Protocolo de Ligação de Dados

Redes de Computadores

João das Neves Fernandes 202108867

Pedro Filipe Pinho Oliveira 202108669

Índice

1 Sumário

Este trabalho foi desenvolvido no âmbito da unidade curricular de Redes de Computadores, com o objetivo de implementar um protoloco de ligação de dados e uma aplicação simples que usa o esse protocolo para a transmissão e receção de ficheiros entre computadores ligados por uma porta de série.

Com este projeto pudemos consolidar o conhecimento sobre o protocolo de ligação de dados, o funcionamento da porta de série e rever conceitos como byte stuffing.

2 Introdução

O projeto foi realizado visando o estudo e compreensão do funcionamento de um protocolo de ligação de dados do modelo Stop & Wait, tal como a análise da sua eficiência. Também foi desenvolvido um protocolo de aplicação, que permite a transferência de ficheiros entre um computador transmissor e outro recetor utilizando a interface do protocolo de ligação. Neste relatório pode se encontrar as seguintes secções:

- Arquitetura: blocos funcionais e interfaces implementadas.
- Estrutura do Código: principais estruturas de dados e funções utilizadas no projeto, e a sua relação com a arquitetura.
- Casos de uso principais: uso da aplicação e sequência de chamadas de funções.
- Protocolo de ligação lógica: funcionalidades e implementação do protocolo de ligação lógica.
- Protocolo de aplicação: funcionalidades e implementação do protocolo de aplicação.
- Validação: testes efetuados para validação do projeto.
- Eficiência do protocolo de ligação: caracterização estatística da eficiência do protocolo implementado.
- Conclusões: síntese da informação exposta e sua discussão.

3 Arquitetura

O projeto está dividido em dois blocos funcionais: a camada do protocolo de ligação de dados, LinkLayer, e camada da aplicação. Estas camadas estão desenvolvidas de forma a serem independentes, pelo que a aplicação não precisa de saber como funciona a LinkLayer, usando apenas a sua interface.

4 Estrutura do Código

4.1 Nível de Ligação de Dados

4.1.1 Estrutura de Dados

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

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

```
typedef enum {
    START, FLAG, A, C, D,
    DD, BCC, BCC2, STOP
} State;

typedef enum {
    RCV_SET, RCV_UA, WRITE,
    READ, CLOSETX, CLOSERX
} Action;
```

4.1.2 Funções

```
int parseFrame(Action act, State* state, unsigned char* received,
               int* index);
int llopen(LinkLayer connectionParameters);
int llwrite(const unsigned char *buf, int bufSize);
```

```
int sendDataResponse(int valid, unsigned char control);
int llread(unsigned char *packet);
int llclose(int showStatistics);
```

4.2 Nível da Aplicação

4.2.1 Funções

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

5 Casos de uso

Os dois casos de uso existentes são o do transmissor e o do recetor. Para iniciar o programa corre-se o binário com os parâmetros: porta de série (/dev/ttySX), função (tx para o transmissor, rx para o recetor), e nome do ficheiro, que no transmissor vai ser o nome do ficheiro a ser enviado e no recetor o nome em que o ficheiro recebido ficará guardado. De seguida estão as sequências de chamadas de funções de cada caso.

Caso transmissor:

- **llopen()**: abre a conexão, usando as definições da porta de série que estão na struct LinkLayer.
- **applicationWrite()**: cria os pacotes do ficheiro a ser enviados, sejam de controlo ou de informação.
- **llwrite()**: cria as tramas e envia-as pela porta de série. Depois usa a função parseFrame para ler a resposta do recetor e agir acordemente.

- **llclose()**: termina a conexão, restaurando as definições originais da porta de série.

Caso recetor:

- **llopen()**: abre a conexão, usando as definições da porta de série que estão na struct LinkLayer.
- **applicationRead()**: lê os pacotes do ficheiro recebidos, sejam de controlo ou de informação.
- **llread()**: lê as tramas recebidas, fazendo parse das mesmas. Usa o sendDataResponse() para determinar a resposta a ser enviada e age acordemente com essa.
- **llclose()**: termina a conexão, restaurando as definições originais da porta de série.

6 Protocolo da Ligação Lógica

A Ligação Lógica é responsável pela comunicação entre o transmissor e o recetor e é a única camada que tem acesso à porta de série. Também é a camada responsável pela deteção de erros em tramas e ações em resposta a esses mesmos erros.

A conexão é estabelecida com o **llopen()**, onde o transmissor envia uma trama de supervisão SET e espera pela resposta do recetor, que envia uma trama de supervisão UA. Depois do transmissor receber a trama UA, a função retorna com sucesso e ambos os lados estão prontos para transmitir ou receber tramas de informação.

As tramas de informação são transmitidas pelo **llwrite()** e recebidas pelo **llread** e têm como objetivo transmitir/receber informação passada como argumento. Estas tramas contém um prefixo com algumas informações e estão delimitadas por *flags*. Por isso, precisam de um mecanismo de transparência, também conhecido como *byte stuffing*. Esse mecanismo funciona à base da codificação de octetos importantes com caractéres de escape, que posterioriamente serão descodificados pelo recetor.

No final da trama de informação também está presente um octeto para verificar erros na trama, chamado BCC2. O recetor consegue, portanto, verificar se a trama recebida

contém algum erro e rejeitar a trama se necessário. Devido ao mecanismo de Stop & Wait, o recetor também precisa de enviar respostas para o transmissor, indicando se aceitou a trama anterior, se esta continha erros ou se estava repetida. Caso o transmissor não receba nenhuma resposta, retorna a mandar depois de esperar um *timeout* e, se não obter resposta depois de um certo número de retransmissões, termina.

A função responsável por terminar a ligação é a **llclose()**, na qual o transmissor envia uma trama de supervisão DISC, espera que o recetor retorne outra DISC e, por fim, envia uma trama de supervisão UA e fecha a ligação.

7 Protocolo da Aplicação

A Aplicação é responsável pela leitura e divisão do ficheiro a enviar por parte do transmissor e pela construção e escrita do ficheiro a receber por parte do recetor. A Aplicação não tem qualquer conhecimento ou controlo sobre como é que os pacotes são enviados ou recebidos e apenas usa as funções da camada de Ligação Lógica para interagir com os pacotes, sem se preocupar com aspetos como erros e atrasos no envio.

Depois da ligação estar estabelecida com o **llopen()**, o transmissor vai ler o tamanho, nome e conteúdos do ficheiro. Depois, utilizando o **llwrite()**, vai enviar um pacote inicial de controlo do tipo TLV (type, length, value) que contém o tamanho e nome do ficheiro para o recetor saber quantos pacotes terá de receber. Depois deste pacote inicial, o transmissor divide o ficheiro em pacotes mais pequenos e envia esses pacotes de dados para o recetor, que está à espera. No final volta a enviar um pacote de controlo TLV para indicar o fim da transmissão e tenta fechar a conexão com **llclose()**.

Durante este processo, o recetor vai receber o tamanho do ficheiro no pacote de controlo TLV, calcular o número de pacotes a receber e receber cada pacote de dados, com os quais vai construir um novo ficheiro. Todos estes pacotes são recebidos unicamente com o **llread()**. Quando o recetor atingir o número de pacotes pretendido, confirma o pacote de controlo final e procede a fechar a ligação com **llclose()**.

8 Validação

Desenvolvemos um programa robusto para o protocolo Stop & Wait, capaz de resistir a erros e desconexões momentâneas durante o envio dos ficheiros. Testamos o programa em vários cenários, virtualmente com *socat* e fisicamente em laboratório. Realizamos testes com diferentes tamanhos de ficheiros, tramas e *baudrates*. Os principais testes foram desconexões físicas e ruído na porta de série física por meio de curtos-circuitos. Os mesmos parâmetros foram testados virtualmente com a ajuda de simulação do tempo de propagação e a simulação de taxa de erros das tramas na transmissão.

Em todos estes casos o programa teve o comportamento esperado e só fechou depois de realizar o número máximo de retransmissões, com um tempo de espera de um *timeout* para cada retransmissão.

9 Estatísticas

9.1 Variação da Capacidade de Ligação (*Baudrate*)

Nesta estatística, variámos o *baudrate* para ver como afetava o bitrate e a eficiência do nosso programa. Quando o *baudrate* é muito baixo, por exemplo em 2400, o mecanismo comporta-se de forma inesperada (ver resultados). Depois, à medida que vai aumentando, a eficiência vai gradualmente diminuindo e, quando é muito alto, atinge um limite e mantém-se constante.

9.2 Variação do Tamanho do Pacote de Dados)

Nesta estatística, variámos o tamanho dos pacotes de dados, que por sua vez vai alterar o tamanho das tramas de informação (I), e observamos alterações no bitrate e na eficiência do programa. Concluímos que à medida que o tamanho aumenta, a eficiência também aumenta. No entanto, é importante notar que, em situações com falhas na transmissão (FER > 0), a eficiência pode diminuir com o aumento do tamanho das tramas, embora esse cenário não esteja aqui representado.

9.3 Variação do Frame Error Rate (FER)

Nesta estatística, simulamos uma taxa de erros nas tramas que eram enviadas e estudamos a sua influência na eficiência do programa. Concluimos que a eficiência diminui muito com o aumento da taxa de erros, o que era de esperar, uma vez que são necessários mais reenvios por parte do transmissor.

9.4 Variação do Tempo de Propagação

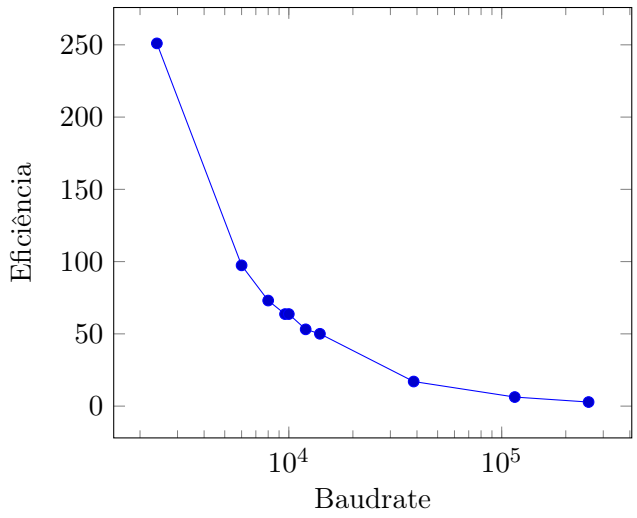
Nesta estatística, simulamos o tempo de propagação do transmissor e do recetor ao enviar/receber tramas. A conclusão retirada foi que um aumento no tempo de propagação corresponde a uma diminuição da eficiência, que foi o esperado visto que, com o aumento do tempo de propagação, o tempo total será maior para a mesma quantidade de bits.

10 Conclusão

Em resumo, o projeto teve foco na implementação do protocolo de ligação de dados, utilizando a modalidade de Stop & Wait, assim como a análise de eficiência do protocolo implementado. Foram seguidas as especificações expostas no guião de trabalho, mantendo a independência pedida entre as camadas de ligação e de aplicação.

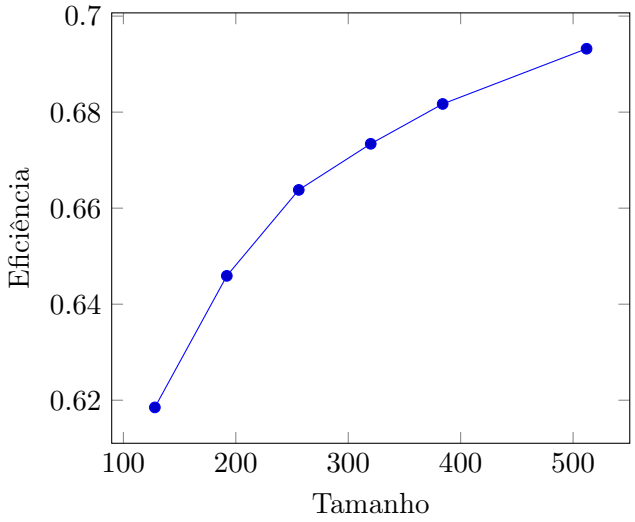
Este projeto permitiu consolidar os conceitos e teoria dada nas aulas teóricas, permitindo um conhecimento mais profunto sobre o protocolo de ligação de dados.

11 Anexo A - Estatísticas



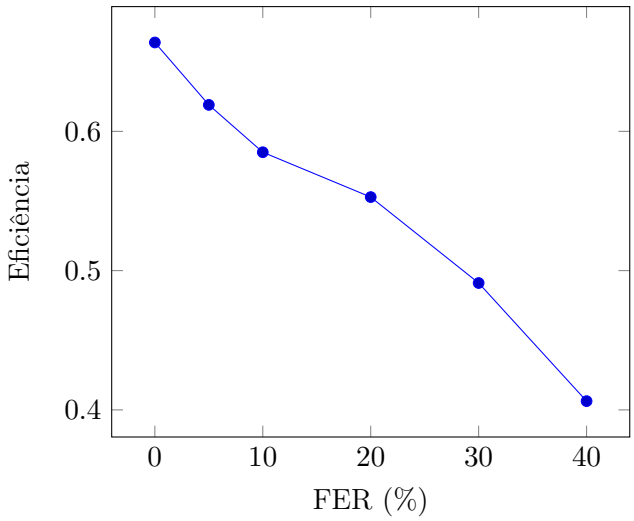
Baudrate (bit/s)	Tempo (s)	Bitrate (bit/s)	Eficiência (%)
2400	14.565761	6023.990	250.999
6000	15.013909	5844.181	97.402
8000	15.014347	5844.010	73.050
9600	13.778426	6368.216	63.682
10000	13.774104	6370.214	63.702
12000	13.762885	6375.448	53.128
14000	12.522729	7006.779	50.048
38600	13.353057	6571.080	17.014
115200	12.149927	7221.771	6.275
256000	12.151274	7220.971	2.821

Table 1: Variação do Baudrate com Tempo, Bitrate e Eficiência



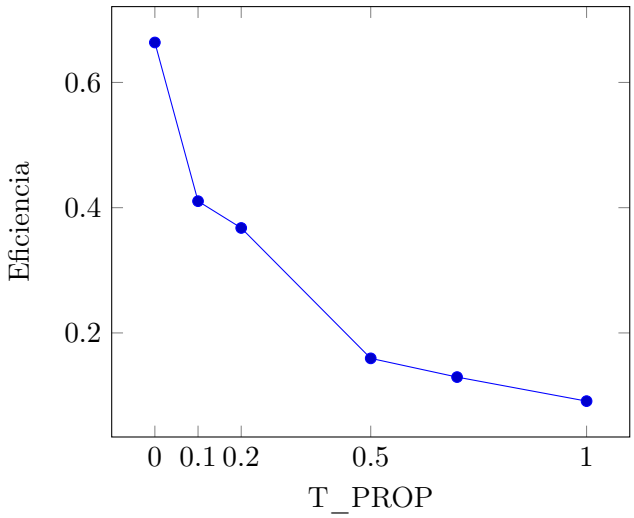
Tamanho (bytes)	Tempo (s)	Bitrate (bit/s)	Eficiência (%)
128	14.751256	5948.239	61.85
192	14.144861	6203.242	64.59
256	13.778426	6368.216	66.38
320	13.563178	6468.480	67.34
384	13.408285	6544.014	68.17
512	13.185520	6654.725	69.32

Table 2: Variação do Tamanho da Trama com Tempo, Bitrate e Eficiência



FER (%)	Tempo (s)	Bitrate (bit/s)	Eficiência (%)
0	13.778426	6368.216	66.38
5	14.730805	5956.496	61.90
10	15.366808	5709.969	58.50
20	16.327363	5374.046	55.28
30	18.611284	4714.559	49.11
40	22.436842	3910.710	40.63

Table 3: Variação do FER com Tempo, Bitrate e Eficiência



T_PROP	Tempo (s)	Bitrate (bit/s)	Eficiencia (%)
0	13.778426	6368.216	66.38
0.1	22.372953	3945.547	41.04
0.2	30.975263	3528.682	36.76
0.5	56.772846	1534.413	15.94
0.7	70.374259	1243.865	12.95
1	99.772940	878.028	9.11

Table 4: Variação do T_PROP com Tempo, Bitrate, Eficiencia e Estimated Bitrate

12 Anexo B - Código Fonte

```
// Link layer protocol implementation

#include "link_layer.h"

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

#define TRUE 1
#define FALSE 0

#define FLAG_RCV 0x7E
#define A_T 0x03
#define A_R 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B
#define CI_0 0x00
#define CI_1 0x40
#define ESC 0x7D
#define ESC_XOR 0x20
#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81
```

```
// GLOBALS
int alarmEnabled = FALSE;
int alarmCount = 0;
int fd = 0;
struct termios oldtio; // old settings to restore
int frameNumber = 0;
unsigned char escFlag[] = {ESC, 0x5E};
unsigned char escEsc[] = {ESC, 0x5d};
int timout = 0;
int nRetransmissions = 0;
LinkLayerRole role;


long bytesSent = 0;
long bytesReceived = 0;
int errorsSent = 0;
int errorsReceived = 0;


int DEBUG = FALSE;


int SIM_ERROR = FALSE;
int ERROR_RATE = 10; // % error rate (0-100)


// Alarm function handler
void alarmHandler(int signal) {
```

```
    alarmEnabled = FALSE;
    alarmCount++;
    if(DEBUG) printf("Alarm triggered, #%d\n", alarmCount);
}

int cHandler(Action act, unsigned char buf) {
    switch(act) {
        case RCV_SET:
            if(buf == C_SET) return 1;
            break;
        case RCV_UA:
            if(buf == C_UA) return 1;
            break;
        case WRITE:
            if(buf == RR0 || buf == RR1 || buf == REJ0 ||
            buf == REJ1)
                return 1;
            break;
        case READ:
            if(buf == CI_0 || buf == CI_1) return 1;
        case CLOSETX:
        case CLOSERX:
            if(buf == C_DISC) return 1;
            break;
        default:
            break;
    }
    return 0;
```

```
}

int parseFrame(Action act, State* state, unsigned char* received,
int* index) {

    int stop = FALSE;
    unsigned char buf;
    int bytes = read(fd, &buf, 1);
    if(bytes < 1) {
        return stop;
    }
    bytesReceived += bytes;
    switch(*state) {
        case START:
            if(buf == FLAG_RCV) {
                *state = FLAG;
                received[*index] = buf;
                *index+=1;
            }
            break;
        case FLAG:
            if(buf == A_R && act == CLOSETX) {
                *state = A;
                received[*index] = buf;
                *index+=1;
            }
            else if (buf == A_T && act != CLOSETX) {
                *state = A;
```

```
            received[*index] = buf;
            *index+=1;
        }
        else if(buf != FLAG_RCV) {
            *state = START;
            *index = 0;
        }
        break;
    case A:
        if(cHandler(act, buf)) {
            *state = C;
            received[*index] = buf;
            *index+=1;
        }
        else if(buf == FLAG_RCV) {
            *state = FLAG;
            *index = 1;
        }
        else {
            *state = START;
            *index = 0;
        }
        break;
    case C:
        if(buf == (received[1] ^ received[2])) {
            *state = BCC;
            received[*index] = buf;
            *index+=1;
```

```
            }
            else if(buf == FLAG_RCV) {
                *state = FLAG;
                *index = 1;
            }
            else {
                *state = START;
                *index = 0;
            }
            break;
        case BCC:
            if(buf == FLAG_RCV) {
                stop = TRUE;
                received[*index] = buf;
                *index+=1;
            }
            else {
                *state = START;
                *index = 0;
            }
            break;
        default:
            break;
    }
    return stop;
}
```

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

    const char *serialPortName = connectionParameters.serialPort;
    role = connectionParameters.role;
    nRetransmissions = connectionParameters.nRetransmissions;
    timout = connectionParameters.timeout;

    (void)signal(SIGALRM, alarmHandler);

    fd = open(serialPortName, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        perror(serialPortName);
        exit(-1);
    }

    struct termios newtio;

    if (tcgetattr(fd, &oldtio) == -1) {
        perror("tcgetattr");
        exit(-1);
    }

    memset(&newtio, 0, sizeof(newtio));
```

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

    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0;
    newtio.c_cc[VMIN] = 0;

    tcflush(fd, TCIOFLUSH);

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

// ------------------------------------------------------------

    int stop = FALSE;
    unsigned char received[5] = {0};
    int index = 0;
    State state = START;
    unsigned char set_command[] = {FLAG_RCV, A_T, C_SET,
                                   A_T ^ C_SET, FLAG_RCV};
    unsigned char ua_reply[] = {FLAG_RCV, A_T, C_UA,
                                A_T ^ C_UA, FLAG_RCV};
    int nRepeated = 0;
```

```
switch (role) {
    case LlTx:

        while(stop == FALSE && nRepeated < nRetransmissions)
        {

            int byte1 = write(fd, set_command, 5);
            if(byte1 < 5) {
                printf("Error writing SET\n");
                return -1;
            }
            bytesSent += byte1;
            if(DEBUG)
                printf("%d bytes written (SET)\n", byte1);

            while (stop == FALSE && alarmCount < timout) {
                if (alarmEnabled == FALSE) {
                    alarm(1);
                    // Set alarm to be triggered in 1s
                    alarmEnabled = TRUE;
                }

                stop = parseFrame(RCV_UA, &state, received,
                        &index);

            }
            alarmCount = 0;
            nRepeated++;
```

```
        }
        if(stop  == FALSE) {
            printf("Error receiving UA\n");
            return -1;
        }
        break;


    case LlRx:

        while (stop == FALSE) {
            stop = parseFrame(RCV_SET, &state, received,
                    &index);
        }

        if(write(fd, ua_reply, 5) < 5) {
            printf("Error writing UA\n");
            return -1;
        }
        bytesSent += 5;
        break;


    default:
        break;
}

return 0;
```

```
}

void writeByte(const unsigned char* byte, unsigned char* buffer,
int* idx) {

    if(*byte == FLAG_RCV) {
        memcpy(&buffer[*idx], escFlag, 2);
        *idx = *idx + 2;
    } else if(*byte == ESC) {
        memcpy(&buffer[*idx], escEsc, 2);
        *idx = *idx + 2;
    } else {
        memcpy(&buffer[*idx], byte, 1);
        *idx = *idx + 1;
    }
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize) {

    unsigned char control = frameNumber == 0 ? CI_0 : CI_1;
    unsigned char header[] = { FLAG_RCV, A_T, control,
                               A_T ^ control};
    unsigned char buffer[bufSize * 2];
    int idx = 4;
```

```
memcpy(buffer, header, 4);

unsigned char bcc2 = 0;

int bytesWritten = 0;
while(bytesWritten < bufSize) {
    bcc2 ^= buf[bytesWritten];
    writeByte(&buf[bytesWritten++], buffer, &idx);
}

writeByte(&bcc2, buffer, &idx);
buffer[idx++] = FLAG_RCV;
int bytes;
int stop = FALSE;
int good_packet = FALSE;
int retransmission = TRUE;
State state = START;
int nRepeated = 0;
int error;

while(stop == FALSE && nRepeated <= nRetransmissions) {

    if(retransmission == TRUE) {
        // simulate error
        if(SIM_ERROR) {
            error = rand() % 10000;
            if(error < ERROR_RATE * 100) {
                if(DEBUG)
```

```
                        printf("Simulating error on frame number
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d...\n", frameNumber);
                    errorsSent++;
                    buffer[4] = buffer[4] ^ 0xFF; // flips a byte
                }
            }
            bytes = write(fd, buffer, idx);
            nRepeated++;
            if(SIM_ERROR && error < ERROR_RATE * 100) {
                buffer[4] = buffer[4] ^ 0xFF; // undo flip
            }
            if(bytes < idx) {
                printf("Error writing DATA\n");
                return -1;
            }
            bytesSent += bytes;
        }
        unsigned char received[5] = {0};
        int index = 0;
        good_packet = FALSE;
        state = START;

        while (good_packet == FALSE && alarmCount < timout) {
            if (alarmEnabled == FALSE) {
                alarm(1);
                alarmEnabled = TRUE;
            }
            good_packet = parseFrame(WRITE, &state, received,
```

```
                    &index);
}
int response_number = received[2] == RR0 ? 0 : 1;
switch(received[2]){
    case RR0:
    case RR1:
        if(DEBUG)
            printf("RR%d received\n", response_number);
        if(frameNumber != response_number) {
            frameNumber = response_number;
            stop = TRUE;
        }
        else {
            retransmission = FALSE;
            nRepeated = 0;
        }
        break;
    case REJ0:
        retransmission = frameNumber == 0 ? TRUE : FALSE;
        if(DEBUG)
            printf("REJ0 received retransmission: %d\n",
            retransmission);
        nRepeated = 0;
        break;
    case REJ1:
        retransmission = frameNumber == 1 ? TRUE : FALSE;
        if(DEBUG)
            printf("REJ1 received , retransmission: %d\n",
```

```
                    retransmission);
                nRepeated = 0;
                break;
            default:
                break;
        }
        alarmCount = 0;

    }
    if(nRepeated >= nRetransmissions) {
        printf("Error sending frame due to max number of
        retransmissions\n");
        return -1;
    }
    return bytes;
}

int sendDataResponse(int valid, unsigned char control) {
    unsigned char responseC;
    int accept = FALSE;
    if(valid) {
        if(control == frameNumber) {
            responseC = frameNumber == 0 ? RR1 : RR0;
            frameNumber ^= 1;
            accept = TRUE;
        } else {
            responseC = frameNumber == 0 ? RR0 : RR1;
            accept = FALSE;
```

```
        }
        if(DEBUG)
        printf("packet␣received,␣RR%d␣sent\n",
        responseC == RR0 ? 0 : 1);
    } else {
        if(control == frameNumber) {
            if(DEBUG) printf("error␣received,␣REJ%d␣sent\n",
            frameNumber);
            responseC = frameNumber == 0 ? REJ0 : REJ1;
            accept = FALSE;
        } else {
            if(DEBUG) printf("error␣received,␣RR%d␣sent\n",
            frameNumber);
            responseC = frameNumber == 0 ? RR0 : RR1;
            accept = FALSE;
        }
        errorsReceived++;
    }
    unsigned char response[] = {FLAG_RCV, A_T, responseC,
                                A_T ^ responseC, FLAG_RCV};
    int bytes = write(fd, response, 5);
    if(bytes < 5) {
        printf("Error␣writing␣response\n");
        return -1;
    }
    bytesSent += bytes;
    if(DEBUG) printf("%d␣bytes␣data␣response␣written\n", bytes);
    return accept;
```

```
}

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

    State state = START;
    int stop = FALSE;
    int index = 0;
    unsigned char bcc2 = 0;
    unsigned char buf;
    unsigned char control;

    while(stop == FALSE) {

        int bytes = read(fd, &buf, 1);
        if(bytes < 1) continue;
        bytesReceived += bytes;

        switch (state) {
            case START:
                if(buf == FLAG_RCV) {
                    state = FLAG;
                }
                break;
            case FLAG:
                if(buf == A_T) {
```

```
            state = A;
        }
        else if(buf != FLAG_RCV) {
            state = START;
        }
        break;
    case A:
        if(cHandler(READ, buf)) {
            control = buf;
            state = C;
        }
        else if(buf == FLAG_RCV) {
            state = FLAG;
        }
        else {
            state = START;
        }
        break;
    case C:
        if(buf == (A_T ^ control)) {
            state = D;
        }
        else if(buf == FLAG_RCV) {
            state = FLAG;
        }
        else {
            state = START;
        }
```

```
        break;
    case D:
        if(buf == FLAG_RCV) {
            unsigned char bcc2Received = packet[--index];
            bcc2 ^= bcc2Received;
            packet[index] = '\0';
            int accept =
                sendDataResponse(bcc2Received == bcc2,
                control == CI_0 ? 0 : 1);
            if(accept == -1) {
                return -1;
            }
            if(accept == TRUE) {
                stop = TRUE;
            }
            else {
                state = START;
                index = 0;
                bcc2 = 0;
            }
        } else if (buf == ESC) {
            state = DD;
        } else {
            packet[index++] = buf;
            bcc2 ^= buf;
        }
        break;
    case DD:
```

```
                packet[index++] = buf ^ ESC_XOR;
                bcc2 ^= buf ^ ESC_XOR;
                state = D;
                break;
            default:
                break;
        }

    }

    return index;
}

int sendDISC() {
    unsigned char disc[] = {FLAG_RCV,
                            role == LlTx ? A_T : A_R,
                            C_DISC,
                            (role == LlTx ? A_T : A_R) ^ C_DISC,
                            FLAG_RCV};
    int bytes = write(fd, disc, 5);
    bytesSent += bytes;
    if(bytes < 5) {
        printf("Error writing DISC\n");
        return -1;
    }
    if(DEBUG) printf("%d bytes DISC written\n", bytes);
    return 0;
}
```

```
////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {
    int stop = FALSE;
    State state = START;
    int index = 0;
    unsigned char received[5] = {0};
    unsigned char ua_reply[] = {FLAG_RCV, A_T, C_UA,
                                A_T ^ C_UA, FLAG_RCV};

    switch (role) {
        case LlTx:
            while(stop == FALSE) {
                if(sendDISC() == -1) {
                    printf("Error sending DISC\n");
                    return -1;
                }
                while(stop == FALSE && alarmCount < timout) {
                    if (alarmEnabled == FALSE) {
                        alarm(1);
                        // Set alarm to be triggered in 1s
                        alarmEnabled = TRUE;
                    }
                    stop = parseFrame(CLOSETX, &state, received,
                            &index);
                }
```

```
            alarmCount = 0;
        }
        if(stop == TRUE){
            if(DEBUG) printf("DISC received\n");
            if(write(fd, ua_reply, 5) < 5) {
                printf("Error writing UA\n");
                return -1;
            }
            bytesSent += 5;
        }
        break;
    case LlRx:
        while (stop == FALSE) {
            stop = parseFrame(CLOSERX, &state, received,
                    &index);
        }
        stop = FALSE;
        while (stop == FALSE)
        {
            if(sendDISC() == -1) {
                printf("Error sending DISC\n");
                return -1;
            }
            while (stop == FALSE && alarmCount < timout) {
                if (alarmEnabled == FALSE) {
                    alarm(1);
                    // Set alarm to be triggered in 1s
```

```
                    alarmEnabled = TRUE;
                }
                stop = parseFrame(RCV_UA, &state, received,
                        &index); // RECEIVES UA
            }
            alarmCount = 0;
        }
        break;
    default:
        break;
}

if(showStatistics) {
    printf("Error frames sent: %d\n", errorsSent);
    printf("Error frames received: %d\n", errorsReceived);
    printf("Total Bytes Sent: %ld\n",
    bytesSent);
    printf("Total Bytes Received: %ld\n",
    bytesReceived);
}

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

close(fd);
```

```
    return 0;
}

// Application layer protocol implementation

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

#define PACKET_SIZE 256
#define CONTROL_DATA 0x01
#define CONTROL_START 0x02
#define CONTROL_END 0x03
#define FILE_SIZE_T 0x00
#define FILE_NAME_T 0x01

extern int DEBUG;
long globalFileSize = 0;

int applicationWrite(const char *filename) {
    FILE *file = fopen(filename, "r");

    if(file == NULL) {
        printf("Failed to open file\n");
        return 1;
    }

    fseek(file, 0, SEEK_END);
```

```
long fileSize = ftell(file);
globalFileSize = fileSize;

fseek(file, 0, SEEK_SET);

long nPackets = ((fileSize + PACKET_SIZE) / PACKET_SIZE);
int nameSize = strlen(filename);
// Control Packet -> 0x02 /
//                   0x00 /
//                   size of fileSize /
//                   fileSize
//                   (/ 0x01 /
//                   size of filename/
//                   filename)

int fileSizeBytes = 0;
long aux = fileSize;
while(aux > 0) {
    aux = aux >> 8;
    fileSizeBytes++;
}

unsigned char* controlPacket = malloc(5 + fileSizeBytes +
                                      nameSize);
controlPacket[0] = CONTROL_START;
controlPacket[1] = FILE_SIZE_T;
controlPacket[2] = fileSizeBytes;
```

```
    aux = fileSize; int i = 0;
    while(i < fileSizeBytes) {
        controlPacket[3 + i] = aux & 0xFF;
        aux = aux >> 8;
        i++;
    }
    controlPacket[fileSizeBytes + 3] = FILE_NAME_T;
    controlPacket[fileSizeBytes + 4] = nameSize;
    for(int i = 0; i < nameSize; i++) {
        controlPacket[fileSizeBytes + 5 + i] = filename[i];
    }

    if(DEBUG){
        printf("Printing control packet:\n");
        for(int i = 0; i < 5 + fileSizeBytes + nameSize; i++) {
            printf("0x%x ", controlPacket[i]);
        }
        printf("\n");
    }
    if(llwrite(controlPacket, 5 + fileSizeBytes + nameSize) < 5 + fileSizeBytes +
        printf("Failed to send control packet\n");
        return 1;
    }

    controlPacket[0] = CONTROL_END;
//-------------------------------------------------------------

    // Data Packet -> 0x01 /
```

```
//                  byte 1 of n of bytes /
//                  byte 2 of n of bytes /
//                  packets...

unsigned char dataPacket [PACKET_SIZE + 3];

dataPacket[0] = CONTROL_DATA;
dataPacket[1] = (PACKET_SIZE >> 8) & 0xFF;
dataPacket[2] = PACKET_SIZE & 0xFF;

long nPacket = 0;

while(nPackets--) {
    for(int i = 0; i < PACKET_SIZE; i++) {
        int byte = fgetc(file);
        dataPacket[3 + i] = (unsigned char) byte;
        if(byte == EOF) {
            dataPacket[1] = (i >> 8) & 0xFF;
            dataPacket[2] = i & 0xFF;
            if(llwrite(dataPacket, 3 + i) < 3 + i) {
                printf("Failed to send data packet\n");
                return 1;
            }
            if(llwrite(controlPacket,
                    5 + fileSizeBytes + nameSize)
            < 5 + fileSizeBytes + nameSize) {
                printf("Failed to send control packet\n");
                return 1;
```

```
                }
                free(controlPacket);
                fclose(file);
                return 0;
            }
        }
        if(llwrite(dataPacket, PACKET_SIZE + 3)
        < PACKET_SIZE + 3) {
            printf("Failed to send data packet\n");
            return 1;
        }
        if(DEBUG) printf("Data packet %ld \n", nPacket);

        nPacket++;
    }

    if(llwrite(controlPacket, 5 + fileSizeBytes + nameSize)
    < 5 + fileSizeBytes + nameSize) {
            printf("Failed to send control packet\n");
            return 1;
    }
    fclose(file);
    free(controlPacket);
    return 0;
}

int applicationRead(const char *filename) {
    FILE *file = fopen(filename, "w");
```

```
if(file == NULL) {
    printf("Failed to open file\n");
    return 1;
}

unsigned char controlPacket[517] = {0}; // 5 + 2^8 * 2

int bytes = llread(controlPacket);
if(bytes < 7) {
    printf("Failed to receive control packet\n");
    return 1;
}

if(DEBUG){
    printf("Printing control packet:\n");
    for(int i = 0; i < bytes; i++) {
        printf("0x%x ", controlPacket[i]);
    }
    printf("\n");}

if(controlPacket[0] != CONTROL_START) {
    printf("Invalid control packet: Start was 0x%x\n",
    controlPacket[0]);
    return 1;
}

long fileSize = 0;
```

```
    if(controlPacket[1] != FILE_SIZE_T) {
        printf("Invalid control packet:
        File size type was 0x%x\n", controlPacket[1]);
        return 1;
    }

    for(int i = 0; i < controlPacket[2]; i++) {
        fileSize += (controlPacket[3 + i] << (8*i));
    }
    if(DEBUG) printf("filesize: %ld\n",fileSize);
    globalFileSize = fileSize;

    if(controlPacket[3 + controlPacket[2]] != FILE_NAME_T) {
        printf("Invalid control packet:
        File name type was 0x%x\n",
        controlPacket[3 + controlPacket[2]]);
        return 1;
    }

    int nameSize = controlPacket[4 + controlPacket[2]];
    char* name = malloc(nameSize);
    for(int i = 0; i < nameSize; i++) {
        name[i] = controlPacket[5 + controlPacket[2] + i];
    }

    long nPackets = ((fileSize + PACKET_SIZE) / PACKET_SIZE);
    long aux = nPackets;
```

```
if(DEBUG) printf("nPackets: %ld\n", nPackets);

unsigned char dataPacket [PACKET_SIZE + 3 +1];

while(nPackets--) {
    if(llread(dataPacket) < 3) {

        printf("Failed to receive data packet\n");
        return 1;
    }

    if(dataPacket[0] != CONTROL_DATA) {
        printf("Invalid data packet, byte 0:%x\n",
        dataPacket[0]);
        return 1;
    }
    int packetSize = (dataPacket[1] << 8) + dataPacket[2];

    if(DEBUG)
    printf("Data packet %ld\n", aux - nPackets - 1);

    fwrite(dataPacket + 3, 1, packetSize, file);

}

memset(controlPacket, 0, 517);

if(llread(controlPacket) < 7) {
```

```
        printf("Failed to receive control packet\n");
        return 1;
    }

    if(controlPacket[0] != CONTROL_END) {
        printf("Invalid control packet. End was 0x%x\n",
        controlPacket[0]);
        return 1;
    }

    if(controlPacket[1] != FILE_SIZE_T) {
        printf("Invalid control packet- File size t was 0x%x\n",
        controlPacket[1]);
        return 1;
    }

    long fileSize2 = 0;
    for(int i = 0; i < controlPacket[2]; i++) {
        fileSize2 += (controlPacket[3 + i] << (8*i));
    }

    if(fileSize != fileSize2) {
        printf("FileSize does not match\n");
        return 1;
    }

    if(controlPacket[3 + controlPacket[2]] != FILE_NAME_T) {
        printf("Invalid control packet\n");
```

```
        return 1;
    }

    int nameSize2 = controlPacket[4 + controlPacket[2]];

    if(nameSize != nameSize2) {
        printf("NameSize does not match\n");
        return 1;
    }

    for(int i = 0; i < nameSize; i++) {
        if(name[i] != controlPacket[5 + controlPacket[2] + i]) {
            printf("Name does not match\n");
            return 1;
        }
    }

    if(DEBUG)
    printf("\nFile with name %s and size %ld
    received and named %s\n", name, fileSize, filename);

    fclose(file);

    free(name);

    return 0;
}
```

```
void applicationLayer(const char *serialPort, const char *role,
int baudRate, int nTries, int timeout, const char *filename) {
    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort, serialPort);
    if(strcmp(role, "rx") == 0)
        connectionParameters.role = LlRx;
    else if(strcmp(role, "tx") == 0)
        connectionParameters.role = LlTx;
    else return;
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    if(llopen(connectionParameters)) {
        printf("Failed to open connection\n");
        return;
    }

    clock_t start = clock();

    switch (connectionParameters.role) {
        case LlTx:
            applicationWrite(filename);
            break;

        case LlRx:
```

```
            applicationRead(filename);
            break;

        default:
            break;
    }

    clock_t end = clock();

    if(llclose(TRUE)) {
        printf("Failed to close connection\n");
        return;
    }

    printf("Time elapsed: %f\n",
    (double)(end - start) / CLOCKS_PER_SEC);
    double bitRate = (double) globalFileSize * 8 /
                    (double)(end - start) * CLOCKS_PER_SEC;
    printf("Bitrate: %f\n", bitRate);

    return;

}
```