Redes de Computadores

1º Trabalho Laboratorial

MIEIC

(21 de Outubro de 2018)

Miguel Moura	up201609149@fe.up.pt
Pedro Fernandes	up201603846@fe.up.pt
João Miguel	up201604241@fe.up.pt

ÍNDICE

Sumário

Este relatório foi elaborado no âmbito da U.C de Redes de Computadores(RCOM), como parte da entrega do primeiro trabalho laboratorial, "Protocolo de ligação de dados".

Este primeiro trabalho consiste no desenvolvimento de uma aplicação capaz de enviar ficheiros de um computador para outro, por via de uma porta série.

O trabalho foi concluído com sucesso uma vez que a transferência de dados ocorre sem erros e cumpre todos os requerimentos

Introdução

O objetivo deste primeiro trabalho é implementar um protocolo de ligação de dados entre dois computadores, por via de uma porta série. Esta porta série é usada por ser um mecanismo simples, facilitando a compreensão de uma transmissão de dados a baixo nível.

Este relatório serve como documentação de todo o projeto bem como a explicação da sua componente teórica e segue a seguinte estrutura:

- **Arquitetura**

  Apresentação dos blocos funcionais e interfaces presentes.

- **Estrutura do código**

  Apresentação das *APIs*, das principais estruturas de dados, das principais funções e sua relação com a arquitetura.

- **Casos de uso principais**

  Identificação dos principais casos de uso bem como as sequencias de chamadas de funções

- **Protocolo de ligação lógica**

  Identificação dos principais aspetos funcionais bem como a descrição da estratégia de implementação dos mesmos, complementada com a apresentação de extratos de código.

- **Protocolo de aplicação**

  Identificação dos principais aspetos funcionais bem como a descrição da estratégia de implementação dos mesmos, complementada com apresentação de extratos de código.

- **Validação**
  Descrição dos testes efetuados com apresentação quantificada dos resultados.

- **Eficiência do protocolo de ligação de dados**

  Caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido, usando uma comparação com um protocolo Stop&Wait

- **Conclusão**

  Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

O projeto está divido em duas camadas. A camada da aplicação do utilizador, onde ocorre a leitura e escrita de dados e a camada de ligação de dados, onde se encontra implementado o protocolo de transferência de dados.

Estas duas camadas estão ainda divididas no bloco do emissor e no bloco do recetor, que incorporam ambas as camadas.

Estrutura de código

O código está dividido por dois ficheiros de código principais, o **sender.c** e o **receiver.c**, sendo que estes recorrem ao **protocol.c** onde se encontram implementadas as funções do protocolo de transmissão de dados. Há ainda o módulo **utils.c** que contem simplesmente funções auxiliares.

O **sender.c**, é responsável pelas funções de emissão e o **receiver.c** pelas de receção.

São descritas em seguida as principais funções da camada de aplicação e da camada de ligação:

Funções principais:

Camada de ligação:

- **llopen()**
- **llwrite()**
- **llread**()
- **llclose()**

A função llopen() é usada tanto no computador que envia como no que recebe e garante que está tudo pronto para a transmissão de dados. O emissor depois chama a função llwrite() para enviar a informação e o recetor chama a llread(), que fica á espera da receção de uma trama de informação.

No fim da transmissão, tanto o recetor quanto o emissor chamam a função llclose().

Camada de aplicação:

- **getDelimPackage()**,
- **getFragment**()
- **readFile()**
- **transferFile()**
- **receiveFile()**

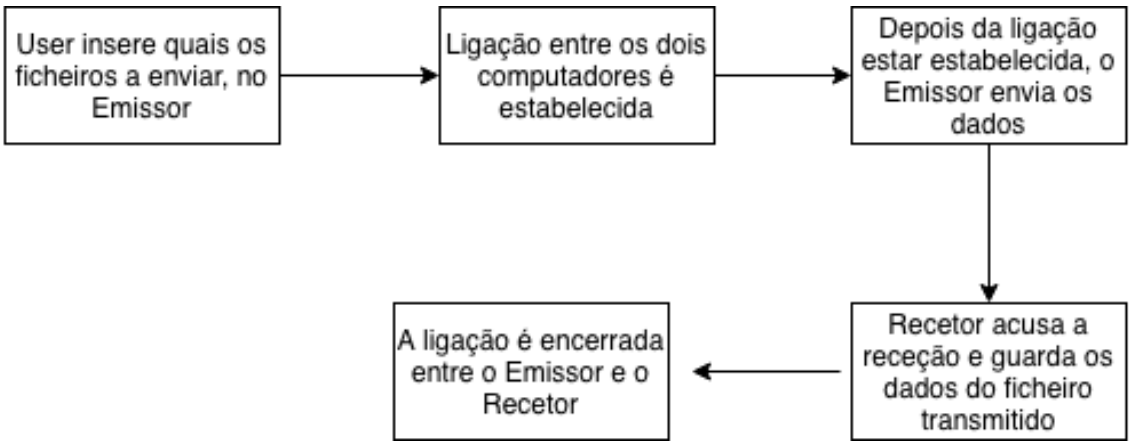
Estas são as principais funções da camada da aplicação, responsáveis pela receção e emissão de tramas. Os pacotes de dados e de controlo são também processados por estas funções, e posteriormente enviados.

Casos de uso principais

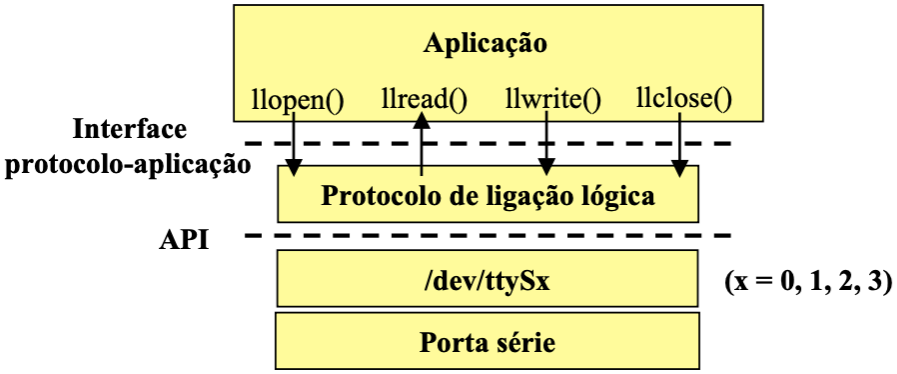
Esta aplicação tem dois principais casos de uso: Um, que ocorre quando está a funcionar em modo **Emissor** e outro que ocorre quando está em modo **Recetor**.

Em ambos os casos existe interação com o utilizador. Quando está a operar em modo Emissor, deve indicar qual a porta série a ser usada e o ficheiro que quer enviar.

Quando opera em modo Recetor deve inserir apenas a porta série.



Esquema da transmissão de dados.



Está, na imagem acima, esquematizada a interface Protocolo-Aplicação.

Protocolo de ligação lógica

O protocolo de ligação de dados, implementado no protocolo de ligação lógica, tem como objetivo estabelecer a ligação entre dois sistemas, por meio de uma porta-série.

A ligação lógica tem como objetivos: configurar a porta-série para a transmissão de dados, tendo em consideração a configuração prévia, estabelecer, efetivamente, essa ligação, enviar mensagens e/ou comandos através da mesma e é responsável pelo processo de *stuffing* e *destuffing* dos pacotes a serem transmitidos. Em todos estes passos devem ser verificados os erros.

Seguem-se as principais funções neste processo:

Alarmes:

```
void alarm_handler()
```

```
void setUpAlarmHandler()
```

Estas funções são responsáveis pela instalação de um *handler* para o sinal SIGALARM, que é usado para implementar o sistema de *timeout* do lado do emissor.

A cada receção do alarme, é ativada uma *flag* e incrementado um contador dos alarmes recebidos na atual função.

SetUpPort:

```
void setUpPort(int port, int *fd, struct termios *oldtio, speed_t baudrate)
```

Esta função e responsável por preparar o descritor da porta de série para a transferência de tramas. Coloca o *baudrate* passado como argumento (para facilitar o teste da eficiência) e guarda o estado anterior em *oldtio* para ser reposto no final do programa.

LLOPEN

```
int llopen(int flag, int fd)
```

Esta função tem a responsabilidade de estabelecer a ligação entre o emissor e o recetor.

Esta função é invocada pelo Emissor e envia a trama de controlo *SET* e ativa o temporizador que é desativado após receber resposta (*UA*).

Se não receber resposta dentro de um tempo *time-out*, a trama de controlo SET é reenviada. Este mecanismo de retransmissão só é repetido 3 vezes e se este número for atingido o programa termina.

No Recetor, esta função espera pela chegada de uma trama de controlo SET para responder com um UA.

LLWRITE

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

Esta é a função no emissor responsável pelo envio das tramas e pelo *stuffing* das mesmas.

Recebe um buffer no inicio e é feito o framing da mensagem, cálculo do BCC2 e o stuffing da mensagem através da função auxiliar **stuffing(unsigned char *data, unsigned char BCC2, int dataSize, int *size).**

Tenta depois enviar a mensagem através da porta-serie.

O envio da trama tem o mesmo mecanismo de *time-out* e retransmissão que o envio do *SET* no ***llopen***. Depois de enviar a trama é acionado um alarme até à receção de uma resposta (*RR* ou *REJ*) e se atingido esse alarme a mensagem é reenviada. Esta situação ocorre no máximo 3 vezes (limite imposto por nós).

LLREAD

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

Esta é a função no recetor responsável pela receção das tramas e pelo *destuffing* das mesmas. Tudo isto é feito através da chamada à função **receivedIMessage().**

A leitura é feita carater a carater e todas as verificações são feitas dentro desta função. Estas incluem verificar se não existem erros no cabeçalho da trama (BCC1) ou nos dados (BCC2). Neste último caso, dever-se-ia responder com um REJ. Tem-se também atenção à ocorrência de pacotes duplicados, para que não seja escrito no ficheiro informação não desejada.

LLCLOSE

```
int llclose(int fd, int flag)
```

Esta função tem a responsabilidade de terminar a ligação entre o emissor e o recetor.

Recebe uma *flag* como parâmetro que indica se foi invocada pelo Recetor ou pelo Emissor. Chama depois uma das funções **llclose_receiver()** ou **llclose_transmitter(),** de acordo com essa *flag*.

No emissor, é enviado a trama de Supervisão *DISC* com ajuda da função **sendSupervisionMessage().**

O recetor fica a aguardar um comando DISC. Aquando da receção do mesmo, é reenviado e passa a ficar á espera de um UA.

A ligação entre a porta-série é depois encerrada.

Protocolo de aplicação

A camada do protocolo de aplicação tem como objetivos: O envio dos pacotes de controlo(Start e End), a divisão do ficheiro em fragmentos aquando do envio e concatenação dos fragmentos recebidos no recetor. É ainda responsável pelo encapsulamento individual dos fragmentos de dados, bem como a leitura/criação do ficheiro.

Seguem-se as principais funções neste processo:

getDelimPackage()

```
unsigned char *getDelimPackage(unsigned char C, int fileLength, char *fileName,
                               int stringLength, int *size)
```

Esta função recebe como argumentos o comprimento do ficheiro, o nome do ficheiro, o comprimento do nome e o tamanho do ficheiro. O caracter C indica se estamos a lidar com um package de inicio (Start_C) ou de fim (End_C).

Com esta informação é obtido um *Delim Package* que contém *arrays* TLV com a informação do nome e tamanho do *package*.

readFile()

```
unsigned char *readFile(const char *fileName, off_t *size)
```

Esta é a função responsável por ler e guardar a informação do ficheiro cujo nome é passado pelo argumento **fileName.**

getFragment()

```
unsigned char *getFragment(int seqNum, unsigned char *data, int K)
```

Esta função é responsável pelo fragmento do objeto (*Data Fragment*). Este objeto é gerado com o seqNum passado como argumento, os dados a serem efetivamente guardados e o tamanho dos mesmos. O fragmento resultante está pronto para ser escrito usando **llwrite()**.

receiveFile()

```
int receiveFile(char *port);
```

Esta função é responsável por, no lado do recetor, efetuar todo o processo de transferência de um ficheiro. Recebendo como argumento o nome da porta de série, chama as funções da camada do protocolo, estabelecendo ligação lógica, recebendo os dados do ficheiro e criando um novo, e terminando a ligação.

transferFile()

```
int transferFile(char *fileName, char *port, unsigned char *fileData, off_t fileSize);
```

À semelhança da função anterior, atua no lado do transmissor de forma a percorrer todo o processo de transferência de um ficheiro. A diferença está no corpo da função, onde em vez de recorrer a **llread()**, usa **llwrite()**,s sendo que tem um ciclo onde percorre a informação do ficheiro, cria os fragmentos de dados, e envia pela porta de série.

Validação

De modo a testar a robustez do programa foram testadas inúmeras situações, entre elas:

- Envio de ficheiros de vários tamanhos.
- Criação de um curto-circuito a meio do envio do ficheiro.
- Interrupção da ligação por alguns segundos durante o envio de um ficheiro

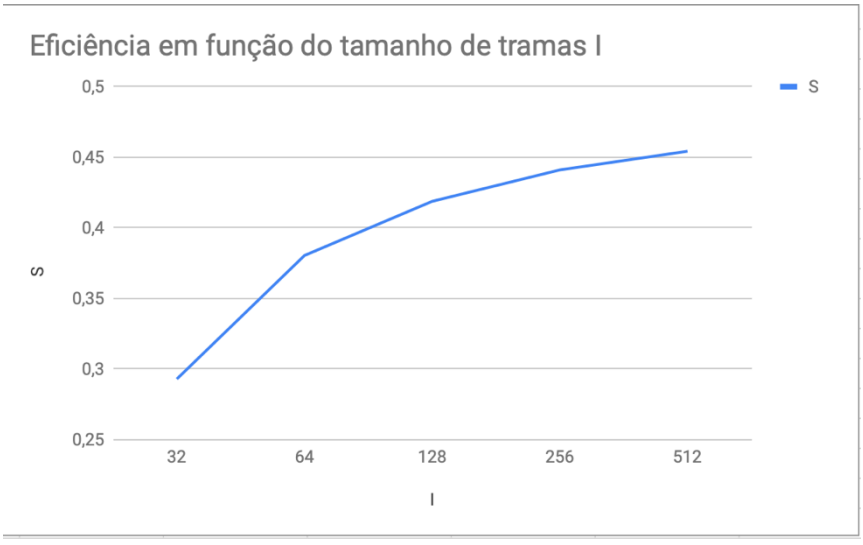
Todos os testes foram concluídos com sucesso.

Eficiência do Protocolo da ligação de dados

De modo a avaliar a eficiência do protocolo desenvolvido foram efetuados 4 testes diferentes. Cada um dos testes tem um gráfico respetivo. De frisar que todos os valores são obtidos através de log files implementadas no código.

Variação do tamanho de Tramas I:

É possível verificar que quanto maior o tamanho das tramas I, maior é a eficiência. Como a quantidade de informação por pacote aumenta, o numero de pacotes a enviar vai ser menor.

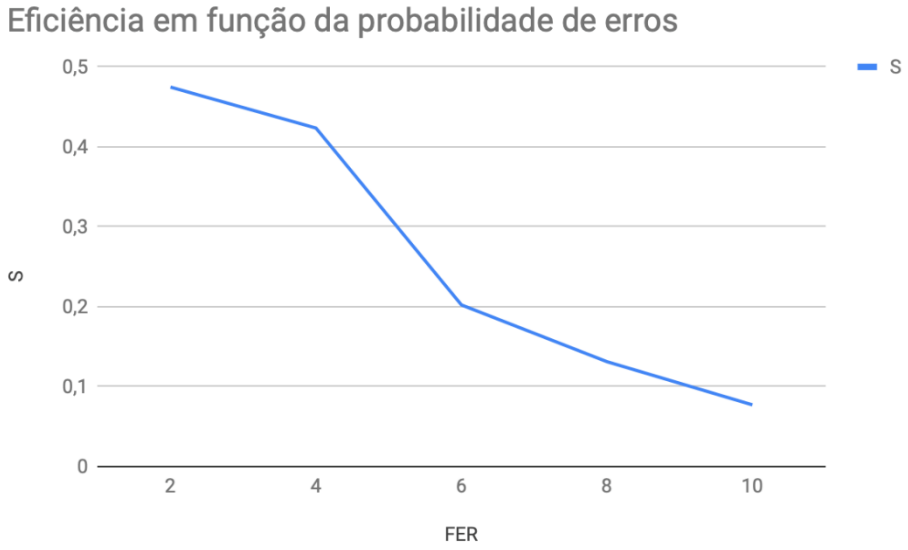


Variação da probabilidade de erros (FER):

A geração de erros do tipo BCC1 e BCC2 afeta consideravelmente a eficiência do programa.

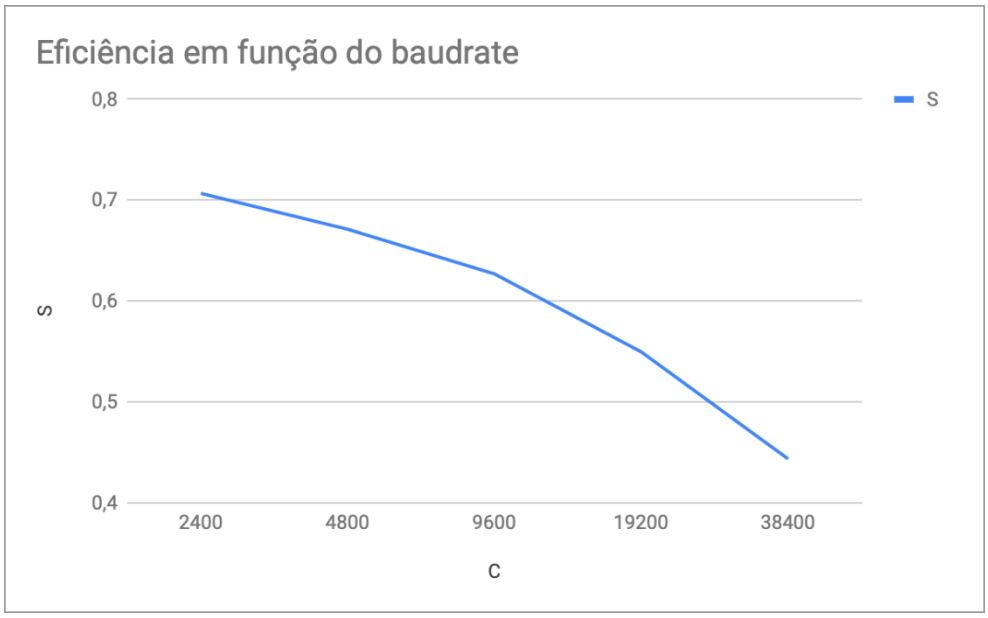
No caso dos erros do tipo BCC1, deve-se ao facto de que se este erro ocorrer, o recetor fica impedido de responder uma quantidade de segundos definida por nós. Quando se trata de eficiência, esses segundos tornam-se vitais.

No caso dos erros do tipo BCC2, o impacto não é tão grande porque estes apenas fazem com que a trama de informação seja reenviada.



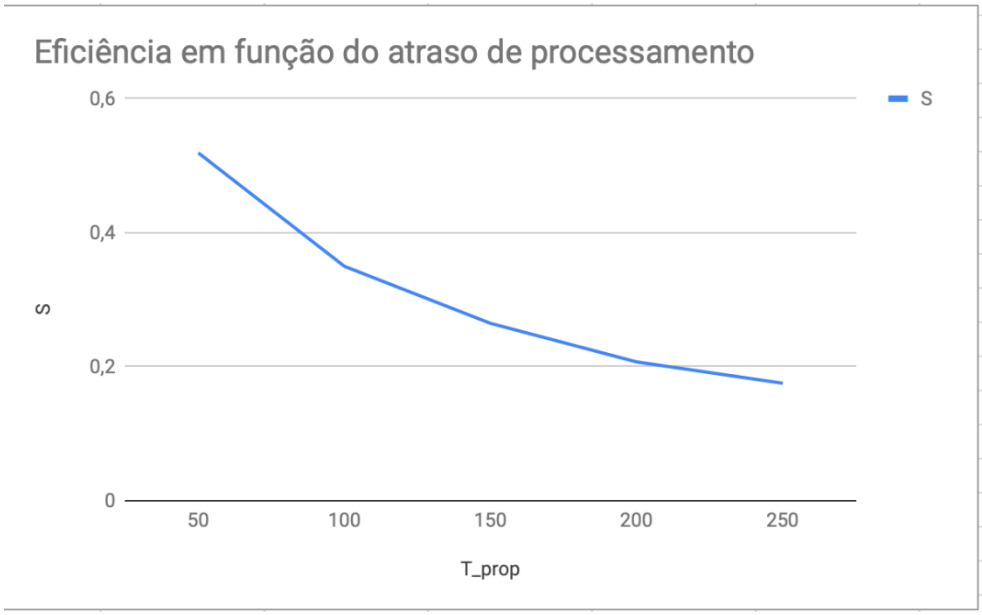
Variação do Baudrate:

Quanto maior o *baudrate* (Capacidade da ligação), menor a eficiência.



Variação do atraso de processamento:

Quanto maior o atraso do processamento (ms), menor a eficiência.



Conclusões

Em jeito de conclusão, é de salientar que todos os objetivos do trabalho foram cumpridos. O programa funciona como deve, resiste a todos os testes e toda a informação circula de forma segura e independente. Do ponto de vista interpessoal todos os objetivos foram também alcançados, com todos os membros do grupo a participar ativamente no projeto e a inteirarem-se a matéria em questão.

No decorrer do desenvolvimento e das aulas práticas foi nos dado a conhecer o conceito de independência entre camadas. Todo o projeto cumpre com esta prática. Na camada da ligação de dados não há nenhum processamento relativo aos pacotes que circulam, contendo as tramas de Informação.

Na camada da aplicação, não lhe são fornecidas informações relativas ao protocolo de ligação de dados, apenas a forma como o serviço é acedido.

Anexos

Receiver.h

```
#pragma once

#include <stdbool.h>
#include <stdio.h>

/**
 * @brief Prints the program correct usage and gives one example.
 *
 * @param argv
 * @return int
 */
int usage(char **argv);

/**
 * @brief Receives a Delim Package and get and save the file's information it
contains
 *
 * @param data : Delim Package
 * @param filename : Variable that will be set with the file's name saved at
the received Delim Package
 */
void handleStart(unsigned char *data, unsigned char *filename);

/**
 * @brief Receives a Fragment/Data Package and gets the data it contains (K
bytes of data)
 *        to save it at the file named 'filename'
 *
 * @param data : Fragment/Data Package
 * @param file : file where the Fragment's data will be saved
 * @return true
 * @return false
 */
bool handleData(unsigned char *data, FILE *file);

/**
 * @brief Reads the Start Delim Package and uses its information to create a
file (using handleStart).
 *        Reads the Fragments sent by the sender and saves the data they
contain at the created file (using handleData).
 *        Reads the End Delim.
 *        Read all the Packages from the Serial Port (with descriptor fd)
using llread function.
 *
```

```
 * @param fd : Serial Port Descriptor
 */
void readFile(int fd);

/**
 * @brief Enables protocol communication, receiving a file through the serial
port
 *
 * @param port
 * @return int
 */
int receiveFile(char *port);

/**
 * @brief Log test results to a text file
 *
 * @param stats file
 * @param time_spent time spent on file transfer
 * @param R transfer debit
 */
void log_test(FILE *stats, double time_spent, double R);
```

Receiver.c

```
#include "receiver.h"
#include "protocol.h"
#include "utils.h"

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

#define __USE_XOPEN
#include <unistd.h>

int length = 0, test_no = 0;
test_t test = INV;
useconds_t delays[NUMBER_OF_TESTS] = {50000, 100000, 150000, 200000, 250000};
speed_t baud_rates[NUMBER_OF_TESTS] = {B2400, B4800, B9600, B19200, B38400};
int baud_rates_i[NUMBER_OF_TESTS] = {2400, 4800, 9600, 19200, 38400};
const char *baud_rates_s[NUMBER_OF_TESTS] = {"B2400", "B4800", "B9600",
"B19200", "B38400"};
int messageSizes[NUMBER_OF_TESTS] = {32, 64, 128, 256, 512};

extern int fail_prob[NUMBER_OF_TESTS];

int usage(char **argv)
```

```
{
  printf("Usage: %s <COM>\n", argv[0]);
  printf("Option -t: allows to test effiency. Only one argument allowed, "
         "don\'t\nforget to use it on both sides.\n");
  printf("    Arguments: I - vary I message length\n");
  printf("               C - vary baudrate\n");
  printf("               T_prop - vary processing time\n");
  printf("               FER - vary frame error ratio\n");
  printf("ex: %s 0\n", argv[0]);

  return 1;
}

void handleStart(unsigned char *data, unsigned char *filename)
{
  unsigned char T = data[1];
  unsigned char size = data[2];

  if (T == T_LENGTH)
  {
    memcpy(&length, data + 3, size);
    memcpy(filename, data + size + 3 + 2, data[3 + size + 1]);
  }
  else if (T == T_NAME)
  {
    memcpy(filename, data + 3, size);
    memcpy(&length, data + size + 3 + 2, data[3 + size + 1]);
  }
}

bool handleData(unsigned char *data, FILE *file)
{
  unsigned char C = data[0];

  if (C == F_C)
  {
    int K = 256 * data[3] + data[2];

    fwrite(data + 4, 1, K, file);
  }

  return C == END_C;
}

void readFile(int fd)
{
  FILE *file;
  unsigned char filename[MAX_FILENAME_SIZE];
  unsigned char fragment[MAX_BUF_SIZE];
  unsigned char delim[DELIM_SIZE];
  bool end = false;
```

```
  int size = 0;

  if (llread(fd, delim) <= 0)
  {
    fprintf(stderr, "llread error\n");
    exit(-1);
  }

  handleStart(delim, filename);

  file = fopen((char *)filename, "wb+");

  while (!end)
  {
    if (test == T_prop)
      usleep(delays[test_no]);

    size = llread(fd, fragment);

    if (size == 0 || size == -1)
    {
      debug_print("llread error\n");
      continue;
    }

    if (size != -2)
      end = handleData(fragment, file);
  }

  fclose(file);
}

int receiveFile(char *port)
{
  int fd = 0;
  struct termios oldtio;
  speed_t baudrate = B38400;

  if (test == C)
    baudrate = baud_rates[test_no];

  printf("rate: %d\n", baudrate);

  setUpPort(atoi(port), &fd, &oldtio, baudrate);

  if (llopen(RECEIVER, fd) != 0)
  {
    fprintf(stderr, "llopen error\n");
    exit(-1);
  }
```

```
  readFile(fd);

  if (llclose(fd, RECEIVER) != 0)
  {
    fprintf(stderr, "llclose error\n");
    exit(-1);
  }

  closeFd(fd, &oldtio);

  return 0;
}

void log_test(FILE *stats, double time_spent, double R)
{
  char test_value[20];
  char test_type[50];

  switch (test)
  {
  case C:
    strcpy(test_type, "baudrate (bits/s)");
    strcpy(test_value, baud_rates_s[test_no]);
    break;
  case I:
    strcpy(test_type, "size (bytes)");
    sprintf(test_value, "%d", messageSizes[test_no]);
    break;
  case FER:
    strcpy(test_type, "prob (%)");
    sprintf(test_value, "%d", fail_prob[test_no]);
    break;
  case T_prop:
    strcpy(test_type, "delay (ms)");
    sprintf(test_value, "%d", delays[test_no] / 1000);
    break;
  default:
    break;
  }

  double S = test == C ? R/baud_rates_i[test_no] : R/38400;

  fprintf(stats,
          "test no.%d: %s - %s --- time taken (s) - %.2f  --- R (bit/s) - %f -
-- S - %f\n",
          test_no, test_type, test_value, time_spent, R, S);
}

int main(int argc, char **argv)
{
  int numTests = 1;
```

```
  FILE *stats;
  unsigned long long begin, end;
  double time_spent, R;
  const char *testNames[] = {"INV", "C", "I", "T_prop", "FER"};

  if ((argc != 2 && argc != 4) ||
      ((strcmp("0", argv[1]) != 0) && (strcmp("1", argv[1]) != 0)))
    return usage(argv);
  else if (argc == 4 && strcmp(argv[2], "-t") != 0)
    return usage(argv);
  else if (argc == 4 && ((test = processTestArgument(argv, 3)) == INV))
    return usage(argv);

  srand(time(NULL));
  startTime();
  stats = fopen("stats.txt", "w");

  if (test > INV)
  {
    numTests = NUMBER_OF_TESTS;
    fprintf(stats, "TEST TYPE %s\n", testNames[test]);

    if (test == T_prop)
      setUpAlarmHandler();
  }

  for (; test_no < numTests; test_no++)
  {
    begin = getTime();
    receiveFile(argv[1]);
    end = getTime();
    time_spent = (end - begin) / 100000.0;
    R = length * 8.0 / time_spent;
    log_test(stats, time_spent, R);
  }

  if (test > INV)
    printf("Please consult test results on stats.txt\n");

  fclose(stats);

  return 0;
}
```

Sender.h

```
#pragma once

#include <unistd.h>

#define FRAG_K 260

/**
 * @brief Prints the program correct usage and gives one example.
 *
 * @param argv : command line arguments
 * @return int
 */
int usage(char **argv);

/**
 * @brief Creates TLV array that saves file's lenght.
 *
 * @param fileLength : file's length to save in TLV array
 * @return unsigned char* : TLV array
 */
unsigned char *getTLVLength(int fileLength);

/**
 * @brief Creates TLV array that saves file's name.
 *
 * @param fileName : file's name to sabe in TLV array
 * @param stringLength : length of file's name
 * @return unsigned char* : TLV array
 */
unsigned char *getTLVName(char *fileName, int stringLength);

/**
 * @brief Get the Delim Package object
 *        The Delim Package contains TLV arrays with information of the file's
name and lenght
 *        generated by the getTLVLenght and getTLVName functions.
 *
 *
 * @param C : Control field that indicates if it's a Start or an End package
(START_C or END_C)
 * @param fileLength : File's lenght to save in the package
 * @param fileName : File's name to save in the package
 * @param stringLength : Length of file's name
 * @param size : Variable that will be updated with the package size
 * @return unsigned char* : Delim Package
 */
unsigned char *getDelimPackage(unsigned char C, int fileLength, char
*fileName,
                               int stringLength, int *size);
```

```
/**
 * @brief Reads the file with name 'fileName' and saves ist data and its size
 *
 * @param fileName : Name of the file to read
 * @param size : Variable that will be set with file's size
 * @return unsigned char* : file's data
 */
unsigned char *readFile(const char *fileName, off_t *size);

/**
 * @brief Get the Fragment object
 *        Generates one Data Package (Fragment) with the received sequence
number,
 *        the data to be saved and number of bytes of the data to be saved
 *
 * @param seqNum : sequence number of th Fragment
 * @param data : Data to be saved at the Fragment
 * @param K : Number of bytes of the Data that are stored in each Fragment
 * @return unsigned char* : Fragment
 */
unsigned char *getFragment(int seqNum, unsigned char *data, int K);

/**
 * @brief Sends the Start Delim Package
 *        Sends Fragments/Data Packages containing parts of the file (named
 * filename) data. Sends the End Delim Package to end the File transmition.
All
 * the Packages are sent trough the Serial Port (with descriptor fd) using
 * llwrite function. Fragments are created after reading the file data and
using
 * the getFragment function.
 *
 *
 * @param fd : Serial Port descriptor
 * @param filename : Name of the file to send
 * @param messageSize : Serial Port descriptor
 * @param fileData : Data of the file to send
 * @param fileSize : Size of the file to send
 */
void writeFile(int fd, char *filename, int messageSize, unsigned char*
fileData, off_t fileSize);

/**
 * @brief Using protocol functions, enables serial port communication and
 * writes the given file
 *
 * @param fileName Name of the file to send
 * @param port COM port
 * @param fileData Data of the file to send
 * @param fileSize Size of the file to send
 * @return int
```

```
 */
int transferFile(char *fileName, char *port, unsigned char *fileData, off_t
fileSize);
```

Sender.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "protocol.h"
#include "sender.h"
#include "utils.h"

speed_t baud_rates[NUMBER_OF_TESTS] = {B2400, B4800, B9600, B19200, B38400};
int messageSizes[NUMBER_OF_TESTS] = {32, 64, 128, 256, 512};
test_t test = INV;
int test_no = 0;

int usage(char **argv)
{
  printf("Usage: %s <COM> <FILENAME> [-t ...]\n", argv[0]);
  printf("Option -t: allows to test effiency. Only one argument allowed,
don\'t\nforget to use it on both sides.\n");
  printf("    Arguments: I - vary I message length\n");
  printf("               C - vary baudrate\n");
  printf("               T_prop - vary processing time\n");
  printf("               FER - vary frame error ratio\n");
  printf("ex: %s 0 pinguim.gif\n", argv[0]);

  return 1;
}

unsigned char *getTLVLength(int fileLength)
{
  int size = sizeof(fileLength);
  unsigned char *TLV = malloc(2 + size);

  TLV[0] = T_LENGTH;
  TLV[1] = size;
  memcpy(TLV + 2, &fileLength, size);

  return TLV;
}

unsigned char *getTLVName(char *fileName, int stringLength)
{
  unsigned char *TLV = malloc(2 + stringLength + 1);
```

```
  TLV[0] = T_LENGTH;
  TLV[1] = stringLength + 1;
  memcpy(TLV + 2, fileName, stringLength + 1);

  return TLV;
}

unsigned char *getDelimPackage(unsigned char C, int fileLength, char
*fileName,
                               int stringLength, int *size)
{
  *size = 1 + 2 + 2 + stringLength + 1 + sizeof(fileLength);
  unsigned char *delim = malloc(*size * sizeof(unsigned char));

  delim[0] = C;

  unsigned char *TLVLength = getTLVLength(fileLength);
  unsigned char *TLVName = getTLVName(fileName, stringLength);

  memcpy(delim + 1, TLVLength, 2 + sizeof(fileLength));

  memcpy(delim + 1 + 2 + sizeof(fileLength), TLVName, 2 + stringLength + 1);

  free(TLVLength);
  free(TLVName);

  return delim;
}

unsigned char *readFile(const char *fileName, off_t *size)
{
  unsigned char *data;
  FILE *file;
  struct stat info;

  if ((file = fopen(fileName, "rb+")) == NULL)
  {
    perror("fopen");
    exit(1);
  }

  if (stat(fileName, &info) != 0)
  {
    perror("stat");
    exit(1);
  }

  *size = info.st_size;
  data = malloc(*size);
```

```
  fread(data, sizeof(unsigned char), *size, file);

  fclose(file);

  return data;
}

unsigned char *getFragment(int seqNum, unsigned char *data, int K)
{
  unsigned char *fragment = malloc((4 + K) * sizeof(unsigned char));

  fragment[0] = F_C;
  fragment[1] = seqNum % 255;
  fragment[2] = K % 256;
  fragment[3] = K / 256;
  memcpy(fragment + 4, data, K);

  return fragment;
}

void writeFile(int fd, char *filename, int messageSize, unsigned char
*fileData, off_t fileSize)
{
  int delimSize = 0;

  unsigned char *start = getDelimPackage(START_C, fileSize, filename,
                                         strlen(filename), &delimSize);

  if (llwrite(fd, start, delimSize) <= 0)
  {
    fprintf(stderr, "llwrite error\n");
    exit(-1);
  }

  int rest = fileSize % messageSize;
  int numPackages = fileSize / messageSize;

  unsigned char *fragment;

  int i = 0;
  for (; i < numPackages; i++)
  {
    fragment = getFragment(i, fileData + messageSize * i, messageSize);

    if (llwrite(fd, fragment, messageSize + 4) <= 0)
    {
      fprintf(stderr, "llwrite error\n");
      exit(-1);
    }
  }
```

```
  if (rest != 0)
  {
    fragment = getFragment(i, fileData + messageSize * i, rest);
    if (llwrite(fd, fragment, rest + 4) <= 0)
    {
      fprintf(stderr, "llwrite error\n");
      exit(-1);
    }
  }

  unsigned char *end = malloc(delimSize * sizeof(unsigned char));
  memcpy(end, start, delimSize);
  end[0] = END_C;

  if (llwrite(fd, end, delimSize) <= 0)
  {
    fprintf(stderr, "llwrite error\n");
    exit(-1);
  }

  free(start);
  free(end);
  free(fragment);
}

int transferFile(char *fileName, char *port, unsigned char *fileData, off_t
fileSize)
{
  int fd = 0;
  struct termios oldtio;
  int messageSize = FRAG_K;
  speed_t baudrate = B38400;

  if (test == I)
    messageSize = messageSizes[test_no];
  else if (test == C)
    baudrate = baud_rates[test_no];

  printf("rate: %d\n", baudrate);
  printf("I size: %d\n", messageSize);

  setUpPort(atoi(port), &fd, &oldtio, baudrate);

  if (llopen(TRANSMITTER, fd) != 0)
  {
    fprintf(stderr, "llopen error\n");
    exit(-1);
  }

  writeFile(fd, fileName, messageSize, fileData, fileSize);
```

```
  if (llclose(fd, TRANSMITTER) != 0)
  {
    fprintf(stderr, "llclose error\n");
    exit(-1);
  }

  closeFd(fd, &oldtio);

  return 0;
}

int main(int argc, char **argv)
{
  int numTests = 1;
  off_t fileSize = 0;
  unsigned char *fileData;

  if ((argc != 3 && argc != 5) ||
      ((strcmp("0", argv[1]) != 0) &&
       (strcmp("1", argv[1]) != 0)))
    return usage(argv);
  else if (argc == 5 && strcmp("-t", argv[3]) != 0)
    return usage(argv);
  else if (argc == 5 && ((test = processTestArgument(argv, 4)) == INV))
    return usage(argv);

  if (test > INV)
    numTests = NUMBER_OF_TESTS;

  fileData = readFile(argv[2], &fileSize);

  for (; test_no < numTests; test_no++)
    transferFile(argv[2], argv[1], fileData, fileSize);

  if (test > INV)
    printf("Please consult test results on stats.txt\n");

  free(fileData);

  return 0;
}
```

Protocol.h

```
#pragma once

#include <stdbool.h>
#include <termios.h>

typedef enum
{
```

```
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC1_OK,
    DATA_RCV,
    BCC2_OK,
    END
} state_t;

#define TRANSMITTER 0
#define RECEIVER 1

#define FLAG 0x7E
#define ESC 0x7D
#define ESC_2 0x5E
#define ESC_3 0x5D
#define A_03 0x03
#define A_01 0x01
#define SET_C 0x03
#define UA_C 0x07
#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81
#define C_I0 0x00
#define C_I1 0x40
#define START_C 0x02
#define F_C 0x01
#define END_C 0x03
#define T_LENGTH 0x00
#define T_NAME 0x01
#define DISC 0x0B

#define TIME_OUT 3
#define MAX_RETRY_NUMBER 3

#define BAUDRATE B38400
#define SUPERVISION_SIZE 5
#define PORT_SIZE 11

/**
 * @brief Alarm Handler, function that is called when an alarm signal is
received
 *
 */
void alarm_handler();

/**
 * @brief Setup timeout alarm handler
 *
```

```
 */
void setUpAlarmHandler();

/**
 * @brief Opens the conection between the two computers trought the serial
port (associated to descripor fd).
 *        If flag equals TRANSMITTER calls llopen_transmitter to open the
connection in the sender program.
 *        If flag equals RECEIVER calls llopen_receiver to open the connection
in the receiver program.
 *
 * @param flag : Indicates if the connetcion is done in the receiver or in the
sender program/computer (TRANSMITTER/SENDER)
 * @param fd : Descriptor of the serial port, used to send the supervision
messages to establish connection.
 * @return int : 0 if sucessfull, -1 if reveives a non existent flag.
 */
int llopen(int flag, int fd);

/**
 * @brief Send a SET message to receiver through the serial port (using write
and fd descriptor).
 *        Install alarm handler, activated in periods of TIME_OUT seconds.
 *        Waits TIME_OUT seconds to receive a UA_C message from receiver,
readinf the serial port (using fd descriptor).
 *        If UA_C is not received in TIME_OUT seconds, send another SET
message and waits again.
 *        If SET is sent MAX_RETRY_NUMBER times without success, the
connection fails, and the function returns -1.
 *
 * @param fd : Serial port descriptor
 * @return int : 0 if connection is established successfully, -1 otherwise
 */
int llopen_transmitter(int fd);

/**
 * @brief Try to read SET message from sender, reading the serial port.
 *        If a SET message is received, indicating the sender wants to
establish a connection,
 *        it responds sending a UA_C message through the serial port,
indicating the receiver is ready.
 *
 * @param fd : Serial port descriptor
 * @return int : 0 if receives SET message and sends UA_C successfully, -1
otherwise
 */
int llopen_receiver(int fd);

/**
 * @brief Sets up serial port communication
 *
```

```
 * @param port
 * @param fd
 * @param oldtio
 */
void setUpPort(int port, int *fd, struct termios *oldtio, speed_t baudrate);

/**
 * @brief Writes a Package as an Information message in the Serial Port (using
its descriptor fd).
 *        Receives an application Package, applies the stuffing function to it
and
 *        uses it to create the Information message with calcFinalMessage
function.
 *        After the Information is sent to the receiver, it waits for its
answer indicating
 *        if the Informationmessage was correct and well received.
 *
 * @param fd : Serial Port descriptor
 * @param buffer : Package to send
 * @param length :Package's length
 * @return int : 0 if Information message was successfully sent, -1 otherwise
 */
int llwrite(int fd, unsigned char *buffer, int length);

/**
 * @brief Receives original Fragment and returns it stuffed
 *
 * @param data : non stuffed Fragment
 * @param BCC2 : Fragments's size
 * @param dataSize : Fragments's size
 * @param size : Variable to be set with stuffed fragment's size
 * @return unsigned char* : Stuffed Fragment
 */
unsigned char *stuffing(unsigned char *data, unsigned char BCC2, int dataSize,
int *size);

/**
 * @brief Calculates BBC2 that protects the Fragment of data that will be
saved at the information message.
 *
 * @param data : Fragment
 * @param size : Fragment's size
 * @return unsigned char : BBC2
 */
unsigned char calcBCC2(unsigned char *data, int size);

/**
 * @brief Creates Information Message receiving the Fragment/Data Package
generated at the application
 *        level and the BBC2 calculated previously
 *
```

```
 * @param data : Fragment/Data Package
 * @param size : Fragment's size
 * @return unsigned char* : Information message
 */
unsigned char *calcFinalMessage(unsigned char *data, int size);

/**
 * @brief Reads an Information Message from the Serial Port (using fd
descriptor).
 *
 * @param fd : Serial Port descriptor
 * @param buffer
 * @return int : -2 if reads an END message, -1 in case of invalid state
 */
int llread(int fd, unsigned char *buffer);

/**
 * @brief Receives rec_BCC2 and verifies if it's equal the BBC2 of the
received Fragment/Data Package.
 *
 *
 * @param rec_BCC2
 * @param data : Fragment/Data Package
 * @param size : Fragment's size
 * @return true : if equals
 * @return false : if not equals
 */
bool checkBCC2(unsigned char rec_BCC2, unsigned char *data, int size);

/**
 * @brief Receives the Fragment/Data Package, does it .
 *        When a FLAG is received it checks the Fragment's BCC2 and the
duplicates,
 *        responging with a signal according to it: if everything is correct
it sends
 *        RR0 or RR1, otherwise it sends REJ0 or REJ1.
 *
 * @param fd
 * @param buf
 * @param data
 * @param i
 * @param state
 * @param wait
 */
void receiveData(int fd, unsigned char buf, unsigned char *data, int *i,
state_t *state, bool *wait);

/**
 * @brief Verifies if an Information message is being received, trying to read
it from the Serial Port (using fd).
```

```
 *        Saves its data (Fragment) at the variable data (using receiveData
function).
 *        Saves the data's size at the variable size.
 *
 * @param fd : Serial Port descriptor
 * @param size
 * @return unsigned char*
 */
int receiveIMessage(int fd, int *size, unsigned char* data);


/**
 * @brief Ends the connection between the two computers/programs (sender and
receiver) through the Serial Port.
 *        If flag equals TRANSMITTER calls llclose_transmitter to close the
connection in the sender program.
 *        If flag equals RECEIVER calls llclose_receiver to close the
connection in the receiver program.
 *
 *
 * @param fd : Serial Port descriptor
 * @param flag : Indicates if the conn
 * @brief etcion is closed in the receiver or in the sender program/computer
(TRANSMITTER/SENDER)
 * @return int : 0 if sucessfull, -1 if reveives a non existent flag
 */
int llclose(int fd, int flag);

/**
 * @brief Try to read DISC message (sent by the llclose_transmitter) from the
Serial Port using fd.
 *        If it receives the DISC message, sends a DISC message to the
transmitter as answer trought the Serial Port.
 *        After sending the DISC, waits for a UA_C from the transmitter,
reading the Serial Port again.
 *
 * @param fd : Serial Port Descriptor
 * @return int : 0 if messages are received/sent and the connection closes
successfully, -1 otherwise
 */
int llclose_receiver(int fd);

/**
 * @brief Sends a DISC message to the transmitter trhough the Serial Port
(using fd) to close the conection.
 *        Waits TIME_OUT seconds for DISC answer from the llclose_receiver-
 *        If doesn't receive the DISC answer after this time, send another
DISC message.
 *        If doesn't receive an answer after trying to senD the DISC message
MAX_RETRY_NUMBER times, returns -1.
 *        If receives the answer, sends an UA_C as answer to the receiver.
```

```
 * @param fd : SetialPort descriptor
 * @return int : 0 if messages are received/sent and the connection closes
successfully, -1 otherwise
 */
int llclose_transmitter(int fd);

/**
 * @brief Check if a specific supervision message is received: with fields A
and C specified in the parameters.
 *        Try to read the message from the Serial Port using fd.
 *        Verifies if the message readed is the specified one and if it's
correct using stateMachineSupervisionMessage.
 *
 * @param fd : SerialPort descriptor.
 * @param A : Address field: A_01 or A_03
 * @param C : Control field: message type
 * @return int : 1 if the specified Supervision message is received
 */
int receiveSupervisionMessage(int fd, unsigned char A, unsigned char C);

/**
 * @brief Sends a supervision message with all fields needed: {FLAG, A, C,
A^C, FLAG}.
 *
 * @param fd : Serial Port descriptor
 * @param A : A_01 if the sender sends a command and the receiver sends the
answer to it, A_03 otherwise
 * @param C : Defines the message type (SET, UA, ...)
 * @return int
 */
int sendSupervisionMessage(int fd, unsigned char A, unsigned char C);

/**
 * @brief Receives a byte (field) of the supervision message readed from
Serial Port (buf).
 *        Changes the state parameter denpending on the previous state
received,
 *        the field readed from Serial Port and the A and C values.
 *        If state is set with END, a supervision message with A and C values
was readed.
 *
 *
 * @param state : Current state, will be uptaded
 * @param buf : Byte readed previously from Serial Port
 * @param A : Address field
 * @param C : Control field
 * @param COptions
 * @return int : -1 if receives an invalid state
 */
int stateMachineSupervisionMessage(state_t *state, unsigned char buf,
```

```
                                    unsigned char A, unsigned char *C,
                                    unsigned char *COptions);

/**
 * @brief Sets the original/old attributes of the Serial Port and closes it
using its descriptor.
 *
 * @param fd : Serial Port descriptor
 * @param oldtio : termios with Serial Port original (old) attributes
 */
void closeFd(int fd, struct termios *oldtio);
```

Protocol.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "protocol.h"
#include "sender.h"
#include "utils.h"

speed_t baud_rates[NUMBER_OF_TESTS] = {B2400, B4800, B9600, B19200, B38400};
int messageSizes[NUMBER_OF_TESTS] = {32, 64, 128, 256, 512};
test_t test = INV;
int test_no = 0;

int usage(char **argv)
{
  printf("Usage: %s <COM> <FILENAME> [-t ...]\n", argv[0]);
  printf("Option -t: allows to test effiency. Only one argument allowed,
don\'t\nforget to use it on both sides.\n");
  printf("    Arguments: I - vary I message length\n");
  printf("               C - vary baudrate\n");
  printf("               T_prop - vary processing time\n");
  printf("               FER - vary frame error ratio\n");
  printf("ex: %s 0 pinguim.gif\n", argv[0]);

  return 1;
}

unsigned char *getTLVLength(int fileLength)
{
  int size = sizeof(fileLength);
  unsigned char *TLV = malloc(2 + size);

  TLV[0] = T_LENGTH;
  TLV[1] = size;
```

```
  memcpy(TLV + 2, &fileLength, size);

  return TLV;
}

unsigned char *getTLVName(char *fileName, int stringLength)
{
  unsigned char *TLV = malloc(2 + stringLength + 1);

  TLV[0] = T_LENGTH;
  TLV[1] = stringLength + 1;
  memcpy(TLV + 2, fileName, stringLength + 1);

  return TLV;
}

unsigned char *getDelimPackage(unsigned char C, int fileLength, char
*fileName,
                               int stringLength, int *size)
{
  *size = 1 + 2 + 2 + stringLength + 1 + sizeof(fileLength);
  unsigned char *delim = malloc(*size * sizeof(unsigned char));

  delim[0] = C;

  unsigned char *TLVLength = getTLVLength(fileLength);
  unsigned char *TLVName = getTLVName(fileName, stringLength);

  memcpy(delim + 1, TLVLength, 2 + sizeof(fileLength));

  memcpy(delim + 1 + 2 + sizeof(fileLength), TLVName, 2 + stringLength + 1);

  free(TLVLength);
  free(TLVName);

  return delim;
}

unsigned char *readFile(const char *fileName, off_t *size)
{
  unsigned char *data;
  FILE *file;
  struct stat info;

  if ((file = fopen(fileName, "rb+")) == NULL)
  {
    perror("fopen");
    exit(1);
  }

  if (stat(fileName, &info) != 0)
```

```
  {
    perror("stat");
    exit(1);
  }

  *size = info.st_size;
  data = malloc(*size);

  fread(data, sizeof(unsigned char), *size, file);

  fclose(file);

  return data;
}

unsigned char *getFragment(int seqNum, unsigned char *data, int K)
{
  unsigned char *fragment = malloc((4 + K) * sizeof(unsigned char));

  fragment[0] = F_C;
  fragment[1] = seqNum % 255;
  fragment[2] = K % 256;
  fragment[3] = K / 256;
  memcpy(fragment + 4, data, K);

  return fragment;
}

void writeFile(int fd, char *filename, int messageSize, unsigned char
*fileData, off_t fileSize)
{
  int delimSize = 0;

  unsigned char *start = getDelimPackage(START_C, fileSize, filename,
                                         strlen(filename), &delimSize);

  if (llwrite(fd, start, delimSize) <= 0)
  {
    fprintf(stderr, "llwrite error\n");
    exit(-1);
  }

  int rest = fileSize % messageSize;
  int numPackages = fileSize / messageSize;

  unsigned char *fragment;

  int i = 0;
  for (; i < numPackages; i++)
  {
    fragment = getFragment(i, fileData + messageSize * i, messageSize);
```

```
    if (llwrite(fd, fragment, messageSize + 4) <= 0)
    {
      fprintf(stderr, "llwrite error\n");
      exit(-1);
    }
  }

  if (rest != 0)
  {
    fragment = getFragment(i, fileData + messageSize * i, rest);
    if (llwrite(fd, fragment, rest + 4) <= 0)
    {
      fprintf(stderr, "llwrite error\n");
      exit(-1);
    }
  }

  unsigned char *end = malloc(delimSize * sizeof(unsigned char));
  memcpy(end, start, delimSize);
  end[0] = END_C;

  if (llwrite(fd, end, delimSize) <= 0)
  {
    fprintf(stderr, "llwrite error\n");
    exit(-1);
  }

  free(start);
  free(end);
  free(fragment);
}

int transferFile(char *fileName, char *port, unsigned char *fileData, off_t
fileSize)
{
  int fd = 0;
  struct termios oldtio;
  int messageSize = FRAG_K;
  speed_t baudrate = B38400;

  if (test == I)
    messageSize = messageSizes[test_no];
  else if (test == C)
    baudrate = baud_rates[test_no];

  printf("rate: %d\n", baudrate);
  printf("I size: %d\n", messageSize);

  setUpPort(atoi(port), &fd, &oldtio, baudrate);
```

```
  if (llopen(TRANSMITTER, fd) != 0)
  {
    fprintf(stderr, "llopen error\n");
    exit(-1);
  }

  writeFile(fd, fileName, messageSize, fileData, fileSize);

  if (llclose(fd, TRANSMITTER) != 0)
  {
    fprintf(stderr, "llclose error\n");
    exit(-1);
  }

  closeFd(fd, &oldtio);

  return 0;
}

int main(int argc, char **argv)
{
  int numTests = 1;
  off_t fileSize = 0;
  unsigned char *fileData;

  if ((argc != 3 && argc != 5) ||
      ((strcmp("0", argv[1]) != 0) &&
       (strcmp("1", argv[1]) != 0)))
    return usage(argv);
  else if (argc == 5 && strcmp("-t", argv[3]) != 0)
    return usage(argv);
  else if (argc == 5 && ((test = processTestArgument(argv, 4)) == INV))
    return usage(argv);

  if (test > INV)
    numTests = NUMBER_OF_TESTS;

  fileData = readFile(argv[2], &fileSize);

  for (; test_no < numTests; test_no++)
    transferFile(argv[2], argv[1], fileData, fileSize);

  if (test > INV)
    printf("Please consult test results on stats.txt\n");

  free(fileData);

  return 0;
}
```

Utils.h

```c
#pragma once

#include <stdbool.h>

#define MAX_BUF_SIZE 600
#define NUMBER_OF_TESTS 5
#define NUMBER_OF_ALARMS 1
#define MAX_FILENAME_SIZE 50
#define DELIM_SIZE 21

typedef enum {
  INV, C, I, T_prop, FER
} test_t;

#ifdef DEBUG
#define DEBUG_TEST 1
#else
#define DEBUG_TEST 0
#endif

#define debug_print(fmt, ...)
\
  do {
\
    if (DEBUG_TEST)
\
      fprintf(stderr, fmt, ##__VA_ARGS__);
\
  } while (0)

/**
 * @brief Searches for byte (unsigned char) in array of bytes (unsigned chars)
with name array.
 *
 * @param byte : unsigned char to be found
 * @param array : array of bytes we want to check if has byte or not
 * @return true : if byte exists in array
 * @return false : otherwise
 */
bool findByteOnArray(unsigned char byte, unsigned char *array);

/**
 * @brief Gets the current time
 *
 * @return unsigned long long
 */
unsigned long long getTime();

/**
 * @brief Sets up time measuring variables
 *
```

```
 */
void startTime();

/**
 * @brief Processes a -t CLI argument, checking if it's valid.
 *
 * @param argv
 * @param argNum
 * @return int
 */
int processTestArgument(char **argv, int argNum);
```

Utils.c

```
#define _POSIX_C_SOURCE 199309L

#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "utils.h"

unsigned long long start_ms;
time_t start_s;

bool findByteOnArray(unsigned char byte, unsigned char *array)
{
  for (int i = 0; (array + i) != NULL; i++)
  {
    if (array[i] == byte)
      return true;
  }

  return false;
}

unsigned long long getTime() {
  unsigned long long now_ms;
  time_t now_s;

  struct timespec spec;
  clock_gettime(CLOCK_REALTIME, &spec);
  now_ms = round(spec.tv_nsec / 1.0e4);
  now_s = spec.tv_sec;

  if (now_ms > 99999) {
    now_s++;
    now_ms = 0;
  }
```

```
  return ((now_s - start_s) * 100000) + (now_ms - start_ms);
}

void startTime() {
  struct timespec spec;
  clock_gettime(CLOCK_REALTIME, &spec);
  start_ms = round(spec.tv_nsec / 1.0e4);
  start_s = spec.tv_sec;

  if (start_ms > 99999) {
    start_s++;
    start_ms = 0;
  }
}

int processTestArgument(char **argv, int argNum) {
  if (strcmp(argv[argNum], "C") == 0)
    return C;

  if (strcmp(argv[argNum], "I") == 0)
    return I;

  if (strcmp(argv[argNum], "T_prop") == 0)
    return T_prop;

  if (strcmp(argv[argNum], "FER") == 0)
    return FER;

  return INV;
}
```