Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

UC Redes de Computadores

Protocolo de Ligação de Dados

Professor Ricardo Morla

Rita Martinho, up201709727
Gonçalo Xavier, up201604506
João Loureiro, up201604453

Novembro 2019

Índice

Introdução e Sumário

O seguinte trabalho laboratorial foi realizado no âmbito da unidade curricular de Redes de Computadores, e consiste no desenvolvimento e implementação de um protocolo de ligação de dados entre dois computadores, através da conexão entre portas série (entre o recetor e o emissor). O trabalho foi feito seguindo uma metodologia de *Divide and Conquer*, i.e, foi-se resolvendo progressivamente problemas de menor complexidade (e testando à medida que se ia escrevendo o código) até chegar a uma solução final.

Como não podíamos estar sempre no laboratório, fomos testando o nosso trabalho criando uma ligação entre duas portas virtuais usando a ulilidade *socat* e o seguinte comando:

sudo socat PTY,link=/dev/ttyS0 PTY,link=/dev/ttyS4;

De notar que tínhamos sempre presente que isto é uma situação utópica e possivelmente não representativa dos erros e complicações que uma ligação porta série real impõe.

Neste relatório iremos focarmo-nos nos seguintes aspetos:

- Estrutura do código – onde explicamos os diversos ficheiros de linguagem C que criámos, como é que eles se interligam entre si e como promovem a independência entre camadas.
- Camada de ligação lógica – onde falamos especificamente sobre esta camada e as funções que lhe são alicerce.
- Camada de aplicação – onde explicamos a interação com o utilizador, a diferença entre modo *sender* e modo *receiver* e como é que esta camada se interliga com a anterior (camada de ligação lógica).
- *"Tools.c"* – onde explicamos este ficheiro C, a sua importância no trabalho e as diversas funções que o compoêm.
- Casos de usos principais – explicação da função main().
- Validação – onde indicamos a que tipo de testes subtemos o nosso programa e a sua conformidade com o protocolo que seguimos.
- Elementos de valorização – onde explicamos algumas funções e características extra-guião que implementámos de forma a valorizar o nosso trabalho.
- Caracterização estatística da eficência – onde expomos cálculos estatísticos que expoêm a eficiência da transmissão do nosso trabalho em relação a diversos fatores.
- Conclusão – síntese da informação descrita ao longo do relatório e reflexão sobre os objetivos de aprendizagem alcançados.
- Anexo – onde é apresentado o código fonte (num pdf à parte) e a Caracterização estatśitica da eficência.

Estrutura do Código

O nosso trabalho está dividido em 3 partes essenciais: a camada de aplicação, a camada de ligação de dados e um ficheiro .c a que demos o nome de “tools” por incluir todas as ferramentas que fomos precisando ao longo do projeto.

Camada de Ligação de Dados

A camada de ligação de dados é representada, neste contexto, pelos ficheiros *datalink.c* e *datalink.h* e a sua principal função é garantir a transmissão de tramas (i.e, dados) entre as duas portas série (entre os 2 computadores). É responsável por estabelecer e terminar a comunicação , escrever os dados na porta série e ler e verificar a integridade dos mesmos. Esta verificação é feita através do processo de *stuffing* (relativa à escrita de dados) e de *destuffing* (relativo à leitura).

É nesta camada que implementámos as funções *llopen(), llclose(), llwrite() e llread()* que implementam o supramencionado. É de realçar a implementação de timers e timeouts na função *llopen* – no lado do emissor – e na função *llwrite*, que apresentam mais uma forma de robustez a erros. Em caso de erro, todas as funções retornam um valor adequado.

Camada de Aplicação

A camada de aplicação é representada pelos nossos ficheiro ApplicationLayer.c, sender.c, receiver.c e respetivos ficheiros .h. Esta camada é responsável pela criação de pacotes de dados e de controlo, divisão por pacotes dos dados do ficheiro que se pretende transmitir, pela transferência do mesmos e pela leitura e escrita dos dados recebidos num ficheiro destino. É no sender.c que se trata do procedimento de envio do ficheiro e no receiver.c que se trata do processo de receção e escrita no ficheiro. É nesta camada que são chamadas as funções pretencentes à camada de ligação de dados.

Tools

No ficheiro tools.c é onde implementámos as diversas funções auxiliares que utilizamos tanto na camada de ligação como na de aplicação. Neste conjunto de funções encontra-se funções que, por exemplo, pegando nos dados do ficheiro, constroem pacotes de dados conforme o protocolo, outra função que recebe um pacote e o transforma em uma trama (novamente, seguindo o protocolo), outras funções que fazem o procedimento oposto. É neste ficheiro que existem também as funções que escutam a porta e lêm uma trama.

No ficheiro tools.h estão definidos diversos *defines*, por exemplo, em relação ao *baudrate*, ao tamanho do pacote, ao tamanho da trama, as diversas estruturas usadas no projeto, etc.

Camada de Ligação de Dados

llopen()

Numa primeira instância, a camada de aplicação trata de configurar a porta série e seguidamente de chamar a llopen(). Esta função é a responsável pelo estabelecimento da ligação através da porta série. Recebe como argumentos o *file descriptor(fd)* da porta série e o modo de conexão, isto é, se queremos chamar a função no modo sender ou modo receiver.

Genericamente, a função constrói tramas SET e UA a serem usadas tanto pelo receiver como pelo sender. Caso se trate do modo *sender*, a função primeiramente envia a trama SET escrevendo na porta série – o que significa que o sender está disponível a iniciar uma conexão, de seguida a função fica à espera da resposta UA do *receiver*, neste estado, a função fica à espera 3s (valor possivelmente alterado) e se não receber nada, volta a enviar a trama SET – este procedimento está configurado para apenas acontecer 3 vezes e caso isso acontece, a função retorna com uma mensagem de erro. Caso receba a trama UA, a conexão é estabelecida e a função retorna um valor apropriado.

Se, por outro lado, se tratar do *receiver,* a função começa por ficar à espera do comando SET e caso o receba, envia pela porta, o comando UA.

O mecanismo de *timeouts* é implementado nesta camada com bastante frequência de modo ao programa não ficar à espera infinitamente por algum evento.

llclose()

Esta função é, grosso modo, bastante semelhante à *llopen()* - recebe os mesmos argumentos e é dividida de igual forma entre *sender* e *receiver*. É responsável pelo término da conexão. Caso se trate do sender, a função envia pela porta o comando DISC, ficando à espera (com o mesmo mecanismo de timeouts usado na *llopen()*) de receber um DISC de volta do receiver, caso o receba, envia um comando UA para o mesmo e a conexão é terminada.

Caso se trate do modo receiver, este começa por ficar à espera da trama DISC e quando a recebe, encarrega-se de enviar outra trama DISC, de seguida, fica à espera novamente à espera, mas desta vez da trama UA que o *sender* terá enviado. Quando a receber, a conexão é terminada.

Depois de chamada na aplicação e caso retorne um valor correspondente a sucesso, a camada de aplicação dá reset das configurações da porta série que foram impostas antes da chamada da *llopen()*.

llwrite()

A função llwrite() só é chamada em modo *sender* e é a responsável pela escrita de dados. Como argumentos, recebe o fd da porta, um pacote da aplicação e o tamanho do mesmo. Inicialmente começa por construir as tramas RR e REJ (em conformidade com o

currente valor do ns – 0 ou 1), de seguida, e este é um passo de extrema importância, é criada uma trama a partir do pacote enviado pela camada de aplicação, para isso usando a função *buildFrame()* da tools.c. A função envia através da porta série a trama construída e fica à espera (em mecanismo *timeout*) de receber um RR (*receiver ready*), caso o receiver tenha recebido sem erros essa mesma trama ou um REJ (*reject*) caso o receiver tenha recebido a trama com erros. Se receber um RR, a função atualiza o seu valor de ns (de 1 para 0 ou de 0 para 1) e retorna o tamanho do que escreveu. Caso receba um REJ, a função tenta enviar a trama já enviada anteriormente não atualizando o ns.

llread()

Esta função só é chamada em modo *receiver* e é a responsável pela leitura de dados. Recebe o fd da porta série e tem outro argumento que é um *unsigned char* * que será utilizado para enviar por agumento a trama lida. A função começa por tentar ler da porta uma trama para isso chamando a função *readFrom Port()* da *tools.c* de seguida verifica se se trata duma trama duplicada, o recetor envia na mesma um RR mas não escreve na trama a enviar. Depois disto verifica o BCC1, isto é, o campo de proteção de dados do cabeçalho, e se estiver incorreto é de imediato enviada uma trama REJ, caso esteja correto, procede-se com a verificação de erros. É feito de seguida a operação de *destuffing* de dados e com isso, verificado o BCC2 (a integridade dos dados) , caso o BCC2 original – que está contido na própria mensagem – for igual ao BCC2 calculado com os dados obtidos depois do processo de *destuffing*, então os dados estão sem erros e é enviado um RR, caso os BCC2's não coincidam, é enviado um REJ.

É de realçar que aquando da transmissão do RR, o valor de nr é atualizado e a trama lida copiada para a trama a enviar para a camada de aplicação (sem os cabeçalhos). Caso seja um REJ, o valor de nr não é atualizado. A função retorna o tamanho da trama enviada sem os cabeçalhos impostos pela camada de ligação de dados (tamanho – 6 = tamanho do pacote).

Camada de Aplicação

sender()

É esta a principal função responsável pelo procedimento que compete ao computador que transmite fazer. Antes desta função ser chamada na *main()*, é chamada a função *Al_setter()* que abre o ficheiro selecionado para transmissão e obtém o tamanho do mesmo através da função *fileLength()* (presente no ficheiro tools.c) – é uma função preparatória à função *sender()* em si. Uma vez chamada esta função, começa por converter o nome e tamanho do ficheiro em octetos através da chamada à função *tlv_setter()*. Esta função converte o nome e tamanho do ficheiro em octetos, de forma a serem guardados na estrutura tvl correspondentes. Isto é feito através do uso da função de sistema *snprintf()* e tem como utilidade transmitir os parâmetros de controlo dos pacotes de dados. De seguida, constrói-se o pacote de controlo START (que indica o início da transmissão do ficheiro) com a estrutura tlv editada na função *tvl_setter()* e envia-se através da envocação da função pertencente à

camada de ligação de dados, *llwrite()*. Segue-se então a repartição em "chuncks" de tamanho SIZE_DATAPACKAGE, do ficheiro a ser enviado. (isto é feito analisando a quantidade de bytes que restam ler do ficheiro e, caso necessário, adaptar o tamanho do último pacote de dados a ser enviado). Estando este processo feito, procede-se à construção dum pacote de dados com esses “chuncks” e, através da função *llwrite()* enviá-los. Quando o ficheiro acaba de ser enviado, esta função encarrega-se de construir o pacote de controlo do tipo END que sinaliza o fim da transmissão do ficheiro.

receiver()

Esta é a função responsável pelo comportamento do recetor do ficheiro. É ela que é responsável pela leitura dos dados da camada de ligação e pela sua escrita no ficheiro destino. É baseada numa espécie de máquina de estados: a função começa por ler (através da envocação da função *llread()*) os dados recebidos, verificando se se trata dum pacote de controlo do tipo START (recorrendo ao cabeçalho C), se o for, reconstrói-o e trata de reconverter de octetos para dados normais o contéudo deste pacote. Abre também o ficheiro destino onde os dados que virão a ser recebidos irão ser escritos. De seguida, lê novamente e verifica se se trata dum tipo de pacote de dados, se for, reconstrói-o chamando a função *rebuildDataPackage()* e escreve-o no ficheiro. O programa segue este procedimento até encontrar um pacote que tenha como campo de controlo a flag END, assim que recebe este, faz o mesmo processo que fez no caso do pacote de controlo do tipo START. A função termina após fechar o ficheiro aberto anteriormente através da função do sistema *close()*.

Tools

Nesta parte do relatório iremos explicar resumidamente a utilidade de cada função auxiliar e das estruturas usadas.

```
int setPort(char *port, struct termios *oldtio);
```

Esta função é responsável por “abrir” a porta série através duma chamada à função *open()* e de inicializar diversos parâmetros desta, como por exemplo, o *baudrate*.

```
int resetPort(int fd, struct termios *oldtio);
```

O seu papel é precisamente reverter o que foi feito na *setPort()* e, obviamente, chamar a função *close()* como argumento o fd associado à porta série.

```
void buildConnectionFrame( unsigned char *connectionFrame, unsigned char A, unsigned char C);
```

Função apenas usada para ler da porta série quando no contexto se trata duma trama referente à conexão ou término da ligação, isto é – se se tratar duma trama SET, UA ou DISC. É baseada na máquina de estados presente nos conteúdos da UC e garante que o que foi lido da porta tem exatamente a estrutura suposta.

```
int buildFrame( unsigned char * frame, int C_ns, unsigned char* message, int lenght);
```

Esta função recebe como argumento um pacote da camada de aplicação, ao qual chamamos message e constrói uma trama a partir dele. É feita a análise ao BCC1, ao BCC2 e também se procede ao stuffing dos dados.

```
unsigned char buildBCC2(unsigned char *message, int lenght)
```

Função chamada na *buildFrame()* e apenas trata do processo de cálculo do BCC2.

```
int stuffing (int length, unsigned char* buffer, unsigned char* frame, int frame_length, unsigned char BCC2);
```

Também chamada na *buildFrame()*. É responsável pelo stuffing dos dados e potencialmente do BCC2 (se este for igual ao carácter de escape ou à *flag).*

```
int destuffing(int length, unsigned char* buffer, unsigned char* frame);
```

Executa o processo contrário à função acima. É chamada na função *llread()*.

```
int buildDataPackage(unsigned char* buffer, unsigned char* package, int size, int seq_n);
```

Esta função trata da construção dum pacote da aplicação (segundo o protocolo) a partir dos dados do ficheiro que recebe em *unsigned char * buffer*.

```
void rebuildDataPackage(unsigned char* packet, DataPackage *packet_data);
```

Função que faz o oposto à anterior, recebe os dados da trama (exceto cabeçalhos) e reconstói o pacote. Isto é feito com o auxílio duma estrutura do tipo DataPackage. A cada posição do que recebemos da camada de ligação, atribuímos o tipo de dados que se tratam, sejam eles o N, o L2, o L1 ou o *file_data* em si.

```
int buildControlPackage(unsigned char C, unsigned char* package, ControlPackage *tlv);
```

Nesta função procede-se à construção de pacotes de controlo da camada de aplicação. É feito através da estrutura do tipo ControlPackage. Atribui-se primeiramente o respetivo C (START ou END), o T, o L e o V.

```
void rebuildControlPackage(unsigned char* package, ControlPackage *tlv);
```

Reconstrói um pacote do tipo de controlo. Recebe como argumento uma trama (sem cabeçalhos) da camada de ligação e relaciona, tal como a função rebuildDataPackage(), os dados aos respetivos campos.

```
int fileLenght(int fd);
```

Pequena função que retorna o tamanho dum ficheiro recorrendo à *standard library.*

```
int readFromPort(int fd, unsigned char* frame);
```

Função que escuta a porta e vai metendo o que lê (char a char) na *frame*, que é retornada por argumento. Dado que está à espera de receber tramas, começa por iniciar a leitura quando recebe uma FLAG e acaba apenas quando recebe novamente uma FLAG.

```
void printProgressBar(float current, float total);
```

Função auxiliar que iremos falar mais aprofundademente na parte deste relatório correspondente aos elementos de valorização.

```
typedef struct ControlPackage;
```

Estrutura que apenas ajuda a clarificar o tipo de dados presentes (sejam eles T, L ou V).

```
typedef struct DataPackage;
```

Como a anterior, ajuda a clarificar o tipo de dados com os quais estamos a lidar (N, L2, L1, data…).

```
typedef enum ConnectionState;
```

Enumeração de extrema utilidade na função buildConnectionFrame dado que podemos usar os nomes por nós definidos num *switch* e ter uma leitura e escrita do programa muito mais facilitada.

Caso de Usos Principais

A *main()* está presente na camada de aplicação (faz sentido pois é esta a camada mais “superior” e é daí que começamos o nosso programa). Esta função recebe como argumento a porta série a ser utlizada e trata de inicialiar e repor as configurações da mesma. Numa

fase inicial, trata de questionar o utilizador qual o papel que este pretende ter na ligação (*sender* ou *receiver*) e o nome do ficheiro a ser enviado/recebido. De seguida, chama a função *llopen()* que é chamada por ambos os intervenientes de forma a inicializar a ligação. Estando esta fase concluída, é chamada a função *sender()* ou *receiver()* caso se trate do computador que envia ou do computador que recebe o ficheiro. Ambas as funções retornam 0 em sucesso e -1 se algum erro ocorreu. Após retornarem, é chamada a função *llclose()* (também por ambos os intervenientes) de forma a fechar a ligação.

Validação

De forma a verificar que o nosso programa cumpria os objetivos propostos, aplicámos diversos testes com a orientação do professor responsável, que ao longo do decorrer do trabalho laboratorial nos ia alertando para eventuais casos que o nosso código devia estar preparado.

O primeiro teste óbvio é, então, enviar um ficheiro (no caso, “penguin.gif”) e verificar se o computador que está a receber, o recebe de facto e se recebe sem erros. O segundo teste ao qual submetemos o nosso programa, foi enviar um ficheiro e durante a transmissão, carregar no botão de transmissão, interrompendo momentanemente a ligação. Outra teste importante é induzir ruído (de forma a verificar a nossa correta implementação dos RR e REJ) através dum fio de ligação. Também se procedeu à combinação de todos os testes: enviar um ficheiro, interromper a ligação e induzir ruído.

Verificou-se que o nosso programa respondeu bem a todos os testes, verificando-se também, a correta implementação de *timers* e do *timeout*, induzindo esses casos. Procedeu-se também ao envio de outro tipo de ficheiros que não .gif e de diferentes tamanhos. Todos foram bem transmitidos, todos com tamanho igual ao ficheiro original correspondente.

Elementos de Valorização

Sentimos a necessidade de implementar uma utilidade extra-guião: uma barra de progresso (*void printProgressBar(float,float)*) que nos indica visualmente a percentagem de transferência do ficheiro que já foi concluída (aumentando a barra). Esta percentagem é determinada pelo número de caracteres já escritos no ficheiro destino em função do tamanho do ficheiro original. É chamada tanto pelo *sender* como pelo *receiver* e é imprimida no terminal, parecendo ao utilizador permanente uma vez que usamos a função *fflush()*. A função é apresentada de seguida:

```c
void printProgressBar(float current, float total) {
    int bar_length = 51;
    float percentage = 100.0 * current / total;

    printf("\rCompleted: %6.2f%% [", percentage);

    int i;
    int pos = percentage * bar_length / 100.0;

    for (i = 0; i < bar_length; i++){
        if(i <= pos)
            printf("=");
        else printf(" ");
    }
    printf("]");

    fflush(stdout);
}
```

Caracterização Estatística da Eficiência

Conclui-se que a eficiência do nosso protocolo ronda os 80% quando não é imposto nenhum erro .Ver anexos para informações sobre este tema.

Conclusão

Podemos dizer com bastante satisfação que cumprimos os objetivos que nos foram propostos. Conseguimos com aprovação concluir o programa até à data limite acordada com o professor. Devemos contudo, realçar o facto de que numa fase inicial não conseguiamos perceber o que fazer, como é que as camadas se interligavam, o que cada camada fazia, etc. Estas dúvidas foram sendo levantadas à medida que investigávamos mais sobre o assunto e à medida que íamos escrevendo o código (à medida que também o íamos testando e falhando...várias vezes): inerentemente sentíamos o que devíamos fazer. Concluímos que nos envolvemos bastante com o projeto, que ficámos a perceber mais como é que um protocolo funciona e que desenvolvemos as nossas *skills* em linguagem C.

Anexos

Caracterização Estatística da Eficiência

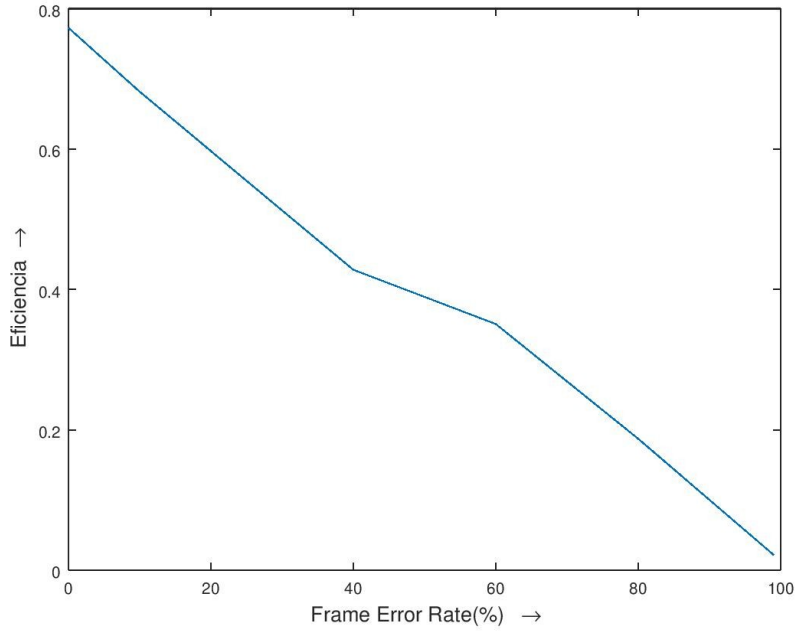
- **Variar Frame Error Rate**

Para conseguir estes resultado estatístico criou-se a função *errorGenerator()*, que é chamada dentro da llread() depois de ler a trama. Em seguida usando a função “rand() % 101” que gera um número aleatório entre 0 e 100, é comparado com o valor da FER de modo simular um momento probabilístico real, e se a condição se verificar é introduzido 0x00 em qualquer índice do buffer exceto se esse mesmo índice corresponder a flags de inicio/fim. A eficiência, neste caso, é calculada comparando o tempo ideal que demoraria a transmissão em comparação com o tempo real.

```
void errorGenerator(unsigned char *buffer, int size){

  int i=0, err=0;

  err = rand() % 101;

  if(err < FER){

    do {
      i = rand() % (size - 3) + 1;
    } while(buffer[i] == 0x7D || //to make sure we dont interfere with the framing (optimal cenario)
      buffer[i] == 0x7E ||
      buffer[i] == 0x5D ||
      buffer[i] == 0x5E);

  }

  buffer[i]=0x00; // error input right here
}
```

Para os valores de Baudrate = 38400, um tamanho da trama = 519 e para o ficheiro “penguin.gif” obtiveram-se os seguintes resultados:

Nºda medida	FER(%)	Ideal Time(s)	Time DataLink (s)	S
1	0	2.46208	3.1862	0.7727
2	10	2.46208	3.6119	0.6817
3	40	2.46208	5.7426	0.4280
4	60	2.46208	7.0196	0.3507
5	80	2.46208	13.1554	0.1872
6	99	2.46208	112.6094	0.0219

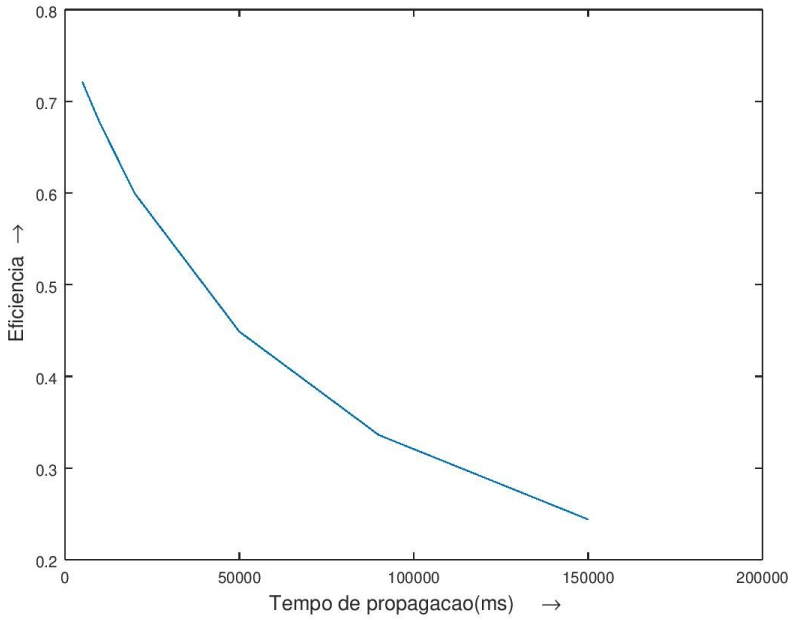


- **Variar o tempo de propagação**

Para conseguir estes resultado estatísticos foi inserida a função usleep() na llread() a seguir a esta função ler uma trama da porta. Esta função suspende a execução da thread onde é executada por um intervalo de tempo (microsegundos) enviado por argumento. Este procedimento simula o aumento do tempo de propagação: como se o cabo que liga os dois computadores fosse maior do que o que ele é na realidade.

Para os valores de Baudrate = 38400 um tamanho da trama = 519 e para o ficheiro “penguin.gif” obtiveram-se os seguintes resultados:

Nºda medida	**T_PROP_usleep(t)**	***Time DataLink (s)***	***S***
1	5000	3.4151	0.7209
2	10000	3.6482	0.6749
3	20000	4.1069	0.5995
4	50000	5.4857	0.4488
5	90000	7.3271	0.3360
6	150000	10.0861	0.2440

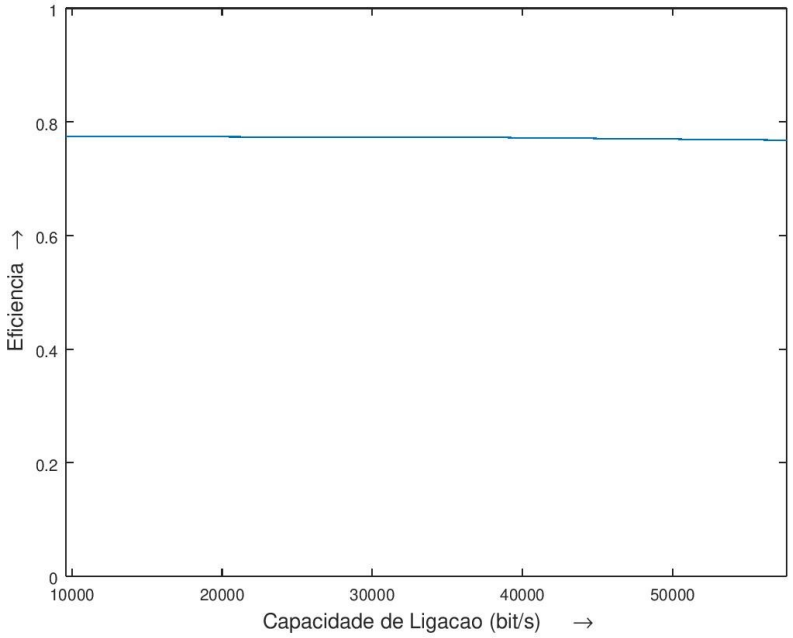


- **Variar C (capacidade de ligação)**

Para conseguir estes resultado estatísticos variou-se o valor da Baudrate (C), definido no ficheiro tools.h.

Para o valor de tamanho da trama = 519 e para o ficheiro “penguin.gif” obtiveram-se os seguintes resultados:

Nºda medida	Baudrate	Time DataLink (s)	S
1	9600	12.7146	0.7704
2	19200	6.3631	0.7738
3	38400	3.1872	0.7725
4	57600	1.0687	0.7679

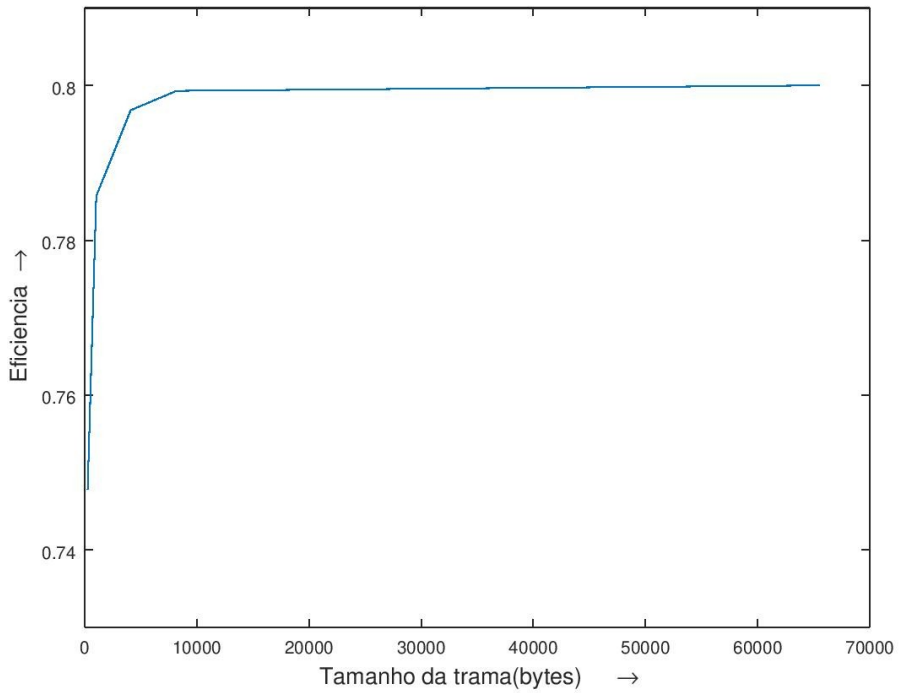


- **Variar o tamanho da frame**

Para conseguir estes resultados estatísticos variou-se o tamanho do package e consequentemente o tamanha da frame.

Para o valor de Baudrate = 38400, e para o ficheiro “penguin.gif” obtiveram-se os seguintes resultados:

Nºda medida	Size_Package	Size_Frame	Time DataLink (s)	S
1	128	263	3.3820	0.7478
2	512	1031	3.1437	0.7857
3	2048	4103	3.4590	0.7968
4	4096	8199	3.9742	0.7993
5	32768	65543	11.4216	0.800



Código

datalink.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/time.h>
#include <inttypes.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include "string.h"
#include "ApplicationLayer.h"
#include "tools.h"
#include "datalink.h"
#include "alarme.h"

int ns=0;
int nr=0;


int llopen(int fd, ConnectionMode mode){

  int connected = 0, state=0, res=0;
  unsigned char SET[5], UA[5];
  char * frame;

  //Building frames
  buildConnectionFrame(SET,A_S,C_SET);
  buildConnectionFrame(UA,A_S,C_UA);
  switch (mode){
    case SEND:
        while(connected==0){

          switch(state){ // like a state machine to know if it is
    sending SET or waiting for UA

            case 0: //SENDS SET
              tcflush(fd,TCIOFLUSH); // clears port to making sure
    we are only sending SET

              if((res = write(fd,SET,5)) <5){
                perror("write():");
                return -1;
              }
```

```
44                else printf("SET SENT\n");

46                state =1;
                  break;
48
                case 1: // GETTING UA
50
                  printf("WAITING FOR UA\n");
52                setAlarm(3);
                  frame = NULL;
54                while(frame == NULL){
                    frame= connectionStateMachine(fd);
56                  byteS_counter+=res; //calc eficiencia

58                  if(timeout){
                      if(n_timeout >= MaxTries){
60                      stopAlarm();
                        printf("Nothing received for 3 times\n");
62                      return -1;
                      }
64                    else{
                        printf("Nothing was received after 3 seconds\n"
    );
66                      printf("Going to try again!\n\n\n");
                        state=0; //tries to send again
68                      timeout=0;
                        break;
70                    }
                    }
72                }

74                if( frame != NULL && UA[2]==frame[2] ){
                  stopAlarm(); // something has been received by this
    point
76                  printf("Connection established!\n");
                    connected=1;
78                }
                  else state=0;
80                break;
            }
82      }
        break;
84
    case RECEIVE:
86    while(connected==0){
        switch(state){ // like a state machine to know if it is
      sending UA or waiting for SET
88
          case 0: //getting SET
90
                  printf("WAITING FOR SET\n");
92                frame= connectionStateMachine(fd);

94                  if(frame!= NULL && SET[2]==frame[2]){
                      state=1;
96                  }
                  break;
```

```
98
        case 1: // sending UA
100
              tcflush(fd,TCIOFLUSH); // clears port to making sure
    we are only sending UA
102
              if((res = write(fd,UA,5)) <5){
104             perror("write():");
                return -1;
106           }

108           byteS_counter+=res; //calc eficiencia
              printf("Connection Established!\n");
110           connected=1;
              break;
112       }
        }
114   break;
  }
116 return 0;
}
118
int llclose(int fd, ConnectionMode mode){
120
  int connected = 0, state=0, res=0, n_timeout=0;
122 unsigned char DISC[5], UA[5];
  char * frame;
124
  //Building frames
126 buildConnectionFrame(UA,A_S,C_UA);
  buildConnectionFrame(DISC, A_S, C_DISC);
128
  switch (mode){
130   case SEND:
        while(connected==0){
132
          switch(state){ // like a state machine to know if it is
    sending DISC (or UA) or waiting for DISC
134
            case 0: //SENDS DISC
136               tcflush(fd,TCIOFLUSH); // clears port to making
    sure we are only sending SET

138               if((res = write(fd,DISC,5)) <5){
                    perror("write():");
140                 return -1;
                  }
142               else printf("\nDISC SENT\n");

144               state =1;
                  break;
146         case 1: // GETTING DISC

148           printf("\nWAITING FOR DISC\n");
              setAlarm(3);
150           frame = NULL;
              while (frame == NULL){
```

```
152                 frame = connectionStateMachine(fd);

154                 if( timeout ){
                      n_timeout++;
156                   if(n_timeout >= MaxTries){
                        stopAlarm();
158                     printf("Nothing received for 3 times\n");
                        return -1;
160                   }
                      else{
162                     printf("\nWAITING FOR DISC: Nothing was
    received for 3 seconds\n");
                        printf("Going to try again!\n\n\n");
164                     state=0;
                        timeout=0;
166                     break;
                      }
168                 }
                  }
170               stopAlarm();

172               if(frame!= NULL && DISC[2]==frame[2] ){ //GOT DISC
                      state=2;
174               }
                  else state=0;
176               break;

178             case 2:
                  tcflush(fd, TCIOFLUSH);  //clear port
180

182               if((res = write(fd, UA, 5)) < 5) {  //0 ou 5?
                    perror("write()");
184                 return -1;
                  }
186               byteS_counter+=res; //calc eficiencia

188               printf("\nConnection terminated.\n");
                  connected = 1;
190               break;
            }
192       }
    break;
194
  case RECEIVE:
196     while(connected==0){

198       switch(state){ // like a state machine to know if it is
    sending DISC

200         case 0: //getting DISC

202           printf("\nWAITING FOR DISC\n");
              frame= connectionStateMachine(fd);
204
                if(frame!=NULL && DISC[2]==frame[2]){
206               state=1;
```

```
              }
208             break;

210           case 1: // sending DISC back

212             tcflush(fd,TCIOFLUSH); // clears port to making sure we
      are only sending UA

214             if((res = write(fd,DISC,5)) <5){
                  perror("write():");
216               return -1;
                }
218             //calc eficiencia

220             else{
                  byteS_counter+=res;
222               state=2;
                }
224             break;

226           case 2: //waiting for UA

228             printf("WAITING FOR UA\n");
                frame= connectionStateMachine(fd);
230
                  if(UA[2]==frame[2]){
232                 printf("\nConnection Terminated!\n");
                    connected=1;
234               }
                break;
236       }
        }
238   break;
    }
240 return 0;

242 }

244 int llwrite(int fd, unsigned char* buffer,int length ){

246   int transfering=1, res=0, frame_size=0, done=0;
    unsigned char frame_to_send[SIZE_FRAME], frame_to_receive[
      SIZE_FRAME];
248 unsigned char RR[5], REJ[5];

250 //BUILD RR and REJ for comparison
    if(ns==0){
252   buildConnectionFrame(RR,A_S,C_RR1);
      buildConnectionFrame(REJ,A_S,C_REJ0);
254 }
    else if (ns==1){
256   buildConnectionFrame(RR,A_S,C_RR0);
      buildConnectionFrame(REJ,A_S,C_REJ1);
258 }
    tcflush(fd, TCIOFLUSH);
260 frame_size= buildFrame(frame_to_send, ns, buffer, length);
```

```
262   while (transfering)
      {
264     //TIMEOUT CAUSION
        res = write(fd, frame_to_send, frame_size);
266     byteS_counter+=res;
        setAlarm(3);
268     done=0;
        while(!done) {
270       done = readFromPort(fd, frame_to_receive);
          if(timeout){
272         if(n_timeout >=MaxTries){
              stopAlarm();
274           printf("Nothing received for 3 times\n");
              return -1;
276         }
            else{
278           printf("WAITING FOR WRITE ACKOLEGMENT: Nothing was
      received after 3 seconds\n");
              printf("Going to try again!\n\n\n");
280           timeout=0;
              //done=0;
282           break;
            }
284       }
        }
286
        if( memcmp(RR, frame_to_receive, 5) == 0 ){ //CHECK TO SEE IF RR
288
          stopAlarm(); //something has been received by this point
290       ns = 1 -ns;
          transfering=0;
292     }
        if(memcmp(REJ, frame_to_receive, 5)==0 ){ //REJ CASE
294       continue;
        }
296   }
      return res;
298 }

300 int llread(int fd, unsigned char* frame_to_AL ){

302   int done=0, state=0, res=0, i=0, j=0, discard=0;
      int destuffed_data_size = 0;
304   unsigned char frame_from_port[SIZE_FRAME];
      unsigned char data_frame_destuffed[SIZE_FRAME];
306   unsigned char RR[5], REJ[5];
      unsigned char BCC2 = 0x00;
308   unsigned char BCC2aux = 0x00;


310
      while(!done){
312
        switch(state){
314
          case 0://reads from port
316
            res=readFromPort(fd,frame_from_port);
```

```
318
        if(res==-1 || res== -2){
320
          return -1;
322       }

324       errorGenerator(frame_from_port, res);

326       byteS_counter+=res; //estatistica

328       usleep(TPROP); //usleep para simular t_prop

330       state=2;
          break;
332
        case 1:
334
          if(frame_from_port[2]== C_NS0 && (nr==1)){
336
            discard =1;
338         nr=0;
            state=5;
340
          }
342       else if(frame_from_port[2] == C_NS1 && (nr==0)){

344         discard =1;
            nr=1;
346         state=5;

348       }

350       break;

352     case 2: //check BCC1

354       if((frame_from_port[1]^frame_from_port[2])!=frame_from_port
      [3]){ //wrong BCC1

356         state=6;
          }
358       else state =3;
          break;
360
        case 3://DESTUFFING
362       destuffed_data_size = destuffing(res-1, frame_from_port,
      data_frame_destuffed);
          state=4;
364       break;
        case 4:  //check BCC2
366
          BCC2=data_frame_destuffed[destuffed_data_size-1];
368       BCC2aux=data_frame_destuffed[0];

370       for(int k=1; k<destuffed_data_size-1; k++){
            BCC2aux= BCC2aux ^ data_frame_destuffed[k];
372       }
```

```
374         if(BCC2!=BCC2aux){
              state=6;
376           break;
            }
378         else state=5;
            break;
380       case 5:

382           if(frame_from_port[2]== C_NS0 && nr==0){

384             nr=1; // update nr
                buildConnectionFrame(RR,A_S,C_RR1);
386           }
              else if(frame_from_port[2]== C_NS1 && nr==1){
388
                nr=0;// update nr
390             buildConnectionFrame(RR, A_S,C_RR0);
              }
392           if(discard==0){

394             //stuff well read, then send it to AppLayer
                for (i = 0, j = 0; i < destuffed_data_size-1; i++, j++)
     {
396               frame_to_AL[j] = data_frame_destuffed[i];
                }
398           }
              //sends RR
400           tcflush(fd,TCIOFLUSH);

402           if( write(fd, RR, 5) < 5){
                perror(" Write() RR:");
404             return -1;
              }
406
              done=1;
408           break;
      case 6: //REJ case
410
          if(frame_from_port[2]== C_NS0 && nr==0){// frame 0, rej0
412           buildConnectionFrame(REJ, A_S,C_REJ0);
            }
414         else if(frame_from_port[2]== C_NS1 && nr==1){// frame 1,
     rej1
              buildConnectionFrame(REJ, A_S,C_REJ1);
416         }

418         tcflush(fd, TCIOFLUSH);
            if( write( fd, REJ, 5)< 5){
420
              perror("Write () REJ:");
422           return -1;
            }
424         state=0; // trying again
            break;
426     }
    }
```

```
428     return res-6;
}
```

datalink.c

datalink.h

```
#ifndef DATALINK
#define DATALINK
#include "ApplicationLayer.h"

char* connectionStateMachine(int fd);
int llopen(int fd, ConnectionMode mode);
int llwrite(int fd, unsigned char* buffer,int length );
int llread(int fd, unsigned char* frame_to_AL );
int llclose(int fd, ConnectionMode mode);

#endif
```

datalink.h

ApplicationLayer.c

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

#include "string.h"
#include "ApplicationLayer.h"
#include "tools.h"
#include "datalink.h"
#include "sender.h"
#include "receiver.h"

ApplicationLayer Al;
ApplicationLayer Alr;

int main(int argc, char** argv){

    timeDatalink=0;
    struct timespec start, stop;
    int fd=0, res=0;
    struct termios oldtio;
    int done = 0;
```

```
    char file[90];
32  if ( (argc < 2) ||
         ((strcmp("/dev/ttyS0", argv[1])!=0) &&
34        (strcmp("/dev/ttyS4", argv[1])!=0) )) {
      printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\
    n");
36    exit(1);
    }
38
  fd= setPort(argv[1], &oldtio);
40  if(fd<0){ // set port configs
    perror("setPort():");
42  exit(1);
  }
44
  //*****************************************
46  printf("Select Connection mode\n");
  printf("1: SEND    2:RECEIVE\n");
48  ConnectionMode mode;
  int i;
50  scanf("%d",&i);
  mode =i-1;
52
  switch (mode)
54  {
    case SEND:
56    printf("What's the name of the file you wanna transfer?\n");
      while (!done) {
58      printf("\nFILENAME: ");

60        if (scanf("%s", file) == 1){
                done = 1;}
62        else
            printf("Invalid input. Try again:\n");
64
          (Al.file_name) = file;
66
      }
68    break;

70  case RECEIVE:
      printf("How do you wanna name the incoming file?\n");
72
        while (!done) {
74      printf("\nFILENAME: ");

76      if (scanf("%s", file) == 1){
            done = 1;}
78      else
            printf("Invalid input. Try again:\n");
80     }
        (Alr.file_name) = file;
82
      break;
84  }

86  if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {
```

```
    perror("clock_gettime()");
88  return -1;
}
90
if((res=llopen(fd, mode))==-1){
92  printf("llopen not working \n");
    printf("Connection not possible, check cable and try again.\n")
    ;
94  return 1;
}
96
if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {
98  perror("clock_gettime()");
    return -1;
100 }


102
switch (mode)
104 {
    case SEND:
106
      if (sender(fd)<0 ){
108    return -1;
      }
110   close(Al.fd);
      break;
112
    case RECEIVE:
114
        if(clock_gettime(CLOCK_MONOTONIC, &start) < 0) {
116     perror("clock_gettime()");
        return -1;
118    }
        if(receiver(fd, Alr)<0){
120       return -1;
        }
122     if(clock_gettime(CLOCK_MONOTONIC, &stop) < 0) {
          perror("clock_gettime()");
124       return -1;
        }
126
        printf("\n\tFER: %d %%\n", FER);
128     printf("\n\tAPI runtime:\t%" PRId64 "ns", transform(&stop)
  - transform(&start));
        printf("\n\tTime spent in Data-Link Layer:\t%" PRId64 "ns\n
  ", timeDatalink);
130     printf("\tTime spent in AppLayer:\t%" PRId64 "ns\n\n", (
  transform(&stop) - transform(&start)) - timeDatalink);
        printf("\tTotal bytes  = %d\n", byteS_counter);
132
      break;
134 }

136 if((res=llclose(fd, mode))==-1){
    printf("llclose not working \n");
138 }
```

```
140   if(resetPort(fd,&oldtio)<0){
        perror("resetPort():");
142     exit(-1);
      }
144
      return 0;
146 }
```

ApplicationLayer.c

ApplicationLayer.h

```
#pragma once //it only needs to be compiled once
2
#include <stdio.h>
4
typedef enum {
6   SEND, RECEIVE
} ConnectionMode;
8
typedef struct
10 {
      //file descriptor
12    int fd;
      // Type of connection (Sender or Receiver)
14    ConnectionMode mode;
      //file to be transfered
16    char *file_name;
      int file_size;
18 }ApplicationLayer;

20 typedef enum {
      PARAM_FILE_SIZE, PARAM_FILE_NAME
22 } T_type;

24 extern ApplicationLayer Al;
```

ApplicationLayer.h

sender.c

```

2 #include <sys/types.h>
#include <sys/stat.h>
4 #include <unistd.h>
#include <fcntl.h>
6 #include <stdlib.h>
#include <inttypes.h>
8 #include <inttypes.h>
#include <time.h>
10 #include <sys/time.h>
#include <stdio.h>
12 #include "string.h"
```

```c
#include "ApplicationLayer.h"
#include "tools.h"
#include "datalink.h"


//sets Al struct with paramenters
int Al_setter(){
    int file_fd = open(Al.file_name, O_RDONLY);
    if(file_fd < 0){
        perror("open():");
        return -1;
    }
    Al.fd=file_fd;
    (Al.file_size)=fileLenght(Al.fd);
    printf("FILE LENGTH : %d  \n\n", Al.file_size);
    return 0;
}
void tlv_setter(ControlPackage *tlv){
    //convert file_size to oct
    char fileSizeBuf[10];
    snprintf(fileSizeBuf, sizeof fileSizeBuf, "%d", Al.file_size);

    tlv[0].T=PARAM_FILE_SIZE;
    int i=(Al.file_size), count=0;
    while(i != NULL){
        i/=10;
        count++;
    }
    snprintf(&(tlv[0].L), count, "%d", count);

   // tlv[0].L=sizeof(Al.file_size);
    tlv[0].V = (unsigned char *) malloc(sizeof(fileSizeBuf));
    for(int i=0;i < count; i++){
        tlv[0].V[i]=fileSizeBuf[i];
    }
    tlv[1].T=PARAM_FILE_NAME;
    tlv[1].L= strlen(Al.file_name);
    tlv[1].V = (unsigned char *) malloc(sizeof(tlv[1].L));
    for(int i = 0; i<strlen(Al.file_name); i++){
        tlv[1].V[i]=Al.file_name[i];
    }
}

int sender(int fd){

  fer_counter = 0;


     if(Al_setter()<0){

            printf("error setting Al\n");
            return -1;
          }

    //convert file_size to oct
    char fileSizeBuf[10];
    snprintf(fileSizeBuf, sizeof fileSizeBuf, "%d", Al.file_size);
```

```
70
    ControlPackage tlv_start[2];
72  tlv_setter(tlv_start);

74  unsigned char Start_Controlpackage[10];
    int sizeControlPackage=buildControlPackage(AP_START,
    Start_Controlpackage, tlv_start);
76
    llwrite(fd, Start_Controlpackage,sizeControlPackage);
78
    unsigned char fileBuf[SIZE_DATAPACKAGE-4]; //-4 cause of the
    data headers
80  unsigned int bytesread=0;
    unsigned char DataPackage[SIZE_DATAPACKAGE];
82  int i=0;

84  int readsize, bytesleft, data_size, total_bytesread=0,
    byteswritten;
    while( total_bytesread < Al.file_size ) {
86      bytesleft=Al.file_size-total_bytesread;
        if(bytesleft > SIZE_DATAPACKAGE -4) {
88          readsize = SIZE_DATAPACKAGE-4;
        } else {
90          readsize =bytesleft;
        }
92      total_bytesread += readsize;
        if((bytesread = read(Al.fd, fileBuf, readsize)) < 0) {
94          printf("Error reading from file\n");
            return -1;
96      }

98      if((data_size = buildDataPackage(fileBuf, DataPackage,
    bytesread, (i++)%255)) < 0) {
            perror("buildDataPackage");
100         return -1;
        }
102

104     if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {   //
    CLOCKING
          perror("clock_gettime()");                        //
    CLOCKING
106       return -1;                                        //
    CLOCKING
        }
108     byteswritten = llwrite(fd, DataPackage,data_size);
        if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {   //
    CLOCKING
110       perror("clock_gettime()");                        //
    CLOCKING
          return -1;                                        //
    CLOCKING
112     }
        timeDatalink+= transform(&finito) - transform(&inito);  //
    CLOCKING
114
```

```
116         if(byteswritten < 0) {
                printf("Connection LOST, check cable and try again\n");
118             return -1;
            }
120
            memset(fileBuf, 0, SIZE_DATAPACKAGE);
122
            if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {    //
    CLOCKING
124           perror("clock_gettime()");                          //
    CLOCKING
              return -1;                                          //
    CLOCKING
126         }
            printProgressBar(total_bytesread, Al.file_size);
128         if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {    //
    CLOCKING
              perror("clock_gettime()");                          //
    CLOCKING
130           return -1;                                          //
    CLOCKING
            }
132


134         timeDatalink+= transform(&finito) - transform(&inito);  //
    CLOCKING

136         //Projeto de um indicador de BitRate
            /*
138         bitRateTimer=transform(&finito) - transform(&inito);
            printf("%" PRId64 "bytes/s\n", (uint64_t)total_bytesread/
    bitRateTimer);
140         */


142


144     }

146     ControlPackage tlv_end[2];
        tlv_setter(tlv_end);
148
        unsigned char End_Controlpackage[SIZE_DATAPACKAGE-4];
150     buildControlPackage(AP_END, End_Controlpackage, tlv_end);

152     if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {    //CLOCKING
          perror("clock_gettime()");                          //CLOCKING
154       return -1;                                          //CLOCKING
        }
156     llwrite(fd, End_Controlpackage, SIZE_DATAPACKAGE);
        if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {    //CLOCKING
158       perror("clock_gettime()");                          //CLOCKING
          return -1;                                          //CLOCKING
160     }

162     timeDatalink+= transform(&finito) - transform(&inito);  //
    CLOCKING
```

```
164
166
  return 1;
168 }
```

sender.c

sender.h

```
#include <stdio.h>
2
#include "ApplicationLayer.h"
4 #include "tools.h"
//sets Al struct with paramenters
6 int Al_setter();

8 //sets tlv parameters
void tlv_setter( ControlPackage *tlv);
10
int sender(int fd);
```

sender.h

receiver.c

```
#include <sys/types.h>
2 #include <sys/stat.h>
#include <unistd.h>
4 #include <time.h>

6 #include <inttypes.h>
#include <fcntl.h>
8 #include <sys/time.h>
#include <inttypes.h>
10 #include <stdlib.h>
#include <stdio.h>
12 #include "string.h"
#include "ApplicationLayer.h"
14 #include "tools.h"
#include "datalink.h"
16

18 //returns 0 in succes, -1 if error
int receiver(int fd, ApplicationLayer Alr){
20
  int res=0, done=0, state=0, name_size=0, output_file=0, its_data
    =0, c_value=0;
22  int bytes_written=0, total_bytes_written=0;
  unsigned char data_from_llread[SIZE_DATAPACKAGE];
24  unsigned char package[SIZE_DATAPACKAGE-1];
  ControlPackage start[TLV_N], end[TLV_N];
26  DataPackage data;
```

```
28    if(res<0){
        perror("llopen(receive):");
30      return -1;
      }
32
       while(!done){
34      switch(state){

36        case 0: //reading start packages and full Al struct

38          if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {
              perror("clock_gettime()");
40            return -1;
            }
42
            res=llread(fd,data_from_llread);
44
            if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {
46              perror("clock_gettime()");
                return -1;
48            }


50
            timeDatalink+=transform(&finito) - transform(&inito);
52


54          if(res<0){
              perror("llread()");
56            return -1;
            }
58
            c_value=data_from_llread[0];
60
            for(int i=0; i<res-1;i++){ //[i+1] so it doesnt send the C
      -> it's not necessary at this point and we don't count it in
       our functions from tools
62            package[i]=data_from_llread[i+1];
            }
64
            if(c_value==AP_START){
66
              rebuildControlPackage(package,start);
68
              for(int i=0; i<TLV_N; i++){
70
                if(start[i].T==PARAM_FILE_SIZE){
72                Al.file_size=atoi(&start[i].V[0]);
                }
74              if(start[i].T==PARAM_FILE_NAME){
                  name_size=(int)start[i].L;
76                Al.file_name=(char*)malloc(name_size);
                  strcpy(Al.file_name, (char*)start[i].V);
78              }
              }
80            printf("TOTAL FILE SIZE: %d\n", Al.file_size);
            }
```

```
82
        else if(c_value==AP_DATA){
84
          its_data=2;
86        state=1;
          break;
88      }
        else {
90        printf(" wrong c %d\n", c_value);
          return -1;
92      }

94      output_file=open(Alr.file_name, O_CREAT | O_APPEND |
    O_WRONLY, S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP |S_IWGRP |
    S_IXGRP | S_IROTH | S_IXOTH);
        c_value=0;
96      break;

98    case 1:

100   if(its_data==0){ // we do this if so it wont read again if we
     know already from case 1 that it is data

102     if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {
            perror("clock_gettime()");
104         return -1;
        }
106
        res=llread(fd, data_from_llread);
108
        if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {
110         perror("clock_gettime()");
            return -1;
112       }
          timeDatalink+= transform(&finito) - transform(&inito);
114     if(res<0){
          perror("llread()");
116     }

118     c_value=data_from_llread[0];
        its_data=1;
120   }

122   if(its_data==1){

124       for(int i=0; i<res-1;i++){ //[i+1] so it doesnt send the
    C -> it's not necessary at this point and we don't count it in
    our functions from tools
          package[i]=data_from_llread[i+1];
126     }
      }
128
      if(its_data==2 || its_data==1){
130     if(c_value==AP_DATA){
          rebuildDataPackage(package,&data);
132     }
```

```
134           if(c_value==AP_END){
                state=2;
136             break;
              }
138         }

140         bytes_written=write(output_file, data.file_data, 256*(int)
        data.L2+(int)data.L1);
            if(bytes_written<0){
142           perror("write() to output file:");
              return -1;
144         }
            total_bytes_written+=bytes_written;
146
            if(clock_gettime(CLOCK_MONOTONIC, &inito) < 0) {
148           perror("clock_gettime()");
              return -1;
150         }
            printProgressBar(total_bytes_written, Al.file_size);
152
            if(clock_gettime(CLOCK_MONOTONIC, &finito) < 0) {
154           perror("clock_gettime()");
              return -1;
156         }

158       timeDatalink+= transform(&finito) - transform(&inito);


160
            memset(package, 0, SIZE_DATAPACKAGE); //because we are
        reusing it to read various (depends on the file)
        data_from_llread
162         its_data=0; // so it can read more
            c_value=0;
164
            break;
166       case 2:

168         for(int i=0; i<res-1;i++){ //[i+1] so it doesnt send the C ->
         it's not necessary at this point and we don't count it in our
        functions from tools
              package[i]=data_from_llread[i+1];
170         }

172         rebuildControlPackage(package,end);

174         for(int i=0; i<TLV_N; i++){

176           if(end[i].T==PARAM_FILE_SIZE){
                Al.file_size= atoi(&end[i].V[0]);
178           }

180           if(end[i].T==PARAM_FILE_NAME){
                name_size=(int)end[i].L;
182             Al.file_name=(char*)malloc(name_size);
                strcpy(Al.file_name, (char*)end[i].V);
184           }
            }
```

```
186         done=1;
            break;
188       }
    }
190
    if(close(output_file)<0){
192
      perror("close():");
194     return -1;
    }
196
    return 0;
198 }
```

receiver.c

receiver.h

```
#include <stdio.h>
2
#include "ApplicationLayer.h"
4 #include "tools.h"

6 int receiver(int fd, ApplicationLayer Alr);
```

receiver.h

tools.c

```
#include <sys/types.h>
2 #include <sys/stat.h>
#include <fcntl.h>
4 #include <termios.h>
#include <stdio.h>
6 #include <stdlib.h>
#include <unistd.h>
8 #include <string.h>
#include <sys/time.h>
10 #include <time.h>
#include <errno.h>
12 #include <math.h>
#include <time.h>
14 #include <inttypes.h>
#include <signal.h>
16 #include <inttypes.h>

18 #include "tools.h"

20 uint64_t transform(struct timespec* aux) {

22     return aux->tv_sec * (uint64_t)1000000000L + aux->tv_nsec;
}
24
```

```
void errorGenerator(unsigned char *buffer, int size){

  int i=0, err=0;

  err = rand() % 101;
  if(err < FER){

    do {
      i = rand() % (size - 3) + 1;
    } while(buffer[i] == 0x7D || //to make sure we dont mess up the
     frames (optimal cenario)
      buffer[i] == 0x7E ||
      buffer[i] == 0x5D ||
      buffer[i] == 0x5E);
  }

  buffer[i]=0x00; // error input right here
}

//returns -1 in error
int setPort(char *port, struct termios *oldtio){

    if((strcmp("/dev/ttyS0", port) != 0) && (strcmp("/dev/ttyS4",
    port) != 0)) {
    perror("setPort(): wrong argument for port");
    return -1;
  }
  /*
    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.
  */
  struct termios newtio;

  int fd;
  if ((fd = open(port, O_RDWR | O_NOCTTY )) < 0) {
    perror(port);
    return -1;
  }

  if ( tcgetattr(fd, oldtio) == -1) { /* save current port settings
     */
    perror("tcgetattr");
    return -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]    = 1;   /* inter-character timer unused (
    estava a 0)*/
```

```
76    newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars
        received (estava a 5)*/

78    tcflush(fd, TCIOFLUSH);

80    if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
        perror("tcsetattr");
82      return -1;
      }
84
      printf("\nNew termios structure set\n");
86
      return fd;
88  }

90  //returns -1 in error
    int resetPort(int fd, struct termios *oldtio) {
92
      if ( tcsetattr(fd, TCSANOW, oldtio) == -1) {  //volta a por a
        configuracao original
94      perror("tcsetattr");
        return -1;
96    }
      close(fd);
98
      return 0;
100 }

102 void buildConnectionFrame( unsigned char *connectionFrame, unsigned
        char A, unsigned char C){ // belongs to DATALINK

104   connectionFrame[0] = FLAG;
      connectionFrame[1] = A;
106   connectionFrame[2] = C;
      connectionFrame[3] = connectionFrame[1]^connectionFrame[2];
108   connectionFrame[4] = FLAG;
    } //supervisionFrame()
110
    int buildFrame( unsigned char * frame, int C_ns, unsigned char*
        message, int lenght){ //belongs to DATALINK
112
      int l=0;
114   unsigned char BCC2;
      BCC2=buildBCC2(message, lenght);
116
      frame[l++]=FLAG;
118   frame[l++]=A_S;

120   if(C_ns){
        frame[l++]= C_NS1;
122   }
      else frame[l++]=C_NS0;
124
      frame[l++]= frame[1]^frame[2]; // BBC1
126
      l=stuffing(lenght,message, frame, l, BCC2);
128
```

```
  frame[1]=FLAG;

  return 1+1; //returns lenght of frame (counts the 0 position)
}

unsigned char buildBCC2(unsigned char *message, int lenght){ //
    belongs to datalink

  unsigned char BCC2=0;

  for(int i=0; i< lenght; i++){
    BCC2 ^=message[i];
  }
  return BCC2;
}

//builds data package from file
int buildDataPackage(unsigned char* buffer, unsigned char* package,
     int size, int seq_n){

  int i=0, aux=0;
  package[0]= AP_DATA; //C
  package[1]=(char)(seq_n);

  aux=size%256;
  package[2]=(size -aux)/256;
  package[3]=aux;

  for(i=0; i<size; i++){
    package[i+4]=buffer[i]; // data read from file into application
     package
  }

  return i+4; // returns size of package
}

//rebuild packet from datalink into DataPackage specific struct
void rebuildDataPackage(unsigned char* packet, DataPackage *
    packet_data){

  int i=0, j=0;
  int size_of_data=0;

  (*packet_data).N = packet[0];
  (*packet_data).L2= packet[1];
  (*packet_data).L1= packet[2];
  (*packet_data).file_data=(unsigned char*)malloc(256*(int)packet
    [1]+(int)packet[2]); //as shown in "guiao-PDF"

  size_of_data= 256*(int)(*packet_data).L2+(int)(*packet_data).L1;

  for( i=3, j=0; j<size_of_data; i++, j++){

    (*packet_data).file_data[j]=packet[i]; //for each byte of data
    in packet, put in packet_data
  }
```

```
180 }

182 int buildControlPackage(unsigned char C, unsigned char* package,
        ControlPackage *tlv){

184   int l=0, size=0;
      package[l++]=C; //control
186
      for(int i=0; i<TLV_N ; i++){
188     package[l++]= tlv[i].T;
        package[l++]= tlv[i].L;
190
        size=(int)tlv[i].L;
192
        for(int j=0; j< size && tlv[i].V!=NULL ; j++){
194       package[l++] = tlv[i].V[j];
        }
196   }
      return l; // returns lenght of control package created
198 }

200 void rebuildControlPackage(unsigned char* package, ControlPackage *
        tlv){

202   int i=0, size_v=0;
      for( int z=0; z< TLV_N; z++){
204
        tlv[z].T = package[i];
206     i++;
        tlv[z].L= package[i];
208
        size_v=(int)(tlv[z].L);
210     tlv[z].V= (unsigned char*)malloc(size_v);

212     for(int j=0; j< size_v; j++){
          i++;
214       tlv[z].V[j]= package[i];
        }
216     i++;
      }
218 }

220 int fileLenght(int fd){

222   int lenght=0;

224   if((lenght= lseek(fd,0,SEEK_END))<0){

226     perror("lseek():");
        return -1;
228   }

230   if(lseek(fd,0, SEEK_SET)<0){

232     perror("lseek()");
        return -1;
234   }
```

```
  return lenght;
}

char* connectionStateMachine(int fd){

  connectionState currentState = START_CONNECTION;
  char c;
  static char message[5];
  int done = 0, i = 0;

  while (!done){

    if (currentState == STOP_CON){
      done = 1;
    }
    else if (read(fd, &c, 1)== 0){
      return NULL;
    }

    switch(currentState){

      case START_CONNECTION:

        if(c == FLAG){
          message[i++] = c;
          currentState = FLAG_RCV;
        }
        break;

      case FLAG_RCV:
        if (c == A_R || c == A_S){
          message[i++] = c;
          currentState = A_RCV;
        }
        else if(c!=FLAG){
          i = 0;
          currentState = START_CONNECTION;
        }
        break;

      case A_RCV:

        if (c == C_SET || c== C_UA || c==C_DISC){
          message[i++] = c;
          currentState = C_RCV;
        }
        else if(c == FLAG){
          i = 1;
          currentState = FLAG;
        }
        else{
          i = 0;
          currentState = START_CONNECTION;
        }
        break;
      case C_RCV:
```

```
292         if (c == (A_S^C_SET) || c== (A_S^C_UA) || c==(A_S^C_DISC)
    || c== (A_R^C_SET) || c== (A_R^C_UA) || c==(A_R^C_DISC)) {
              message[i++] = c;
294           currentState = BCC_OK;
            }
296         else if(c == FLAG){
              i = 1;
298           currentState = FLAG_RCV;
            }
300         else{
              i = 0;
302           currentState = START_CONNECTION;
            }
304         break;
          case BCC_OK:
306
            if (c == FLAG){
308           message[i++] = c;
              currentState = STOP_CON;
310         }
            else {
312           i = 0;
              currentState = START_CONNECTION;
314         }
            break;
316
          case STOP_CON: {
318         message[i] = 0;
            done = 1;
320         break;
          }
322     }
  }
324 return message;
}
326

328 //returns lenght of frame read from port, -1 in error
int readFromPort(int fd, unsigned char* frame){
330
    unsigned char tmp;
332 int done=0, res=0, l=0;

334 memset(frame, 0, SIZE_FRAME);

336 while(!done){
    res=read(fd, &tmp,1);
338     if(res==-1){
          perror("read() from port = -1");
340       return -1;
        }
342 else if(res==0){
       return 0;
344 }
        else if(tmp== FLAG){ // evaluate if end or start point
346
            if(l==0){ //start point
```

```
348                     frame[l++]=tmp;
                    }
350                 else{ // somewhere else in the middle, starts again

352                     if(frame[l-1] == FLAG){
                memset(frame, 0, SIZE_FRAME);
354                         l=0;
                            frame[l++]=FLAG;
356                     }
                        else{ // in the end
358                         frame[l++]= tmp;
                            done=1;
360                     }
                    }
362             }
                else{
364                 if(l>0){ // put in frame what reads in the middle
                        frame[l++]=tmp;
366                 }
                }
368     }
        return l;
370 }

372 int stuffing (int length, unsigned char* buffer, unsigned char*
        frame, int frame_length, unsigned char BCC2){

374   for(int i=0; i<length; i++){
        if(buffer[i]== FLAG){//a flag is in the middle of the data
376
          frame[frame_length++]=ESC;
378       frame[frame_length++]=FLAG_PPP;
        }
380     else if(buffer[i]==ESC){

382       frame[frame_length++]=ESC;
          frame[frame_length++]=ESC_PPP;
384     }
        else frame[frame_length++]= buffer[i];
386   }

388   if(BCC2==FLAG){
        frame[frame_length++]=ESC;
390     frame[frame_length++]=FLAG_PPP;
    }
392   else if(BCC2==ESC){
        frame[frame_length++]= ESC;
394     frame[frame_length++]=ESC_PPP;
    }
396   else frame[frame_length++]=BCC2;
    return frame_length;
398 }

400 //HERE BUFFER = PRE-DESTUFFING AND FRAME = AFTER-DESTUFFING
    int destuffing(int length, unsigned char* buffer, unsigned char*
        frame){
402
```

```
  int frame_length=0;
  for(int i = 4; i< length; i++){
    if( buffer[i] == ESC) { //remove the next one
      if(buffer[i+1] == FLAG_PPP){
        frame[frame_length++] = FLAG;
      }
      else if(buffer[i+1] == ESC_PPP){ //remove the next one
        frame[frame_length++] = ESC;
      }
      i++;
    }
    else frame[frame_length++]=buffer[i];
  }
  return frame_length;
}

void printProgressBar(float current, float total) {
  int bar_length = 51;
  float percentage = 100.0 * current / total;

  printf("\rCompleted: %6.2f%% [", percentage);

  int i;
  int pos = percentage * bar_length / 100.0;

  for (i = 0; i < bar_length; i++){
    if(i <= pos)
      printf("=");
    else printf(" ");
  }
  printf("]");

  fflush(stdout);
}
```

tools.c

tools.h

```
#ifndef TOOLS
#define TOOLS

#include <inttypes.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define SIZE_DATAPACKAGE 256// size of data packages = 255*256+255
#define SIZE_FRAME (SIZE_DATAPACKAGE+1)*2+5
#define TLV_N 2// name of file, size of file
#define FER 0 //Frame Error Rate em percentagem
#define TPROP 0 //in microseconds

```

```
//DATALINK LEVEL
#define FLAG 0x7E //0111 1110
#define FLAG_PPP 0x5E
#define ESC 0x7D
#define ESC_PPP 0x5D
#define A_S 0x03 //0000 0011
#define A_R 0x01 //0000 0001
#define C_SET 0x03 //0000 0011
#define C_UA 0x07 //0000 0111
#define C_DISC 0x0B //00001011
#define C_RR0  0x05  //0000 0101
#define C_REJ0 0x01 //0000 0001
#define C_RR1  0x85  //1000 0101
#define C_REJ1 0x81  //1000 0001
#define C_NS0 0x00 //0000 0000
#define C_NS1 0x40 //0100 0000

//APPLICATION LEVEL
#define AP_START 0x02 //0000 0010
#define AP_DATA 0x01 //0000 0001
#define AP_END 0x03 //0000 0011

#define MaxTries 3

#define MICRO 1000L
#define MILI 1000000L

extern int timeout;
extern int n_timeout;

////////////////////////////////////////////////////
int fer_counter;          //counts REJs
uint64_t timeDatalink;
int byteS_counter;
//uint64_t bitRateTimer;  Ver sender() em printProgressBar call
struct timespec inito, finito;
////////////////////////////////////////////////////


typedef struct {
  //unsigned char C;
  unsigned char T;
  unsigned char L;
  unsigned char *V;
} ControlPackage;

typedef struct {

  //unsigned char C;
  unsigned char N;
  unsigned char L1;
  unsigned char L2;
  unsigned char *file_data;
} DataPackage;

typedef enum {
  START_CONNECTION, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP_CON
```

```
74 } connectionState;
76 void errorGenerator(unsigned char *buffer, int size);
78 uint64_t transform(struct timespec* aux);
80
   int setPort(char *port, struct termios *oldtio);
82 int resetPort(int fd, struct termios *oldtio);
84 void buildConnectionFrame( unsigned char *connectionFrame, unsigned
       char A, unsigned char C);
   int buildFrame( unsigned char * frame, int C_ns, unsigned char*
       message, int lenght);
86 unsigned char buildBCC2(unsigned char *message, int lenght);
88 int buildDataPackage(unsigned char* buffer, unsigned char* package,
       int size, int seq_n);
   void rebuildDataPackage(unsigned char* packet, DataPackage *
       packet_data);
90
   int buildControlPackage(unsigned char C, unsigned char* package,
       ControlPackage *tlv);
92 void rebuildControlPackage(unsigned char* package, ControlPackage *
       tlv);
94 int fileLenght(int fd);
   char* connectionStateMachine(int fd);
96
   int readFromPort(int fd, unsigned char* frame);
98
100 int stuffing (int length, unsigned char* buffer, unsigned char*
       frame, int frame_length, unsigned char BCC2);
    int destuffing (int length, unsigned char* buffer, unsigned char*
       frame);
102 void printProgressBar(float current, float total);
    #endif
```

tools.h

alarme.c

```
1  #include <unistd.h>
2  #include <signal.h>
   #include <stdio.h>
4
   #define TRUE 1
6  #define FALSE 0
8
   int timeout;
10 int n_timeout=0;
   void handler()                    // handler alarme
12 {
```

```
  printf("alarme # %d\n", n_timeout+1);
  timeout=TRUE;
  n_timeout++;
}

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

   sigaction(SIGALRM, &action, NULL);
   timeout=FALSE;
   n_timeout=0;
  alarm(0);
}

void setAlarm(int time){
   struct sigaction action;
   action.sa_handler= handler;
   sigemptyset(&action.sa_mask); //inicializes the signal set to
    empty
   action.sa_flags = 0;

   timeout=FALSE;
   sigaction(SIGALRM, &action, NULL);
   alarm(time);
}
```

alarme.c

alarme.h

```

extern int timeout;
extern int n_timeout;
void handler();

void setAlarm();

void stopAlarm();
```

alarme.h