FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

1º Trabalho Laboratorial
"Ligação de Dados"

António Cunha Seco Fernandes de Almeida - up201505836@fe.up.pt
Diogo Luis Rey Torres - up201506428@fe.up.pt
João Paulo Madureira Damas - up201504088@fe.up.pt

Turma 1 / Bancada 5

Data da Demonstração: 27 de outubro de 2017
Data de Entrega: 3 de novembro de 2017

Índice

Sumário

Este projeto teve como objetivo a implementação de uma aplicação simples que permitisse a transferência de ficheiros entre dois computadores ligados fisicamente por uma porta de série RS-232 através de uma comunicação assíncrona. Assim, foram desenvolvidos diferentes protocolos de acordo com o abordado nas aulas teóricas (Aplicação e Ligação de Dados) de forma a concretizar o objetivo esperado.

A realização deste projeto/relatório permitiu-nos aplicar os conhecimentos teóricos previamente adquiridos numa vertente mais prática, e assim possibilitou uma melhor aprendizagem por parte do grupo nestes conteúdos. Para além disso, concluiu-se que nenhum protocolo de ligação de dados é completamente seguro e que existem inúmeros processos (geralmente mais complexos do que o que foi implementado) para melhorar este tipo de protocolos.

Introdução

Os objetivos propostos para este primeiro trabalho laboratorial consistem em implementar um protocolo de ligação de dados, de acordo com a especificação fornecida e testar esse protocolo com uma aplicação simples de transferência de ficheiros, igualmente especificada.

Este relatório encontra-se dividido em diversas secções retratando diferentes partes do projeto:

●**Arquitetura**, onde são apresentados os blocos funcionais e as interfaces

●**Estrutura do Código**, referindo as APIs implementadas e as principais estruturas de dados

●**Casos de Uso Principais**, fazendo a sua identificação e apresentando as sequências de chamada de funções

●**Protocolo de Ligação Lógica**, onde se identificam as estratégias de implementação do mesmo, com referências oportunas ao código

●**Protocolo de Aplicação**, onde se faz uma descrição análoga à do outro protocolo

●**Validação**, onde são descritos os testes efetuados de modo a verificar o correto funcionamento do programa

●**Eficiência do protocolo de ligação de dados**, onde se caracteriza a estatística da eficiência do protocolo, feito com recurso a medidas sobre o código desenvolvido. É ainda feita uma comparação dos resultados obtidos com a caracterização teórica de um protocolo Stop&Wait.

●**Conclusões**, elaborando uma síntese das secções apresentadas e uma breve reflexão sobre os objectivos alcançados

Arquitetura

O projeto desenvolvido encontra-se estruturado em duas camadas, Aplicação e Ligação de Dados. Esta estrutura em camadas permite que as camadas sejam funcionalmente invisíveis entre si, ou seja, não conhecem os detalhes inerentes a cada uma.

Assim, a camada de aplicação não conhece os detalhes do protocolo de ligação de dados mas é a partir dela que são acedidos os serviços presentes na ligação de dados. Esta camada atua de forma diferente de acordo com o modo do programa (emissor ou recetor). O emissor é responsável pela leitura da informação do ficheiro e envio desses dados para a camada da ligação de dados, enquanto que no recetor é realizada a receção de dados e posterior escrita no ficheiro. Todos os detalhes como a

transparência, o formato das tramas, o seu mecanismo de proteção e eventuais retransmissões são exclusivamente realizadas na camada de ligação de dados.

Por outro lado, na camada de ligação de dados são disponibilizadas as funções genéricas de ligação de dados: abertura e término da ligação, controlo de erros, controlo de fluxo, sincronização e retransmissões/confirmações através de respostas positivas/negativas. Nesta camada não é feito qualquer processamento que incida sobre o cabeçalho dos pacotes a transportar em tramas de informação: não existe qualquer distinção entre pacotes de controlo e de dados, nem é relevante a numeração dos pacotes de dados.

De destacar ainda uma pequena interface no início do programa que permite ao utilizador definir variáveis para a transferência do ficheiro, como o baudrate, tamanho dos pacotes de dados, valor de timeout de ligação e número máximo de retransmissões.

Estrutura do código

Existem duas principais structs onde são guardadas as informações iniciais provenientes do utilizador e outras que importem manter ao longo da execução do programa: *ApplicationLayer* e *LinkLayer*.

A primeira possui um descritor do ficheiro, o modo de ligação usado, o nome do ficheiro a enviar e o respetivo tamanho dos pacotes. Como descrito na arquitetura, apenas trata de empacotar (ou desempacotar) pacotes do ficheiro (dados e/ou controlo), tendo funções para estabelecer a transferência (*startTransfer)* e realizar a transferência em si (*sendData, receiveData, sendControlPackage, receiveControlPackage*, *sendDataPackage*, *receiveDataPackage*).

Por outro lado, a segunda contém o mesmo descritor do ficheiro, o nome da porta de série, o modo de ligação usado, a taxa de transmissão de informação, o tamanho máximo da trama I, o número de sequência (inicialmente zero), valor de timeout e duas structs que contém as configurações anteriores e atuais da porta de série. Possui também, apenas para efeitos estatísticos, o número de respostas positivas/negativas (RR/REJ) recebidas/enviadas. Em termos de funções principais desta camada, destacam-se as funções da API definida no guião (*llopen, llwrite*, *llread*, *llclose*). De salientar, ainda, funções auxiliares às anteriores para enviar uma mensagem de um tipo especificado (*sendSUMessage, sendIMessage*), ler uma mensagem (*readMessage*), obter atributos de uma mensagem (*getMessageSequeceNumber, getIMessageSize*) e para realizar a transparência por byte stuffing e cálculo do BCC2 (*stuff, destuff, getBCC2*).

Casos de usos principais

O programa desenvolvido é constituído maioritariamente por duas sequências de funcionamento, uma presente ao executar no modo *emissor* e outra no modo *recetor*. Na secção de Anexo II encontram-se um diagrama de chamadas de funções que ilustram o fluxo do programa consoante o modo em que é iniciado.

Protocolo de ligação lógica

Na camada de ligação de dados estão implementadas funções críticas para a sincronização de tramas, estabelecimento/terminação da ligação, numeração de tramas, confirmação positiva, controlo de erros e por fim o controlo de fluxo.

Num nível mais elevado, foram implementadas as quatros funções da API prevista. Primeiramente, é chamada a função **llopen**, que para além de abrir a porta série e configurar a struct *termios*, estabelece a ligação segundo o protocolo definido (envio de SET pelo emissor seguido de resposta UA pelo receptor). No final é chamada **llclose**, que trata de encerrar a ligação (pela sequência de comandos DISC (emissor), DISC (receptor), UA (emissor)), repor a struct *termios* e por fim fechar o file descriptor da porta série. Durante a execução do programa são utilizadas pelo protocolo da Aplicação **llwrite** pelo emissor e **llread** pelo receptor. A primeira recebe um pacote da Aplicação, cria uma trama do tipo I (com os mecanismos de proteção e transparência adequados) e envia-a, aguardando uma resposta positiva (RR) que permita o avanço para o próximo pacote, ou negativa (REJ), retransmitindo a trama até um número máximo de tentativas. A segunda trata de fazer a operação inversa, recebendo uma mensagem e processando-a, enviando a resposta adequada e, se for o caso, passando a secção de dados à Aplicação para que esta possa processar o pacote a si destinado.

Todas as funções da API fazem uso de funções auxiliares genéricas para o envio/receção de mensagens e verificação de propriedades das mesmas. Nas figuras 1 e 2, em anexo, pode-se observar a presença de algumas destas funções em llwrite e llread. Na figura 3 é possível ver uma implementação de uma função auxiliar, concluindo que também estas se recorrem entre si. Esta separação permitiu a reutilização destas funções por toda a camada de ligação de dados, poupando bastante trabalho.

Protocolo de aplicação

Nesta camada é feita a transferência do ficheiro no nível mais alto (mais abstrato). Permite o controlo do tipo de pacotes enviados pela porta de série, existindo dois tipos possíveis: controlo e dados. Os pacotes de controlo são enviados para enviar a informação do ficheiro a transferir, sinalizando o início e fim da transferência ao mesmo tempo, enquanto que os pacotes de dados contém os fragmentos do ficheiro a transferir. A transferência começa sempre na função **startTransfer** que se bifurca consoante o modo em que é iniciado o programa.

Tratando-se do emissor, a transferência realiza-se através de **sendData**, caso contrário por **receiveData**. Em ambos os casos as funções servem-se, novamente, de auxiliares para criar (e mandar) pacotes de controlo/dados ou lê-los. Desta forma, foram implementadas ***xxx*ControlPackage** e ***xxx*DataPackage** (xxx = receive ou send consoante o modo). No caso do modo send, nas funções é criado o pacote respetivo e efetuada por fim a chamada à função llwrite da camada supra descrita, com esse mesmo pacote. No caso do modo receive, é chamada inicialmente llread e efetuado posterior processamento do pacote (extração de conteúdo ou informações do ficheiro no caso de ser de dados ou controlo, respetivamente). Assim, nas duas primeiras funções mencionadas no parágrafo, é feita uma chamada inicial e final às funções dos pacotes de controlo, com uma chamada cíclica da respetiva chamada relativamente aos pacotes de dados. Nas figuras 4 e 5 é possível observar essa sequência de chamadas.

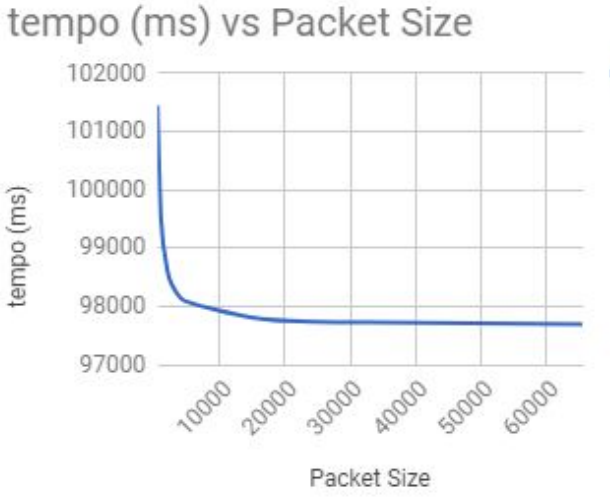
Validação

Para validar o protocolo desenvolvido foram realizados vários testes durante o desenvolvimento e demonstração do mesmo, desde uma transferência normal, transferência com interrupções momentâneas de conexão e envio esporádico de erros através de curto-circuito na porta de série. Também foram testados ficheiros com tamanhos diferentes, enviados a baudrates e com tamanhos de pacotes de informação diferentes. Apesar de não possuirmos imagens, todos os casos mencionados foram testados na demonstração e validados pelo professor.

Eficiência do protocolo de ligação de dados

Variável - Tamanho da trama I

Penguin.gif	BaudRate=57600			
File Size = 11 kB	Tempo(ms)			
Packet Size(kB)	Emissor	Recetor	R(bits/s)	S=R/C
512	101447.02588	101447.99170	44144.78715	0.76640
1024	99534.86939	99535.83276	44992.84203	0.78113
2048	98601.84229	98602.76294	45418.60559	0.78852
4096	98142.33325	98143.27759	45631.24556	0.79221
16384	97786.63037	97787.55835	45797.23715	0.79509
32768	97730.43994	97731.33545	45823.58339	0.79555
65535	97699.19824	97700.13037	45838.21928	0.79580

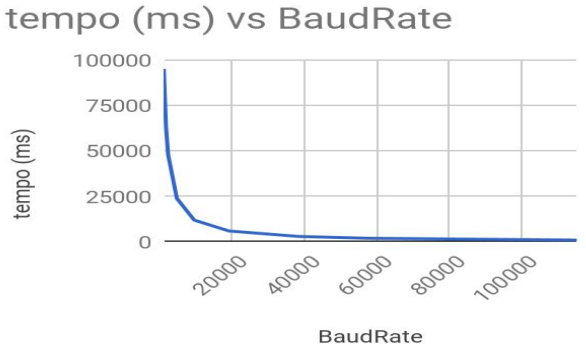


Através deste teste verificou-se que o tamanho do pacote transferido tem um impacto no tempo de transferência total. Contudo, esse impacto foi mais notório em tamanhos de pacotes pequenos - entre pacotes de 512B e 16kB verificou-se uma diferença de, aproximadamente, 3.7s no tempo de transferência, enquanto entre 16kB e 64kB a diferença foi de 0.087 s.

Por outro lado, analisando o valor de S, pode-se afirmar que há medida que o tamanho de pacote aumenta, o valor do mesmo aproxima-se do valor teórico.

Variável - Taxa de Transferência

teste.jpg	Packet Size = 512			
File Size=559.8kB	Tempo(ms)			
BaudRate	Emissor	Recetor	R(bits/s)	S=R/C
1200	95117.53467	95118.38379	922.47152	0.76873
1800	63411.98023	63412.51148	1383.70170	0.76872
2400	47875.04565	47875.50757	1832.75342	0.76365
4800	23941.19263	23941.47534	3664.93705	0.76353
9600	11973.74170	11973.89282	7327.94266	0.76333
19200	5989.01318	5989.55981	14649.49057	0.76299
38400	2996.57178	2996.61304	29281.05795	0.76253
57600	1999.60498	1999.55396	43881.78663	0.76184
115200	1002.33203	1002.33643	87539.47051	0.75989



Através deste teste, pode-se concluir que a taxa de transferência tem um impacto significativo no tempo de transferência (o aumento de baudrate implica tempo de transferência menor). Por outro lado,o valor obtido da velocidade de transferência real é em média cerca de 76% do valor teórico, sendo o mais próximo obtido com um valor de baudrate de 1200 com 76.87%.

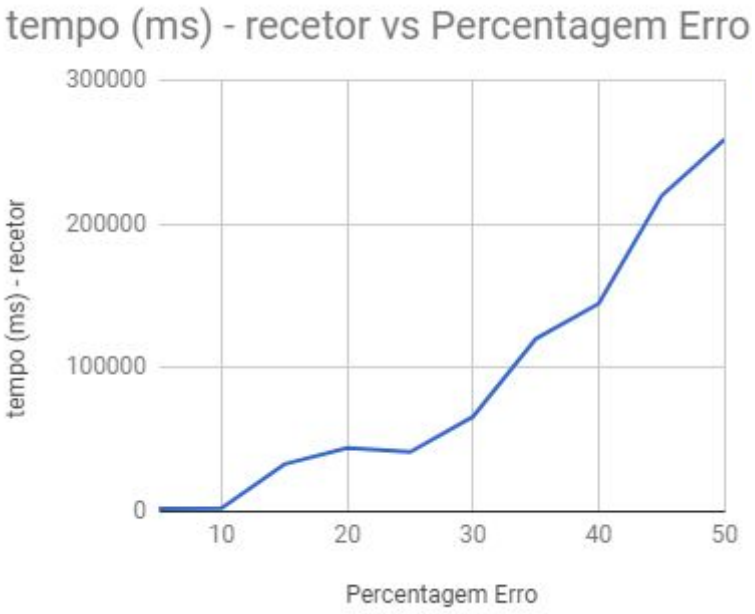
Variável - T_prop

Baudrate = 57600	Packet Size = 2048
Image Size = 262.1kB	Tempo(ms)
Tempo de Atraso(ms)	Recetor
0	1947.6
50	2347.73
100	2747.74
150	3147.83
200	3547.65
250	3947.68
500	5947.58
1000	9947.73

Através deste teste, pode-se afirmar que o aumento de tempo de propagação tem um impacto linear no tempo de transferência total, ou seja, de acordo com o esperado.

Variável - FER

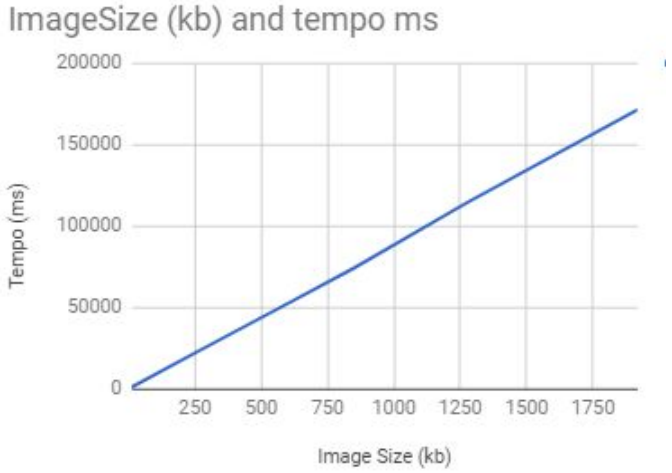
Baudrate=57600	Packet Size = 512
Image Size = 262.1kB	Tempo(ms)
Probabilidade de Erro(%)	Recetor
5	1999.66
10	2277.94
15	32828.36
20	44266.64
25	41362.82
30	66055.03
35	120291.44
40	144828.29
45	219847.04
50	259053.7



Através deste teste conclui-se que a quantidade de erros - nos cabeçalhos ou nos campos de dados de tramas I - têm um impacto significativo no tempo de transferência, verificando-se um aumento de 12,000 % do primeiro para o último caso. Por outro lado, o teste permitiu confirmar a robustez dos protocolos implementados, e em particular, da nossa implementação dos mesmos.

Variável - Tamanho do ficheiro

Baudrate = 115200	Packet Size = 65535			
	Tempo(ms)			
ImageSize (kB)	Emissor	Recetor	R(bits/s)	S=R/C
11	965.46851	965.51880	91142.71011	0.79117
262.1	22969.97119	22969.59766	91285.88282	0.79241
559.8	48863.54028	48863.97070	91650.34965	0.79558
848.3	74087.88232	74089.14722	91597.76101	0.79512
1271	113863.19873	113864.29541	89299.28353	0.77517
1918	171476.52881	171473.66333	89483.12937	0.77676



Foi realizado um teste adicional não previsto no guião do trabalho, dado o propósito da aplicação desenvolvida e as conclusões práticas que o mesmo implica. Assim, foi testado o envio de imagens com diversos tamanhos, mantendo o baudrate e o packet size constante.

Com este teste adicional verificou-se que a eficiência de transmissão de dados decresce à medida que o tamanho do ficheiro aumenta, sendo o pico de eficiência apresentado por uma imagem com 559.8kB (79.55%). Contudo, a perda de eficiência foi diminuta ao ponto de se poder afirmar com confiança que o tempo de transmissão aumentou linearmente com o tamanho da imagem.

Conclusão

Este projeto permitiu a utilização de uma variante de transmissão de Stop and Wait para implementar um protocolo de ligação de dados. Foi desenvolvida uma aplicação simples e funcional para transferir ficheiros entre dois computadores via porta de série devidamente estruturas e independente entre as suas camadas. Os cálculos estatísticos revelaram a boa eficiência da solução. Foi possível realizar uma aprendizagem mais aprofundada relativamente ao uso da porta de série, máquinas de estados aplicadas a protocolos e protocolos usados em redes, portanto o grupo concluiu que foi conseguido um bom aproveitamento tendo em conta o objetivo proposto.

Em suma, o desenvolvimento do projeto foi exigente devido às especificidades pretendidas - principalmente a forma de interação e sincronização da camada de aplicação e de dados em ambos os computadores - mas a conclusão final é de que os conhecimentos teóricos aprendidos ao longo das aulas ficaram bem consolidados.

Anexo I - Código Fonte

ApplicationLayer.h

```
#pragma once

#define I_PACKAGE_DATA 1
#define I_PACKAGE_START 2
#define I_PACKAGE_END 3

#define PARAM_FILE_SIZE 0
#define PARAM_FILE_NAME 1

#define SEND 0
#define RECEIVE 1

typedef struct {
    int fd;
    int mode;
    char* fileName;
    int maxSize;
} ApplicationLayer;

int initApplicationLayer(const char* port, int mode, int baudrate, int messageDataMaxSize,
int numRetries, int timeout, char* file);
int getFileSize(char* filePath);
int startTransfer();
int sendData();
int receiveData();
int sendControlPackage(int C, char* fileSize, char* fileName);
int receiveControlPackage(int* ctrl, int* fileLength, char** fileName);
int sendDataPackage(int N, const char* buf, unsigned int length);
int receiveDataPackage(int* N, char** buf, unsigned int* length);
```

ApplicationLayer.c

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include "LinkLayer.h"
#include "Utilities.h"
#include "ApplicationLayer.h"

#define S_TO_MS 1e3
#define NS_TO_MS 1e-6

ApplicationLayer* al;
```

```
int initApplicationLayer(const char* port, int mode, int baudrate, int messageDataMaxSize,
int numRetries, int timeout, char* file) {
    al = (ApplicationLayer*) malloc(sizeof(ApplicationLayer));

    initLinkLayer(port, mode, baudrate, messageDataMaxSize, numRetries, timeout);

    //Setup connection
    al->fd = llopen(port, mode);
    if (al->fd <= 0)
        return -1;

    al->mode = mode;
    al->fileName = file;
    al->maxSize = messageDataMaxSize;

    struct timespec starting_time_temp;
    if(clock_gettime(CLOCK_REALTIME, &starting_time_temp) == -1)
        printf("ERROR: %s\n", strerror(errno));
    double starting_time = (double)(starting_time_temp.tv_sec*S_TO_MS +
starting_time_temp.tv_nsec*NS_TO_MS);

    startTransfer();

    struct timespec final_time_temp;
    if(clock_gettime(CLOCK_REALTIME, &final_time_temp) == -1)
        printf("ERROR : %s\n", strerror(errno));
    double final_time = (double)(final_time_temp.tv_sec*S_TO_MS +
final_time_temp.tv_nsec*NS_TO_MS);

    printf("Transfer time: %.2f ms\n", final_time - starting_time);


    if (!llclose(al->mode)) {
        printf("ERROR: Serial port was not closed.\n");
        return -1;
    }


    return 1;
}

int getFileSize(char* filePath) {
    int size;
    struct stat temp;
    if(stat(filePath, &temp) != 0){
        printf("ERROR: Unable to get file size\n");
        return -1;
    }

    size = temp.st_size;
    return size;
}
```

```
int startTransfer() {
    switch (al->mode) {
    case RECEIVE:
        printf("Starting transfer as RECEIVER.\n");
        if(receiveData() <= 0)
               exit(-1);
        break;
    case SEND:
        printf("Starting transfer as TRANSMITTER.\n");
        if(sendData() <= 0)
               exit(-1);
        break;
    default:
        break;
    }

    return 1;
}

int sendData() {
    FILE* file = fopen(al->fileName, "rb");
    if (file == NULL) {
        printf("ERROR: Could not open file to be sent.\n");
        return -1;
    }

    // Get file size (using stat utility)
    int fileSize = getFileSize(al->fileName);
    if(fileSize < 0)
        return -1;
    char fileSizeBuf[sizeof(unsigned int) * 3 + 2];
    snprintf(fileSizeBuf, sizeof(fileSizeBuf), "%d", fileSize);

    if (!sendControlPackage(I_PACKAGE_START, fileSizeBuf, al->fileName))
        return -1;

    char* fileBuf = malloc(al->maxSize);

    printf("Starting file data transfer\n");

    unsigned int nBytesRead = 0, i = 0;
    while ((nBytesRead = fread(fileBuf, sizeof(char), al->maxSize, file)) > 0) {
        if (!sendDataPackage((i++) % 255, fileBuf, nBytesRead)) {
               free(fileBuf);
               return -1;
        }

        memset(fileBuf, 0, al->maxSize);
    }
    printf("\n\n");

    printf("Finished file data transfer\n");

    free(fileBuf);
```

```
    if (fclose(file) != 0) {
        printf("ERROR: Unable to close file.\n");
        return -1;
    }

    if (!sendControlPackage(I_PACKAGE_END, fileSizeBuf, al->fileName))
        return -1;

    return 1;
}

int receiveData() {
    int controlStart;
    int controlEnd;
    int fileSize;
    char* fileName = (char*)malloc(MAX_SIZE);
    bzero(fileName, MAX_SIZE); //Otherwise, random characters will be appended to fileName

    printf("Waiting for START control package.\n");
    if (!receiveControlPackage(&controlStart, &fileSize, &fileName))
        return -1;

    if (controlStart != I_PACKAGE_START) {
        printf("ERROR: Expected START control package, got control package number %d
instead\n", controlStart);
        return -1;
    }

    FILE* outputFile = fopen(fileName, "wb");
    if (outputFile == NULL) {
        printf("ERROR: Could not create output file.\n");
        return -1;
    }

    printf("\n");
    printf("Created output file: %s\n", fileName);
    printf("Expected file size: %d (bytes)\n", fileSize);
    printf("\n");

    printf("Starting file data reception\n");

    int N = -1;
    unsigned int totalSizeRead = 0;
    while (totalSizeRead != fileSize) {
        int lastN = N;
        char* fileBuf = NULL; //Is initialized inside the function with the appropriate
read length
        unsigned int length = 0;

        if (!receiveDataPackage(&N, &fileBuf, &length)) {
                printf("ERROR: Could not receive data package.\n");
                free(fileBuf);
                return -1;
```

```
        }

        if (N != 0 && lastN + 1 != N) {
                printf("ERROR: Expected sequence %d, received %d instead\n", lastN+1, N);
                free(fileBuf);
                return -1;
        }

        fwrite(fileBuf, sizeof(char), length, outputFile);
        free(fileBuf);

        totalSizeRead += length;
        printf("Read data package with %d bytes, %d/%d bytes read so far\n", length,
totalSizeRead, fileSize);
    }

    printf("\nFinished file data reception\n");

    if (fclose(outputFile) != 0) {
        printf("ERROR: Closing output file.\n");
        return -1;
    }

    if (!receiveControlPackage(&controlEnd, 0, NULL))
        return -1;

    if (controlEnd != I_PACKAGE_END) {
        printf("ERROR: Expected START control package, got control package number %d
instead\n", controlEnd);
        return -1;
    }

    return 1;
}

int sendControlPackage(int C, char* fileSize, char* fileName) {
    int packageSize = 5 + strlen(fileSize) + strlen(fileName); //Guaranteed at least always
5 bytes: C, T1, L1, T2, L2
    unsigned int index = 0;

    // Creating the package
    unsigned char package[packageSize];
    package[index++] = C;
    package[index++] = PARAM_FILE_SIZE;
    package[index++] = strlen(fileSize);
    unsigned int i;
    for (i = 0; i < strlen(fileSize); i++)
        package[index++] = fileSize[i];
    package[index++] = PARAM_FILE_NAME;
    package[index++] = strlen(fileName);
    for (i = 0; i < strlen(fileName); i++)
        package[index++] = fileName[i];

    if (C == I_PACKAGE_START) {
```

```
        printf("\nFile: %s\n", fileName);
        printf("Size: %s (bytes)\n\n", fileSize);
    }

    // Actually sending
    if (!llwrite(package, packageSize)) {
        printf("ERROR: Could not write control package.\n");
        free(package);
        return -1;
    }

    return 1;
}

int receiveControlPackage(int* controlPackageType, int* fileLength, char** fileName) {
    unsigned char* package;
    unsigned int packageSize = llread(&package);
    if (packageSize < 0) {
        printf("ERROR: Could not read from link layer while receiving control
package.\n");
        return -1;
    }

    *controlPackageType = package[0];

    // END package just symbolizes transfer process is over, no need to process anything
further
    if (*controlPackageType == I_PACKAGE_END)
        return 1;

    unsigned int i;
    unsigned int index = 1;
    unsigned int numBytes = 0;
    for (i = 0; i < 2; ++i) {
        int paramType = package[index++];

        if(paramType == PARAM_FILE_SIZE){
                numBytes = (unsigned int) package[index++];

                char* length = (char*)malloc(numBytes);
                memcpy(length, &package[index], numBytes);

                *fileLength = atoi(length);
                free(length);
        }
        else if(paramType == PARAM_FILE_NAME){
                numBytes = (unsigned char) package[index++];
                memcpy(*fileName, &package[index], numBytes);
        }
        else
                printf("Unrecognised file parameter, skipping\n");

        index += numBytes;
    }
```

```
    return 1;
}

int sendDataPackage(int N, const char* buffer, unsigned int length) {
    unsigned char C = I_PACKAGE_DATA;
    unsigned char L2 = length / 256;
    unsigned char L1 = length % 256;

    unsigned int packageSize = 4 + length; //4 bytes are from C, N, L2 and L1

    unsigned char* package = (unsigned char*) malloc(packageSize);
    package[0] = C;
    package[1] = N;
    package[2] = L2;
    package[3] = L1;
    memcpy(&package[4], buffer, length);

    // Actually send package
    if (!llwrite(package, packageSize)) {
        printf("ERROR: Could not write data package.\n");
        free(package);
        return -1;
    }

    free(package);

    return 1;
}

int receiveDataPackage(int* N, char** buf, unsigned int* length) {
    unsigned char* package;

    unsigned int packageSize = llread(&package);
    if (packageSize < 0) {
        printf("ERROR: Could not read data package.\n");
        return -1;
    }

    int C = package[0];
    *N = (unsigned char) package[1];
    int L2 = package[2];
    int L1 = package[3];

    if (C != I_PACKAGE_DATA) {
        printf("ERROR: Received package is not a data package.\n");
        return -1;
    }

    *length = (L2 << 8) + L1;
    *buf = (char*)malloc(*length);
    memcpy(*buf, &package[4], *length);
    free(package);
    return 1;
```

```
}
```

LinkLayer.h

```
#pragma once

#include <termios.h>
#include "Utilities.h"

void atende(int signal);
void setAlarm();
void stopAlarm();

#define RANDOM_ER_PROB 0

#define SEND 0
#define RECEIVE 1

#define C_SET 0x03
#define C_UA 0x07
#define C_RR 0x05
#define C_REJ 0x01
#define C_DISC 0x0B

#define SET 0
#define UA 1
#define RR 2
#define REJ 3
#define DISC 4

#define SU_MESSAGE 0
#define I_MESSAGE 1
#define INV_MESSAGE -1

#define SU_MESSAGE_SIZE 5*sizeof(char)
#define I_MESSAGE_BASE_SIZE 6*sizeof(char)

typedef enum {
    START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP
} machineState;

typedef struct {
    int fd;
    char port[20];
    int mode;
    int baudRate;
    int messageDataSize;
    unsigned int ns;
    unsigned int timeout;
    unsigned int numTries;
    struct termios oldtio, newtio;
    int IMessagesSent;
    int IMessagesReceived;
```

```
    int timeouts;
    int RRSent;
    int RRReceived;
    int REJSent;
    int REJReceived;
} LinkLayer;

void initLinkLayer(const char* port, int mode, int baudrate, int messageDataSize, int
timeout, int numRetries);

int llopen(const char* port, int mode);
int llwrite(const unsigned char* buf, unsigned int bufSize);
int llread(unsigned char** message);
int llclose(int mode);

void sendSUMessage(int command);
void sendIMessage(const unsigned char* buf, unsigned int bufSize);
unsigned char* readMessage(int* type, int* hasBCC2Error);
int messageIs(unsigned char* msg, int msgType, int command);
int getMessageSequenceNumber(unsigned char* msg, int type);
int getIMessageSize(unsigned char* msg);

unsigned int stuff(unsigned char** buf, unsigned int bufSize);
unsigned int destuff(unsigned char** buf, unsigned int bufSize);
unsigned char getBCC2(const unsigned char* buf, unsigned int size);
```

LinkLayer.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include "ApplicationLayer.h"
#include "LinkLayer.h"

LinkLayer* ll;

const int FLAG = 0x7E;
const int A = 0x03;
const int ESCAPE = 0x7D;
int alarmWentOff = 0;

void atende(int signal) {
    alarmWentOff = 1;
    ll->timeouts++;
    printf("Timeout. Retrying\n");

    alarm(ll->timeout);
}
```

```c
void setAlarm() {
    struct sigaction action;
    action.sa_handler = atende;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarmWentOff = 0;

    alarm(ll->timeout);
}

void stopAlarm() {
    struct sigaction action;
    action.sa_handler = NULL;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGALRM, &action, NULL);

    alarm(0);
}

void initLinkLayer(const char* port, int mode, int baudrate, int messageDataSize, int
numRetries, int timeout) {
    ll = (LinkLayer*) malloc(sizeof(LinkLayer));

    strcpy(ll->port, port);
    ll->mode = mode;
    ll->baudRate = baudrate;
    ll->messageDataSize = messageDataSize;
    ll->ns = 0;
    ll->timeout = timeout;
    ll->numTries = 1 + numRetries;

    ll->IMessagesSent = 0;
    ll->IMessagesReceived = 0;
    ll->timeouts = 0;
    ll->RRSent = 0;
    ll->RRReceived = 0;
    ll->REJSent = 0;
    ll->REJReceived = 0;
}

int llopen(const char* port, int mode) {
    printf("Estabilishing connection\n");

    // Open serial port device for reading and writing and not as controlling
    // tty because we don't want to get killed if linenoise sends CTRL-C.
    ll->fd = open(port, O_RDWR | O_NOCTTY);
    if(ll->fd < 0){
        printf("ERROR: Couldn't open serial port\n");
```

```
        return -1;
    }

    //Set new termios structure, saving the old one to undo the change in the end
    tcgetattr(ll->fd, &ll->oldtio);

    bzero(&ll->newtio, sizeof(ll->newtio));
    ll->newtio.c_cflag = ll->baudRate | CS8 | CLOCAL | CREAD;
    ll->newtio.c_iflag = IGNPAR;
    ll->newtio.c_oflag = 0;
    ll->newtio.c_lflag = 0;
    ll->newtio.c_cc[VTIME] = 10*ll->timeout;
    ll->newtio.c_cc[VMIN] = 0;

    tcflush(ll->fd, TCIOFLUSH);

    if (tcsetattr(ll->fd, TCSANOW, &ll->newtio) != 0)
        return -1;

    printf("New termios structure set.\n");

    int try = 0, connected = 0;
    switch (mode) {
    case SEND: {
        while (!connected) {
                if (try == 0 || alarmWentOff) {
                        alarmWentOff = 0;
                        if (try >= ll->numTries) {
                                stopAlarm();
                                printf("ERROR: Maximum number of retries reached.
Aborting\n");
                                return -1;
                        }

                        sendSUMessage(SET);
                        if (++try == 1)
                                setAlarm();
                }

                int type;
                unsigned char* receivedMessage = readMessage(&type, NULL);
                if (messageIs(receivedMessage, type, UA)) {
                        connected = 1;
                        printf("Estabilished connection\n");
                }
        }
        stopAlarm();
        break;
    }
    case RECEIVE: {
        while (!connected) {
                int type;
                unsigned char* receivedMessage = readMessage(&type, NULL);
                if (messageIs(receivedMessage, type, SET)) {
```

```
                        sendSUMessage(UA);
                        connected = 1;
                        printf("Estabilished connection\n");
                }
        }
        break;
    }
    default:
        break;
    }

    return ll->fd;
}

int llwrite(const unsigned char* buf, unsigned int bufSize) {
    int try = 0, transferring = 1;
    while (transferring) {
        if (try == 0 || alarmWentOff) {
                alarmWentOff = 0;
                if (try >= ll->numTries) {
                        stopAlarm();
                        printf("ERROR: Maximum number of retries reached. Aborting\n");
                        return -1;
                }

                sendIMessage(buf, bufSize);
                if (++try == 1)
                        setAlarm();
        }

        int type;
        unsigned char* receivedMessage = readMessage(&type, NULL);

        if (messageIs(receivedMessage, type, RR)) {
                int nr = getMessageSequenceNumber(receivedMessage, type);
                if (ll->ns != nr)
                        ll->ns = nr;

                stopAlarm();
                transferring = 0;
        }
        else if (messageIs(receivedMessage, type, REJ)) {
                stopAlarm();
                try = 0;
        }
    }

    stopAlarm();
    return 1;
}

int llread(unsigned char** message) {
    unsigned char* msg = NULL;
    int msgSize = 0;
```

```
    int done = 0;
    while (!done) {
        int type, hasBCC2Error, msgNs;
        msg = readMessage(&type, &hasBCC2Error);
        switch (type) {
        case INV_MESSAGE:
                printf("INVALID message received.\n");
                if (hasBCC2Error)
                        sendSUMessage(REJ);
                break;
        case SU_MESSAGE:
                if (messageIs(msg, type, DISC))
                        done = 1;
                break;
        case I_MESSAGE:
                msgNs = getMessageSequenceNumber(msg, type);
                msgSize = getIMessageSize(msg);
                if (ll->ns == msgNs) {
                        *message = malloc(msgSize);
                        memcpy(*message, &msg[4], msgSize);
                        free(msg);

                        ll->ns = !msgNs;
                        sendSUMessage(RR);
                        done = 1;
                } else {
                        printf("Unexpected sequence number: ignoring content but confirming
with RR.\n");
                        sendSUMessage(RR);
                }
                break;
        }
    }

    stopAlarm();
    return msgSize;
}

int llclose(int mode) {
    printf("Ending connection\n");
    int try = 0, disconnected = 0;
    int type;
    unsigned char* receivedMessage;

    switch (mode) {
    case SEND: {
        while (!disconnected) {
                if (try == 0 || alarmWentOff) {
                        alarmWentOff = 0;
                        if (try >= ll->numTries) {
                                stopAlarm();
                                printf("ERROR: Maximum number of retries reached.
Aborting\n");
                                return -1;
```

```
                        }

                        sendSUMessage(DISC);
                        if (++try == 1)
                                setAlarm();
                }

                receivedMessage = readMessage(&type, NULL);
                if (messageIs(receivedMessage, type, DISC))
                        disconnected = 1;
        }

        stopAlarm();
        sendSUMessage(UA);
        sleep(1);
        break;
    }
    case RECEIVE: {
        while (!disconnected) {
                receivedMessage = readMessage(&type, NULL);
                if (messageIs(receivedMessage, type, DISC))
                        disconnected = 1;

                }

        int uaReceived = 0;
        while (!uaReceived) {
                if (try == 0 || alarmWentOff) {
                        alarmWentOff = 0;

                        if (try >= ll->numTries) {
                                stopAlarm();
                                printf("ERROR: Maximum number of retries reached.
Aborting.\n");
                                return -1;
                        }

                        sendSUMessage(DISC);
                        if (++try == 1)
                                setAlarm();
                }

                receivedMessage = readMessage(&type, NULL);
                if (messageIs(receivedMessage, type, UA))
                        uaReceived = 1;
        }

        stopAlarm();
        printf("Ended connection\n");
    }
    default:
        break;
    }
```

```
    if (tcsetattr(ll->fd, TCSANOW, &ll->oldtio) == -1) {
        perror("tcsetattr");
        return -1;
    }

    close(ll->fd);

    switch (mode) {
        case SEND:
        printf("Connection ended\n");
        printf("SENT I Messages : %d\n", ll->IMessagesSent);
        printf("Timeouts : %d\n", ll->timeouts);
        printf("RECEIVED RR Commands : %d\n", ll->RRReceived);
        printf("RECEIVED REJ Commands : %d\n", ll->REJReceived);
        break;
        case RECEIVE:
        printf("RECEIVED I Messages : %d\n", ll->IMessagesReceived);
        printf("SENT RR Commands : %d\n", ll->RRSent);
        printf("SENT REJ Commands : %d\n", ll->REJSent);
        break;
    }

    return 1;
}

void sendSUMessage(int command) {
    int ctrlField;
    switch (command) {
    case SET:
        ctrlField = C_SET;
        break;
    case UA:
        ctrlField= C_UA;
        break;
    case RR:
        ctrlField= C_RR;
        break;
    case REJ:
        ctrlField= C_REJ;
        break;
    case DISC:
        ctrlField= C_DISC;
        break;
    default:
        return; //Unrecognised command
    }

    //Creating the package
    unsigned char* messageBuf = (unsigned char*)malloc(SU_MESSAGE_SIZE);

    messageBuf[0] = FLAG;
    messageBuf[1] = A;
    messageBuf[2] = ctrlField;
    if (ctrlField == C_REJ || ctrlField == C_RR)
```

```
        messageBuf[2] |= (ll->ns << 7);
    messageBuf[3] = messageBuf[1] ^ messageBuf[2];
    messageBuf[4] = FLAG;

    //Actually sending
    if (write(ll->fd, messageBuf, SU_MESSAGE_SIZE) != SU_MESSAGE_SIZE) {
        printf("ERROR: Could not write command with C 0x%x.\n", ctrlField);
        return;
    }

    free(messageBuf);

    if (command == REJ)
        ll->REJSent++;
    else if (command == RR)
        ll->RRSent++;
}

void sendIMessage(const unsigned char* message, unsigned int messageSize) {
    //Creating the package
    unsigned char* msg = malloc(I_MESSAGE_BASE_SIZE + messageSize);

    unsigned char C = ll->ns << 6;
    unsigned char BCC1 = A ^ C;
    unsigned char BCC2 = getBCC2(message, messageSize);

    msg[0] = FLAG;
    msg[1] = A;
    msg[2] = C;
    msg[3] = BCC1;
    memcpy(&msg[4], message, messageSize);
    msg[4 + messageSize] = BCC2;
    msg[5 + messageSize] = FLAG;

    messageSize += I_MESSAGE_BASE_SIZE;
    messageSize = stuff(&msg, messageSize);

    //Actually sending
    unsigned int numWrittenBytes = write(ll->fd, msg, messageSize);
    if (numWrittenBytes != messageSize){
        printf("Could not write data packet correctly\n");
        return;
    }

    free(msg);
    ll->IMessagesSent++;
}

unsigned char* readMessage(int* type, int* hasBCC2Error) {
    *type = INV_MESSAGE;
    machineState state = START;
    unsigned int size = 0;
    unsigned char* message = malloc(ll->messageDataSize);
    volatile int done = 0;
```

```
while (!done) {
    unsigned char c;
    if (state != STOP) {
            int numReadBytes = read(ll->fd, &c, 1);
            if (numReadBytes == 0) {
                    printf("ERROR: Nothing received.\n");
                    free(message);
                    *type = INV_MESSAGE;
                    if(hasBCC2Error != NULL)
                            hasBCC2Error = 0;

                    return NULL;
            }
    }

    switch (state) {
    case START:
            if (c == FLAG) {
                    message[size++] = c;
                    state = FLAG_RCV;
            }
            break;
    case FLAG_RCV:
            if (c == A) {
                    message[size++] = c;
                    state = A_RCV;
            } else if (c != FLAG) {
                    size = 0;
                    state = START;
            }
            break;
    case A_RCV:
            if (c != FLAG) {
                    message[size++] = c;
                    state = C_RCV;
            } else if (c == FLAG) {
                    size = 1;
                    state = FLAG_RCV;
            } else {
                    size = 0;
                    state = START;
            }
            break;
    case C_RCV:
            if (c == (message[1] ^ message[2])) {
                    message[size++] = c;
                    state = BCC_OK;
            } else if (c == FLAG) {
                    size = 1;
                    state = FLAG_RCV;
            } else {
                    size = 0;
                    state = START;
```

```
                }
                break;
        case BCC_OK:
                if (c == FLAG) {
                        if(*type == INV_MESSAGE)
                                *type = SU_MESSAGE;
                        message[size++] = c;
                        state = STOP;
                } else if (c != FLAG) {
                        if(*type == INV_MESSAGE)
                                *type = I_MESSAGE;

                        // If extra memory is needed
                        if (size % ll->messageDataSize == 0)
                                message = (unsigned char*) realloc(message, size +
ll->messageDataSize);

                        message[size++] = c;
                }
                break;
        case STOP:
                message[size] = 0;
                done = 1;
                break;
        default:
                break;
        }
    }

    size = destuff(&message, size);
    unsigned char A = message[1];
    unsigned char C = message[2];
    unsigned char BCC1 = message[3];

    // Force errors on control packet (based on random value)
    int random = rand() % 100;
    if(random < RANDOM_ER_PROB) {
        printf("Forced ERROR on BCC1.\n");
        BCC1 = (A ^ C) + 1111;
    }

    if (BCC1 != (A ^ C)) {
        printf("ERROR: invalid BCC1.\n");
        free(message);
        *type = INV_MESSAGE;
        if(hasBCC2Error != NULL)
                *hasBCC2Error = 0;

        return NULL;
    }

    if (*type == SU_MESSAGE) {
        int controlField = message[2];
```

```
        if ((controlField & 0x0F) == C_REJ) {
                printf("Received REJ command.\n");
                ll->REJReceived++;
        }
        else if ((controlField & 0x0F) == C_RR) {
                printf("Received RR command.\n");
                ll->RRReceived++;
        }
    }
    else if (*type == I_MESSAGE) {
        unsigned char calcBCC2 = getBCC2(&message[4], size - I_MESSAGE_BASE_SIZE);
        unsigned char BCC2 = message[4 + size - I_MESSAGE_BASE_SIZE];

        // Force errors on data packet (based on random value)
        int random = rand() % 100;
        if(random < RANDOM_ER_PROB) {
                printf("Forced ERROR on BCC2.\n");
                calcBCC2 = BCC2 + 1111;
        }

        if (calcBCC2 != BCC2) {
                printf("ERROR: Invalid BCC2: 0x%02x != 0x%02x.\n", calcBCC2, BCC2);
                free(message);
                *type = INV_MESSAGE;
                if(hasBCC2Error != NULL)
                        *hasBCC2Error = 1;

                return NULL;
        }

        ll->IMessagesReceived++;
    }

    return message;
}

int messageIs(unsigned char* msg, int msgType, int command) {
    if(msgType != SU_MESSAGE) return 0;
    switch(command){
    case SET:
        return msg[2] == C_SET;
    case UA:
        return msg[2] == C_UA;
    case RR:
        return (msg[2] & 0x0F) == C_RR;
    case REJ:
        return (msg[2] & 0x0F) == C_REJ;
    case DISC:
        return msg[2] == C_DISC;
    default:
        return 0;
    }
}
```

```
unsigned int stuff(unsigned char** buf, unsigned int bufSize) {
    unsigned int newBufSize = bufSize;

    //Calculating new size by checking how many octates need to be escaped
    int i;
    for (i = 1; i < bufSize - 1; ++i){
        if ((*buf)[i] == FLAG || (*buf)[i] == ESCAPE)
                newBufSize++;
    }

    unsigned char* newBuf = (unsigned char*)malloc(newBufSize);
    int escapedBytesOffset = 0;
    //Flag on each end
    newBuf[0] = FLAG;
    newBuf[newBufSize - 1] = FLAG;

    //Copy the rest with respective byte stuffing when necessary
    for(i = 1; i < bufSize - 1; ++i){
        if((*buf)[i] == FLAG || (*buf)[i] == ESCAPE){
                newBuf[i + escapedBytesOffset] = ESCAPE;
                newBuf[i + escapedBytesOffset + 1] = (*buf)[i] ^ 0x20;
                ++escapedBytesOffset;
        }
        else
                newBuf[i + escapedBytesOffset] = (*buf)[i];
    }

    free(*buf);
    *buf = newBuf;
    return newBufSize;
}

unsigned int destuff(unsigned char** buf, unsigned int bufSize) {
    unsigned int newBufSize = bufSize;
    int i;
    for(i = 1; i < bufSize - 1; ++i) {
        if((*buf)[i] == ESCAPE)
                --newBufSize;
    }

    unsigned char* newBuf = (unsigned char*)malloc(newBufSize);
    int escapedBytesOffset = 0;

    //Flag on each end
    newBuf[0] = FLAG;
    newBuf[newBufSize - 1] = FLAG;
    //Copy the rest with respective byte stuffing when necessary
    for(i = 1; i < bufSize - 1; ++i){
        if((*buf)[i] == ESCAPE){
                newBuf[i - escapedBytesOffset] = (*buf)[i+1] ^ 0x20;
                ++i;
                ++escapedBytesOffset;
        }
        else
```

```
                newBuf[i - escapedBytesOffset] = (*buf)[i];
    }

    free(*buf);
    *buf = newBuf;
    return newBufSize;
}

unsigned char getBCC2(const unsigned char* buf, unsigned int size) {
    unsigned char BCC2 = buf[0];

    unsigned int i;
    for (i = 1; i < size; ++i)
        BCC2 ^= buf[i];

    return BCC2;
}

int getMessageSequenceNumber(unsigned char* msg, int type){
    if(type == SU_MESSAGE) return (msg[2] >> 7) & 0x01;
    if(type == I_MESSAGE) return (msg[2] >> 6) & 0x01;
    return -1;
}

int getIMessageSize(unsigned char* message){
    if(message[4] == 1) //Data packet
        //4 bytes (C, N, L1, L2) + data size calculated by 256*L2+L1
        return 4 + (message[6] << 8) + message[7];
    //Else, a control package
    //5 bytes (C, T1, L1, T2, L2) + size of each file parameter given by L1 and L2
    return 5 + message[6] + message[7+message[6]+1];
}
```

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include "ApplicationLayer.h"
#include "Utilities.h"

void start();

int main(int argc, char** argv) {
    start();
    return 0;
}

void start() {
    printf("Is this the transmitter (0) or the receiver (1)? ");
    int mode = getIntegerInRange(0, 1);
```

```
    printf("\n");

    printf("Choose the baudrate (1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200): ");
    int baudrate = getBaudrate();
    printf("\n");

    printf("Choose the maximum size for a data packet: ");
    int messageDataMaxSize = getIntegerInRange(1, 65535);
    printf("\n");

    printf("Choose maximum number of retries before aborting: ");
    int numRetries = getIntegerInRange(0, 10);
    printf("\n");

    printf("Choose the timeout value: ");
    int timeout = getIntegerInRange(1, 10);
    printf("\n");

    printf("Choose the port (x in /dev/ttySx) to be used: ");
    int portNum = getIntegerInRange(0, 9);

    char port[20];
    sprintf(port, "/dev/ttyS%d", portNum);
    printf("Using port: %s\n", port);
    printf("\n");

    char file[MAX_SIZE];
    if(mode == SEND){
        printf("Filename: ");
        scanf("%s", file);
        printf("\n");
    }

    // Initialize random seed
    srand((unsigned) time(NULL));

    initApplicationLayer(port, mode, baudrate, messageDataMaxSize, numRetries, timeout,
file);
}
```

Utilities.h

```
#pragma once

#define MAX_SIZE 256

int getIntegerInRange(int lower, int upper);
int getBaudrate();
void clearStdIn();
```

Utilities.c

```
#include <termios.h> // For baudrate constants
#include <stdio.h>
#include "Utilities.h"

int getIntegerInRange(int lower, int upper) {
    int number;
    int valid = 0;
    while (!valid) {
        if (scanf("%d", &number) == 1 && lower <= number && number <= upper)
                valid = 1;
        else
                printf("Number must be in range [%d, %d]: ", lower, upper);
        clearStdIn();
    }
    return number;
}

int getBaudrate() {
    int number;
    do{
        number = getIntegerInRange(0, 115200);
        switch (number) {
        case 1200:
                return B1200;
        case 1800:
                return B1800;
        case 2400:
                return B2400;
        case 4800:
                return B4800;
        case 9600:
                return B9600;
        case 19200:
                return B19200;
        case 38400:
                return B38400;
        case 57600:
                return B57600;
        case 115200:
                return B115200;
        default:
                break;
        }
        printf("Invalid baudrate. Choose one from the shown list: ");
    }while(1);
}

void clearStdIn() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}
```

Anexo II - Diagramas de chamadas de funções

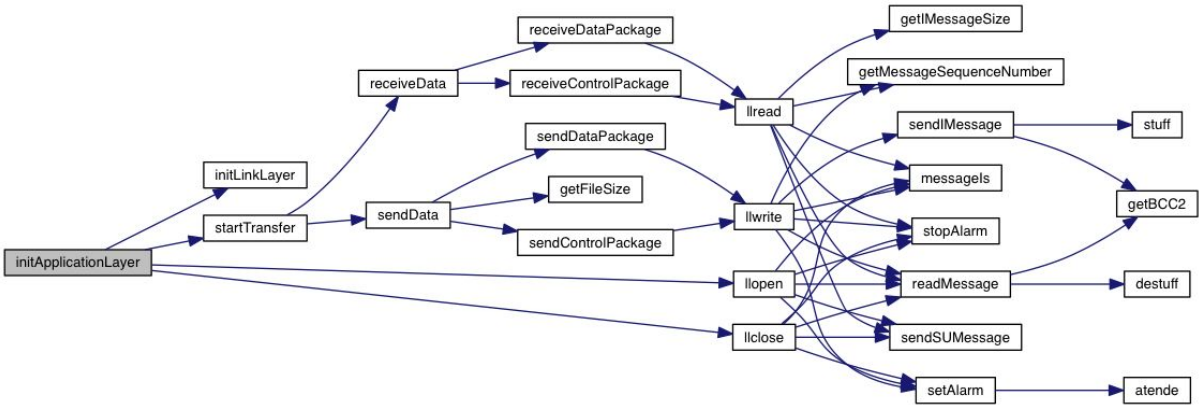


Diagrama 1 - Chamada de funções do programa

Anexo III - Figuras

```c
int llwrite(const unsigned char* buf, unsigned int bufSize) {
    int try = 0, transferring = 1;
    while (transferring) {
        if (try == 0 || alarmWentOff) {
            /* Aborto do processo caso
            o número de tentativas exceda o máximo */
            sendIMessage(buf, bufSize);
            //Programação inicial do alarme
        }

        int type;
        unsigned char* receivedMessage = readMessage(&type, NULL);

        if (messageIs(receivedMessage, type, RR)) {
            int nr = getMessageSequenceNumber(receivedMessage, type);
            //processamento do número de sequência
            //sucesso: sair do ciclo
        }
        else if (messageIs(receivedMessage, type, REJ))
            //reiniciar mecanismo
    }
    //retorno com sucesso
}
```

```c
int llread(unsigned char** message) {
    unsigned char* msg = NULL;
    int msgSize = 0;
    int done = 0;
    while (!done) {
        int type, hasBCC2Error, msgNs;
        msg = readMessage(&type, &hasBCC2Error);
        switch (type) {
        case INV_MESSAGE:
            printf("INVALID message received.\n");
            if (hasBCC2Error)
                sendSUMessage(REJ);
            break;
        case SU_MESSAGE:
            //sair do ciclo se mensagem recebida for DISC
        case I_MESSAGE:
            msgNs = getMessageSequenceNumber(msg, type);
            msgSize = getIMessageSize(msg);
            if (ll->ns == msgNs) {
                //atualizar ns e retirar bloco de dados da trama
                sendSUMessage(RR);
            } else {
                //bloco provavelmente repetido, confirmar na mesma
                sendSUMessage(RR);
            }
            break;
        }
    }
    //desativar alarme e retornar
}
```

Figuras 1 e 2 - Excertos de código demonstrando a presença das principais funções auxiliares na API do protocolo de ligação de dados (excertos de código cortados para facilitar leitura)

```
void sendIMessage(const unsigned char* message, unsigned int messageSize) {
  //Creating the package
  //Reserva de espaço para a trama e determinação de bytes de endereço, controlo...
  unsigned char BCC2 = getBCC2(message, messageSize);

  //Criação da trama

  messageSize += I_MESSAGE_BASE_SIZE;
  messageSize = stuff(&msg, messageSize);

  //Actually sending
  //Envio da trama
}
```

Figura 3 - Excerto da implementação da função auxiliar sendIMessage que envia uma trama do tipo I que faz uso das auxiliares getBCC2 e stuff (excertos de código cortados para facilitar leitura)

```c
int sendData() {
  //Abrir ficheiro e obter tamanho, em bytes

  if (!sendControlPackage(I_PACKAGE_START, fileSizeBuf, al->fileName))
    return -1;

  char* fileBuf = malloc(al->maxSize);
  unsigned int nBytesRead = 0, i = 0;
  while ((nBytesRead = fread(fileBuf, sizeof(char), al->maxSize, file)) > 0) {
    if (!sendDataPackage((i++) % 255, fileBuf, nBytesRead))
      //Não conseguiu mandar corretamente: liberta recursos,
      //aborta processo e retorna com erro
  }

  //Fecha ficheiro

  if (!sendControlPackage(I_PACKAGE_END, fileSizeBuf, al->fileName))
    return -1;
  return 1;
}
```

```c
int receiveData() {
  //Declaração de variáveis auxiliares locais
  if (!receiveControlPackage(&controlStart, &fileSize, &fileName))
    //Não conseguiu ler o START control package: aborta e retorna com erro
  if (controlStart != I_PACKAGE_START)
    //Outro package chegou no lugar do START quando devia ser este último
    //Aborta e retorna com erro

  //Criação do ficheiro de output com o nome proveniente do START package
  int N = -1;
  unsigned int totalSizeRead = 0;
  while (totalSizeRead != fileSize) {
    //Declaração de variáveis auxiliares
    if (!receiveDataPackage(&N, &fileBuf, &length))
      //Não conseguiu ler pacote corretamente
      //Aborta função, liberta recursos e retorna com erro

    //Controlo do número do pacote
    //Erro caso não seja o esperado
    fwrite(fileBuf, sizeof(char), length, outputFile);
  }
  //Fecho do ficheiro de output
  if (!receiveControlPackage(&controlEnd, 0, NULL))
    //Não conseguiu ler o END control package: aborta e retorna com erro
  if (controlEnd != I_PACKAGE_END)
    //Outro package chegou no lugar do END quando devia ser este último
    //Aborta e retorna com erro
  return 1;
}
```

Figuras 4 e 5 - Overview do uso das funções auxiliares no processo de transferência do ficheiro do lado do emissor e receptor, respetivamente (excertos de código cortados para facilitar leitura)