Faculdade de Engenharia da Universidade do Porto

1º trabalho laboratorial - Ligação de dados

Relatório

Redes de Computadores

Turma 3 | Grupo 4

António Cadilha da Cunha Bezerra	up201806854
Pedro Jorge Fonseca Seixas	up201806227

Sumário

Este relatório incidirá sobre o primeiro trabalho proposto na unidade curricular de Redes de Computadores cujo principal objetivo era o desenvolvimento de uma aplicação que fosse capaz de transmitir ficheiros entre dois computadores de forma assíncrona através de uma porta de série.

A aplicação é capaz de transferir corretamente e recuperar da ocorrência de erros na transferência de um ficheiro de um computador para o outro.

Introdução

O objetivo do relatório é a examinar a componente teórica do trabalho realizado, cujo objetivo foi a implementação de um protocolo de ligação de dados entre dois computadores usando uma porta de série para ser possível a transferência de informação entre eles.

O relatório está organizado da seguinte forma:

- **Arquitetura** - blocos funcionais e interfaces
- **Estrutura do código** - APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura
- **Casos de uso principais** - identificação; sequências de chamada de funções
- **Protocolo de ligação lógica** - identificação dos principais aspectos funcionais
- **Protocolo de aplicação** - identificação dos principais aspectos funcionais
- **Validação** - descrição dos testes efectuados
- **Eficiência do protocolo de ligação de dados** - caracterização estatística da eficiência do protocolo
- **Conclusões** - síntese da informação apresentada nas secções anteriores; reflexão sobre os objectivos de aprendizagem alcançados

Arquitetura

A aplicação desenvolvida consiste em duas camadas, a ***Data-Link-Layer*** e a ***Application-Layer***, apresentadas de seguida.

Data-Link-Layer

Esta camada encontra-se no ficheiro ***ll.c*** e contém as funções necessárias para iniciar e terminar a ligação entre os dois computadores através da porta de série, bem como ler e escrever para esta usando como apoio as funções do ficheiro ***message.c*** e ***utils.c***.

Application-Layer

Esta camada encontra-se no ficheiro ***application.c***, ***transmitter.c*** e ***receiver.c,*** e serve como ponte entre o utilizador e a Data-Link-Layer, recebendo os argumentos do utilizador e tratando de tudo o que seja relacionado com abertura, leitura e escrita de e para ficheiros.

Estrutura do código

O código do programa está dividido em sete ficheiros, consoante as camadas (Application ou Data-Link), o papel correspondente (Transmissor ou Recetor) e funcionalidades que sejam implementadas. A cada um destes está também associado um *header file*. O programa inclui ainda dois *header files* que definem macros.

Application - application.c

Contém as funções e estruturas de dados mais gerais da **Application Layer**.

Funções principais

- **main:** Recebe os argumentos da linha de comandos e procede à execução do programa de acordo com estes.
- **checkArgs:** Função para *parse* dos argumentos da linha de comandos, deteta erros e coloca os valores adequados numa *struct* **applicationArgs**.

Estruturas de Dados

- **applicationArgs:** Armazena os argumentos da aplicação.

Receiver - receiver.c

Contém as funções relativas ao **Recetor** que fazem parte da **Application Layer**.

Funções principais

- **receiverApplication:** Contém o ciclo de leitura de tramas de informação. Invoca a função **llread** e chama a função **parsePacket** para serem processadas.
- **parsePacket:** Processa as tramas consoante sejam tramas de controlo ou dados.
- **parseControlPacket:** Processa as tramas de controlo.

Transmitter - transmitter.c

Contém as funções relativas ao **Transmissor** que fazem parte da **Application Layer**.

Funções principais

- **transmitterApplication:** Contém o ciclo de envio de tramas de informação. Invoca a

função **llwrite** para envio das tramas de dados e a função **sendControlPacket** para enviar as tramas de controlo.

- **sendControlPacket:** Envia as tramas de controlo.

Link Layer - ll.c

Contém as principais funções e macros da **Link Layer.**

Funções Principais

- **llopen:** Abre a porta de série e troca as tramas SET e UA.
- **llread:** Lê as tramas I recorrendo à função **readMessage** e envia a correspondente trama de supervisão (RR ou REJ).
- **llwrite:** Envia as tramas I recorrendo à função **sendDataMessage**.
- **llclose:** Efetua a troca de tramas DISC e UA, consoante o papel da aplicação utiliza as funções **recv_disc** ou **trans_disc**, e fecha a ligação à porta de série.

Macros

- **BAUDRATE:** Capacidade de ligação.

Messages - message.c

Contém as funções que recebem e enviam informação através da porta de série, usando chamadas ao sistema **write** e **read**.

Funções Principais

- **sendSupervisionMessage:** Envia uma trama de supervisão (SET, UA, DISC, RR, REJ). Para a escrita na porta de série recorre às funções **sendMessageWithResponse** e **sendMessageWithoutResponse**.
- **sendDataMessage:** Envia uma trama de dados (tipo I). Para a escrita na porta de série recorre às funções **sendMessageWithResponse** e **sendMessageWithoutResponse**.
- **sendMessageWithResponse:** Escreve uma trama na porta de série e aguarda a resposta do outro lado. O tipo de resposta esperado é transmitido à máquina de estados para controlar determinar quando o envio foi bem sucedido.
- **sendMessageWithoutResponse:** Escreve uma trama na porta de série sem esperar por uma resposta do outro lado.
- **readMessage:** Lê uma trama da porta de série.
- **alarmHandler:** Signal handler para SIGALARM.

State - state.c

Contém as funções que definem a máquina de estados que permite o mecanismo de *stop and wait*.

Funções Principais

- **getState/getLastResponse/getRole:** Devolvem o estado atual, a última resposta recebida e o papel da aplicação, respetivamente.
- **setStateMachineRole:** Define o papel (Transmissor/Recetor) da aplicação.
- **configStateMachine:** Adequa o funcionamento da máquina de estados ao tipo de resposta pretendida.
- **stateHandlers:** Implementam a máquina de estados para cada flag para cada um dos modos de funcionamento.

Estruturas de Dados

- **stateMachine:** Armazena os campos de interesse para a máquina de estados.

Utilities - utils.c

Contém funções usadas em vários locais da aplicação.

Funções Principais

- **messageStuffing/messageDestuffing:** Realizam o *stuffing* e *destuffing* para proteger mensagens que contenham bytes de dados iguais a *flags*.
- **Progress Bar:** funções que mostram uma barra de progresso para o envio do ficheiro.

Macros - message_defs.h, packet.h

Ficheiros que contém macros úteis no envio de mensagens ou sobre os tipos e campos das tramas, respetivamente.

Casos de uso

A aplicação deve ser compilada executando o ficheiro **run.sh**. Este ficheiro irá criar o executável **app** que permite ao utilizador usar a aplicação. Deve ser executada da seguinte forma:

Receiver: ./app -p <porta> -r <destino>

Transmitter: ./app -p <porta> -t <ficheiro>

<porta> : Número da porta (/dev/ttyS<porta>)

<destino> : Localização onde irá ser guardado o ficheiro recebido

<ficheiro> : Localização do ficheiro a ser enviado

O *receiver* deverá ser iniciado em primeiro lugar. Caso contrário, a aplicação do *transmitter* irá enviar mensagens durante 9 segundos (3 tentativas de 3 segundos de espera pela resposta) e após esse tempo, se a aplicação do *receiver* ainda não tiver iniciado, irá terminar o programa.

Após o início das duas aplicações, o *transmitter* irá chamar a função **transmitterApplication** que irá tratar de abrir e recolher a informação sobre o ficheiro a ser enviado e chamar a função **llwrite** para enviar essa informação para o *receiver*. Por sua vez, o *receiver* irá invocar a função **receiverApplication** que, através do **llread**, obtém a informação e escreve para o novo ficheiro.

Protocolo de ligação lógica

O protocolo de ligação lógica permite:

- A configuração inicial e a terminação da ligação da porta de série
- Envio e a leitura de informação da porta

Utilizando para isso funções auxiliares que ajudam tanto na construção das mensagens para envio como nas rotinas de leitura e escrita para a prevenção de erros.

As seguintes funções foram implementadas:

llopen()

Esta função tem o propósito de iniciar a comunicação entre os dois computadores utilizando a porta de série.

Começa por chamar a função **open**, abrindo a porta de série com as flags de leitura e escrita. É então configurada a porta com o **VTIME** a 1 e o **VMIN** a 0, para que a função read não esteja à espera de um caractere antes de retornar.

Consoante o *role* passado como argumento, irá invocar a função **trans_init** ou **recv_init** logo após configurar o *role* da state-machine para futuras leituras.

A primeira função, do *Transmitter*, irá enviar a mensagem **SET** e esperar por uma mensagem **UA** usando a função auxiliar **sendSupervisionMessage** que recebe como argumentos a porta, o campo *address* e o campo *control* da mensagem, e o tipo de resposta que pretende receber. Esta função irá então construir a mensagem, enviá-la, e fazer o ciclo de leitura para receber o **UA** que só retorna caso tenha lido ou caso tenha ocorrido um erro

após 3 tentativas de envio e leitura.

A segunda função, do *Receiver*, chama a função **readMessage** que trata de ler uma mensagem do tipo enviado por parâmetro, no caso, do tipo **SET**, e, após essa leitura, envia uma mensagem do tipo **UA** usando a função **sendSupervisionMessage**.

llwrite()

Esta função permite o envio de pacotes de dados pela porta de série.

Usa a função **sendDataMessage** que está responsável por construir o pacote, adicionando as *flags*, *address*, *control* e os BCCs para a deteção de erros. Após isso, faz *stuffing* da mensagem e envia-a para a função **sendMessageWithResponse** que trata de enviar o pacote e esperar por uma resposta do tipo **RR** ou **REJ** consoante o pacote tenha sido, ou não, corretamente enviado e recebido pelo *Receiver*. Caso tenha recebido um **REJ**, a função tenta enviar mais duas vezes, retornando erro caso continue a receber **REJ** mesmo após estas tentativas.

llread()

Esta função permite a leitura de pacotes de dados pela porta de série.

Usa a função **readMessage** para ler uma trama de informação. Após isso, faz *destuffing* da mensagem e verifica o **BCC2** para certificar que a mensagem recebida não contém erros. Caso o **BCC2** calculado dos dados seja diferente do **BCC2** recebido, é enviada a mensagem do tipo **REJ**, é descartada a mensagem e é retornado erro. Caso esteja tudo correto, é guardada a mensagem e é enviada uma resposta do tipo **RR**.

llclose()

Esta função tem o propósito de terminar a ligação da porta de série entre os dois computadores.

Esta função, consoante o *role*, irá invocar a função **trans_disc** ou **recv_disc**.

A primeira função, do *Transmitter*, irá enviar a mensagem **DISC** e esperar por uma mensagem **DISC** usando a função auxiliar **sendSupervisionMessage**. Após esta receção, é enviada a mensagem **UA** para que o *Receiver* saiba que a conexão foi terminada.

A segunda função, do *Receiver*, chama a função **readMessage** que trata de ler uma mensagem do tipo **DISC**, envia uma mensagem do tipo **DISC** e espera por uma mensagem do tipo **UA** usando a função **sendSupervisionMessage**.

Depois disto, são repostas as configurações iniciais da porta e é terminada a ligação chamando a função **close**.

Protocolo de aplicação

O protocolo de aplicação é responsável por:

- Abertura e leitura do ficheiro a transmitir
- Escrita para o ficheiro de destino
- Construir e enviar pacotes de controlo e de dados

As seguintes funções foram implementadas:

sendControlPacket()

Esta função permite construir e enviar um pacote de controlo.

Recebe como argumentos o tipo de pacote (**START** ou **STOP**), o tamanho do ficheiro e o nome do ficheiro e envia o pacote de controlo com os campos devidamente preenchidos usando a função **llwrite**.

parseControlPacket()

Esta função faz o processamento de um pacote de controlo, obtendo o nome e o tamanho do ficheiro que está a ser recebido.

parsePacket()

Esta função permite o processamento de um pacote.

Recebe como argumento o pacote e verifica o primeiro byte do pacote, que indica o seu tipo. Caso seja **START**, é chamada a função **parseControlPacket** e é criado o novo ficheiro de destino com o nome do ficheiro recebido. Caso seja **STOP**, é fechado o ficheiro de destino e é verificado se o tamanho do ficheiro é efetivamente igual ao tamanho recebido no pacote de controlo. Caso seja **DATA**, é verificado o número de sequência do pacote e é escrita a informação para o ficheiro de destino. É retornado o tipo de pacote recebido.

transmitterApplication()

Esta função permite encapsular as rotinas que o transmissor deve executar para que seja possível a transferência correta do ficheiro.

Começa por obter o tamanho do ficheiro a enviar e abri-lo através das funções **stat** e **open**. Após isso, executa a função **sendControlPacket** para enviar o pacote de controlo **START**. Enviado o pacote, começa então um ciclo de leitura do ficheiro em porções de tamanho previamente definido, construindo e enviando o pacote com o respectivo header, através do **llwrite**. No fim de enviar todo o ficheiro, é invocada novamente a função **sendControlPacket** para enviar o pacote **STOP** e termina com o fecho do ficheiro.

receiverApplication()

Esta função permite encapsular as rotinas que o recetor deve executar para que seja possível a transferência correta do ficheiro.

A função consiste num ciclo de leitura, que apenas retorna quando é recebido o pacote **STOP** ou em caso de muitos erros de leitura, que podem significar que o transmissor falhou. É chamada a função **llread** para ler os pacotes, sendo estes enviados para a função **parsePacket** para serem processados.

Validação

Foram efetuados diversos testes à nossa aplicação:

- **Envio de ficheiros de diversos tamanhos -** Pinguim.gif (10968 bytes), big.jpg (3309702 bytes)
- **Envio do ficheiro com pacotes de diferentes tamanhos -** 32, 64, 128, 256, 512, 1024, 2048, 4096
- **Envio do ficheiro com valores diferentes de Baudrate -** 9600, 19200, 38400, 57600, 115200
- **Envio do ficheiro com introdução de erros no BCC1 ou no BCC2 -** 0%, 2%, 5%, 10% e 20% de erros
- **Envio do ficheiro com interrupção no cabo**
- **Envio do ficheiro com timeouts para simular o tempo de propagação -** 0s, 0.1s, 0.2s, 0.5s, 1s

Todos estes testes foram bem sucedidos e verificados usando o comando **md5sum**.

Eficiência do protocolo de ligação de dados

Todos os testes efetuados foram realizados nos computadores dos laboratórios, quer presencialmente quer por ligação SSH, para que fossem obtidos resultados viáveis. *Todos os gráficos e tabelas dos testes encontram-se no **Anexo II**.*

Variação da Percentagem de Erros no BCC1 e BCC2

Com o aumento da percentagem de erros no BCC1 é possível observar uma decadência exponencial da eficiência do protocolo devido aos *timeouts* dados quando é encontrado um erro no cabeçalho da mensagem. Em relação aos erros no BCC2, a decadência é linear, visto que apenas é necessária a retransmissão da mensagem para obter o pacote correto.

Variação do Tempo de Propagação

Com o aumento do tempo de propagação a decadência da eficiência do protocolo é também exponencial pois o aumento do tempo faz com que cada mensagem demore mais a ser enviada, tanto como de dados como respostas.

Variação do Tamanho dos Pacotes

O aumento do tamanho do pacote faz aumentar linearmente a eficiência visto que para pacotes maiores serão necessários menos pacotes, e consequentemente, menos *frame headers* e *frame tails*.

Variação do Baudrate

Para os valores testados, não foi observada nenhuma alteração significativa na eficiência do programa.

Conclusões

Este trabalho consistia na implementação de um protocolo de ligação de dados para comunicação através de uma porta de série. Um dos principais objetivos do trabalho era a compreensão de que deveriam ser criadas duas camadas. Cada uma com uma responsabilidade atribuída, e independentes entre si, tal que o modo de funcionamento de uma em nada deve influenciar o modo de funcionamento da outra.

Ao realizar este trabalho foi clara a distinção destas camadas, a Data-Link e a Application, sendo a primeira responsável pela comunicação através da porta de série, tratando da construção das mensagens, stuffing, erros, respostas, etc... enquanto a Application apenas utiliza as funções desta camada, mas sem o conhecimento de nenhum destes mecanismos de tratamento de erros, ou de qualquer outro tipo de funcionamento da camada.

O trabalho foi concluído com sucesso, cumprindo os objetivos preconizados e contribuindo para um aprofundamento do conhecimento sobre os mecanismos de envio de informação e, mais concretamente, da interface com uma porta de série.

No entanto, Devido ao contexto pandémico atual e ao modelo de funcionamento das aulas que por isso foi implementado, foi um tanto difícil assegurar que o trabalho funcionava como esperado. O facto de estar apenas um elemento do grupo na sala com acesso à porta de série física fez com que a deteção de erros que não eram facilmente identificáveis com portas de série virtuais demorasse mais tempo.

Anexo I - Código Fonte

application.c

```
#include "application.h"

int main(int argc, char** argv)
{
    applicationArgs app;
    app.port = -1;
    app.role = -1;
    app.path[0] = '\0';

    if (checkArgs(argc, argv, &app) < 0) {
         fprintf(stderr, "Invalid args. Usage: ./app -p <port> (-r | -t)
<path>");
        return -1;
    }

    printf("App initialized!\nPath: %s\n", app.path);

    int fd;

    if ((fd = llopen(app.port, app.role)) < 0) {
        fprintf(stderr, "llopen failed\n");
        return -1;
    }

    // Start Clock
    struct timeval beginTime, endTime;
    gettimeofday(&beginTime, NULL);

    if (app.role == TRANSMITTER) {
        if (transmitterApplication(fd, app.path) < 0) {
            fprintf(stderr, "Transmitter Application failed\n");
            return -1;
        }
    }
    else {
        if (receiverApplication(fd, app.path) < 0) {
            fprintf(stderr, "Receiver Application failed\n");
            return -1;
        }
    }

    // Get Elapsed Time
    gettimeofday(&endTime, NULL);

    double elapsed = (endTime.tv_sec - beginTime.tv_sec) * 1e6;
    elapsed = (elapsed + (endTime.tv_usec - beginTime.tv_usec)) * 1e-6;

    printf("Elapsed: %.5lf seconds\n", elapsed);
```

```
    if (llclose(fd) < 0){
        fprintf(stderr, "llclose failed\n");
        return -1;
    }

    return 0;
}

int checkArgs(int argc, char ** argv, applicationArgs * app) {
    if (argc != 5) {
        return -1;
    }

    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-t")) {
            if (app->role != -1)
                return -1;
            app->role = TRANSMITTER;
        }
        else if (!strcmp(argv[i], "-r")) {
            if (app->role != -1)
                return -1;
            app->role = RECEIVER;
        }
        else if (!strcmp(argv[i],"-p")){
            if (i + 1 == argc || app->port != -1){
                return -1;
            }
            if (atoi(argv[i+1]) >= 0){
                app->port = atoi(argv[i+1]);
                i++;
            }
            else return -1;
        }
        else if (argv[i][0] != '-' && app->path[0] == '\0')
            strcpy(app->path, argv[i]);
        else {
            return -1;
        }
    }

    if (app->path[0] == '\0') return -1;
    if (app->role != TRANSMITTER && app->role != RECEIVER) return -1;

    return 0;
}
```

application.h

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>
#include <stdio.h>
#include <termios.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

#include "ll.h"
#include "receiver.h"
#include "transmitter.h"

typedef struct {
    int port;
    int role;
    char path[256];
} applicationArgs;

int checkArgs(int argc, char ** argv, applicationArgs * app);
```

ll.c

```
/* Application layer functions */

#include "ll.h"

struct termios oldtio;

int llopen(int port, int role) {
    char portString[15];
    sprintf(portString, "/dev/ttyS%d", port);

    int fd = open(portString, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        perror("Error opening serial port");
        return -1;
    }

    struct termios newtio;

     if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings
*/
        perror("tcgetattr failed");
        return -1;
    }

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

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

    newtio.c_cc[VTIME]    = 1;   /* inter-character timer unused */
      newtio.c_cc[VMIN]       = 0;    /* blocking read until 5 chars
received */

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr failed to set new termios struct");
        return -1;
    }

    printf("New termios structure set\n");

    signal(SIGALRM, alarm_handler);

    switch (role) {
        case RECEIVER:
            setStateMachineRole(RECEIVER);
            if (recv_init(fd) < 0){
```

```
                fprintf(stderr, "Could not start RECEIVER\n");
                return -1;
            }
            break;
        case TRANSMITTER:
            setStateMachineRole(TRANSMITTER);
            if (trans_init(fd) < 0){
                fprintf(stderr, "Could not start TRANSMITTER\n");
                return -1;
            }
            break;
        default:
            return -1;
    }

    return fd;
}

int recv_init(int fd) {
    unsigned char message[5];
    if (readMessage(fd, message, COMMAND_SET) < 0) return -1;
     return sendSupervisionMessage(fd, MSG_A_RECV_RESPONSE, MSG_CTRL_UA,
NO_RESPONSE);
}

int trans_init(int fd) {
             return   sendSupervisionMessage(fd,   MSG_A_TRANS_COMMAND,
MSG_CTRL_SET, RESPONSE_UA);
}

int llwrite(int fd, unsigned char * buffer, int lenght) {
    static int packet = 0;

    int ret;
    int numTries = 0;

    while (numTries < 3) {
        numTries++;
        if ((ret = sendDataMessage(fd, buffer, lenght, packet)) > -1) {
            packet = (packet + 1) % 2;
            return ret;
        }
        fprintf(stderr, "sendDataMessage failed\n");
    }

    return -1;
}

int llread(int fd, unsigned char * buffer) {
    static int packet = 0;
                  unsigned    char    stuffedMessage[MAX_BUFFER_SIZE],
unstuffedMessage[MAX_PACKET_SIZE + 7]; // MAX MESSAGE SIZE
    int numBytesRead;
```

```
     if ((numBytesRead = readMessage(fd, stuffedMessage, COMMAND_DATA))
< 0) {
         fprintf(stderr, "Read operation failed\n");
         return -1;
     }
      int res = messageDestuffing(stuffedMessage, 1, numBytesRead - 1,
unstuffedMessage);

     unsigned char receivedBCC2 = unstuffedMessage[res - 1];
      unsigned char receivedDataBCC2 = BCC2(unstuffedMessage, res - 1,
4);

       if (receivedBCC2 == receivedDataBCC2 && unstuffedMessage[2] ==
MSG_CTRL_S(packet)) {
         packet = (packet + 1) % 2;
                   if (sendSupervisionMessage(fd, MSG_A_RECV_RESPONSE,
MSG_CTRL_RR(packet), NO_RESPONSE) < 0) return -1;
         memcpy(buffer, &unstuffedMessage[4], res-5);
         return res - 5;
     }
     else if (receivedBCC2 == receivedDataBCC2) {
                         sendSupervisionMessage(fd, MSG_A_RECV_RESPONSE,
MSG_CTRL_RR(packet), NO_RESPONSE);
         fprintf(stderr, "Duplicate Packet!\n");
         tcflush(fd, TCIFLUSH);
         return -1;
     } else {
                         sendSupervisionMessage(fd, MSG_A_RECV_RESPONSE,
MSG_CTRL_REJ(packet), NO_RESPONSE);
         fprintf(stderr, "Error in BCC2, sent REJ!\n");
         tcflush(fd, TCIFLUSH);
         return -1;
     }
}

int llclose(int fd) {
     switch (getRole()) {
         case RECEIVER:
             if (recv_disc(fd) < 0){
                 fprintf(stderr, "Could not disconnect RECEIVER\n");
                 return -1;
             }
             break;
         case TRANSMITTER:
             if (trans_disc(fd) < 0){
                  fprintf(stderr, "Could not disconnect TRANSMITTER\n");
                 return -1;
             }
             break;
         default:
             return -1;
     }
```

```
    sleep(1);
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
        perror("tcsetattr failed to set old termios struct");
        exit(-1);
    }

    return close(fd);
}

int recv_disc(int fd) {
    printf("DISCONNECTING RECEIVER...\n");
    unsigned char message[5];
    if (readMessage(fd, message, COMMAND_DISC) < 0) return -1;
             return   sendSupervisionMessage(fd,   MSG_A_RECV_COMMAND,
MSG_CTRL_DISC, RESPONSE_UA);
}

int trans_disc(int fd) {
    printf("DISCONNECTING TRANSMITTER...\n");
     if (sendSupervisionMessage(fd, MSG_A_TRANS_COMMAND, MSG_CTRL_DISC,
COMMAND_DISC) < 0) return -1;
            return   sendSupervisionMessage(fd,   MSG_A_TRANS_RESPONSE,
MSG_CTRL_UA, NO_RESPONSE);
}
```

ll.h

```
/* Application layer functions */

#pragma once

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

#include "utils.h"
#include "state.h"
#include "message.h"

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

int llopen(int port, int role);
int recv_init(int fd);
int trans_init(int fd);

int llclose(int fd);
int recv_disc(int fd);
int trans_disc(int fd);

int llwrite(int fd, unsigned char * buffer, int lenght);
int llread(int fd, unsigned char * buffer);
```

message.c

```
#include "message.h"

int alarm_flag = FALSE;

void alarm_handler() {
    alarm_flag = TRUE;
}

int sendSupervisionMessage(int fd, unsigned char address, unsigned char
control, mode responseType) {
    unsigned char msg[5] = {
        MSG_FLAG,
        address,
        control,
        BCC(address, control),
        MSG_FLAG
    };

    if (responseType != NO_RESPONSE) {
                if  (sendMessageWithResponse(fd,  msg,  MSG_SET_SIZE,
responseType) < 0)
            return -1;

        return 0;
    }
    else {
        if (sendMessageWithoutResponse(fd, msg, MSG_SET_SIZE) < 0)
            return -1;

        return 0;
    }
}

int  sendDataMessage(int  fd,  unsigned  char  *  data,  int  dataSize,  int
packet) {
    int msgSize = dataSize + 5;

    unsigned char msg[msgSize];

    msg[0] = MSG_FLAG;
    msg[1] = MSG_A_TRANS_COMMAND;
    msg[2] = MSG_CTRL_S(packet);
    msg[3] = BCC(MSG_A_TRANS_COMMAND, MSG_CTRL_S(packet));
    unsigned char bcc2 = data[0];
    for (int i = 0; i < dataSize; i++) {
        msg[i + 4] = data[i];
        if (i > 0) bcc2 ^= data[i];
    }
    msg[dataSize + 4] = bcc2;

    unsigned char stuffedData[msgSize * 2];
```

```
    msgSize = messageStuffing(msg, 1, msgSize, stuffedData);
    stuffedData[msgSize] = MSG_FLAG;
    msgSize++;

    int numTries = 0;
    int receivedACK = FALSE;
    int ret;

    do {
        numTries++;
             ret = sendMessageWithResponse(fd, stuffedData, msgSize,
RESPONSE_RR_REJ);

        response_type response = getLastResponse();

         if (ret > 0 && ((packet == 0 && response == R_RR1) || (packet
== 1 && response == R_RR0))) {
            receivedACK = TRUE;
        } else if (ret > 0) {
                fprintf(stderr, "Received response is invalid. Trying
again...\n");
        }
    } while (numTries < 3 && !receivedACK);

    if (!receivedACK) {
        fprintf(stderr, "Failed to get ACK\n");
        return -1;
    }
    else
        return ret;
}

int writeMessage(int fd, unsigned char * msg, int messageSize) {
    static int wrongBcc1 = 0;
    static int wrongBcc2 = 0;

    int ret;

    if (messageWithError(BCC1_ERROR_PERCENTAGE)) {
        wrongBcc1++;
        fprintf(stderr, "Number of BCC1 errors: %d\n", wrongBcc1);
        unsigned char msgWithError[MAX_BUFFER_SIZE];
        memcpy(msgWithError, msg, messageSize);
        msgWithError[2] ^= 0xFF;
        if ((ret = write(fd, msgWithError, messageSize)) == -1) {
            fprintf(stderr, "Write failed\n");
        }
    }
    else if (messageWithError(BCC2_ERROR_PERCENTAGE)) {
        wrongBcc2++;
        fprintf(stderr, "Number of BCC2 errors: %d\n", wrongBcc2);
        unsigned char msgWithError[MAX_BUFFER_SIZE];
        memcpy(msgWithError, msg, messageSize);
```

```
        msgWithError[messageSize - 2] ^= 0xFF;
        if ((ret = write(fd, msgWithError, messageSize)) == -1) {
            fprintf(stderr, "Write failed\n");
        }
    }
    else {
        if ((ret = write(fd, msg, messageSize)) == -1) {
            fprintf(stderr, "Write failed\n");
        }
    }

    return ret;
}

int sendMessageWithResponse(int fd, unsigned char * msg, int
messageSize, mode responseType) {
    configStateMachine(responseType);

    int numTries = 0;
    int ret;

    do {
        numTries++;
        alarm_flag = FALSE;

        ret = writeMessage(fd, msg, messageSize);

        alarm(3);

        int res;
        unsigned char buf[MAX_BUFFER_SIZE];
        while (getState() != STOP && !alarm_flag) {
            res = read(fd, buf, 1);
            if (res == 0) continue;
            updateState(buf[0]);
        }

    } while (numTries < 3 && getState() != STOP);

    alarm(0);

    if (getState() != STOP) {
        fprintf(stderr, "Failed to get response!\n");
        return -1;
    }

    usleep(T_PROP);

    return ret;
}

int sendMessageWithoutResponse(int fd, unsigned char * msg, int
messageSize) {
```

```
    if (write(fd, msg, messageSize) == -1) {
        fprintf(stderr, "Write failed\n");
    }
    return 0;
}

int readMessage(int fd, unsigned char * message, mode responseType) {
    configStateMachine(responseType);
    int res, numBytesRead = 0;
    unsigned char buf[MAX_BUFFER_SIZE];
    alarm_flag = FALSE;

    alarm(7);

       while (getState() != STOP && !alarm_flag && numBytesRead <
MAX_BUFFER_SIZE) {
        res = read(fd, buf, 1);
        if (res == 0) continue;
        alarm(0);
        message[numBytesRead++] = buf[0];
        updateState(buf[0]);
        alarm(7);
    }

    alarm(0);

    if (alarm_flag) {
        fprintf(stderr, "Alarm fired. readMessage took too long\n");
        return -1;
    }

    if (getState() != STOP) {
        fprintf(stderr, "Failed to read message\n");
        return -1;
    }

    usleep(T_PROP);

    return numBytesRead;
}
```

message.h

```
#pragma once

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

#include "utils.h"
#include "state.h"

#define FALSE 0
#define TRUE 1

#define NO_RESPONSE -1

void alarm_handler();

int sendSupervisionMessage(int fd, unsigned char address, unsigned char
control, mode responseType);

int sendDataMessage(int fd, unsigned char * data, int dataSize, int
packet);

int sendMessageWithResponse(int fd, unsigned char * msg, int
messageSize, mode responseType);

int sendMessageWithoutResponse(int fd, unsigned char * msg, int
messageSize);

int readMessage(int fd, unsigned char * message, mode responseType);
```

message_defs.h

```
#pragma once

#define MAX_PACKET_SIZE 256
#define MAX_BUFFER_SIZE (MAX_PACKET_SIZE * 2 + 7)

#define BCC1_ERROR_PERCENTAGE 0
#define BCC2_ERROR_PERCENTAGE 0
#define T_PROP 0

#define MSG_FLAG 0x7e
#define ESCAPE 0x7d

#define MSG_A_TRANS_COMMAND 0x03
#define MSG_A_RECV_RESPONSE 0x03
#define MSG_A_TRANS_RESPONSE 0x01
#define MSG_A_RECV_COMMAND 0x01

#define MSG_CTRL_SET 0x03
#define MSG_CTRL_UA 0x07
#define MSG_CTRL_RR(r) ((r == 0) ? 0x05 : 0x85)
#define MSG_CTRL_REJ(r) ((r == 0) ? 0x01 : 0x81)
#define MSG_CTRL_DISC 0x0b
#define MSG_CTRL_S(r) ((r == 0) ? 0x00 : 0x40)

#define BCC(addr, ctrl) (addr^ctrl)

#define MSG_SET_SIZE 5

#define COM0 0
#define COM1 1
#define COM10 10
#define COM11 11

#define TRANSMITTER 0
#define RECEIVER 1
```

packet.h

```
#pragma once

#include "message_defs.h"

#define DATA_PACKET 1
#define START_PACKET 2
#define END_PACKET 3

#define FILE_SIZE 0
#define FILE_NAME 1
```

receiver.c

```
#include "receiver.h"

int receiverApplication(int fd, char* path) {
    int res;
    int nump = 0;
    int numTries = 0;

    while (1) {
        unsigned char buf[MAX_PACKET_SIZE];
        if ((res = llread(fd, buf)) < 0) {
            if (numTries > 9) return -1;
            numTries++;
            continue;
        }

        if (numTries > 0)
            fprintf(stderr, "It took %d tries to read!\n", numTries);

        numTries = 0;
        nump++;

        int ret;
        if ((ret = parsePacket(buf, res, path)) == END_PACKET)
            break;
        else if (ret == -1)
            return -1;
    }

    printf("Received %d packets\n", nump);
    return 0;
}

int parsePacket(unsigned char * buffer, int lenght, char* path) {
    static int destinationFile;
    static int filesize = 0;

    if (buffer[0] == START_PACKET) {
        parseControlPacket(buffer, lenght, path, &filesize);

         if ((destinationFile = open(path, O_WRONLY | O_CREAT, 0777)) <
0) {
            perror("Error opening destination file!");
            return -1;
        }

        return 0;
    } else if (buffer[0] == END_PACKET) {
        if (close(destinationFile) < 0) {
            perror("Error closing destination file!");
            return -1;
        }
```

```
        if (checkFileSize(path, filesize) != 0) {
                    fprintf(stderr, "Recieved file size different from
expected\n");
            return -1;
        }

        return END_PACKET;
    } else if (buffer[0] == DATA_PACKET) {
        if (checkSequenceNumber(buffer[1]) != 0) {
            fprintf(stderr, "Error, invalid sequence number\n");
            return -1;
        }
        unsigned dataSize = buffer[3] + 256 * buffer[2];
        if (write(destinationFile, &buffer[4], dataSize) < 0) {
            perror("Error writing to destination file!");
            return -1;
        }
        return 0;
    } else {
        printf("Failed on: '");
        for (int i = 0; i < lenght; i++) {
            printf("0x%02x ", buffer[i]);
        }
        printf("'\n");
        return -1;
    }
}

void parseControlPacket(unsigned char * buffer, int lenght, char* path,
int* filesize) {
    for (int i = 1; i < lenght; i++) {
        if (buffer[i] == FILE_SIZE) {
               i++; // i is now in the byte with information about the
number of bytes
            for (int j = 0; j < buffer[i]; j++) {
                *filesize |= (buffer[i+j+1] << (8*j));
            }
            i += buffer[i];
        }

        if (buffer[i] == FILE_NAME) {
               i++; // i is now in the byte with information about the
number of bytes
            char fileName[buffer[i] + 1];
            for (int j = 0; j < buffer[i]; j++) {
                fileName[j] = buffer[i + j + 1];
            }
            fileName[buffer[i]] = '\0';
            strcat(path, fileName);
            i += buffer[i];
        }
    }
```

```
}

int checkFileSize(char* path, int filesize) {
    struct stat file_stat;

    // Reads file info using stat
    if (stat(path, &file_stat)<0){
        perror("Error getting file information.");
        return -1;
    }

    return (filesize != file_stat.st_size);
}

int checkSequenceNumber(unsigned r_sn) {
    static unsigned sequence_num = 0;

    if (r_sn == sequence_num) {
        sequence_num = (sequence_num + 1) % 255;
        return 0;
    }

    return -1;
}
```

receiver.h

```
#pragma once

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#include "ll.h"
#include "packet.h"

int receiverApplication(int fd, char* path);

int parsePacket(unsigned char * buffer, int lenght, char* path);

void parseControlPacket(unsigned char * buffer, int lenght, char* path,
int* filesize);

int checkFileSize(char* path, int filesize);

int checkSequenceNumber(unsigned r_sn);
```

state.c

```
#include "state.h"
stateMachine state;

msg_state getState() {
    return state.currentState;
}

response_type getLastResponse() {
    return state.last_response;
}

int getRole() {
    return state.role;
}

void setStateMachineRole(int role) {
    state.role = role;
}

void configStateMachine(mode stateMachineMode) {
    resetState();
    state.currentMode = stateMachineMode;
}

void resetState() {
    state.currentState = START;
    state.last_response = R_NULL;
}

void updateState(unsigned char byte) {
    switch (state.currentState) {
        case START:
            if (byte == MSG_FLAG)
                state.currentState = FLAG_RCV;
            break;
        case FLAG_RCV:
            FlagRCV_stateHandler(byte);
            break;
        case A_RCV:
            ARCV_stateHandler(byte);
            break;
        case C_RCV:
            CRCV_stateHandler(byte);
            break;
        case WAITING_DATA:
            WaitingData_stateHandler(byte);
            break;
        case BCC_OK:
            if (byte == MSG_FLAG)
                state.currentState = STOP;
            else
```

```
                state.currentState = START;
            break;
        case STOP:
            break;
    }
}

void FlagRCV_stateHandler(unsigned char byte) {
    if (byte == MSG_FLAG)
        return;

    switch (state.currentMode) {
        case RESPONSE_UA:
        case RESPONSE_RR_REJ:
                        if ((state.role == TRANSMITTER && byte ==
MSG_A_RECV_RESPONSE) || (state.role == RECEIVER && byte ==
MSG_A_TRANS_RESPONSE)) {
                state.currentState = A_RCV;
                state.address = byte;
                return;
            }
            break;
        case COMMAND_SET:
        case COMMAND_DISC:
        case COMMAND_DATA:
             if ((state.role == RECEIVER && byte == MSG_A_TRANS_COMMAND)
|| (state.role == TRANSMITTER && byte == MSG_A_RECV_COMMAND)) {
                state.currentState = A_RCV;
                state.address = byte;
                return;
            }
            break;
    }

    state.currentState = START;
}

void ARCV_stateHandler(unsigned char byte) {
    if (byte == MSG_FLAG) {
        state.currentState = FLAG_RCV;
        return;
    }

    switch (state.currentMode) {
        case RESPONSE_UA:
            if (byte == MSG_CTRL_UA) {
                state.currentState = C_RCV;
                state.control = byte;
                return;
            }
            break;
        case RESPONSE_RR_REJ:
               if (byte == MSG_CTRL_RR(0) || byte == MSG_CTRL_RR(1) ||
```

```
byte == MSG_CTRL_REJ(0) || byte == MSG_CTRL_REJ(1)) {
                state.currentState = C_RCV;
                state.control = byte;
                switch (byte) {
                    case MSG_CTRL_RR(0):
                        state.last_response = R_RR0;
                        break;
                    case MSG_CTRL_RR(1):
                        state.last_response = R_RR1;
                        break;
                    case MSG_CTRL_REJ(0):
                        state.last_response = R_REJ0;
                        break;
                    case MSG_CTRL_REJ(1):
                        state.last_response = R_REJ1;
                        break;
                }
                return;
            }
            break;
        case COMMAND_SET:
            if (byte == MSG_CTRL_SET) {
                state.currentState = C_RCV;
                state.control = byte;
                return;
            }
            break;
        case COMMAND_DISC:
            if (byte == MSG_CTRL_DISC) {
                state.currentState = C_RCV;
                state.control = byte;
                return;
            }
            break;
        case COMMAND_DATA:
            if (byte == MSG_CTRL_S(0) || byte == MSG_CTRL_S(1)) {
                state.currentState = C_RCV;
                state.control = byte;
                return;
            }
            break;
    }

    state.currentState = START;
}

void CRCV_stateHandler(unsigned char byte) {
    if (byte == MSG_FLAG) {
        state.currentState = FLAG_RCV;
        return;
    }

    if (byte == BCC(state.address, state.control))
```

```
        if (state.currentMode == COMMAND_DATA)
            state.currentState = WAITING_DATA;
        else
            state.currentState = BCC_OK;
    else
        state.currentState = START;
}

void WaitingData_stateHandler(unsigned char byte) {
    if (byte == MSG_FLAG) {
        state.currentState = STOP;
        return;
    }
    else return;
}
```

state.h

```
#pragma once

#include <stdio.h>

#include "message_defs.h"

typedef enum {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, WAITING_DATA,
STOP} msg_state;
typedef enum {RESPONSE_UA, RESPONSE_RR_REJ, COMMAND_SET, COMMAND_DISC,
COMMAND_DATA} mode;
typedef enum {R_RR0, R_RR1, R_REJ0, R_REJ1, R_NULL} response_type;

typedef struct {
    msg_state currentState;
    mode currentMode;
    int role;
    unsigned char control;
    unsigned char address;
    response_type last_response;
} stateMachine;

msg_state getState();

response_type getLastResponse();

int getRole();

void setStateMachineRole(int role);

void configStateMachine(mode stateMachineMode);

void resetState();

void updateState(unsigned char byte);

void FlagRCV_stateHandler(unsigned char byte);

void ARCV_stateHandler(unsigned char byte);

void CRCV_stateHandler(unsigned char byte);

void WaitingData_stateHandler(unsigned char byte);
```

transmitter.c

```
#include "transmitter.h"

int transmitterApplication(int fd, char* path) {
    int file_fd;
    struct stat file_stat;

    // Reads file info using stat
    if (stat(path, &file_stat)<0){
        perror("Error getting file information.");
        return -1;
    }

    // Opens file to transmit
    if ((file_fd = open(path, O_RDONLY)) < 0){
        perror("Error opening file.");
        return -1;
    }

    // Get the file name
    char * filename = getFilename(path);

    // Sends START packet
          if  (sendControlPacket(fd,  START_PACKET,  file_stat.st_size,
filename) < 0) {
        fprintf(stderr, "Error sending START packet.\n");
        return -1;
    }

    // Sends DATA packets
    unsigned char buf[MAX_PACKET_SIZE];
    unsigned bytes_to_send;
    unsigned sequenceNumber = 0;
    unsigned progress = 0;

     while ((bytes_to_send = read(file_fd, buf, MAX_PACKET_SIZE - 4)) >
0) {
        unsigned char dataPacket[MAX_PACKET_SIZE];
        dataPacket[0] = DATA_PACKET;
        dataPacket[1] = sequenceNumber % 255;
        dataPacket[2] = (bytes_to_send / 256);
        dataPacket[3] = (bytes_to_send % 256);
        memcpy(&dataPacket[4], buf, bytes_to_send);

        progress += bytes_to_send;
        printProgressBar(progress, file_stat.st_size);
               if  (llwrite(fd,  dataPacket,  ((bytes_to_send  +  4)  <
MAX_PACKET_SIZE)? (bytes_to_send + 4) : MAX_PACKET_SIZE) < 0) { // Only
sends max packet if the last packet is of that size
            fprintf(stderr, "llwrite failed\n");
            return -1;
        }
```

```
        // printf("Sent %d data bytes\n", bytes_to_send+4);

        sequenceNumber++;
        clearProgressBar();
    }

    printProgressBar(1, 1);
    printf("Data packets sent: %d\n",sequenceNumber);

    // Sends END packet
     if (sendControlPacket(fd, END_PACKET, file_stat.st_size, filename)
< 0) {
        fprintf(stderr, "Error sending END packet.\n");
        return -1;
    }

    return close(file_fd);
}



int sendControlPacket(int fd, unsigned char ctrl_field, unsigned
file_size, char* file_name) {
    unsigned L1 = sizeof(file_size);
    unsigned L2 = strlen(file_name);
    unsigned packet_size = 5 + L1 + L2;

    unsigned char packet[packet_size];
    packet[0] = ctrl_field;
    packet[1] = FILE_SIZE;
    packet[2] = L1;
    memcpy(&packet[3], &file_size, L1);
    packet[3+L1] = FILE_NAME;
    packet[4+L1] = L2;
    memcpy(&packet[5+L1], file_name, L2);

    return llwrite(fd, packet, packet_size);
}
```

transmitter.h

```
#pragma once

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#include "ll.h"
#include "packet.h"
#include "utils.h"

int transmitterApplication(int fd, char* path);

int sendControlPacket(int fd, unsigned char ctrl_field, unsigned
file_size, char* file_name);
```

utils.c

```
#include "utils.h"

int messageStuffing(unsigned char * buffer, int startingByte, int lenght, unsigned char * stuffedMessage) {
    int messageSize = 0;

    for (int i = 0; i < startingByte; i++)
        stuffedMessage[messageSize++] = buffer[i];

    for (int i = startingByte; i < lenght; i++) {
        if (buffer[i] == MSG_FLAG || buffer[i] == ESCAPE) {
            stuffedMessage[messageSize++] = 0x7d;
            stuffedMessage[messageSize++] = buffer[i] ^ 0x20;
        }
        else {
            stuffedMessage[messageSize++] = buffer[i];
        }
    }

    return messageSize;
}

int messageDestuffing(unsigned char * buffer, int startingByte, int lenght, unsigned char * destuffedMessage) {
    int messageSize = 0;

    for (int i = 0; i < startingByte; i++) {
        destuffedMessage[messageSize++] = buffer[i];
    }

    for (int i = startingByte; i < lenght; i++) {
        if (buffer[i] == ESCAPE) {
            destuffedMessage[messageSize++] = buffer[i + 1] ^ 0x20;
            i++;
        }
        else {
            destuffedMessage[messageSize++] = buffer[i];
        }
    }

    return messageSize;
}

unsigned char BCC2(unsigned char * data, int dataSize, int startingByte) {
    unsigned char bcc = data[startingByte];

    for(int i = startingByte + 1; i < dataSize; i++)
        bcc ^= data[i];

    return bcc;
```

```
}

void clearProgressBar() {
    int i;
    for (i = 0; i < NUM_BACKSPACES; ++i) {
        fprintf(stdout, "\b");
    }
    fflush(stdout);
}

void printProgressBar(int progress, int total) {
    int i, percentage = (int)((((double)progress) / total) * 100);
        int  num_separators  =  (int)((((double)progress)  /  total)  *
PROGRESS_BAR_SIZE);;
    fprintf(stdout, "[");
    for (i = 0; i < num_separators; ++i) {
        fprintf(stdout, "%c", SEPARATOR_CHAR);
    }
    for (; i < PROGRESS_BAR_SIZE; ++i) {
        fprintf(stdout, "%c", EMPTY_CHAR);
    }
    fprintf(stdout, "]  %2d%%  ", percentage);
    fflush(stdout);
}

char * getFilename(char * path) {
    char * filename = path, *p;
    for (p = path; *p; p++) {
        if (*p == '/' || *p == '\\' || *p == ':') {
            filename = p;
        }
    }
    return filename;
}

int messageWithError(int percentage) {
    int error = rand() % 100;

    if (error < percentage)
        return 1;

    return 0;
}
```

utils.h

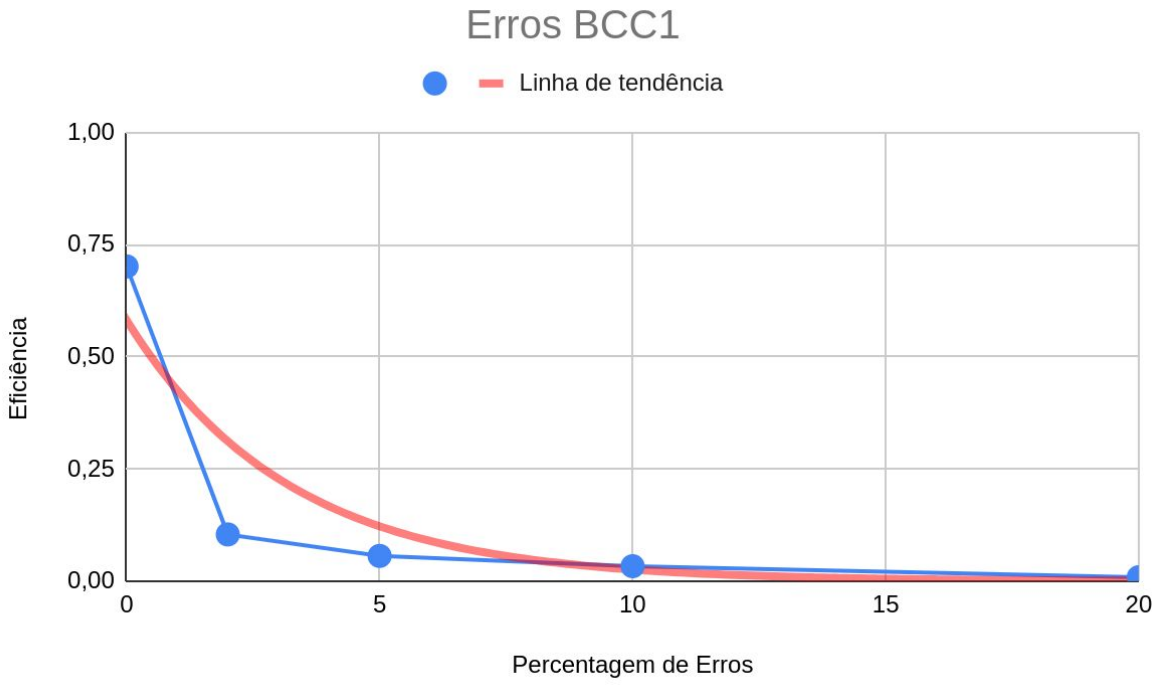
```
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "message_defs.h"

#define PROGRESS_BAR_SIZE   30
#define SEPARATOR_CHAR      '#'
#define EMPTY_CHAR          '.'
#define NUM_BACKSPACES      PROGRESS_BAR_SIZE + 9

int messageStuffing(unsigned char * buffer, int startingByte, int lenght, unsigned char * stuffedMessage);

int messageDestuffing(unsigned char * buffer, int startingByte, int lenght, unsigned char * destuffedMessage);

unsigned char BCC2(unsigned char * data, int dataSize, int startingByte);

void clearProgressBar();

void printProgressBar(int progress, int total);

char * getFilename(char * path);

int messageWithError(int percentage);
```

Anexo II - Testes

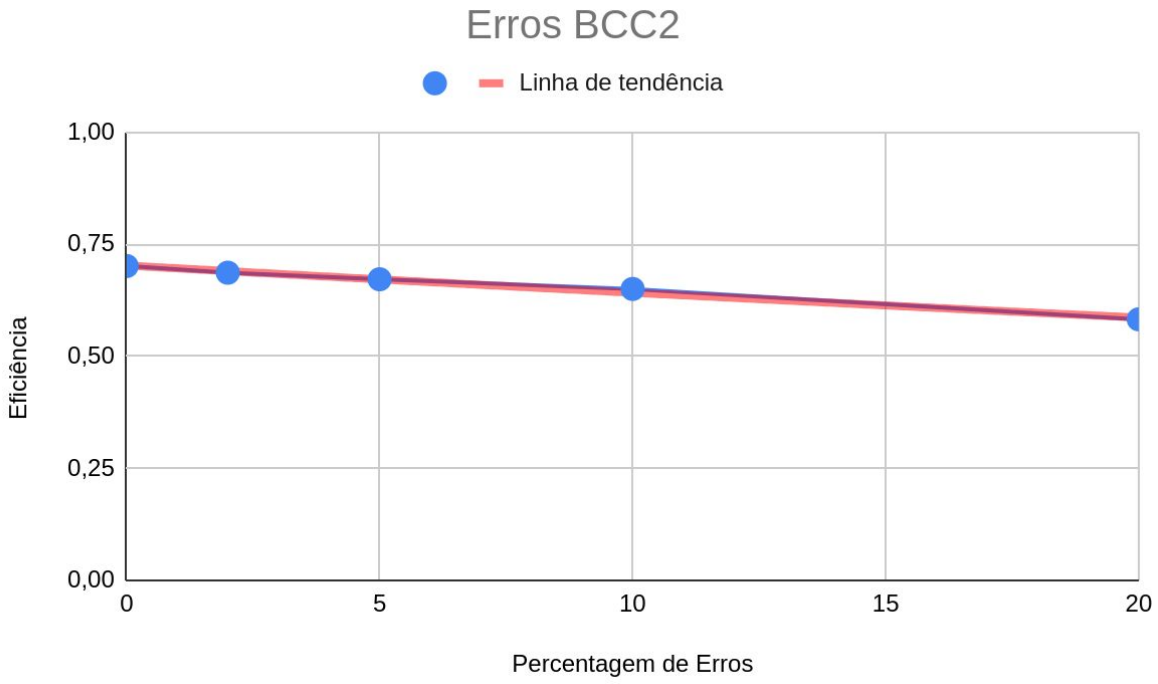
Ficheiro de Teste: Pinguim.gif (10968 Bytes)

Variação da Percentagem de Erros no BCC1 e BCC2

Packet	Baudrate	BCC1 Error Percentage				
128	57600	0	2	5	10	20
Tempo (s)		2,17189	14,61909	27,06263	45,63454	173,1011
Velocidade (b/s)		40399,83609	6002,015173	3242,256943	1922,754124	506,8945258
Eficiência		0,7013860432	0,1042016523	0,05628918303	0,03338114799	0,008800252184

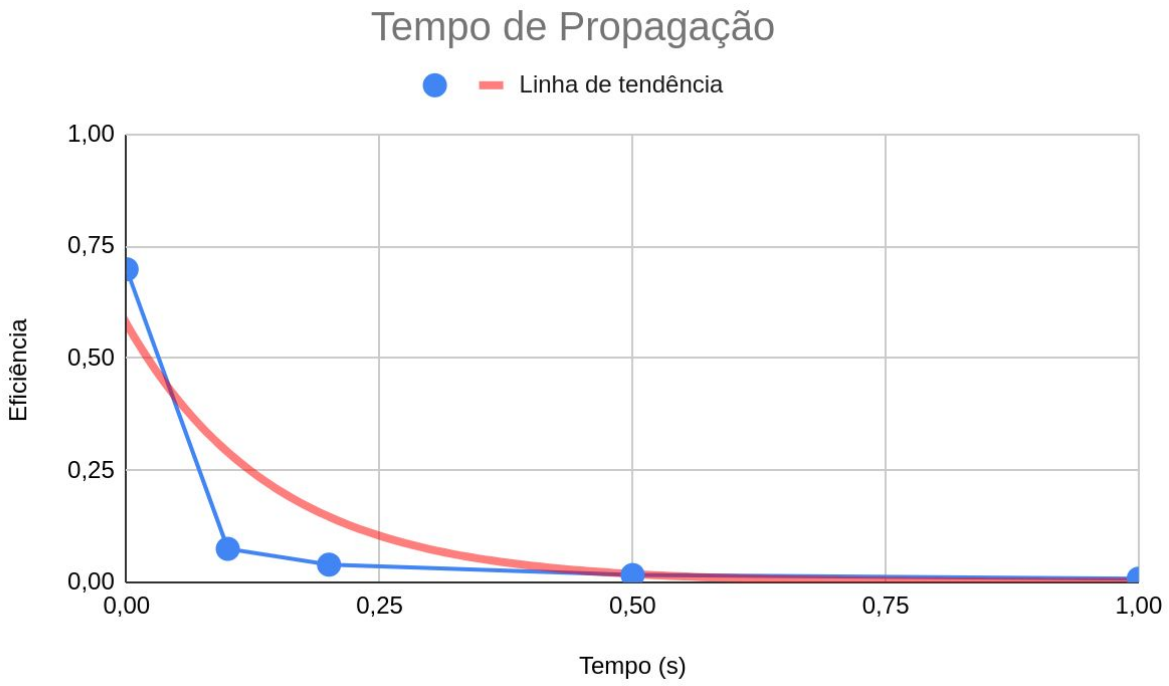


Packet	Baudrate	BCC2 Error Percentage				
128	57600	0	2	5	10	20
Tempo (s)		2,17189	2,21898	2,26742	2,34274	2,61546
Velocidade (b/s)		40399,83609	39542,4925	38697,72693	37453,58	33548,20949
Eficiência		0,7013860432	0,6865016058	0,671835537	0,6502357638	0,5824341926



Variação do Tempo de Propagação

Packet	Baudrate	Tempo de Propagação (s)				
128	57600	0	0,1	0,2	0,5	1
Tempo (s)		2,18067	20,38212	38,58575	93,18433	184,18425
Velocidade (b/s)		40237,17481	4304,949632	2274,000117	941,6175445	476,3925254
Eficiência		0,6985620627	0,07473870889	0,03947916869	0,01634752681	0,008270703566

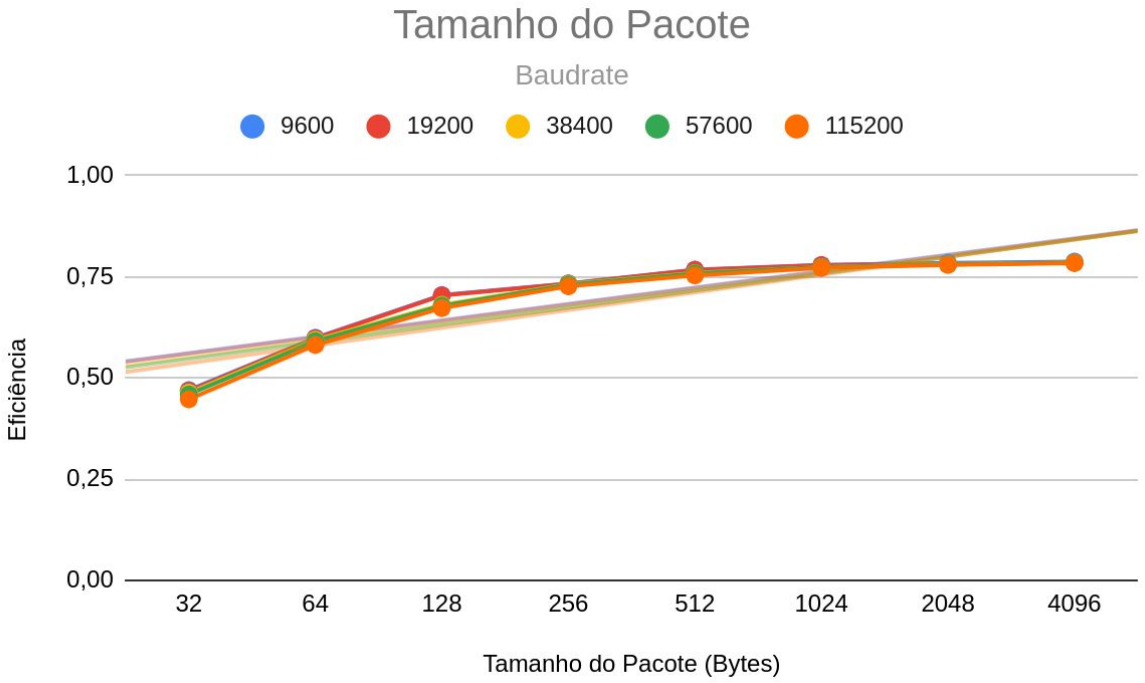


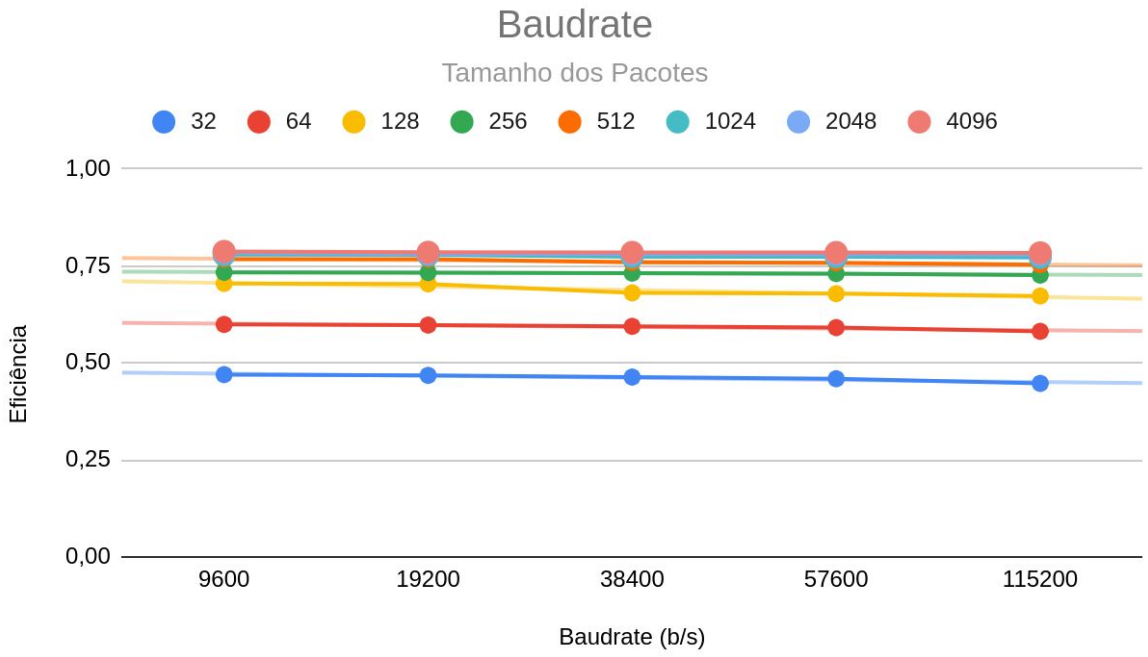
Variação do Tamanho dos Pacotes e Baudrate

Tempo (s)		Packet Size							
		32	64	128	256	512	1024	2048	4096
BAUDRATE	9600	19,45744	15,25716	12,97208	12,46413	11,91382	11,73907	11,65984	11,61183
	19200	9,77116	7,64971	6,49604	6,23749	5,95979	5,87116	5,8524	5,82185
	38400	4,93106	3,84629	3,35894	3,12491	3,01082	2,95373	2,92755	2,91229
	57600	3,31785	2,57824	2,24602	2,08668	2,00953	1,97039	1,95286	1,94232
	115200	1,70357	1,3104	1,13306	1,04891	1,01163	0,98713	0,97772	0,9721

Velocidade (b/s)		Packet Size							
		32	64	128	256	512	1024	2048	4096
BAUDRATE	9600	4509,534656	5751,004774	6764,065593	7039,721184	7364,892201	7474,527369	7525,317672	7556,431674
	19200	8979,895939	11470,23874	13507,30599	14067,1969	14722,6664	14944,91719	14992,82346	15071,49789
	38400	17794,14568	22812,63243	26122,52675	28078,88867	29142,89131	29706,16813	29971,81944	30128,86766
	57600	26446,04186	34032,51831	39066,43752	42049,57157	43663,94132	44531,28568	44931,02424	45174,84246
	115200	51505,95514	66959,70696	77439,85314	83652,5536	86735,26882	88887,98841	89743,48484	90262,31869

Eficiência		Packet Size							
		32	64	128	256	512	1024	2048	4096
BAUDRATE	9600	0,4697431933	0,5990629973	0,704590166	0,73330429	0,7671762709	0,7785966009	0,7838872575	0,7871282993
	19200	0,4677029135	0,5974082678	0,7035055203	0,7326665053	0,7668055418	0,7783811036	0,7808762217	0,7849738485
	38400	0,4633892104	0,5940789696	0,6802741341	0,7312210592	0,7589294611	0,7735981285	0,7805161312	0,7846059287
	57600	0,4591326713	0,5908423317	0,6782367625	0,7300272842	0,7580545368	0,7731125987	0,7800525042	0,7842854593
	115200	0,447100305	0,5812474562	0,6722209474	0,7261506389	0,7529103196	0,7715971216	0,7790233059	0,783527072


Tamanho do Pacote
Baudrate
9600
19200
38400
57600
115200
1,00
0,75
0,50
0,25
0,00
Eficiência
32
64
128
256
512
1024
2048
4096
Tamanho do Pacote (Bytes)


Baudrate
Tamanho dos Pacotes
32
64
128
256
512
1024
2048
4096
1,00
0,75
0,50
0,25
0,00
Eficiência
9600
19200
38400
57600
115200
Baudrate (b/s)