

1º Trabalho Laboratorial: Ligação de Dados

Relatório



Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Turma 1 Grupo 2:

André Cruz - 201503776
Bruno Piedade - 201505668
Edgar Carneiro - 201503748

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

7 de Novembro de 2017

Conteúdo

1	Sumário	3
2	Introdução	3
3	Arquitetura	3
4	Estrutura do Código	4
5	Casos de uso principais	6
6	Protocolo de ligação lógica	7
7	Protocolo de aplicação	8
8	Validação	8
9	Eficiência do protocolo de ligação de dados	9
10	Conclusões	10
11	Anexo I	12
12	Anexo II	13
13	Anexo III	16

1 Sumário

Este trabalho foi realizado no âmbito da cadeira de Redes de Computadores, e era pedido aos alunos a implementação de um protocolo de comunicação assíncrona para a transmissão de um ficheiro através de uma Porta Série RS-232.

As principais conclusões obtidas no desenvolvimento do Relatório foram que o protocolo implementado foi compreensivo o suficiente para o resultado esperado e escala conforme a teoria lecionada nas aulas teóricas. O protocolo executa com cerca de 77% de eficácia, como podemos ver pelas estatísticas geradas.

2 Introdução

O trabalho tinha como objetivo a implementação de um protocolo de ligação de dados, de acordo como uma especificação fornecida, através de um guião. Era também pedido aos alunos que desenvolvessem um simples aplicação, de forma a testar o protocolo implementado.

O Relatório encontra-se dividido em diversas secções, nas quais se pode encontrar a seguinte informação:

- **Arquitetura**, onde são descritos os diferentes blocos funcionais e interfaces.
- **Estrutura do código**, apresentando as API's, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais**, onde são identificados os principais casos de uso e as suas sequências de chamada de funções.
- **Protocolo de ligação lógica**, identificando os principais aspetos funcionais, bem como a descrição da estratégia de implementação.
- **Protocolo de aplicação**, identificando os principais aspetos funcionais, bem como a descrição da estratégia de implementação.
- **Validação**, descrevendo os testes efetuados.
- **Eficiência do protocolo de ligação de dados**, onde é realizada a caracterização estatística da eficiência do protocolo implementado.
- **Conclusões**, onde é feita uma tese da informação apresentada nas secções anteriores, bem como uma reflexão sobre os objectivos de aprendizagem alcançados.

3 Arquitetura

Blocos Funcionais

No Trabalho é possível distinguir a existência de duas camadas bem definidas: a camada do protocolo de ligação de dados - *LinkLayer* - e a camada da aplicação - *ApplicationLayer*. Os ficheiros *LinkLayer.h* e *LinkLayer.c* representam a camada de ligação de dados. Os ficheiros *ApplicationLayer.h*, *ApplicationLayer.c*, *Packets.h* e *Packets.c* representam a camada da aplicação.

A camada de ligação de dados é a camada responsável pelo estabelecimento de ligação e, portanto, tem todas as funções que asseguram a consistência do protocolo, como o tratamento de erros, envio de mensagens de comunicação, entre outros. É também nesta camada que a interação com a porta série é feita, nomeadamente, a sua abertura, a escrita e leitura desta e o seu fecho.

A camada da aplicação é responsável pela envio e receção de ficheiros, segmentando o ficheiro a enviar em tramas de tamanho definível pelo utilizador. Esta camada faz uso da interface da camada de ligação de dados, chamando as suas funções para o envio e receção de segmentos do ficheiro a receber / enviar. A camada da aplicação é sub-dividida em duas sub-camadas, daí o uso dos ficheiros *ApplicationLayer.h* e *ApplicationLayer.c* para representar a camada mais abstrata, responsável pelo envio do ficheiro e a receção do ficheiro, e que faz uso da camada menos abstrata, representada nos ficheiros *Packets.h* e *Packets.c*, que é responsável pela segmentação do ficheiro em pacotes e envio de pacotes de controlo e informação.

Interface

Na interface da linha de comandos é permitido ao utilizador correr o programa usando o mesmo binário, independentemente de ser o recetor ou o emissor. É necessário o utilizador especificar se será o emissor / recetor, qual o Serial Port a ser usado e, no caso do recetor, qual o ficheiro a transmitir. No entanto, existem parâmetros opcionais que permitem definir outras definições relacionadas com a transmissão de informação, tais como: *baudrate*, tamanho dos segmentos de informação, número de tentativas no reenvio de tramas e tempo esperado até ao reenvio de uma trama. Assim, a aplicação pode correr com valores inseridos pelo utilizador, ou com os seus valores por defeito.

4 Estrutura do Código

Application Layer

Os ficheiros *ApplicationLayer.h* e *ApplicationLayer.c*, representantes da sub-camada mais abstrata da camada da aplicação, fazem uso de uma estrutura de dados que guarda o descritor do ficheiro da porta série, o nome do ficheiro a ser transmitido, o tamanho máximo de mensagem a ser transmitido e ainda o tipo de conexão a ser usado - emissor ou recetor.

```
typedef struct {
    int fd; // serial port's file descriptor
    char * fileName;
    ConnectionType type;
    int maxDataMsgSize;
} ApplicationLayer;
```

As principais funções desta sub-camada são:

```
int initApplicationLayer(const char * port, int baudrate, int timeout, int
    numRetries, ConnectionType type, int maxDataMsgSize, char * file);
void destroyApplicationLayer();
int sendFile();
```

```
int receiveFile();
```

Os ficheiros *Packets.h* e *Packets.c*, representantes da sub-camada menos abstrata da camada da aplicação, fazem uso de três estruturas de dados: a estrutura *Packet* que guarda um apontador para a informação, e o tamanho dessa informação; a estrutura *DataPacket* que guarda o número sequencial do pacote a ser enviado, o seu tamanho e o apontador para essa informação; a estrutura *ControlPacket* que guarda o tipo de pacote de Controlo - início ou fim -, o nome do ficheiro, o tamanho do ficheiro e o número de argumentos do pacote de controlo.

```
typedef struct {
    uchar * data;
    uint size;
} Packet;
```

```
typedef struct {
    uchar seqNr;
    uint size;
    uchar * data;
} DataPacket;
```

```
typedef struct {
    PacketType type;
    char fileName[MAX_FILE_NAME];
    uint fileSize;
    uint argNr;
} ControlPacket;
```

As principais funções desta sub-camada são:

```
int sendDataPacket(int fd, DataPacket * src);
int sendControlPacket(int fd, ControlPacket * src);
int receiveDataPacket(int fd, DataPacket * dest);
int receiveControlPacket(int fd, ControlPacket * dest);
```

Link Layer

A camada da ligação de dados é representada através de uma estrutura de dados onde é guardado a porta série utilizada, o *baudrate* utilizado, o número de sequência da trama esperada, tempo esperado até ao reenvio de uma trama, e o número de tentativas de reenvio de uma trama.

```
typedef struct {
    char port[MAX_PORT_NAME];
    int baudRate;
    uint seqNumber;
    uint timeout;
    uint numRetries;
} LinkLayer;
```

As principais funções desta camada são:

```
int initLinkLayer(int porta, int baudRate, uint timeout, uint numTransmissions);
int openSerialPort();
int llopen(ConnectionType type);
int llclose(int fd);
int llwrite(int fd, uchar ** buffer, int length);
int llread(int fd, uchar ** buffer);
```

5 Casos de uso principais

Existem dois casos de uso principais bem distintos: correr o programa como emissor ou correr o programa como recetor. Em cada um destes casos é possível correr o programa usando o mesmo binário, apenas dependendo os argumentos usados na chamada do programa, sendo estes:

```
printf("Usage:\t%s <SerialPort> <r/w> <FILE_NAME> [BAUDRATE] [DATA_BYTES]  
[NUM_RETRIES] [TIMEOUT]\n", progName);  
printf("Arguments between [ ] are optional\n");
```

No caso em que o programa é executado como **recetor** a sequência de chamada de funções, considerando as de maior relevância, é:

- **receiveFile**, que tem como objetivo receber o ficheiro indicado e que faz uso de funções como a **receiveControlPacket**, **receiveDataPacket**, **llopen** e **llclose**.
- **receiveControlPacket**, que tem como objetivo enviar um pacote de controlo, do tipo *START* no início da transmissão e do tipo *END* no fim da transmissão, e que faz uso de funções como **fillControlPacketArg** e **llread**.
- **receiveDataPacket**, que tem como objetivo enviar um pacote de informação, e que faz uso de funções como **llread**.
- **fillControlPacketArg**, que tem como objetivo preencher os argumentos de um control packet, com informação recebida.
- **llread**, que tem como objetivo ler da Porta Série informação, aplicando-lhe *Byte Destuffing* e *Deframing*. Faz uso das funções **byteDestuffing**, **deframingInformation**, **sendControlFrame** e **read**.

No caso em que o programa é executado como **emissor** a sequência de chamada de funções, considerando as de maior relevância, é:

- **sendFile**, que tem como objetivo enviar o ficheiro indicado e que faz uso de funções como a **sendControlPacket**, **sendDataPacket**, **llopen** e **llclose**.
- **sendControlPacket**, que tem como objetivo enviar um pacote de controlo, do tipo *START* no início da transmissão e do tipo *END* no fim da transmissão, e que faz uso de funções como **makeControlPacket** e **llwrite**.
- **sendDataPacket**, que tem como objetivo enviar um pacote de informação, e que faz uso de funções como **makeDataPacket** e **llwrite**.
- **makeControlPacket**, que tem como objetivo criar um pacote de controlo.
- **makeDataPacket**, que tem como objetivo criar um pacote de informação.
- **llwrite**, que tem como objetivo escrever para a Porta Série a informação recebida como argumento, após aplicar uma *frame* e *Byte Stuffing* à informação. Faz uso das funções **framingInformation**, **byteStuffing**, **readControlFrame** e **write**.

As funções de mais baixo nível, associadas à camada da ligação de dados, são usadas quer pelo emissor quer pelo recetor, sendo estas:

- **llopen**, que tem como objetivo abrir a ligação da Porta Série. Faz uso das funções **openSerialPort**, **sendControlFrame** e **readControlFrame**.
- **llclose**, que tem como objetivo terminar a ligação da Porta Série. Faz uso das funções **llcloseTransmitter**, **llcloseReceiver** - conforme seja Emissor ou Recetor - e **close**.
- **sendControlFrame**, que tem como objetivo enviar uma trama de controlo. Faz uso das funções **createControlFrame** e **write**.
- **readControlFrame**, que tem como objetivo receber uma trama de controlo. Faz uso da função **readFromSerialPort**.

6 Protocolo de ligação lógica

A camada de ligação de dados é a camada de mais baixo nível e é a camada responsável pela interação direta com a Porta Série. Algumas das funcionalidades implementada por esta camada são: abertura e fecho da Porta Série; escrita de um tramas de informação e controlo; leitura de um tramas de informação e controlo; criação de tramas de controle; *byte stuffing* e *byte destuffing* de uma trama; *framing* e *deframing* de uma trama.

A nível da API da camada de ligação de dados foram implementadas as quatro funções previstas: **llopen**, **llclose**, **lread** e **llwrite**.

A função **llopen** é responsável por estabelecer a ligação através da Porta Série. Faz recurso à função **openSerialPort** que abre a Porta Série e configura uma nova *struct termios*. De seguida, e segundo o protocolo especificado no guião, envia uma trama de controlo do tipo SET e espera pela receção de uma trama de controlo do tipo UA, no caso do emissor, e faz o processo oposto no caso do recetor - espera pela receção de uma trama de controlo do tipo SET e após a sua receção envia uma trama de controlo do tipo UA.

A função **llclose** é responsável pelo término da ligação estabelecida através da Porta Série. Segundo o protocolo especificado no guião, o término da ligação é realizado através do envio de uma trama de controlo do tipo DISC por parte do emissor, recessão do DISC por parte do recetor, envio de uma trama também do tipo DISC por parte do recetor, receção do DISC enviado pelo recetor, envio de uma trama de controlo do tipo UA por parte do emissor, receção do UA por parte do recetor. Após a verificação deste protocolo de terminação, é reposta a *struct termios* anterior à configurada pela função **llopen** e fecha-se a ligação usando o descritor de ficheiro da Porta Série.

A função **lread** é responsável pela leitura de informação da Porta Série, sendo que irá aplicar *destuffing* e *deframing* à trama recebida. Se ocorrer um erro no *destuffing* ou um erro na *frame* que não seja no *BCC2* a trama é descartada, ficando assim à espera de uma nova trama. Se houver um erro, e for no *BCC2*, a trama é descartada, o recetor continua a espera de uma nova trama, mas envia um trama de controle do tipo REJ. Se a trama for corretamente recebida, esta é retornada e o recetor envia uma trama de controle do tipo RR.

A função **llwrite** é responsável pelo envio de informação através da Porta Série. Esta recebe a mensagem a enviar da camada superior e aplica-lhe *framing* e *stuffing*. De seguida,

tenta escrever a trama, sendo que se não receber uma resposta do tipo RR durante um intervalo de tempo previamente definido - *default* é 3 segundos - este reenvia a trama. O reenvio da trama é feito um número de vezes previamente definido - *default* é 3 tentativas. Se ao fim desse número de tentativas não tiver obtido sucesso, retorna erro.

Consultar [Anexo II](#) para extratos de código.

7 Protocolo de aplicação

A camada de aplicação é a camada de alto nível responsável pelo processo de envio e receção do ficheiro fonte fazendo uso do API disponibilizado pela camada ligação de dados. As funcionalidades implementadas por esta camada são: inicializar a ligação; ler o ficheiro fonte e dividi-lo em pacotes de dados (caso seja emissor); reconstruir o ficheiro fonte a partir de pacotes de dados (caso seja recetor); receber e enviar os pacotes; terminar a ligação.

Para gerir a interação com os pacotes de dados foi desenvolvido o API (Packet.c) composto pelas seguintes funções:

Em relação aos pacotes de dados: A função **makeDataPacket** que cria o pacote de dados a partir da informação original do ficheiro fonte; **sendDataPacket** que envia o pacote de dados utilizando *llwrite*; **receiveDataPacket** que recebe o pacote de dados utilizando *llread* e recolhe a informação do ficheiro fonte.

Em relação aos pacote de controlo: A função **makeControlPacket** que cria o pacote de controlo; **sendControlPacket** que envia o pacote de controlo utilizando *llwrite*; **receiveControlPacket** que recebe o pacote utilizando *llread*.

Foram implementadas as seguintes funções:

- A função **sendFile** (utilizado no caso do emissor) é responsável por inicializar a ligação, ler o ficheiro fonte e dividi-lo em pacotes de dados, enviar os mesmos e terminar a ligação. Faz recurso a 3 funções do API da camada de ligação de dados: *llopen*, *llwrite* e *llclose* e API de pacotes.
- A função **receiveFile** (utilizado no caso do recetor) é responsável por inicializar a ligação, ler os pacotes de dados, reconstruir o ficheiro fonte utilizando os mesmos e terminar a ligação. Faz recurso a 3 funções do API da camada de ligação de dados: *llread*, *llclose* e API de pacotes.

Consultar [Anexo III](#) para extratos de código.

8 Validação

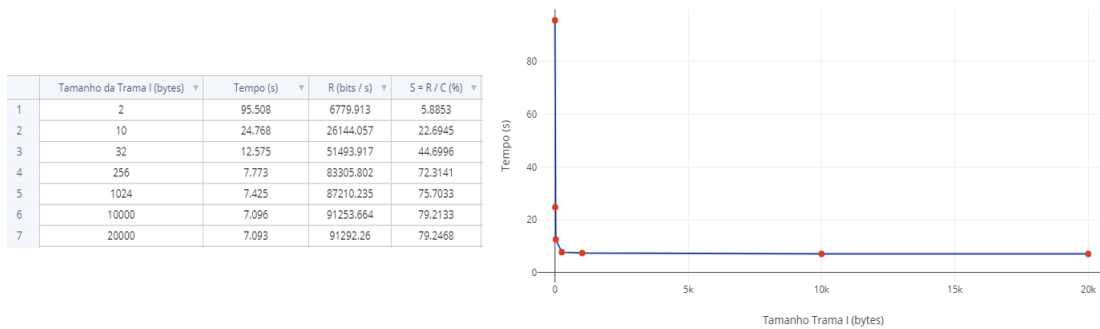
Para validação do programa desenvolvido, e para garantir que funcionava de acordo com o protocolo especificado, foram realizados constantemente testes durante o desenvolvimento do programa e também na sua demonstração. Foram testados ficheiros de diferentes tamanhos e enviados com diferentes baudrates e diferentes tamanhos de pacotes de informação. Foram realizados, simultaneamente com os testes já referidos, testes de interrupção da comunicação

na Porta Série e testes de introdução de erros através do curto-circuito existente nas portas séries. Todos testes terão sido também realizados na presença do professor, aquando do momento de avaliação.

9 Eficiência do protocolo de ligação de dados

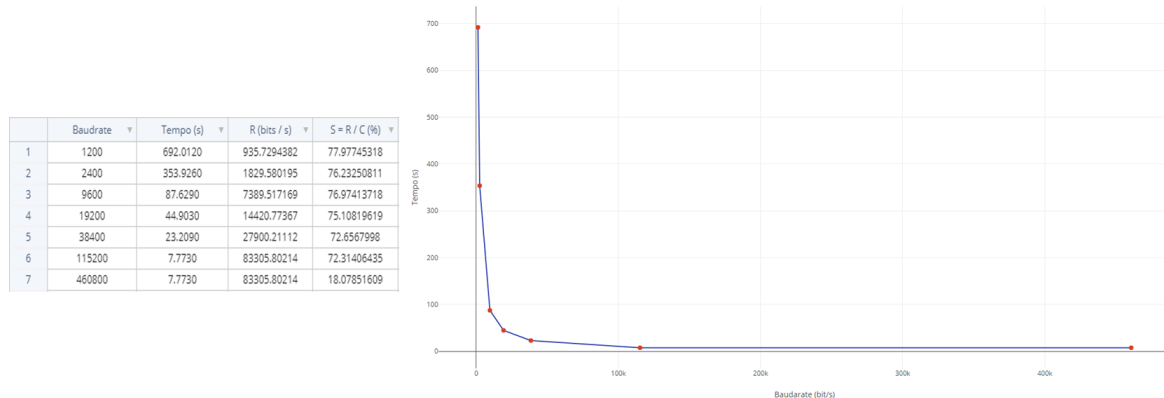
Variável: Tamanho da trama I

Usando um *baudrate* constante de 115200 e uma imagem de tamanho constante 80942 *bytes*, fazendo variar o tamanho da trama I, obteve-se:



Variável: Capacidade da ligação

Usando uma imagem de tamanho constante 80942 *bytes* e um tamanho de trama I constante de valor 256, fazendo variar o *baudrate*, obteve-se:



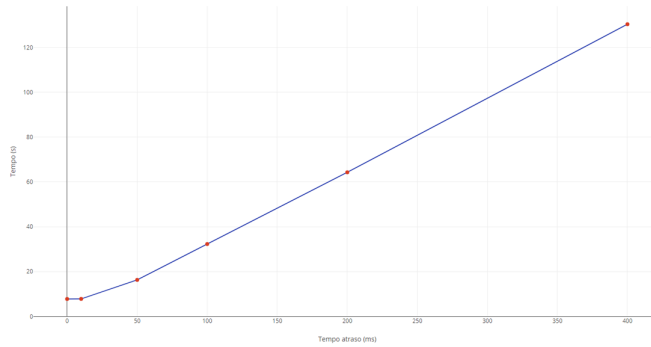
Variável: Tempo de Propagação

Usando uma imagem de tamanho constante 80942 *bytes*, um tamanho de trama I constante de valor 256 e um *baudrate* constante de 460800, fazendo variar o tempo de processamento de cada trama recebida, obteve-se:

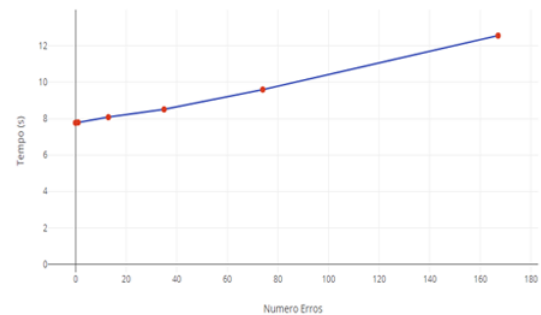
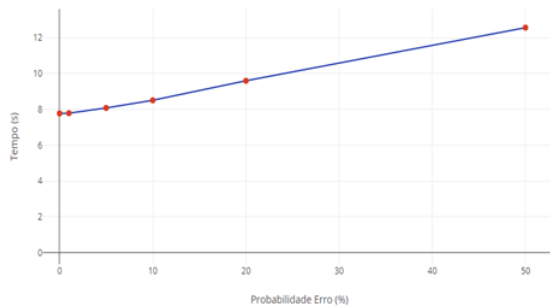
Variável: *Frame Error Ratio*

Usando uma imagem de tamanho constante 80942 *bytes*, um tamanho de trama I constante de valor 256 e um *baudrate* constante de 460800, fazendo variar a probabilidade de ocorrência de erros no cabeçalho das tramas I, obteve-se:

	Tempo atraso (ms)	Tempo (s)
1	0	7.773
2	10	7.802
3	50	16.281
4	100	32.278
5	200	64.280
6	400	129.237



	Probabilidade Erro (%)	Tempo (s)	Num Errors
1	0	7.773	0
2	1	7.788	1
3	5	8.081	13
4	10	8.507	35
5	20	9.597	74
6	50	12.563	167



10 Conclusões

O protocolo implementado pode ser dividido em duas camadas: *LinkLayer* e *Application-Layer*. A *LinkLayer* como sendo a camada de mais baixo nível e fazendo iteração direta com a porta série e a *ApplicationLayer* como sendo a camada de mais alto nível e que faz uso da API da *LinkLayer* para enviar / receber o ficheiro. O protocolo foi implementado com robustez a erros e interrupções, conseguindo em ambos casos retomar a transferência com sucesso. A eficiência do protocolo é de cerca de 77%, aproximando-se bastante dos valores teóricos esperados.

No final, o grupo conclui que a implementação de protocolos representa um desafio, principalmente devido aos mecanismos de deteção de erros e de recuperação. Ainda assim, o grupo considera ter dominado os conceitos teóricos implícitos no trabalho, tais como o uso de *stuffing*, *destuffing*, *framing*, *deframing* e *Stop&Wait* com *sequence numbers*. O grupo considera que o desenvolvimento do protocolo consolidou os conhecimentos teóricos previamente adquiridos.

11 Anexo I

12 Anexo II

Listing 1: llopen

```
int llopen(ConnectionType type) {
    connectionType = type;

    int fd = openSerialPort();

    switch (type) {
    case TRANSMITTER:
        if ((sendControlFrame(fd, SET) > 0) && (readControlFrame(fd, UA) == OK))
            return fd;
        break;
    case RECEIVER:
        if ((readControlFrame(fd, SET) == OK) && (sendControlFrame(fd, UA) > 0))
            return fd;
        break;
    }

    logError("Failed llopen");
    return -1;
}
```

Listing 2: llclose

```
int llclose(int fd) {
    if (TRANSMITTER == connectionType)
        llcloseTransmitter(fd);
    else if (RECEIVER == connectionType)
        llcloseReceiver(fd);
    else
        logError("llclose: no connection type set");

    //Reset terminal to previous configuration
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
        perror("tcsetattr");
        return -1;
    }

    if (close(fd) < 0) {
        perror("failed close(fd)");
        return -1;
    }
    return OK;
}
```

Listing 3: llread

```
int llread(int fd, uchar ** dest) {
    int ret;
```

```

while ( 1 ){
    if ( (ret = readFromSerialPort(fd, dest)) > 0 ) {
        if (byteDestuffing(*dest, &ret) == ERROR) {
            logError("llread: Failed byteDestuffing");
            free(*dest);
            continue;
        }

        ret = deframingInformation(dest, &ret);
        if (ret == ERROR2) {
            sendControlFrame(fd, REJ);
        }
        if (ret != OK) {
            logError("llread: Failed to deframe information");
            free(*dest);
            continue;
        }

        sendControlFrame(fd, RR);
        return ret;
    }
}

return -1;
}

```

Listing 4: llwrite

```

int llwrite(int fd, uchar ** bufferPtr, int length) {
    int res = 0;

    if (framingInformation(bufferPtr, &length) == ERROR) {
        logError("llwrite: Failed to create Information Frame");
        return -1;
    }

    if (byteStuffing(bufferPtr, &length) == ERROR) {
        logError("llwrite: Failed to create Information Frame");
        return -1;
    }

    uint tries = 0;
    setAlarm();
    do {
        alarmWentOff = FALSE;
        if ((res = write(fd, *bufferPtr, length)) < length) {
            logError("llwrite error: * Bad write *");
            continue;
        }

        alarm(ll->timeout);
    }
}

```

```
    } while ( (readControlFrame(fd, RR) != OK) && (++tries < (ll->numRetries)));  
  
    stopAlarm();  
  
    if (tries >= ll->numRetries)  
        return ERROR;  
  
    return res;  
}
```

13 Anexo III

Listing 5: sendFile

```
int sendFile() {
    if (al == NULL)
        return logError("AL not initialized");

    FILE * file = fopen(al->fileName, "r");
    if (file == NULL)
        return logError("Error while opening file");

    // establish connection
    al->fd = llopen(al->type);
    if (al->fd < 0)
        return logError("Failed llopen");

    ControlPacket ctrlPacket;
    ctrlPacket.type = START;
    strcpy(ctrlPacket.fileName, al->fileName);
    ctrlPacket.fileSize = getFileSize(file);
    ctrlPacket.argNr = CTRL_PACKET_ARGS;

    if (sendControlPacket(al->fd, &ctrlPacket) != OK)
        return logError("Error sending control packet");

    DataPacket dataPacket;
    uchar * fileBuffer = (uchar *) malloc(al->maxDataMsgSize * sizeof(char));
    uint res, progress = 0, currentSeqNr = 0;
    int state = OK;
    printProgressBar(0, ctrlPacket.fileSize);

    while ( (res = fread(fileBuffer, sizeof(char), al->maxDataMsgSize, file)) > 0 )
    {
        dataPacket.seqNr = currentSeqNr;
        currentSeqNr = (currentSeqNr + 1) % 256;
        dataPacket.size = res;
        dataPacket.data = fileBuffer;
        if (sendDataPacket(al->fd, &dataPacket) != OK) {
            state = logError("Error sending data packet");
            break;
        }

        progress += res;
        printProgressBar(progress, ctrlPacket.fileSize);
    }
    free(fileBuffer);

    if (fclose(file)) {
        perror("Error while closing file");
        return ERROR;
    }
}
```

```

ctrlPacket.type = END;
if ((state == OK) && sendControlPacket(al->fd, &ctrlPacket) != OK)
    return logError("Error sending control packet");

if (state != OK || llclose(al->fd) != OK)
    return ERROR;

printf("\nFile sent successfully.\n");

return OK;
}

```

Listing 6: receiveFile

```

int receiveFile() {
    if (al == NULL)
        return logError("AL not initialized");

    al->fd = llopen(al->type);
    if (al->fd < 0)
        return logError("Failed llopen");

    ControlPacket ctrlPacket;
    if (receiveControlPacket(al->fd, &ctrlPacket) != OK || ctrlPacket.type !=
        START) {
        return logError("Error receiving control packet");
    }

    if (al->fileName == NULL) {
        al->fileName = malloc(sizeof(char) * MAX_FILE_NAME);
        strncpy(al->fileName, ctrlPacket.fileName, MAX_FILE_NAME);
    }

    FILE * outputFile = fopen(al->fileName, "wb");
    if (outputFile == NULL)
        return logError("Could not create output file");

    printf("Created file %s with expected size %d.\n", al->fileName,
        ctrlPacket.fileSize);

    DataPacket dataPacket;
    uint progress = 0, currentSeqNr = 0;
    int state = OK;
    printProgressBar(0, ctrlPacket.fileSize);

    while (progress < ctrlPacket.fileSize) {
        if ( (state = receiveDataPacket(al->fd, &dataPacket)) != OK) {
            logError("Error receiving data packet");
            break;
        }
    }
}

```



```

    if (dataPacket.seqNr < currentSeqNr) {
        printf("\tReceived duplicate packet: %d. Current: %d\n", dataPacket.seqNr,
            currentSeqNr);
        continue;
    }
    currentSeqNr = (currentSeqNr + 1) % 256;
    progress += (uint) dataPacket.size;

    printProgressBar(progress, ctrlPacket.fileSize);

    if (fwrite(dataPacket.data, sizeof(char), dataPacket.size, outputFile) == 0) {
        return logError("sendFile: fwrite returned 0");
    }

    free(dataPacket.data);
}

if (fclose(outputFile)) {
    perror("fclose failed");
    return ERROR;
}

if ((state == OK) && (receiveControlPacket(al->fd, &ctrlPacket) != OK ||
    ctrlPacket.type != END)) {
    return logError("Error receiving control packet");
}

if (state != OK || llclose(al->fd) != OK)
    return logError("llclose failed");

printf("\nFile received successfully.\n");

return OK;
}

```

Listing 7: makeDataPacket

```

void makeDataPacket(DataPacket * src, Packet * dest) {
    int packetSize = HEADER_SIZE + (src->size);
    uchar * data = (uchar *) malloc(packetSize);

    data[CTRL_FIELD_IDX] = DATA;
    data[SEQ_NUM_IDX] = src->seqNr;
    data[DATA_PACKET_SIZE2_IDX] = (uchar) (src->size / SIZE2_MUL);
    data[DATA_PACKET_SIZE1_IDX] = (uchar) (src->size % SIZE2_MUL);

    memcpy(&data[HEADER_SIZE], src->data, src->size);

    dest->data = data;
    dest->size = packetSize;
}

```

```
}
```

Listing 8: makeControlPacket

```
void makeControlPacket(ControlPacket * src, Packet * dest) {
    int fileNameSize = strlen(src->fileName, MAX_FILE_NAME);
    int packetSize = 1 + 2 * (src->argNr) + fileNameSize + FILE_SIZE_LENGTH;

    uchar * data = (uchar *) malloc(packetSize);

    data[CTRL_FIELD_IDX] = src->type;

    int index = 1;
    data[index++] = FILE_SIZE_ARG;
    data[index++] = sizeof(int);

    uchar fileSize[sizeof(int)];
    convertIntToBytes(fileSize, src->fileSize);
    memcpy(&data[index], fileSize, FILE_SIZE_LENGTH);

    index += FILE_SIZE_LENGTH;

    data[index++] = FILE_NAME_ARG;
    data[index++] = (uchar) fileNameSize;
    memcpy(&data[index], src->fileName, fileNameSize);

    dest->data = data;
    dest->size = packetSize;
}
```
