Faculdade de Engenharia da Universidade do Porto

Protocolo de ligação de dados

1º trabalho laboratorial

Redes de Computadores

Turma 4 Grupo 7

Estudantes:

Alexandre Fernandes Lopes (up202207015)

Hugo Alexandre Almeida Barbosa (up202205774)

Resumo

Este projeto, desenvolvido no âmbito da unidade curricular de Redes de Computadores, teve como foco a implementação do protocolo de comunicação *Stop & Wait*, para permitir que a troca de dados entre dois dispositivos ocorra de forma segura, sem que a transmissão dos mesmos seja concluída com erros.

Com este relatório, pretende-se expor as consequências e conclusões de usar o referido protocolo de comunicação, no que toca à velocidade de transmissão, performance, eficiência e integridade dos dados. Além disso, pretende-se analisar o impacto do uso de pacotes de diferentes tamanhos, avaliando como influenciam a latência, a taxa de erros e a eficiência global da transmissão de dados.

1. Introdução

O objetivo deste projeto foi o desenvolvimento de um protocolo de ligação de dados que garante a transferência confiável de dados entre dois dispositivos utilizando a porta série.

O relatório está dividido em várias secções:

- **Arquitetura**: Blocos funcionais e interfaces utilizadas
- **Estrutura do Código**: Apresentação das principais APIs, estruturas de dados e funções e a sua relação com a arquitetura
- **Principais Casos de Uso**: Identificação dos casos de uso e da sequência de chamada das funções
- **Protocolo de Ligação Lógica**: Principais aspetos funcionais e estratégia de implementação
- **Protocolo de Aplicação**: Aspetos essenciais da camada de aplicação e estratégia de implementação
- **Validação**: Testes realizados e resultados obtidos
- **Eficiência do Protocolo de Ligação de Dados**: Caracterização estatística da eficiência do protocolo Stop & Wait implementado
- **Conclusões**: Resumo da informação apresentada e reflexão sobre os objetivos de aprendizagem alcançados.

2. Arquitetura

O sistema é composto por duas camadas principais: a ***Application Layer*** e a ***Link Layer***.

A ***Application Layer***, declarada no ficheiro *application_layer.h* e definida no ficheiro

application_layer.c, opera acima da ***Link Layer*** e é a camada mais próxima do utilizador. O seu principal objetivo é fornecer uma interface direta e simples entre o utilizador e o sistema, permitindo a configuração de parâmetros essenciais, como a velocidade de transferência e o número máximo de retransmissões. A ***Application Layer*** depende da ***Link Layer*** para a transmissão e receção dos pacotes de dados.

A ***Link Layer***, declarada no ficheiro *link_layer.h* e definida no ficheiro *link_layer.c*, é responsável pela comunicação direta com o *hardware* através de uma porta série para a transmissão e receção de pacotes de dados. Esta camada é responsável por estabelecer e terminar a ligação, para além de gerir o envio de dados, a verificação de erros e a garantia da integridade dos dados, incluindo a criação, o envio e a validação das tramas recebidas.

3. Estrutura do Código

A **API** oferecida pelo *Link Layer* é constituída por 4 funções primárias, llopen(), llread(), llwritte() e llclose(), e 4 funções auxiliares, alarmHandler(), byteDestuff(), checkControl() e closeReceiver():

```
unsigned char checkControl();
int closeReceiver();
unsigned char byteDestuff(unsigned char byte);
void alarmHandler(int signal);
int llopen(LinkLayer connectionParameters);
int llwrite(const unsigned char *buf, int bufSize);
int llread(unsigned char *packet);
int llclose(int showStatistics);
```

Nesta camada, foram também definidas 4 estruturas de apoio, que são as seguintes: LinkLayer(usada para guardar a informação da porta), LinkLayerRole(distinção entre recetor e transmissor), SenderState(estado de leitura do transmissor) e ReceiverState(estado de leitura do recetor):

```
typedef enum {START_R,FLAG_RCV,A_RCV,C_RCV,BCC_OK_R,STOP_RCV,
    READ_DATA,ESC_FOUND,DISC_RCV,} ReceiverState;
typedef enum {START_S,FLAG_SDR,A_SDR,C_SDR,BCC_OK_S,STOP_SDR,
} SenderState;

typedef enum {LlTx,LlRx,} LinkLayerRole;

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

Já a **API** *Application Layer* é constituída por 1 função principal, applicationLayer() , e 5

funções auxiliares, parseControlPacket(), parseDataPacket(), getData(), getDataPacket(), getControlPacket():

```
void applicationLayer(const char *serialPort, const char *role, int
baudRate,int nTries, int timeout, const char *filename);
unsigned char* parseControlPacket(unsigned char* packet, int size,
unsigned long int *fileSize);
void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer);
unsigned char * getControlPacket(const unsigned int c, const char*
filename, long int length, unsigned int* size);
unsigned char * getDataPacket(unsigned char sequence, unsigned char
*data, int dataSize, int *packetSize);
unsigned char * getData(FILE* fd, long int fileLength);
```

Nesta API não foi definida qualquer tipo de estrutura de apoio.

4. Principais Casos de Uso

O sistema pode operar nos modos **recetor** e **transmissor,** e as funções e a ordem em que são chamadas variam de acordo com o modo selecionado.

Modo Transmissor

1. **llopen()** - Estabelece a conexão entre o transmissor e o recetor, através do envio e receção de pacotes de controlo. Também configura a porta série para a comunicação.
2. **getControlPacket()** - Cria e retorna um pacote de controlo com informações do ficheiro a ser transmitido, como o nome e o tamanho do ficheiro.
3. **getData()** - Lê o conteúdo do ficheiro a ser enviado e retorna-o em forma de buffer para que possa ser enviado em pacotes.
4. **getDataPacket()** - Divide o conteúdo do ficheiro em pacotes de tamanho adequado, atribuindo um número de sequência a cada pacote de forma a garantir a ordem correta.
5. **llwrite()** - Cria e envia pacotes de controlo e de dados para o recetor, garantindo a retransmissão em caso de falha na receção.
6. **checkControl()** - Valida as respostas do recetor após o envio de cada pacote, garantindo que os dados recebidos estão corretos e no formato esperado.
7. **llclose()** - Termina a comunicação, enviando um pacote de controlo final para finalizar a transferência e garantir que o recetor complete a receção.

Modo Recetor

1. **llopen()** - Recebe um comando de estabelecimento de comunicação e responde com um frame UA, ficando as portas ligadas.
2. **llread()** - Lê os pacotes de controlo ou de dados enviados pelo transmissor e valida a sua receção.
3. **parseControlPacket()** - Processa os pacotes de controlo recebidos, extraindo informações como nome e tamanho do ficheiro.
4. **parseDataPacket()** - Processa os pacotes de dados, verificando a sequência e a integridade, e escreve no ficheiro de destino.
5. **llclose()** - Recebe um comando de desconexão, e em seguida, chama closeReceiver() para finalizar o processo.
6. **closeReceiver()** - Cria e envia um pacote de controlo DISC para o transmissor e aguarda a resposta UA para finalizar a comunicação.

5. Protocolo de Ligação Lógica

Esta camada é responsável pela gestão completa da comunicação pela porta série, incluindo o envio e a leitura de dados, bem como o estabelecimento e encerramento da conexão entre as portas de comunicação. No caso do transmissor, a função llopen() envia um comando SET ao recetor e aguarda uma resposta UA, com um tempo de espera limitado por um alarme. Se o número de timeouts ultrapassar o limite predefinido, o programa encerra devido a uma falha na conexão. O recetor, por sua vez, aguarda a receção do comando SET e responde com um UA assim que o comando é validado. Uma vez estabelecida a comunicação, o transmissor utiliza a função llwrite() para enviar cada *packet* de informação, desde o *packet* de controlo inicial aos *packets* de dados e ao *packet* de controlo final. Antes de ser enviado um novo *packet* ao recetor, o mesmo tem que confirmar a receção correta do pacote anterior através de uma resposta positiva RR(0 ou 1), assegurando que o mesmo chegou sem erros ao destino. Caso contrário, é enviado ao transmissor uma resposta negativa REJ(0 ou 1) e o *frame* de informação é reenviado, na tentativa de fazer chegar a informação correta ao recetor. Este envio de *frames* também é controlado pelo alarme, sendo a transmissão cancelada caso o recetor não envie uma resposta positiva dentro do tempo e tentativas estipulado. Sempre que é recebido um REJ, o alarme é reiniciado. Já o recetor, através da função llread(), lê cada *packet* de informação recebido e verifica a integridade dos dados, respondendo com *feedback* positivo ou negativo, dependendo da situação. Esta função só retorna após a leitura completa de um

frame ou quando deteta que o *frame* recebido contém erros. Os erros podem acontecer no *header* do *frame* quanto na própria informação do ficheiro. Se for um frame novo, é enviada uma resposta REJ; caso contrário, a receção é confirmada com um RR. A deteção de erros é realizada comparando os bytes BCC1 e BCC2 dos frames recebidos com os valores calculados pela função correspondente. O BCC1 corresponde ao *header* do *frame*, e o BCC2 corresponde à informação do ficheiro. Após o envio do ficheiro, a função llclose() é chamada por ambas as portas para encerrar a comunicação de forma controlada. O transmissor envia um comando DISC que, ao ser lido pelo recetor, faz com que este envie de volta um comando DISC e aguarde por uma resposta UA. Neste caso, tanto o transmissor como o recetor usam o alarme para limitar o envio dos comandos. Caso o número definido de tentativas seja ultrapassado e o recetor ainda não tenha recebido o UA, ou o transmissor o DISC, o programa termina com falha na desconexão.

A leitura de dados das portas de série é feita byte a byte, e é apoiada pela utilização de máquinas de estados, no sentido em que um *frame* só é aceite se a máquina de estados chegar a um estado final.

6. Protocolo de Aplicação

A camada de aplicação é responsável pela interação direta entre o utilizador e o sistema, permitindo a configuração de parâmetros essenciais, como a porta série a ser utilizada, a velocidade de transferência, o número máximo de retransmissões, o tempo máximo de espera para uma resposta e o ficheiro a ser transferido. Para realizar a comunicação a camada de aplicação utiliza a API da *Link Layer*, que abstrai a comunicação em nível de *hardware* e converte os pacotes de dados em tramas de informação.

Após a ligação entre o transmissor e o recetor ser estabelecida com a função llopen(), o transmissor prepara o ficheiro, obtém os seus dados através da função getData() e fragmenta o conteúdo do ficheiro em pacotes de dados. O primeiro pacote enviado contém informações sobre o ficheiro, como o seu nome e tamanho, no formato TLV (Type, Length, Value), gerado pela função **getControlPacket()**. O recetor vai interpretar este pacote com a função **parseControlPacket()**, que extrai os dados e aloca memória para armazenar o ficheiro.

O ficheiro é transmitido em partes através de pacotes de dados gerados pela função **getDataPacket()**, que são depois enviados utilizando a função **llwrite()**. Após o envio de cada pacote, o recetor utiliza a função **llread()** para ler os pacotes recebidos. A função **parseDataPacket()** é então utilizada para processar e armazenar os dados extraídos de cada pacote. Se o pacote for aceite, o recetor responde com uma confirmação, permitindo

que o transmissor envie o próximo pacote. Se for rejeitado, o transmissor vai retransmitir o pacote.

A transferência de dados termina quando o transmissor envia um pacote de controlo final, indicando o fim da transferência, o qual é interpretado pelo recetor, que então escreve os dados recebidos no ficheiro. Finalmente, a ligação é fechada usando **llclose()**.

7. Validação

Com o objetivo de validar a exatidão do código, foram realizados diversos testes ao código. Num momento inicial, priorizou-se uma testagem do tipo *black box testing*, no *Link Layer* e no *Application Layer*. Numa fase avançada, testamos a integração do *Link Layer* com o *Application Layer*. Para averiguar se o ficheiro é enviado com sucesso, independentemente das condições de comunicação, foi simulado uma variedade de ambientes em que o envio ocorre:

1) Desligar e voltar a ligar o cabo de ligação;
2) Aumentar o BER(*Bit Error Rate*);
3) Aumentar o *Propagation Delay*;
4) Variar a *Baud Rate*;

Um último teste que foi feito, foi a avaliação da eficiência do envio, onde foram criados ambientes mistos de condições, como por exemplo, juntar o aumento da BER com o *Propagation Delay*, e ainda com a variação da *Baud Rate*.

8. Eficiência do Protocolo de Ligação de Dados

Para avaliar a eficiência do protocolo, fizemos as seguintes medidas, com um ficheiro de 10968 bytes:

Variação de FER (*Frame Error Rate*)

Para uma *BaudRate* de 9600 bits/s, um tamanho de *frame* de 500 bytes e um *Propagation Delay* de 0 segundos, os resultados são os seguintes:

FER(%)	Tempo Médio de execução(s)	Eficiência(%)
1	11,93	76,61
5	17,52	52,16
10	19,12	47,80
25	35,13	26,01

50	98,12	9,32

Como é possível observar, o aumento da taxa de erros nas tramas (*FER*) resulta num aumento significativo do tempo de execução, pois o protocolo *Stop & Wait* requer retransmissões em caso de erro, o que aumenta o tempo de execução. Assim, a eficiência diminui à medida que a taxa de erros aumenta.

Variação de *BaudRate*

Para uma FER de 0%, um tamanho de *frame* de 500 bytes e um *Propagation Delay* de 0 segundos, os resultados são os seguintes:

BaudRate(bits/s)	Tempo de execução(s)	Eficiência(%)
2400	47,69	76,66
4800	23,86	76,61
9600	11,93	76,61
19200	5,97	76,54
38400	2,99	76,42

Como é possível observar, o tempo de execução diminui com o aumento do *BaudRate*. Apesar disso, a eficiência diminui pouco com o seu aumento, pelo que é inversamente proporcional.

Variação do *Propagation Delay*

Para uma *BaudRate* de 9600 bits/s, um tamanho de *frame* de 500 bytes e um FER de 0%, os resultados são os seguintes:

Propagation Delay(s)	Tempo de execução(s)	Eficiência(%)
0	11,93	76,61
0,010	12,44	73,47
0,050	14,42	63,38
0,100	16,92	54,01
0,500	36,94	24,74

Como é possível observar, o aumento do *Propagation Delay* leva a um aumento do tempo

de execução, o que reduz a eficiência. Isto acontece porque o protocolo *Stop & Wait* depende de ciclos de envio e de espera, e qualquer atraso adicional na transmissão resulta em mais tempo gasto no processo de comunicação, o que prejudica a eficiência.

Variação do tamanho dos *frames* enviados

Para uma *BaudRate* de 9600 bits/s, um *Propagation Delay* de 0 segundos e um FER de 0%, os resultados são os seguintes:

Tamanho do frame(bytes)	Tempo de execução(s)	Eficiência(%)
250	12,27	74,49
500	11,93	76,61
1000	11,76	77,72
1500	11,70	78,11
2000	11,68	78,25

Como é possível observar, a eficiência aumenta pouco com o aumento do tamanho do *frame*, tornando-o um fator de menor influência. No entanto, com um *BER* alto, frames menores reduzem os erros e aceleram a transmissão. Já com um *Propagation Delay* elevado, frames maiores são mais eficientes, pois exigem menos transmissões para enviar a mesma quantidade de dados para conseguir entregar a informação toda, sendo assim mais eficiente a transmissão.

9. Conclusões

Com a realização deste trabalho, foi possível compreender melhor o funcionamento do protocolo de comunicação *Stop & Wait* e a importância de uma implementação eficiente para garantir a fiabilidade da transmissão de dados. Observou-se como o protocolo responde a diversas condições físicas e de *hardware*, evidenciando a necessidade de tratar cenários de perdas de pacotes e tempos de retransmissão. Também foi importante observar como o tempo de execução é influenciado pelo tamanho das tramas enviadas. Verificou-se que, em cenários com uma BER elevada, o uso de tramas menores é mais eficiente, pois reduz a necessidade de retransmissões. Por outro lado, quando o *Propagation Delay* é mais elevado e a BER é baixa, tramas maiores permitem otimizar o tempo total de transmissão.

Apêndices

Apêndice I - Código Fonte

link_layer.h

```
#ifndef _LINK_LAYER_H_
#define _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>
#include <time.h>

#define BAUDRATE B38400
//MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

#define FALSE 0
#define TRUE 1

#define BUF_SIZE 256
#define FRAME_SIZE 5

#define FLAG 0x7E
#define ADDRESS_SENT_TRANSMITTER 0x03
#define ADDRESS_ANSWER_RECEIVER 0x03
#define ADDRESS_SENT_RECEIVER 0X01
#define ADDRESS_ANSWER_TRANSMITTER 0X01
#define CONTROL_SET 0X03
#define CONTROL_UA 0X07
```

```
#define C_N(Ns) ((Ns) << 6)
#define ESC 0x7D
#define RR0 0xAA
#define RR1 0xAB
#define REJ0 0X54
#define REJ1 0X55
#define DISC 0X0B

#define ALARM_MAX_RETRIES 4

typedef enum {
    START_R,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK_R,
    STOP_RCV,
    READ_DATA,
    ESC_FOUND,
    DISC_RCV,
} ReceiverState;

typedef enum {
    START_S,
    FLAG_SDR,
    A_SDR,
    C_SDR,
    BCC_OK_S,
    STOP_SDR,
} SenderState;

typedef enum {
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
    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
    #define PAYLOAD_SIZE_100 100
    #define PAYLOAD_SIZE_200 200
    #define PAYLOAD_SIZE_300 300
    #define PAYLOAD_SIZE_400 400
    #define PAYLOAD_SIZE_500 500
    #define PAYLOAD_SIZE_600 600
    #define PAYLOAD_SIZE_700 700
    #define PAYLOAD_SIZE_800 800
    #define PAYLOAD_SIZE_900 900

    // MISC
    #define FALSE 0
    #define TRUE 1

    unsigned char checkControl();
    int closeReceiver();
    unsigned char byteDestuff(unsigned char byte);

    // 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(const unsigned char *buf, int bufSize);

    // Receive data in packet.
    // Return number of chars read, or "-1" on error.
```

```
int llread(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_
```

link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"

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

volatile int STOP = FALSE;

unsigned char frameNumberT = 0;
unsigned char frameNumberR = 0;

int isTx = FALSE;

int alarmEnabled = FALSE;
int responseReceived = FALSE;
int alarmCount = 0;
int retransmissions = 0;
int timeout = 0;

//statistics
int framesSent = 0;
int framesReceived = 0;
int retransmissionsNumber = 0;
int timeouts = 0;
int framesRejected = 0;
clock_t startTime;
```

```
    int duplicateFrames = 0;

    // Alarm function handler
    void alarmHandler(int signal) {
        alarmEnabled = FALSE;
        alarmCount++;
        timeouts++;
        if (alarmCount > 0) {
            printf("\nAlarm #%d\n\n", alarmCount);
        }
    }

    unsigned char checkControl() {
        unsigned char byte;
        unsigned char c = 0;
        SenderState state = START_S;

        while (state != STOP_SDR && alarmEnabled == TRUE) {
            if (readByteSerialPort(&byte)) {
                switch (state) {
                    case START_S:
                        if (byte == FLAG) state = FLAG_SDR;
                        break;

                    case FLAG_SDR:
                            if (byte == ADDRESS_ANSWER_RECEIVER) state =
A_SDR;
                        else if (byte == FLAG) state = FLAG_SDR;
                        else state = START_S;
                        break;

                    case A_SDR:
                          if (byte == RR0 || byte == RR1 || byte == REJ0
|| byte == REJ1) {
                            state = C_SDR;
                            c = byte;
                        }
                        else if (byte == FLAG) state = FLAG_SDR;
                        else state = START_S;
                        break;
```

```
                case C_SDR:
                     if (byte == (ADDRESS_ANSWER_RECEIVER ^ c)) state
= BCC_OK_S;
                    else if (byte == FLAG) state = FLAG_SDR;
                    else state = START_S;
                    break;

                case BCC_OK_S:
                    if (byte == FLAG) {
                        state = STOP_SDR;
                        return c;
                    }
                    else state = START_S;
                    break;

                default:
                    break;
            }
        }
    }
    return c;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {

    startTime = clock();

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

    retransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;

    switch (connectionParameters.role) {
        case LlTx: {
```

```
                isTx = TRUE;
                printf("\nNew termios structure set\n\n");

                (void)signal(SIGALRM, alarmHandler);

                // Create string to send
                            unsigned  char  bufS[FRAME_SIZE]  =  {FLAG,
ADDRESS_SENT_TRANSMITTER,   CONTROL_SET,   ADDRESS_SENT_TRANSMITTER   ^
CONTROL_SET, FLAG};

                SenderState senderState = START_S;
                alarmCount = 0;
                alarmEnabled = FALSE;

                   while (alarmCount < retransmissions && senderState !=
STOP_SDR) {
                    if (!alarmEnabled) {
                                                          int   bytes   =
writeBytesSerialPort(bufS,FRAME_SIZE);
                                                //  bufS[2]  =  CONTROL_SET;
//SO PARA TESTE
                        if (bytes < 0) {
                                perror("Failed to write bytes to serial
port");
                            return -1;
                        }
                        sleep(1); //important
                        printf("%d bytes written\n", bytes);
                        alarm(timeout);

                        alarmEnabled = TRUE;
                    }

                    unsigned char response_byte;

                    if (readByteSerialPort(&response_byte) == -1) {
                        perror("Failed to read byte from serial port");
                        continue;
                    }
```

```
                    switch(senderState) {
                        case START_S:
                            //printf("start\n");
                                if (response_byte == FLAG) senderState =
FLAG_SDR;
                            break;
                        case FLAG_SDR:
                            //printf("flag\n");
                                                    if  (response_byte  ==
ADDRESS_ANSWER_RECEIVER) senderState = A_SDR;
                             else if (response_byte != FLAG) senderState
= START_S;
                            break;
                        case A_SDR:
                            //printf("A\n");
                             if (response_byte == CONTROL_UA) senderState
= C_SDR;
                              else if (response_byte == FLAG) senderState
= FLAG_SDR;
                            else senderState = START_S;
                            break;
                        case C_SDR:
                            //printf("C\n");
                                                    if  (response_byte  ==
(ADDRESS_ANSWER_RECEIVER ^ CONTROL_UA)) senderState = BCC_OK_S;
                              else if (response_byte == FLAG) senderState
= FLAG_SDR;
                            else senderState = START_S;
                            break;
                        case BCC_OK_S:
                            //printf("BCC\n");
                            if (response_byte == FLAG) {
                                senderState = STOP_SDR;
                                alarm(0);
                            }
                            else senderState = START_S;
                            break;
                        default:
                            senderState = START_S;
                            break;
```

```
                }
            }

            alarm(0);

            if (senderState == STOP_SDR) {
                //printf("STOP\n");
                                        printf("\nReceived  UA  frame
successfully\n\nConnection sucessfull!\n\n");
                return 1;
            }

            else {
                    printf("\nNo response from receiver\n\nCanceling
operation...\n\n");
                return -1;
            }
            break;
        }

        case LlRx: {
            printf("\nNew termios structure set\n");

            // Loop for input
            unsigned char byte2;
            ReceiverState ReceiverState = START_R;
            while (ReceiverState != STOP_RCV) {
                if (readByteSerialPort(&byte2)) {
                     //printf(":%s:%d\n", buf, bytes); //prints frame
received
                    switch(ReceiverState) {
                        case START_R:
                        //printf("start\n");
                                    if (byte2 == FLAG) ReceiverState =
FLAG_RCV;
                            else ReceiverState = START_R;
                            break;
                        case FLAG_RCV:
                            //printf("flag\n");
                               if (byte2 == ADDRESS_SENT_TRANSMITTER)
```

```
ReceiverState = A_RCV;
                                else if (byte2 == !FLAG) ReceiverState =
START_R;
                                break;
                            case A_RCV:
                                //printf("A\n");
                                 if (byte2 == CONTROL_SET) ReceiverState
= C_RCV;
                                else if (byte2 == FLAG)
                                    ReceiverState = FLAG_RCV;
                                else ReceiverState = START_R;
                                break;
                            case C_RCV:
                                //printf("C\n");
                                 if (byte2 == (ADDRESS_SENT_TRANSMITTER ^
CONTROL_SET)) {
                                    ReceiverState = BCC_OK_R;
                                }
                                  else if (byte2 == FLAG) ReceiverState =
FLAG_RCV;
                                else ReceiverState = START_R;
                                break;
                            case BCC_OK_R:
                                //printf("BCC\n");
                                      if (byte2 == FLAG) ReceiverState =
STOP_RCV;
                                else ReceiverState = START_R;
                                break;
                            default:
                                ReceiverState = START_R;
                                break;
                        }
                    }
                }
                //printf("STOP\n");
                            unsigned char uaFrame[FRAME_SIZE] = {FLAG,
ADDRESS_ANSWER_RECEIVER,    CONTROL_UA,    ADDRESS_ANSWER_RECEIVER    ^
CONTROL_UA, FLAG};
                if (writeBytesSerialPort(uaFrame, FRAME_SIZE) < 0) {
                    perror("Failed to send UA frame");
```

```
                    return -1;
                }
                printf("\nSent UA frame\n\n");
                return 1;
                break;
            }

            default:
                break;
        }
        return spfd;
    }

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

        if (stuffed_frame == NULL) {
            perror("Memory allocation failed");
            return -1;
        }

        stuffed_frame[0] = FLAG;
        stuffed_frame[1] = ADDRESS_SENT_TRANSMITTER;
        stuffed_frame[2] = C_N(frameNumberT);
        stuffed_frame[3] = ADDRESS_SENT_TRANSMITTER ^ C_N(frameNumberT);

        unsigned char BCC2 = 0;
        for (unsigned int i = 0; i < bufSize; i++) {
            BCC2 ^= buf[i]; // doing XOR of each byte with BCC2
        }

        printf("Frame BCC2 = 0x%02X\n", BCC2); // Debugging BCC2 value

        int j = 4;
        for (int i = 0; i < bufSize; i++) {
```

```
        if (buf[i] == FLAG) {
            stuffed_frame[j++] = ESC;
            stuffed_frame[j++] = FLAG ^ 0X20;
        } else if (buf[i] == ESC) {
            stuffed_frame[j++] = ESC;
            stuffed_frame[j++] = ESC ^ 0x20;
        } else {
            stuffed_frame[j++] = buf[i];
        }
    }

    if (BCC2 == FLAG) {
        stuffed_frame[j++] = ESC;
        stuffed_frame[j++] = FLAG ^ 0x20;
    } else if (BCC2 == ESC) {
        stuffed_frame[j++] = ESC;
        stuffed_frame[j++] = ESC ^ 0x20;
    } else {
        stuffed_frame[j++] = BCC2;
    }

    stuffed_frame[j++] = FLAG;

    inf_frame_size = j;

    int current_transmission = 0;
    int rejected = 0;
    int accepted = 0;
    alarmEnabled = FALSE;
    alarmCount = 0;

    while (current_transmission <= retransmissions) {
        rejected = 0;
        accepted = 0;

        if (!alarmEnabled) {
            current_transmission++;
            if (current_transmission > retransmissions + 1) break;
            alarmEnabled = TRUE;
            int bytesW = writeBytesSerialPort(stuffed_frame, j);
```

```
                framesSent++;
                if (bytesW < 0) {
                    perror("Failed to write bytes to serial port");
                    free(stuffed_frame);
                    return -1; // Handle error
                }
                alarm(timeout);
                   printf("%d bytes written\n\n", bytesW); // Debugging:
bytes written to serial port
            }

            unsigned char command = checkControl();
                  printf("Receiver response = 0x%02X\n", command); //
Debugging: command received

            if (command == REJ0 || command == REJ1) {
                rejected = 1;
             } else if ((command == RR0 && frameNumberT == 1) || (command
== RR1 && frameNumberT == 0)) {
                accepted = 1;
                frameNumberT = (frameNumberT + 1) % 2;
                  printf("New Frame number = 0x%02X\n\n", frameNumberT);
// Debugging: frame number
            }

            if (accepted) {
                alarm(0);
                break;
            } else if (rejected) {
                alarm(0);
                alarmEnabled = FALSE;
                alarmCount = -1;
                current_transmission = 0;
                  printf("Frame was rejected. Resending data bytes.\n");
// Debugging: bytes rewritten on rejection
            }
        }

        alarm(0);
        free(stuffed_frame);
```

```
        if (accepted) {
            printf("Frame delivered with success!\n\n");
            return inf_frame_size;
        } else {
            printf("Frame could not be delivered..\n\n");
            return -1;
        }

        return -1;
    }

    ////////////////////////////////////////////////
    // LLREAD
    ////////////////////////////////////////////////
    int llread(unsigned char *packet) {
        unsigned char byte;
        unsigned char c = 0;
        int x = 0;
        ReceiverState state = START_R;

        while (state != STOP_RCV) {
            if (readByteSerialPort(&byte)) {
                //printf("0x%02X ", byte);
                switch (state) {
                    case START_R:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;

                    case FLAG_RCV:
                           if (byte == ADDRESS_SENT_TRANSMITTER) state =
A_RCV;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START_R;
                        break;

                    case A_RCV:
                          if (byte == C_N(0) || byte == C_N(1)/* || byte
== DISC*/) {
                              state = C_RCV;
```

```
                            c = byte;
                        }
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START_R;
                        break;

                    case C_RCV:
                        if (byte == (ADDRESS_SENT_TRANSMITTER ^ c)) {
                            state = READ_DATA;
                             if ((c == C_N(0) && frameNumberR == 1) || (c
== C_N(1) && frameNumberR == 0)) {
                                state = STOP_RCV;
                                   unsigned char cResponse = frameNumberR
== 0 ? RR0 : RR1;
                                                                unsigned  char
supervisionFrame[FRAME_SIZE]     =     {FLAG,     ADDRESS_ANSWER_RECEIVER,
cResponse, ADDRESS_ANSWER_RECEIVER ^ cResponse, FLAG};

                                                                int  bytesW  =
writeBytesSerialPort(supervisionFrame, FRAME_SIZE);
                                        printf("\nReceived duplicated data
frame\n%d positive response bytes written\n", bytesW);
                                duplicateFrames++;
                                return 0;
                            }
                            //else if (c == DISC) state = DISC_RCV;
                        } else if (byte == FLAG) state = FLAG_RCV;
                        else state = START_R;
                        break;

                    /*case DISC_RCV:
                        if (byte == FLAG) {
                            closeReceiver();
                        }
                        else {
                            state = START_R;
                        }
                        */
                    case READ_DATA:
                        if (byte == ESC) state = ESC_FOUND;
```

```
                        else if (byte == FLAG) {
                            unsigned char bcc2 = packet[--x];
                            printf("\nFrame BCC2 = 0x%02X\n", bcc2);

                            unsigned char acc = 0;
                            for (unsigned int i = 0; i < x; i++) {
                                acc ^= packet[i];
                                      //printf("0x%02X ", packet[i]); //
Debugging: packet content
                            }

                            printf("\nCalculated BCC2 = 0x%02X\n", acc);

                            if (bcc2 == acc) {
                                framesReceived++;
                                state = STOP_RCV;
                                   unsigned char cResponse = frameNumberR
== 0 ? RR1 : RR0;
                                                                unsigned  char
supervisionFrame[FRAME_SIZE]     =     {FLAG,     ADDRESS_ANSWER_RECEIVER,
cResponse, ADDRESS_ANSWER_RECEIVER ^ cResponse, FLAG};

                                                                int  bytesW  =
writeBytesSerialPort(supervisionFrame, FRAME_SIZE);
                                    printf("\nGot new frame!\n%d positive
response bytes written\n", bytesW);
                                frameNumberR = (frameNumberR + 1) % 2;
                                return x;
                            } else {
                                   if ((c == C_N(0) && frameNumberR == 1)
|| (c == C_N(1) && frameNumberR == 0)) {
                                    state = STOP_RCV;
                                              unsigned char cResponse =
frameNumberR == 0 ? RR0 : RR1;
                                                                unsigned  char
supervisionFrame[FRAME_SIZE]     =     {FLAG,     ADDRESS_ANSWER_RECEIVER,
cResponse, ADDRESS_ANSWER_RECEIVER ^ cResponse, FLAG};

                                                                int  bytesW  =
writeBytesSerialPort(supervisionFrame, FRAME_SIZE);
```

```
                                    printf("\n%d Error in frame data but
is a duplicated frame.\n positive response bytes written\n", bytesW);
                                    free(packet);
                                    packet = NULL;
                                                packet = (unsigned char
*)malloc(PAYLOAD_SIZE_500);
                                    return 0;
                                } else {
                                     printf("\nError in data, asking for
retransmission\n");
                                                unsigned char cResponse =
frameNumberR == 0 ? REJ0 : REJ1;
                                                                unsigned char
supervisionFrame[FRAME_SIZE]    =    {FLAG,    ADDRESS_ANSWER_RECEIVER,
cResponse, ADDRESS_ANSWER_RECEIVER ^ cResponse};
                                                                int bytesW =
writeBytesSerialPort(supervisionFrame, FRAME_SIZE);
                                    framesRejected++;
                                     printf("\n%d negative response bytes
written\n", bytesW);
                                    /*state = START_R;
                                    x = 0;*/
                                    free(packet);
                                    packet = NULL;
                                                packet = (unsigned char
*)malloc(PAYLOAD_SIZE_500);
                                    return -1;
                                }
                            }
                        } else {
                            packet[x++] = byte;
                        }
                        break;

                    case ESC_FOUND:
                        packet[x++] = byteDestuff(byte);
                        state = READ_DATA;
                        break;

                    case STOP_RCV:
```

```
                    break;

                default:
                    state = START_R;
                    break;
            }
        }
    }

    return -1;
}

unsigned char byteDestuff(unsigned char byte) {
    if (byte == (FLAG ^ 0x20)) {
        return FLAG;
    }
    else if (byte == (ESC ^ 0x20)) {
        return ESC;
    }
    return byte;
}

int closeReceiver() {

        unsigned char supervisionFrame[FRAME_SIZE] = {FLAG,
ADDRESS_SENT_RECEIVER, DISC, ADDRESS_SENT_RECEIVER ^ DISC, FLAG};

    ReceiverState receiverState = START_R;

    alarmEnabled = FALSE;
    alarmCount = 0;
    (void)signal(SIGALRM, alarmHandler);

        while (alarmCount <= retransmissions && receiverState !=
STOP_RCV) {

        if (!alarmEnabled) {
                int bytes = writeBytesSerialPort(supervisionFrame,
FRAME_SIZE);
            printf("\n%d DISC bytes written to transmitter\n\n",
```

```
bytes);
                alarm(timeout);

                alarmEnabled = TRUE;
            }

            unsigned char response_byte;

            if (readByteSerialPort(&response_byte)) {
                //printf("0x%02X ", response_byte);
                switch(receiverState) {
                    case START_R:
                        //printf("start\n");
                             if (response_byte == FLAG) receiverState =
FLAG_RCV;
                        else receiverState = START_R;
                        break;
                    case FLAG_RCV:
                        //printf("flag\n");
                         if (response_byte == ADDRESS_ANSWER_TRANSMITTER)
receiverState = A_RCV;
                          else if (response_byte == FLAG) receiverState =
FLAG_RCV;
                        else receiverState = START_R;
                        break;
                    case A_RCV:
                        //printf("A\n");
                         if (response_byte == CONTROL_UA) receiverState =
C_RCV;
                          else if (response_byte == FLAG) receiverState =
FLAG_RCV;
                        else receiverState = START_R;
                        break;
                    case C_RCV:
                        //printf("C\n");
                         if (response_byte == (ADDRESS_ANSWER_TRANSMITTER
^ CONTROL_UA)) receiverState = BCC_OK_R;
                          else if (response_byte == FLAG) receiverState =
FLAG_RCV;
                        else receiverState = START_R;
```

```
                    break;
                case BCC_OK_R:
                    //printf("BCC\n");
                    if (response_byte == FLAG) {
                        alarm(0);
                        receiverState = STOP_RCV;
                    }
                    else receiverState = START_R;
                    break;
                case STOP_RCV:
                    break;
                default:
                    receiverState = START_S;
                    break;
            }
        }
    }

    alarm(0);

    if (receiverState == STOP_RCV) {
        printf("Received UA frame successfully\n\n");
        return 1;
    }

    else {
        printf("Did not receive UA from transmitter\n\n");
        return -1;
    }

    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    unsigned char byte;
    ReceiverState state = START_R;
```

```
       switch (isTx) {

       case TRUE:

           (void)signal(SIGALRM, alarmHandler);

              unsigned char bufS[5] = {FLAG, ADDRESS_SENT_TRANSMITTER,
DISC, ADDRESS_SENT_TRANSMITTER ^ DISC, FLAG};

           SenderState senderState = START_S;

           alarmCount = 0;

           alarmEnabled = FALSE;

               while (alarmCount <= retransmissions && senderState !=
STOP_SDR) {
               if (!alarmEnabled) {
                   int bytes = writeBytesSerialPort(bufS,FRAME_SIZE);
                           printf("\n%d DISC command bytes written to
receiver\n\n", bytes);
                   alarm(timeout);
                   alarmEnabled = TRUE;
               }

               unsigned char response_byte;

               if (readByteSerialPort(&response_byte)) {
                   //printf("Received byte: 0x%02X\n", response_byte);
                   switch(senderState) {
                       case START_S:
                           //printf("start\n");
                              if (response_byte == FLAG)  senderState =
FLAG_SDR;
                           else senderState = START_S;
                           break;
                       case FLAG_SDR:
                           //printf("flag\n");
                             if (response_byte == ADDRESS_SENT_RECEIVER)
```

```
senderState = A_SDR;
                            else if (response_byte == FLAG) senderState
= FLAG_SDR;
                        else senderState = START_S;
                        break;
                    case A_SDR:
                        //printf("A\n");
                            if (response_byte == DISC) senderState =
C_SDR;
                            else if (response_byte == FLAG) senderState
= FLAG_SDR;
                        else senderState = START_S;
                        break;
                    case C_SDR:
                        //printf("C\n");
                            if (response_byte == (ADDRESS_SENT_RECEIVER
^ DISC)) senderState = BCC_OK_S;
                            else if (response_byte == FLAG) senderState
= FLAG_SDR;
                        else senderState = START_S;
                        break;
                    case BCC_OK_S:
                        //printf("BCC\n");
                        if (response_byte == FLAG) {
                            alarm(0);
                            senderState = STOP_SDR;
                                                printf("Received  DISC
acknowledgment\n\n");
                        }
                        else senderState = START_S;
                        break;
                    case STOP_SDR:
                        break;
                    default:
                        senderState = START_S;
                        break;
                }
            }
        }
```

```
            alarm(0);

            if (senderState == STOP_SDR) {
                printf("Read DISC frame successfully\n\n");
                            unsigned char uaFrame[FRAME_SIZE] = {FLAG,
ADDRESS_ANSWER_TRANSMITTER, CONTROL_UA, ADDRESS_ANSWER_TRANSMITTER ^
CONTROL_UA, FLAG};
                writeBytesSerialPort(uaFrame, FRAME_SIZE);
                printf("Sent UA frame\n\nDisconnect completed!\n\n");
            }
               else printf("Did not receive DISC command from receiver
(retry limit reached)\n\n");

            clock_t endTime = clock();
                 double elapsedTime = (double)(endTime - startTime) /
CLOCKS_PER_SEC;

            if (showStatistics) {
                printf("Communication Statistics:\n");
                printf("Data Frames Sent: %d\n", framesSent);
                    printf("Number of duplicate frames received: %d\n",
duplicateFrames);
                printf("Number of timeouts: %d\n", timeouts);
                        printf("Total execution time: %.2f seconds\n",
elapsedTime);
            }

            break;

        case FALSE:

            while (state != STOP_RCV) {
                if (readByteSerialPort(&byte)) {
                    switch (state) {
                    case START_R:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;

                    case FLAG_RCV:
                            if (byte == ADDRESS_SENT_TRANSMITTER) state =
```

```
A_RCV;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_R;
                    break;

                case A_RCV:
                    if (byte == DISC) state = C_RCV;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_R;
                    break;

                case C_RCV:
                    if (byte == (ADDRESS_SENT_TRANSMITTER ^ DISC)) {
                        state = DISC_RCV;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_R;
                    break;

                case DISC_RCV:
                    if (byte == FLAG) {
                                printf("\nDISC command received from
transmitter\n\n");
                        state = STOP_RCV;
                    }
                    else {
                        state = START_R;
                    }
                    break;

                case STOP_RCV:
                    break;

                default:
                state = START_R;
                    break;
                }
            }
        }
```

```
            if (state == STOP_RCV) {
                printf("Sending DISC to transmitter\n");
                if (closeReceiver() == -1) {
                    printf("Error on disconecting\n\n");
                    return -1;
                }
            }


            if (showStatistics) {
                printf("Communication Statistics:\n");
                       printf("Data Frames Received Sucessfully: %d\n",
framesReceived);
                printf("Frames rejected: %d\n", framesRejected);
            }

            break;
        default:
            break;
        }


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

application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include "link_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* parseControlPacket(unsigned char* packet, int size, unsigned long int *fileSize);
void parseDataPacket(const unsigned char* packet, const unsigned int packetSize, unsigned char* buffer);
unsigned char * getControlPacket(const unsigned int c, const char* filename, long int length, unsigned int* size);
unsigned char * getDataPacket(unsigned char sequence, unsigned char *data, int dataSize, int *packetSize);
unsigned char * getData(FILE* fd, long int fileLength);

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include <stdio.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.baudRate = baudRate;
    linklayer.nRetransmissions = nTries;
    linklayer.timeout = timeout;
    linklayer.role = strcmp(role, "tx") ? LlRx : LlTx;

    if (llopen(linklayer) == -1) {
        fprintf(stderr, "Error: Could not establish connection\n");
        exit(1);
```

```
        }

        FILE *file;

        unsigned char *packet;
        unsigned long int receivedFileSize = 0;
        int packetSize;


        switch(linklayer.role) {
            case LlTx:
                file = fopen(filename, "rb");
                if (file == NULL) {
                    perror("Error opening file\n");
                    exit(-1);
                }

                int prev = ftell(file);
                fseek(file,0L,SEEK_END);
                long int fileSize = ftell(file)-prev;
                fseek(file,prev,SEEK_SET);
                   printf("Info: File size = %ld bytes\n\n", fileSize);
//debug


                unsigned int controlPacketSize;
                     unsigned char *controlPacket = getControlPacket(2,
filename, fileSize, &controlPacketSize);
                if (llwrite(controlPacket, controlPacketSize) == -1) {
                     fprintf(stderr, "Error: Failed to send start control
packet\n");
                    free(controlPacket);
                    fclose(file);
                    exit(-1);
                }
                free(controlPacket);

                unsigned char sequence = 0;
                unsigned char* content = getData(file, fileSize);
                long int bytesLeft = fileSize;
```

```
            while (bytesLeft > 0) {
                    int dataSize = (bytesLeft > PAYLOAD_SIZE_500) ?
PAYLOAD_SIZE_500 : bytesLeft;
                            unsigned char *data = (unsigned char
*)malloc(dataSize);

                memcpy(data, content, dataSize);
                int packetSize;
                     unsigned char* packet = getDataPacket(sequence,
data, dataSize, &packetSize);

                if (dataSize > PAYLOAD_SIZE_500) {
                       fprintf(stderr, "Error: Payload size exceeded
maximum limit\n");
                    free(packet);
                    free(data);
                    fclose(file);
                    exit(-1);
                }

                if (llwrite(packet, packetSize) == -1) {
                         fprintf(stderr, "Error: Failed to send data
packet\n");
                    free(packet);
                    free(data);
                    fclose(file);
                    exit(-1);
                }

                bytesLeft -= dataSize;

                content += dataSize;

                         printf("\n\nInfo:\nPayload size sent = %d
bytes\nFrame size (with headers) = %d bytes\nRemaining file size = %ld
bytes\n\n", dataSize, packetSize, bytesLeft);
                // Clean up
                free(packet);
                free(data);
```

```
                    // Update the sequence number
                    sequence = (sequence + 1) % 256;

                       printf("Info: Remaining file size = %ld bytes\n",
bytesLeft);
                }

                  unsigned char *endControlPacket = getControlPacket(3,
filename, 0, &controlPacketSize);
                 if (llwrite(endControlPacket, controlPacketSize) == -1)
{
                     fprintf(stderr, "Error: Failed to send end control
packet\n");
                    free(endControlPacket);
                    fclose(file);
                    exit(-1);
                }
                free(endControlPacket);

                llclose(1);
                break;

            case LlRx:
                packet = (unsigned char *)malloc(PAYLOAD_SIZE_500);
                receivedFileSize = 0;
                packetSize = 0;
                while (packetSize <= 0) {
                    packetSize = llread(packet);
                }

                if (packetSize < 0) {
                     fprintf(stderr, "Error: Failed to read start control
packet\n");
                    free(packet);
                    llclose(1);
                    exit(1);
                }

                                unsigned  char  *filenameReceived  =
```

```
parseControlPacket(packet, packetSize, &receivedFileSize);
                if (filenameReceived == NULL) {
                        fprintf(stderr, "Error: Could not parse start
control packet\n");
                    free(packet);
                    llclose(1);
                    exit(-1);
                }

                file = fopen((char*)filename, "wb+");
                if (file == NULL) {
                    perror("Error creating file\n");
                    free(packet);
                    llclose(1);
                    exit(-1);
                }

                while (1) {
                    packetSize = 0;
                    while (packetSize <= 0) {
                        packetSize = llread(packet);
                    }
                    if(packet[0] != 3){
                                    unsigned char *buffer = (unsigned
char*)malloc(packetSize);
                        parseDataPacket(packet, packetSize, buffer);
                                fwrite(buffer, sizeof(unsigned char),
packetSize-4, file);
                        free(buffer);
                    }
                    else break;
                }

                fclose(file);
                free(packet);
                llclose(1);
                break;

            default:
                fprintf(stderr, "Error: Unknown role\n");
```

```
                llclose(1);
                exit(1);
            break;
        }
        printf("\n\nEnding Program!\n");
    }

    unsigned char* parseControlPacket(unsigned char* packet, int size,
unsigned long int *fileSize) {
        if (size < 3) {
                printf("Packet is too small to contain minimum required
data\n");
            return NULL;
        }

        *fileSize = 0;

        unsigned char fileSizeBytes = packet[2];

        if (size < 3 + fileSizeBytes) {
                printf("Packet is too small to contain the declared file
size field.\n");
            return NULL;
        }

        for (unsigned int i = 0; i < fileSizeBytes; i++) {
            *fileSize = (*fileSize << 8 | packet[3+i]);
        }



        if (size < 3 + fileSizeBytes + 2) {
             printf("Packet is too small to contain the file name length
byte.\n");
            return NULL;
        }

        unsigned char fileNameBytes = packet[3 + fileSizeBytes + 1];

        if (size < 3 + fileSizeBytes + 2 + fileNameBytes) {
                printf("Packet is too small to contain the declared file
```

```
name field.\n");
            return NULL;
        }

        unsigned char *name = (unsigned char*)malloc(fileNameBytes+1);
        memcpy(name, packet + 3 + fileSizeBytes + 2, fileNameBytes);
        name[fileNameBytes] = '\0';

        return name;
    }

    void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer) {
        if (packetSize < 5) {
            printf("Invalid packet size\n");
            return;
        }

        unsigned char controlField = packet[0];

        if (controlField != 2) {
            printf("Not a data packet\n");
            return;
        }

        unsigned char sequenceNumber = packet[1];

        unsigned short dataLength = (packet[2] << 8) | packet [3];

        if (packetSize < 4 + dataLength) {
            printf("Packet size does not match data length\n");
            return;
        }

        memcpy(buffer, packet + 4, dataLength);
        buffer[dataLength] = '\0';
    }

    unsigned char * getControlPacket(const unsigned int c, const char*
filename, long int length, unsigned int* size) {
```

```
        if (!filename || length < 0 || !size) return NULL;

        unsigned char fileSizeType = 0;
        unsigned char fileSizeLength = sizeof(length);
        unsigned char fileNameType = 1;
        unsigned char fileNameLength = strlen(filename);

        *size = 1 + 2 + fileSizeLength + 2 + fileNameLength;
        unsigned char *packet = (unsigned char*)malloc(*size);
        if (!packet) return NULL;

        unsigned int index = 0;
        packet[index++] = (unsigned char) c;
        packet[index++] = 0; // file size type
        packet[index++] = fileSizeLength;

        for(int i = fileSizeLength - 1; i >= 0; i--) {
            packet[index++] = (length >> (i * 8)) & 0xFF;
        }

        packet[index++] = 1; // file name type
        packet[index++] = fileNameLength;
        memcpy(packet + index, filename, fileNameLength);

        return packet;
    }

    unsigned char * getDataPacket(unsigned char sequence, unsigned char
*data, int dataSize, int *packetSize) {
        *packetSize = 4 + dataSize;

        unsigned char *packet = (unsigned char*)malloc(*packetSize);
        if (packet == NULL) return NULL;

        int index = 0;

        packet[index++] = 2; // control field for data packet
        packet[index++] = sequence;
         packet[index++] = (dataSize >> 8) & 0xFF; // high byte of data
size
```

```
        packet[index++] = dataSize & 0xFF; // low byte of data size

        memcpy(packet + index, data, dataSize);

        return packet;
    }

    unsigned char *getData(FILE* spfd, long int fileLength) {
        unsigned char *data = (unsigned char *)malloc(fileLength);
        if (!data) {
            fprintf(stderr, "Error allocating memory for file data\n");
            return NULL;
        }

             size_t  bytesRead  =  fread(data,  sizeof(unsigned  char),
fileLength, spfd);
        if (bytesRead != fileLength) {
            if (feof(spfd)) {
                printf("Info: Reached end of file\n");
            } else if (ferror(spfd)) {
                    printf("Error: An error occurred while reading the
file\n");
            }
        }
        return data;
    }
```