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** – Caraterização 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 papeis 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 reunem-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 3^{o} 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.

```
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.

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

6 Protocolo de aplicação

O protocolo de aplicação implementado teve como objetivo a tranferência 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:

```
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*:

```
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:

```
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:

```
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:

```
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 *llread()*.

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

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 teóricamente 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

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "aplic.h"


int main(int argc, char** argv) {
    if (argc != 4) {
        printf("Usage:\tsender SerialPort Path NameToGive\n\tex: sender <i> <path> <name>\n");
        return -1;
    }

    int porta = atoi(argv[1]);

    if (send_file(porta, argv[2], strlen(argv[2]), argv[3]) < 0) {
        return -1;
    }

    return 0;
}
```

Listing 1: sender.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "aplic.h"


int main(int argc, char** argv) {
    if (argc != 2) {
        printf("Usage:\treceiver SerialPort\n\tex: receiver <i>\n");
        return -1;
    }

    int porta = atoi(argv[1]);

    if (receive_file(porta) < 0 ) {
        return -1;
    }

    return 0;
}
```

Listing 2: receiver.c

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

int receive_file(int porta);
```

Listing 3: aplic.h

```c
#include <stdint.h>

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
```

```c
#define FALSE 0
#define TRUE 1

#define DATA_PACKET_MAX_SIZE 1000

typedef enum type {
    TRANSMITTER,
    RECEIVER
} type_t;

int llopen(int porta, type_t type);

int llclose(int fd, type_t type);

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

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

Listing 4: linklayer.h

```c
#include "aplic.h"
#include "linklayer.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>

#define CONTROL_PACKET_MAX_SIZE 500
#define PACKET_MAX_SIZE (CONTROL_PACKET_MAX_SIZE > DATA_PACKET_MAX_SIZE ?
    CONTROL_PACKET_MAX_SIZE : DATA_PACKET_MAX_SIZE)
#define FILE_NAME_MAX_SIZE 255

#define C_DATA 0x1
#define C_START 0x2
#define C_END 0x3

#define N(seq) ((seq) % 255)

#define L1(K) ((K) & 0b11111111)
#define L2(K) (((K) >> 8) & 0b11111111)
#define K(L1,L2) (256*(L2)+(L1))

#define T_FILE_SIZE 0x0
#define T_FILE_NAME 0x1


static off_t get_file_size(int fd) {
    struct stat s;
    if (fstat(fd, &s) == -1) {
        return -1;
    }

    return s.st_size;
}

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 to 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, &control_packet_size)) == NULL) {
129         close(fd_file);
130         llclose(fd_serial_port, TRANSMITTER);
131         return -1;
132     }
133 
134     // Control packet start
135     if (llwrite(fd_serial_port, control_packet, control_packet_size) < 0) {
136         free(control_packet);
137         close(fd_file);
138         llclose(fd_serial_port, TRANSMITTER);
139         return -1;
140     }
141 
142     if (send_packaged_file(fd_serial_port, fd_file) != 0) {
143         free(control_packet);
144         close(fd_file);
145         llclose(fd_serial_port, TRANSMITTER);
146         return -1;
147     }
148 
149     // Control packet end
150     control_packet[0] = C_END;
151     if (llwrite(fd_serial_port, control_packet, control_packet_size) < 0) {
152         free(control_packet);
153         close(fd_file);
154         llclose(fd_serial_port, TRANSMITTER);
155         return -1;
156     }
157 
158     free(control_packet);
159     close(fd_file);
160     llclose(fd_serial_port, TRANSMITTER);
161     return 0;
162 }
163 
164 int receive_file(int porta) {
```

```
    int fd_serial_port;
    if ((fd_serial_port = llopen(porta, RECEIVER)) < 0) {
        return -1;
    }

    uint8_t *packet = malloc(PACKET_MAX_SIZE);
    if (packet == NULL) {
        llclose(fd_serial_port, RECEIVER);
        return -1;
    }

    int fd_file_to_write = -1;
    int sequence_number = 0;
    off_t file_size = 0;

    int not_end_packet = TRUE;
    while (not_end_packet) {
        int packet_size = 0;
        if ((packet_size = llread(fd_serial_port, packet)) < 0) {
            free(packet);
            llclose(fd_serial_port, RECEIVER);
            return -1;
        }

        uint8_t control_field = packet[0];
        switch (control_field) {
        case C_DATA:
            if (fd_file_to_write == -1) { // Control start packet didn't
    arrive yet
                break;
            }

            int N = packet[1];

            if (N == sequence_number) {
                int num_octets = K(packet[3], packet[2]);
                if (write(fd_file_to_write, &packet[4], num_octets) == -1) {
                    free(packet);
                    llclose(fd_serial_port, RECEIVER);
                    return -1;
                }
                sequence_number = (sequence_number + 1) % 255;
            } else {
                // wrong sequence number
                printf("Wrong packet sequence number. Expected: %d ; Got: %d\
    n", sequence_number, N);
                free(packet);
                llclose(fd_serial_port, RECEIVER);
                return -1;
            }

            break;

        case C_START:;
            int i = 1;
            while (i < packet_size) {
                uint8_t T = packet[i++];
                uint8_t L = packet[i++];

                if (T == T_FILE_SIZE) {
                    memcpy(&file_size, &packet[i], L);

```

```
                } else if (T == T_FILE_NAME) {
                    char file_name[FILE_NAME_MAX_SIZE];
                    memcpy(file_name, &packet[i], L);

                    fd_file_to_write = open(file_name, O_WRONLY | O_APPEND |
    O_CREAT, 0644);
                    if (fd_file_to_write == -1) {
                        free(packet);
                        llclose(fd_serial_port, RECEIVER);
                        return -1;
                    }

                } else {
                    printf("ERROR: not supposed to reach this\n");
                }

                i += L;
            }

            break;

        case C_END:
            if (fd_file_to_write != -1) {
                close(fd_file_to_write);
                not_end_packet = FALSE;
            }
            // else Control start packet didn't arrive yet, so wait for it

            break;

        default:
            break; // invalid control field (ignore packet)
        }
    }

    if (llclose(fd_serial_port, RECEIVER) < 0) {
        free(packet);
        return -1;
    }

    free(packet);
    return 0;
}
```

Listing 5: aplic.c

```
#include "linklayer.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <stdint.h>

#define FLAG 0x7E
#define ESC 0x7D
#define A 0x03
```

```c
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0X07

#define C_RR(r) (0x05 | (((r) << 7) & 0x80))
#define C_REJ(r) (0x01 | (((r) << 7) & 0x80))
#define C_I(s) ((s) << 6)

#define CONTROL_SIZE 5

#define STUFFER 0x20

#define TIME_OUT_TIME 3
#define MAX_NO_TIMEOUT 3

#define HEADER_AND_TAIL_SIZE 10 // more than enough

typedef enum control_frame_type {
    SET,
    DISC,
    UA,
    RR,
    REJ
}  control_frame_type_t;

typedef enum state_sv_frame {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    RR_RCV,
    REJ_RCV,
    BCC_OK,
    STOP
} state_sv_frame_t;

typedef enum state_info_rcv {
    I_START,
    I_GOT_FLAG,
    I_IGNORE,
    I_GOT_A,
    I_GOT_C,
    I_GOT_BCC1,
    I_DATA_COLLECTION,
    I_GOT_ESC,
    I_GOT_END_FLAG,
    I_TEST_DUP_RR,
    I_TEST_DUP_REJ,
    I_RR_DONT_STORE,
    I_RR_STORE,
    I_REJ,
    I_STOP
} state_info_rcv_t;

static struct termios oldtio;
static volatile int g_count = 0;

static uint8_t S = 0;
static uint8_t next_S = 0;
static uint8_t R = 0;

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 
160     return 0;
161 }
162 
163 static int update_state_set_ua(uint8_t c, state_sv_frame_t *state, uint8_t
    byte) {
164     if (state == NULL) {
165         return 1;
166     }
167 
168     switch (*state) {
169         case START:
170             if (byte == FLAG) {
171                 *state = FLAG_RCV;
172             }
173             break;
174 
175         case FLAG_RCV:
176             if (byte == A) {
177                 *state = A_RCV;
178             } else if (byte != FLAG) {
179                 *state = START;
180             }
181             break;
182 
183         case A_RCV:
184             if (byte == c) {
185                 *state = C_RCV;
186             } else if (byte == FLAG) {
187                 *state = FLAG_RCV;
188             } else {
189                 *state = START;
190             }
191             break;
192 
193         case C_RCV:
194             if (byte == (A^c)) {
195                 *state = BCC_OK;
196             } else if (byte == FLAG) {
197                 *state = FLAG_RCV;
198             } else {
199                 *state = START;
200             }
201             break;
```

```
202
203         case BCC_OK:
204             if (byte == FLAG) {
205                 *state = STOP;
206             } else {
207                 *state = START;
208             }
209             break;
210
211         case STOP:
212             break;
213
214         default:
215             printf("ERROR: not supposed to reach this\n");
216             break;
217     }
218
219
220     return 0;
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):
```

```
            *state = I_DATA_COLLECTION;
            break;

        case (I_GOT_END_FLAG):
            if (byte) *state = I_TEST_DUP_RR; // byte = is bcc2 valid ?
            else *state = I_TEST_DUP_REJ;
            break;

        case (I_TEST_DUP_RR):
            if (byte) *state = I_RR_DONT_STORE; // byte = is dup ?
            else *state = I_RR_STORE;
            break;

        case (I_TEST_DUP_REJ):
            if (byte) *state = I_RR_DONT_STORE; // byte = is dup ?
            else *state = I_REJ;
            break;

        case (I_RR_DONT_STORE):
            *state = I_START;
            break;

        case (I_RR_STORE):
            *state = I_STOP;
            break;

        case (I_REJ):
            *state = I_START;
            break;

        case (I_STOP):
            break;

        default:
            break;
    }

    return 0;
}

static void time_out() {
    printf("alarme # %d\n", g_count);
    g_count++;
}

static int setup_alarm() {
    struct sigaction new;
    sigset_t smask;

    if (sigemptyset(&smask)==-1) {
        perror ("sigsetfunctions");
        return 1;
    }

    new.sa_handler = time_out;
    new.sa_mask = smask;
    new.sa_flags = 0;

    if (sigaction(SIGALRM, &new, NULL) == -1) {
        perror ("sigaction");
        return 1;
    }
```

```
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
```

```
    return close(fd);
}

static void R_invert(){
    R = ((!R) << 7) >> 7;
}

int llopen(int porta, type_t type) {
    state_sv_frame_t state;
    int fd = common_open(porta);
    if (fd < 0) {
        printf("Failed to open serial port.\n");
        return -1;
    }

    printf("%d opened fd\n", fd);

    uint8_t set[CONTROL_SIZE];
    control_frame_builder(SET, set);

    uint8_t ua[CONTROL_SIZE];
    control_frame_builder(UA, ua);

    if (setup_alarm() != 0) {
        common_close(fd);
        return -1;
    }
    g_count = 0;

    switch (type) {
    case TRANSMITTER:;
        int ua_received = FALSE;
        int res;
        while (g_count < MAX_NO_TIMEOUT && !ua_received) {
            state = START;

            res = write(fd, set, CONTROL_SIZE * sizeof(uint8_t));
            if (res == -1) {
                printf("llopen() -> write() TRANSMITTER error\n");
                common_close(fd);
                return -1;
            }
            printf("SET sent.\n");
            printf("%d bytes written\n", res);

            alarm(TIME_OUT_TIME);

            int timed_out = FALSE;
            while (!timed_out && state != STOP) {
                uint8_t byte_read = 0;

                res = read(fd, &byte_read, 1);
                if (res == 1) {
                    if (update_state_set_ua(C_UA, &state, byte_read) != 0) {
                        common_close(fd);
                        alarm(0);
                        return -1;
                    }
                    ua_received = (state==STOP);

                } else if (res == -1) {
                    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 timedout\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                         }
```

```
                        ua_received = (state==STOP);

                    } else if (res == -1) {
                        timed_out = TRUE;

                    } else {
                        printf("DEBUG: not supposed to happen\n");
                    }
                }

                alarm(0);
            }

            if (ua_received) {
                printf("UA received.\n");
            }
        }

        break;
    }

    res = common_close(fd);

    return res;
}

int message_stuffing(uint8_t in_msg[], unsigned int in_msg_size, uint8_t **
    out_msg){

    int size_counter = 0;
    *out_msg = malloc(in_msg_size*2);

   uint8_t * out_message = * out_msg;

    for (int i = 0; i < in_msg_size; i++){
        switch (in_msg[i]){
        case FLAG:
            out_message[size_counter++] = ESC;
            out_message[size_counter++] = FLAG ^ STUFFER;
            break;
        case ESC:
            out_message[size_counter++] = ESC;
            out_message[size_counter++] = ESC ^ STUFFER;
            break;
        default:
            out_message[size_counter++] = in_msg[i];
            break;
        }
    }
    return size_counter;
}

int message_destuffer(uint8_t in_msg[], unsigned int in_msg_size, uint8_t **
    out_msg){

    int size_counter = 0;
    *out_msg = malloc(in_msg_size);

    uint8_t * out_message = * out_msg;

    for (int i = 0; i < in_msg_size; i++){
        if (in_msg[i] == ESC){
```

```
            out_message[size_counter] = (in_msg[++i] ^ STUFFER);
        } else {
            out_message[size_counter] = in_msg[i];
        }
        size_counter++;
    }

    return size_counter;
}

uint8_t bcc2_builder(uint8_t msg[], unsigned int msg_size){

    if (msg_size == 1) {
        return msg[0];
    } else if ( msg_size < 0) {
        return 0;
    }

    uint8_t ret = msg[0];

    for (int i = 1; i < msg_size; i++){
        ret ^= msg[i];
    }

    return ret;
}

int llwrite(int fd, uint8_t * buffer, int length){

    int write_successful = 0;
    int ret = 0;
    uint8_t bcc2 = bcc2_builder(buffer, length);
    uint8_t *unstuffed_msg = malloc((length+1) * sizeof(uint8_t));
    memcpy(unstuffed_msg, buffer, length);
    unstuffed_msg[length] = bcc2;
    uint8_t *stuffed_msg = NULL;
    int stuffed_msg_len = message_stuffing(unstuffed_msg, length+1, &
    stuffed_msg);
    free(unstuffed_msg);
    int total_msg_len = stuffed_msg_len + CONTROL_SIZE;
    uint8_t *info_msg = malloc(total_msg_len);

    info_msg[0] = FLAG;
    info_msg[1] = A;
    info_msg[2] = C_I(S);
    info_msg[3] = A ^ C_I(S);
    memcpy(&(info_msg[4]), stuffed_msg, stuffed_msg_len);
    info_msg[total_msg_len-1] = FLAG;

    setup_alarm();
    g_count = 0;

    while(!write_successful && g_count < MAX_NO_TIMEOUT) {

        printf("----- TASK: WRITING MESSAGE\n");

        if (write(fd, info_msg, total_msg_len * sizeof(uint8_t)) == -1) {
            printf("llwrite() -> write() error\n");
            free(info_msg);
            free(stuffed_msg);
            return -1;
        }
```

```
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