U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

2018/2019

PROTOCOLO DE LIGAÇÃO DE DADOS

MIEIC *Turma05*

. Diogo Filipe da Silva Yaguas up201606165@fe.up.pt
. Gonçalo Nuno Botelho Amaral Rolão Bernardo up201606058@fe.up.pt
. Joana Sofia Mendes Ramos up201605017@fe.up.pt

Sumário

Este relatório contextualiza todo o trabalho realizado desde o início do semestre. Este consiste na transferência de ficheiros, através de uma porta de série, tendo sido implementado um conjunto de funções de leitura, escrita e tratamento de dados.

O projeto foi apresentado com sucesso, tendo sido realizada a transferência de dados sem qualquer perda ou erros. Quando perdida da ligação, o programa foi capaz de restabelecer a transmissão e recuperar os dados.

Introdução

O objetivo do trabalho é implementar um protocolo de ligação de dados que fornece um serviço de comunicação assíncrona de dados fiável entre dois sistemas ligados por um cabo série, testando o protocolo com uma aplicação simples de transferência de ficheiros, assegurando que apesar de haver interrupções e interferências, todos os dados são fornecidos corretamente. Todo o trabalho está de acordo com a informação fornecida no guião.

O relatório apresenta a seguinte estrutura:

1. **Arquitetura**: Funcionamento da aplicação e da interface do utilizador;
2. **Estrutura do código**: Demonstração das API's, estruturas de dados utilizadas, principais funções e a sua correlação com a arquitetura;
3. **Casos de uso principais**: Identificação de casos de usos principais e sequência de chamadas de funções;
4. **Protocolo de ligação lógica**: Identificação dos principais aspectos funcionais e descrição da estratégia de implementação;
5. **Protocolo de aplicação**: Identificação dos principais aspectos funcionais e descrição da estratégia de implementação;
6. **Validação**: Descrição dos testes efectuados com apresentação quantificada dos resultados;
7. **Eficiência do protocolo de ligação de dados**: Caracterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido;
8. **Conclusão**: Síntese da informação apresentada nas secções anteriores e reflexão sobre os objectivos de aprendizagem alcançados.

Arquitetura

A aplicação está organizada em duas camadas bem definidas. A **camada de ligação de dados** é a responsável pelo estabelecimento de ligação e, por essa razão, todas as suas funções asseguram a consistência de do protocolo, sendo a camada de mais baixo nível na aplicação. É feita nesta camada a interação com a porta de série, sendo feita a abertura, fecho, escrita e leitura, tratando das necessidades de comunicação, como tratamento de erros e *stuffing* e *destuffing* de pacotes, tendo sido desenvolvida nos ficheiros *linkLayer.c* e *linkLayer.h*.

A **camada da aplicação** é a responsável pelo envio e receção de ficheiros, sendo a camada lógica situada acima da camada de ligação de dados. Faz uso da interface da camada de ligação de dados, chamando as suas funções para o envio e receção de ficheiros a receber e enviar. Esta camada foi desenvolvida nos ficheiros *applicationLayer.c* e *applicationLayer.h*.

Na **Interface** do utilizador é permitido inserir o valor do *BaudRate*, bem como, o tamanho da trama a enviar. Mais tarde, é pedido ao utilizador que insira o nome do ficheiro a ser enviado.

Estrutura do código

Link Layer

A camada da ligação de dados é representada através de uma estrutura de dados onde e guardado a porta de série , o baudrate, o número de sequência da trama esperada, tempo esperado até ao reenvio de uma trama, e o número de tentativas de reenvio, o máximo de tentativas de reenvio, um valor que indica se é necessário reenviar e o tamanho da *frame* S.

```
struct linkLayer {
    char port[20];
    int baudRate;
    unsigned int sequenceNumber;
    unsigned int timeout;
    unsigned int numRetransmissions;
    unsigned int maxRetransmissions;
    unsigned char SET[5];
    unsigned char UAck[5];
    unsigned char DISC[5];
    unsigned char RR[5];
    unsigned char REJ[5];
    size_t frameSLength;
    volatile int retransmit;
};
```

As principais funções desta camada são:

```
int establishConnection(int fd, int status);
int llopen(char * serialport, int status);
int llwrite(int fd, unsigned char * buffer, unsigned int length);
int llread(int fd, unsigned char ** buffer);
int llclose(int fd, int status);
```

Application Layer

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

```
struct applicationLayer {
    int fileDescriptor; // file
    int fd; // serial port
    unsigned int fragmentSize;
    int status;
    char * filename;
    unsigned char * file_data;
    off_t fileSize;
    char controlPacket[];
};
```

As principais funções desta camada são:

```
int sendData();
int receiveData();
int sendControlPacket(unsigned char control_byte);
int sendPacket(int seqNumber, unsigned char * buffer, int length);
int receiveControlPacket();
int receivePacket(unsigned char ** buffer, int seqNumber);
```

Casos de uso principais

Transmissor

O sistema é executado em modo transmissor. Após estabelecer a ligação, é perguntado ao utilizador o nome do ficheiro a enviar. Sendo a abertura e leitura do ficheiro possível, o sistema irá depois enviar pacotes (de acordo com o estabelecido pelo protocolo) para a porta de série utilizando o mecanismo *Stop and Wait*. Concluído o envio, é feita a terminação da ligação.

Sequência de chamada das principais funções:

1. ***llopen**:* abertura da porta de série para escrita e leitura.
2. ***establishConnection**:* envio de uma trama não numerada SET, e receção de uma trama não numerada UA.
3. ***sendData**:* ciclo principal do envio de informação com recurso às funções ***sendControlPacket*** e ***sendPacket**:*
    a. ***sendControlPacket**:* envio de pacote de controlo start com recurso à função ***llwrite**,* perguntando primeiro ao utilizador o nome do ficheiro a transmitir.
    I. llwrite**:* envio de trama de informação com o pacote inserido no campo de dados; receção de trama de supervisão RR/REJ (com recurso a função ***receiveRRREJ). Por cada REJ recebido é retransmitida a trama de informação. O mesmo acontece se o transmissor não tiver recebido nenhuma trama de supervisão em relação à trama enviada durante três segundos.
    b. ***sendPacket***: envio de pacote de dados com recurso à função ***llwrite***
    c. ***sendControlPacket**:* envio de pacote de controlo end, com recurso à função llwrite
    d. ***llclose**:* envio de trama não numerada DISC, receção de trama não numerada DISC, envio de trama não numerada UA e fecho da porta de série.

Recetor

O sistema é executado em modo emissor. Após estabelecer a ligação, o sistema irá receber pacotes e enviar tramas de supervisão RR e REJ consoante. Concluída a receção, é feita a terminação da ligação.

Sequência de chamada das principais funções:

1. ***llopen**:* abertura da porta de série para escrita e leitura.
2. ***establishConnection**:* receção de uma trama não numerada SET, envio de uma trama não numerada UA se não se verificarem erros.
3. ***receiveData**:* ciclo principal da receção de informação com recurso às funções ***receiveControlPacket*** *e* ***receivePacket**:*
    a. ***receiveControlPacket**:* receção de pacote de controlo start com recurso à função ***llread**.*
    I. llread: receção de trama de informação: se se verificar algum erro no cabeçalho, no campo de dados ou no número de sequência é enviada uma trama de supervisão REJ e a execução permanece num ciclo até a receção não ter erros, sendo enviada uma trama de supervisão RR nesse caso.

b. ***receivePacket***: receção de pacote de dados com recurso à função ***llread***.
c. ***receiveControlPacket****:* receção de pacote de controlo end, com recurso à função ***llread***.
d. ***llclose****:* receção de trama não numerada DISC, envio de trama não numerada DISC, receção de trama não numerada UA e fecho da porta de série.

Protocolo de ligação lógica

Aspetos funcionais

- Estabelecimento e terminação da ligação
- Enviar e receber informação através da porta de série recorrendo a *framing*
- Numeração de tramas
- Controlo de erros
- Confirmação
- *Stuffing* e *destuffing* dos pacotes da camada da aplicação

Estratégia de implementação

- **Estabelecimento e terminação da ligação**

A função ***establishConnection*** é responsável por estabelecer a ligação enviando uma trama SET e recebendo uma trama UA (modo transmissor) ou recebendo uma trama SET e enviando uma trama UA (modo recetor): ver o anexo II.

- **Enviar e receber informação através da porta de série recorrendo a *framing***

As funções ***llwrite*** e ***llread*** fornecem à camada da aplicação esses serviços: ver o anexo III.

- **Numeração de tramas**

O transmissor/recetor nesta camada sabe a todo o momento o número de sequência da trama que vai enviar/receber, e vai atualizando-o convenientemente: ver o anexo IV.

- **Controlo de erros**

Ao receber uma trama é feita a verificação do BCC do cabeçalho, do BCC do campo de dados (se aplicável e após *destuffing*) e do número de sequência. Exemplo destas verificações do lado do recetor: ver o anexo V.

- **Confirmação**

Após processar uma trama, o recetor envia uma confirmação positiva/negativa (através de tramas RR ou REJ) consoante a verificação da trama e o número de sequência: ver o anexo VI.

- ***Stuffing* e *destuffing* dos pacotes da camada da aplicação**

Ver o anexo VII.

Protocolo de aplicação

Aspetos funcionais

- Fragmentação/desfragmentação da informação recorrendo a números de sequência
- Envio/receção de pacotes de dados e de controlo
- Leitura/escrita do ficheiro
- Criação do ficheiro (no caso do recetor)
-

Estratégia de implementação

- **Fragmentação/desfragmentação da informação recorrendo a números de sequência**

O transmissor, por exemplo, lê o tamanho fixo do fragmento do ficheiro e constrói um pacote contendo essa informação, sabendo sempre o número de sequência atual. O recetor, mediante verificação do número de sequência, escreve o fragmento que recebe no ficheiro em modo *append*: ver o anexo VIII.

- **Envio/receção de pacotes de dados e de controlo**

As funções ***sendControlPacket*** (transmissor) e ***receiveControlPacket*** (recetor) são responsáveis por construir e enviar/receber pacotes de controlo: ver o anexo IX.

- **Leitura/escrita do ficheiro**

Ver o anexo X.

- **Criação do ficheiro (no caso do recetor)**

Na receção do pacote de controlo *start* é feita a criação do ficheiro: ver o anexo XI.

Validação

Para validação do programa desenvolvido e verificação da transferência do ficheiro *pinguim.gif*, foram efetuados vários testes. No momento de avaliação, foi efetuado teste de interrupção da comunicação na porta de série, bem como, teste de introdução de erros através do curto-circuito existente na porta de série. Posteriormente foram efetuados testes com variação de capacidade de ligação, variação do tamanho dos pacotes, variação da percentagem de erros simulados e geração de atraso de propagação.

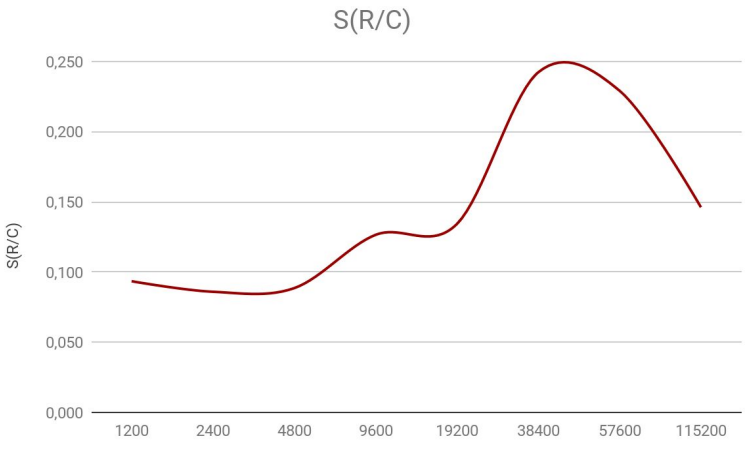
Eficiência do protocolo de ligação de dados

Variação da Capacidade de Ligação

Usando uma imagem de tamanho de 10968 bytes e um tamanho de trama I constante de valor 512, fazendo variar o baudrate, obteve-se:

Baudrate (C)	Tempo (s)	R (bits/s)	S(R/C)
1200	782.366	112.152	0.093
2400	425.402	206.262	0.086
4800	206.419	425.076	0.089
9600	72.241	1214.600	0.127
19200	34.038	2577.852	0.134
38400	9.432	9303.206	0.242
57600	6.648	13198.211	0.229
115200	5.212	16833.814	0.146

Após análise dos resultados obtidos, podemos verificar que quanto maior o *baudrate*, menor será o tempo de transferência, o que não implica que a eficiência (S) seja de igual forma alterada. Confirmamos que existe um pico de eficiência, em que o valor é máximo (*Baudrate* = 38400, S = 0.242). Para além disso, sabemos que existe uma grande alteração na passagem de 19200 para 38400, em que existe um grande aumento de eficiência.

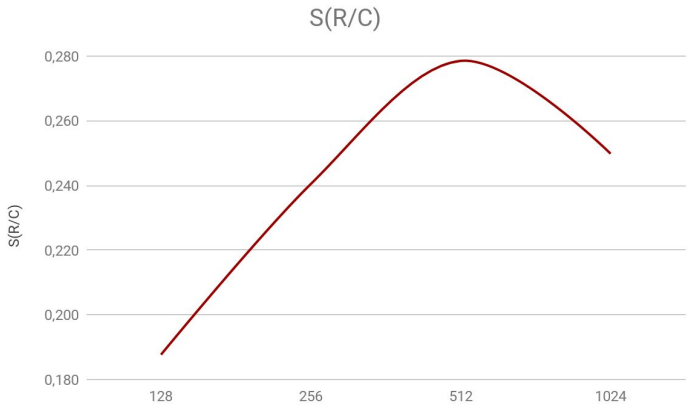


Variação do tamanho das Tramas I

Usando um *baudrate* constante de 38400 e uma imagem de tamanho de 10968 bytes, fazendo variar o tamanho da trama I, obteve-se:

Tamanho (bytes)	Tempo (s)	R (bits/s)	S(R/C)
128	12.175	7206.858	0.188
256	9.503	9233.767	0.240
512	8.204	10695.262	0.279
1024	9.147	9593.033	0.250

Após a análise dos resultados obtidos e do gráfico podemos verificar que o aumento do tamanho das tramas I, implica uma redução do tempo e, sucessivamente, um aumento de eficiência. No entanto, atingindo o valor de 1024 bytes, concluímos que há novamente um aumento do tempo de transferência e redução da eficiência. Por essa razão, temos um máximo no valor 512 bytes.

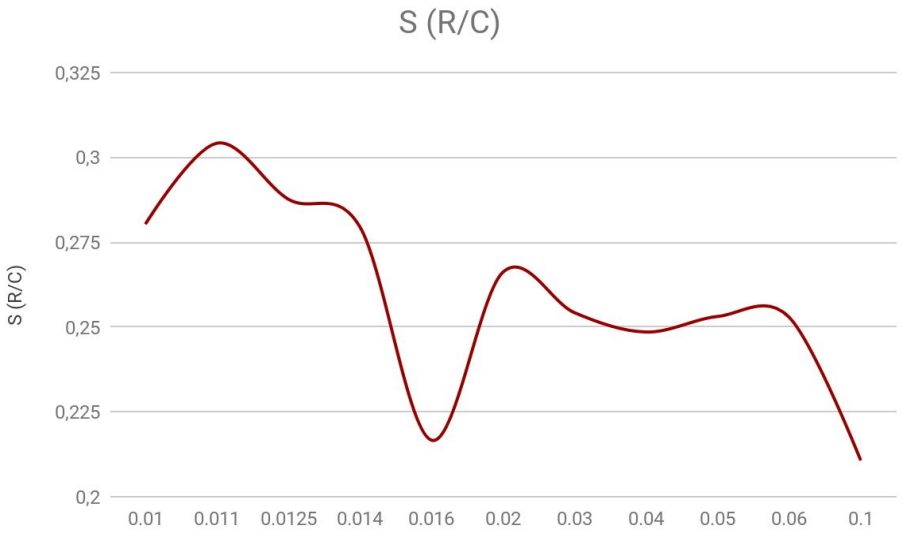


Geração aleatória de erros em tramas de Informação

Utilizando a mesma imagem com o mesmo tamanho, baudrate de 38400 e fazendo variar a probabilidade de ocorrência de erros no cabeçalho dos tramas I, foram obtidos os seguintes dados:

Percentagem	Nº de REJ	Tempo	R	S (R/C)
0.1	2	10.845	8090,672	0,211
0.06	1	9.036	9710,229	0,253
0.05	2	9.028	9719,401	0,253
0.04	0	9.194	9543,108	0,249
0.03	1	8.989	9761,598	0,254
0.02	0	8.584	10221,468	0,266
0.0125	0	7.944	11045,081	0,288
0.01	0	8.152	10763,424	0,280

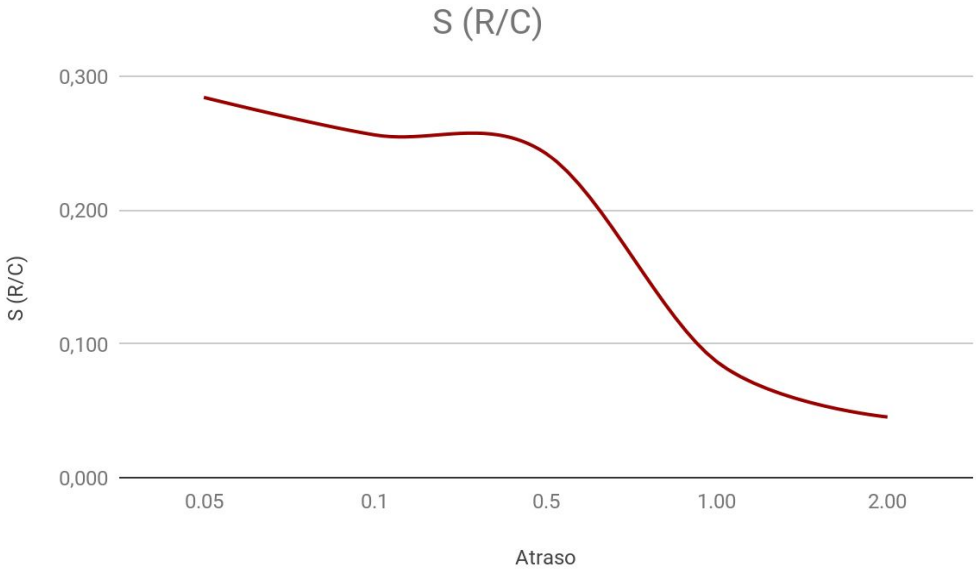
Após uma análise aos dados recebidos com a nossa aplicação, é possível verificar um impacte significativo e questionável na relação entre a probabilidade de erros no cabeçalho dos tramas e tempo de transferência do ficheiro.



Geração de atraso de propagação simulado

Atraso	Tempo	R	S (R/C)
0.05	8.410	10911,622	0,284
0.1	8.921	9835,397	0,256
0.5	9.425	9309,645	0,242
1.00	26.328	3332,134	0,087
2.00	50.393	1741,186	0,045

Como seria de esperar, o resultado foi de acordo, tendo em conta que hipótese será de haver uma relação diretamente proporcional entre o tempo de propagação e o tempo total da transferência do ficheiro.



Conclusão

A implementação deste protocolo de ligação de dados permitiu o uso em prática de conceitos como *Stop and Wait* através de uma variante da mesma, sendo que ao longo do período de trabalho a equipa conseguiu interiorizar as definições necessárias para se trabalhar num código eficiente e coeso, capaz de passar os testes de pré-requisitos na transferência de ficheiros entre dois computadores através de uma porta de série estruturada e independente, na sua totalidade, das suas restantes camadas.

A equipa afirma que, de um modo geral, o desenvolvimento do projeto apresentou alguns desafios, dando em destaque numa fase inicial a dificuldade em conseguir interpretar a interface entre as camadas, e concluir uma forma de conseguir adequar esse sistema em prática. No entanto, a manutenção do projeto tornou-se muito mais fluida após resolvida a interface, sendo que a equipa acha que está adequada a distinção no projeto-fonte e aplicação em geral.

Anexo I - Código fonte

```c
/*Non-Canonical Input Processing*/
#include "applicationLayer.h"

#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */

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

  struct termios oldtio, newtio;

  int baudRate = B38400; // Default Values
  al.fragmentSize = 256;

  if ((argc < 3)) {
    printf("ERROR! This program takes 2 arguments\n");
  }

  if((strcmp("/dev/ttyS0", argv[1]) != 0) && (strcmp("/dev/ttyS1", argv[1]) != 0)) {
    printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
    exit(1);
  }

  if (strcmp("r", argv[2]) == 0) {
    al.status = RECEIVER;
  } else if (strcmp("t", argv[2]) == 0) {
    al.status = TRANSMITTER;
  } else printf("Usage:\tstatus\n\tex: status t");

  al.fragmentSize = getPacketSize();
  st.packetSize = al.fragmentSize;

  baudRate = getBaudrate();

  al.fd = llopen(argv[1], al.status);

  ll.timeout = 3;
  st.timeout = 3;
  ll.maxRetransmissions = 3;
  ll.numRetransmissions = ll.maxRetransmissions;
  ll.frameSLength = 5;
  ll.retransmit = FALSE;
  ll.sequenceNumber = 0;

  if (tcgetattr(al.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] = 1; /* inter-character timer unused */
  newtio.c_cc[VMIN] = 0;  /* blocking read until 1 char received */

  /*
```

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

  tcflush(al.fd, TCIOFLUSH);

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

  system("clear"); //*nix

  if (al.status == RECEIVER) {
    printf("<<< New termios structure set >>>\n\n");
  } else
    printf("<<< New termios structure set >>>\n\nEstablishing conection...\n");

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

  // al do your thing
  go();

  printStatistics(al.status);

  tcsetattr(al.fd, TCSANOW, &oldtio);
  close(al.fd);
  return 0;
}

```

```
#include "applicationLayer.h"

void go() {
  establishConnection(al.fd, al.status);

```

```
60        printf("Error in sendPacket\n");
61        free(buffer);
62        return -1;
63      }
64
65      seqNumber++;
66      seqNumber %= 255;
67    }
68
69    if (sendControlPacket(CONTROLEND) < 0) {
70      printf("Error in sendControlPacket\n");
71      return -1;
72    }
73
74    st.msgSent++;
75
76      if (close(al.fileDescriptor) < 0) {
77      printf("Error closing the file.\n");
78      return -1;
79    }
80
81    free(al.filename);
82    free(buffer);
83    llclose(al.fd, al.status);
84
85    clock_gettime(CLOCK_REALTIME, &finish);
86    calculateTime();
87    system("clear"); //*nix
88    printf("<<< Finished >>>\n");
89    return 0;
90  }
91
92  int receiveData() {
93
94    clock_gettime(CLOCK_REALTIME, &start);
95
96    if (receiveControlPacket() < 0) {
97      printf("error in receiveControlPacket\n");
98      return -1;
99    }
100
101   st.msgRcvd++;
102
103   int bytesRead = 0, seqNumber = 0, counter = 0;
104   unsigned char * buffer;
105
106   while(counter < al.fileSize) {
107
108     st.msgRcvd++;
109
110     bytesRead = receivePacket(&buffer, seqNumber);
111     if(bytesRead < 0) {
112       printf("receivePacket didn't read \n");
113       continue;
114     }
115
116     counter+= bytesRead;
117     if(write(al.fileDescriptor, buffer, bytesRead) <= 0) {
118       perror("Couldn't write to file");
119     }
```

```
120
121     seqNumber++;
122     free(buffer);
123     }
124
125   if (receiveControlPacket() < 0) {
126     printf("Error in receiveControlPacket\n");
127     return -1;
128   }
129
130   st.msgRcvd++;
131
132     if (close(al.fileDescriptor) < 0) {
133     printf("Error closing the file.\n");
134     return -1;
135   }
136
137   free(al.filename);
138   free(al.file_data);
139   llclose(al.fd, al.status);
140   clock_gettime(CLOCK_REALTIME, &finish);
141   calculateTime();
142   system("clear"); //*nix
143   printf("<<< Finished >>>\n");
144   return 0;
145 }
146
147 int sendControlPacket(unsigned char control_byte) {
148
149   if (control_byte == CONTROLSTART) {
150     if(setFile() < 0) {
151       printf("Error getting the file\n");
152       return -1;
153     }
154   }
155
156   struct stat f_information;
157
158   if (fstat(al.fileDescriptor, &f_information) < 0) {
159     perror("Couldn't obtain information regarding the file");
160     return -1;
161   }
162
163   off_t f_size = f_information.st_size; /* total size in bytes, in signed integer */
164   st.filesize = f_size;
165   unsigned int l1 = sizeof(f_size);
166   unsigned int l2 = strlen(al.filename) + 1;
167
168   int startpackage_len = 5 + l1 + l2;
169
170   unsigned char startpackage[startpackage_len];
171
172   startpackage[0] = control_byte; // CONTROLSTART or CONTROLEND
173   startpackage[1] = CONTROLT1;
174   startpackage[2] = l1;
175   *((off_t *)(startpackage + 3)) = f_size;
176   startpackage[3 + l1] = CONTROLT2;
177   startpackage[3 + l1 + 1] = l2; /* +1 from CONTROLT2 */
178
179   strcat((char *) startpackage + 5 + l1, al.filename);
```

```
180
181   if (llwrite(al.fd, startpackage, startpackage_len) < 0) {
182     printf("Couldn't write control package.\n");
183     return -1;
184   }
185
186   return 0;
187 }
188
189 int sendPacket(int seqNumber, unsigned char * buffer, int length) {
190
191   int totalLength = length + 4;
192   unsigned char dataPacket[totalLength];
193
194   dataPacket[0] = CONTROLDATA;
195   dataPacket[1] = seqNumber;
196   dataPacket[2] = length / 256;
197   dataPacket[3] = length % 256;
198
199   memcpy(&dataPacket[4], buffer, length);
200
201   if (llwrite(al.fd, dataPacket, totalLength) < 0) {
202     printf("Error sending data packet");
203     return -1;
204   }
205
206   return 0;
207 }
208
209 int receiveControlPacket() {
210   unsigned char * read_package;
211   unsigned int package_size = llread(al.fd, &read_package);
212   int i;
213
214   if (package_size < 0) {
215     perror("Couldn't read linklayer whilst receiving package.");
216     return -1;
217   }
218
219   int pck_index = 0;
220
221   if (read_package[pck_index] == CONTROLEND) {
222     free(read_package);
223     return 0;
224   } /*End of transfer process, nothing to process any further.*/
225
226   pck_index++; //move on to T1
227   unsigned int n_bytes;
228
229   unsigned char pck_type; //T
230   for (i = 0; i < 2; i++) {
231     pck_type = read_package[pck_index++]; // read T, update to L
232
233     switch (pck_type) {
234     case CONTROLT1:
235       n_bytes = (unsigned int) read_package[pck_index++]; // read L1, update to V1
236
237       al.fileSize = *((off_t *)(read_package + pck_index));
238       st.filesize = al.fileSize;
```

```
239       al.file_data = (unsigned char *) malloc(al.fileSize); /* Allocating file
length not inicialized */
240       pck_index += n_bytes; //update to T2
241       break;
242
243     case CONTROLT2:
244       n_bytes = (unsigned int) read_package[pck_index++]; // read L2, update to V2
245       al.filename = (char *) malloc(n_bytes + 1); /* Allocating filename memory
block not inicialized */
246       memcpy(al.filename, (char *)&read_package[pck_index+1], n_bytes + 1); /*
Transfering block of memory to a.layer's filename */
247       getFile();
248       break;
249
250     default:
251       printf("T: %x \n", pck_type);
252       printf("T parameter in start control packet couldn't be recognised, moving
ahead...\n");
253     }
254   }
255
256   free(read_package);
257   return 0;
258 }
259
260 int receivePacket(unsigned char ** buffer, int seqNumber) {
261
262   unsigned char * information;
263   int K = 0; // number of octets
264
265   if (llread(al.fd, &information) < 0) {
266     printf("error in llread\n");
267     return -1;
268   }
269
270   if (information == NULL) {
271     printf("receivePacket: information = NULL\n");
272     return -1;
273   }
274
275   unsigned char C = information[0]; // control field
276   int N = information[1]; // sequence number
277
278   if (C != CONTROLDATA) {
279     printf("receivePacket: C doesn't indicate data\n");
280     return -1;
281   }
282
283   if (N != seqNumber) {
284     printf("receivePacket: wrong sequence number\n");
285     return -1;
286   }
287
288   int L2 = information[2];
289   int L1 = information[3];
290   K = 256 * L2 + L1;
291
292   *buffer = (unsigned char *) malloc(K);
293   memcpy((*buffer), (information + 4), K);
294
```

```
295   free(information);
296
297   return K;
298 }
299
```

```
1  #include "linkLayer.h"
2
3  #define CONTROLDATA  0x01
4  #define CONTROLSTART 0x02 // control byte in control packet with value start
5  #define CONTROLEND   0x03 // control byte in control packet with value end
6  #define CONTROLT1    0x00 // file's size
7  #define CONTROLT2    0x01 // file's name
8
9  struct applicationLayer {
10   int fileDescriptor; // file
11   int fd; // serial port
12   unsigned int fragmentSize;
13   int status;
14   char * filename;
15   unsigned char * file_data;
16   off_t fileSize;
17   char controlPacket[];
18 };
19
20 struct applicationLayer al;
21
22 void go();
23
24 int setFile();
25
26 int getFile();
27
28 int sendData();
29
30 int receiveData();
31
32 int sendControlPacket(unsigned char control_byte);
33
34 int sendPacket(int seqNumber, unsigned char * buffer, int length);
35
36 int receiveControlPacket();
37
38 int receivePacket(unsigned char ** buffer, int seqNumber);
39
```

```c
#include "linkLayer.h"

void retransmission(int signum) { ll.retransmit = TRUE; }

void setSET() {
  ll.SET[0] = FLAG;
  ll.SET[1] = TRANSMITTERSA;
  ll.SET[2] = SETUP;
  ll.SET[3] = ll.SET[1] ^ ll.SET[2];
  ll.SET[4] = FLAG;
}

void setUAck(int status) {
  ll.UAck[0] = FLAG;

  if (status == TRANSMITTER) {
    ll.UAck[1] = TRANSMITTERSA;
  } else if (status == RECEIVER) {
    ll.UAck[1] = RECEIVERSA;
  }

  ll.UAck[2] = UA;
  ll.UAck[3] = ll.UAck[1] ^ ll.UAck[2];
  ll.UAck[4] = FLAG;
}

void setDisc(int status) {
  ll.DISC[0] = FLAG;

  if (status == TRANSMITTER) {
    ll.DISC[1] = TRANSMITTERSA;
  } else if (status == RECEIVER) {
    ll.DISC[1] = RECEIVERSA;
  }

  ll.DISC[2] = C_DISC;
  ll.DISC[3] = ll.DISC[1] ^ ll.DISC[2];
  ll.DISC[4] = FLAG;
}

void setRR() {
  ll.RR[0] = FLAG;
  ll.RR[1] = RECEIVERSA;
  ll.RR[3] = ll.RR[1] ^ ll.RR[2];
  ll.RR[4] = FLAG;
}

void setRR0() {
  ll.RR[2] = RR_CONTROL0;
  setRR();
}

void setRR1() {
  ll.RR[2] = RR_CONTROL1;
  setRR();
}

void setREJ() {
  ll.REJ[0] = FLAG;
  ll.REJ[1] = RECEIVERSA;
```

```
61    ll.REJ[3] = ll.REJ[1] ^ ll.REJ[2];
62    ll.REJ[4] = FLAG;
63  }
64
65  void setREJ0() {
66    ll.REJ[2] = REJ_CONTROL0;
67    setREJ();
68  }
69
70  void setREJ1() { ll.REJ[2] = REJ_CONTROL1; }
71
72  void sendSFrame(int fd, unsigned char *frame, int triggerAlarm) {
73    int res;
74
75    // mandar trama de supervisão
76    res = write(fd, frame, ll.frameSLength);
77
78    if (res < 0) {
79      perror("Writing S frame error");
80      exit(-1);
81    }
82
83    printf("\nFrame sent (bytes: %d)\n", res);
84
85    if (triggerAlarm) {
86      alarm(ll.timeout);
87    }
88  }
89
90  void receiveSFrame(int fd, int senderStatus, unsigned char controlByte,
91                     unsigned char *retransmit, unsigned int retransmitSize) {
92    enum receiveStates { INIT, F, FA, FAC, FACBCC } receiveState;
93
94    receiveState = INIT;
95    int res;
96    unsigned char byte, byteA;
97    int unreceived = TRUE;
98
99    while (unreceived) {
100
101     if (retransmit != NULL) {
102       if (ll.retransmit) {
103         if (ll.numRetransmissions == 0) {
104           printf("\nNo more retransmissions, leaving...\n");
105           exit(-1);
106         }
107         res = write(fd, retransmit, retransmitSize);
108         printf("\nFrame sent again (bytes: %d)\n", res);
109         ll.numRetransmissions--;
110         alarm(ll.timeout);
111         ll.retransmit = FALSE;
112       }
113     }
114
115     res = read(fd, &byte, 1);
116     if (res < 0) {
117       perror("Receiving reading error");
118     }
119       switch (receiveState) {
120       case INIT:
```

```
121         if (byte == FLAG)
122           receiveState = F;
123         break;
124
125       case F:
126         if (senderStatus == TRANSMITTER) {
127           if (byte == TRANSMITTERSA) {
128             receiveState = FA;
129             byteA = byte;
130           } else if (byte == FLAG) {
131             receiveState = F;
132           } else
133             receiveState = INIT;
134         } else if (senderStatus == RECEIVER) {
135           if (byte == RECEIVERSA) {
136             receiveState = FA;
137             byteA = byte;
138           } else if (byte == FLAG) {
139             receiveState = F;
140           } else
141             receiveState = INIT;
142         }
143         break;
144
145       case FA:
146         if (byte == controlByte) {
147           receiveState = FAC;
148         } else if (byte == FLAG) {
149           receiveState = F;
150         } else
151           receiveState = INIT;
152         break;
153       case FAC:
154         if (byte == (byteA ^ controlByte)) {
155           receiveState = FACBCC;
156         } else if (byte == FLAG) {
157           receiveState = F;
158         } else
159           receiveState = INIT;
160         break;
161
162       case FACBCC:
163         if (byte == FLAG) {
164           unreceived = FALSE;
165           alarm(0);
166           ll.numRetransmissions = ll.maxRetransmissions;
167           ll.retransmit = FALSE;
168           printf("Received frame\n");
169         } else
170           receiveState = INIT;
171         break;
172
173       default:
174         break;
175       }
176   }
177 }
178
179 /* só termina quando ler rr ou quando acabarem as retransmissões */
180 void receiveRRREJ(int fd, unsigned char rr, unsigned char rej,
```

```
                unsigned char *retransmit, unsigned int retransmitSize) {
enum receiveStates { INIT, F, FA, FAC, FACBCC } receiveState;

receiveState = INIT;
int res;
unsigned char byte, controlByte;
int unreceived = TRUE;

while (unreceived) {

  if (retransmit != NULL) {
    if (ll.retransmit) {
      if (ll.numRetransmissions == 0) {
        printf("No more retransmissions, leaving.\n");
        exit(-1);
      }
      res = write(fd, retransmit, retransmitSize);
      printf("\nFrame sent again (bytes: %d)\n", res);
      ll.numRetransmissions--;
      alarm(ll.timeout);
      ll.retransmit = FALSE;
    }
  }

  res = read(fd, &byte, 1);
  if (res < 0) {
    perror("Receiving reading error");
  } else if (res == 0) {
      res = write(fd, retransmit, retransmitSize);
    res = read(fd, &byte, 1);
}

//printf("%x ", byte);

    switch (receiveState) {
    case INIT:
      if (byte == FLAG)
        receiveState = F;
      break;

    case F:
      if (byte == RECEIVERSA) {
        receiveState = FA;
      } else if (byte == FLAG) {
        receiveState = F;
      } else
        receiveState = INIT;
      break;

    case FA:
      if (byte == rr) {
        controlByte = rr;
        receiveState = FAC;
        alarm(0);
      } else if (byte == rej) {
        controlByte = rej;
        ll.retransmit = TRUE;
        ll.numRetransmissions++;
      } else if (byte == FLAG) {
        receiveState = F;
```

```
        } else
          receiveState = INIT;
        break;

      case FAC:
        if (byte == (RECEIVERSA ^ controlByte)) {
          receiveState = FACBCC;
        } else if (byte == FLAG) {
          receiveState = F;
        } else {
          receiveState = INIT;
          printf("BCC error while reading\n");
        }
        break;
      case FACBCC:
        if (byte == FLAG) {
          unreceived = FALSE;
          alarm(0);
          ll.numRetransmissions = ll.maxRetransmissions;
          ll.retransmit = FALSE;
          if (controlByte == rr) {
            st.rrRcvd++;
            printf("Received RR frame\n");
          } else if (controlByte == rej) {
            st.rejRcvd++;
            printf("Received REJ frame\n");
          }
        } else
          receiveState = INIT;
        break;

      default:
        break;
      }
  }
}

int llopen(char *serialport, int status) {
  int fd;
  fd = open(serialport, O_RDWR | O_NOCTTY);

  if (fd < 0) {
    perror("Error opening serial port");
    exit(-1);
  }

  signal(SIGALRM, retransmission);

  return fd;
}

int establishConnection(int fd, int status) {
  setSET();
  setUAck(RECEIVER);

  if (status == TRANSMITTER) {
    sendSFrame(fd, ll.SET, TRUE);
    receiveSFrame(fd, RECEIVER, UA, ll.SET, ll.frameSLength);
  } else if (status == RECEIVER) {
    receiveSFrame(fd, TRANSMITTER, SETUP, NULL, 0);
```

```
301     sendSFrame(fd, ll.UAck, FALSE);
302   }
303
304   printf("\nEstablished connection. Moving on to packets\n\n");
305
306   return 0;
307 }
308
309 unsigned char *byteStuffing(unsigned char *frame, unsigned int *length) {
310   unsigned char *stuffedFrame = (unsigned char *)malloc(*length);
311   unsigned int finalLength = *length;
312
313   int i, j = 0;
314   stuffedFrame[j++] = FLAG;
315
316   // excluir FLAG inicial e final
317   for (i = 1; i < *length - 1; i++) {
318     if (frame[i] == FLAG) {
319       stuffedFrame = (unsigned char *)realloc(stuffedFrame, ++finalLength);
320       stuffedFrame[j] = ESCAPE;
321       stuffedFrame[++j] = PATTERNFLAG;
322       j++;
323       continue;
324     } else if (frame[i] == ESCAPE) {
325       stuffedFrame = (unsigned char *)realloc(stuffedFrame, ++finalLength);
326       stuffedFrame[j] = ESCAPE;
327       stuffedFrame[++j] = PATTERNESCAPE;
328       j++;
329       continue;
330     } else {
331       stuffedFrame[j++] = frame[i];
332     }
333   }
334
335   stuffedFrame[j] = FLAG;
336
337   *length = finalLength;
338
339   return stuffedFrame;
340 }
341
342 unsigned char *byteDestuffing(unsigned char *data, unsigned int *length) {
343   unsigned int finalLength = 0;
344   unsigned char *newData = malloc(finalLength);
345
346   int i;
347   for (i = 0; i < *length; i++) {
348
349     if (data[i] == ESCAPE) {
350       if (data[i + 1] == PATTERNFLAG) {
351         newData = (unsigned char *)realloc(newData, ++finalLength);
352         newData[finalLength - 1] = FLAG;
353         i++;
354         continue;
355       } else if (data[i + 1] == PATTERNESCAPE) {
356         newData = (unsigned char *)realloc(newData, ++finalLength);
357         newData[finalLength - 1] = ESCAPE;
358         i++;
359         continue;
360       }
```

```
    }

    else {
      newData = (unsigned char *)realloc(newData, ++finalLength);
      newData[finalLength - 1] = data[i];
    }
  }

  *length = finalLength;
  return newData;
}

int randomError() {

  return ((random()%10) == 0) ? 1 : 0;
}

/*
Pra mandar tramas i com a mensagem buffer no campo de dados
*/
int llwrite(int fd, unsigned char *buffer, unsigned int length) {
  unsigned int totalLength = 6 + length;
  unsigned char IFrame[totalLength], BCC2, oldBCC;

  IFrame[0] = FLAG;
  IFrame[1] = TRANSMITTERSA; // só o emissor chama a llwrite (o recetor não
                             // envia tramas I)

  if (ll.sequenceNumber == 0) {
    IFrame[2] = CONTROL0;
  } else if (ll.sequenceNumber == 1) {
    IFrame[2] = CONTROL1;
  }

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

  int i;
  IFrame[4] = buffer[0];
  BCC2 = IFrame[4];
  for (i = 5; i < length + 4; i++) {
    IFrame[i] = buffer[i - 4];
    BCC2 = BCC2 ^ IFrame[i];
  }

  IFrame[totalLength - 2] = BCC2;
  IFrame[totalLength - 1] = FLAG;

  unsigned char *stuffedFrame = byteStuffing(IFrame, &totalLength);

  oldBCC = stuffedFrame[totalLength - 2];
  stuffedFrame[totalLength - 2] += randomError();
  int res = write(fd, stuffedFrame, totalLength);
  printf("\nllwrite: sent I frame\n");

  stuffedFrame[totalLength - 2] = oldBCC;

  alarm(ll.timeout);

  if (ll.sequenceNumber == 0) {
    receiveRRREJ(fd, RR_CONTROL1, REJ_CONTROL0, stuffedFrame, totalLength);
```

```
421     ll.sequenceNumber = 1;
422   } else if (ll.sequenceNumber == 1) {
423     receiveRRREJ(fd, RR_CONTROL0, REJ_CONTROL1, stuffedFrame, totalLength);
424     ll.sequenceNumber = 0;
425   }
426
427   free(stuffedFrame);
428
429   ll.retransmit = FALSE;
430   ll.numRetransmissions = ll.maxRetransmissions;
431
432   return res;
433 }
434
435 /*
436 Para ler tramas i
437 retornar nr de caracteres lidos
438 colocar no buffer caracteres lidos
439 */
440 int llread(int fd, unsigned char **buffer) {
441   enum states { INIT, F, FA, FAC, FACBCCD, FACBCCDBCCF } state;
442   state = INIT;
443
444   int i;
445
446   unsigned char byte, controlByte;
447   int unreceived = TRUE;
448
449   unsigned int length = 0;
450   unsigned char *dbcc = (unsigned char *)malloc(length);
451   *buffer = (unsigned char *)malloc(0);
452   unsigned char *destuffed;
453
454   while (unreceived) {
455     int res = read(fd, &byte, 1);
456
457     if (res < 0) {
458       perror("llread: receiving reading error");
459     } else if (res > 0) {
460
461       switch (state) {
462
463       case INIT:
464         if (byte == FLAG)
465           state = F;
466         break;
467       case F:
468         if (byte == TRANSMITTERSA) {
469           state = FA;
470         } else if (byte == FLAG)
471           state = F;
472          else state = INIT;
473         break;
474       case FA:
475         if (byte == CONTROL0 && ll.sequenceNumber == 0) {
476           controlByte = CONTROL0;
477           state = FAC;
478         } else if (byte == CONTROL1 && ll.sequenceNumber == 1) {
479           controlByte = CONTROL1;
480           state = FAC;
```

```
481         } else if (byte == FLAG)
482           state = F;
483          else state = INIT;
484         break;
485       case FAC:
486         if (byte == (TRANSMITTERSA ^ controlByte)) {
487           state = FACBCCD;
488         } else if (byte == FLAG)
489           state = F;
490          else state = INIT;
491         break;
492
493       case FACBCCD:
494         if (byte == FLAG) {
495
496           destuffed = byteDestuffing(dbcc, &length);
497
498           if (!checkBCC(destuffed, length)) {
499             printf("llread: sending REJ\n");
500             st.rejSent++;
501             if (ll.sequenceNumber == 0) {
502               setREJ0();
503               sendSFrame(fd, ll.REJ, FALSE);
504             } else if (ll.sequenceNumber == 1) {
505               setREJ1();
506               sendSFrame(fd, ll.REJ, FALSE);
507             }
508             length = 0;
509             dbcc = (unsigned char *) realloc(dbcc, length);
510             state = INIT;
511             break;
512           } else {
513             state = FACBCCDBCCF;
514             break;
515           }
516         } else {
517           dbcc = (unsigned char *)realloc(dbcc, ++length);
518           dbcc[length - 1] = byte;
519         }
520         break;
521
522       case FACBCCDBCCF:
523         unreceived = FALSE;
524         break;
525
526       default:
527         break;
528       }
529     }
530   }
531
532   // copiar tudo exceto BCC2
533
534   *buffer = (unsigned char *)realloc(*buffer, --length);
535
536   for (i = 0; i < length; i++) {
537     (*buffer)[i] = destuffed[i];
538   }
539
540   printf("llread: sending RR\n");
```

```
541   st.rrSent++;
542   if (ll.sequenceNumber == 0) {
543     setRR1();
544     sendSFrame(fd, ll.RR, FALSE);
545     ll.sequenceNumber = 1;
546   } else if (ll.sequenceNumber == 1) {
547     setRR0();
548     sendSFrame(fd, ll.RR, FALSE);
549     ll.sequenceNumber = 0;
550   }
551
552   free(dbcc);
553   free(destuffed);
554   return length;
555 }
556
557 /* data = D1.....Dn BCC2*/
558 int checkBCC(unsigned char *data, int length) {
559   int i;
560   unsigned char BCC2 = data[0];
561   // exluir BCC2 (ocupa 1 byte)
562   for (i = 1; i < length - 1; i++) {
563     BCC2 ^= data[i];
564   }
565   //último elemento -> BCC2
566   if (BCC2 == data[length - 1]) {
567     return TRUE;
568   } else {
569     printf("BCC2 doesn't check\nBCC2: %x, real BCC2: %x\n", data[length - 1],
570            BCC2);
571     return FALSE;
572   }
573 }
574
575 void sendControlMessage(int fd, unsigned char c) {
576   unsigned char message[5];
577   message[0] = FLAG;
578   message[1] = RECEIVERSA;
579   message[2] = c;
580   message[3] = message[1] ^ message[2];
581   message[4] = FLAG;
582   write(fd, message, 5);
583 }
584
585 int llclose(int fd, int status) {
586
587   if (status == TRANSMITTER) {
588     setDisc(TRANSMITTER);
589     sendSFrame(fd, ll.DISC, TRUE);
590
591     receiveSFrame(fd, RECEIVER, C_DISC, ll.DISC, ll.frameSLength);
592
593     setUAck(TRANSMITTER);
594     sendSFrame(fd, ll.UAck, FALSE);
595   } else if (status == RECEIVER) {
596     receiveSFrame(fd, TRANSMITTER, C_DISC, NULL, 0);
597
598     setDisc(RECEIVER);
599     sendSFrame(fd, ll.DISC, TRUE);
600     receiveSFrame(fd, TRANSMITTER, UA, ll.DISC, ll.frameSLength);
```

```
601   }
602
603   return 0;
604 }
605
```

```c
#include "utils.h"

// S Frames
#define FLAG 0x7E
#define TRANSMITTERSA 0x03
#define RECEIVERSA 0x01
#define SETUP 0x03
#define UA 0x07
#define C_DISC 0x0B
#define RR_CONTROL0 0x05 // RR (receiver ready / positive ACK)
#define RR_CONTROL1 0x85
#define REJ_CONTROL0 0x01 // REJ (reject / negative ACK)
#define REJ_CONTROL1 0x81

#define ESCAPE 0x7D
#define PATTERNFLAG 0x5E
#define PATTERNESCAPE 0x5D

// I Frames
#define CONTROL0 0x00
#define CONTROL1 0X40

struct linkLayer {
  char port[20];
  int baudRate;
  unsigned int sequenceNumber;
  unsigned int timeout;
  unsigned int numRetransmissions;
  unsigned int maxRetransmissions;
  unsigned char SET[5];
  unsigned char UAck[5];
  unsigned char DISC[5];
  unsigned char RR[5];
  unsigned char REJ[5];
    size_t frameSLength;
    volatile int retransmit;
};

struct linkLayer ll;

void retransmission(int signum);

int llopen(char * serialport, int status);
int llwrite(int fd, unsigned char * buffer, unsigned int length);
unsigned char * byteStuffing(unsigned char * frame, unsigned int * length);
int llread(int fd, unsigned char ** buffer);

void setSET();
void setUAck(int status);
void setDisc(int status);
void setRR();
void setRR0();
void setRR1();
void setREJ();
void setREJ0();
void setREJ1();

int establishConnection(int fd, int status);

/*
```

```
61 frame: trama a enviar
62 triggerAlarm: acionar timeout? TRUE ou FALSE
63 */
64 void sendSFrame(int fd, unsigned char * frame, int triggerAlarm);
65
66 /*
67 senderStatus: quem mandou a trama que queremos ler? TRANSMITTER ou RECEIVER
68 controlByte: C da trama que estamos a ler (SET, UA, DISC, RR, ...)
69 retransmit: no caso de haver timeout, que trama queremos retransmitir? se não houver
   timeout retransmit deve ser NULL
70 retransmitSize: tamanho da trama de retransmissão
71 */
72 void receiveSFrame(int fd, int senderStatus, unsigned char controlByte, unsigned char
   * retransmit, unsigned int retransmitSize);
73 void receiveRRREJ(int fd, unsigned char rr, unsigned char rej, unsigned char *
   retransmit, unsigned int retransmitSize);
74
75 int byteStuffingMechanism(unsigned char* message, unsigned char* charsRead, int*
   lengthOfCharsRead);
76
77 int checkBCC(unsigned char* message, int sizeMessage);
78
79 void sendControlMessage(int fd, unsigned char c);
80
81 int llclose(int fd, int status);
82
```

```c
#include "utils.h"

int getPacketSize() {
  int packetSize = -1;
  while (packetSize <= 127 || packetSize > 1024) {
        system("clear"); //*nix
    printf("<<< What is the maximum packet size? >>>\n");
        printf("[128, 256, 512, 1024]\n\n:::");
    scanf("%d", &packetSize);
        if(IsPowerOfTwo(packetSize) -1) {
            packetSize = -1;
        }
  }

  return packetSize;
}

int IsPowerOfTwo(int x) {
    return (x != 0) && ((x & (x - 1)) == 0);
}

int getBaudrate() {
  int choice = -1;
  while (getBaudrateNumber(choice) < 0) {
  system("clear"); //*nix
  printf("<<< What is the baudrate value? >>>\n");
  printf("[300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200]\n\n::: ");
  scanf("%d", &choice);
  }
  st.c = choice;
  return getBaudrateNumber(choice);
}

int getBaudrateNumber(int choice) {
  switch (choice) {
  case 300:
    strcpy(st.speed, "B300");
    return B300;
  case 600:
    strcpy(st.speed, "B600");
    return B600;
  case 1200:
    strcpy(st.speed, "B1200");
    return B1200;
  case 2400:
    strcpy(st.speed, "B2400");
    return B2400;
  case 4800:
    strcpy(st.speed, "B4800");
    return B4800;
  case 9600:
    strcpy(st.speed, "B9600");
    return B9600;
  case 19200:
    strcpy(st.speed, "B19200");
    return B19200;
  case 38400:
    strcpy(st.speed, "B38400");
    return B38400;
  case 57600:
```

```
61      strcpy(st.speed, "B57600");
62      return B57600;
63    case 115200:
64      strcpy(st.speed, "B115200");
65      return B115200;
66    default:
67      return -1;
68    }
69  }
70
71  void calculateTime() {
72
73    long seconds = finish.tv_sec - start.tv_sec;
74    long ns = finish.tv_nsec - start.tv_nsec;
75
76      if (start.tv_nsec > finish.tv_nsec) { // clock underflow
77        --seconds;
78        ns += 1000000000;
79      }
80    st.time = (double)seconds + (double)ns/(double)1e9; //calulate total time
81
82    return;
83  }
84
85  void printStatistics(int status) {
86    printf("\n");
87    printf("<<< Statistics >>>\n\n");
88    printf("Baud Rate: %s\n", st.speed);
89    printf("Packet Size: %d\n", st.packetSize);
90    printf("Timeouts: %d\n\n", st.timeout);
91
92    if(status == TRANSMITTER) {
93      printf("Sent messages: %d\n", st.msgSent);
94      printf("Received RR: %d\n", st.rrRcvd);
95      printf("Received REJ: %d\n\n", st.rejRcvd);
96    } else {
97    printf("Received messages: %d\n", st.msgRcvd);
98    printf("Sent RR: %d\n", st.rrSent);
99    printf("Sent REJ: %d\n\n", st.rejSent);
100 }
101
102   printf("Filesize: %d bytes\n", st.filesize);
103   printf("Transfer time: %.3f s\n", st.time);
104   float r = (st.filesize*8)/(st.time);
105   printf("R: %.3f bits/s\n", r);
106   float s = (r/st.c);
107   printf("S: %.3f \n\n", s);
108 }
109
```

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

#define TRANSMITTER 1
#define RECEIVER 0
#define FALSE 0
#define TRUE 1

struct statistics {
  char speed[20];
  unsigned int timeout;
  unsigned int packetSize;
  unsigned int msgSent;
  unsigned int msgRcvd;
  unsigned int rrSent;
  unsigned int rrRcvd;
  unsigned int rejSent;
  unsigned int rejRcvd;
  int filesize;
  float time;
  int c;
};

struct statistics st;

 struct timespec start, finish;

int getPacketSize();

int IsPowerOfTwo(int x);

int getBaudrate();

int getBaudrateNumber(int choice);

void calculateTime();

void printStatistics();

```

```
# Makefile for RCOM - Project 1

COMPILER_TYPE = gnu
CC = gcc

PROG = rcom
SRCS = main.c utils.c applicationLayer.c linkLayer.c

CFLAGS= -Wall -g

$(PROG): $(SRCS)
  $(CC) $(CFLAGS) $(SRCS) -o $(PROG)

clean:
  rm -f $(PROG)

```

Anexo II - Estabelecimento e terminação da ligação

```
if (status == TRANSMITTER) {
  sendSFrame(fd, ll.SET, TRUE);
  receiveSFrame(fd, RECEIVER, UA, ll.SET, ll.frameSLength);
} else if (status == RECEIVER) {
  receiveSFrame(fd, TRANSMITTER, SETUP, NULL, 0);
  sendSFrame(fd, ll.UAck, FALSE);
}
```

A função ***llclose*** é responsável por terminar a ligação enviando uma trama DISC, recebendo uma trama DISC e enviando uma trama UA (modo transmissor) ou recebendo uma trama DISC, enviando uma trama DISC e recebendo uma trama UA (modo recetor):

```
if (status == TRANSMITTER) {
  setDisc(TRANSMITTER);
  sendSFrame(fd, ll.DISC, TRUE);

  receiveSFrame(fd, RECEIVER, C_DISC, ll.DISC, ll.frameSLength);

  setUAck(TRANSMITTER);
  sendSFrame(fd, ll.UAck, FALSE);
} else if (status == RECEIVER) {
  receiveSFrame(fd, TRANSMITTER, C_DISC, NULL, 0);

  setDisc(RECEIVER);
  sendSFrame(fd, ll.DISC, TRUE);

  receiveSFrame(fd, TRANSMITTER, UA, ll.DISC, ll.frameSLength);
}
```

Anexo III - Enviar e receber informação através da porta de série recorrendo a *framing*

```
t llwrite(int fd, unsigned char * buffer, unsigned int length)
```

```
t llread(int fd, unsigned char ** buffer)
```

Anexo IV - Numeração de tramas

```
ruct linkLayer {
har port[20];
nt baudRate;
nsigned int sequenceNumber;
nsigned int timeout;
.
```

Anexo V - Controlo de erros

Na função ***llread***…

```
se FA:
f (byte == CONTROL0 && ll.sequenceNumber == 0) {
controlByte = CONTROL0;
state = FAC;
 else if (byte == CONTROL1 && ll.sequenceNumber == 1) {
controlByte = CONTROL1;
state = FAC;
```

```
 case FAC:
   if (byte == (TRANSMITTERSA ^ controlByte)) {
     state = FACBCCD;
   }
```

Na função ***checkBCC*** (para verificação do BCC do campo de dados)...

```
exluir BCC2 (ocupa 1 byte)
or (i = 1; i < length - 1; i++) {
BCC2 ^= data[i];

último elemento -> BCC2
f (BCC2 == data[length - 1]) {
return TRUE;
 else {
printf("BCC2 doesn't check\nBCC2: %x, real BCC2: %x\n", data[length - 1],
      BCC2);
return FALSE;
```

Do lado do emissor é também ativado um temporizador de 3 segundos após um envio de qualquer trama I, DISC ou SET, que desencadeia uma retransmissão.

Anexo VI - Confirmação

Na função ***llread***…

```
case FACBCCD:
  if (byte == FLAG) {

    destuffed = byteDestuffing(dbcc, &length);

    if (!checkBCC(destuffed, length)) {
      printf("llread: sending REJ\n");
      st.rejSent++;
      if (ll.sequenceNumber == 0) {
        setREJ0();
        sendSFrame(fd, ll.REJ, FALSE);
      } else if (ll.sequenceNumber == 1) {
        setREJ1();
        sendSFrame(fd, ll.REJ, FALSE);
      }
```

```
f (ll.sequenceNumber == 0) {
 setRR1();
 sendSFrame(fd, ll.RR, FALSE);
 ll.sequenceNumber = 1;
  else if (ll.sequenceNumber == 1) {
 setRR0();
 sendSFrame(fd, ll.RR, FALSE);
 ll.sequenceNumber = 0;
```

Anexo VII - *Stuffing* e *destuffing* dos pacotes da camada da aplicação

Na função ***byteStuffing***…

```
tuffedFrame[j++] = FLAG;

or (i = 1; i < *length - 1; i++) {
 if (frame[i] == FLAG) {
   stuffedFrame = (unsigned char *)realloc(stuffedFrame, ++finalLength);
   stuffedFrame[j] = ESCAPE;
```

```
  stuffedFrame[++j] = PATTERNFLAG;
  j++;
  continue;
} else if (frame[i] == ESCAPE) {
  stuffedFrame = (unsigned char *)realloc(stuffedFrame, ++finalLength);
  stuffedFrame[j] = ESCAPE;
  stuffedFrame[++j] = PATTERNESCAPE;
  j++;
  continue;
} else {
  stuffedFrame[j++] = frame[i];
}
```

Na função ***byteDestuffing***…

```
or (i = 0; i < *length; i++) {

if (data[i] == ESCAPE) {
  if (data[i + 1] == PATTERNFLAG) {
    newData = (unsigned char *)realloc(newData, ++finalLength);
    newData[finalLength - 1] = FLAG;
    i++;
    continue;
  } else if (data[i + 1] == PATTERNESCAPE) {
    newData = (unsigned char *)realloc(newData, ++finalLength);
    newData[finalLength - 1] = ESCAPE;
    i++;
    continue;
  }
}

else {
  newData = (unsigned char *)realloc(newData, ++finalLength);
  newData[finalLength - 1] = data[i];
}
```

Anexo VIII - Fragmentação/desfragmentação da informação recorrendo a números de sequência

Na função ***sendData*** (transmissor)…

```
ile ((bytesRead = read(al.fileDescriptor, buffer, al.fragmentSize))
) {

  st.msgSent++;

  if (sendPacket(seqNumber, buffer, bytesRead) < 0) {
    ...
  }
```

Na função ***receiveData*** (recetor)...

```
  bytesRead = receivePacket(&buffer, seqNumber);
  if(bytesRead < 0) {
    printf("receivePacket didn't read \n");
    continue;
  }

  counter+= bytesRead;
  if(write(al.fileDescriptor, buffer, bytesRead) <= 0) {
    perror("Couldn't write to file");
  }
```

Anexo IX - Envio/receção de pacotes de dados e de controlo

```
t sendControlPacket(unsigned char control_byte)
```

```
t receiveControlPacket()
```

Anexo X - Leitura/escrita do ficheiro

Leitura na função ***sendData***…

```
ile ((bytesRead = read(al.fileDescriptor, buffer, al.fragmentSize)) > 0

.
```

Escrita na função ***receiveData***…

```
  bytesRead = receivePacket(&buffer, seqNumber);
  if(bytesRead < 0) {
    printf("receivePacket didn't read \n");
    continue;
  }

  counter+= bytesRead;
  if(write(al.fileDescriptor, buffer, bytesRead) <= 0) {
    perror("Couldn't write to file");
  }
```

Anexo XI - Criação do ficheiro

Função invocada pela função ***receiveControlPacket***:

```
nt getFile() {
 if((al.fileDescriptor = open(al.filename, O_CREAT|O_WRONLY|O_APPEND,
_IWUSR|S_IRUSR)) < 0) {
   perror("Error opening the file");
   return -1;
 }

 return 0;
```