1º Trabalho Laboratorial

Relatório

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Bancada 1 - 3MIEIC07:
André Reis - up201403057
Bernardo Ferreira dos Santos Aroso Belchior - up201405381
Edgar de Lemos Passos - up201404131
José Pedro Teixeira Monteiro - up201406458

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

26 de Novembro de 2016

Conteúdo

1 Sumário

Este relatório tem como objetivo complementar o primeiro projeto da Unidade Curricular Redes de Computadores. Este projecto consistiu no desenvolvimento de uma aplicação que fosse capaz de transferir ficheiros entre dois computadores ligados por uma porta de série assíncrona. Esta transmissão deveria ser capaz de ultrapassar a introdução de erros na comunicação e a desconexão da ligação.

Este projeto foi cumprido, sendo que foi desenvolvida uma única aplicacao capaz de enviar e receber os dados corretamente e de restabelecer a ligação após qualquer falha.

2 Introdução

O objetivo deste relatorio e expor os aspetos mais teóricos da realizacao do trabalho, que, dada a sua natureza teórica, não terão sido avaliados na demonstração do projeto.

O objetivo do trabalho era implementar um dado protocolo de ligação de dados, especificado no guião, de forma a fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por uma porta de série.

Para isto, foi necessário desenvolver funções de criação e sincronização de tramas (*framing*), estabelecimento e terminação da ligação, numeração de tramas, confirmação de receção de uma trama sem erros e na sequência correta, controlo de erros e de fluxo.

Este relatório será organizado da seguinte forma:

- Arquitetura - Demonstração dos blocos funcionais e interfaces.
- Estrutura do Código - Exposição das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- Casos de Uso Principais - Identificação dos casos de uso principais e sequências de chamada de funções.
- Protocolo de Ligação Lógica - Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
- Protocolo de Aplicação - Identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos.
- Validação - Descrição dos testes efetuados com apresentação quantificada dos resultados.
- Elementos de Valorização - Identificação dos elementos de valorização implementados e descrição da estratégia de implementação.
- Conclusão - Síntese da informação apresentada anteriormente e reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura e Estrutura do Código

A nossa aplicação foi desenvolvida com duas principais camadas lógicas, para além da Interface com o utilizador.

Iremos então apresentar estes blocos.

3.1 Camada de Ligação de Dados *(Data Link Layer)*

A camada de ligação de dados, desenvolvida nos ficheiros *data_link_layer.c* e *data_link_layer.h*, é a camada lógica de mais baixo nível na nossa aplicação, funcionando como ponte entre a porta de série e a camada de aplicação. É responsável pela comunicação entre as duas máquinas. Contém, por isso, as funções que configuram, iniciam e terminam a ligação, assim como as que escrevem e lêem dados da porta de série, tratando das necessidades da comunicação - nomeadamente tratamento de erros e *stuffing / destuffing de pacotes*.

Para facilitar o uso de variáveis necessárias à ligação de dados, utilizou-se a seguinte estrutura *data_link*:

```
struct {
  char port[20]; /* Serial port device e.g. /dev/ttyS0 */
  int baud_rate;
  unsigned int sequence_num; /* Frame sequence number (0 or 1) */
  unsigned int timeout;      /* Time to timeout e.g. 1 second */
  unsigned int num_retries;  /* Maximum number of retries */
  status stat;
  struct sigaction old_action;
  int baudrate;
} data_link;
```

3.2 Camada de Aplicação *(Application Layer)*

A camada de aplicação, desenvolvida nos ficheiros *application_layer.c* e *application_layer.h* é a camada lógica situada diretamente acima da camada de ligação de dados, servindo de ponte entre esta e a interface com o utilizador. É responsável pela utilização da camada de lógica para comunicar e transferir ficheiros de acordo com os parâmetros que lhe são passados pelo utilizador.

Para facilitar o uso de variáveis necessárias à aplicação, utilizou-se a seguinte estrutura *app_layer*:

```
typedef struct {
  int file_descriptor;      /* Serial port file descriptor */
  status app_layer_status; /* TRANSMITTER | RECEIVER */
} app_layer;
```

3.3 Interface *(Command Line Interface*

Por fim, o ficheiro *interface.c* contém o código de interação com o utilizador, para recolher os parâmetros de configuração necessários à transferência de dados, como a informação de ser Recetor ou Transmissor de dados, a *Baud Rate*, o número de *timeouts* permitidos e tempo necessário para se considerar um *timeout*.

4 Casos de Uso Principais

A aplicação desenvolvida apenas precisa de um parâmetro, o dispositivo de porta de série (normalmente /dev/ttyS0). Depois, ao executar, a aplicação pede ao utilizador para inserir alguns outros parâmetros, como o modo de execução (Transmissor ou Recetor), o Baud Rate pretendido, o número máximo de timeouts admissíveis e o tempo para considerar um timeout. Após isto, a aplicação vai automaticamente buscar o ficheiro "pinguim.gif"se estiver em modo Transmissor e começa a ligação e a enviar o ficheiro.

Se estiver em modo Recetor, fica à espera que algum Transmissor inicie uma ligação.

5 Protocolo de Ligação Lógica

O protocolo de ligação lógica implementado tem como principais aspetos funcionais:

- Configuração da porta de série;
- Estabelecimento de ligação pela porta de série;
- Transferência de dados pela porta de série, fazendo *stuffing* e *destuffing* dos mesmos;
- Recuperação de erros durante a transferência de dados.

Para isto, foi necessário implementar as funções:

5.1 llopen() e llclose()

Estas funções são as necessárias para iniciar e terminar a ligação pela porta de série. Para isso a função **llopen** começa por alterar as configurações da porta de série para as pretendidas.

Após isto, se a aplicação for Transmissor, cria uma trama **SET** (*Set Up*) e envia-a. Para isto foi criada uma função *send_US_frame()* (envia trama U ou S, pois SET é uma trama US), que enquanto não recebe a resposta pretendida - neste caso **UA** - activa um alarme com duração definada pelo utilizador e tenta enviar a trama que lhe é passada. Se o alarme for desencadeado, conta como um *timeout* e tenta enviar a trama novamente. Caso o número de *timeouts* máximo permitido for excedido, esta função termina com um estado de erro - informando a função llopen que não conseguiu estabelecer a comunicação com o recetor, retornando -1.

Se a aplicação for Recetor, fica à espera até receber uma trama **SET**, e quando isto acontece, responde com uma trama **UA** (*Unnumbered Acknowledgment*), estabelecendo assim, a ligação com sucesso.

A função llclose, do lado do Transmissor, tenta terminar a ligação ao enviar uma trama **DISC** utilizando a função *send_US_frame*, que espera pela resposta do Recetor, terá de ser uma trama **DISC**. Ao recebê-la, responde com uma trama **UA**, informando o Recetor que recebeu a sua intenção de finalizar a ligação. Após isto, repõe as configurações anteriores da porta de série e termina com sucesso.

O Recetor espera até receber uma trama **DISC**, respondendo com uma trama do mesmo tipo. Após transmissão da resposta, espera pela trama **UA** do Transmissor e repõe as configurações anteriores da porta de série, terminando com sucesso.

5.2 llwrite() e llread()

Estas funções são as principais responsáveis pela escrita e leitura de dados no decorrer da aplicação.

A função **llwrite** recebe um pacote e envia-o pela porta de série depois de o encapsular numa trama *I* (trama de Informação). Para isto criou-se outra função, a *create_I_frame* (cria trama do tipo I). Esta função trata de gerar todos os campos da trama **I** e de fazer o *byte stuffing* necessário.

A função **llread** espera até receber uma trama. Se for uma trama **DISC**, então procede ao fecho da ligação estabelecida; senão confere a validade do cabeçalho da trama recebida, rejeitando-a se for inválida (enviando um comando **REJ**). Caso o cabeçalho seja válido, a função procede com o processo de verificação, fazendo *destuff* ao pacote contido na trama e ao *Block Check Character* correspondente ao pacote. De seguida, confirma se o pacote é válido. Caso o resultado seja afirmativo, envia uma

trama **RR** - mesmo que o pacote seja duplicado. Caso contrário, transmite uma trama **REJ**, exceto se se tratar de um pacote duplicado, confirmando-o com **RR**.

6 Protocolo de Aplicação

O protocolo de aplicação implementado tem como principais aspetos funcionais:

- Geração e transferência dos pacotes de controlo e de dados;
- Leitura e Escrita do ficheiro a transferir.

Para isto, foi necessário implementar as funções:

6.1 send_data() e receive_data()

A função **send_data** é a função responsável pelo comportamento principal do Transmissor, e envia dados para a camada inferior enviar pela porta de série.

Para isto, a função começa por criar um pacote de controlo, o **Start Packet**, que contém a informação codificada em **TLV**s *(Type, Length, Value)*, isto é, para cada parâmetro a passar nesse pacote, é necessário passar o tipo do parâmetro, depois o seu tamanho e só depois o valor do parâmetro em si. Na aplicação desenvolvida, é passada neste pacote a informação das permissões do ficheiro, do seu nome e do seu tamanho. A função serve-se depois da função da camada inferior **llwrite** para codificar e enviar este pacote.

Após isto, a função lê o ficheiro que se quer transferir e entra no ciclo principal do programa, em que se vai construindo pacotes de dados com os bytes lidos do ficheiro e enviando esses pacotes com a função **llwrite**. Quando o ficheiro acaba de ser escrito, a função envia finalmente um **End Packet** e termina.

A função **receive_data** é a função responsável pelo comportamento principal do Transmissor, recebe dados da camada inferior para compor o ficheiro lido.

Para isto, a função começa por ler, usando a função **llread** da camada inferior, o pacote de controlo e daí tirar o nome do ficheiro a ler, as suas permissões e o seu tamanho final. Depois, enquanto receber pacotes válidos e que não sejam o **End Packet**, continua a escrever os bytes de informação vindos de **llread** para o ficheiro, acabando quando o receber o **End Packet**.

7 Validação

Para verificar a robustez da aplicação desenvolvida, foram aplicados os seguintes testes:

- Enviar um ficheiro;
- Enviar um ficheiro, carregar no botão de interrupção durante a transmissão e re-abrindo a transmissão depois;
- Enviar um ficheiro e desligar o cabo da porta de série durante a transmissão, voltando a ligar depois;
- Enviar um ficheiro e introduzir erros na ligação com o cabo de cobre;
- Enviar um ficheiro, interromper a ligação e introduzir erros com o cabo de cobre.

A aplicação foi capaz de superar todos estes testes, verificando-se isto tanto pelos *bytes* do ficheiro estarem correctos como pela demonstração no ecrã do estado do envio e verificação do resultado através do comando *md5sum*.

8 Elementos de Valorização

Os elementos de valorização implementados foram:

8.1 Seleção de parâmetros pelo Utilizador

A interface por linha de comandos permite ao utilizador escolher a **Baud Rate**, o número máximo de *timeouts* e o tempo de *timeout*.

8.2 Implementação do REJ

```
/* Create a BCC2 for the I frame
check if the received one is correct*/
char calculated_bcc2 = 0;
int i;
for (i = 0; i < *packet_len; i++)
  calculated_bcc2 ^= packet[i];

if (calculated_bcc2 ==
    expected_bcc2) { // valid BCC2 - may still be a duplicate

  reply = create_US_frame(&reply_len, RR);

  /* Only need to check sequence number if packet is a data packet.
  * If it is, and the sequence number is invalid, discard the packet
  * by setting its length to 0 */
  if (!has_valid_sequence_number(frame[2], s)) {
    *packet_len = 0;
    printf("Found duplicate frame. Discarding...\n");
  } else {
    //Only flip sequence number if the whole frame is valid
    //And not a duplicate.
    s = !s;
  }

  read_succesful = 1;
} else { // BCC2 does not match -> check sequence number
  if (has_valid_sequence_number(frame[2], s)) { // new frame, request retry
    reply = create_US_frame(&reply_len, REJ);
    printf("Found new incorrect frame. Rejecting...\n");
  } else {
    reply = create_US_frame(&reply_len,
                            RR); // duplicate frame, send RR and discard
    read_succesful = 1;
    *packet_len = 0;
    printf("Found duplicate frame. Discarding...\n");
```

```
  }
}

if (write_to_tty(fd, reply, reply_len) != 0) {
  printf("Error write_to_tty() in function llread().\n");
  return -1;
}
```

8.3 Verificação da integridade dos dados pela Aplicação

A aplicação verifica o tamanho dado do ficheiro com o número de bytes de dados recebidos.

8.4 Registo de ocorrências

A aplicação regista o número de *timeouts* que aconteceram ao longo da sua execução e imprime esse número no ecrã ao terminar.

9 Conclusão

Em geral, pensamos que o objetivo principal deste projeto foi alcançado. Os conceitos necessários para o arrancar deste projeto são um pouco complexos no que toca à aprendizagem, mas sentimos que após a compreensão do essencial, o guião e a documentação cumprem bem o seu papel.

Quanto ao trabalho realizado, fez com que todos os elementos do grupo entendessem bem o sistema de independência de camadas, pois como mostrámos, a camada de aplicação serve-se da camada de ligação de dados mas é independente do modo de agir desta, apenas lhe interessa como acede ao serviço disponibilizado.

Quanto a possíveis melhorias, há a escolha do ficheiro a transferir (embora tenha sido implementada facilmente na demonstração), a adição de maior detalhe no registo de ocorrências e na verificação da integridade pela aplicação e a geração aleatória de erros em tramas de Informação.

A Código fonte

A.1 application_layer.h

```
#ifndef APPLICATION_LAYER_H
#define APPLICATION_LAYER_H

#include "data_link_layer.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <termios.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdint.h>

typedef struct {
  int file_descriptor;      /* Serial port file descriptor */
  status app_layer_status; /* TRANSMITTER | RECEIVER */
} app_layer;

app_layer application;

/**
 * Establishes a connection between the receiver and the transmitter
 * calls ll_open
 * returns -1 on error
 */
int set_up_connection(char *terminal, status stat);

/**
* Gets a full combination of packets and parses its information.
*/
int receive_data();

/**
* Writes file to serial port.
*/
int send_data(char *path, char *filename);
/**
* Prints the bar of the readed bytes.
*/
void print_current_status(size_t elapsed_bytes, size_t total_bytes, int status);

#endif
```

A.2 application_layer.c

```
#include "application_layer.h"
#include <errno.h>

#define FILE_SIZE_BYTE 0
#define FILE_NAME_BYTE 1
#define FILE_PERMISSIONS_BYTE 2

#define PACKET_SIZE 256
#define PACKET_HEADER_SIZE 4
#define PACKET_DATA_SIZE PACKET_SIZE - PACKET_HEADER_SIZE
#define FILE_SIZE 10968

int num_bytes_read=0;

int set_up_connection(char *terminal, status stat) {
  if (stat != TRANSMITTER && stat != RECEIVER) {
    printf("application_layer :: set_up_connection() :: Invalid status.\n");
    return -1;
  }

  application.app_layer_status = stat;

  int port;
  if (strcmp(terminal, COM1_PORT) == 0)
    port = COM1;
  else if (strcmp(terminal, COM2_PORT) == 0)
    port = COM2;
  else {
    printf("application_layer :: set_up_connection() :: Invalid terminal\n");
  }

  if ((application.file_descriptor =
          llopen(port, application.app_layer_status)) < 0) {
    printf("application_layer :: set_up_connection() :: llopen failed\n");
    return -1;
  }

  return application.file_descriptor;
}

off_t get_file_size(char *packet, int packet_len) {
  int i = 1;
  while (i < packet_len) {
    if (packet[i] == FILE_SIZE_BYTE)
```

```
      return *((off_t *)(packet + i + 2));

    i += 2 + packet[i + 1];
  }

  return 0;
}

char *get_file_name(char *packet, int packet_len) {
  int i = 1;
  while (i < packet_len) {
    if (packet[i] == FILE_NAME_BYTE) {
      char *file_name = (char *)malloc((packet[i + 1] + 1) * sizeof(char));
      memcpy(file_name, packet + i + 2, packet[i + 1]);
      file_name[(packet[i + 1] + 1)] = 0;
      return file_name;
    }

    i += 2 + packet[i + 1];
  }

  return NULL;
}

mode_t get_file_permissions(char *packet, int packet_len) {
  int i = 1;
  while (i < packet_len) {
    if (packet[i] == FILE_PERMISSIONS_BYTE)
      /*
      * mode_t is 4 bytes long, but the information sent is
      * only 2 bytes. As such, we need to read it as a 2 byte int
      * and cast it to mode_t.
      */
      return *((mode_t *)(packet + i + 2));

    i += 2 + packet[i + 1];
  }

  return -1;
}

int send_data(char *path, char *filename) {
  char *full_path =
      (char *)malloc(sizeof(char) * (strlen(path) + 1 + strlen(filename)));
  strcpy(full_path, path);
  strcat(full_path, "/");
```

```
strcat(full_path, filename);
int fd = open(full_path, O_RDONLY);
// free(full_path);

if (fd < 0) {
  printf("Error opening file. Exiting...\n");
  return -1;
}

struct stat file_info;
fstat(fd, &file_info);

int filename_len = strlen(filename);
off_t file_size = file_info.st_size;
mode_t file_mode = file_info.st_mode;

/*
* START PACKET
*/
int start_packet_len = 7 + sizeof(mode_t) + sizeof(file_info.st_size) + filename_l
char *start_packet = (char *)malloc(sizeof(char) * start_packet_len);
start_packet[0] = START_PACKET_BYTE;
start_packet[1] = FILE_PERMISSIONS_BYTE;
start_packet[2] = sizeof(mode_t);
*((mode_t *)(start_packet + 3)) = file_mode;

start_packet[3 + sizeof(mode_t)] = FILE_SIZE_BYTE;
start_packet[4 + sizeof(mode_t)] = sizeof(file_info.st_size);
*((off_t *)(start_packet + 5 + sizeof(mode_t))) = file_size;
start_packet[5 + sizeof(mode_t) + sizeof(file_info.st_size)] = FILE_NAME_BYTE;
start_packet[6 + sizeof(mode_t) + sizeof(file_info.st_size)] = filename_len;
strcat(start_packet + 7 + sizeof(mode_t) + sizeof(file_info.st_size), filename);
llwrite(application.file_descriptor, start_packet, start_packet_len);

/*
* DATA PACKET
*/
char data[PACKET_DATA_SIZE];
int i = 0;
off_t bytes_remaining = file_size;

while (bytes_remaining > 0) {
  int read_chars;
  if ((read_chars = read(fd, data, PACKET_DATA_SIZE)) <= 0) {
    printf("Error reading from file. Exiting...\n");
    return -1;
```

```
    }

    char information_packet[PACKET_SIZE];
    int packet_size = read_chars + PACKET_HEADER_SIZE;
    information_packet[0] = DATA_PACKET_BYTE;
    information_packet[1] = i % 256;
    information_packet[2] = read_chars / 256;
    information_packet[3] = read_chars % 256;

    memcpy(information_packet + PACKET_HEADER_SIZE, data, read_chars);
    if(llwrite(application.file_descriptor, information_packet, packet_size) == -1)
      printf("Connection lost. Exiting program...\n");
      close(fd);
      return -1;
    }

    print_current_status(file_size - bytes_remaining, file_size, TRANSMITTER);

    bytes_remaining -= read_chars;
    i++;
  }

  print_current_status(file_size - bytes_remaining, file_size, TRANSMITTER);

  /*
  * END PACKET
  */
  char end_packet[] = {END_PACKET_BYTE};
  llwrite(application.file_descriptor, end_packet, 1);
  close(fd);

  printf("Total timeouts: %d.\n",getTotalTimeouts());

  return 0;
}

int receive_data() {
  char packet[PACKET_SIZE];

  int packet_len;
  /**
  * Reading and parsing start packet
  */
  do {
    if (llread(application.file_descriptor, packet, &packet_len) != 0) {
      printf("Error llread() in function receive_data().\n");
```

```
    exit(-1);
  }
} while (packet_len == 0 || packet[0] != (unsigned char)START_PACKET_BYTE);

printf("Receiving data...\n");

off_t file_size = get_file_size(packet, packet_len);
char *file_name = get_file_name(packet, packet_len);
mode_t file_mode = get_file_permissions(packet, packet_len);

int fd = open(file_name, O_WRONLY | O_CREAT | O_TRUNC);

if (fd < 0) {
  printf("Error opening file. Exiting...\n");
  return -1;
}

/**
* Reading and parsing data packets
*/
char cur_seq_num = 0;
if (llread(application.file_descriptor, packet, &packet_len) != 0) {
  printf("Error llread() in function receive_data().\n");
  close(fd);
  exit(-1);
}

while (packet_len == 0 || packet[0] != (unsigned char)END_PACKET_BYTE) {
  if (packet_len > 0 && cur_seq_num == packet[1]) {
    unsigned int data_len =
        (unsigned char)packet[2] * 256 + (unsigned char)packet[3];
    write(fd, packet + 4, data_len);

    num_bytes_read+=data_len;

    print_current_status(num_bytes_read, file_size, RECEIVER);
    cur_seq_num++;
  }

  if (llread(application.file_descriptor, packet, &packet_len) != 0) {
    printf("Error llread() in function receive_data().\n");
    close(fd);
    exit(-1);
  }
}
```

```
  struct stat file_info;
  fstat(fd, &file_info);
  printf("Expected %lu bytes.\nReceived %lu bytes.\n", file_size,
         file_info.st_size);

  close(fd);
  chmod(file_name, file_mode);

  return 0;
}

void print_current_status(size_t elapsed_bytes, size_t total_bytes, int status){

  int i = 0;
  static int count = 0;
  int n_ellipsis;

  n_ellipsis = count++ % 3;
  printf("\r\033[3F\033[G\033[J\033[E");

        float perc = ((float)elapsed_bytes / total_bytes) * 100;

  status == TRANSMITTER ? printf("Sending data") : printf("Receiving data");
  while(i++ <= n_ellipsis){
    printf(".");
  }

        printf("\n|");

        float t;
        for(t = 0; t < perc; t+= 3) {
                printf("-");
        }


        for(t = 100 - t; t > 0; t -= 3) {
                printf(" ");
        }

        printf("|  %.2f %%\n", perc);
        fflush(stdout);
}
```

A.3 data_link_layer.h

```
#ifndef DATA_LINK_LAYER_H
#define DATA_LINK_LAYER_H

#define FLAG 0x7E
#define ESCAPE 0x7D
#define STUFFING_BYTE 0x20

#define SEND 0x03
#define RECEIVE 0x01

#define SET 0x03
#define UA 0x07
#define DISC 0x0B
#define RR 0x05
#define REJ 0x01

#define COM1 0
#define COM2 1
#define COM1_PORT "/dev/ttyS0"
#define COM2_PORT "/dev/ttyS1"

#define US_FRAME_LENGTH 5
#define I_FRAME_HEADER_SIZE 6

#define DATA_PACKET_BYTE 1
#define START_PACKET_BYTE 2
#define END_PACKET_BYTE 3

typedef enum { TRANSMITTER, RECEIVER } status;

/**
* Opens the terminal refered to by terminal.
* Updates the port settings and saves the old ones to be reset.
* Depending on status, it send a SET or UA frame.
* Returns the according file descriptor on success,
* returning -1 otherwise.
*/
int llopen(int port, status stat);

/**
* Writes the given msg with len length to the
* given fd.
* Returns -1 on error.
*/
```

```
int llwrite(int fd, char *msg, int len);

/**
* Reads the message from fd and places it on
* msg, updating len accordingly.
* Returns -1 on error.
*/
int llread(int fd, char *msg, int *packet_len);

/**
* Closes the given fd and sets the port settings.
* Returns -1 on error.
*/
int llclose(int fd);

/**
* Closes the given fd and resets the port settings.
* Should be used when there is a problem with the connection.
*/
void force_close(int fd);

/**
* Initialize data_link struct variables.
*/
void init_data_link(int time_out,int number_retries, int baudrate);

/**
* Get the number of timeouts at the end.
*/
int getTotalTimeouts();

#endif
```

A.4 data_link_layer.c

```
#include "data_link_layer.h"
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <termios.h>
#include <unistd.h>

#define BAUDRATE B9600

volatile int STOP = 0;
struct termios old_port_settings;
int connection_timeouts = 0;
int ignore_flag = 0;
char r = 0;
int total_time_outs=0;

struct {
  char port[20]; /* Serial port device e.g. /dev/ttyS0 */
  int baud_rate;
  unsigned int sequence_num; /* Frame sequence number (0 or 1) */
  unsigned int timeout;      /* Time to timeout e.g. 1 second */
  unsigned int num_retries;  /* Maximum number of retries */
  status stat;
  struct sigaction old_action;
  int baudrate;
} data_link;

//'Private' functions
int write_to_tty(int fd, char *buf, int buf_length);
int read_from_tty(int fd, char *frame, int *frame_len);
int send_US_frame(int fd, char *frame, int len, int (*is_reply_valid)(char *));
int send_I_frame(int fd, char *frame, int len);
char *create_I_frame(int *frame_len, char *packet, int packet_len);
char *create_US_frame(int *frame_len, int control_byte);
int is_frame_UA(char *reply);
int is_frame_RR(char *reply, int reply_len);
int is_frame_REJ(char *reply, int reply_len);
int is_frame_DISC(char *reply);
int is_I_frame_header_valid(char *frame, int frame_len);
```

```
void timeout(int signum);
int has_valid_sequence_number(char control_byte, int s);
int reset_settings(int fd);
int close_receiver_connection(int fd);
void print_as_hexadecimal(char *msg, int msg_len);

/**
* Change the terminal settings
* return -1 on error
*/
int set_terminal_attributes(int fd);
/**
* Stuffing the frame given.
*/
char *stuff(char *packet, int *packet_len);
/**
* Destuffing the frame given.
*/
void destuff(char *packet, char *destuffed, int *packet_len);

/*
* Definitions
*/

int set_terminal_attributes(int fd) {
  struct termios new_port_settings;

  if (tcgetattr(fd, &old_port_settings) ==
      -1) { /* save current port settings */
    printf("Error getting port settings.\n");
    close(fd);
    return -1;
  }

  bzero(&new_port_settings, sizeof(new_port_settings));

  new_port_settings.c_cflag = data_link.baudrate | CS8 | CLOCAL | CREAD;
  new_port_settings.c_iflag = IGNPAR;
  new_port_settings.c_oflag = 0;

  /* set input mode (non-canonical, no echo,...) */
  new_port_settings.c_lflag = 0;

  new_port_settings.c_cc[VTIME] =
      0; /* inter-character timer unused in 1/10th of a second*/
  new_port_settings.c_cc[VMIN] = 1; /* blocking read until x chars received */
```

```
  tcflush(fd, TCIOFLUSH);

  if (tcsetattr(fd, TCSANOW, &new_port_settings) == -1) {
    printf("Error setting port settings.\n");
    close(fd);
    return -1;
  }

  return 0;
}

int is_I_frame_header_valid(char *frame, int frame_len) {
  if (frame_len < 6)
    return 0;

  return frame[0] == FLAG && frame[1] == SEND &&
         frame[3] == (frame[1] ^ frame[2]);
}

/**
* TODO:
* if stat == TRANSMITTER -> send SET, receive UA
* reverse if stat == RECEIVE
*/
int llopen(int port, status stat) {

  int fd; // value to be returned
  int frame_len;

  switch (port) {
  case COM1:
    strcpy(data_link.port, COM1_PORT);
    break;

  case COM2:
    strcpy(data_link.port, COM2_PORT);
    break;

  default:
    printf("data_link_layer :: llopen() :: invalid port!\n");
    return -1;
  }

  if (stat != TRANSMITTER && stat != RECEIVER) {
    printf("data_link_layer :: llopen() :: Invalid status.\n");
```

```
  return -1;
}

data_link.stat = stat;
/**
* Opening the serial port
*/
if ((fd = open(data_link.port, O_RDWR | O_NOCTTY)) < 0) {
  printf("Error opening terminal '%s'\n", data_link.port);
  return -1;
}

if (set_terminal_attributes(fd) != 0) {
  printf("Error set_terminal_attributes() in function llopen().\n");
  return -1;
}

struct sigaction new_action;
new_action.sa_handler = timeout;
new_action.sa_flags &=
    !SA_RESTART; // Needed in order to block read from restarting.

if (sigaction(SIGALRM, &new_action, &data_link.old_action) == -1)
  printf("Error installing new SIGALRM handler.\n");

if (stat == TRANSMITTER) {
  char *frame = create_US_frame(&frame_len, SET);
  if (send_US_frame(fd, frame, frame_len, is_frame_UA) == -1) {
    printf("data_link_layer :: llopen() :: send_US_frame failed\n");
    close(fd);
    return -1;
  }

} else {
  char msg[255];
  int msg_len;
  if (read_from_tty(fd, msg, &msg_len) == -1) {
    printf("Error read_from_tty() in function llopen().\n");
    return -1;
  }

  char *frame = create_US_frame(&frame_len, UA);
  if (write_to_tty(fd, frame, frame_len) == -1) {
    printf("Error write_to_tty() in function llopen().\n");
    return -1;
  }
```

```
  }

  printf("data_link_layer :: llopen() :: connection succesfully "
         "established.\n");

  printf("DATA_LINK values: baud_rate %d, timeout: %d, num_retries: %d.\n\n\n\n\n", da

  return fd;
}

int llwrite(int fd, char *packet, int packet_len) {
  // Writes and checks for validity
  // Using send_I_frame
  int frame_len;
  char *frame = create_I_frame(&frame_len, packet, packet_len);
  return send_I_frame(fd, frame, frame_len);
}

/**
* Read a frame from the serial port and check its validity and make sure
* it is not a duplicate
* After the check, send the appropriate response to the TRANSMITTER
*/
int llread(int fd, char *packet, int *packet_len) {
  char *reply;
  int reply_len;

  char frame[512];
  int frame_len;
  ignore_flag = 0;
  static int s = 0;

  int read_succesful = 0;
  while (!read_succesful) {
    read_from_tty(fd, frame, &frame_len);
    if (is_frame_DISC(frame)) {
      close_receiver_connection(fd);
      reset_settings(fd);
      return -1;
    }

    if (!is_I_frame_header_valid(frame, frame_len)) {// Invalid frame
      printf("Invalid frame header. Rejecting frame..\n");
      reply = create_US_frame(&reply_len, REJ);
    } else { // Frame is valid
      // Updates the packet length.
```

```
*packet_len = frame_len - I_FRAME_HEADER_SIZE;

char expected_bcc2;
if (frame[frame_len - 3] == ESCAPE) {
  expected_bcc2 = frame[frame_len - 2] ^ STUFFING_BYTE;
  // If the BCC was stuffed, the frame header is one byte bigger
  // So the packet length will be one byte shorter.
  *packet_len = *packet_len - 1;
} else
  expected_bcc2 = frame[frame_len - 2];

destuff(frame + 4, packet, packet_len);

/* Create a BCC2 for the I frame
check if the received one is correct*/
char calculated_bcc2 = 0;
int i;
for (i = 0; i < *packet_len; i++)
  calculated_bcc2 ^= packet[i];

if (calculated_bcc2 ==
    expected_bcc2) { // valid BCC2 - may still be a duplicate

  reply = create_US_frame(&reply_len, RR);

  /* Only need to check sequence number if packet is a data packet.
  * If it is, and the sequence number is invalid, discard the packet
  * by setting its length to 0 */
  if (!has_valid_sequence_number(frame[2], s)) {
    *packet_len = 0;
    printf("Found duplicate frame. Discarding...\n");
  } else {
    //Only flip sequence number if the whole frame is valid
    //And not a duplicate.
    s = !s;
  }

  read_succesful = 1;
} else { // BCC2 does not match -> check sequence number
  if (has_valid_sequence_number(frame[2], s)) { // new frame, request retry
    reply = create_US_frame(&reply_len, REJ);
    printf("Found new incorrect frame. Rejecting...\n");
  } else {
    reply = create_US_frame(&reply_len,
                            RR); // duplicate frame, send RR and discard
    read_succesful = 1;
```

```
            *packet_len = 0;
            printf("Found duplicate frame. Discarding...\n");
          }
        }

        if (write_to_tty(fd, reply, reply_len) != 0) {
          printf("Error write_to_tty() in function llread().\n");
          return -1;
        }
    }

      if(!read_succesful)
        ignore_flag = 1;


    }

    return 0;
}

int llclose(int fd) {
    char *frame;
    int frame_len = 0;

    if (data_link.stat == TRANSMITTER) {
      frame = create_US_frame(&frame_len, DISC);
      if (send_US_frame(fd, frame, frame_len, is_frame_DISC) != 0) {
        printf("Couldn't send frame on llclose().\n");
        reset_settings(fd);
        return -1;
      }

      if (write_to_tty(fd, create_US_frame(&frame_len, UA), frame_len) != 0) {
        printf("Couldn't write to tty on llclose()\n");
        reset_settings(fd);
        return -1;
      }

    } else {
      char msg[256];
      int msg_len = 0;

      if (read_from_tty(fd, msg, &msg_len) != 0) {
        printf("Couldn't read from tty on llclose()\n");
        reset_settings(fd);
        return -1;
```

```
    }

    if (is_frame_DISC(msg)){
      close_receiver_connection(fd);
    }

    if (reset_settings(fd) == 0)
        printf("Connection succesfully closed.\n");
    }

  return 0;
}

void print_as_hexadecimal(char *msg, int msg_len) {
  int i;
  for (i = 0; i < msg_len; i++)
    printf("%02X ", msg[i] & 0xFF);
  fflush(stdout);
}

void timeout(int signum) {

  total_time_outs++;
  connection_timeouts++;
}

char *create_US_frame(int *frame_len, int control_byte) {
  static char r = 0;
  char *buf = (char *)malloc(US_FRAME_LENGTH * sizeof(char));
  buf[0] = FLAG;

  if (data_link.stat == TRANSMITTER) {
    if (control_byte == SET || control_byte == DISC)
      buf[1] = SEND;
    else
      buf[1] = RECEIVE;
  } else {
    if (control_byte == RR || control_byte == REJ || control_byte == UA)
      buf[1] = SEND;
    else
      buf[1] = RECEIVE;
  }

  if (control_byte == RR || control_byte == REJ) {
    buf[2] = r << 7 | control_byte;
    r = !r;
```

```
  } else
    buf[2] = control_byte;

  buf[3] = buf[1] ^ buf[2];
  buf[4] = FLAG;
  *frame_len = US_FRAME_LENGTH;

  return buf;
}

int write_to_tty(int fd, char *buf, int buf_length) {
  int total_written_chars = 0;
  int written_chars = 0;

  while (total_written_chars < buf_length) {
    written_chars = write(fd, buf, buf_length);

    if (written_chars <= 0) {
      printf("Written chars: %d\n", written_chars);
      printf("%s\n", strerror(errno));
      return -1;
    }

    total_written_chars += written_chars;
  }

  return 0;
}

char *create_I_frame(int *frame_len, char *packet, int packet_len) {
  static char s = 1;
  s = !s;

  // Calculate BCC2
  char bcc2 = 0;

  int i;
  for (i = 0; i < packet_len; i++)
    bcc2 ^= packet[i];

  // This is executed here in order to set the correct array size.
  int bcc_len = 1;
  char *stuffed_bcc = stuff(&bcc2, &bcc_len);
  char *stuffed_packet = stuff(packet, &packet_len);

  *frame_len = 5 + packet_len + bcc_len;
```

```
  char *frame = (char *)malloc(*frame_len * sizeof(char));

  frame[0] = FLAG;
  frame[1] = SEND;
  frame[2] = s << 6;
  frame[3] = frame[1] ^ frame[2];
  memcpy(frame + 4, stuffed_packet, packet_len);          // Copy packet content
  memcpy(frame + packet_len + 4, stuffed_bcc, bcc_len); // Copy bcc2

  frame[packet_len + 4 + bcc_len] = FLAG;
  return frame;
}

int read_from_tty(int fd, char *frame, int *frame_len) {
  int read_chars = 0;
  char buf;
  *frame_len = 0;
  int initial_flag = 0;

  STOP = 0;
  while (!STOP) {                    /* loop for input */
    read_chars = read(fd, &buf, 1); /* returns after x chars have been input */

    if (read_chars > 0) { // If characters were read
      if (buf == FLAG) { //If the char is a FLAG
        //Set frame start to true.
        if(!ignore_flag) {
          initial_flag = !initial_flag;

          //If it is the second flag, then the frame
          //has ended
          if (!initial_flag)
          STOP = 1;

          frame[*frame_len] = buf;
          (*frame_len)++;
        } else
          ignore_flag = 0;
      } else {
        //If the char is not a flag and
        //the final flag has not been found
        //then add it to the frame.
        if(initial_flag) {
          frame[*frame_len] = buf;
          (*frame_len)++;
        }
```

```
      }
    } else {// If no characters were read or there was an error
      //printf("Error!!! %s\n", strerror(errno));
      return -1;
    }
  }

  return 0;
}

void destuff(char *packet, char *destuffed, int *packet_len) {
  int destuff_iterator = 0;

  int i;
  for (i = 0; i < *packet_len; i++) {
    if (packet[i] == ESCAPE) {
      destuffed[destuff_iterator] = packet[i + 1] ^ STUFFING_BYTE;
      i++;
    } else
      destuffed[destuff_iterator] = packet[i];
    destuff_iterator++;
  }

  *packet_len = destuff_iterator;
}

int is_frame_UA(char *reply) {

  return (reply[0] == FLAG &&
          reply[1] == ((data_link.stat == TRANSMITTER) ? SEND : RECEIVE) &&
          reply[2] == UA && reply[3] == (reply[1] ^ reply[2]) &&
          reply[4] == FLAG);
}

int is_frame_DISC(char *reply) {
  return (reply[0] == FLAG &&
          reply[1] == ((data_link.stat == TRANSMITTER) ? RECEIVE : SEND) &&
          reply[2] == DISC && reply[3] == (reply[1] ^ reply[2]) &&
          reply[4] == FLAG);
}

// is_reply_valid is a function that checks if the reply received is
// valid.
// If it is, the send_US_frame ends correctly. If not, it continues its
// loop.
int send_US_frame(int fd, char *frame, int len, int (*is_reply_valid)(char *)) {
```

```
  char reply[255];
  int reply_len;

  connection_timeouts = 0;
  while (connection_timeouts <= data_link.num_retries + 1) {
    if (write_to_tty(fd, frame, len)) {
      printf("Failed write.\n");
      return -1;
    }

    alarm(data_link.timeout);

    if (read_from_tty(fd, reply, &reply_len) == 0) {
      connection_timeouts = 0;
      // If the read() was successful
      if (is_reply_valid(reply)) {
          alarm(0);
          return 0;
      }
    }

    alarm(0);
    if (connection_timeouts > 0 && connection_timeouts < data_link.num_retries)
    printf("Connection failed. Retrying %d out of %d...\n",
    connection_timeouts, data_link.num_retries);
  }

  return -1;
}

int send_I_frame(int fd, char *frame, int len) {

  char reply[255];
  int reply_len;

  connection_timeouts = 0;
  while (connection_timeouts <= data_link.num_retries + 1) {
    if (write_to_tty(fd, frame, len)) {
      printf("Failed write.\n");
      return -1;
    }

    alarm(data_link.timeout);

    if (read_from_tty(fd, reply, &reply_len) == 0) {
      // If the read() was successful
```

```
      if(connection_timeouts > 0)
      printf("Connection reestablished. Resuming...\n");

      connection_timeouts = 0;

      alarm(0);
      // If a RR is received, proceed to the next frame
      if (is_frame_RR(reply, reply_len)) {
          r = !r;
          return 0;
      } else if(is_frame_REJ(reply, reply_len)) {
        // If a REJ is received, resend the frame.
        connection_timeouts = 0;
      }
    }

    if (connection_timeouts > 0 && connection_timeouts <= data_link.num_retries)
      printf("Connection failed. Retrying %d out of %d...\n",
        connection_timeouts, data_link.num_retries);
    alarm(0);
  }

  return -1;
}

int is_frame_RR(char *reply, int reply_len) {
  if(reply_len != 5)
    return 0;

  return (
      reply[0] == (unsigned char)FLAG &&
      reply[1] ==
          (unsigned char)((data_link.stat == TRANSMITTER) ? SEND : RECEIVE) &&
      (unsigned char)reply[2] == (unsigned char)(r << 7 | RR) &&
      (unsigned char)reply[3] == (unsigned char)(reply[1] ^ reply[2]) &&
      reply[4] == (unsigned char)FLAG);
}

int is_frame_REJ(char *reply, int reply_len) {
  if(reply_len != 5)
    return 0;

  return (
      reply[0] == (unsigned char)FLAG &&
      reply[1] ==
          (unsigned char)((data_link.stat == TRANSMITTER) ? SEND : RECEIVE) &&
```

```
      (unsigned char)reply[2] == (unsigned char)(r << 7 | REJ) &&
      (unsigned char)reply[3] == (unsigned char)(reply[1] ^ reply[2]) &&
      reply[4] == (unsigned char)FLAG);
}

char *stuff(char *packet, int *packet_len) {
  // TODO: Variable size.
  char *stuffed = (char *)malloc(((*packet_len) + 255) * sizeof(char));

  int packet_iterator;
  int stuff_iterator = 0;

  for (packet_iterator = 0; packet_iterator < *packet_len; packet_iterator++) {

    if (packet[packet_iterator] == ESCAPE || packet[packet_iterator] == FLAG) {

      stuffed[stuff_iterator] = ESCAPE;
      stuffed[++stuff_iterator] = packet[packet_iterator] ^ STUFFING_BYTE;
    } else
      stuffed[stuff_iterator] = packet[packet_iterator];

    stuff_iterator++;
  }

  *packet_len = stuff_iterator;
  return stuffed;
}

int has_valid_sequence_number(char control_byte, int s) {
  return (control_byte == (s << 6));
}

int close_receiver_connection(int fd) {
  int frame_len = 0;
  char *frame = create_US_frame(&frame_len, DISC);
  if (send_US_frame(fd, frame, frame_len, is_frame_UA) != 0) {
    printf("Couldn't send frame on llclose().\n");
    return -1;
  }

  return 0;
}

int reset_settings(int fd) {
  if (sigaction(SIGALRM, &data_link.old_action, NULL) == -1)
    printf("Error setting SIGALRM handler to original.\n");
```

```
  if (tcsetattr(fd, TCSANOW, &old_port_settings) == -1)
    printf("Error settings old port settings.\n");

  if (close(fd)) {
    printf("Error closing terminal file descriptor.\n");
    return -1;
  }

  return 0;

}

void force_close(int fd) {
  printf("Forcing close application...\n");
  reset_settings(fd);
  close(fd);
}

void init_data_link(int time_out,int number_retries, int baudrate){

  data_link.timeout=time_out;
  data_link.num_retries=number_retries;
  data_link.baudrate = baudrate;

}

int getTotalTimeouts(){
  return total_time_outs;
}
```

A.5 interface.c

```
#include "data_link_layer.h"
#include "application_layer.h"
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

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

void select_parameters(){

  int temp_num_transmissions;
  int num_transmissions_flag=0;
  int temp_time_out;
  int time_out_flag=0;
  int chosen_baudrate;
  int baudrate_flag = 0;
  int baudrate;

  printf("Choose the number of times you want to retry on timeout: ");

  do{

    scanf("%d",&temp_num_transmissions);

    if(temp_num_transmissions > 0 && temp_num_transmissions<=6){
      printf("Number of retries will be %d.\n",temp_num_transmissions);
      num_transmissions_flag=1;
    }
    else{
      printf("\nInvalid value, choose again [1,...,6]: ");
    }


  }while(!num_transmissions_flag);


  printf("Choose the time for timeout: ");
```

```
do{

  scanf("%d",&temp_time_out);

  if(temp_time_out > 0 && temp_time_out <=6){
    printf("Timeout between the transmissions will be %d.\n",temp_time_out);
    time_out_flag=1;
  }
  else
  {
    printf("\nInvalid value, choose again [1,...,6]: ");
  }

}while(!time_out_flag);


printf("Choose Baud Rate:\n1 - B2400\n2 - B4800\n3 - B9600\n\n");
do{

  scanf(" %d",&chosen_baudrate);

  if(chosen_baudrate > 0 && chosen_baudrate <= 3){
    baudrate_flag=1;
  }
  else
  {
    printf("\nInvalid value, choose again [1,...,3]: ");
  }

}while(!baudrate_flag);

switch(chosen_baudrate){
  case 1:
    baudrate = B2400;
    printf("Baudrate: B2400\n");
    break;

  case 2:
    baudrate = B4800;
    printf("Baudrate: B4800\n");
    break;

  case 3:
    baudrate = B9600;
    printf("Baudrate: B9600\n");
```

```
      break;
}

  init_data_link(temp_time_out,temp_num_transmissions, baudrate);

}

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

  if ((argc < 2) || ((strcmp("/dev/ttyS0", argv[1]) != 0) &&         // For developme
                     (strcmp("/dev/ttyS1", argv[1]) != 0))) {       // and /dev/ttyS
    printf("Usage:\tnserial <SerialPort>\n\tex: nserial /dev/ttyS1\n");
    exit(1);
  }

  printf("Hello ! Please choose the mode, Receiver(R)/Transmitter(T): ");
  int flagMode=0;
  char mode[1];

  do{

    //mode=getchar();

    scanf("%s",mode);

    if(strcmp(mode,"R")==0 || strcmp(mode,"T")==0){
      flagMode=1;
    }
    else{
      //fflush(stdout);
      printf("\nNot a possible mode, please select again: ");
    }

  }while(!flagMode);

  select_parameters();

  if(strcmp(mode,"R")==0){ //RECEIVER

    printf("\nRECEIVER.\n\n");

    int fd = set_up_connection(argv[1], RECEIVER);

    if (fd < 0) {
      printf("Error opening file descriptor. Exiting...\n");
      return -1;
```

```
    }

    receive_data();

    llclose(fd);

  } else if(strcmp(mode,"T")==0){ //TRANSMITTER

    printf("\nTRANSMITTER.\n\n");

    int fd = set_up_connection(argv[1], TRANSMITTER);

    if(fd < 0) {
      printf("Error opening file descriptor. Exiting...\n");
      return -1;
    }

    char path[] = ".";
    char filename[] = "pinguim.gif";
    if(send_data(path, filename) == -1)
      force_close(fd);
    else
      llclose(fd);


  }

  return 0;

}
```

A.6 makefile

```
CC=gcc
CFLAGS=-Wall -I.

nserial: interface.c data_link_layer.o application_layer.o
	$(CC) interface.c data_link_layer.o application_layer.o -o nserial $(CFLAGS)

clean:
	rm -f nserial *.o
```