

# FEUP – Redes de Computadores 2021/2022

## 1.º Trabalho Laboratorial

Diogo Costa  
up201906731@edu.fe.up.pt

Francisco Colino  
up201905405@edu.fe.up.pt

11 de dezembro de 2021

### Sumário

Este projeto foi realizado como o sendo o 1.º projeto laboratorial da unidade curricular *Redes de Computadores*, fazendo esta parte da *Licenciatura em Engenharia Informática e Computação* da *FEUP*. O projeto consistiu na implementação de um protocolo de ligação de dados que permite a transferência confiável de dados entre dois computadores através da porta série. Para além deste protocolo foi implementada uma aplicação de transferência de ficheiros que faz uso do serviço fornecido pelo protocolo de ligação de dados.

Todos os objetivos foram atingidos na medida em que foi implementado com sucesso um protocolo de ligação de dados confiável e a aplicação que faz uso desse protocolo. Esta implementação foi testada em contexto laboratorial e provou ser resistente a interrupções e interferências. Foi ainda feita uma análise estatística experimental e comparados os resultados aos expectados teoricamente.

## 1 Introdução

O objetivo deste trabalho é implementar um protocolo de ligação de dados, de acordo com o guião fornecido, que permite fazer a transmissão de ficheiros de forma assíncrona através de portas série assegurando a integridade dos ficheiros. Esta integridade deve ser assegurada mesmo com interrupções e interferências. Este relatório procura expor a teoria por de trás deste projeto, como é que os objetivos foram alcançados e os testes efetuados à eficiência do protocolo.

Este relatório está estruturado da seguinte forma:

- **Arquitetura** – Blocos funcionais e interfaces.
- **Estrutura do Código** – Demonstração das *APIs*, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais** – Identificação dos casos de uso e representação das sequências de chamada de funções.
- **Protocolo de ligação lógica** – Identificação dos principais aspetos funcionais da ligação lógica e descrição das estratégias usadas na implementação destes aspetos com extratos de código.
- **Protocolo de aplicação** – Identificação dos principais aspetos funcionais da aplicação e descrição das estratégias usadas na implementação destes aspetos com extratos de código.
- **Validação** – Descrição dos testes efetuados com apresentação quantificada dos resultados.
- **Eficiência do protocolo de dados** – Caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.
- **Conclusão** – Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

## 2 Arquitetura

O projeto está dividido em dois blocos funcionais principais, **Data Link** e **Application**, podendo desempenhar dois papéis distintos, emissor e recetor. Estas duas camadas são independentes com o intuito de tornar o código mais modular de modo a que a camada mais baixo possa ser usado com outras aplicações.

A **camada de ligação de dados (Data Link)** é o nível mais baixo. Esta trabalha com a porta série e oferece uma interface, que permite a abertura, fecho, leitura e escrita numa porta série. Esta permite comunicação assíncrona e fidedigna entre dois computadores com a capacidade de deteção e tratamento apropriado de erros, interrupções e interferências sem que haja perdas de dados ou transferência de dados incorretos.

A **camada da aplicação (Application)** é uma interface que usa a linha de comandos para comunicar com o utilizador. Esta oferece dois serviços: emissor e recetor. Em ambos, o utilizador tem a liberdade de escolher a porta série a utilizar e, no caso do emissor, escolher o ficheiro a enviar e que nome dar a este no envio ao recetor. A aplicação é também responsável pela divisão do ficheiro original em pacotes de um tamanho predefinido para envio na camada de ligação de dados.

## 3 Estrutura do código

O código encontra-se dividido em 4 ficheiros *.c* de modo a facilitar a divisão nas camadas mencionadas.

Ao **Data Link** corresponde o ficheiro *linklayer.c*, à **Application** correspondem os ficheiros *aplic.c*, *receiver.c* e *sender.c*.

Funções principais da camada **Data Link**:

- *llopen()* – estabelece a ligação entre as máquinas através de tramas de Supervisão (S)
- *llwrite()* – envia tramas de Informação (I) e recebe tramas de Supervisão (S)
- *llread()* – lê tramas de Informação (I) e envia tramas de Supervisão (S)
- *llclose()* – termina a ligação entre as máquinas através de tramas de Supervisão (S)

Macros principais da camada **Data Link**:

- *BAUDRATE* – valor da *baud rate* a ser utilizada na comunicação
- *TIME\_OUT\_TIME* – segundos que as funções esperam pelo envio de dados antes de entrarem em *timeout*
- *MAX\_NO\_TIMEOUT* – número máximo de *timeouts* consecutivos
- *DATA\_PACKET\_MAX\_SIZE* – tamanho máximo que os dados do campo de informação, antes de *stuffing*, podem ter por envio de pacote

Funções principais da camada **Application**:

- *send\_file()* – reparte um ficheiro em pacotes de tamanho predefinido e faz uso da função *llwrite()* para os enviar
- *receive\_file()* – recebe diversos pacotes de dados, fazendo uso da função *llread()*, e organiza-os de forma a montar o ficheiro recebido

Macros principais da camada ***Application***:

- `CONTROL_PACKET_MAX_SIZE` – tamanho máximo de um pacote de controlo da aplicação
- `PACKET_MAX_SIZE` – tamanho máximo de um pacote da aplicação, obtido como sendo o máximo entre o `CONTROL_PACKET_MAX_SIZE` e o `DATA_PACKET_MAX_SIZE`
- `FILE_NAME_MAX_SIZE` – tamanho máximo do nome de um ficheiro em *linux*

## 4 Casos de uso principais

### 4.1 Transmissor

A aplicação é executada em modo ***sender***. O utilizador escolhe a porta série a utilizar, o caminho do ficheiro a mandar ao recetor e o nome que deve ser dado ao ficheiro na sua receção. Primeiro é estabelecida a ligação entre o transmissor e o recetor, verificando que o ficheiro passado à aplicação é válido este é então enviado em pacotes através duma porta série fazendo uso do mecanismo *Stop-and-Wait*. Após o envio a ligação é terminada. Exemplo:

```
./sender 0 "pinguim.gif" "p1.gif"
```

Uma sequência mais detalhada do que acontece:

1. Abre a porta série e estabelece a conexão com `fd = llopen("/dev/ttyS0", TRANSMITTER)`
2. Envia pacotes de controlo e o ficheiro repartido em pacotes de dados através da função `llwrite()`
3. Fecha a porta série, terminando assim a ligação, com `llclose()`

### 4.2 Recetor

A aplicação é executada em modo ***receiver***. O utilizador escolhe a porta série a utilizar. Primeiro é estabelecida a ligação entre o transmissor e o recetor e é feita a leitura pacote a pacote do ficheiro a ser recebido. Após a leitura a ligação é terminada. Exemplo:

```
./receiver 4
```

Uma sequência mais detalhada do que acontece:

1. Abre a porta série e estabelece a conexão com `fd = llopen("/dev/ttyS4", RECEIVER)`
2. Os pacotes enviados pelo transmissor são lidos sequencialmente através da função `llwrite()`
3. Fecha a porta série, terminando assim a ligação, com `llclose()`

## 5 Protocolo de ligação lógica

O protocolo de ligação lógica teve como objetivo fornecer um serviço de comunicação fiável entre dois sistemas ligados por um meio de comunicação, neste caso um cabo série. Este é responsável por algumas funcionalidades genéricas:

- Sincronismo de trama – dados organizados em tramas
- Estabelecimento e encerramento da ligação
- Byte stuffing das informações das tramas assegurando transparência

- Transmissão de tramas
- Receção das trams com envio de resposta
- Controlo de erros
- Controlo de fluxo

## 5.1 Estabelecimento da ligação

Para este efeito o protocolo quando utilizado como transmitter envia uma trama de Supervisão (S) com o seguinte formato, sendo que C terá valor de SET.

- F – Flag
- A – Campo de Endereço
- C – Campo de Controlo:
  - SET – set up
  - DISC – disconnect
  - UA – unnumbered acknowledgment
  - RR – receiver ready
  - REJ – reject
- $BCC_1$  – Campo de Proteção (cabeçalho)
- F – Flag

No lado do receiver ao receber uma trama (S) com C = SET, manda uma outra trama (S) mas com c = UA, funcionando como resposta de ACK. Para a construção destas tramas é usada a função:

```
1 static void control_frame_builder(control_frame_type_t cft, uint8_t msg[]);
```

O transmitter tem incorporado um timeout que ao enviar a trama espera pela resposta de receiver, se ao fim de 3 segundos não obtiver resposta reenvia a trama, este processo é repetido 3 vezes sendo que o programa termina no 3º timeout.

Se a receção da trama (S) com C = UA for feita com sucesso pelo transmitter, então a ligação é dada como estabelecida. Todo este processo é realizado pela função:

```
1 int llopen(int porta, type_t type);
```

## 5.2 Envio de tramas

Depois da confirmação de que a ligação foi estabelecida, pode-se começar a enviar tramas de informação. Os pacotes a serem enviados, através destas tramas, serão referidos como P. As tramas de informação têm o seguinte formato:

- F – flag
- A – Campo de Endereço
- C – Campo de Controlo:
- $BCC_1$  – Campo de Proteção (cabeçalho)
- $D_1...D_n$  – Campo de informação (contém pacote gerado pela aplicação)

- $BCC_2$  – Campo de Proteção (dados)
- F – flag

O  $BCC_2$  é calculado através da aplicação da operação XOR entre todos os bytes do pacote P, um *foldright* com a operação XOR. Isto é feito através da função:

```
1 uint8_t bcc2_builder(uint8_t msg[], unsigned int msg_size);
```

Após a construção do  $BCC_2$  procede-se a fazer stuffing do pacote P, resultando no campo  $D_1...D_n$ , e do próprio  $BCC_2$ . Esta operação é importante porque assegura a transparência.

Agora reúnem-se as condições para enviar a trama. Após o envio desta, o transmitter fica à espera de uma mensagem de resposta do receiver que será uma trama (S) com  $C = RR$  se o receiver aceitar a trama e quiser a próxima e  $C = REJ$  se o receiver rejeitar a trama e precisar que o transmitter a reenvie.

O transmitter tem incorporado um timeout que ao enviar a trama espera pela resposta de receiver, se ao fim de 3 segundos não obtiver resposta reenvia a trama, este processo é repetido 3 vezes sendo que o programa termina no 3º timeout.

```
1 int llwrite(int fd, uint8_t *buffer, int length);
```

### 5.3 Receção de tramas

O receiver fica à espera de que lhe sejam enviadas tramas de informação (I) no formato referido anteriormente, e à medida que vai recebendo dados vai validando-os através de uma máquina de estados.

Alguns comportamentos importantes a salientar são os comportamentos na receção dos BCC. No caso de erro de  $BCC_1$  o receiver descarta todos os dados dessa trama e não envia resposta, o que leva o transmitter a dar time-out e a reenviar a trama.

Após a receção de todos os dados, que é detetada com a recção da Flag final, o campo de informação +  $BCC_2$  são *destuffed* e a partir do campo de informação é computado um segundo  $BCC_2$ .

Na comparação do  $BCC_2$  recebido com o computado:

- Inválido
  - Se não for uma trama duplicada é enviado REJ
  - Se for duplicada, não são guardados os dados e é enviado RR
- Válido
  - Se não for uma trama duplicada, são guardados os dados e é enviado RR
  - Se for duplicada, não são guardados os dados e é enviado RR

De notar que as mensagens de REJ e RR são montadas com o valor de  $R=N(r)$  que depois é usado para comparação com o valor  $S=N(s)$  recebido pelo campo C das tramas de Informação, influenciando as validações do cabeçalho.

```
1 int llread(int fd, uint8_t *buffer);
```

### 5.4 Terminação da ligação

Para encerrar a ligação, o transmitter manda uma trama de (S) com  $C = DISC$ , o receiver quando recebe DISC manda ele próprio um DISC e o transmitter por fim manda UA, sinalizando assim a terminação correta da ligação. Estas operações também estão protegidas por time-outs similares aos mencionados anteriormente.

```
1 int llclose(int fd, type_t type);
```

## 6 Protocolo de aplicação

O protocolo de aplicação implementado teve como objetivo a transferência de um ficheiro fazendo uso da interface fornecida pela camada de ligação lógica. O protocolo implementado tem as seguintes características:

### 6.1 Tipos de pacotes

A aplicação faz uso de dois tipos de pacotes: pacotes de dados, usados para transferir pedaços de ficheiro, e pacotes de controlo, usados para sinalizar o início e fim de uma transferência.

A seguinte função definida em *aplic.c* monta um pacote de controlo:

```
1 static uint8_t* get_control_packet(off_t file_size, char *file_name, int
    file_name_size, int *length);
```

Nesta função, é gerado o pacote de controlo START. Ela recebe como parâmetros o tamanho e o nome do ficheiro e retorna um pointer para um pacote de controlo assim como um retorno por parâmetro do tamanho desse pacote. O pacote de controlo tem a seguinte composição:

- C – 1 byte de campo de controlo (2: START; 3: END)
- TLV (Type, Length, Value) – os necessários, neste caso 2 (tamanho e nome de ficheiro)
  - T – 1 byte que indica o tipo de parâmetro (0: tamanho de ficheiro; 1: nome do ficheiro)
  - L – 1 byte que indica o tamanho do campo seguinte, V
  - V – valor do parâmetro que ocupa o número de bytes indicado em L

Por outro lado, os pacotes de dados têm a seguinte composição:

- C – 1 byte de controlo (1: DADOS)
- N – 1 byte de número de sequência em módulo 255
- $L_2L_1$  – 2 bytes com o número de bytes (K) do campo de dados
- $P_1...P_k$  – K bytes de dados

Estes pacotes são montados e enviados na seguinte função definida em *aplic.c*:

```
1 static int send_packaged_file(int fd_serial_port, int fd_file);
```

### 6.2 Envio de ficheiro

O envio de um ficheiro é efetuado recorrendo à seguinte função:

```
1 int send_file(int porta, char *path, int path_size, char *file_name);
```

Esta abre o ficheiro a enviar e abre uma conexão usando *llopen()*, da camada de ligação lógica, na porta série que recebeu como argumento. De seguida, envia o pacote de controlo START, previamente montado, usando *llwrite()*. Após isso torna-se necessário enviar o ficheiro repartido por vários pacotes de dados. Para tal recorre-se à seguinte função já referida anteriormente:

```
1 static int send_packaged_file(int fd_serial_port, int fd_file);
```

Ela vai enviando os pacotes usando *llwrite()* à medida que lê pedaços de ficheiro usando *read()*. Nestes pacotes, como já referido na secção Tipos de pacotes, vão também outras informações.

Após o ficheiro ser enviado na totalidade, é então enviado o pacote de controlo END e após isso basta fechar a conexão usando *llclose()* e fechar o ficheiro aberto usando *close()*.

### 6.3 Receção de ficheiro

A receção de um ficheiro é efetuado recorrendo à seguinte função:

```
1 int receive_file(int porta);
```

Esta função começa por abrir uma conexão na porta série usando *llopen()* e de seguida vai começar a ler pacotes usando *lread()*.

Numa execução ela começará por receber o pacote de controlo START. Após isto, já sabe o nome do ficheiro e o tamanho do mesmo. Com esta informação cria um ficheiro e abre-o em modo *append*. De seguida vai recebendo pacotes de dados dos quais retira as porções de ficheiro e acrescenta-as ao final do ficheiro criado anteriormente. No final, irá receber um pacote de controlo END e saberá que a transmissão chegou ao fim podendo fechar o ficheiro usando *close()* e a conexão da porta série usando *llclose()*.

## 7 Validação

De modo a validar o correto funcionamento dos protocolos de ligação de dados e de aplicação foram efetuados múltiplos testes, tanto em ambiente simulado usando o utilitário da linha de comandos *socat* e o ficheiro de exemplo fornecido, *cable.c*, como em ambiente laboratorial no laboratório I321 na FEUP. Em ambiente laboratorial foi testado o envio de diversos ficheiro de 4 modos distintos:

- Sem interrupções e sem interferências.
- Com interrupções mas sem interferências.
- Sem interrupções mas com interferências.
- Com interrupções e com interferências.

Em todas as situações, os protocolados provaram ser robustos uma vez que garantiram o correto envio sem erros dos ficheiros enviados. Esta certeza foi garantida através do uso do utilitário da linha de comandos *diff* que foi utilizado para comparar na máquina recetora o ficheiro reebido com uma cópia que a mesma já tinha do ficheiro.

Foi também testada a ordem de execução dos programas: *receiver* seguido pelo *sender* e *sender* seguido pelo *receiver* obtendo em ambas as situações um correto funcionamento do envio de ficheiros.

## 8 Eficiência do protocolo de ligação de dados

De forma a avaliar a eficiência do protocolo de ligação de dados implementado, foi efetuada uma caracterização estatística da eficiência (S) em função de *a* e do *FER* (Frame error ratio). Os valores resultantes foram então comparados com valores teóricos.

Os valores que deram origem aos gráficos que se seguem podem ser lidos em anexo.

### 8.1 $S = f(a)$

De modo a variar experimentalmente o valor de *a*, foi variado o valor de  $T_{prop}$  introduzindo um ligeiro atrasado usando *usleep()*. Isto é possível dado que  $a = T_{prop}/T_f$ . Foi obtido o seguinte gráfico que mostra que os valores medidos seguem a tendência teórica:

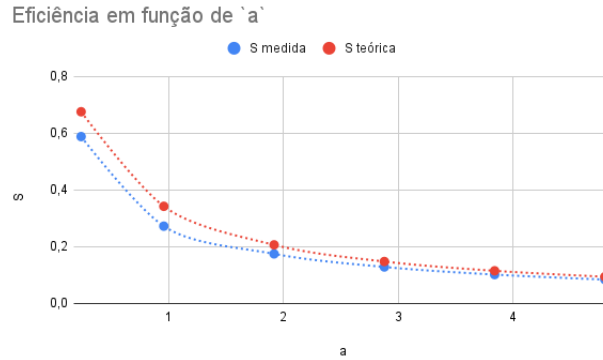


Figura 1: Eficiência em função de a.

## 8.2 $S = g(\text{FER})$

De modo a variar experimentalmente o valor de FER foram introduzidos erros fictícios nos  $BCC_1$  e  $BCC_2$  de acordo com o FER pretendido e obteve-se o seguinte gráfico que mostra que os valores medidos seguem a tendência teórica:

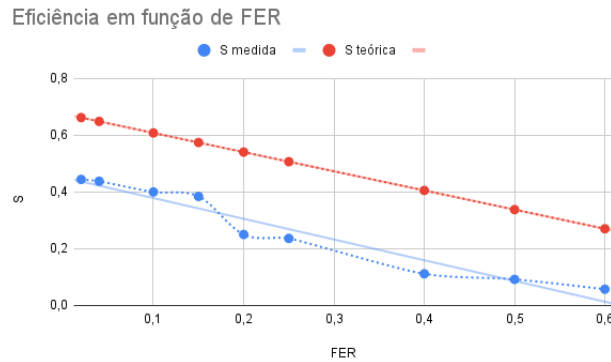


Figura 2: Eficiência em função de FER.

## 9 Conclusões

Foram implementados em C dois protocolos robustos, ligação de dados e aplicação, para transferir ficheiros entre computadores usando a porta série. Foi garantida independência entre camadas e a correta implementação do mecanismo de *Stop-and-Wait* para controlo de erros no protocolo de ligação de dados. Foi ainda escrito este relatório que inclui uma análise de eficiência. Assim, foram cumpridos todos os objetivos deste projeto.

A realização deste projeto permitiu lidar na prática com os detalhes abordados teoricamente nas aulas que de outra forma nos passariam despercebidos. Assim sendo, a sua conceção demonstrou ser uma forma de estudo imersiva dos conteúdos lecionados em *Redes de Computadores*.



## 10 Anexos

### 10.1 Código Fonte

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #include "aplic.h"
6
7
8 int main(int argc, char** argv) {
9     if (argc != 4) {
10         printf("Usage:\tsender SerialPort Path NameToGive\n\tex: sender <i> <path>
11             <name>\n");
12         return -1;
13     }
14
15     int porta = atoi(argv[1]);
16
17     if (send_file(porta, argv[2], strlen(argv[2]), argv[3]) < 0) {
18         return -1;
19     }
20
21     return 0;
22 }
```

Listing 1: sender.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "aplic.h"
5
6
7 int main(int argc, char** argv) {
8     if (argc != 2) {
9         printf("Usage:\treceiver SerialPort\n\tex: receiver <i>\n");
10         return -1;
11     }
12
13     int porta = atoi(argv[1]);
14
15     if (receive_file(porta) < 0 ) {
16         return -1;
17     }
18
19     return 0;
20 }
```

Listing 2: receiver.c

```
1 int send_file(int porta, char *path, int path_size, char *file_name);
2
3 int receive_file(int porta);
```

Listing 3: aplic.h

```
1 #include <stdint.h>
2
3 #define BAUDRATE B38400
4 #define _POSIX_SOURCE 1 /* POSIX compliant source */
5 #define FALSE 0
6 #define TRUE 1
7
```

```

8 #define DATA_PACKET_MAX_SIZE 1000
9
10 typedef enum type {
11     TRANSMITTER,
12     RECEIVER
13 } type_t;
14
15 int llopen(int porta, type_t type);
16
17 int llclose(int fd, type_t type);
18
19 int llwrite(int fd, uint8_t *buffer, int length);
20
21 int llread(int fd, uint8_t *buffer);

```

Listing 4: linklayer.h

```

1 #include "aplic.h"
2 #include "linklayer.h"
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <termios.h>
7 #include <stdio.h>
8 #include <string.h>
9 #include <strings.h>
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include <stdint.h>
13
14 #define CONTROL_PACKET_MAX_SIZE 500
15 #define PACKET_MAX_SIZE (CONTROL_PACKET_MAX_SIZE > DATA_PACKET_MAX_SIZE ?
    CONTROL_PACKET_MAX_SIZE : DATA_PACKET_MAX_SIZE)
16 #define FILE_NAME_MAX_SIZE 255
17
18 #define C_DATA 0x1
19 #define C_START 0x2
20 #define C_END 0x3
21
22 #define N(seq) ((seq) % 255)
23
24 #define L1(K) ((K) & 0b11111111)
25 #define L2(K) (((K) >> 8) & 0b11111111)
26 #define K(L1,L2) (256*(L2)+(L1))
27
28 #define T_FILE_SIZE 0x0
29 #define T_FILE_NAME 0x1
30
31
32 static off_t get_file_size(int fd) {
33     struct stat s;
34     if (fstat(fd, &s) == -1) {
35         return -1;
36     }
37
38     return s.st_size;
39 }
40
41 static uint8_t* get_control_packet(off_t file_size, char *file_name, int
    file_name_size, int *length) {
42     uint8_t* control_packet = malloc(CONTROL_PACKET_MAX_SIZE);
43     if (control_packet == NULL) {
44         return NULL;
45     }
46

```

```

47     size_t i = 0;
48
49     control_packet[i++] = C_START;
50     control_packet[i++] = T_FILE_SIZE; // T1
51     control_packet[i++] = sizeof(off_t); // L1
52
53     memcpy(&control_packet[i], &file_size, sizeof(off_t)); // V1
54     i += sizeof(off_t);
55
56     control_packet[i++] = T_FILE_NAME; // T2
57
58     control_packet[i++] = (uint8_t)file_name_size; // L2
59
60     memcpy(&control_packet[i], file_name, file_name_size); // V2
61
62     *length = i + file_name_size;
63     return control_packet;
64 }
65
66 static int send_packaged_file(int fd_serial_port, int fd_file) {
67     uint8_t *data_packet = malloc(DATA_PACKET_MAX_SIZE);
68     if (data_packet == NULL) {
69         return -1;
70     }
71
72     uint8_t sequence_number = 0;
73     data_packet[0] = C_DATA;
74
75     while (1) {
76         data_packet[1] = sequence_number;
77         sequence_number = (sequence_number+1) % 255;
78
79         ssize_t num = read(fd_file, &data_packet[4], DATA_PACKET_MAX_SIZE-4);
80
81         if (num == -1) {
82             free(data_packet);
83             return -1;
84         } else if (num == 0) {
85             break;
86         } else {
87             data_packet[2] = L2(num);
88             data_packet[3] = L1(num);
89
90             if (llwrite(fd_serial_port, data_packet, num+4) < 0) {
91                 free(data_packet);
92                 return -1;
93             }
94         }
95     }
96
97     free(data_packet);
98     return 0;
99 }
100
101 int send_file(int porta, char *path, int path_size, char *file_name) {
102     int file_name_size = strlen(file_name);
103     if (file_name_size > FILE_NAME_MAX_SIZE) {
104         printf("File name too big.\n");
105         return -1;
106     }
107
108     int fd_file;
109     if ((fd_file = open(path, O_RDONLY)) < 0) {
110         printf("File not found.\n");

```

```

111     return -1;
112 }
113
114 off_t file_size = 0;
115 if ((file_size = get_file_size(fd_file)) < 0) {
116     close(fd_file);
117     return -1;
118 }
119
120 int fd_serial_port;
121 if ((fd_serial_port = llopen(porta, TRANSMITTER)) < 0) {
122     close(fd_file);
123     return -1;
124 }
125
126 uint8_t* control_packet = NULL;
127 int control_packet_size = 0;
128 if ((control_packet = get_control_packet(file_size, file_name, file_name_size,
129     &control_packet_size)) == NULL) {
130     close(fd_file);
131     llclose(fd_serial_port, TRANSMITTER);
132     return -1;
133 }
134
135 // Control packet start
136 if (llwrite(fd_serial_port, control_packet, control_packet_size) < 0) {
137     free(control_packet);
138     close(fd_file);
139     llclose(fd_serial_port, TRANSMITTER);
140     return -1;
141 }
142
143 if (send_packaged_file(fd_serial_port, fd_file) != 0) {
144     free(control_packet);
145     close(fd_file);
146     llclose(fd_serial_port, TRANSMITTER);
147     return -1;
148 }
149
150 // Control packet end
151 control_packet[0] = C_END;
152 if (llwrite(fd_serial_port, control_packet, control_packet_size) < 0) {
153     free(control_packet);
154     close(fd_file);
155     llclose(fd_serial_port, TRANSMITTER);
156     return -1;
157 }
158
159 free(control_packet);
160 close(fd_file);
161 llclose(fd_serial_port, TRANSMITTER);
162 return 0;
163 }
164
165 int receive_file(int porta) {
166     int fd_serial_port;
167     if ((fd_serial_port = llopen(porta, RECEIVER)) < 0) {
168         return -1;
169     }
170
171     uint8_t *packet = malloc(PACKET_MAX_SIZE);
172     if (packet == NULL) {
173         llclose(fd_serial_port, RECEIVER);
174         return -1;

```

```

174     }
175
176     int fd_file_to_write = -1;
177     int sequence_number = 0;
178     off_t file_size = 0;
179
180     int not_end_packet = TRUE;
181     while (not_end_packet) {
182         int packet_size = 0;
183         if ((packet_size = llread(fd_serial_port, packet)) < 0) {
184             free(packet);
185             llclose(fd_serial_port, RECEIVER);
186             return -1;
187         }
188
189         uint8_t control_field = packet[0];
190         switch (control_field) {
191             case C_DATA:
192                 if (fd_file_to_write == -1) { // Control start packet didn't arrive
yet
193                     break;
194                 }
195
196                 int N = packet[1];
197
198                 if (N == sequence_number) {
199                     int num_octets = K(packet[3], packet[2]);
200                     if (write(fd_file_to_write, &packet[4], num_octets) == -1) {
201                         free(packet);
202                         llclose(fd_serial_port, RECEIVER);
203                         return -1;
204                     }
205                     sequence_number = (sequence_number + 1) % 255;
206                 } else {
207                     // wrong sequence number
208                     printf("Wrong packet sequence number. Expected: %d ; Got: %d\n",
sequence_number, N);
209                     free(packet);
210                     llclose(fd_serial_port, RECEIVER);
211                     return -1;
212                 }
213
214                 break;
215
216             case C_START:;
217                 int i = 1;
218                 while (i < packet_size) {
219                     uint8_t T = packet[i++];
220                     uint8_t L = packet[i++];
221
222                     if (T == T_FILE_SIZE) {
223                         memcpy(&file_size, &packet[i], L);
224
225                     } else if (T == T_FILE_NAME) {
226                         char file_name[FILE_NAME_MAX_SIZE];
227                         memcpy(file_name, &packet[i], L);
228
229                         fd_file_to_write = open(file_name, O_WRONLY | O_APPEND |
O_CREAT, 0644);
230                         if (fd_file_to_write == -1) {
231                             free(packet);
232                             llclose(fd_serial_port, RECEIVER);
233                             return -1;
234                         }

```

```

235         } else {
236             printf("ERROR: not supposed to reach this\n");
237         }
238     }
239
240     i += L;
241 }
242
243 break;
244
245 case C_END:
246     if (fd_file_to_write != -1) {
247         close(fd_file_to_write);
248         not_end_packet = FALSE;
249     }
250     // else Control start packet didn't arrive yet, so wait for it
251
252     break;
253
254 default:
255     break; // invalid control field (ignore packet)
256 }
257 }
258
259 if (llclose(fd_serial_port, RECEIVER) < 0) {
260     free(packet);
261     return -1;
262 }
263
264 free(packet);
265 return 0;
266 }

```

Listing 5: aplic.c

```

1 #include "linklayer.h"
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <termios.h>
6 #include <stdio.h>
7 #include <string.h>
8 #include <strings.h>
9 #include <stdlib.h>
10 #include <unistd.h>
11 #include <signal.h>
12 #include <stdint.h>
13
14 #define FLAG 0x7E
15 #define ESC 0x7D
16 #define A 0x03
17 #define C_SET 0x03
18 #define C_DISC 0x0B
19 #define C_UA 0x07
20
21 #define C_RR(r) (0x05 | (((r) << 7) & 0x80))
22 #define C_REJ(r) (0x01 | (((r) << 7) & 0x80))
23 #define C_I(s) ((s) << 6)
24
25 #define CONTROL_SIZE 5
26
27 #define STUFFER 0x20
28
29 #define TIME_OUT_TIME 3
30 #define MAX_NO_TIMEOUT 3

```

```

31
32 #define HEADER_AND_TAIL_SIZE 10 // more than enough
33
34 typedef enum control_frame_type {
35     SET,
36     DISC,
37     UA,
38     RR,
39     REJ
40 } control_frame_type_t;
41
42 typedef enum state_sv_frame {
43     START,
44     FLAG_RCV,
45     A_RCV,
46     C_RCV,
47     RR_RCV,
48     REJ_RCV,
49     BCC_OK,
50     STOP
51 } state_sv_frame_t;
52
53 typedef enum state_info_rcv {
54     I_START,
55     I_GOT_FLAG,
56     I_IGNORE,
57     I_GOT_A,
58     I_GOT_C,
59     I_GOT_BCC1,
60     I_DATA_COLLECTION,
61     I_GOT_ESC,
62     I_GOT_END_FLAG,
63     I_TEST_DUP_RR,
64     I_TEST_DUP_REJ,
65     I_RR_DONT_STORE,
66     I_RR_STORE,
67     I_REJ,
68     I_STOP
69 } state_info_rcv_t;
70
71 static struct termios oldtio;
72 static volatile int g_count = 0;
73
74 static uint8_t S = 0;
75 static uint8_t next_S = 0;
76 static uint8_t R = 0;
77
78 static void control_frame_builder(control_frame_type_t cft, uint8_t msg[]){
79     msg[0] = FLAG;
80     msg[1] = A;
81
82     switch (cft) {
83     case SET:
84         msg[2] = C_SET;
85         break;
86
87     case DISC:
88         msg[2] = C_DISC;
89         break;
90
91     case UA:
92         msg[2] = C_UA;
93         break;
94

```

```

95     case RR:
96         msg[2] = C_RR(R);
97         break;
98
99     case REJ:
100         msg[2] = C_REJ(R);
101         break;
102
103     default:
104         break;
105 }
106
107 msg[3] = msg[1] ^ msg[2];
108 msg[4] = FLAG;
109 }
110
111 static int update_state_rr_rej(state_sv_frame_t *state, uint8_t byte) {
112
113     if (state == NULL) {
114         return 1;
115     }
116     switch (*state) {
117
118     case START:
119         if (byte == FLAG) *state = FLAG_RCV;
120         else *state = START;
121         break;
122
123     case FLAG_RCV:
124         if (byte == A) *state = A_RCV;
125         else *state = START;
126         break;
127
128     case A_RCV:
129         if ((byte & 0x0F) == 0x05) *state = RR_RCV;
130         else if ((byte & 0x0F) == 0x01) *state = REJ_RCV;
131         else *state = START;
132         next_S = (byte >> 7) & 0x01;
133         break;
134
135     case RR_RCV:
136         if (byte == (A^C_RR(next_S))) *state = BCC_OK;
137         else if (byte == FLAG) *state = FLAG_RCV;
138         else *state = START;
139         break;
140
141     case REJ_RCV:
142         if (byte == (A^C_REJ(next_S))) *state = BCC_OK;
143         else if (byte == FLAG) *state = FLAG_RCV;
144         else *state = START;
145         break;
146
147     case BCC_OK:
148         if (byte == FLAG) *state = STOP;
149         else *state = START;
150         break;
151
152     case STOP:
153         break;
154
155     default:
156         printf("ERROR: not supposed to reach this\n");
157         break;
158 }

```



```

159     return 0;
160 }
161
162 static int update_state_set_ua(uint8_t c, state_sv_frame_t *state, uint8_t byte) {
163     if (state == NULL) {
164         return 1;
165     }
166
167     switch (*state) {
168     case START:
169         if (byte == FLAG) {
170             *state = FLAG_RCV;
171         }
172         break;
173
174     case FLAG_RCV:
175         if (byte == A) {
176             *state = A_RCV;
177         } else if (byte != FLAG) {
178             *state = START;
179         }
180         break;
181
182     case A_RCV:
183         if (byte == c) {
184             *state = C_RCV;
185         } else if (byte == FLAG) {
186             *state = FLAG_RCV;
187         } else {
188             *state = START;
189         }
190         break;
191
192     case C_RCV:
193         if (byte == (A^c)) {
194             *state = BCC_OK;
195         } else if (byte == FLAG) {
196             *state = FLAG_RCV;
197         } else {
198             *state = START;
199         }
200         break;
201
202     case BCC_OK:
203         if (byte == FLAG) {
204             *state = STOP;
205         } else {
206             *state = START;
207         }
208         break;
209
210     case STOP:
211         break;
212
213     default:
214         printf("ERROR: not supposed to reach this\n");
215         break;
216 }
217
218 return 0;
219 }
220
221 }
222

```

```

223 static int update_state_info_rcv(state_info_rcv_t *state, uint8_t byte){
224
225     switch (*state){
226         case (I_START):
227             if (byte == FLAG) *state = I_GOT_FLAG;
228             else *state = I_IGNORE;
229             break;
230
231         case (I_GOT_FLAG):
232             if (byte == FLAG) *state = I_GOT_FLAG;
233             else if (byte == A) *state = I_GOT_A;
234             else *state = I_IGNORE;
235             break;
236
237         case (I_IGNORE):
238             if (byte == FLAG) *state = I_GOT_FLAG;
239             else *state = I_IGNORE;
240             break;
241
242         case (I_GOT_A):
243             if (byte == C_I(R)) *state = I_GOT_C;
244             else *state = I_IGNORE;
245             break;
246
247         case (I_GOT_C):
248             if (byte == (C_I(R)^A)) *state = I_GOT_BCC1;
249             else *state = I_IGNORE;
250             break;
251
252         case (I_GOT_BCC1):
253             if (byte == ESC) *state = I_GOT_ESC;
254             else *state = I_DATA_COLLECTION;
255             break;
256
257         case (I_DATA_COLLECTION):
258             if (byte == FLAG) *state = I_GOT_END_FLAG;
259             else if (byte == ESC) *state = I_GOT_ESC;
260             else *state = I_DATA_COLLECTION;
261             break;
262
263         case (I_GOT_ESC):
264             *state = I_DATA_COLLECTION;
265             break;
266
267         case (I_GOT_END_FLAG):
268             if (byte) *state = I_TEST_DUP_RR; // byte = is bcc2 valid ?
269             else *state = I_TEST_DUP_REJ;
270             break;
271
272         case (I_TEST_DUP_RR):
273             if (byte) *state = I_RR_DONT_STORE; // byte = is dup ?
274             else *state = I_RR_STORE;
275             break;
276
277         case (I_TEST_DUP_REJ):
278             if (byte) *state = I_RR_DONT_STORE; // byte = is dup ?
279             else *state = I_REJ;
280             break;
281
282         case (I_RR_DONT_STORE):
283             *state = I_START;
284             break;
285
286         case (I_RR_STORE):

```

```

287         *state = I_STOP;
288         break;
289
290     case (I_REJ):
291         *state = I_START;
292         break;
293
294     case (I_STOP):
295         break;
296
297     default:
298         break;
299 }
300
301 return 0;
302 }
303
304 static void time_out() {
305     printf("alarm # %d\n", g_count);
306     g_count++;
307 }
308
309 static int setup_alarm() {
310     struct sigaction new;
311     sigset_t smask;
312
313     if (sigemptyset(&smask)==-1) {
314         perror ("sigsetfunctions");
315         return 1;
316     }
317
318     new.sa_handler = time_out;
319     new.sa_mask = smask;
320     new.sa_flags = 0;
321
322     if (sigaction(SIGALRM, &new, NULL) == -1) {
323         perror ("sigaction");
324         return 1;
325     }
326
327     return 0;
328 }
329
330 static int common_open(int porta) {
331     int fd = -1;
332     struct termios newtio;
333
334     /*
335     Open serial port device for reading and writing and not as controlling tty
336     because we don't want to get killed if linenoise sends CTRL-C.
337     */
338
339     char buffer[20];
340     if (sprintf(buffer, "/dev/ttyS%d", porta) < 0) {
341         perror("");
342         return -1;
343     }
344
345
346     fd = open(buffer, O_RDWR | O_NOCTTY );
347     if (fd < 0) {
348         perror(buffer);
349         return -1;
350     }

```

```

351
352     if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings */
353         perror("tcgetattr");
354         return -1;
355     }
356
357     bzero(&newtio, sizeof(newtio));
358     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
359     newtio.c_iflag = IGNPAR;
360     newtio.c_oflag = 0;
361
362     /* set input mode (non-canonical, no echo,...) */
363     newtio.c_lflag = 0;
364
365     newtio.c_cc[VTIME]      = 0;   /* inter-character timer unused */
366     newtio.c_cc[VMIN]       = 1;   /* blocking read until 1 char received */
367
368     tcflush(fd, TCIOFLUSH);
369
370     if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
371         perror("tcsetattr");
372         return -1;
373     }
374
375     printf("New termios structure set\n");
376
377     return fd;
378 }
379
380 static int common_close(int fd) {
381     if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
382         perror("tcsetattr");
383         close(fd);
384         return -1;
385     }
386
387     return close(fd);
388 }
389
390 static void R_invert(){
391     R = ((!R) << 7) >> 7;
392 }
393
394 int llopen(int porta, type_t type) {
395     state_sv_frame_t state;
396     int fd = common_open(porta);
397     if (fd < 0) {
398         printf("Failed to open serial port.\n");
399         return -1;
400     }
401
402     printf("%d opened fd\n", fd);
403
404     uint8_t set[CONTROL_SIZE];
405     control_frame_builder(SET, set);
406
407     uint8_t ua[CONTROL_SIZE];
408     control_frame_builder(UA, ua);
409
410     if (setup_alarm() != 0) {
411         common_close(fd);
412         return -1;
413     }
414     g_count = 0;

```

```

415
416 switch (type) {
417     case TRANSMITTER::
418         int ua_received = FALSE;
419         int res;
420         while (g_count < MAX_NO_TIMEOUT && !ua_received) {
421             state = START;
422
423             res = write(fd, set, CONTROL_SIZE * sizeof(uint8_t));
424             if (res == -1) {
425                 printf("llopen() -> write() TRANSMITTER error\n");
426                 common_close(fd);
427                 return -1;
428             }
429             printf("SET sent.\n");
430             printf("%d bytes written\n", res);
431
432             alarm(TIME_OUT_TIME);
433
434             int timed_out = FALSE;
435             while (!timed_out && state != STOP) {
436                 uint8_t byte_read = 0;
437
438                 res = read(fd, &byte_read, 1);
439                 if (res == 1) {
440                     if (update_state_set_ua(C_UA, &state, byte_read) != 0) {
441                         common_close(fd);
442                         alarm(0);
443                         return -1;
444                     }
445                     ua_received = (state==STOP);
446
447                 } else if (res == -1) {
448                     timed_out = TRUE;
449
450                 } else {
451                     printf("DEBUG: not supposed to happen\n");
452                 }
453             }
454
455             alarm(0);
456         }
457
458         if (ua_received) {
459             printf("UA received.\n");
460             printf("ACK\n");
461         } else {
462             common_close(fd);
463             return -1;
464         }
465
466         break;
467
468     case RECEIVER:
469         alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
470         state = START;
471         while (state != STOP) {
472             uint8_t byte_read = 0;
473             res = read(fd, &byte_read, 1);
474
475             if (res == 1) {
476                 if (update_state_set_ua(C_SET, &state, byte_read) != 0) {
477                     common_close(fd);
478                     alarm(0);

```

```

479         return -1;
480     }
481     } else if (res == -1) {
482         if (g_count > 0) {
483             printf("llopen timedout\n");
484         } else {
485             printf("llopen() -> read() RECEIVER error\n");
486         }
487         common_close(fd);
488         alarm(0);
489         return -1;
490     } else {
491         printf("DEBUG: not supposed to happen\n");
492     }
493 }
494
495 printf("SET received.\n");
496 if (write(fd, ua, CONTROL_SIZE) < 0) {
497     printf("llopen() -> write() RECEIVER error\n");
498     common_close(fd);
499     alarm(0);
500     return -1;
501 }
502
503 printf("UA sent.\n");
504 printf("ACK\n");
505 alarm(0);
506 break;
507 }
508
509 return fd;
510 }
511
512 int llclose(int fd, type_t type) {
513     state_sv_frame_t state;
514     uint8_t disc[CONTROL_SIZE];
515     control_frame_builder(DISC, disc);
516
517     uint8_t ua[CONTROL_SIZE];
518     control_frame_builder(UA, ua);
519
520     int res = 0;
521     int disc_received = FALSE;
522     g_count = 0;
523
524     switch (type) {
525     case TRANSMITTER:
526         while (g_count < MAX_NO_TIMEOUT && !disc_received) {
527             state = START;
528
529             res = write(fd, disc, CONTROL_SIZE * sizeof(uint8_t));
530             if (res == -1) {
531                 printf("llclose() -> write() TRANSMITTER error\n");
532                 return -1;
533             }
534             printf("DISC sent.\n");
535             printf("%d bytes written\n", res);
536
537             alarm(TIME_OUT_TIME);
538
539             int timed_out = FALSE;
540             while (!timed_out && state != STOP) {
541                 uint8_t byte_read = 0;
542

```

```

543         res = read(fd, &byte_read, 1);
544         if (res == 1) {
545             if (update_state_set_ua(C_DISC, &state, byte_read) != 0) {
546                 common_close(fd);
547                 alarm(0);
548                 return -1;
549             }
550             disc_received = (state==STOP);
551
552         } else if (res == -1) {
553             timed_out = TRUE;
554
555         } else {
556             printf("DEBUG: not supposed to happen\n");
557         }
558     }
559
560     alarm(0);
561 }
562
563 if (disc_received) {
564     printf("DISC received.\n");
565     res = write(fd, ua, CONTROL_SIZE * sizeof(uint8_t));
566     printf("UA sent.\n");
567 } else {
568     printf("DISC not received.\n");
569 }
570
571 break;
572
573 case RECEIVER:
574     alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
575     state = START;
576     while (state != STOP) {
577         uint8_t byte_read = 0;
578
579         res = read(fd, &byte_read, 1);
580         if (res == 1) {
581             if (update_state_set_ua(C_DISC, &state, byte_read) != 0) {
582                 common_close(fd);
583                 alarm(0);
584                 return -1;
585             }
586             disc_received = (state==STOP);
587         } else if (res == -1) {
588             if (g_count > 0) {
589                 printf("llclose timeout\n");
590             } else {
591                 printf("llclose() -> read() RECEIVER error\n");
592             }
593             alarm(0);
594             common_close(fd);
595             return -1;
596             break;
597         } else {
598             printf("DEBUG: not supposed to happen\n");
599         }
600     }
601
602     alarm(0);
603
604     if (res != -1) {
605         printf("DISC received.\n");
606         int ua_received = FALSE;

```

```

607     g_count = 0;
608     while (g_count < MAX_NO_TIMEOUT && !ua_received) {
609         state = START;
610
611         res = write(fd, disc, CONTROL_SIZE * sizeof(uint8_t));
612         if (res == -1) {
613             printf("llclose() -> write() RECEIVER error\n");
614             alarm(0);
615             return -1;
616         }
617         printf("DISC sent.\n");
618         printf("%d bytes written\n", res);
619
620         alarm(TIME_OUT_TIME);
621
622         int timed_out = FALSE;
623         while (!timed_out && state != STOP) {
624             uint8_t byte_read = 0;
625
626             res = read(fd, &byte_read, 1);
627             if (res == 1) {
628                 if (update_state_set_ua(C-UA, &state, byte_read) != 0) {
629                     common_close(fd);
630                     alarm(0);
631                     return -1;
632                 }
633                 ua_received = (state==STOP);
634
635             } else if (res == -1) {
636                 timed_out = TRUE;
637
638             } else {
639                 printf("DEBUG: not supposed to happen\n");
640             }
641         }
642
643         alarm(0);
644     }
645
646     if (ua_received) {
647         printf("UA received.\n");
648     }
649 }
650
651 break;
652 }
653
654 res = common_close(fd);
655
656 return res;
657 }
658
659 int message_stuffing(uint8_t in_msg[], unsigned int in_msg_size, uint8_t **
out_msg){
660
661     int size_counter = 0;
662     *out_msg = malloc(in_msg_size*2);
663
664     uint8_t * out_message = * out_msg;
665
666     for (int i = 0; i < in_msg_size; i++){
667         switch (in_msg[i]){
668             case FLAG:
669                 out_message[size_counter++] = ESC;

```



```

670         out_message[size_counter++] = FLAG ^ STUFFER;
671         break;
672     case ESC:
673         out_message[size_counter++] = ESC;
674         out_message[size_counter++] = ESC ^ STUFFER;
675         break;
676     default:
677         out_message[size_counter++] = in_msg[i];
678         break;
679     }
680 }
681 return size_counter;
682 }
683
684 int message_destuffer(uint8_t in_msg[], unsigned int in_msg_size, uint8_t **
out_msg){
685
686     int size_counter = 0;
687     *out_msg = malloc(in_msg_size);
688
689     uint8_t * out_message = * out_msg;
690
691     for (int i = 0; i < in_msg_size; i++){
692         if (in_msg[i] == ESC){
693             out_message[size_counter] = (in_msg[++i] ^ STUFFER);
694         } else {
695             out_message[size_counter] = in_msg[i];
696         }
697         size_counter++;
698     }
699
700     return size_counter;
701 }
702
703 uint8_t bcc2_builder(uint8_t msg[], unsigned int msg_size){
704
705     if (msg_size == 1) {
706         return msg[0];
707     } else if ( msg_size < 0) {
708         return 0;
709     }
710
711     uint8_t ret = msg[0];
712
713     for (int i = 1; i < msg_size; i++){
714         ret ^= msg[i];
715     }
716
717     return ret;
718 }
719
720 int llwrite(int fd, uint8_t * buffer, int length){
721
722     int write_successful = 0;
723     int ret = 0;
724     uint8_t bcc2 = bcc2_builder(buffer, length);
725     uint8_t *unstuffed_msg = malloc((length+1) * sizeof(uint8_t));
726     memcpy(unstuffed_msg, buffer, length);
727     unstuffed_msg[length] = bcc2;
728     uint8_t *stuffed_msg = NULL;
729     int stuffed_msg_len = message_stuffing(unstuffed_msg, length+1, &stuffed_msg);
730     free(unstuffed_msg);
731     int total_msg_len = stuffed_msg_len + CONTROL_SIZE;
732     uint8_t *info_msg = malloc(total_msg_len);

```

```

733 info_msg[0] = FLAG;
734 info_msg[1] = A;
735 info_msg[2] = C_I(S);
736 info_msg[3] = A ^ C_I(S);
737 memcpy(&(info_msg[4]), stuffed_msg, stuffed_msg_len);
738 info_msg[total_msg_len-1] = FLAG;
739
740
741 setup_alarm();
742 g_count = 0;
743
744 while(!write_successful && g_count < MAX_NO_TIMEOUT) {
745
746     printf("----- TASK: WRITING MESSAGE\n");
747
748     if (write(fd, info_msg, total_msg_len * sizeof(uint8_t)) == -1) {
749         printf("llwrite() -> write() error\n");
750         free(info_msg);
751         free(stuffed_msg);
752         return -1;
753     }
754
755     printf("----- TASK: DONE\n");
756
757     uint8_t byte_read = 0;
758     int res = 0;
759     state_sv_frame_t state = START;
760
761     printf("----- TASK: READING REPLY\n");
762
763     alarm(TIME_OUT_TIME);
764
765     while(state != STOP){
766         res = read(fd, &byte_read, 1);
767
768         if (res == -1) {
769             write_successful = 0;
770             break;
771         }
772
773         update_state_rr_rej(&state, byte_read);
774         printf("BYTE: 0x%x; STATE: %d\n", byte_read, state);
775
776         if (state == RR_RCV) {
777             write_successful = 1;
778         } else if (state == REJ_RCV) {
779             write_successful = 0;
780         }
781     }
782
783     printf("----- TASK: DONE\n");
784 }
785
786 alarm(0);
787
788 S = next_S;
789
790 free(info_msg);
791 free(stuffed_msg);
792
793 if (g_count >= MAX_NO_TIMEOUT) {
794     ret = -1;
795 } else {
796     ret = total_msg_len;

```

```

797     }
798
799     return ret;
800 }
801
802 int llread(int fd, uint8_t *buffer) {
803
804     state_info_rcv_t state;
805     uint8_t byte_read = 0;
806     uint8_t data_read[DATA_PACKET_MAX_SIZE * 2 + HEADER_AND_TAIL_SIZE];
807     int msg_size = 0;
808
809     uint8_t *unstuffed_msg = NULL;
810     int unstuffed_size = 0;
811
812     setup_alarm();
813     g_count = 0;
814
815     state = I_START;
816
817     printf("--- NEW READ ---\n");
818
819     while (state != I_STOP){
820
821         printf("--- TRY READ ---\n");
822
823         alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
824
825         msg_size = 0;
826         int rcv_s = -1;
827
828         while (state != I_GOT_BCC1){
829             printf("PHASE 1 ; START_STATE : %d ; ", state);
830             if (read(fd, &byte_read, 1) == -1) {
831                 printf("llread() -> read() 1. error.\n");
832                 alarm(0);
833                 free(unstuffed_msg);
834                 return -1;
835             }
836
837             if (g_count) {
838                 alarm(0);
839                 free(unstuffed_msg);
840                 return -1;
841             }
842             if (state == I_GOT_C) rcv_s = byte_read >> 6;
843             update_state_info_rcv(&state, byte_read);
844             printf("END_STATE : %d\n", state);
845         }
846
847         while(state != I_GOT_END_FLAG) {
848             printf("PHASE 2 ; START_STATE : %d ; ", state);
849             if (read(fd, &byte_read, 1) == -1) {
850                 printf("llread() -> read() 2. error.\n");
851                 alarm(0);
852                 free(unstuffed_msg);
853                 return -1;
854             }
855
856             if (g_count) {
857                 alarm(0);
858                 free(unstuffed_msg);
859                 return -1;
860             }

```

```

861     update_state_info_rcv(&state, byte_read);
862     data_read[msg_size] = byte_read;
863     msg_size++;
864     printf("BYTE : 0x%x ; END_STATE : %d\n", byte_read, state);
865 }
866
867 unstuffed_size = 0;
868 uint8_t rej_msg[CONTROL_SIZE];
869 uint8_t rr_msg[CONTROL_SIZE];
870
871 free(unstuffed_msg);
872 unstuffed_size = message_destuffer(data_read, msg_size-1, &unstuffed_msg);
873
874 while (state != I_STOP && state != I_START){
875
876     printf("PHASE 3 ; START_STATE : %d ; ", state);
877
878     uint8_t res = 0;
879
880     switch(state){
881         case (I_GOT_END_FLAG):
882             res = unstuffed_msg[unstuffed_size-1] == bcc2_builder(
unstuffed_msg, unstuffed_size-1);
883             break;
884
885         case (I_TEST_DUP_REJ):
886             res = rcv_s != R;
887             break;
888
889         case (I_TEST_DUP_RR):
890             res = rcv_s != R;
891             break;
892
893         case (I_RR_DONT_STORE):
894             R_invert();
895             control_frame_builder(RR, rr_msg);
896             if (write(fd, rr_msg, CONTROL_SIZE) == -1) {
897                 printf("llread() -> write() 1. error.\n");
898                 alarm(0);
899                 free(unstuffed_msg);
900                 return -1;
901             }
902             break;
903
904         case (I_RR_STORE):
905             R_invert();
906             control_frame_builder(RR, rr_msg);
907             memcpy(buffer, unstuffed_msg, unstuffed_size-1);
908             if (write(fd, rr_msg, CONTROL_SIZE) == -1) {
909                 printf("llread() -> write() 2. error.\n");
910                 alarm(0);
911                 free(unstuffed_msg);
912                 return -1;
913             }
914             break;
915
916         case (I_REJ):
917             control_frame_builder(REJ, rej_msg);
918             if (write(fd, rej_msg, CONTROL_SIZE) == -1) {
919                 printf("llread() -> write() 3. error.\n");
920                 alarm(0);
921                 free(unstuffed_msg);
922                 return -1;
923             }

```

```

924         break;
925
926     case (I_STOP):
927         break;
928
929     case (I_START):
930         break;
931
932     default:
933         printf("NOT SUPPOSED TO REACH THIS\n");
934         break;
935     }
936     update_state_info_rcv(&state, res);
937     printf("RES : %d ; END_STATE : %d\n", res, state);
938 }
939 }
940
941 alarm(0);
942
943 free(unstuffed_msg);
944
945 return msg_size;
946 }

```

Listing 6: linklayer.c

## 10.2 Eficiência em função de 'a'

N.º total bytes	10968
N.º total bits	87744
C - Baudrate (bit/s)	38400
Packet size (bytes)	100
Tf (s)	0.020833333

Tprop (ms)	a	Tempo medido (s)	R medido (bits/s)	S medido (R/C)	S medida	S teórica
5	0.24	3.9103	22439.19904	0.584354142	0.588166106	0.675675676
		3.9146	22414.55066	0.583712257		
		3.834	22885.759	0.595983307		
		3.8542	22765.81392	0.592859737		
		3.9132	22422.56976	0.583921088		
20	0.96	8.385	10464.40072	0.272510435	0.272496799	0.342465753
		8.3891	10459.28646	0.272377251		
		8.3843	10465.27438	0.272533187		
		8.3844	10465.14956	0.272529937		
		8.3843	10465.27438	0.272533187		
40	1.92	13.067	6714.930742	0.174867988	0.174874145	0.20661157
		13.0648	6716.061478	0.174897434		
		13.0662	6715.341874	0.174878695		
		13.0669	6714.98213	0.174869326		
		13.0678	6714.519659	0.174857283		
60	2.88	17.7466	4944.271015	0.128757058	0.128759525	0.147928994
		17.7452	4944.661091	0.128767216		
		17.7485	4943.741725	0.128743274		
		17.745	4944.716822	0.128768667		
		17.746	4944.438183	0.128761411		
80	3.84	22.4271	3912.40954	0.101885665	0.101888664	0.115207373
		22.4244	3912.880612	0.101897933		
		22.4272	3912.392095	0.101885211		
		22.4277	3912.304873	0.101882939		
		22.4258	3912.636339	0.101891571		
100	4.8	27.1065	3237.009573	0.084297124	0.084295072	0.094339623
		27.1072	3236.925983	0.084294947		
		27.1066	3236.997632	0.084296813		
		27.109	3236.711055	0.08428935		
		27.1065	3237.009573	0.084297124		

### 10.3 Eficiência em função de FER

N.º total bytes	10968
N.º total bits	87744
C - Baudrate	38400
Packet size (byte)	100
Tf	0.020833333
Tprop	0.005
a	0.24

Probabilidade de erro em cada trama	Tempo medido (s)	R medido (bits/s)	S medida (R/C)	S medida	S teórica
0.02	5.1894	16908.3131	0.440320654	0.444216044	0.662162162
	5.1345	17089.10313	0.445028727		
	5.1851	16922.33515	0.440685811		
	5.0141	17499.45155	0.455714884		
	5.2011	16870.27744	0.439330142		
0.04	5.2426	16736.73368	0.43585244	0.437880091	0.648648649
	5.2312	16773.20691	0.436802263		
	5.2279	16783.79464	0.437077985		
	5.2011	16870.27744	0.439330142		
	5.1892	16908.96477	0.440337624		
0.1	5.7261	15323.51863	0.399049964	0.400102558	0.608108108
	5.7189	15342.81068	0.399552361		
	5.7023	15387.47523	0.400715501		
	5.6933	15411.79983	0.401348954		
	5.7147	15354.08683	0.399846011		
0.15	5.9496	14747.88221	0.384059433	0.384372322	0.574324324
	5.9487	14750.11347	0.384117538		
	5.9369	14779.43034	0.384880999		
	5.9201	14821.37126	0.38597321		
	5.9687	14700.68859	0.382830432		
0.2	9.1329	9607.463128	0.250194352	0.250253184	0.540540541
	9.1201	9620.947139	0.250545498		
	9.1388	9601.260559	0.250032827		
	9.1393	9600.735286	0.250019148		
	9.1227	9618.205137	0.250474092		
0.25	9.5778	9161.185241	0.238572532	0.236351215	0.506756757
	9.2341	9502.1713	0.247452378		
	9.6456	9096.790246	0.236895579		
	9.9673	8803.18642	0.229249646		
	9.9527	8816.100154	0.229585942		
0.4	20.4503	4290.597204	0.111734302	0.111694152	0.405405405
	20.8563	4207.074121	0.109559222		
	20.6345	4252.295912	0.110736873		
	20.1297	4358.932324	0.113513863		
	20.2344	4336.377654	0.112926501		
0.5	24.9386	3518.401193	0.091625031	0.092012363	0.337837838
	24.8567	3529.993925	0.091926925		
	24.6453	3560.273156	0.092715447		
	24.7844	3540.291474	0.09219509		
	24.9456	3517.413893	0.09159932		
0.6	40.002	2193.490325	0.057122144	0.057280818	0.27027027
	40.3425	2174.976761	0.05664002		
	39.8787	2200.272326	0.057298758		
	40.0154	2192.755789	0.057103015		
	39.2341	2236.421888	0.058240153		