Faculdade de Engenharia da Universidade do Porto

Redes de Computadores

1º Trabalho Laboratorial

Turma 11 - Grupo 1

João Lima (up 202108891@up.pt)

Rodrigo Rodrigues (up202108749@up.pt)

Porto, 10 de novembro de 2023

Sumário

O nosso trabalho realizado no âmbito da unidade curricular Redes de Computadores tem como principal objetivo implementar um protocolo de comunicação de dados para a transmissão de ficheiros utilizando a Porta Série RS-232.

Através da realização deste projeto, foi possível implementar o protocolo em questão consolidando assim o nosso conhecimento sobre o funcionamento da estratégia *Stop-and-Wait.*

Introdução

O principal objetivo do trabalho, como já foi anteriormente referido, foi desenvolver um protocolo de ligação de dados que permitisse transferir um ficheiro a partir da porta de série. O nosso relatório está dividido em 8 pontos:

- **Arquitetura:** Blocos funcionais e interfaces utilizadas
- **Estrutura do código:** APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura
- **Casos de uso principais:** Indentificação dos funcionamentos do projeto e da sequência de chamadas das funções
- **Protocolo de ligação lógica:** Funcionamento da ligação lógica e estratégias de implementação
- **Protocolo de aplicação:** Funcionamento da camada de aplicação e estratégias de implementação
- **Validação:** Descrição dos testes efetuados para avaliar a implementação
- **Eficiência do protocolo de ligação de dados:** Quantização da eficiência do protocolo
- **Conclusões:** Síntese da informação apresentada e reflexão sobre os objetivos do trabalho

Arquitetura

Relativamente à arquitetura, o programa está estruturado em duas camadas, a *ApplicationLayer* e a *LinkLayer.*

A LinkLayer consiste na implementação do protocolo e está presente nos ficheiros link_layer.h e link_layer.c. Esta camada acaba por ser responsável pelo estabelecimento e terminação da ligação de dados, criação e envio de tramas de informação através da porta de série e por fim a validação das tramas recebidas e o envio de mensagens de erro caso tenha ocorrido algum problema na transmissão.

A ApplicationLayer foi implementada nos ficheiros application_layer.h e application_layer.c e utiliza a API da LinkLayer para a transferência e receção de pacotes de dados constituintes de um determinado ficheiro. Por isso, esta acaba por ser a camada mais próxima do utilizador e é nela onde é possível definir o tamanho das tramas de informação, a velocidade da transferência e o número máximo de retransmissões.

- **Interface**

```
Modo Transmissor:
        $ make run_tx

Modo Recetor:
        $ make run_rx
```

Estrutura do Código

- **ApplicationLayer**

Na implementação desta camada não foi necessária a criação de estruturas de dados auxiliares. Estas foram as funções implementadas:

```
// Principal função da camada
void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename);

//Criação de um pacote de controlo
unsigned char *getControl_packet(int control, const char *filename, long int fileSize, int *cpsize);

//Criação de um pacote de dados
unsigned char *getData(unsigned char *data, int dataSize, int *packetSize);

// Informação sobre  uma data packet recebida, incluind o tamanho e a representação hexadecimal do seu conteúdo.
void printPacket(const unsigned char *packet, int packetSize);

// Interpretação de um pacote de controlo
char *parseControlPacket(unsigned char* packet, long int *fileSize, const char *appendix);
```

- **LinkLayer**

Na implementação desta camada já foi necessário criar estruturas de dados auxiliares: LinkLayer, onde são caracterizados os parâmetros associados à transferência de dados; LinkLayerRole, que identifica se o computador é um transmissor ou um recetor. Para além disto, foi também necessária a criação de Macros específicas que representam os diferentes estados da leitura e receção das tramas de informação.

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

```
typedef struct
{
   char* serialPort;
   LinkLayerRole role;
   int baudRate;
   int nRetransmissions;
   int timeout;

}LinkLayer;
```

As macros utilizadas nesta camada foram:

```
// MACRO do número máximo de bytes por pacote
#define MAX_PAYLOAD_SIZE 1000

// STATE MACHINE
#define STATE_START 0 define MAX_PAYLOAD_SIZE 1000
#define STATE_RCV 1
#define STATE_A_RCV 2
#define STATE_C_RCV 3
#define STATE_DATA 4
#define STATE_ESC_FOUND 5
#define STATE_BCC_OK 6
#define STATE_STOP 7

//Macro da Flag
#define FLAG 0x7E

// MACROS relativas ao procedimento de Byte Stuffing
#define ESC 0x7D
#define AFTER_ESC 0x5D
#define AFTER_FLAG 0x5E

// MACROS do byte de endereço
#define ADDRESS_S 0x03
#define ADDRESS_R 0x01

// MACROS do byte de controlo
#define CONTROL_SET 0x03
#define CONTROL_UA 0x07
#define CONTROL_RR(N) ((N << 7) | 0x05)
#define CONTROL_REJ(N) ((N << 7) | 0x01)
#define CONTROL_INF(N) (N << 6)
#define CONTROL_DISC 0x0B
```

As funções implementadas foram:

```
// Open a connection using the "port" parameters defined in struct linkLayer.
int llopen(LinkLayer connectionParameters);

// Envio de tramas
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receção de tramas
int llread(int fd, unsigned char *packet);
// Termino da ligação
int llclose(int showStatistics);
```

```
// Configuração do alarme
void alarmHandler(int signal);

// Receção e interpretação de tramas de supervisão
unsigned char check_control(int fd, unsigned char Address);

// Configuração da porta série
int openSerialPort(const char *serialPort, int baudRate);
```

Casos de uso principais

Como já foi referido anteriormente, o programa pode ser executado nos modos transmissor e recetor. Por isso, as funções usadas e a sequência de chamadas dependerão do modo.

- **Transmissor**
  1. **llopen**(), cria a ponte de ligação entre o transmissor e o recetor e trata da conexão com a porta série;
  2. **getControlPacket**(), cria um pacote de controlo;
  3. getData(), retorna um segmento do ficheiro através do pacote de dados passado como argumento;
  4. **llwrite**(), cria e envia pela porta série uma trama de informação com base no pacote recebido;
  5. **check_control**(), é feita a leitura e a validação da resposta do recetor através de uma máquina de estados;
  6. **llclose**(), termina a ligação entre o transmissor e o recetor.

- **Recetor**
  1. **llopen**();
  2. **parseControlPacket**(), retorna as características do ficheiro a ser transferido contidas no pacote de controlo no formato TLV;
  3. **llread**(), lê e valida a receção de tramas de controlo e informação
  4. **sendFrame**(), cria e envia uma trama de supervisão de acordo com a trama lida por llread();
  5. **llclose**().

Protocolo de Ligação lógica

A camada de ligação de dados é aquela que interage principalmente com a porta série, sendo responsável pela comunicação entre o emissor e o recetor.

O estabelecimento da ligação, é tratado pela função llopen(). Inicialmente, há a abertura da porta e a sua respetiva configuração. Após este primeiro passo estar feito, o emissor vai enviar uma trama SET que vai ser lido pelo recetor. Quando o recetor receber a trama SET, vai enviar uma trama UA de maneira que o transmissor saiba que a mensagem foi recebida. Se o transmissor receber a trama UA, a ligação entre eles fica estabelecida. Após este espaço estar concluído, o transmissor irá passar a enviar informação para o recetor.

O envio de informação é responsabilidade da função llwrite(). Esta função recebe um pacote de dados ou de controlo e aplica-lhe a técnica de byte stuffing de modo a evitar conflitos com os bytes de dados que sejam iguais às flags das tramas. Para além disso, transforma esse pacote numa trama (framing), envia-a para o recetor e espera pela sua resposta. Se a trama for rejeitada, o envio é realizado novamente até ser aceite ou exceder o número máximo de tentativas (cada tentativa tem um número limite, timeout).

A leitura de informação por parte do recetor é feita através da função llread(). Esta lê a informação recebida pela porta série e faz a validação da mesma. Inicialmente faz destuffing do campo de dados da trama (enviada pelo transmissor) e vai validar o BCC1 e BCC2. Caso haja algum erro, envia uma trama REJ para o transmissor.

O término da ligação é responsabilidade da função llclose(). Esta apenas vai ser invocada pelo emissor se o número de tentativas fracassadas for excedido ou quando a transferência de dados estiver concluída. Para tal, o emissor enviará uma trama de supervisão DISC e ficará à espera de que o recetor mande o mesmo. Quando o transmissor receber a resposta do recetor, este envia uma trama UA e a ligação de dados é terminada.

Protocolo de Aplicação

A camada de aplicação é aquela que interage diretamente com o utilizador. Nela podemos definir o ficheiro que quisermos transferir, a porta série, a velocidade da transferência, o número de bytes de dados do ficheiro inseridos em cada pacote (MAXPAYLOAD), o número máximo de retransmissões e o tempo máximo que o transmissor espera por uma resposta do recetor. Esta transferência vai ser realizada através da API da LinkLayer.

Após a ligação entre o emissor e o recetor estar estabelecida, todo o conteúdo do ficheiro vai ser copiado para um buffer local. O primeiro pacote a ser enviado pelo transmissor tem dados em formato TLV, criado pela função getControl_packet, onde são expressos o tamanho do ficheiro em bytes e o nome do ficheiro. No lado do recetor, o pacote vai ser desempacotado pela função parseControlPacket de forma a criar e alocar o espaço necessário para receber o ficheiro.

Cada fragmento de ficheiro vai ser inserido num pacote que vai conter os dados através da função getData e vai ser posteriormente enviado pela porta série usando a função llwrite. Para além disso, por cada envio, o recetor diz se aceita ou rejeita o pacote. Se aceitar, o transmissor manda o fragmento seguinte, senão, volta a enviar o mesmo fragmento. Depois cada pacote será recebido/avaliado pelo recetor através das funções llread.

Quando o ficheiro acabar de ser transmitido pelo transmissor (ou quando o número máximo de tentativas fracassadas for ultrapassado), a camada da aplicação irá invocar função llclose para terminar a ligação entre as duas máquinas.

Validação

Durante o desenvolvimento do programa, foram conduzidos testes abrangentes para garantir a integridade e consistência do protocolo implementado. Os testes incluíram:

- ✓ Transferência de arquivos com diferentes nomes.
- ✓ Transferência de arquivos com tamanhos variados.
- ✓ Transferência de arquivos com velocidades de transferência distintas.
- ✓ Transferência de arquivos com pacotes de dados de tamanhos diversos.
- ✓ Transferência de arquivos com simulação de interrupção parcial e/ou total da porta serial.

Ao longo do processo de desenvolvimento, foram aplicados testes unitários para validar cada aspeto específico da aplicação. Os resultados dos testes foram positivos, demonstrando um desempenho bem-sucedido durante a apresentação do trabalho.

Conclusões

O desenvolvimento do protocolo seguiu estritamente as especificações fornecidas. O protocolo de ligação lógica desempenha um papel crucial na preparação da conexão da porta serial e na gestão de erros, enquanto o protocolo de aplicação é responsável por interpretar as informações essenciais para a execução do programa, além de lidar com a leitura e escrita do conteúdo do arquivo. Este projeto permitiu a consolidação dos conceitos fundamentais, como Byte Stuffing, Framing, e o entendimento do funcionamento do protocolo Stop-and-Wait, incluindo sua capacidade de detecção e tratamento de erros.

Anexo I – link_layer.h

```
// Link layer header.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

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

typedef enum
{
   LlTx,
   LlRx,
} LinkLayerRole;

typedef struct
{
   char *serialPort;
   LinkLayerRole role;
   int baudRate;
   int nRetransmissions;
   int timeout;
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

void alarmHandler(int signal);
unsigned char check_control(int fd, unsigned char Address);
int openSerialPort(const char *serialPort, int baudRate);


// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);
```

```
// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(int fd, const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(int fd, unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

#endif // _LINK_LAYER_H_
```

Anexo II – link_layer.c

```
// Link layer protocol implementation

#include "../include/link_layer.h"

int alarmCount = 0;
int alarmEnabled = FALSE;
int baudRate = 0;
int nRetransmissions = 0;
int timeout = 0;
unsigned char tramaTx = 0;
unsigned char tramaRx = 0;
LinkLayer connection;


struct termios oldtio;
struct termios newtio;

void printFrame(const unsigned char *packet, int packetSize) {
   printf("Received packet with size %d:\n", packetSize);
   printf("Contents: { ");
   for (int i = 0; i < packetSize; i++) {
      printf("%02X ", packet[i]);
   }
   printf("}\n\n");
}

void alarmHandler(int signal)
{
   alarmEnabled = FALSE;
   alarmCount--;
```

```
   printf("--- Alarm #%d ---\n", alarmCount);
}

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

int sendFrame(int fd, unsigned char Address, unsigned char control) {
   unsigned char frame[5] = {FLAG, Address, control, Address ^ control, FLAG};
   return write(fd,&frame,5);

}



unsigned char check_control(int fd, unsigned char Address) {

   unsigned char byte, control = 0xFF;
   int state = STATE_START;
   unsigned char previous_alarm_status = alarmEnabled;
   alarmEnabled = TRUE;

   while(state != STATE_STOP && alarmEnabled == TRUE) {

      if(read(fd,&byte,1) > 0) {
         switch(state) {
            case STATE_START:
               if(byte == FLAG) state = STATE_RCV;
               else state = STATE_START;
               break;
            case STATE_RCV:
               if(byte == Address) state=STATE_A_RCV;
               else state = STATE_START;
               break;
            case STATE_A_RCV:
               if(byte == CONTROL_SET || byte == CONTROL_UA || byte == CONTROL_DISC ||
byte == CONTROL_RR(0) || byte == CONTROL_RR(1) || byte == CONTROL_REJ(0) || byte ==
CONTROL_REJ(1)) {

                  control = byte;
                  state = STATE_C_RCV;

               }
               else if (byte == FLAG) state = STATE_RCV;
               else state = STATE_START;
               break;
            case STATE_C_RCV:
               if(byte == (Address ^ control)) state = STATE_BCC_OK;
               else if (byte == FLAG) state = STATE_RCV;
               else state = STATE_START;
               break;
            case STATE_BCC_OK:
               if(byte == FLAG) {
```

```
                if (previous_alarm_status == TRUE) alarm(0);
                alarmEnabled = FALSE;
                state = STATE_STOP;
              }
              else state = STATE_START;
              break;
            default:
              break;
          }
        }
    }

    return control;
}

int openSerialPort(const char *serialPort, int baudRate)
{
    int fd = open(serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0) return -1;


    if (tcgetattr(fd, &oldtio) == -1) return -1;


    memset(&newtio, 0, sizeof(newtio));
    newtio.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 5; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0;  // Read without blocking
    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) return -1;


    return fd;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    connection = connectionParameters;
    connection.role = connectionParameters.role;
    baudRate = connectionParameters.baudRate;
    timeout = connectionParameters.timeout;
    nRetransmissions = connectionParameters.nRetransmissions;
    alarmCount = nRetransmissions;
```

```
int fd = openSerialPort(connectionParameters.serialPort, connectionParameters.baudRate);
if (fd < 0) return -1;



int state = STATE_START;
unsigned char byte;
switch(connectionParameters.role) {
   case LlTx:
   (void)signal(SIGALRM, alarmHandler);

   while (1){

      if(alarmCount == 0) return -1;

      if (alarmEnabled == FALSE){
         sendFrame(fd,ADDRESS_S,CONTROL_SET);

         alarm(timeout);
         alarmEnabled = TRUE;
      }

      if (check_control(fd,ADDRESS_R) == CONTROL_UA) {
         break;
      }

      }
      break;

   case LlRx:

         while (state != STATE_STOP){

      if(read(fd, &byte, 1) > 0) {
         switch(state) {
         case STATE_START:

         if (byte == FLAG) state = STATE_RCV;

         break;
         case STATE_RCV:

         if (byte == ADDRESS_S) state = STATE_A_RCV;
         else if (byte == FLAG) state = STATE_RCV;
         else state = STATE_START;

         break;
         case STATE_A_RCV:

         if (byte == CONTROL_SET) state = STATE_C_RCV;
         else if (byte == FLAG) state = STATE_RCV;
```

```
            else state = STATE_START;

            break;
            case STATE_C_RCV:

            if (byte == (ADDRESS_S ^ CONTROL_SET)) state = STATE_BCC_OK;
            else if (byte == FLAG) state = STATE_RCV;
            else state = STATE_START;

            break;
            case STATE_BCC_OK:
            if (byte == FLAG)
            {
               state = STATE_STOP;
               sendFrame(fd,ADDRESS_R,CONTROL_UA);
            }
            else state = STATE_START;
            break;

            default:
            return -1;
                }
              }
            }
   }
   return fd;
}
////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(int fd, const unsigned char *buf, int bufSize){
   //printf("Packet to send:\n\n");
   //printf("{ ");
   //for (int i = 0; i < bufSize; i++)
      //printf("%02X ", buf[i]);
   //printf("}\n\n");

   int frameSize = bufSize + 6; //{FLAG, A, C, BCC1} + Data size + {BCC2, FLAG}
   unsigned char *frame = malloc(frameSize);

   frame[0] = FLAG;
   frame[1] = ADDRESS_S;
   frame[2] = CONTROL_INF(tramaTx);
   frame[3] = frame[1] ^ frame[2];

   memcpy(frame + 4, buf, bufSize); //passar a data para o frame

   unsigned char BCC2 = buf[0];
   for(int i = 1; i < bufSize; i++) BCC2 ^= buf[i]; // calcular o bcc2

   for(int i = 4; i < frameSize - 2; i++) { //byte stuffing
      if(frame[i] == FLAG) {
```

```
      unsigned char byte1 = 0x7D;
      unsigned char byte2 = 0x5E;

          frame = realloc(frame,++frameSize);
      memmove(frame + i + 1, frame + i, frameSize - i - 1);

      frame[i++] = byte1;
      frame[i] = byte2;
    }
    else if (frame[i] == ESC) {
      unsigned char byte1 = 0x7D;
      unsigned char byte2 = 0x5D;

          frame = realloc(frame,++frameSize);
      memmove(frame + i + 1, frame + i, frameSize - i - 1);

      frame[i++] = byte1;
      frame[i] = byte2;
    }
}

frame[frameSize-2] = BCC2;
frame[frameSize-1] = FLAG;

//printf("Frame after-stuffing:\n\n");
//printf("{ ");
//for (int i = 0; i < frameSize; i++)
   //printf("%02X ", frame[i]);
//printf("} %d\n\n", frameSize);

int state = STATE_START;
alarmCount = nRetransmissions;
unsigned char Control;

//printf("Alarm count - %d  \n ", alarmCount);

(void)signal(SIGALRM, alarmHandler);

while (state != STATE_STOP && alarmCount != 0){
   if(alarmEnabled == FALSE) {

      write(fd,frame,frameSize);
      alarm(timeout);
      alarmEnabled = TRUE;
   }

   //check control
   Control = check_control(fd,ADDRESS_R);

   if (Control == CONTROL_RR(0) || Control == CONTROL_RR(1))
   {
      state = STATE_STOP;
```

```
            tramaTx = (tramaTx+1) % 2;
        }

        else if(Control == CONTROL_REJ(0) || Control == CONTROL_REJ(1)) {
            alarm(0);
            alarmEnabled = FALSE;
        }
    }

    if (state != STATE_STOP) return -1;

    free(frame);
    return frameSize;
}


////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(int fd, unsigned char *packet) {

    int state = STATE_START;
    unsigned char byte;
    int packetIndex = 0;
    unsigned char control = 0; // Variável para armazenar o byte de controle
    unsigned char bcc2 = 0;    // Variável para calcular BCC2
    unsigned char acc = 0;     // Variável para verificar BCC2

    while (state != STATE_STOP) {
        if (read(fd, &byte, 1) > 0) {
            switch (state) {
                case STATE_START:
                    if (byte == FLAG) {
                        state = STATE_RCV;
                    }

                    break;
                case STATE_RCV:
                    if (byte == ADDRESS_S) {
                        state = STATE_A_RCV;
                    } else if (byte != FLAG) {
                        state = STATE_START;
                    }

                    break;
                case STATE_A_RCV:

                    if (byte == CONTROL_INF(0) || byte == CONTROL_INF(1)) {
                        control = byte; // Armazena o byte de controle
                        state = STATE_C_RCV;
                    }
                    else if (byte == CONTROL_DISC) {
```

```
        // Enviar mensagem de desconexão
        // Após o envio, retorne 0 para indicar desconexão
        sendFrame(fd, ADDRESS_R, CONTROL_DISC);
        return 0;
    }
    else if (byte == FLAG) {
        state = STATE_RCV;
    }
    else state = STATE_START;

    break;

case STATE_C_RCV:

    if (byte == (ADDRESS_S ^ control)) {
        state = STATE_DATA;
    } else if (byte == FLAG) {
        state = STATE_RCV;
    }
    else{
        state = STATE_START;
    }
    break;
case STATE_DATA:

    if (byte == ESC) state = STATE_ESC_FOUND;
    else if (byte == FLAG) {
        bcc2 = packet[packetIndex - 1];
        packetIndex--;
        acc = packet[0];

        for (unsigned int i = 1; i < packetIndex; i++)
        {
            acc ^= packet[i];
        }

        // Desempacotamento bem-sucedido
        if (bcc2 == acc) {

            state = STATE_STOP;
            sendFrame(fd, ADDRESS_R, CONTROL_RR(tramaRx));
            tramaRx = (tramaRx +1) %2;

            return packetIndex;
        } else {
            // Erro de verificação do BCC2, envie REJ
            sendFrame(fd,ADDRESS_R,CONTROL_REJ(tramaRx));
            return -1; // Retorna erro
        }
    }
    else {
        // Adicione o byte ao pacote desempacotado
```

```
                packet[packetIndex++] = byte;
            }
            break;
        case STATE_ESC_FOUND:
            printf("ESC found \n");
            state = STATE_DATA;
            if (byte == AFTER_ESC) {
                packet[packetIndex++] = ESC;
            }
            else if(byte == AFTER_FLAG) {
                packet[packetIndex++] = FLAG;
            }
            else {
                packet[packetIndex++] = ESC;
                packet[packetIndex++] = byte;
            }
            break;
        default:
            break;
      }
    }
  }
  return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int fd)
{
  switch(connection.role) {
    case LlTx:
      alarmCount = nRetransmissions;

      (void)signal(SIGALRM, alarmHandler);
      while(1) {

        if(alarmCount == 0) return -1;

        if(alarmEnabled == FALSE){
          sendFrame(fd, ADDRESS_S,CONTROL_DISC);
          alarm(timeout);
          alarmEnabled = TRUE;
        }

        if(check_control(fd,ADDRESS_R) == CONTROL_DISC) {
          break;
        }
      }
      sendFrame(fd,ADDRESS_S, CONTROL_UA);
      break;
```

```
        case LlRx:
            alarmCount = nRetransmissions;

            while(check_control(fd,ADDRESS_S) != CONTROL_DISC);

            (void)signal(SIGALRM, alarmHandler);
            while(1) {

                if(alarmCount == 0) return -1;

                if(alarmEnabled == FALSE){
                    sendFrame(fd, ADDRESS_R,CONTROL_DISC);
                    alarm(timeout);
                    alarmEnabled = TRUE;
                }

                if(check_control(fd,ADDRESS_S) == CONTROL_UA) {
                    break;
                }
            }
            break;
        default:
            return -1;
            break;
    }
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1) return -1;

    return close(fd);
}
```

Anexo III – application_layer.h

```
// Application layer protocol header.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);
unsigned char *getControl_packet(int control, const char *filename, long int fileSize, int
*cpsize);
unsigned char *getData(unsigned char *data, int dataSize, int *packetSize);
void printPacket(const unsigned char *packet, int packetSize);
```

```
char *parseControlPacket(unsigned char* packet, long int *fileSize, const char *appendix);
#endif // _APPLICATION_LAYER_H_
```

Anexo IV – application_layer.c

```
// Application layer protocol implementation

#include "../include/application_layer.h"
#include "../include/link_layer.h"

const char *Appendix = "-received";

char *getNewFilename(const char *filename, const char *appendix, int fileNameSize)
{
   const char *dotPosition = strrchr(filename, '.');

   char *newFilename = (char *) malloc(fileNameSize);

   strncpy(newFilename, filename, (int)(dotPosition - filename));

   newFilename[(int)(dotPosition - filename)] = '\0';

   strcat(newFilename, appendix);

   strcat(newFilename, dotPosition);

   return newFilename;
}


char *parseControlPacket(unsigned char* packet, long int *fileSize, const char *appendix)
{
   unsigned char size_bytes = packet[2];
   unsigned char name_bytes = packet[4 + size_bytes];

   for (int i = size_bytes - 1; i >= 0; i--)
      *fileSize = (*fileSize << 8) | packet[3 + i];

   unsigned char *name = (unsigned char *) malloc(name_bytes * sizeof(unsigned char));

   memcpy(name, packet + 3 + size_bytes + 2, name_bytes);

   name[name_bytes] = '\0';

   char *newFileName = getNewFilename((const char *) name, appendix, name_bytes +
strlen(appendix) + 1);

   return newFileName;
}
```

```
void printPacket(const unsigned char *packet, int packetSize) {
   printf("Received packet with size %d:\n", packetSize);
   printf("Contents: { ");
   for (int i = 0; i < packetSize; i++) {
      printf("%02X ", packet[i]);
   }
   printf("}\n\n");
}

unsigned char *getData(unsigned char *data, int dataSize, int *packetSize) {

   *packetSize = 3 + dataSize;

   unsigned char *start_packet = (unsigned char *) malloc(*packetSize);

   start_packet[0] = 1;
   start_packet[1] = (unsigned char) dataSize / 256;
   start_packet[2] = (unsigned char) dataSize % 256;

   memcpy(start_packet+3,data,dataSize);
   return start_packet;
}

unsigned char *getControl_packet(int control, const char *filename, long int fileSize, int
*cpsize) {

   int fileSize_bytes = (int) ceil(log2f((float)fileSize)/8.0);
   int fileName_bytes = strlen(filename);

   *cpsize = 5 + fileSize_bytes + fileName_bytes;

   unsigned char *packet = malloc(*cpsize);

   int packet_pos = 0;
   packet[packet_pos++] = control;
   packet[packet_pos++] = 0;
   packet[packet_pos++] = (unsigned char) fileSize_bytes;

   for(int i = 0; i<fileSize_bytes; i++) {
      packet[3 + i] = (fileSize >> (8*i)) && 0xFF;
      packet_pos++;
   }
   packet[packet_pos++] = 1;
   packet[packet_pos] = fileName_bytes;

   memcpy(packet + packet_pos, filename, fileName_bytes);

   return packet;
}
```

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

   Received.serialPort = serialPort;
   Received.baudRate = baudRate;
   Received.nRetransmissions = nTries;
   Received.role = strcmp(role, "tx") ? LlRx : LlTx;
   Received.timeout = timeout;

   int fd = llopen(Received);
   if(fd < 0) exit(-1);

   switch(Received.role) {
      case LlTx:
      {
      FILE *file = fopen(filename, "rb");
      if (file == NULL) {
         printf("File Not Found!\n");
         exit(-1);
      }

      int previous = ftell(file);
      fseek(file,0L,SEEK_END);
      long int fileSize = ftell(file)-previous;
      fseek(file,previous,SEEK_SET);


      int cpsize;
      unsigned char *Control_packet_start = getControl_packet(2, filename, fileSize, &cpsize);


      if(llwrite(fd, Control_packet_start, cpsize) == -1) exit(-1);
      //printPacket(Control_packet_start, cpsize);

      long int bytesLeft = fileSize;

      unsigned char *file_data = (unsigned char *) malloc(sizeof(unsigned char) * fileSize);

      fread(file_data, sizeof(unsigned char), fileSize, file);

      while(bytesLeft > 0) {
         int dataSize = 0;
         if(bytesLeft > MAX_PAYLOAD_SIZE) dataSize = MAX_PAYLOAD_SIZE;
         else dataSize = bytesLeft;

         unsigned char* data =(unsigned char*) malloc(dataSize);
         memcpy(data,file_data,dataSize);

         int packetSize;
         unsigned char *packet = getData(data,dataSize,&packetSize);
```

```
    if(llwrite(fd,packet,packetSize) == -1) exit(-1);

    file_data += dataSize;
    bytesLeft -= MAX_PAYLOAD_SIZE;
}

unsigned char *packet_end = getControl_packet(3, filename, fileSize, &cpsize);

if(llwrite(fd, packet_end, cpsize) == -1) exit(-1);

llclose(fd);

break;
}
case LlRx:
{
    unsigned char *packet = (unsigned char *) malloc(MAX_PAYLOAD_SIZE);
    int packetSize = 0;
    long int fileSize = 0;

    while ((packetSize = llread(fd, packet)) < 0);

    //printPacket(packet, packetSize);

    char *name = parseControlPacket(packet, &fileSize, Appendix);

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

    while (1) {
      if(packetSize == 0)  break;

        while ((packetSize = llread(fd, packet)) < 0);

        if (packet[0] == 1) {
            unsigned char *buffer = (unsigned char *) malloc(packetSize - 3);

            memcpy(buffer, packet + 3, packetSize - 3);

            fwrite(buffer, sizeof(unsigned char), packetSize - 3, newFile);

            free(buffer);


        } else if (packet[0] == 3) {
            //printPacket(packet, packetSize);
            long int fileSize2;
            name = parseControlPacket(packet, &fileSize2, Appendix);
            if (fileSize == fileSize2) {
                fclose(newFile);
                printf("File transfer completed.\n");
                break;
            }
```

```
            } else {
                printf("Invalid packet received. Terminating.\n");
                break;
            }
        }
        printf("Going for llclose \n");
        llclose(fd);
        break;
    }
    default:
    exit(-1);
    break;
  }
}
```