Faculdade de Engenharia da Universidade do Porto

1º Trabalho Laboratorial

Protocolo de Ligação de Dados

L.EIC

Redes de Computadores

Turma 10 - Grupo 9

Domingos Neto - up202108728@up.pt

Luana Lima - up202206845@up.pt

10 de novembro de 2024

Resumo

Este projeto foi desenvolvido no contexto da Unidade Curricular de Redes de Computadores e tem como objetivo estabelecer um protocolo de ligação de dados, implementando para isso funcionalidades de emissor e recetor para transmitir um ficheiro entre dois computadores utilizando a porta série RS-232.

1. Introdução

O objetivo deste trabalho foi desenvolver uma comunicação confiável e estável entre dois computadores através da porta série, utilizando um protocolo Stop & Wait, de acordo com as especificações do guião fornecido. Este relatório documenta os principais aspetos da implementação, descrevendo as interfaces e as funcionalidades da camada de ligação e de aplicação.

Assim, este está organizado nas seguintes 9 secções:

2. Arquitetura (Blocos funcionais e interfaces do sistema)

A arquitetura do sistema foi dividida em dois blocos principais: a **LinkLayer**, que implementa o protocolo de ligação de dados, e a **ApplicationLayer**, responsável por iniciar a transmissão e controlar o envio dos pacotes de dados.

- **Camada de Ligação de Dados**: Implementa as funções de controlo de fluxo e erro, sendo responsável pela criação e envio de frames de dados (I), supervisão (S) e não numerados (U). Esta camada disponibiliza uma interface de API (**llopen**, **llwrite**, **llread**, **llclose**) que permite à camada de aplicação comunicar através da porta série.
- **Camada de Aplicação**: Divide o ficheiro a ser transmitido em pacotes que são enviados à camada de ligação, sendo possível nesta definir o tamanho das tramas de informação, a velocidade de transmissão e o número máximo de retransmissões.

A independência entre camadas foi mantida conforme a arquitetura de camadas, onde cada camada opera de forma independente, com uma interface de comunicação clara e definida entre elas.

3. Estrutura do código

- **Application Layer**

Na implementação desta camada não foram criadas estruturas auxiliares. Nos ficheiros *aplication_layer.h* e *application_layer.c* foram definidas as seguintes funções:

```
// Envia um pacote de controlo (de início ou de fim) contendo o tipo do pacote, o nome do ficheiro e o seu tamanho
int sendControlPacket(int packetType, const char* fileName, long fileSize);

// Lê um pacote de controlo e extrai o tamanho do ficheiro e o seu nome
int readControlPacket(unsigned char expectedPacketType, unsigned char* buffer, size_t* fileSize, char* fileName);

// Envia um pacote de dados com o conteúdo a ser transmitido
int sendDataPacket(unsigned char* data, int dataSize);

// A função principal da camada de aplicação que coordena a transmissão ou receção de ficheiros de acordo com o role (LlTx para emissor, LlRx para recetor)
void applicationLayer(const char *serialPort, const char *role, int baudRate, int maxRetries, int timeout, const char *filename);
```

- **Link Layer**

Na implementação desta camada foram utilizadas duas estruturas de dados já fornecidas ***LinkLayerRole*** e ***LinkLayer*** e foi criada uma terceira estrutura de dados ***State***.

Além disso foram definidas cinco funções nos ficheiros *link_layer.h* e *link_layer.c.*

```
// Armazena a informação de se o computador é recetor ou emissor
typedef enum {
    LlTx,
    LlRx,
} LinkLayerRole;

// Armazena a informação dos parâmetros associados à conexão
typedef struct {
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

// Armazena a informação do estado de leitura e receção das tramas
typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC1_OK,
    BCC2_OK,
    STOP_STATE
} State;

// Cria um alarme
void alarmHandler(int signal);

// Analisa o estado de leitura e receção das tramas de informação
State StateMachine(State *state, int func, LinkLayerRole role);

// Abre uma conexão e estabelece a ligação entre o emissor e recetor
int llopen(LinkLayer connectionParameters);

// Envia as tramas
int llwrite(const unsigned char *buf, int bufSize);

// Recebe e lê as tramas
int llread(unsigned char *packet);

// Termina a conexão aberta previamente
int llclose(int showStatistics);
```

4. Principais Casos de Uso

Os principais casos de uso do protocolo foram estruturados para cobrir os cenários de conexão, envio e receção de dados e finalização de comunicação. Assim, segue-se a sequência seguida para cada caso e para modo (Emissor ou Recetor):

1. Conexão

- **Emissor:**

A camada de aplicação invoca a função *llopen*, que inicia a conexão enviando um frame SET para o recetor. Em seguida, o emissor aguarda pela resposta UA do recetor, indicando que a conexão foi estabelecida com sucesso. Após a receção do UA, o emissor considera a conexão ativa e pronta para transmissão de dados.

- **Recetor:**

Quando o recetor recebe o frame SET enviado pelo emissor e responde com um frame UA. Esta resposta confirma ao emissor que o recetor está pronto para receber dados. Após enviar UA, o recetor entra em modo de espera, pronto para processar frames de dados enviados pelo emissor.

2. Transmissão de Dados

- **Emissor:**

O emissor divide o ficheiro em pacotes e usa *llwrite* para enviar cada pacote como um frame I. Após enviar, aguarda uma resposta do recetor. Um RR confirma a receção e permite o envio do próximo frame, enquanto um REJ indica um erro, exigindo retransmissão. Se houver timeout, o emissor reenvia o frame até atingir o limite de retransmissões.

- **Recetor:**

O recetor utiliza *llread* para ler cada frame I e verifica o campo BCC para validar os dados. Se o frame estiver correto, responde com RR e entrega os dados à camada de aplicação; em caso de erro, envia REJ. Frames duplicados são ignorados, e RR é reenviado para evitar retransmissões desnecessárias.

3. Finalização da Conexão

- **Emissor:**

Após enviar todos os dados, o emissor invoca *llclose* e envia um frame DISC ao recetor. Ao receber UA de confirmação, encerra a conexão, libertando os recursos e, opcionalmente, exibindo estatísticas como retransmissões e timeouts.

- **Recetor:**

Ao receber DISC, o recetor invoca *llclose*, responde com UA para confirmar o encerramento e liberta os recursos alocados.

5. Protocolo de Ligação Lógica

O protocolo de ligação implementa o método Stop & Wait para controlo de fluxo e correção de erros, garantindo que cada frame é confirmado antes do envio do próximo.

Principais Funcionalidades:

- Numeração de Frames: O protocolo usa uma contagem de módulo-2 (N0 e N1) para distinguir frames I consecutivos.

```
int llwrite(const unsigned char *buf, int bufSize) {
    [...]
    buf_W[0] = FLAG;
    buf_W[1] = A;
    if (info == 0) {
        buf_W[2] = C_N0;
    } else {
        buf_W[2] = C_N1;
    }
    buf_W[3] = A ^ buf_W[2];
[...]
```

- Controlo de Erros: Em caso de erro (detetado pelo BCC1 ou BCC2), o recetor envia um frame REJ, solicitando a retransmissão do último frame.

```
int llread(unsigned char *packet){
      [...]
      printf("BCC2 check failed, sending REJ\n");
      unsigned char rej;
      if (control_field == C_N0) {
          rej = C_REJ0;
          printf("Control Field is C_N0, sending REJ0\n");
      } else {
          rej = C_REJ1;
          printf("Control Field is C_N1, sending REJ1\n");
      }
      unsigned char buf_W[5] = {FLAG, A, rej, A ^ rej, FLAG};
      writeBytesSerialPort(buf_W, 5);
      [...]
```

- Transparência: A técnica de byte stuffing foi implementada para evitar que a flag (0x7E) apareça acidentalmente dentro do frame de dados. Cada byte 0x7E encontrado nos dados é substituído pela sequência 0x7D 0x5E durante a transmissão, garantindo que o recetor

consegue identificar os limites do frame sem confusão. Além disso, se o byte 0x7D ocorrer isoladamente dentro do frame este é também substituído pela sequência 0x7D 0x5D.

```
int llwrite(const unsigned char *buf, int bufSize) {
    [...]
    for(int i = 0; i < bufSize; i++) {
        if(buf[i] == FLAG || buf[i] == ESCAPE) {
            buf_W[data++] = ESCAPE;
            buf_W[data++] = (buf[i]^0x20);
        } else {
            buf_W[data++] = buf[i];
        }
    }
    if(BCC2 == FLAG || BCC2 == ESCAPE) {
        buf_W[data++] = ESCAPE;
        buf_W[data++] = (BCC2^0x20);
    } else {
        buf_W[data++] = BCC2;
    }
    buf_W[data++] = FLAG;
    [...]
```

6. Protocolo de Aplicação

A camada de aplicação utiliza dois tipos de pacotes: pacotes de controlo e de dados.

Principais Funcionalidades:

- Pacote de Início e Fim de Transmissão: Um pacote de controlo sinaliza o início (START) e o fim (END) da transmissão, contendo informações sobre o tamanho do ficheiro e nome.

```
void applicationLayer(const char *serialPort, const char *role, int baudRate,
int maxRetries, int timeout, const char *filename) {
      [...]
      if (readControlPacket(START_PACKET, buffer, &fileSize,
receivedFileName) < 0) {
         printf("Error reading control packet\n");
         free(buffer);
         exit(-1);
      }
      printf("Start packet received\n");
      [...]
      if (sendControlPacket(END_PACKET, filename, fileSize) < 0) {
         printf("Error sending end packet\n");
         exit(-1);
      }
      printf("End packet sent\n");
      [...]
```

7. Validação

De modo a garantir o correto funcionamento do projeto foram realizados os seguintes testes de validação:

- Desconexão temporária do cabo
- Desconexão total do cabo, impedindo a troca de dados
- Adição de ruído à porta série
- Utilização de diferentes valores de baudrate
- Utilização de diferentes tamanhos de payload

8. Eficiência do Protocolo de Ligação de Dados

Devido à implementação atual do nosso programa, não é possível variar os valores *Frame Error Rate (FER).* O nosso protocolo de implementação está definido para abortar a transmissão quando um limite de retransmissões é excedido. As retransmissões não são efetuadas com a precisão desejada, prevenindo assim a medição do tempo total de transmissão nestes casos.

Definimos como valores padrão de T_prop, C, I frame size de 0ms, 9600 bit/s, 1000 bytes respetivamente.

Variação do tempo de propagação:

T_prop (ms)	T (s)	R (bit/s)	S (%)
0.000	**15.000**	**7467.667**	**77.79**
10.000	**16.000**	**7000.000**	**72.92**
100.000	**18.000**	**6222.222**	**64.81**
1000.000	**41.000**	**2731.707**	**28.46**

À medida que o tempo de propagação aumenta, a eficiência diminui, demonstrando assim a vulnerabilidade do protocolo Stop & Wait perante tempos de propagação elevados.

Variação da capacidade de ligação:

C (bit/s)	T (s)	R (bit/s)	S (%)
4800	27.000	4148.148	86.42
9600	16.000	7000.000	72.92
38400	14.000	8000.000	20.83
115200	14.000	8000.000	6.94

O aumento da capacidade de ligação, proporciona um menor tempo de transmissão. No entanto, apresenta pouca eficiência devido à limitação do bitrate.

Variação de tamanho de payload:

Frame Size (bytes)	T (s)	R (bit/s)	S (%)
256	46.000	2048.000	21.33
512	25.000	3932.160	40.96
1000	16.000	7000.000	72.92

O tempo de transmissão diminui drasticamente com o aumento do tamanho da trama.Quanto menos tramas forem necessárias serem transmitidas, maior será a eficiência do protocolo.

9. Conclusões

Este protocolo de ligação de dados foi desenvolvido com uma arquitetura em duas camadas principais: *LinkLayer*, que abrangeu a interação com a porta série e a manipulação de tramas de informação e *ApplicationLayer*, que abrangeu a interação com o arquivo que seria transferido. Além disso, este projeto ajudou a fortalecer conceitos teóricos vistos nas aulas, como byte stuffing, framing e o protocolo Stop & Wait, principalmente no que se refere a deteção e tratamento de erros.

Apesar do progresso, a implementação não conseguiu lidar totalmente com o ruído na transmissão, o que impactou a robustez do protocolo. Esse desafio demonstrou a importância de uma estratégia mais robusta para lidar com ruídos e interferências na comunicação, realçando a complexidade de uma implementação confiável de Stop & Wait.

Apêndices

Apêndice I - link_layer.c

```
// Link layer protocol implementation

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

#define FALSE 0
#define TRUE 1
#define FLAG 0x7e
#define A 0x03
#define A_ 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_RR0 0xAA
#define C_RR1 0xAB
#define C_REJ0 0x54
#define C_REJ1 0x55
#define C_DISC 0x0B
#define C_N0 0x00
#define C_N1 0x80
#define ESCAPE 0x7d
#define TX 1
#define RX 0

typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
```

```
    BCC1_OK,
    BCC2_OK,
    STOP_STATE
} State;


int alarmEnabled = FALSE;
int alarmCount = 0;
int timeout;
int retransmissions;
int info = 0;
int role;

int i_frames = 0;
int u_frames = 0;
int s_frames = 0;
int dup_frames = 0;
int rej_frames = 0;


void alarmHandler(int signal) {
    alarmEnabled = TRUE;
    alarmCount++;
    printf("Alarm #%d\n", alarmCount);
}

State StateMachine(State *state, int func, LinkLayerRole role) {
    if ((func != 0 && role != -1) || (func == 0 && role == -1)) {
        return -1;
    } else {
        unsigned char control_field;
        unsigned char byte;
        while(*state != STOP_STATE && alarmEnabled == FALSE) {
            int bytes_R = readByteSerialPort(&byte);
            if(bytes_R > 0) {
                printf("0x%02X\n", byte);
                switch (*state) {
                    case START:
                        if (byte == FLAG) {
                            *state = FLAG_RCV;
                        } else {
```

```
                                printf("Unexpected byte, staying in
START\n");
                        }
                        break;


                    case FLAG_RCV:
                        if (byte == FLAG)  {
                                printf("FLAG received again, staying in
FLAG_RCV\n");
                            continue;
                        } else if (byte == A) {
                            *state = A_RCV;
                        } else {
                            *state = START;
                                  printf("Unexpected byte, returning to
START\n");
                        }
                        break;

                    case A_RCV:
                        if (byte == FLAG) {
                            *state = FLAG_RCV;
                                    printf("FLAG received, returning to
FLAG_RCV\n");
                         } else if (func == 0 && role == LlTx && byte ==
C_UA) {
                            control_field = byte;
                            *state = C_RCV;
                         } else if (func == 0 && role == LlRx && byte ==
C_SET) {
                            control_field = byte;
                            *state = C_RCV;
                          } else if (func == 1 && role == -1 && (byte ==
C_RR0 || byte == C_RR1 || byte == C_REJ0 || byte == C_REJ1 || byte ==
C_DISC)) {
                            control_field = byte;
                            *state = C_RCV;
                           } else if (func == 3 && role == -1 && byte ==
C_DISC) {
                            control_field = byte;
```

```
                            *state = C_RCV;
                        } else {
                            *state = START;
                                printf("Unexpected byte, returning to
START\n");
                        }
                        break;

                    case C_RCV:
                        if (byte == FLAG) {
                            *state = FLAG_RCV;
                                 printf("FLAG received, returning to
FLAG_RCV\n");
                        } else if (byte == (control_field ^ A)) {
                            *state = BCC1_OK;
                        } else {
                            *state = START;
                                printf("Unexpected byte, returning to
START\n");
                        }
                        break;

                    case BCC1_OK:
                        if (byte == FLAG) {
                            *state = STOP_STATE;
                        } else {
                            *state = START;
                                printf("Unexpected byte, returning to
START\n");
                        }
                        break;

                    default:
                        break;
                }
            }
        }
        return control_field;
    }
}
```

```
////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
                  if     (openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate) < 0) {
        return -1;
    }

    State state = START;
    timeout = connectionParameters.timeout;
    retransmissions = connectionParameters.nRetransmissions;

    switch (connectionParameters.role) {
        case LlTx:
            role = TX;
            (void)signal(SIGALRM, alarmHandler);
              while (connectionParameters.nRetransmissions > 0 && state
!= STOP_STATE) {
                  unsigned char buf_W1[5] = {FLAG, A, C_SET, A ^ C_SET,
FLAG};
                writeBytesSerialPort(buf_W1, 5);
                printf("SET frame sent.\n");
                u_frames++;

                alarm(connectionParameters.timeout);
                alarmEnabled = FALSE;


printf("-----------------------------------------------\n");
                StateMachine(&state, 0, LlTx);
                connectionParameters.nRetransmissions--;
            }
            if(connectionParameters.nRetransmissions <= 0) {
                printf("Maximum retransmissions sent. Aborting\n");

printf("-----------------------------------------------\n");
                return -1;
            }
```

```
            s_frames++;
            break;

        case LlRx:
            role = RX;
            printf("-----------------------------------------------\n");
            StateMachine(&state, 0, LlRx);
            unsigned char buf_W2[5] = {FLAG, A, C_UA, A ^ C_UA, FLAG};
            writeBytesSerialPort(buf_W2, 5);
            printf("UA frame sent.\n");
            u_frames++;
            s_frames++;
            break;

        default:
            return -1;
            break;
    }
    return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    int size = bufSize + 6;
    unsigned char* buf_W = malloc(2*size);

    buf_W[0] = FLAG;
    buf_W[1] = A;

    if (info == 0) {
        buf_W[2] = C_N0;
    } else {
        buf_W[2] = C_N1;
    }
    buf_W[3] = A ^ buf_W[2];

    info = !info;
    int data = 4;
```

```
unsigned char BCC2 = buf[0];
for(int i = 1; i < bufSize; i++) {
    BCC2 ^= buf[i];
}

for(int i = 0; i < bufSize; i++) {
    if(buf[i] == FLAG || buf[i] == ESCAPE) {
        buf_W[data++] = ESCAPE;
        buf_W[data++] = (buf[i]^0x20);
    } else {
        buf_W[data++] = buf[i];
    }
}

if(BCC2 == FLAG || BCC2 == ESCAPE) {
    buf_W[data++] = ESCAPE;
    buf_W[data++] = (BCC2^0x20);
} else {
    buf_W[data++] = BCC2;
}

buf_W[data++] = FLAG;

buf_W = realloc(buf_W, data);

State state = START;
int check = FALSE;
unsigned char control_field = 0;
int bytes_W = 0;

(void) signal(SIGALRM, alarmHandler);

while(retransmissions > 0 && state != STOP_STATE){

    bytes_W = writeBytesSerialPort(buf_W, data);
    i_frames++;
    printf("Packet sent\n");

    alarm(timeout);
    alarmEnabled = FALSE;
```

```
        printf("-----------------------------------------\n");

        sleep(1);

        control_field = StateMachine(&state, 1, -1);
        if(control_field == C_RR0 || control_field == C_RR1) {
            check = TRUE;
            s_frames++;
            break;
        } else if(control_field == C_REJ0 || control_field == C_REJ1) {
            s_frames++;
            continue;
        }
        retransmissions--;
        printf("Retransmissions availible: %d\n", retransmissions);
    }

    free(buf_W);

    if(retransmissions <= 0) {
        printf("Maximum retransmissions sent. Aborting\n");
        printf("-----------------------------------------\n");
        check = FALSE;
    }

    if(check) {
        return bytes_W;
    } else {
        llclose(0);
    }
    return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    State state = START;
    unsigned char byte;
```

```
    unsigned char control_field;
    unsigned char last_control_field = 0x7e;
    int i = 0;
    printf("-----------------------------------------------\n");
    while (state != STOP_STATE) {
        int bytes_R = readByteSerialPort(&byte);
        if(bytes_R > 0) {
            //printf("0x%02X\n", byte);
            switch (state) {
                case START:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                    } else {
                        printf("Unexpected byte, staying in START\n");
                    }
                    break;

                case FLAG_RCV:
                    if (byte == FLAG)  {
                                printf("FLAG received again, staying in
FLAG_RCV\n");
                        continue;
                    } else if (byte == A) {
                        state = A_RCV;
                    } else {
                        state = START;
                                printf("Unexpected byte, returning to
START\n");
                    }
                    break;

                case A_RCV:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                                printf("FLAG received, returning to
FLAG_RCV\n");
                    } else if (byte == C_N0 || byte == C_N1 || byte ==
C_DISC) {
                        state = C_RCV;
                        control_field = byte;
                        i_frames++;
```

```
                        if(control_field == last_control_field) {
                            dup_frames++;
                            printf("Duplicated frame detected\n");
                        }
                    } else {
                        state = START;
                                printf("Unexpected byte, returning to
START\n");
                    }
                    break;

                case C_RCV:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                                  printf("FLAG received, returning to
FLAG_RCV\n");
                    } else if (byte == (A ^ control_field)) {
                        if (control_field ==  C_DISC) {
                            u_frames++;
                             unsigned char buf_W[5] = {FLAG, A_, C_DISC,
A_ ^ C_DISC, FLAG};
                            writeBytesSerialPort(buf_W, 5);
                            return 0;
                        }
                        state = BCC1_OK;
                    } else {
                        state = START;
                                printf("Unexpected byte, returning to
START\n");
                    }
                    break;

                case BCC1_OK:
                    if (byte == ESCAPE) {
                        //printf("ESC received, reading next byte\n");
                        readByteSerialPort(&byte);
                        packet[i++] = byte ^ 0x20;
                        //printf("After ESC: 0x%02X\n", packet[i - 1]);

                    } else if (byte == FLAG){
                        printf("Packet received\n");
```

```
                                //printf("Checking BCC2 and processing
packet\n");
                        unsigned char bcc2 = packet[--i];
                        packet[i] = '\0';
                        unsigned char acc = 0;
                        for (unsigned int j = 0; j < i; j++) {
                            acc ^= packet[j];
                        }
                        if (bcc2 == acc){
                            printf("BCC2 check passed, sending RR\n");
                            state = STOP_STATE;
                            unsigned char rr;
                            if (control_field == C_N0) {
                                rr = C_RR0;
                                  printf("Control Field is C_N0, sending
RR0\n");
                            } else {
                                rr = C_RR1;
                                  printf("Control Field is C_N1, sending
RR1\n");
                            }
                            s_frames++;
                              unsigned char buf_W[5] = {FLAG, A, rr, A ^
rr, FLAG};
                            writeBytesSerialPort(buf_W, 5);
                            sleep(1);
                            return packet[0];
                        } else {
                            printf("BCC2 check failed, sending REJ\n");
                            unsigned char rej;
                            if (control_field == C_N0) {
                                rej = C_REJ0;
                                  printf("Control Field is C_N0, sending
REJ0\n");
                            } else {
                                rej = C_REJ1;
                                  printf("Control Field is C_N1, sending
REJ1\n");
                            }
                            rej_frames++;
                            s_frames++;
```

```
                            unsigned char buf_W[5] = {FLAG, A, rej, A ^
rej, FLAG};
                            writeBytesSerialPort(buf_W, 5);
                            sleep(1);
                            return packet[0];
                        }
                    } else {
                        packet[i++] = byte;
                    }
                    break;
                default:
                    break;
            }
        }
    }
    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    State state = START;

    (void) signal(SIGALRM, alarmHandler);

    while(retransmissions > 0 && state != STOP_STATE) {
        unsigned char buf_W[5] = {FLAG, A, C_DISC, A ^ C_DISC, FLAG};
        writeBytesSerialPort(buf_W, 5);

        alarm(timeout);
        alarmEnabled = FALSE;

        u_frames++;
        StateMachine(&state, 3, -1);
        retransmissions--;

    }
    if(state != STOP_STATE) {
        return -1;
```

```
    }

    unsigned char buf_W[5] = {FLAG, A_, C_UA, A_ ^ C_UA, FLAG};
    writeBytesSerialPort(buf_W, 5);
    u_frames++;

    printf("-----------------------------------------------\n");

    switch(role) {
        case TX:
            printf("I frames sent: %d\n", i_frames);
            printf("U frames sent: %d\n", u_frames);
            printf("S frames received: %d\n", s_frames);
            break;

        case RX:
            printf("I frames received: %d\n", i_frames);
            printf("U frames received: %d\n", u_frames);
            printf("S frames sent: %d\n", s_frames);
            printf("Duplicated frames received: %d\n", dup_frames);
            printf("Rejected frames: %d\n", rej_frames);
            break;
    }

    printf("-----------------------------------------------\n");

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

Apêndice II - application_layer.c

```
// Application layer protocol implementation

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

#define DATA_PACKET 1
#define START_PACKET 2
#define END_PACKET 3
#define SIZE_FIELD 0
#define NAME_FIELD 1

int sendControlPacket(int packetType, const char* fileName, long
fileSize) {
    size_t fileNameLength = strlen(fileName) + 1;
      size_t fileSizeBytes = (fileSize > 0) ? ((sizeof(long) * 8 -
__builtin_clzl(fileSize) + 7) / 8) : 1;
    size_t packetSize = 5  + fileNameLength + fileSizeBytes;
    unsigned char* controlPacket = (unsigned char*)malloc(packetSize);
    int i = 0;
    controlPacket[i++] = packetType;
    controlPacket[i++] = SIZE_FIELD;

    for (size_t j = 0; j < sizeof(size_t); j++) {
        controlPacket[i + j] = (fileSizeBytes >> (j * 8)) & 0xFF;
    }
    i += sizeof(size_t);
    controlPacket[i++] = NAME_FIELD;
    memcpy(controlPacket + i, fileName, fileNameLength);

    if (llwrite(controlPacket, packetSize) < 0) {
        printf("Failed to send control packet\n");
        free(controlPacket);
        return -1;
    }
    free(controlPacket);
    return 1;
```

```
}

int readControlPacket(unsigned char expectedPacketType, unsigned char*
buffer, size_t* fileSize, char* fileName) {
    int bufferSize;
    if ((bufferSize = llread(buffer)) < 0) {
        printf("Error reading control packet\n");
        return -1;
    }
    if (buffer[0] != expectedPacketType) {
        printf("Invalid control packet\n");
        return -1;
    }
    int i = 1;
    unsigned char fieldType;
    while (i < bufferSize) {
        fieldType = buffer[i++];
        if (fieldType == NAME_FIELD) {
            size_t nameLength = buffer[i++];
            memcpy(fileName, &buffer[i], nameLength);
            fileName[nameLength] = '\0';
            i += nameLength;
        }
        else if (fieldType == SIZE_FIELD) {
            *fileSize = 0;
            for (size_t j = 0; j < sizeof(size_t); j++) {
                *fileSize |= ((size_t)buffer[i++] << (j * 8));
            }
        } else {
            printf("Invalid control packet type\n");
            return -1;
        }
    }
    return 1;
}

int sendDataPacket(unsigned char* data, int dataSize) {
    size_t packetSize = dataSize + 3;
    unsigned char* packet = (unsigned char*)malloc(packetSize);
    if (packet == NULL) {
        printf("Memory allocation failed\n");
```

```
        return -1;
    }

    packet[0] = DATA_PACKET;
    packet[1] = (dataSize >> 8) & 0xFF;
    packet[2] = dataSize & 0xFF;
    memcpy(packet + 3, data, dataSize);

    int status = llwrite(packet, packetSize);
    free(packet);

    if (status < 0) {
        printf("Error sending data packet\n");
        return -1;
    }

    return 0;
}

void applicationLayer(const char *serialPort, const char *role, int baudRate, int maxRetries, int timeout, const char *filename) {
    LinkLayer ll;
    sprintf(ll.serialPort, "%s", serialPort);
    ll.role = strcmp(role, "tx") ? LlRx : LlTx;
    ll.baudRate = baudRate;
    ll.nRetransmissions = maxRetries;
    ll.timeout = timeout;

    if (llopen(ll) == -1) {
        perror("Error opening link layer\n");
        exit(1);
    }

    switch (ll.role) {
        case LlRx: {
            size_t fileSize;
            char receivedFileName[0xFF];
                                unsigned char* buffer = (unsigned char*)malloc(MAX_PAYLOAD_SIZE);

            if (buffer == NULL) {
```

```
                printf("Memory allocation failed\n");
                exit(-1);
            }

                if (readControlPacket(START_PACKET, buffer, &fileSize,
receivedFileName) < 0) {
                printf("Error reading control packet\n");
                free(buffer);
                exit(-1);
            }

            printf("Start packet received\n");

            FILE* fileOutput = fopen((char*)filename, "wb+");
            if (fileOutput == NULL) {
                printf("Error opening file\n");
                free(buffer);
                exit(-1);
            }

            printf("File opened for writing\n");
            int dataSize;

            while ((dataSize = llread(buffer)) >= 0) {
                if (dataSize == 0) continue;

                if (buffer[0] == END_PACKET) {
                    printf("End packet received\n");
                    break;
                }

                    fwrite(buffer + 3, 1, (buffer[1] << 8) | buffer[2],
fileOutput);
            }

            free(buffer);
            fclose(fileOutput);
            printf("File closed\n");

            if (llclose(TRUE) < 0) {
                printf("Error closing connection\n");
```

```
                exit(-1);
            }

            printf("Connection closed\n");
            break;
        }

        case LlTx: {
            FILE* file = fopen(filename, "rb");
            if (file == NULL) {
                printf("Error opening file for reading\n");
                exit(-1);
            }

            fseek(file, 0, SEEK_END);
            long fileSize = ftell(file);
            fseek(file, 0, SEEK_SET);

            printf("Sending start packet\n");

              if (sendControlPacket(START_PACKET, filename, fileSize) <
0) {
                printf("Error sending start packet\n");
                fclose(file);
                exit(-1);
            }

                                unsigned   char*   buffer   =   (unsigned
char*)malloc(MAX_PAYLOAD_SIZE - 3);
            if (buffer == NULL) {
                printf("Memory allocation failed\n");
                fclose(file);
                exit(-1);
            }

            int dataSize;
              while ((dataSize = fread(buffer, 1, MAX_PAYLOAD_SIZE - 3,
file)) > 0) {
                if (sendDataPacket(buffer, dataSize) < 0) {
                    printf("Error sending data packet\n");
                    free(buffer);
```

```
                    fclose(file);
                    exit(-1);
                }
            }

            free(buffer);
            fclose(file);

             if (sendControlPacket(END_PACKET, filename, fileSize) < 0)
{
                printf("Error sending end packet\n");
                exit(-1);
            }

            printf("End packet sent\n");

            if (llclose(1) < 0) {
                printf("Error closing connection\n");
                exit(-1);
            }

            printf("Connection closed\n");
            break;
        }

        default:
            printf("Invalid role specified\n");
            break;
    }
}
```

Apêndice III - serial_port.c

```c
// Serial port interface implementation
// DO NOT CHANGE THIS FILE

#include "serial_port.h"

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

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

int fd = -1;           // File descriptor for open serial port
struct termios oldtio; // Serial port settings to restore on closing

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int baudRate)
{
    // Open with O_NONBLOCK to avoid hanging when CLOCAL
    // is not yet set on the serial port (changed later)
    int oflags = O_RDWR | O_NOCTTY | O_NONBLOCK;
    fd = open(serialPort, oflags);
    if (fd < 0)
    {
        perror(serialPort);
        return -1;
    }

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

```
    // Convert baud rate to appropriate flag
    tcflag_t br;
    switch (baudRate)
    {
    case 1200:
        br = B1200;
        break;
    case 1800:
        br = B1800;
        break;
    case 2400:
        br = B2400;
        break;
    case 4800:
        br = B4800;
        break;
    case 9600:
        br = B9600;
        break;
    case 19200:
        br = B19200;
        break;
    case 38400:
        br = B38400;
        break;
    case 57600:
        br = B57600;
        break;
    case 115200:
        br = B115200;
        break;
    default:
          fprintf(stderr, "Unsupported baud rate (must be one of 1200,
1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200)\n");
        return -1;
    }

    // New port settings
    struct termios newtio;
    memset(&newtio, 0, sizeof(newtio));
```

```
    newtio.c_cflag = br | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 1; // Block reading
    newtio.c_cc[VMIN] = 0;  // Byte by byte

    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        close(fd);
        return -1;
    }

    // Clear O_NONBLOCK flag to ensure blocking reads
    oflags ^= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, oflags) == -1)
    {
        perror("fcntl");
        close(fd);
        return -1;
    }

    // Done
    return fd;
}

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort()
{
    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
```

```
        return -1;
    }

    return close(fd);
}

// Wait up to 0.1 second (VTIME) for a byte received from the serial port (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was received.
int readByteSerialPort(unsigned char *byte)
{
    return read(fd, byte, 1);
}

// Write up to numBytes to the serial port (must check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of bytes written.
int writeBytesSerialPort(const unsigned char *bytes, int numBytes)
{
    return write(fd, bytes, numBytes);
}
```