Redes de Computadores

Protocolo de Ligação de Dados

António Carreiro, José Gonçalves e Pedro Ramalho

5 de Novembro de 2017

Sumário

Relatório relativo ao primeiro projeto da unidade curricular Redes de Computadores do curso Mestrado Integrado em Engenharia Eletrotécnica e de Computadores. O projeto consiste na implementação de uma aplicação que transfere imagens entre dois computadores fazendo uso da porta-série. O objectivo é colocar em prática alguns dos conceitos leccionados na unidade curricular relativos a protocolos de ligação de dados.

Este documento apresenta o estado final do projeto, assim como as considerações dos estudantes responsáveis pela sua implementação face ao resultado obtido.

Conteúdo

1 Introdução

No âmbito da unidade curricular Redes de Computadores, do Mestrado Integrado em Engenharia Informática, foi-nos proposta a realização de um projecto laboratorial, que consiste na implementação de uma aplicação que transfere imagens entre dois computadores fazendo uso da porta-série.

A aplicação usa o protocolo de ligação de dados *Stop N Wait ARQ*, que deve assegurar a fiabilidade da emissão mesmo em caso de desconexão. É também usado um protocolo de aplicação que é responsável pelo envio da imagem. O código desenvolvido está estruturado em camadas, respeitando o princípio de encapsulamento, de modo a assegurar que cada protocolo funciona de forma independente.

O projeto utiliza a linguagem de programação C num ambiente com um sistema operativo baseado em Linux. Durante o desenvolvimento foram utilizadas máquinas reais, com uma ligação de porta de série por cabo. O código pode ser verificado em anexo e será referenciado em algumas secções do relatório.

Este relatório tem como objectivo reportar qual o estado final da aplicação desenvolvia, clarificar detalhes do processo de implementação/código e a opinião dos estudantes face ao projecto realizado. No Capítulo 2 são expostas as estruturas e os mecanismos implementados na concepção da aplicação numa perspectiva macro. Os detalhes relativos à implementação dos protocolos são apresentados nos Capítulos 3 e 4. Os detalhes relativos à implementação de componentes extra são apresentados no Capítulo 5. O Capítulo 6 é relativo à validação e aos testes efectuados.

2 Arquitetura, Estrutura de Código e Casos de uso

2.1 Arquitectura

Cada módulo desenvolvido no projecto pode ser identificado pelos *header files* do código fonte que são:

- `DataLink` implementa e disponibiliza as funcionalidades da camada de ligação de dados.
- `Application` implementa e disponibiliza as funcionalidades da camada de aplicação relacionadas com o envio/recepção de pacotes.
- `ByteStuffing` implementa segmentos e disponibiliza funcionalidades ligadas ao stuffing e destuffing.
- `Alarm` disponibiliza funções relacionadas com a função de alarme.
- `Main.c` onde está a função main e são chamadas as principais funções de outro módulos em conjunto com as operações adicionais necessárias.
- `Utils` foi criada com o intuito de ter métodos, estruturas e funcionalidades úteis a todos os módulos desenvolvidos.

2.2 Estrutura do código

Seguidamente são apresentadas as principais estruturas e funções desenvolvidas em cada módulo. Algumas funções estão omissas, dado que são referidas em maior detalhe nos capítulos relativos aos protocolos implementados.

2.2.1 Estruturas de dados

- `Application`: declarada no `Utils.h` contem informações básicas do programa e serve para guardar o descritor de ficheiro da porta série, o modo de ligação selecionado, o nome de ficheiro do ficheiro a ser recebido/enviado e, por fim, o identificador da porta selecionada.
- `DataLink`: declarada no `Utils.h` serve para guardar o número de tentativas antes de abortar o programa, o valor do tempo a ser esperado antes da próxima tentativa e as estruturas do termios usado pelo sistema operativo e o termios usado pelo nosso programa.
- `Statistics`: declarada no `Utils.h` serve para registar ocorrências, como o número de REJs recebidos.

2.2.2 Funções da DataLink.h

- `initDataLink` trata de inicializar a estrutura *DataLink* e as configurações básicas dos termios.
- `createCommand` e `createFrame` são responsáveis pela criação de tramas de supervisão e não numeradas e pela criação de tramas de informação, respetivamente.

- `llopen`, `llwrite`, `llread` e `llclose` que são encarregues de estabelecer as três fases do protocolo de ligação de dados: *llopen* estabelecimento, *llwrite* e *llread* transferência de dados e, por fim, *llclose* terminação.
- `closeSerialPort` que se encarrega de fechar a porta de série.

2.2.3 Funções Application.h

- `sendControlPackage` e `sendDataPackage` responsáveis, respectivamente, pela criação e envio de pacotes de controlo e de dados.
- `receiveControlPackage` e `receiveDataPackage` responsáveis, respectivamente, pela receção de pacotes de controlo e de dados.
- `sendFile` e `receiveFile` a primeira responsável pela leitura e envio de ficheiros e a segunda pela receção e escrita. Invocam as funções de envio e receção supracitadas.
- `initApplication` trata de inicializar as estruturas *Application* e *Statistics* e resolve o *mode* do utilizador.

2.2.4 Funções Utils.h

- `getFileSize` trata de calcular o tamanho do ficheiro a ser enviado.
- `printProgress` responsável por mostrar ao utilizador o estado atual da transferência do ficheiro.
- `printStatistics` mostra ao utilizador as estatísticas decorrentes do envio do ficheiro.

2.2.5 Funções disponibilizadas por Alarm.h

- `connect` e `send` gerem o uso do alarme em conjunto com as funções que as invocam.

2.2.6 Funções disponibilizadas por ByteStuffing.h

- `stuff` e `destuff` realizam o stuffing e destuffing dos dados nas tramas de informação.

2.2.7 Funções da Main.c

- `processArguments` chamam a função `initApplication` com os ajustes necessários, de modo a enviar/receber uma imagem.
- `printUsage` imprime as instruções a seguir na utilização do programa.

2.3 Casos de Uso

A aplicação desenvolvida deve ser chamada na linha de comandos recebendo como argumentos a porta de série a usar (`/dev/ttyS0` ou `/dev/ttyS1`) e um caracter indicador (`-s` ou `-r`) se a aplicação deve correr em modo emissor ou receptor. Para além disso, no caso do emissor, este deve escolher o ficheiro a enviar e o recetor o nome com que pretende guardar o ficheiro.

Do lado do recetor pode ser iniciada a recepção ficando este à espera que o emissor prossiga com o envio do ficheiro.

3 Protocolo de ligação lógica

Para implementar o protocolo de ligação lógica, seguimos as indicações do enunciado do projeto. Sendo assim, usamos a variante *Stop N Wait*, o que significa que o Emissor, após cada mensagem, aguarda uma resposta do Recetor antes de enviar a mensagem seguinte.

O interface deste protocolo disponibiliza 8 funções. Quatro são as definidas pelo guião do trabalho (`llopen`, `llread`, `llwrite`, `llclose`).

Estas 4 funções são responsáveis pelas seguintes funcionalidades:

- Estabelecer e terminar uma ligação
- Criar e enviar comandos e tramas
- Receber e processar comandos e tramas
- Fazer stuff e destuff das tramas criadas

Isto é conseguido da seguinte maneira:

- `llopen` É a função responsável pelo estabelecimento da ligação através da porta série. Assim que a função é chamada envia um comando SET, e fica a espera de um comando UA, como resposta, para que possa prosseguir. Caso não receba a resposta, o programa tenta de novo um número pré-estabelecido de vezes. Se ainda assim não receber o UA o programa termina.
- `llwrite` Esta função recebe um buffer, o tamanho desse buffer e um identificador da ligação de dados, e tenta enviar esse buffer através da porta série e fica à espera de uma resposta. Se a resposta não chegar ao fim de um tempo pré-definido, é feita uma nova tentativa. Se receber uma resposta e esta for RR, é indicativo de que o buffer foi transmitido corretamente, se, pelo contrário, a resposta for REJ, indica que o buffer não foi transmitido corretamente e é feita uma nova tentativa.
- `llread` Esta função recebe um identificador da ligação de dados e um buffer e é responsável por receber dados através da porta série. Envia o comando REJ em resposta à função llwrite, se o buffer não foi recebido corretamente e envia um RR em caso de sucesso na recepção do buffer.
- `llclose` Esta função recebe um identificador de dados e é responsável por terminar a ligação através da porta série. Envia um comando DISC para o receptor indicando o fim da transmissão e recebe outro DISC do receptor como resposta. Por fim envia um comando UA para o receptor.

4 Protocolo de aplicação

O protocolo de aplicação é responsável pelos pacotes de controlo, pacotes de dados e pela recepção e transmissão do ficheiro. Está implementada no ficheiro Application.h. Os pacotes são diferenciados pelo campo de controlo presentes no início de cada pacote. Os pacotes de controlo marcam o início e o fim da transmissão, se o valor do campo de controlo for dois significa que se trata do início da ransmissão, se, pelo contrário, o valor for três trata-se do fim da transmissão. É também nos pacotes de controlo que são enviadas informações do ficheiro, como por exemplo, o tamanho e, opcionalmente, o nome. Os pacotes de dados têm como valor no campo de controlo o número um e transportam os segmentos do ficheiro a ser transmitido.

A transmissão de um ficheiro processa-se do seguinte modo:

1. Inicialmente, é enviado o pacote de controlo de início de transmissão, onde vão informações como o tamanho do ficheiro.
2. De seguida são enviados os pacotes de dados até todo o ficheiro ter sido completamente transferido.
3. Por fim é enviado o pacote de controlo que marca o fim de transmissão. Este último pacote é exatamente igual ao pacote de inicio de transmissão, à exceção do campo de controlo.

5 Elementos de valorização

5.1 Implementação de REJ

Caso seja detectado um erro no BCC1 ou no BCC2 na função llread, é criado e enviado um comando REJ para o emissor para que a trama com o erro seja reenviada.

```
COMMAND = createCommand(REJ);
stats->rejSent++;

if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
        printf("ERROR: Failed to send REJ buffer.\n");
        return -1;
}
```

5.2 Verificação da integridade dos dados

A aplicação verifica se o tamanho do ficheiro recebido é igual ao tamanho do ficheiro enviado. Os pacotes são numerados para garantir que não são enviados pacotes repetidos.

5.3 Registo de ocorrências

Ao longo da execução do programa, vão sendo registadas as várias ocorrências de timeouts, os REJs enviados e recebidos e as tramas enviadas, recebidas ou reenviadas. Estes dados são guardados na *struct* Statistics.

5.4 Representação do progresso

É apresentada uma barra de progresso que vai incrementando à medida que o ficheiro é transferido.

```
void printProgress(float curr, float total) {
        float per = (100.0 * curr) / total;
        printf("Completed: %6.2f%% [", per);

        int i, pos = (per * PROGRESS_BAR) / 100.0;
        for (i = 0; i < PROGRESS_BAR; i++) {
                if (i <= pos)
                        printf("#");
                else
                        printf(" ");
        }

        printf("]\r");
        fflush(stdout);
}
```

6 Validação

Para testar se o programa funcionava de acordo com as especificações, foram utilizadas diferentes imagens com diferentes tamanhos e também um ficheiro de texto. Todas estas transferências de teste ocorreram conforme o previsto.

Também foi testada a capacidade da aplicação funcionar mesmo que sejam provocadas interrupções na ligação da porta série e/ou provocados curto-circuitos e/ou outro de tipo de imprevistos que levassem a transferência de "lixo" em vez dos pacotes de dados criados na camada de aplicação. O programa reconheceu e recuperou de tais circunstâncias.

7 Conclusões

Neste projecto, como acontece frequentemente em projetos desta natureza, há certamente melhorias a efetuar. Em todo o caso, consideramos que a aplicação foi implementada com sucesso dentro do prazo estabelecido. De facto, a aplicação é capaz de transferir imagens entre 2 computadores através de uma porta-série, sendo o envio bem sucedido mesmo em situações de quebra de ligação ou interferência, como ficou claro na demonstração do projeto.

Adicionalmente, a implementação foi feita em camadas, tendo sido observados os princípios de encapsulamento. Consideramos ainda que os conceitos e implicações práticas dos protocolos usados ficaram claros.

Anexos

A Código Fonte

A.1 Main.c

```
1
2  #include <stdio.h>
3  #include <string.h>
4  #include "Application.h"
5
6  void printUsage(char *command) {
7      printf("Usage: %s <mode> <port> <file>\n", command);
8      printf("Where <mode> includes:\n");
9      printf("    -s    Send a file\n");
10     printf("    -r    Receive a file\n");
11     printf("And <port> includes:\n");
12     printf("    /dev/ttyS0\n");
13     printf("    /dev/ttyS1\n");
14 }
15
16 int processArguments(char **argv) {
17     int mode;
18     char *port, *fileName;
19
20     if (strcmp(argv[1], "-s") == 0)
21         mode = SEND;
22     else if (strcmp(argv[1], "-r") == 0)
23         mode = RECEIVE;
24     else {
25         printf("ERROR: Neither send or receive was specified.\n");
26         printUsage(argv[0]);
27         return -1;
28     }
29
30     if (strcmp(argv[2], "/dev/ttyS0") == 0 || strcmp(argv[2], "/dev/
       ttyS1") == 0)
31         port = argv[2];
32     else {
33         printf("ERROR: Please choose a valid port.\n");
34         printUsage(argv[0]);
35         return -1;
36     }
37
38     if (mode == SEND) {
39         FILE *file;
40         file = fopen(argv[3], "r");
41
42         if (file == NULL) {
43             printf("ERROR: File \"%s\" does not exist.\n", argv[3]);
44             printUsage(argv[0]);
45             return -1;
46         } else {
47             fclose(file);
48             fileName = argv[3];
49         }
50     } else if (mode == RECEIVE) {
51         fileName = argv[3];
52     }
53
54     initApplication(mode, port, fileName);
```

```
55
56      return 0;
57  }
58
59  int main(int argc, char **argv) {
60      if (argc != 4) {
61          printf("ERROR: Wrong number of arguments.\n");
62          printUsage(argv[0]);
63          return -1;
64      } else {
65          processArguments(argv);
66      }
67
68      return 0;
69  }
```

A.2 Application.h

```
1
2
3   #ifndef APPLICATION_H
4   #define APPLICATION_H
5
6   #include <fcntl.h>
7   #include <stdio.h>
8   #include <string.h>
9   #include <stdlib.h>
10  #include <unistd.h>
11  #include "Utils.h"
12  #include "DataLink.h"
13  #include "ByteStuffing.h"
14
15  int sendControlPackage(int ctrl, unsigned char *buffer, int length) {
16      int i;
17      unsigned char ctrlPackage[CTRL_PKG_SIZE];
18
19      ctrlPackage[0] = ctrl;
20      ctrlPackage[1] = PARAM_FILE_SIZE;
21      ctrlPackage[2] = length;
22
23      for (i = 0; i < length; i++)
24          ctrlPackage[i + 3] = buffer[i];
25
26      if (llwrite(app->fd, ctrlPackage, CTRL_PKG_SIZE) == -1)
27          return -1;
28
29      if (ctrl == CTRL_PKG_START)
30          printf("Sent START control package.\n");
31      else if (ctrl == CTRL_PKG_END)
32          printf("\nSent END control package.\n");
33
34      return 0;
35  }
36
37  int sendDataPackage(int N, unsigned char *buffer, int length) {
38      int i;
39      unsigned char dataPackage[length + DATA_PKG_SIZE];
40
41      dataPackage[0] = CTRL_PKG_DATA;
42      dataPackage[1] = N % 255;
43      dataPackage[2] = length / 256;
44      dataPackage[3] = length % 256;
45
46      for (i = 0; i < length; i++)
47          dataPackage[i + 4] = buffer[i];
```

```
48
49      int aux = llwrite(app->fd, dataPackage, length + DATA_PKG_SIZE);
50
51      if (aux == -1)
52          return -1;
53      else if(aux == -2)
54          return -2;
55
56      return 0;
57  }
58
59  int receiveControlPackage(int ctrl, long int *size) {
60      unsigned char *answer;
61
62      if (llread(app->fd, &answer) == -1)
63          return -1;
64
65      if (answer[4] == CTRL_PKG_START)
66          printf("Received START control package.\n");
67      else if (answer[4] == CTRL_PKG_END)
68          printf("\nReceived END control package.\n");
69      else
70          return -1;
71
72      int i;
73      unsigned char fileSizeBuf[8];
74      for (i = 0; i < 8; i++)
75          fileSizeBuf[i] = answer[i + 7];
76      memcpy(size, fileSizeBuf, 8);
77
78      return 0;
79  }
80
81  int receiveDataPackage(unsigned char *buf) {
82      int i, readBytes = 0;
83      unsigned char *package;
84
85      readBytes = llread(app->fd, &package);
86      if (readBytes == -1)
87          return -1;
88      else if (readBytes == -2) {
89          return -2;
90      }
91
92      for (i = 0; i < readBytes; i++)
93          buf[i] = package[i + 8];
94
95      return readBytes;
96  }
97
98  int sendFile(FILE *file) {
99      long int fileSize = getFileSize(app->fileName);
100     unsigned char *fileSizeBuf = (unsigned char *)&fileSize;
101     int aux;
102
103     if (sendControlPackage(CTRL_PKG_START, fileSizeBuf, sizeof(
        fileSizeBuf)) == -1) {
104         printf("ERROR: Failed to send the START control package.\n");
105         return -1;
106     }
107
108     unsigned char fileBuf[MAX_SIZE];
109     size_t readBytes = 0, writtenBytes = 0, i = 0;
110
111     while ((readBytes = fread(fileBuf, sizeof(unsigned char), MAX_SIZE,
```

```
            file)) > 0) {
112             aux = sendDataPackage(i++, fileBuf, readBytes);
113             if (aux == -1) {
114                 printf("ERROR: Failed to send one of the DATA packages.\n")
                        ;
115                 return -1;
116             } else if(aux == -2) {
117                 while(sendDataPackage(i++, fileBuf, readBytes) == -2) {
118                     stats->frameResent++;
119                     continue;
120                 }
121             }
122             stats->frameSent++;
123 
124             memset(fileBuf, 0, MAX_SIZE);
125             writtenBytes += readBytes;
126 
127             printProgress(writtenBytes, fileSize);
128         }
129 
130         if (sendControlPackage(CTRL_PKG_END, fileSizeBuf, sizeof(
                fileSizeBuf)) == -1) {
131             printf("ERROR: Failed to send the END control package.\n");
132             return -1;
133         }
134 
135         return 0;
136     }
137 
138     int receiveFile(FILE *file) {
139         long int fileSize;
140         if (receiveControlPackage(CTRL_PKG_START, &fileSize) == -1) {
141             printf("ERROR: Failed to receive the START control package.\n")
                    ;
142             return -1;
143         }
144 
145         unsigned char tempBuf[MAX_SIZE];
146         size_t readBytes = 0, receivedBytes = 0, i = 0;
147         while (readBytes < fileSize) {
148             if ((receivedBytes = receiveDataPackage(tempBuf)) == -2) {
149                 memset(tempBuf, 0, MAX_SIZE);
150                 continue;
151             }
152 
153             stats->frameReceived++;
154             fwrite(tempBuf, 1, receivedBytes, file);
155             memset(tempBuf, 0, MAX_SIZE);
156 
157             readBytes += receivedBytes;
158             printProgress(readBytes, fileSize);
159         }
160 
161         if (receiveControlPackage(CTRL_PKG_END, &fileSize) == -1) {
162             printf("\nERROR: Failed to receive the END control package.\n")
                    ;
163             return -1;
164         }
165 
166         return 0;
167     }
168 
169     int initApplication(int mode, char *port, char *fileName) {
170         app = (Application *)malloc(sizeof(Application));
171         stats = (Statistics *)malloc(sizeof(Statistics));
```

```
172
173     app->mode = mode;
174     app->port = port;
175     app->fileName = fileName;
176     app->fd = open(port, O_RDWR | O_NOCTTY);
177
178     stats->numberTimeout = 0;
179     stats->rejSent = 0;
180     stats->rejReceived = 0;
181     stats->frameSent = 0;
182     stats->frameResent = 0;
183     stats->frameReceived = 0;
184
185     if (app->fd < 0) {
186         printf("ERROR: Failed to open serial port.\n");
187         return -1;
188     }
189
190     initDataLink();
191
192     if (llopen(app->fd, app->mode) == -1) {
193         printf("ERROR: Failed to create a connection.\n");
194         return -1;
195     } else {
196         printf("Connection successful.\n");
197     }
198
199     if (app->mode == SEND) {
200         FILE *file;
201         file = fopen(app->fileName, "rb");
202         if (file == NULL) {
203             printf("ERROR: Failed to open file \"%s\".\n", app->
                    fileName);
204             return -1;
205         }
206
207         if (sendFile(file) == -1) {
208             printf("ERROR: Failed to send file \"%s\".\n", app->
                    fileName);
209             return -1;
210         }
211
212         if (fclose(file) != 0) {
213             printf("ERROR: Failed to close file \"%s\".\n", app->
                    fileName);
214             return -1;
215         }
216     } else if (app->mode == RECEIVE) {
217         FILE *file;
218         file = fopen(app->fileName, "wb");
219         if (file == NULL) {
220             printf("ERROR: Failed to create file \"%s\".\n", app->
                    fileName);
221             return -1;
222         }
223
224         if (receiveFile(file) == -1) {
225             printf("ERROR: Failed to receive file \"%s\".\n", app->
                    fileName);
226             return -1;
227         }
228
229         if (fclose(file) != 0) {
230             printf("ERROR: Failed to close file \"%s\".\n", app->
                    fileName);
```

```
231                 return -1;
232             }
233         }
234 
235         if (llclose(app->fd, app->mode) == -1) {
236             printf("ERROR: Failed to close the connection.\n");
237             return -1;
238         } else {
239             printf("Disconnection successful.\n");
240         }
241 
242         closeSerialPort();
243         printStatistics();
244 }
245 
246 #endif
```

A.3 DataLink.h

```
1  
2  #ifndef DATALINK_H
3  #define DATALINK_H
4  
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "Utils.h"
8  #include "Alarm.h"
9  #include "ByteStuffing.h"
10 
11 volatile int STOP = FALSE;
12 
13 int initDataLink() {
14     dl = (DataLink *)malloc(sizeof(DataLink));
15 
16     dl->ns = 0;
17     dl->retries = 3;
18     dl->timeout = 3;
19 
20     // Save current port settings
21     if (tcgetattr(app->fd, &dl->oldtio) == -1) {
22         printf("ERROR: Could not save current port settings.\n");
23         return -1;
24     }
25 
26     // Set new termios structure
27     bzero(&dl->newtio, sizeof(dl->newtio));
28     dl->newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
29     dl->newtio.c_iflag = IGNPAR;
30     dl->newtio.c_oflag = 0;
31     dl->newtio.c_lflag = 0;
32     dl->newtio.c_cc[VMIN] = 1;
33     dl->newtio.c_cc[VTIME] = 0;
34 
35     tcflush(app->fd, TCIOFLUSH);
36     if (tcsetattr(app->fd, TCSANOW, &dl->newtio) == -1) {
37         printf("ERROR: Failed to set new termios structure.\n");
38         return -1;
39     }
40 
41     return 0;
42 }
43 
44 unsigned char *createCommand(int command) {
45     unsigned char *buf = (unsigned char *)malloc(COMMAND_SIZE);
46 
```

```
47      buf[0] = FLAG;
48      buf[1] = A;
49
50      switch (command) {
51      case UA:
52          buf[2] = C_UA;
53          break;
54      case SET:
55          buf[2] = C_SET;
56          break;
57      case DISC:
58          buf[2] = C_DISC;
59          break;
60      case RR:
61          buf[2] = C_RR;
62          break;
63      case REJ:
64          buf[2] = C_REJ;
65          break;
66      }
67
68      buf[3] = buf[1] ^ buf[2];
69      buf[4] = FLAG;
70
71      return buf;
72  }
73
74  unsigned char *createFrame(unsigned char *buffer, int length) {
75      int i;
76      unsigned char BCC2 = 0x00;
77
78      FRAME_SIZE = length + 6;
79      unsigned char *frame = (unsigned char *)malloc(length + 6);
80
81      frame[0] = FLAG;
82      frame[1] = A;
83      frame[2] = dl->ns << 6;
84      frame[3] = frame[1] ^ frame[2];
85
86      for (i = 0; i < length; i++) {
87          BCC2 ^= buffer[i];
88          frame[i + 4] = buffer[i];
89      }
90
91      frame[length + 4] = BCC2;
92      frame[length + 5] = FLAG;
93
94      return frame;
95  }
96
97  int llopen(int fd, int mode) {
98      int res = 0;
99      unsigned char x, flag;
100     unsigned char answer[COMMAND_SIZE];
101
102     switch (mode) {
103     case SEND:
104         COMMAND = createCommand(SET);
105
106         (void)signal(SIGALRM, connect);
107         alarm(dl->timeout);
108
109         if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
110             printf("ERROR: Failed to send SET buffer.\n");
111             return -1;
```

```
112         }
113
114         while (STOP == FALSE) {
115             res += read(fd, &x, 1);
116
117             if (res == 1 && x == FLAG)
118                 flag = x;
119             else if (res == 1 && x != FLAG)
120                 res = 0;
121             if (x == flag && res > 1)
122                 STOP = TRUE;
123
124             answer[res - 1] = x;
125         }
126
127         if (answer[3] != (A ^ C_UA)) {
128             printf("ERROR: Failure on initial connection.\n");
129             return -1;
130         }
131
132         alarm(0);
133         break;
134     case RECEIVE:
135         while (STOP == FALSE) {
136             res += read(fd, &x, 1);
137
138             if (res == 1 && x == FLAG)
139                 flag = x;
140             else if (res == 1 && x != FLAG)
141                 res = 0;
142             if (x == flag && res > 1)
143                 STOP = TRUE;
144
145             answer[res - 1] = x;
146         }
147
148         if (answer[3] != (A ^ C_SET)) {
149             printf("ERROR: Failure on initial connection.\n");
150             return -1;
151         }
152
153         COMMAND = createCommand(UA);
154         if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
155             printf("ERROR: Failed to send UA buffer.\n");
156             return -1;
157         }
158         break;
159     }
160
161     return 0;
162 }
163
164 int llwrite(int fd, unsigned char *buffer, int length) {
165     int res = 0, newSize = 0;
166     unsigned char x, flag;
167     unsigned char answer[COMMAND_SIZE];
168
169     unsigned char *frame = createFrame(buffer, length);
170     STUFFED = stuff(frame, FRAME_SIZE, &newSize);
171     STUFFED_SIZE = newSize;
172
173     (void)signal(SIGALRM, send);
174     alarm(dl->timeout);
175
176     write(fd, STUFFED, STUFFED_SIZE);
```

```
177
178     STOP = FALSE;
179     while (STOP == FALSE) {
180         res += read(fd, &x, 1);
181
182         if (res == 1 && x == FLAG)
183             flag = x;
184         else if (res == 1 && x != FLAG)
185             res = 0;
186         if (x == flag && res > 1)
187             STOP = TRUE;
188
189         answer[res - 1] = x;
190     }
191
192     if (answer[3] == (A ^ C_RR)) {
193         alarm(0);
194         triesSend = 0;
195         free(STUFFED);
196         return STUFFED_SIZE;
197     } else if (answer[3] == (A ^ C_REJ)) {
198         alarm(0);
199         triesSend = 0;
200         stats->rejReceived++;
201         return -2;
202     }
203
204 }
205
206 int llread(int fd, unsigned char **buffer) {
207     int size = 0;
208     volatile int over = FALSE, state = START;
209     unsigned char *fileBuf = (unsigned char *)malloc((MAX_SIZE + 10) *
        2);
210
211     while (over == FALSE) {
212         unsigned char x;
213
214         if (state != DONE) {
215             if (read(app->fd, &x, 1) == -1)
216                 return -1;
217         }
218
219         switch (state) {
220         case START:
221             if (x == FLAG) {
222                 fileBuf[size++] = x;
223                 state = FLAG_RCV;
224             }
225             break;
226         case FLAG_RCV:
227             if (x == A) {
228                 fileBuf[size++] = x;
229                 state = A_RCV;
230             } else if (x != FLAG) {
231                 size = 0;
232                 state = START;
233             }
234             break;
235         case A_RCV:
236             if (x != FLAG) {
237                 fileBuf[size++] = x;
238                 state = C_RCV;
239             } else if (x == FLAG) {
240                 size = 1;
```

```
241                 state = FLAG_RCV;
242             } else {
243                 size = 0;
244                 state = START;
245             }
246             break;
247         case C_RCV:
248             if (x == (fileBuf[1] ^ fileBuf[2])) {
249                 fileBuf[size++] = x;
250                 state = BCC_OK;
251             } else if (x == FLAG) {
252                 size = 1;
253                 state = FLAG_RCV;
254             } else {
255                 size = 0;
256                 state = START;
257             }
258             break;
259         case BCC_OK:
260             if (x == FLAG) {
261                 fileBuf[size++] = x;
262                 state = DONE;
263             } else if (x != FLAG) {
264                 if (size > MAX) {
265                     COMMAND = createCommand(REJ);
266                     stats->rejSent++;
267 
268                     if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
269                         printf("ERROR: Failed to send REJ buffer.\n");
270                         return -1;
271                     }
272 
273                     return -2;
274                 }
275 
276                 fileBuf[size++] = x;
277             }
278             break;
279         case DONE:
280             fileBuf[size] = 0;
281             over = TRUE;
282             break;
283         }
284     }
285 
286     int k;
287     unsigned char *destuffed = destuff(fileBuf, size);
288     unsigned char BCC2 = 0x00;
289 
290     if (destuffed[3] != (destuffed[1] ^ destuffed[2])) {
291         COMMAND = createCommand(REJ);
292         stats->rejSent++;
293 
294         if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
295             printf("ERROR: Failed to send REJ buffer.\n");
296             return -1;
297         }
298 
299         return -2;
300     } else if (destuffed[2] != C_SET && destuffed[2] != C_UA &&
        destuffed[2] != C_DISC && destuffed[2] != C_RR && destuffed[2]
        != C_REJ) {
301         int i;
302 
303         if (destuffed[4] == CTRL_PKG_START || destuffed[4] ==
```

```
            CTRL_PKG_END)
304             k = destuffed[6] + 3;
305         else if (destuffed[4] == CTRL_PKG_DATA)
306             k = (destuffed[6] * 256) + destuffed[7] + 4;
307
308         for (i = 0; i < k; i++)
309             BCC2 ^= destuffed[i + 4];
310
311         if (BCC2 != destuffed[4 + k]) {
312             COMMAND = createCommand(REJ);
313             stats->rejSent++;
314
315             if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
316                 printf("ERROR: Failed to send REJ buffer.\n");
317                 return -1;
318             }
319
320             return -2;
321         } else {
322             COMMAND = createCommand(RR);
323
324             if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
325                 printf("ERROR: Failed to send RR buffer.\n");
326                 return -1;
327             }
328         }
329     }
330
331     free(fileBuf);
332     *buffer = destuffed;
333
334     return k - 4;
335 }
336
337 int llclose(int fd, int mode) {
338     int res = 0;
339     unsigned char x, flag;
340     unsigned char answer[COMMAND_SIZE];
341
342     switch (mode) {
343     case SEND:
344         COMMAND = createCommand(DISC);
345         if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
346             printf("ERROR: Failed to send DISC buffer.\n");
347             return -1;
348         }
349
350         STOP = FALSE;
351         while (STOP == FALSE) {
352             res += read(fd, &x, 1);
353
354             if (res == 1 && x == FLAG)
355                 flag = x;
356             else if (res == 1 && x != FLAG)
357                 res = 0;
358             if (x == flag && res > 1)
359                 STOP = TRUE;
360
361             answer[res - 1] = x;
362         }
363
364         if (answer[3] != (A ^ C_DISC)) {
365             printf("ERROR: Failure on closing connection.\n");
366             return -1;
367         }
```

```
368
369             COMMAND = createCommand(UA);
370             if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
371                 printf("ERROR: Failed to send UA buffer.\n");
372                 return -1;
373             }
374             break;
375         case RECEIVE:
376             STOP = FALSE;
377             while (STOP == FALSE) {
378                 res += read(fd, &x, 1);
379
380                 if (res == 1 && x == FLAG)
381                     flag = x;
382                 else if (res == 1 && x != FLAG)
383                     res = 0;
384                 if (x == flag && res > 1)
385                     STOP = TRUE;
386
387                 answer[res - 1] = x;
388             }
389
390             if (answer[3] != (A ^ C_DISC)) {
391                 printf("ERROR: Failure on closing connection.\n");
392                 return -1;
393             }
394
395             COMMAND = createCommand(DISC);
396             if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
397                 printf("ERROR: Failed to send DISC buffer.\n");
398                 return -1;
399             }
400
401             res = 0;
402             STOP = FALSE;
403             while (STOP == FALSE) {
404                 res += read(fd, &x, 1);
405
406                 if (res == 1 && x == FLAG)
407                     flag = x;
408                 else if (res == 1 && x != FLAG)
409                     res = 0;
410                 if (x == flag && res > 1)
411                     STOP = TRUE;
412
413                 answer[res - 1] = x;
414             }
415
416             if (answer[3] != (A ^ C_UA)) {
417                 printf("ERROR: Failure on closing connection.\n");
418                 return -1;
419             }
420             break;
421         }
422
423         return 0;
424     }
425
426     int closeSerialPort() {
427         if (tcsetattr(app->fd, TCSANOW, &dl->oldtio) == -1) {
428             perror("tcsetattr");
429             return -1;
430         }
431
432         close(app->fd);
```

```
433     return 0;
434 }
435
436 #endif
```

A.4 Utils.h

```
1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #define TRUE             1
5  #define FALSE            0
6
7  #define SEND             0
8  #define RECEIVE          1
9
10 #define UA               0
11 #define SET              1
12 #define DISC             2
13 #define RR               3
14 #define REJ              4
15
16 #define A                0x03
17 #define FLAG             0x7E
18 #define DATA             0x01
19 #define ESCAPE           0x7D
20
21 #define C_UA             0x03
22 #define C_SET            0x07
23 #define C_DISC           0x0B
24 #define C_RR             0x05
25 #define C_REJ            0x01
26
27 #define START            0
28 #define FLAG_RCV         1
29 #define A_RCV            2
30 #define C_RCV            3
31 #define BCC_OK           4
32 #define DONE             5
33
34 #define COMMAND_SIZE     5
35 #define DATA_PKG_SIZE    4
36 #define CTRL_PKG_SIZE    11
37
38 #define CTRL_PKG_DATA    1
39 #define CTRL_PKG_START   2
40 #define CTRL_PKG_END     3
41
42 #define PARAM_FILE_SIZE  0
43 #define PARAM_FILE_NAME  1
44
45 #define _POSIX_SOURCE    1
46 #define PROGRESS_BAR     45
47 #define MAX_SIZE         512
48 #define MAX 131085
49 #define BAUDRATE         B38400
50
51 #include <stdio.h>
52 #include <termios.h>
53 #include <sys/stat.h>
54
55 typedef struct {
56     int fd, mode;
57     char *port, *fileName;
58 } Application;
```

```
59
60  typedef struct {
61      int ns, retries, timeout;
62      struct termios oldtio, newtio;
63  } DataLink;
64
65  typedef struct {
66      int numberTimeout;
67      int rejSent, rejReceived;
68      int frameSent, frameResent, frameReceived;
69  } Statistics;
70
71  DataLink *dl;
72  Application *app;
73  Statistics *stats;
74
75  volatile int FRAME_SIZE, STUFFED_SIZE;
76
77  long int getFileSize(char *fileName) {
78      struct stat st;
79
80      if (stat(fileName, &st) == 0)
81          return st.st_size;
82
83      printf("ERROR: Could not get file size.\n");
84      return -1;
85  }
86
87  void printBuffer(unsigned char *buf, int size) {
88      int i;
89      for (i = 0; i < size; i++)
90          printf("0x%02x ", buf[i]);
91  }
92
93  void printProgress(float curr, float total) {
94      float per = (100.0 * curr) / total;
95      printf("Completed: %6.2f%% [", per);
96
97      int i, pos = (per * PROGRESS_BAR) / 100.0;
98      for (i = 0; i < PROGRESS_BAR; i++) {
99          if (i <= pos)
100             printf("#");
101         else
102             printf(" ");
103     }
104
105     printf("]\r");
106     fflush(stdout);
107 }
108
109 void printStatistics() {
110     printf("\n - Statistics - \n");
111     printf(" # Number of timeouts: %d\n", stats->numberTimeout);
112     printf(" # Number of REJs sent: %d\n", stats->rejSent);
113     printf(" # Number of REJs received: %d\n", stats->rejReceived);
114     printf(" # Number of Frames sent: %d\n", stats->frameSent);
115     printf(" # Number of Frames resent: %d\n", stats->frameResent);
116     printf(" # Number of Frames received: %d\n", stats->frameReceived);
117 }
118
119 #endif
```

A.5 Alarm.h

```
1
```

```
2  #ifndef ALARM_H
3  #define ALARM_H
4
5  #include <stdio.h>
6  #include <signal.h>
7  #include <unistd.h>
8  #include "Utils.h"
9
10 unsigned char *COMMAND, *STUFFED;
11 int triesSend = 0, triesConnect = 0;
12
13 void connect() {
14     if (triesConnect < dl->retries) {
15         printf("\nNo response. Tries left = %d\n", dl->retries -
               triesConnect);
16         write(app->fd, COMMAND, COMMAND_SIZE);
17         triesConnect++;
18         alarm(dl->timeout);
19     } else {
20         alarm(0);
21         printf("\nERROR: Failed to create a connection.\n");
22         exit(-1);
23     }
24 }
25
26 void send() {
27     if (triesSend < dl->retries) {
28         printf("\nNo response. Tries left = %d\n", dl->retries -
               triesSend);
29         write(app->fd, STUFFED, STUFFED_SIZE);
30         triesSend++;
31         stats->frameResent++;
32         stats->numberTimeout++;
33         alarm(dl->timeout);
34     } else {
35         alarm(0);
36         printf("\nERROR: Failed to send frame.\n");
37         exit(-1);
38     }
39 }
40
41 #endif
```

A.6 ByteStuffing.h

```
1
2  #ifndef BYTESTUFFING_H
3  #define BYTESTUFFING_H
4
5  #include <stdlib.h>
6  #include <string.h>
7  #include "Utils.h"
8
9  unsigned char *stuff(unsigned char *buf, int length, int *newSize) {
10     int i, j = 0;
11     unsigned char *stuffed = (unsigned char *)malloc(length * 2);
12
13     for(i = 0; i < length; i++) {
14         if(buf[i] == FLAG && i != 0 && i != (length - 1)) {
15             stuffed[j] = ESCAPE;
16             stuffed[j + 1] = FLAG ^ 0x20;
17             j = j + 2;
18         } else if(buf[i] == ESCAPE) {
19             stuffed[j] = ESCAPE;
20             stuffed[j + 1] = ESCAPE ^ 0x20;
```

```
21              j = j + 2;
22          } else {
23              stuffed[j] = buf[i];
24              j++;
25          }
26      }
27
28      *newSize = j;
29      return stuffed;
30  }
31
32  unsigned char *destuff(unsigned char *buf, int length) {
33      int i, j = 0;
34      unsigned char *destuffed = (unsigned char *)malloc(length);
35
36      for(i = 0; i < length; i++) {
37          if(buf[i] == ESCAPE && buf[i + 1] == (FLAG ^ 0x20)) {
38              destuffed[j] = FLAG;
39              j++;
40              i++;
41          } else if(buf[i] == ESCAPE && buf[i + 1] == (ESCAPE ^ 0x20)) {
42              destuffed[j] = ESCAPE;
43              j++;
44              i++;
45          } else {
46              destuffed[j] = buf[i];
47              j++;
48          }
49      }
50
51      return destuffed;
52  }
53
54  #endif
```