RECV_COMMAND, MSG_CTRL_DISC, RESPONSE_UA);
                        break;
                        case LlTx:
                        printf("Closing transmitter\n");
                        if (prepareSupervisionMessage(fd, MSG_A_TRANS_COMMAND, MSG_CTRL_DISC, COMMAND_DISC) < 0) return -1;
                                prepareSupervisionMessage(fd, MSG_A_TRANS_RESPONSE, MSG_CTRL_UA, NO_RESPONSE);
                        break;
                        default:
                        return -1;
                        }
                        if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
                        perror("tcsetattr failed to set old termios struct");
                        exit(-1);
                        }

          return close(fd);
          }
```

Link layer.h

```
  // Link layer header.
  // NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_


   #include <sys/types.h>
   #include <sys/stat.h>
   #include <fcntl.h>
   #include <termios.h>
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <signal.h>
   #include <string.h>
   #include "state.h"
   #include "message.h"

#define _POSIX_SOURCE 1 /* POSIX compliant source */

          typedef enum
```

```
        {
        LlTx,
        LlRx,
        } LinkLayerRole;

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


    int llopen(LinkLayer connectionParameters);
    int llclose(int fd);
    int llwrite(int fd, unsigned char * buffer, int lenght);
    int llread(int fd, unsigned char * buffer);

#endif // _LINK_LAYER_H_
```

Message.c

```
#include "include/message.h"

int alarm_flag = FALSE;

        void alarm_handler() {
        alarm_flag = TRUE;
        }

int messageStuffing(unsigned char * buffer, int startingByte, int length, unsigned char * stuffedMessage) {         int messageSize = 0;

                for (int i = 0; i < startingByte; i++)
                stuffedMessage[messageSize++] = buffer[i];

                        for (int i = startingByte; i < length; i++) {
                        if (buffer[i] == MSG_FLAG || buffer[i] == ESCAPE) {
                        stuffedMessage[messageSize++] = 0x7d;
                        stuffedMessage[messageSize++] = buffer[i] ^ 0x20;
                        }
                        else {
                        stuffedMessage[messageSize++] = buffer[i];
                        }
                        }

        return messageSize;
        }

int prepareSupervisionMessage(int fd, unsigned char address, unsigned char control, mode responseType) {         unsigned char msg[5]
= {
                MSG_FLAG,
                address,
                control,
                BCC(address, control),
                MSG_FLAG
                };

                        if (responseType != NO_RESPONSE) {
```

```
38                  if (sendMessage(fd, msg, 5, responseType) < 0)
39                  return -1;
40
41              return 0;
42              }
43              else {
44              if (write(fd, msg, 5) == -1) {
45              fprintf(stderr, "Write failed\n");
46              }
47
48              return 0;
49              }
50              }
51
52          int prepareDataMessage(int fd, unsigned char * data, int dataSize, int packet) {
53          int msgSize = dataSize + 5;
54
55              unsigned char msg[msgSize];
56
57              msg[0] = MSG_FLAG;
58              msg[1] = MSG_A_TRANS_COMMAND;
59              msg[2] = MSG_CTRL_S(packet);
60              msg[3] = BCC(MSG_A_TRANS_COMMAND, MSG_CTRL_S(packet));
61              unsigned char bcc2 = data[0];
62              for (int i = 0; i < dataSize; i++) {
63              msg[i + 4] = data[i];
64              if (i > 0) bcc2 ^= data[i];
65              }
66              msg[dataSize + 4] = bcc2;
67
68          unsigned char stuffedData[msgSize * 2];
69          msgSize = messageStuffing(msg, 1, msgSize, stuffedData);
70          stuffedData[msgSize] = MSG_FLAG;
71          msgSize++;
72
73          int numTries = 0;
74          int receivedACK = FALSE;
75          int ret;
76
77              do {
78              numTries++;
79              ret = sendMessage(fd, stuffedData, msgSize, RESPONSE_RR_REJ);
80
81                  response_type response = getLastResponse();
82
83                  if (ret > 0 && ((packet == 0 && response == R_RR1) || (packet == 1 && response == R_RR0))) {
84                  receivedACK = TRUE;
85                  } else if (ret > 0) {
86                  fprintf(stderr, "Received response is invalid. Trying again...\n");
87                  }
88                  } while (numTries < N_TRIES && !receivedACK);
89
90              if (!receivedACK) {
91              fprintf(stderr, "Failed to get ACK\n");
92              return -1;
93              }
94              else
95              return ret;
96              }
97              int sendMessage(int fd, unsigned char * msg, int messageSize, mode responseType) { 98          configStateMachine(responseType);
99
100          int numTries = 0;
101          int ret;
```

```
102
103             do {
104             numTries++;
105             alarm_flag = FALSE;
106
107             if ((ret = write(fd, msg, messageSize)) == -1) {
108             fprintf(stderr, "Write failed\n");
109             }
110
111             alarm(TIMEOUT);
112
113                 int res;
114                 unsigned char buf[MAX_BUFFER_SIZE];
115                 while (getState() != STOP && !alarm_flag) {
116                 res = read(fd, buf, 1);
117                 if (res == 0) continue;
118                 updateState(buf[0]);
119                 }
120
121              } while (numTries < N_TRIES && getState() != STOP);
122
123             if (getState() != STOP) {
124             fprintf(stderr, "Failed to get response!\n");
125             return -1;
126             }
127
128     return ret; 129 }
130
131     int readMessage(int fd, unsigned char * message, mode responseType) {
132     configStateMachine(responseType);
133     int res, numBytesRead = 0;
134     unsigned char buf[MAX_BUFFER_SIZE];
135     alarm_flag = FALSE;
136
137     alarm(TIMEOUT);
138
139             while (getState() != STOP && !alarm_flag && numBytesRead < MAX_BUFFER_SIZE) {
140             res = read(fd, buf, 1);
141             if (res == 0) continue;
142             alarm(0);
143             message[numBytesRead++] = buf[0];
144             updateState(buf[0]);
145             alarm(TIMEOUT);
146             }
147
148             if (alarm_flag) {
149             fprintf(stderr, "Alarm fired. readMessage took too long\n");
150             return -1;
151             }
152
153             if (getState() != STOP) {
154             fprintf(stderr, "Failed to read message\n");
155             return -1;
156             }
157
158     return numBytesRead;
159     }
```

Message.h

```c
#pragma once

    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <termios.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <string.h>
    #include "state.h"

void alarm_handler();

int prepareSupervisionMessage(int fd, unsigned char address, unsigned char control, mode responseType);

int prepareDataMessage(int fd, unsigned char * data, int dataSize, int packet);

int sendMessage(int fd, unsigned char * msg, int messageSize, mode responseType);

int readMessage(int fd, unsigned char * message, mode responseType);
```

State.c

```c
#include "include/state.h"

stateMachine state;

        msg_state getState() {
        return state.currentState;
        }

         response_type getLastResponse() {
         return state.last_response;
         }

         int getRole() {
         return state.role;
         }

  void setStateMachineRole(int role) {
  state.role = role; 19 }

        void configStateMachine(mode stateMachineMode) {
        state.currentState = START;
        state.last_response = R_NULL;
        state.currentMode = stateMachineMode;
        }

                        void updateState(unsigned char byte) {
                        switch (state.currentState) {
                        case START:
                        if (byte == MSG_FLAG)
                        state.currentState = FLAG_RCV;
                        break;
                        case FLAG_RCV:
                        FlagRCV_stateHandler(byte);
                        break;
```

```
36                          case A_RCV:
37                          ARCV_stateHandler(byte);
38                          break;
39                          case C_RCV:
40                          CRCV_stateHandler(byte);
41                          break;
42                          case WAITING_DATA:
43                          WaitingData_stateHandler(byte);
44                          break;
45                          case BCC_OK:
46                          if (byte == MSG_FLAG)
47                          state.currentState = STOP;
48                          else
49                          state.currentState = START;
50                          break;
51                          case STOP:
52                          break;
53                          }
54                          }
55
56              void FlagRCV_stateHandler(unsigned char byte) {
57              if (byte == MSG_FLAG)
58              return;
59
60          switch (state.currentMode) { 61   case
RESPONSE_UA:
62                          case RESPONSE_RR_REJ:
63                          if ((state.role == 0 && byte == MSG_A_RECV_RESPONSE) || (state.role == 1 && byte ==
                        ↪ MSG_A_TRANS_RESPONSE)) {
64                          state.currentState = A_RCV;
65                          state.address = byte;
66                          return;
67                          }
68                          break;
69                          case COMMAND_SET:
70                          case COMMAND_DISC:
71                          case COMMAND_DATA:
72                          if ((state.role == 1 && byte == MSG_A_TRANS_COMMAND) || (state.role == 0 && byte ==
                        ↪ MSG_A_RECV_COMMAND)) {
73                          state.currentState = A_RCV;
74                          state.address = byte;
75                          return;
76                          }
77                          break;
78                          }
79
80           state.currentState = START;
81           }
82
83              void ARCV_stateHandler(unsigned char byte) {
84              if (byte == MSG_FLAG) {
85              state.currentState = FLAG_RCV;
86              return;
87              }
88
89          switch (state.currentMode) { 90   case
RESPONSE_UA:
91                                      if (byte == MSG_CTRL_UA) {
92                                      state.currentState = C_RCV;
93                                      state.control = byte;
94                                      return;
95                                      }
96                                      break;
```

```
97                                  case RESPONSE_RR_REJ:
98                                  if (byte == MSG_CTRL_RR(0) || byte == MSG_CTRL_RR(1) || byte == MSG_CTRL_REJ(0) || byte ==,→
                                    MSG_CTRL_REJ(1)) {
99                                  state.currentState = C_RCV;
100                                 state.control = byte;
101                                 switch (byte) {
102                                 case MSG_CTRL_RR(0):
103                                 state.last_response = R_RR0;
104                                 break;
105                                 case MSG_CTRL_RR(1):
106                                 state.last_response = R_RR1;
107                                 break;
108                                 case MSG_CTRL_REJ(0):
109                                 state.last_response = R_REJ0;
110                                 break;
111                                 case MSG_CTRL_REJ(1):
112                                 state.last_response = R_REJ1;
113                                 break;
114                                 }
115                                 return;
116                                 }
117                                 break;
118                                 case COMMAND_SET:
119                                 if (byte == MSG_CTRL_SET) {
120                                 state.currentState = C_RCV;
121                                 state.control = byte;
122                                 return;
123                                 }
124                                 break;
125                                 case COMMAND_DISC:
126                                 if (byte == MSG_CTRL_DISC) {
127                                 state.currentState = C_RCV;
128                                 state.control = byte;
129                                 return;
130                                 }
131                                 break;
132                                 case COMMAND_DATA:
133                                 if (byte == MSG_CTRL_S(0) || byte == MSG_CTRL_S(1)) {
134                                 state.currentState = C_RCV;
135                                 state.control = byte;
136                                 return;
137                                 }
138                                 break; 139          }
140
141         state.currentState = START;
142         }
143
144               void CRCV_stateHandler(unsigned char byte) {
145               if (byte == MSG_FLAG) {
146               state.currentState = FLAG_RCV;
147               return;
148               }
149
150         if (byte == BCC(state.address, state.control))
151         if (state.currentMode == COMMAND_DATA) 152          state.currentState = WAITING_DATA;
153                     else
154                     state.currentState = BCC_OK;
155                     else
156                     state.currentState = START;
157                     }
158
159               void WaitingData_stateHandler(unsigned char byte) {
160               if (byte == MSG_FLAG) {
161               state.currentState = STOP;
```

```
162                 return;
163                 }
164                 else return;
165                 }
```

State.h

```
1 #pragma once
2
3 #include <stdio.h>
4
5 #include "globals.h"
6
7   typedef enum {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, WAITING_DATA, STOP} msg_state;
8   typedef enum {RESPONSE_UA, RESPONSE_RR_REJ, COMMAND_SET, COMMAND_DISC, COMMAND_DATA} mode; 9 typedef enum
    {R_RR0, R_RR1, R_REJ0, R_REJ1, R_NULL} response_type;
10
11          typedef struct {
12          msg_state currentState;
13          mode currentMode;
14          int role;
15          unsigned char control;
16          unsigned char address;
17          response_type last_response;
18          } stateMachine;
19
20 msg_state getState();
21
22 response_type getLastResponse();
23
24 int getRole();
25
26 void setStateMachineRole(int role);
27
28 void configStateMachine(mode stateMachineMode);
29
30 void updateState(unsigned char byte);
31
32 void FlagRCV_stateHandler(unsigned char byte);
33
34 void ARCV_stateHandler(unsigned char byte);
35
36 void CRCV_stateHandler(unsigned char byte);
37
38 void WaitingData_stateHandler(unsigned char byte);
```

Transmitter.c

```
1 #include "include/transmitter.h"
2
3           int transmitterApplication(int fd, const char* path) {
4           int input_fd;
5           struct stat file_stat;
6
7                   if (stat(path, &file_stat)<0){//Stat used for getting file size in bytes
8                   perror("Error getting file information.");
```

```
9               return -1;
10              }
11
12               if ((input_fd = open(path, O_RDONLY)) < 0){
13               perror("Error opening file.");
14               return -1;
15               }
16
17
18               if (sendControlPacket(fd, START_PACKET, file_stat.st_size, path) < 0) {
19               fprintf(stderr, "Error sending START packet.\n");
20               return -1;
21               }
22
23          unsigned char buf[MAX_PACKET_SIZE];
24          unsigned bytes_to_send;
25          unsigned sequenceNumber = 0;
26
27               while ((bytes_to_send = read(input_fd, buf, MAX_PACKET_SIZE - 4)) > 0) {
28               unsigned char dataPacket[MAX_PACKET_SIZE];
29               dataPacket[0] = DATA_PACKET;
30               dataPacket[1] = sequenceNumber % 255;
31               dataPacket[2] = (bytes_to_send / 256);
32               dataPacket[3] = (bytes_to_send % 256);
33               memcpy(&dataPacket[4], buf, bytes_to_send);
34
35                    if (llwrite(fd, dataPacket, ((bytes_to_send + 4) < MAX_PACKET_SIZE)? (bytes_to_send + 4) :
                 ,→ MAX_PACKET_SIZE) < 0) { // Only sends max packet if the last packet is of that size
36                    fprintf(stderr, "llwrite failed\n");
37                    return -1;
38                    }
39
40               sequenceNumber++;
41               }
42
43                printf("Data packets sent: %d\n",sequenceNumber);
44
45               if (sendControlPacket(fd, END_PACKET, file_stat.st_size, path) < 0) {
46               fprintf(stderr, "Error sending END packet.\n");
47               return -1;
48               }
49               printf("Total packets sent: %d\n",sequenceNumber+2);
50
51          return close(input_fd);
52          }
53
54
55
56          int sendControlPacket(int fd, unsigned char ctrl_field, unsigned file_size, const char* file_name) {
57          unsigned L1 = sizeof(file_size);
58          unsigned L2 = strlen(file_name);
59          unsigned packet_size = 5 + L1 + L2;
60
61          unsigned char packet[packet_size];
62          packet[0] = ctrl_field;
63          packet[1] = 0;
64          packet[2] = L1;
65          memcpy(&packet[3], &file_size, L1);
66          packet[3+L1] = 1;
67          packet[4+L1] = L2;
68          memcpy(&packet[5+L1], file_name, L2);
69
70          return llwrite(fd, packet, packet_size);
71          }
```

Transmitter.h

```c
#pragma once

    #include <sys/types.h>
    #include <sys/stat.h>
    #include <stdio.h>
    #include <fcntl.h>
    #include <unistd.h>
    #include <string.h>
    #include <errno.h>

     #include "link_layer.h"
     #include "globals.h"

int transmitterApplication(int fd, const char* path);

int sendControlPacket(int fd, unsigned char ctrl_field, unsigned file_size, const char* file_name);
```

Receiver.c

```c
#include "include/receiver.h"

        int receiverApplication(int fd, const char* path) {
        int res;
        int nump = 0;
        int numTries = 0;

                    while (1) {
                    unsigned char buf[MAX_PACKET_SIZE];
                    if ((res = llread(fd, buf)) < 0) {
                    if (numTries > N_TRIES) return -1;
                    numTries++;
                    continue;
                    }

                numTries = 0;
                nump++;

                    int ret;
                    if ((ret = parsePacket(buf, path)) == END_PACKET)
                    break;
                    else if (ret == -1)
                    return -1;
                    }

          printf("Received %d packets\n", nump);
          return 0;
          }

int parsePacket(unsigned char * buffer, const char* path) { 31   static int
destinationFile;

                    if (buffer[0] == START_PACKET) {
                    if ((destinationFile = open(path, O_WRONLY | O_CREAT, 0777)) < 0) {
                    perror("Error opening destination file!");
                    return -1;
```

```
                }

        return 0;
        } else if (buffer[0] == END_PACKET) {            if (close(destinationFile) < 0) {
                perror("Error closing destination file!");
                return -1;
                }

                return END_PACKET;
                } else if (buffer[0] == DATA_PACKET) {
                unsigned dataSize = buffer[3] + 256 * buffer[2];
                if (write(destinationFile, &buffer[4], dataSize) < 0) {
                perror("Error writing to destination file!");
                return -1;
                }
                return 0;
                } else {
                printf("Unmarked packet!\n");
                return -1;
                }
                }
```

Receiver.h

```
#pragma once

  #include <sys/types.h>
  #include <sys/stat.h>
  #include <stdio.h>
  #include <fcntl.h>
  #include <unistd.h>
  #include <string.h>

   #include "link_layer.h"
   #include "globals.h"

int receiverApplication(int fd, const char* path);

int parsePacket(unsigned char * buffer, const char* path);
```

Application layer.c

```
  // Application layer protocol implementation
  #include "include/application_layer.h"

                        void applicationLayer(const char *serialPort, const char *role, int baudRate,
                        int nTries, int timeout, const char *filename)
                        {
                        LinkLayer linkLayer;
                        strcpy(linkLayer.serialPort, serialPort);
                        linkLayer.nRetransmissions = nTries;
                        linkLayer.timeout = timeout;
                        if (!strcmp(role, "tx")) {
                        linkLayer.role = LITx;
                        }
                        else if (!strcmp(role, "rx")) {
```

```
                                linkLayer.role = LlRx;
                                }
                                else {
                                fprintf(stderr, "Wrong role input\n");
                                return;
                                }

                        printf("App initialized!\nPort: %s\n", linkLayer.serialPort);

        int fd;

            if ((fd = llopen(linkLayer)) < 0) {
            fprintf(stderr, "llopen failed\n");
            return;
            }

                    if (linkLayer.role == LlTx) {
                    if (transmitterApplication(fd, filename) < 0) {
                    fprintf(stderr, "Transmitter Application failed\n");
                    return;
                    }
                    }
                    else {
                    if (receiverApplication(fd, filename) < 0) {
                    fprintf(stderr, "Receiver Application failed\n");
                    return;
                    }
                    }

            if (llclose(fd) < 0){
            fprintf(stderr, "llclose failed\n");
            return;
            }
            }
```

Application layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_ #define
_APPLICATION_LAYER_H_


 #include <sys/types.h>
 #include <sys/stat.h>
 #include <sys/time.h>
 #include <fcntl.h>
 #include <stdio.h>
 #include <termios.h>
 #include <stdlib.h>
 #include <string.h>
 #include <unistd.h>
 #include <signal.h>
 #include <errno.h>

 #include "link_layer.h"
 #include "receiver.h"
 #include "transmitter.h"

```

```
typedef struct {
char name[50];
int role;
char path[256];
} applicationArgs;

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

#endif // _APPLICATION_LAYER_H_
```

Main

```
// Main file of the serial port project.
// NOTE: This file must not be changed.

#include <stdio.h>
#include <stdlib.h>

#include "include/application_layer.h"

        // Arguments:
        //   £1: /dev/ttySxx
        //   £2: tx | rx
        //   £3: filename
        int main(int argc, char *argv[])
        {
        if (argc < 4)
        {
        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);
        exit(1);
        }

    const char *serialPort = argv[1];
    const char *role = argv[2];
    const char *filename = argv[3];

            printf("Starting link-layer protocol application\n"
            " - Serial port: %s\n"
            " - Role: %s\n"
            " - Baudrate: %d\n"
            " - Number of tries: %d\n"
            " - Timeout: %d\n"
            " - Filename: %s\n",
            serialPort,
            role,
            BAUDRATE,
            N_TRIES,
            TIMEOUT,
            filename);

             applicationLayer(serialPort, role, BAUDRATE, N_TRIES, TIMEOUT, filename);

    return 0;
    }
```

Globals.h

```
#pragma once

    #define FALSE 0
    #define TRUE 1

    #define BAUDRATE 9600
    #define N_TRIES 3
    #define TIMEOUT 4

     #define MAX_PACKET_SIZE 256
     #define MAX_BUFFER_SIZE (MAX_PACKET_SIZE * 2 + 7)

#define NO_RESPONSE -1

     #define MSG_FLAG 0x7e
     #define ESCAPE 0x7d

     #define MSG_A_TRANS_COMMAND 0x03
     #define MSG_A_RECV_RESPONSE 0x03
     #define MSG_A_TRANS_RESPONSE 0x01
     #define MSG_A_RECV_COMMAND 0x01

     #define MSG_CTRL_SET 0x03
     #define MSG_CTRL_UA 0x07
     #define MSG_CTRL_RR(r) ((r == 0) ? 0x05 : 0x85)
     #define MSG_CTRL_REJ(r) ((r == 0) ? 0x01 : 0x81)
     #define MSG_CTRL_DISC 0x0b
     #define MSG_CTRL_S(r) ((r == 0) ? 0x00 : 0x40)

#define BCC(addr, ctrl) (addr^ctrl)

     #define DATA_PACKET 1
     #define START_PACKET 2
     #define END_PACKET 3
```

8 Statistical Characterization

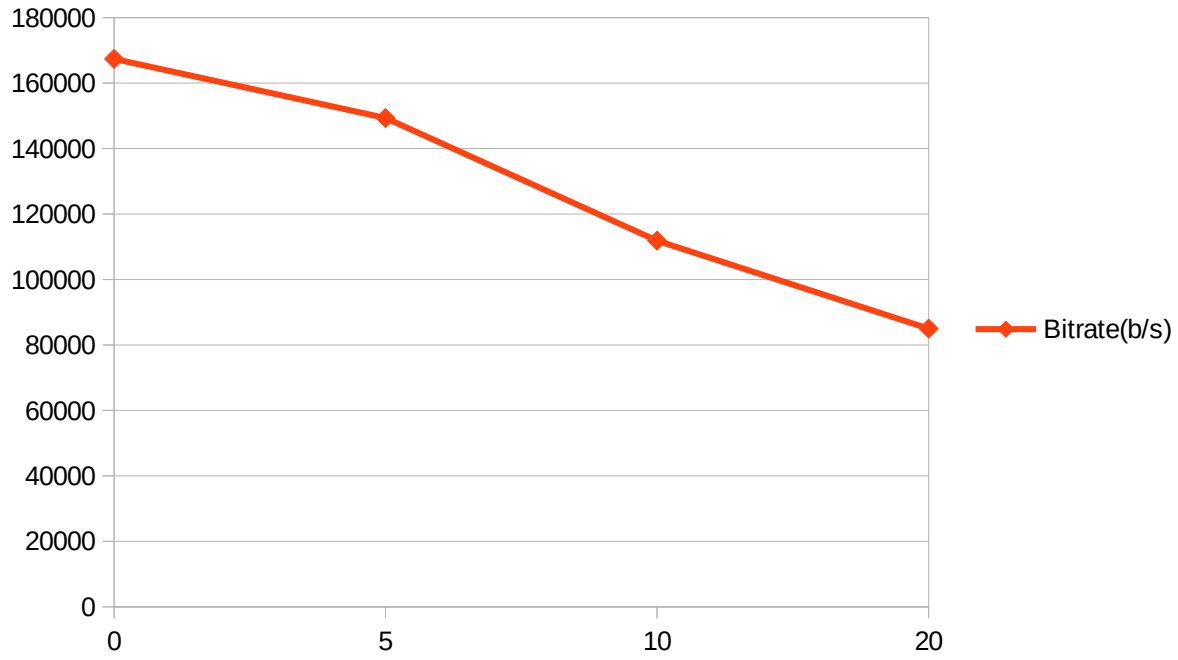
For the first 2 tables the following values will be used:

File used: Penguin.gif (10968 Bytes)

Package size: 256

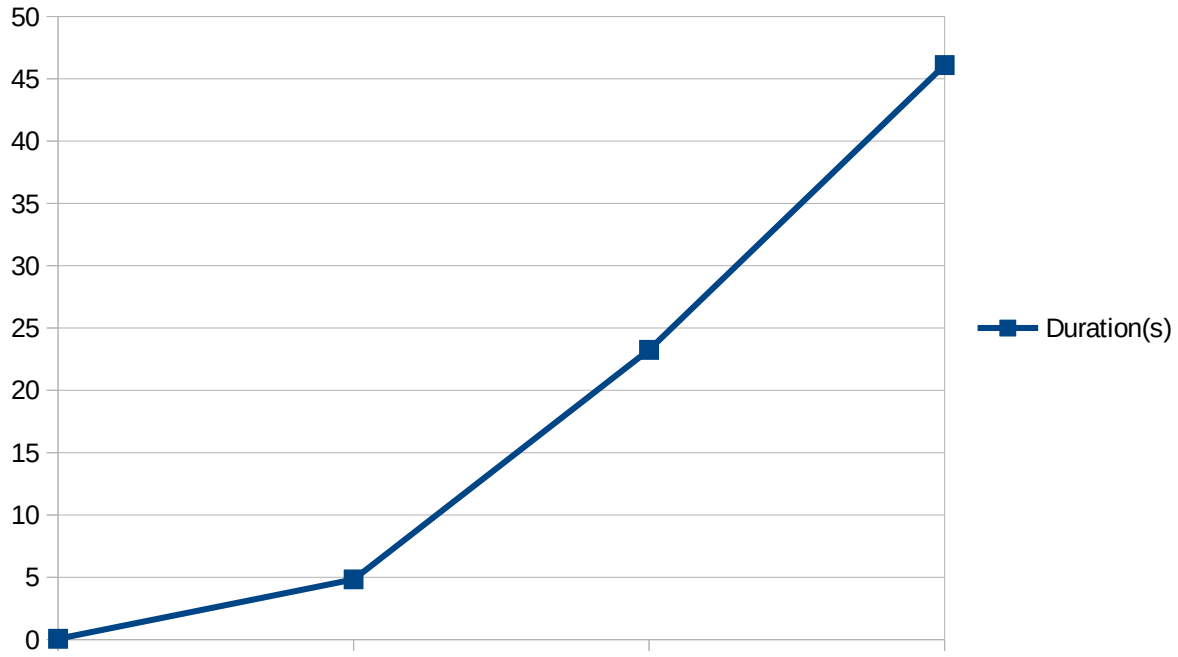
Baudrate: 9600

1. Varied FER – BCC2



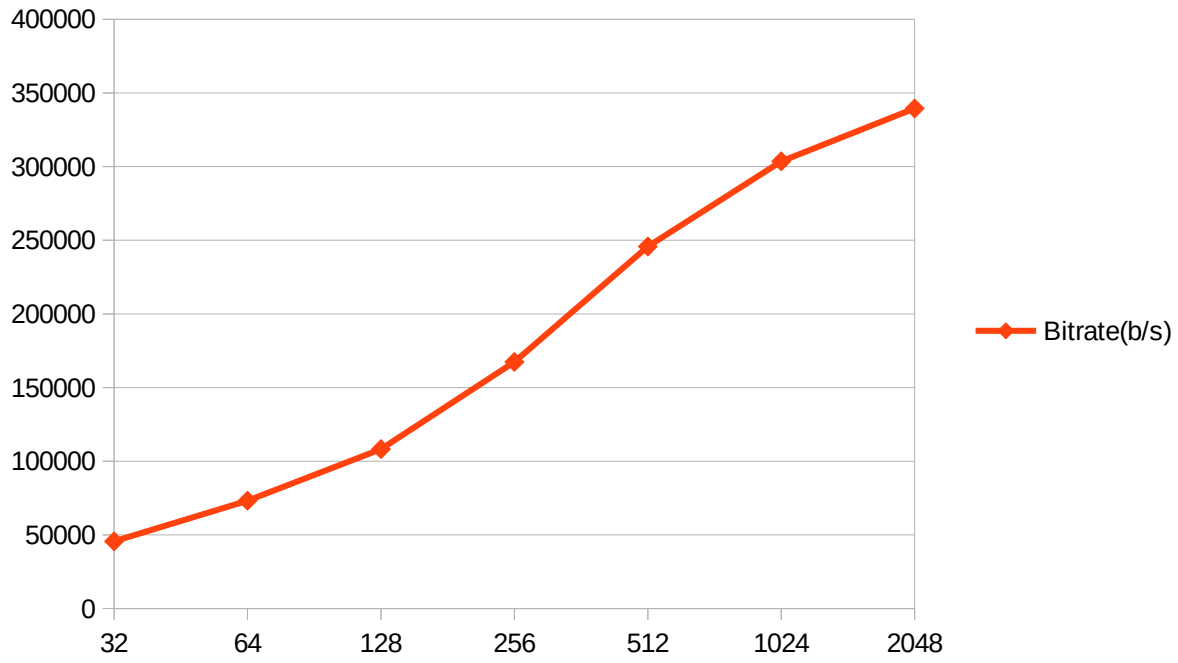
Error Percentage(%)	Duration(s)	Bitrate(b/s)
0	0.06552	167399
5	0.07344	149346
10	0.09804	111872
20	0.12903	85003

2. Varied Propagation Time



Propagation time(ms)	Duration(s)	Bitrate(b/s)
0	0.06552	167399
100	4.83351	2269
500	23.24965	471
1000	46.09829	237

3. Varied Frame Size



Packet Size	Duration(s)	Bitrate(b/s)
32	0.24050	45604
64	0.14977	73232
128	0.10129	108283
256	0.06552	167399
512	0.04462	245809
1024	0.03613	303570
2048	0.03231	339461