Projeto de RCOM

Protocolo de Ligação de Dados

Engenharia Informática e Computação

Regente: Manuel Alberto Pereira Ricardo

Turma 6

- **Eduardo Luís Pinheiro da Silva – up201603135 - up201603135@fe.up.pt**
- **João Pedro Viveiros Franco – up201604503 - up201604503@fe.up.pt**
- **Tomás Nuno Fernandes Novo – up201604503 - up201604503@fe.up.pt**

Sumário

O projeto “Protocolo de Ligação de Dados” consiste na realização de uma transferência de ficheiros entre um computador emissor e um recetor através de uma ligação por cabo pelas portas de série de cada computador. Com o fim de prestar apoio ao projeto, foi elaborado este relatório.

O relatório apresenta conceitos fundamentais para a compreensão do projeto, complementando-o e deixando claro o término com sucesso do mesmo, visto que os objetivos ambicionados foram cumpridos.

1 - Introdução

No âmbito desta unidade curricular, foi-nos proposta como 1ª parte do trabalho prático a realização de um projeto que aborda o Protocolo de Ligação de Dados.

Neste projeto objetivámos não só implementar um protocolo de ligação de dados, de acordo com as determinadas especificações, como também testá-lo com uma simples aplicação de transferência de ficheiros.

Como referido no sumário, o projeto consiste na transferência de ficheiros de um computador para outro a partir das portas série.

A estrutura do relatório foi elaborada com o objetivo de auxiliar na interpretação do projeto concebido, sendo que esta se encontra dividida em:

1. **Introdução –** onde são referidos os objetivos do trabalho e do relatório bem como os tipos de informação que poderão ser encontrados na restante estrutura.
2. **Arquitetura –** secção que explicita a interface do utilizador e aborda os blocos funcionais.
3. **Estrutura do Código –** informações acerca das estruturas de dados usadas e principais funções utilizadas.
4. **Casos de Uso Principais –** funcionalidades sequenciais fornecidas pelo projeto
5. **Protocolo de Ligação Lógica –** na qual se descreve a sua implementação e se identifica os principais aspetos funcionais.
6. **Protocolo de Aplicação –** descrição da sua implementação e identificação dos aspetos funcionais mais relevantes no que refere à aplicação
7. **Validação –** projeção dos testes efetuados
8. **Eficiência do Protocolo de Ligação de Dados –** caraterização estatística da eficiência do protocolo.
9. **Conclusões –** síntese de toda a informação e ilações retiradas da realização do projeto.

2 - Arquitetura

A aplicação de transferência de ficheiros encontra-se organizada em duas camadas: **Aplicação** e **Ligação de Dados**. A camada de **Aplicação** encarrega-se da leitura e escrita do ficheiro a partir dos pacotes de dados recebidos, enquanto que é na camada de **Ligação de Dados** que está implementado o respetivo protocolo de ligação de dados, incluindo as funções responsáveis pelo estabelecimento da ligação e pela sincronização de tramas.

Durante a execução da transferência de dados, podemos verificar interfaces distintas no computador emissor e no recetor. No computador emissor é possível observar a quantidade de bytes que faltam para finalizar a transferência. Por sua vez, o recetor apresenta uma barra de progresso, assim como a percentagem do ficheiro recebido, durante o decorrer da transferência. Em ambos os casos também se pode observar a taxa de transferência do ficheiro. Apesar das diferenças, é possível verificar em ambos mensagens de controlo, indicando o sucesso ou erro da aplicação.

3 – Estrutura do código

A camada da **Aplicação** é reproduzida por uma *struct* designada por ***applicationLayer*** na qual são armazenadas informações acerca do descritor correspondente à porta série, o seu estado, a indicação de emissão/receção e o índice do pacote de dados.

```
typedef struct
{
        int fileDescriptor; /*Descritor correspondente à porta série*/
        int     status; /* 0 - closed, 1 - transfering, 2 - closing */
        int flag; /*TRANSMITTER | RECEIVER*/
        int dataPacketIndex; //Data Packet Number
} applicationLayer;
```

Figura 1- *applicationLayer*

As suas principais funções são:

- ***int sendFile(char* filename, char* device);***
  Responsável pela alocação em memória do ficheiro para posterior divisão em pacotes de dados que serão passados à camada de **Ligação de Dados** para respetivo envio, com recurso a uma **máquina de estados.**
- ***int receiveFile (char *device);***
  A qual recebe o ficheiro e inicia a máquina de estados, consolidando-os num único ficheiro dentro desta.
- ***int readDataPacket(int *fd, applicationLayer *app, char *buffer, char *filename, int *fileSize, int packetSize, int* bytesReceived);***
  Esta recebe os pacotes de dados, interpretando o seu conteúdo.

Tanto o princípio como o final da transferência são sinalizados com o envio por parte do emissor de um pacote de controlo que inclui o nome e tamanho, na qual o valor do Campo de Controlo indicará o respetivo início ou fim da transferência.

Por sua vez, a camada de **Ligação de Dados** é representada por uma estrutura de dados na qual é armazenada a porta série, a Baud Rate, o número de sequência, o intervalo de tempo de *time out* e o número de retransmissões máximo.

```
typedef struct
{
        char port[20];
        int baudRate;
        unsigned int sequenceNumber;
        unsigned int timeout;
        unsigned int numTransmissions;

} linkLayer;
```

Figura 2- *linkLayer*

As suas funções mais essenciais são:

- ***int llopen(int fd, int flag);***
  Esta função é responsável pelo estabelecimento da ligação entre o emissor e o recetor, certificando-se que é possível a transferência enviando tramas de supervisão com diferentes campos de controlo.
- **int llwrite(int fd, char * buffer, int length);**
  Chamada pelo emissor, é incutida a esta função a responsabilidade de criar a estrutura das tramas a partir dos dados que lhe são passados como argumento e de as enviar para o emissor posteriormente após *stuffing*.
- **int llread(int fd, char * buffer);**
  Função que recebe as tramas e analisa o conteúdo dos cabeçalhos e das caudas, efetuando também *destuffing* destas.
- **int llclose(int fd, flag);**
  Função que não só encerra a porta série e termina a ligação como também envia tramas de supervisão indicando o término com sucesso da ligação.

4 – Casos de Uso Principais

Podemos identificar como principais casos aqueles responsáveis pela transferência de ficheiros, nomeadamente a interface que possibilita a escolha de qualquer ficheiro por parte do utilizador e a transferência do respetivo ficheiro do transmissor para o recetor.

A sequência da transmissão de dados é a seguinte:

1. **Escolha do ficheiro a enviar na função na função *main*.**
2. **Estabelecimento de uma ligação através da função *llopen*.**
3. **Criação dos pacotes de dados na máquina de estados do emissor e posterior organização em tramas e envio em *llwrite*.**
4. **Receção das tramas em *llread* e análise do seu cabeçalho (*headCheck*) e cauda (*trailerCheck*). Posterior análise dos pacotes de dados em *readDataPacket* e *checkControlDataPacket*.**
5. **Envio da resposta adequada ao emissor.**
6. **Análise da resposta recebida pelo emissor.**
7. **Escrita dos dados recebidos num ficheiro na máquina de estados do recetor.**
8. **Término da ligação através de *llclose*.**

5 – Protocolo de Ligação Lógica

As funções ***llopen, llwrite, llread*** e ***llclose*** são a base para a transferência e receção dos dados. Usadas pela camada da aplicação, são implementados pelo protocolo de ligação lógica.

A função ***llopen*** é chamada pelo emissor, enviando o comando ***SET*** aguardando o envio do comando ***UA*** por parte do recetor. Se não houver resposta do recetor, o comando ***SET*** é enviado novamente. Se atingir o número máximo de tentativas de envio sem resposta, o programa encerra com erro. Caso haja envio da trama ***UA*** pelo recetor, significa que a ligação foi bem-sucedida.

Por sua vez é a função ***llwrite*** a responsável pelo envio do ficheiro de tamanho *length* contido no *buffer* a transmitir, criando tramas de informação a partir dos pacotes de dados recebidos (além do processo de *stuffing*), sendo posteriormente enviadas.

O recetor invoca a função ***llread*** com o fim de receber as tramas num ciclo contínuo. Recorremos a uma máquina de estados para auxiliar neste aspeto, sendo que a função ***llread*** é responsável pela receção, *destuffing* e análise do cabeçalho e cauda das mesmas.

Após a invocação de ***llclose*** por parte do emissor, é enviada a trama ***DISC*** e é esperada a receção do comando ***DISC*** enviado pelo recetor. Após receber com sucesso a trama enviada pelo recetor, o emissor envia a trama ***UA*** e a ligação é encerrada com sucesso. A receção sem sucesso no máximo das tentativas permitidas destes comandos resulta num término do programa retornando erro.

6 – Protocolo de Aplicação

O programa inicia em cada um dos computadores e é pedido ao utilizador que escreva no emissor o nome do ficheiro a transmitir, estando a aplicação prevenida para um eventual *input* incorreto por parte do utilizador.

A camada da aplicação é inicializada na função ***sendFile*** quando esta chama a máquina de estados auxiliar ao transmissor. Na mesma, ***stateMachine,*** é aberta o *file descriptor* da porta série através da função ***openPort*** inicializando então a camada de ligação de dados, sendo que é esta função que invoca a ***llopen***.

Tanto no emissor como no recetor, a função principal é a da máquina de estados. Na primeira é alocado em memória o ficheiro a transmitir e divisão deste em pacotes de dados, enviando antes destes os pacotes de controlo ***START*** e após os pacotes de dados o pacote de controlo ***END,*** sendo que estas contêm o nome e tamanho do ficheiro a enviar. Os dados são então enviados na função ***llwrite*** e o computador emissor fica à espera de resposta por parte do recetor. Caso obtenha como resposta por parte dele a trama ***REJ***, o programa informa do envio incorreto da trama e envia-a de novo caso o número de sequência seja igual à dele. Caso contrário, envia a trama anterior. Se não houver resposta por parte do recetor no intervalo de tempo esperado, a trama é enviada novamente até ser excedido o número possível de tentativas. No entanto, caso seja recebida a trama ***RR***, a receção foi efetuada sem problemas e prossegue para a transferência da trama seguinte.

Na execução da ***receiveFile,*** é invocada a máquina de estados alusiva ao recetor, sendo que esta última é a que chama ***llread,*** a qual é responsável pela receção dos dados. As funções responsáveis pelas verificações da chegada correta dos dados em si são as funções ***readDataPacket*** e ***checkControlDataPacket***.

Após o ficheiro ter sido totalmente enviado, assim como o pacote de controlo ***END,*** as máquinas de estados de cada computador invocam a função ***llclose*** que encerra a porta série.

7 – Validação

Com o fim de por à prova a aplicação concebida, realizaram-se testes que consistiam na transferência de ficheiros de variados tipos. Todos os testes foram realizados com êxito, inclusive aqueles nos quais a transmissão era interrompida ou eram inseridos *bits* aleatórios nos dados transferidos.

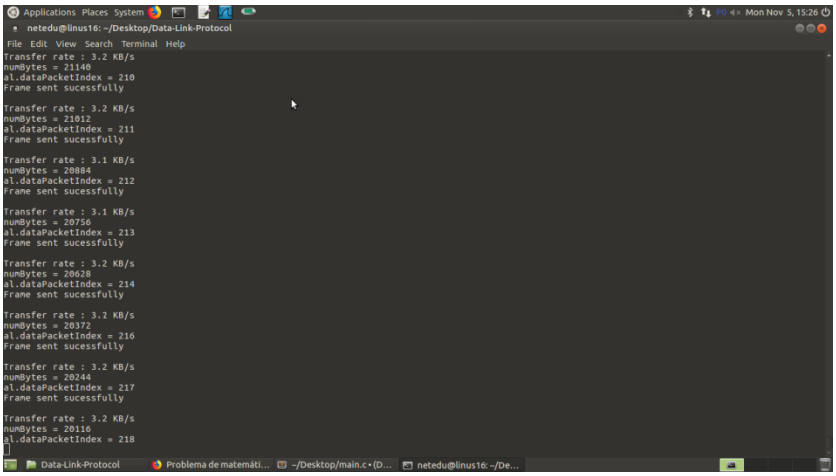


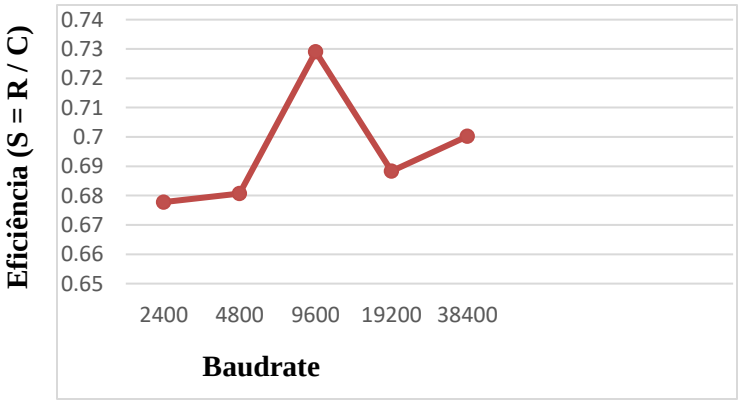
Figura 3- *Visão no emissor*

Figura 4 e 5 - *Visão no recetor quando a meio e no final da transferência.*

8 – Eficiência do protocolo de ligação de dados

Caraterizando estatisticamente a eficiência do nosso protocolo, consideramos ser as suas fundamentais idiossincrasias as seguintes:

- Quanto à eficiência em função da baudrate:



- Quanto à eficiência em função do tamanho da trama:

Tamanho da trama	Eficiência
40	0.551366
80	0.652005
120	0.693418
160	0.716204

9 – Conclusões

Em suma, a realização deste projeto foi benéfico para os elementos constituintes do grupo devido à aquisição de conhecimentos sobre o funcionamento das portas série.

Sintetizando a informação que apresentámos neste relatório, a aplicação encontra-se dividida em duas camadas independentes entre si: **Aplicação** e **Ligação de Dados**. Através de uma máquina de estados em cada computador, é realizada a transferência do ficheiro, sendo a análise dos dados concretos realizada pela camada da aplicação enquanto que o tratamento de erros nas tramas é realizado na camada da ligação de dados.

Os objetivos a atingir foram alcançados, visto que implementámos o pretendido Protocolo de Ligação de Dados conforme nos foi proposto e com uma eficiência bastante semelhante ao espectável, sendo então possível a transmissão com sucesso de um ficheiro de um computador para outro usufruindo das portas-série, sendo tratados eventuais erros que possam ocorrer durante o processo.

Anexos

main.c

```
#include "constants.h"
#include "transmitter.h"
#include "receiver.h"
#include "utilities.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
	if ( (argc != 3) || ((strcmp("/dev/ttyS0", argv[1])!=0) && (strcmp("/dev/ttyS1", argv[1])!=0) &&
	(strcmp("/dev/ttyS2", argv[1])!=0)))
	{
		printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
		exit(1);
	}

	if (strcmp(argv[2], "transmitter") == 0)
	{
		char filename[100], *newLine;
		printf("Filename: ");
		fgets(filename, 100, stdin);

		if ((newLine = strchr(filename, '\n')) != NULL)
		  *newLine = '\0';

		return sendFile(filename, argv[1]);
	}

	else if (strcmp(argv[2], "receiver") == 0)
	{
	return receiveFile(argv[1]);
	}
	else
	{
	printf("Must specify \"transmitter\" or \"receiver\" as second argument\n");
	return -1;
	}
	return 0;
}
```

constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H
#define BAUDRATE B38400 /* bit rate*/
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define BUFFER 255
#define FLAG 0x7E
#define ADDR 0x03
#define SET_C 0x03
#define DISC_C 0x0B
#define UA_C 0x07
#define RR_C 0x06
#define REJ_C 0x01
#define ESCAPE 0x7d
#define TRANSMITTER 0
#define RECEIVER 1
#define MAX_ATTEMPTS 3
#define TIMEOUT 3
#define DATASIZE 128 //Max frame size
#define MAX_FILE_SIZE 100000 //100 KB
#endif
```

transmitter.h

```
#ifndef TRANSMITTER_H
#define TRANSMITTER_H
#include "utilities.h"

linkLayer ll;
applicationLayer al;

int stateMachine(char* device, char* buffer, int size, char* filename);
int llwrite(int fd, char * buffer, int length);
int sendFile(char* filename, char* device);
#endif
```

transmitter.c

```
#include "transmitter.h"
#include "constants.h"
#include "utilities.h"

#include <fcntl.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

struct timeval writeTime, readTime;

void sigalrm_handler(int signal)
{
	if (ll.numTransmissions > 0)
	{
		printf("Message timed out! New attempt\n");
		ll.numTransmissions--;
	}
	else
	{
		printf("Maximum number of attempts reached - Exiting\n");
		exit(1);
	}

	printf("Message timed out!\n");
}

int stateMachine(char* device, char* buffer, int size, char* filename)
{
	al.status = 0;
	al.flag = TRANSMITTER;
	al.dataPacketIndex = 0;

	ll.sequenceNumber = 0;
	ll.numTransmissions = MAX_ATTEMPTS;

	int packageSize = 0, numBytes;

	char** packageArray = calloc((size/DATASIZE + 1) + 2, sizeof(char*));
```

```
while (1)
{
        if (al.status == 0) // Closed
        {
            al.fileDescriptor = openPort(device, al.flag);
           if (al.fileDescriptor > 0)
           {
              al.status = 1;
           }
         }
         else if (al.status == 1) // Transfering
        {
          if (packageArray[al.dataPacketIndex] == NULL)
          {
           packageSize = 0;
           packageArray[al.dataPacketIndex] = malloc(DATASIZE + 4 + 1);
           if (al.dataPacketIndex == 0) // Start
            {
               packageArray[al.dataPacketIndex][1 + packageSize++] = 2; // C (2 - start)
               packageArray[al.dataPacketIndex][1 + packageSize++] = 0; // field type (file size)
               packageArray[al.dataPacketIndex][1 + packageSize++] = sizeof(int); // Number of bytes of field
               memcpy(&packageArray[al.dataPacketIndex][1 + packageSize], &size, sizeof(int));
               packageSize += sizeof(int);
               packageArray[al.dataPacketIndex][1 + packageSize++] = 1; // field type (file size)
               packageArray[al.dataPacketIndex][1 + packageSize++] = strlen(filename)+1; // Number of bytes of field
               memcpy(&packageArray[al.dataPacketIndex][1 + packageSize], filename, strlen(filename) + 1);
               packageSize += strlen(filename) + 1;
           }
            else if (al.dataPacketIndex == (size/DATASIZE + 1 + 1)) // End
            {
                packageArray[al.dataPacketIndex][1 + packageSize++] = 3; // C (3 - end)
                packageArray[al.dataPacketIndex][1 + packageSize++] = 0; // field type (file size)
                packageArray[al.dataPacketIndex][1 + packageSize++] = sizeof(int); // Number of bytes of field
                memcpy(&packageArray[al.dataPacketIndex][1 + packageSize], &size, sizeof(int));
                packageSize += sizeof(int);
                packageArray[al.dataPacketIndex][1 + packageSize++] = 1; // field type (file size)
                packageArray[al.dataPacketIndex][1 + packageSize++] = strlen(filename)+1; // Number of bytes of field
                memcpy(&packageArray[al.dataPacketIndex][1 + packageSize], filename, strlen(filename) + 1);
                packageSize += strlen(filename) + 1;
             }
            else // Data Packages
            {
                 numBytes = (size  - DATASIZE*(al.dataPacketIndex-1));
                 printf("numBytes = %i\n", numBytes);

              if (numBytes > DATASIZE)
                   numBytes = DATASIZE;
```

```
                    packageArray[al.dataPacketIndex][1 + packageSize++] = 1; // C (1 - data)
                    packageArray[al.dataPacketIndex][1 + packageSize++] = (al.dataPacketIndex) % 255; // Sequence number
                    packageArray[al.dataPacketIndex][1 + packageSize++] = numBytes / 256; // The 8 most significant bits in
the packageSize.
                     packageArray[al.dataPacketIndex][1 + packageSize++] = numBytes % 256;
                     memcpy(&packageArray[al.dataPacketIndex][1 + packageSize], buffer+(al.dataPacketIndex-
1)*DATASIZE, numBytes);
                      packageSize += numBytes;
                     }
                packageArray[al.dataPacketIndex][0] = packageSize;
               }

   if (llwrite(al.fileDescriptor, packageArray[al.dataPacketIndex]+1, (unsigned
char)packageArray[al.dataPacketIndex][0]) < 0)
    return -1;

  char received[5];
  alarm(TIMEOUT);
  int bytes = read(al.fileDescriptor, received, 5);
  alarm(0);
  if (bytes > 0)
  {
      ll.numTransmissions = MAX_ATTEMPTS;
      if (gettimeofday(&readTime, NULL) != 0)
          printf("Error getting time!\n");
     unsigned char control = messageCheck(received);
     unsigned char sequenceNumber = (control & 0x80) >> 7;
     control = 0x0F & control;
          if(control == RR_C)
          {
              if(sequenceNumber == al.dataPacketIndex)
                {
                     printf("Sequence Error\n");
                }
               else
                {
                     printf("Frame sent sucessfully\n\n");
                     al.dataPacketIndex++;
                     ll.sequenceNumber = al.dataPacketIndex % 2;
                 }
            }
           else if(control == REJ_C)
            {
              if(sequenceNumber == al.dataPacketIndex)
               {
                  printf("Corrupt frame sent, sending same frame again!\n\n");
                }
```

```
            else
              {
                  printf("Frame ahead of receiver \n");
                  al.dataPacketIndex--;
                  ll.sequenceNumber = al.dataPacketIndex % 2;
               }
             }
            else
              printf("Unknown message\n");

          }

          double deltaTime = (double)(readTime.tv_sec - writeTime.tv_sec) + (double)(readTime.tv_usec -
          writeTime.tv_usec)/1000/1000; // In seconds

          if (((float)DATASIZE / deltaTime)/1024 > 0)
               printf("Transfer rate : %.1f KB/s\n", ((float)DATASIZE / deltaTime)/1024);

          if (al.dataPacketIndex > (size/DATASIZE + 1 + 1))
          {
                    al.status = 2;
          }
        }

else if (al.status == 2) // Closing
  {
          // Frees buffers from file transfer
          int j;
          for (j = 0; j < (size/DATASIZE + 1) + 2; j++)
          {
               free(packageArray[j]);
          }
          free(packageArray);
          ll.numTransmissions = MAX_ATTEMPTS;
           int error;
          error = llclose(al.fileDescriptor, TRANSMITTER);
          break;
   }
  }
 return 0;
}
```

```
int sendFile(char* filename, char* device)
{
	struct sigaction sigalrmaction;

	sigalrmaction.sa_handler = sigalrm_handler;
	sigemptyset(&sigalrmaction.sa_mask);
	sigalrmaction.sa_flags = 0;

	if (sigaction(SIGALRM, &sigalrmaction, NULL) < 0)
	{
		fprintf(stderr,"Unable to install SIGALRM handler\n");
		return 1;
	}

	int fd = open(filename, O_RDONLY);
	struct stat st;

	if (stat(filename, &st) != 0)
	{
		printf("Missing file!\n");
		return 1;
	}

	int size = st.st_size;
	printf("Size = %i\n", size);

	char* buffer = malloc(size);

	int i, bufferSize = 1024, numBytes = bufferSize;

	for (i = 0; numBytes == bufferSize; i++)
	{
		numBytes = read(fd, buffer+i*bufferSize, bufferSize);

		if (numBytes < 0)
		{
			printf("Error reading file!\n");
			return 1;
		}
	}
	close(fd);

	int ret = stateMachine(device, buffer, size, filename);

	free(buffer);
	return ret;
}
```

```
int llwrite(int fd, char * buffer, int length)
{
	char package[6 + 2*length];
	int i, j, packageSize = 6+length;

	package[0] = FLAG;
	package[1] = ADDR;
	package[2] = ll.sequenceNumber << 6;
	printf("al.dataPacketIndex = %u\n", al.dataPacketIndex);

	package[3] = package[1] ^ package[2];

	int packageLength = length;

	for (i = 0; i < length; i++) // Measures size of data
	{
		if (buffer[i] == FLAG || buffer[i] == ESCAPE)
			packageLength++;
	}

	for (i = 0; i < length; i++) // Transfers data from buffer to package and  calculates BCC2
	{
		package[4+i] = buffer[i];

		if (i == 0)
			package[packageSize-2] = buffer[i];
		else
			package[packageSize-2] ^= buffer[i];
	}

	for (i = 4; i < packageSize - 1; i++) // Stuffing
	{
		if (package[i] == FLAG)
		{
			shiftRight(package, packageSize+1, i+1, 1);
			packageSize++;

			package[i] = ESCAPE;
			package[i+1] = 0x5e;
			i++;
		}
		else if (package[i] == ESCAPE)
		{
			shiftRight(package, packageSize+1, i+1, 1);
			packageSize++;

			package[i] = ESCAPE;
```

```
                        package[i+1] = 0x5d;
                        i++;
                }

        }

        package[packageSize-1] = FLAG;

        // printArray(package, packageSize);

        if (gettimeofday(&writeTime, NULL) != 0)
                printf("Error getting time!\n");

        int written = write(fd, package, packageSize);

        if (written < 0)
        {
                printf("Error in transmission\n");
                return -1;
        }

        // printf("Frame sent with %i bytes!\n", written);

        return written;
}
```

reveiver.h

```
#ifndef RECEIVER_H
#define RECEIVER_H

#include "utilities.h"
#include <sys/time.h>

linkLayer ll;
struct timeval writeTime2, readTime2;

int stateMachineReceiver(applicationLayer *al, char* device, int *fileSize, char *filename);
int receiveFile(char *device);
int llread(int fd, char *buffer);
int destuff(char* buffer, int* size);
char headerCheck(char received[]);
int sendAnswer(int fd, char control);
int readDataPacket(int *fd, applicationLayer *app, char *buffer, char *filename, int *fileSize, int packetSize, int*
bytesReceived);
int checkControlDataPacket(int i, char *buffer, char *filename, int *fileSize, int packetSize);
int trailerCheck(char received[], int size);

#endif
```

reveiver.c

```
#include "receiver.h"
#include "constants.h"
#include "transmitter.h"

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

int flag = 0;
```

```
void sigalrm_handlerR(int signal)
{
	if(ll.numTransmissions > 0)
	{
		printf("Message timed out!\nNew attempt\n");
		ll.numTransmissions--;
	}
	else
	{
		printf("Maximum number of attempts reached - Exiting\n");
		exit(1);
	}
}

int receiveFile(char *device)
{
	char filename[100];
	int fileSize;
	applicationLayer al;

	ll.baudRate = BAUDRATE;
	ll.sequenceNumber = 0;
	ll.numTransmissions = MAX_ATTEMPTS;
	//Ciclo
	stateMachineReceiver(&al, device, &fileSize, filename);

	return 0;
}
int stateMachineReceiver(applicationLayer *al, char* device, int *fileSize, char *filename)
{
	al->status = 0;
	al->flag = RECEIVER;
	al->dataPacketIndex = 0;

	char* dataRead = malloc(DATASIZE*2 + 6);
	int packetSize;
	int fd;
	int error;
	unsigned int bytesReceived = 0;

	while (1)
	{
	  if (al->status == 0) // Closed
	  {
		al->fileDescriptor = openPort(device, al->flag);
```

```
        if (al->fileDescriptor > 0)
        {
                al->status = 1;
                al->dataPacketIndex = 0;
        }

          printf("Open for connection\n");
 }
else if (al->status == 1) // Transfering
{
        if (gettimeofday(&readTime2, NULL) != 0)
            printf("Error getting time!\n");

        packetSize = llread(al->fileDescriptor, dataRead);

        if(packetSize == -1) // Error in data package
        {
            sendAnswer(al->fileDescriptor, (ll.sequenceNumber << 7) | REJ_C);
            //printf("Error in llread\n");
                continue;
         }

  error = readDataPacket(&fd, al, dataRead, filename, fileSize, packetSize, &bytesReceived);
        switch(error)
        {
           case -1:
                sendAnswer(al->fileDescriptor, (ll.sequenceNumber << 7) | REJ_C);
                                //printf("Error in Data Packet\n");
                                continue;

            case -2:
             sendAnswer(al->fileDescriptor, (((ll.sequenceNumber + 1) % 2) << 7) | RR_C);
                //printf("Receiver ahead of transmitter\n");
                continue;

            case -3:
            sendAnswer(al->fileDescriptor, (((ll.sequenceNumber + 1) % 2) << 7) | REJ_C);
                //printf("Transmitter ahead of receiver\n");
                continue;

        }
        al->dataPacketIndex++;
        packetSize = 0; // Clears dataRead array
        //printf("Received Packet\n");

        ll.sequenceNumber = (ll.sequenceNumber + 1) % 2;
```

```
                sendAnswer(al->fileDescriptor, (ll.sequenceNumber << 7) | RR_C);

                if (gettimeofday(&writeTime2, NULL) != 0)
                        printf("Error getting time!\n");

                system("clear");

                double deltaTime = (double)(writeTime2.tv_sec - readTime2.tv_sec) + (double)(writeTime2.tv_usec -
readTime2.tv_usec)/1000/1000; // In seconds

                if (((float)DATASIZE / deltaTime)/1024 >= 0)
                   printf("Transfer rate : %.1f KB/s\n", ((float)DATASIZE / deltaTime)/1024);

            if (bytesReceived / (double)*fileSize >= 0 && bytesReceived / (double)*fileSize <= 1)
                        printPercentage(bytesReceived / (double)*fileSize);


                   //printf("fileSize = %i\n", *fileSize);
                }
                else if (al->status == 2) // Closing
                {
                        free(dataRead);
                        close(fd);
                        error = llclose(al->fileDescriptor, RECEIVER);
                        break;
                }
        }
        return 0;
}

int llread(int fd, char * buffer)
{
        int i, j, numBytes = 1, receivedSize = 0;
        char temp[DATASIZE*2 + 6];

        while(1)
        {
                // alarm(TIMEOUT);
                numBytes = read(fd, &temp[receivedSize++], 1);
                // alarm(0);

                if (receivedSize > 1 && temp[receivedSize-1] == FLAG)
                        break;
        }

        int error = headerCheck(temp);
```

```
	if(error < 0)
	{
		printf("Error on header\n");

		int i;
		for (i = 0; i < receivedSize; i++)
			printf("%i, ", temp[i]);

		printf("\n");
		return error;
	}

	int dataPacketsSize = receivedSize - 6;

	if(dataPacketsSize > DATASIZE * 2)
	{
		printf("Too much data incoming\n");
		return -1;
	}
	dataPacketsSize++;
	destuff(temp + 4, &dataPacketsSize);
	dataPacketsSize--;
	if(trailerCheck(temp + 4, dataPacketsSize + 2) < 0)
	{
		printf("Error in Trailer\n");

		int i;
		for (i = 0; i < receivedSize; i++)
			printf("%i, ", temp[i]);

		printf("\n");

		return -1;
	}

	memcpy(buffer, temp + 4, dataPacketsSize);


	//printf("Data Packet size after destuffing: %d\n", dataPacketsSize);
	//ATTENTION: The information beyond dataPacketsSize will be untuched, remaining the same as when the buffer was first read

	return dataPacketsSize; //Application must not know frame structure, thus only the size of the data packets is needed
}
```

```
int destuff(char* buffer, int* size)
{
	int i;
	for (i = 0; i < *size; i++) // Destuffs the data package
	{
		if (buffer[i] == ESCAPE)
		{
			if (buffer[i+1] == 0x5e)
			{
				shiftLeft(buffer, *size, i+1, 1);
				(*size)--;
				buffer[i] = FLAG;
			}
			else if (buffer[i+1] == 0x5d)
			{
				shiftLeft(buffer, *size, i+1, 1);
				(*size)--;
				buffer[i] = ESCAPE;
			}
		}
	}
	return 0;
}


int trailerCheck(char received[], int size)
{
	char bcc2;
	int i;
	for (i = 0; i < size-2; i++)
	{
		if (i == 0)
			bcc2 = received[i];
		else
			bcc2 ^= received[i];
	}
	if(bcc2 != received[size-2])
		return -1;
	if(received[size - 1] != FLAG)
		return -1;
}
```

```
char headerCheck(char received[])
{
	char control, bcc1;
	int i;

	if (received[0] == FLAG && received[1] == ADDR)
	{
		control = received[2];

		control = control >> 6;

		//printf("Control: %d\n", control);
		//printf("Sequence Number: %u \n", ll.sequenceNumber);

		if(control != ll.sequenceNumber)
		{
			printf("Sequence error\n");
		}


		bcc1 = received[3];
		// printf("BCC1: %d\n", bcc1);

		if (bcc1 == received[1] ^ control)
			return control;
	}

	return -1;
}




int sendAnswer(int fd, char control)
{
	char buffer[5];

	buffer[0] = FLAG;
	buffer[1] = ADDR;
	buffer[2] = control;
	buffer[3] = buffer[1] ^ buffer[2];
	buffer[4] = FLAG;

	int written = write(fd, buffer, 5);

	if (written < 5)
		printf("Error sending anwser!\n");
```

```
	return written;
}

int readDataPacket(int *fd, applicationLayer *app, char *buffer, char *filename, int *fileSize, int packetSize, int*
bytesReceived)
{
	int i = 0;
	char controlByte = buffer[i];
	//printf("C: %d\n", controlByte);

	if (controlByte == 2) // Start
	{
		checkControlDataPacket(1, buffer, filename, fileSize, packetSize);
		*fd = open(filename, O_WRONLY | O_TRUNC | O_CREAT, 0777);
	}
	else
		if (controlByte == 1) // Data
		{
			unsigned char N = buffer[i + 1];
			unsigned char L2 = buffer[i + 2], L1 = buffer[i + 3];
			//printf("N: %u\n", N);
			//printf("al.dataPacketIndex-1 = %i\n", (app->dataPacketIndex-1) % 255);


			if(N >  (app->dataPacketIndex) % 255) // Transmitter ahead of receiver
				return -3;
			else if(N < (app->dataPacketIndex) % 255) // Receiver ahead of transmitter
				return -2;

			int K = 256 * L2 + L1;
			// printf("K: %d\n", K);

			if(K < 0)
			{
				printf("Error in packet size\n");
				return -1;
			}

			if(write(*fd, buffer+4, K) < 0)
			{
				printf("Error in writting to local file\n");
				return -1;
			}

			*bytesReceived += K;
```

```
                        //printf("--------- Data Packets Read ----------\n");
                        //printArray(buffer, K);

                }
                else
                        if (controlByte == 3) // End
                        {
                    checkControlDataPacket(packetSize - 3, buffer, filename, fileSize, packetSize);

                                app->status = 2;
                        }
        return 0;
}


int checkControlDataPacket(int i, char *buffer, char *filename, int *fileSize, int packetSize)
{
        int j;
        char T, L;

        for(i = 1; i < packetSize; i += 2 + L)
        {
                T = buffer[i];
                L = buffer[i + 1];

                /*
                printf("T: %d\n", T);
                printf("L: %d\n", L);
                printf("i = %d\n", i); */


                if(T == 0)
                {
                        memcpy(fileSize, buffer + i + 2, L);
                        printf("File Size: %d\n", *fileSize);
                }
                else
                {
                        if(T == 1)
                                memcpy(filename, buffer + i + 2, L);

                        printf("File Name: %s\n", filename);
                }

        }

        return 0;
```

```
}
```

utilities.h

```
#ifndef UTILITIES_H
#define UTILITIES_H

#include <sys/types.h>
#include <termios.h>
#include "constants.h"

typedef struct
{
	int fileDescriptor; /*Descritor correspondente à porta série*/
	int	status; /* 0 - closed, 1 - transfering, 2 - closing */
	int flag; /*TRANSMITTER | RECEIVER*/
	int dataPacketIndex; //Data Packet Number
} applicationLayer;

typedef struct
{
	char port[20];
	int baudRate;
	unsigned int sequenceNumber;
	unsigned int timeout;
	unsigned int numTransmissions;
	char frame[DATASIZE * 2 + 6];
} linkLayer;

struct termios oldtio,newtio;

void swap(char* a, char*b);
void printPercentage(double percentage);
int abs(int a);
void shiftRight(char* buffer, int size, int position, int shift);
void shiftLeft(char* buffer, int size, int position, int shift);
void printArray(char* arr, int length);
int messageCheck(char received[]);
int openPort(char* device, int flag);
int llopen(int fd, int flag);
int llclose(int fd, int flag);

#endif
```

utilities.c

```
#include "utilities.h"
#include "constants.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <fcntl.h>
#include <unistd.h>

void swap(char* a, char*b)
{
	char temp = *a;
	*a = *b;
	*b = temp;
}

int abs(int a)
{
	if (a < 0)
		return -a;
	return a;
}

void printPercentage(double percentage)
{
	printf("<");

	int i, length = 15 /* length of the percentage bar */;
	for (i = 0; i < length; i++)
	{
		if ((double)i/length < percentage)
			printf("|");
		else
			printf(" ");
	}

	printf(">%.1f%%\n", percentage*100);
}
```

```
void shiftRight(char* buffer, int size, int position, int shift)
{
	int i, j;

	for (j = 0; j < shift; j++)
	{
		size++;
		buffer[size-1] = 0;

		for (i = size-2; i >= position; i--)
		{
			swap(&buffer[i], &buffer[i+1]);
		}

		position++;
	}
}

void shiftLeft(char* buffer, int size, int position, int shift)
{
	int i, j;

	for (j = 0; j < shift; j++)
	{

		for (i = position-1; i < size; i++)
		{
			swap(&buffer[i], &buffer[i+1]);
		}

		size--;
		position--;
	}
}


void printArray(char* arr, int length)
{
	int i;
	for (i = 0; i < length; i++)
	{
		printf("%i\n", arr[i]);
	}
```

```
	printf("\n");
}


int messageCheck(char received[])
{
	char control, bcc1, bcc2;
	int i;

	if (received[0] == FLAG && received[1] == ADDR && received[4] == FLAG)
	{
		control = received[2];
		bcc1 = received[3];

		if (bcc1 == received[1] ^ control)
			return control;
	}

	return -1; //Error
}

int openPort(char* device, int flag)
{
	int fd = open(device, O_RDWR | O_NOCTTY);

	if (fd < 0)
	{
		printf("Unable to open serial port\n");
		exit(-1);
	}

	return llopen(fd, flag);
}

int closePort(int fd, int flag)
{
	// llclose(fd);
	// close(fd);

	return 1;
}

int llopen(int fd, int flag)
{
	int c, res;

  /*
```

```
        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.
*/

        if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
          perror("tcgetattr");
          exit(-1);
        }

        bzero(&newtio, sizeof(newtio));
        newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
        newtio.c_iflag = IGNPAR;
        newtio.c_oflag = 0;

        /* set input mode (non-canonical, no echo,...) */
        newtio.c_lflag = 0;

        newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
        newtio.c_cc[VMIN]     = 1;   /* blocking read until 1 chars received */



/*
        VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
        leitura do(s) próximo(s) caracter(es)
*/


        tcflush(fd, TCIOFLUSH);

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

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

        char buf[5];
        int received;



        if (flag == TRANSMITTER)
        {
                buf[0] = FLAG;
                buf[1] = ADDR;
                buf[2] = SET_C;
```

```
		buf[3] = buf[1] ^ buf[2];
		buf[4] = FLAG;

		if(write(fd, buf, 5) < 0)
		{
			printf("Error in transmission\n");
			return -1;
		}

		printf("SET sent!\n");


		alarm(TIMEOUT);

		received = read(fd, buf, 5);

		alarm(0);

		if(received < 0)
		{
			printf("Error in receiving end\n");
			return -1;
		}

		int status = messageCheck(buf);

		if (status != UA_C)
		{
			printf("Unknown message\n");
			return -1;
		}
	}
	else if (flag == RECEIVER)
	{
		// alarm(TIMEOUT);

		received = read(fd, buf, 5);

		// alarm(0);

		if(received < 0)
		{
			printf("Error in receiving end\n");
			return -1;
		}

		int status = messageCheck(buf);
```

```
                if (status != SET_C)
                {
                        printf("Unknown message\n");
                        return -1;
                }

                buf[0] = FLAG;
                buf[1] = ADDR;
                buf[2] = UA_C;
                buf[3] = buf[1] ^ buf[2];
                buf[4] = FLAG;

                if(write(fd, buf, 5) < 0)
                {
                        printf("Error in transmission\n");
                        return -1;
                }

                printf("Message sent!\n");
        }

        return fd;
}


int llclose(int fd, int flag)
{
        char buf[5];
        int received;

        // tcflush(fd, TCIFLUSH);

        if (flag == TRANSMITTER)
        {
                buf[0] = FLAG;
                buf[1] = ADDR;
                buf[2] = DISC_C;
                buf[3] = buf[1] ^ buf[2];
                buf[4] = FLAG;

                if(write(fd, buf, 5) < 0)
                {
                        printf("Error in llclose transmission\n");
                        return -1;
                }
```

```
        while (1)
        {
                received = read(fd, buf, 5);

                if (received < 0)
                {
                        printf("Error in receiving end in llclose\n");
                        continue;
                }

                unsigned char status = messageCheck(buf);

                if (status != DISC_C)
                {
                        // printf("DISC_C not received, received: %u\n", status);
                        continue;
                }

                break;
        }


        buf[2] = UA_C;
        buf[3] = buf[1] ^ buf[2];


        if (write(fd, buf, 5) < 0)
        {
                printf("Error in second llclose transmission\n");
                return -1;
        }

}
else if (flag == RECEIVER)
{
        while (1)
        {
                received = read(fd, buf, 5);

                if (received < 0)
                {
                        printf("Error in receiving end in llclose\n");
                        continue;
                }

                unsigned char status = messageCheck(buf);
```

```
                if (status != DISC_C)
                {
                        // printf("DISC_C not received, trying again\n");
                        continue;
                }

                break;
        }

        //printf("DISC received\n");

        buf[0] = FLAG;
        buf[1] = ADDR;
        buf[2] = DISC_C;
        buf[3] = buf[1] ^ buf[2];
        buf[4] = FLAG;

        if (write(fd, buf, 5) < 0)
        {
                printf("Error in llclose transmission\n");
                return -1;
        }

        //printf("DISC sent by receiver!\n");


        while (1)
        {
                received = read(fd, buf, 5);

                if (received < 0)
                {
                        printf("Error in receiving end in llclose\n");
                        continue;
                }

                unsigned char status = messageCheck(buf);

                if (status != UA_C)
                {
                        // printf("UA_C not received, trying again\n");
                        continue;
                }

                break;
        }
}
```

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

    close(fd);

    printf("Connection closed sucessfully!\n");

    return 0;
}
```

makefile

```
CC          = gcc
CFLAGS      = -g
LDFLAGS     =
DEPS        = main.c receiver.c receiver.h transmitter.c transmitter.h utilities.c utilities.h constants.h makefile
OBJFILES    = receiver.o transmitter.o utilities.o main.o
TARGET      = main

all: $(TARGET)

$(TARGET): $(OBJFILES) $(DEPS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)

clean:
    rm -f $(OBJFILES) $(TARGET) *~
```