U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

Protocolo de Ligação de Dados

Mestrado Integrado em Engenharia Informática e

André Gomes up201806224
Filipe Recharte up201806743

Índice

Sumário

Este projeto foi realizado no âmbito da unidade curricular de Redes de Computadores e consistiu na elaboração de um protocolo de ligação de dados via porta série, assim como a camada da aplicação integral a esta transmissão.

Concluímos este trabalho com sucesso, visto que os objetivos ambicionados foram cumpridos. Garantimos um serviço de comunicação fiável entre dois sistemas ligados através de um cabo série. Um sistema destes envolve várias camadas e por isso foi fundamental garantir que exista uma independência entre estas e um método bem definido de comunicação entre elas.

Introdução

Este trabalho teve como objetivo a implementação de um protocolo de ligação de dados. Serve este relatório para explicar o processo e raciocínio associados à sua implementação. Nesse sentido, o mesmo foi segmentado em 8 partes diferentes.

Em primeiro lugar, apresentaremos a arquitetura geral do projeto e a sua organização. De seguida, no ponto 3, iremos apresentar resumidamente a estrutura do código nas diferentes camadas, as funções que as constituem e a sua relação com a arquitetura. No ponto 4, falaremos sobre os casos de uso do nosso programa, assim como a sequência de chamada das funções. Nos pontos 5 e 6 serão descritos com mais detalhe os protocolos de ligação lógica e aplicação e as suas funções e no ponto 7 descreveremos os testes realizados e os resultados obtidos. Por fim, faremos um resumo de tudo o que apresentamos neste relatório e uma reflexão sobre os objetivos alcançados.

A documentação Doxygen deste projeto pode ser consultada a qualquer momento a partir do seguinte link: https://ca-moes.github.io/RCOM/html/

Arquitetura

O nosso projeto divide-se, como pretendido, em 2 blocos independentes: A *Application* e o *Data Connection Protocol (DCP).*

A Aplicação faz uso das funções principais do DCP e não tem acesso a mais nada.

O DCP usa as funções do módulo *Data Connection Protocol Specification (DCP_spec)* juntamente com as funções da Máquina de Estados (*statemachine*) para formar as funções principais usadas pela Aplicação. Tanto a Aplicação como o DCP usam o módulo Registos (*logs*) para imprimir mensagens apelativas para a consola.

Estrutura do código

No módulo Aplicação está disponibilizada a *struct* *applicationLayer* que guarda informação relativa ao ficheiro a enviar, a localização de destino do ficheiro, o tipo de aplicação e a localização da ligação ao cabo. O módulo contém duas funções principais: *receiverApp* e *transmitterApp* que serão executadas dependendo do tipo de Aplicação que

desejamos. É neste módulo que se encontra a função *main*, fazendo deste o ponto inicial do programa.

O módulo DCP foi feito para apenas conter as 4 funções principais usadas pela Aplicação: *llopen, llwrite, llread* e *llclose*. Dentro destas encontram-se funções de alto nível especificadas noutros módulos que pelo nome e estrutura no código das funções principais dão uma boa perceção sobre o que é suposto fazerem, permitindo uma leitura e entendimento do funcionamento das 4 funções sem precisar de ver mais código. No DCP é usada a *struct linkLayer* que guarda informação sobre a velocidade de transmissão, o *timeout* do alarme, o número de retransmissões e mais.

O módulo *DCP_spec* contém as funções usadas pelo *DCP* para formar as 4 funções principais. Deste módulo é importante realçar 2 funções muito utilizadas: *readingCycle* e *writeCycle* que são usadas pelo resto das funções do módulo para efetuar a escrita e leitura pelo cabo RS-232.

O programa contempla mais 3 módulos: *State Machine*, *Macros* e *Logs*. Sendo os últimos 2 usados para organização de código e o primeiro para guardar todos os métodos relacionados com a nossa máquina de estados.

Casos de uso principais

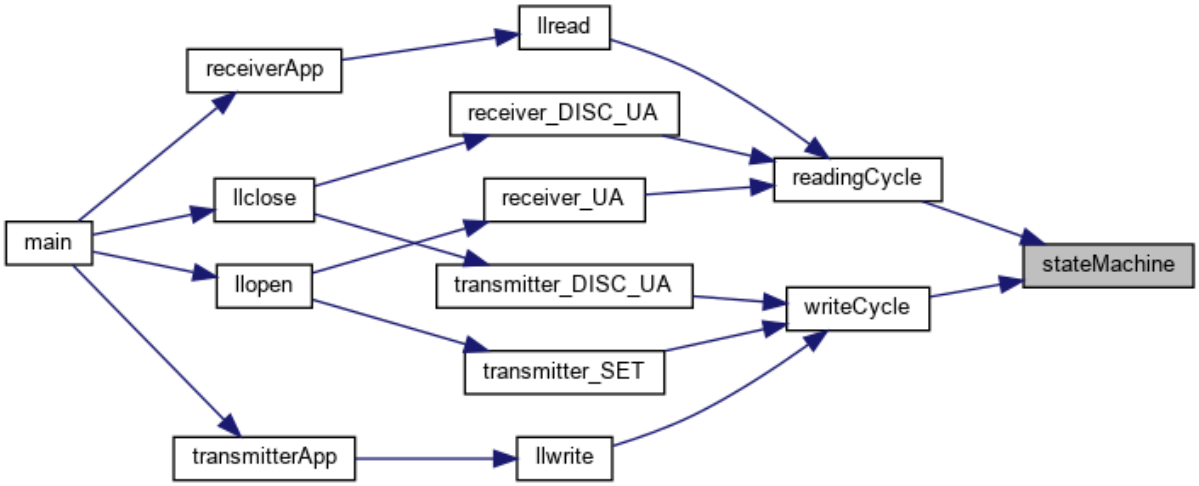
Para poder usar a nossa aplicação o utilizador deve em primeiro lugar compilar a aplicação a partir do *Makefile* disponibilizado, executando o comando “make” dentro da pasta src, usando um terminal à escolha. Com o programa compilado, este pode ser executado na seguinte forma: *./app [receiver|transmitter] [path] [porta]*

[receiver|transmitter] - uma *flag* que terá um dos valores para indicar se a aplicação representará o emissor ou o recetor.
[*path*] - o caminho relativo para o ficheiro a enviar no caso da *flag* anterior ser *transmitter* ou o caminho relativo para o diretório de destino do ficheiro a receber no caso da *flag* anterior ser *receiver*.
[porta] - Um número inteiro que indica a porta pela qual se quer fazer a ligação, este número será substituído por x em */dev/ttySx.*

O programa *receiver* deve ser o primeiro a ser iniciado para esperar que o *transmitter* inicie a ligação. Caso o *transmitter* seja iniciado primeiro, começará a tentar ligar ao *receiver* e se o número de tentativas for excedido, termina.

Assim que a ligação entre os dois programas for estabelecida aparece uma barra de progresso para mostrar que a transferência está a ser efetuada e possíveis mensagens de erro ou mensagens de aviso podem surgir na consola.

Relativamente a sequências de chamadas de funções, é a seguir apresentado um gráfico de chamadas simplificado relativo ao uso das funções principais do *DCP* a partir do módulo Aplicação:



Protocolo de ligação lógica

State Machine

Antes de especificar o que acontece nas funções principais é necessário explicar o módulo relacionado com a Máquina de Estados. Após ter feito as funções principais, o nosso código ficou verboso e repetitivo relativamente à forma como estávamos a fazer a verificação de cada byte nas tramas recebidas, já que tínhamos múltiplas funções a servir de máquinas de estado para cada função principal do *DCP*. Face a isto fizemos um *refactoring* e criamos uma máquina de estados universal, que possibilita diferentes configurações de acordo com o uso que terá.

A configuração da máquina de estados é guardada na *struct* *stateMachineParams* que contém 2 bytes com os valores esperados do byte de *Address* e do byte de *Control*, um *enum* com o tipo da máquina de estados e outro *enum* com o estado da máquina de estados.
Para configurar a máquina de estados antes de a usar, o programa usa a função *stateMachineSetUp* passando como argumentos os valores para preencher a *struc*t de parâmetros.

A função principal *stateMachine* recebe 3 argumentos: um byte a processar pela máquina de estados, um buffer de bytes passado por referência e um inteiro passado por referência. Na maior parte dos usos da máquina de estados a função é chamada na forma *stateMachine(byte, NULL, NULL)*, isto porque os últimos dois argumentos são só usados na função *llread()*: o buffer é usado para guardar os bytes de dados recebidos e a variável do tipo inteiro para guardar o tamanho do *buffer.*

A função *stateMachine* é simples de entender, consiste apenas de um *if* para guardar os bytes recebidos num *buffer*, caso o *type* da máquina de estados seja de leitura, e um *switch* que avalia o *state* da máquina e redirige o processamento do byte para outras funções. Nas funções que representam cada estado é que é feito o processamento do byte de acordo com a configuração da máquina.

Com esta mudança adotamos o método de, antes de usar a máquina de estados, mudamos a configuração para o que for preciso na situação pretendida e deixamos as duas funções explicadas na próxima secção usar a máquina de estados para efetuar a leitura das tramas. Este *refactoring* [Pull Request #28] possibilitou remover 240 linhas de código repetido

do módulo *DCP*, criando o módulo *State Machine* e tornou o uso da máquina de estados algo previsível e simples.

Read and Write Cycles

Todas as funções principais do DCP usam chamadas *read* e *write* com o *file descriptor* do cabo para trocar informações. Como estas transmissões são feitas byte a byte e algumas têm alarmes associados para o eventual caso de não se sucederem corretamente, todas as chamadas de *read* e *write* estavam envolvidas em ciclos *while* ou ciclos *do..while*, o que causava muito código repetido sempre que era preciso, numa função do *DCP*, efetuar uma transferência de dados. Com isso em mente criamos duas funções:

- *readingCycle* que contém um ciclo *while* que lê byte a byte de um *file descriptor* e retorna quando a leitura da trama pretendida for finalizada de acordo com a configuração da máquina de estados.
- *writeCycle* que contém um ciclo *do...while*, para enviar uma trama assegurada por um alarme e um número de tentativas em caso de falha, com um ciclo *while* no interior para efetuar a leitura da trama de resposta de acordo com a configuração da máquina de estados.

Com este *refactoring* [Pull Request #29] foi possível remover 230 linhas do módulo *DCP* e organizar melhor o código resultante. Juntamente com o *refactoring* do módulo *State Machine* tornou-se possível criar funções compreensíveis sem ter que verificar o código das funções interiores, tendo como exemplo a função usada pelo transmissor em *llopen* para mandar a trama não numerada *SET*: *transmitter_SET*

llopen()

No início do desenvolvimento do programa, tendo os ficheiros *writenoncanonical* e *noncanonical* feitos em aulas anteriores, tornou-se fácil a implementação da primeira função do *DCP*. Durante a execução é chamada a função *initConnection* que tem o dever de tratar do *setup* inicial: abrir a ligação com *open*, modificar as definições da porta com as funções *tcgetattr* e *tcsetattr* e inicializar a *struct linkLayer*.

A meio do desenvolvimento do programa verificamos que quando o *alarm* era ativado a meio de uma chamada *read*, o programa executava a função de *handler* do alarme e ao retornar o *read* continuava bloqueado. Para resolver isto tivemos de modificar as *flags* da *struct sigaction* associada ao sinal *SIGALRM* para que o *read* não fique bloqueado ao retornar da função *handler*. [Issue #4] Esta modificação é feita a após fazer *initConnection.*

No final do *llopen* temos um *switch* que, dependendo do tipo de aplicação, executa as funções *transmitter_SET* ou *receiver_UA*. A primeira trata de formar a trama *SET*, configurar a máquina de estados, iniciar um ciclo de escrita e verificar se foi bem sucedido, enquanto que a segunda modifica também a configuração da máquina de estados, inicia um ciclo de leitura e caso este seja bem sucedido, prepara a trama de confirmação *UA* e envia.

Assim que ambas as partes da aplicação conseguem estabelecer ligação o programa retorna da função *llopen* com sucesso. Caso ocorra um erro ambos os programas fecham com indicação na consola relativa ao erro ocorrido e em que secção do código.

llwrite()

De acordo com o guião de trabalho, a função *llwrite* recebe como argumentos o identificador da ligação de dados, um *array* de caracteres a transmitir e o comprimento do *array* de caracteres. Logo de início, caso ocorra um erro no qual o tamanho do *array* é maior do que *MAX_SIZE*, o tamanho máximo em bytes do campo de informação da trama I definido pelo programa, então a função retorna imediatamente com um erro.

Caso isso não se verifique é formado o cabeçalho da trama I composto por quatro bytes: *Flag* (0x7E), campo de endereço (0x03), campo de controlo com o número de sequência guardado na *linkLayer* (0x00 ou 0x40) e *BCC1* (*XOR* entre endereço e controlo). De seguida o *BCC2* é formado a partir dos dados recebidos no *array* de caracteres e é feito o *stuffing* dos dados e do *BCC2*.

A função auxiliar *fillFinalBuffer* trata de receber todos os buffers formados até ao momento e concatena-os de forma correta para formar *finalBuffer*, o *array* de caracteres que é usado de seguida na função *writeCycle* para enviar a trama I para o Recetor.

Antes da função retornar, o número de sequência é atualizado, caso não tenha havido erros na escrita, e a memória alocada dinamicamente para o *buffer* de dados e para o *BCC2* é libertada.

llread()

Como pretendido no guião de trabalho, a função *llread* recebe como argumentos o identificador da ligação de dados e um *array* para guardar os caracteres recebidos. No início da função é alterada a configuração da máquina de estados com recurso à função *stateMachineSetUp* e são criadas 3 variáveis que serão passadas por referência à função *readingCycle*: um buffer sem memória alocada de caracteres, um inteiro com valor zero que guardará o tamanho do buffer de dados lido e um byte que será preenchido com o valor do campo de controlo para a resposta.

Após este *setup* inicial o programa usa a função *readingCycle* para ler a trama de informação e processar cada byte na máquina de estados. Após o cabeçalho ser corretamente processado pelo máquina de estados, esta passa para o estado *BCC*_OK pertencente ao *enum stateMachineState* da *struct* relativa à configuração da máquina de estados. Durante este estado a máquina guarda todos os bytes lidos no *array frame* da *linkLayer* até encontrar o byte 0x7E, correspondente á *flag* final que sinaliza o fim da trama I.

Assim que a máquina de estados deteta que leu o último byte da trama pode proceder a fazer o *destuffing* dos dados recebidos e do *BCC2*, calcular o *BCC2* dos bytes recebidos após *destuffing* e comparar com o *BCC2* recebido para verificar erros na trama recebida e preencher as variáveis do *buffer* de dados e do tamanho do *buffer* de dados passadas por referência. Caso não hajam erros a variável *state* do *enum stateMachineState* toma o valor *DONE* e o ciclo de leitura é concluído, ficando o byte relativo ao campo de controlo de resposta preenchido de acordo com o valor de retorno da função *stateMachine* dentro da função *readingCycle*.

O progresso é retomado em *llread*, onde é criado um *buffer* de resposta que será mandado para o programa emissor com a trama não numerada *RR* ou *REJ* dependendo de como correu a leitura da trama recebida. Como passos finais, é preenchido o *buffer* recebido como argumento com os dados recebidos, a memória alocada dinamicamente é libertada e é retornado o tamanho do *buffer* de dados.

llclose()

De forma semelhante a *llopen*, o *llclose* é composto por um *switch* que reencaminha o programa para uma de duas funções: ou .

Na primeira função a sequência de chamadas usadas anteriormente é repetida: Forma-se a trama a enviar, configura-se a máquina de estados com *stateMachineSetUp* e envia-se a trama, neste caso *DISC*, com auxílio da função *writeCycle*. Se a trama receber como resposta outra trama *DISC* então é formada a trama final *UA* e enviada para o Recetor com recurso a uma chamada *write*. No último envio não precisamos de usar um ciclo de escrita porque o programa não está à espera de receber uma resposta. Com a escrita a ser bem sucedida a função pode retornar o identificador da ligação de dados.

Na função é alterada a configuração da máquina de estados para preparar para a leitura da trama *DISC*. Assim que a leitura é efetuada corretamente é preparada a trama *DISC* para envio com recurso à função *write*. Durante o desenvolvimento decidimos usar a chamada *write* invés da nossa função *writeCycle* para depois ler a trama de confirmação *UA* enviada pelo Emissor porque a função *writeCycle* foi construída de forma a utilizar várias tentativas com o recurso a *alarm*. Se tivéssemos usado a função *writeCycle*, caso a trama *UA* não tivesse sido recebida dentro de um período definido, o Recetor reenviaria a trama *DISC* e esperaria uma resposta. Do lado do emissor, assim que recebe a trama *DISC*, este envia a trama *UA* e retorna, o que faz com que tramas *DISC* que seriam recebidas posteriormente não sejam processadas, fazendo com que o Recetor fique a enviar tramas de forma repetida à espera de uma resposta que nunca chegará.

Para ter o Recetor a fechar corretamente, mesmo sem a resposta *UA*, este admite que ao fim de um período de tempo, como não recebe outra trama *DISC* do Emissor, a significar que o Emissor não recebeu a trama *DISC* de resposta do Recetor, então a trama *UA* foi perdida durante o envio e termina.

Isto é refletido no código da função depois da chamada *write*, na qual a configuração da máquina de estados é alterada para preparar para a leitura da trama *UA* e é ativado um *alarm* com o *timeout* da *linkLayer* mais um segundo. Após isto, um ciclo de leitura é chamado e o programa pode tomar dois rumos: o ciclo de leitura é feito com sucesso, lendo a trama *UA* e a função retorna, ou o ciclo de leitura não recebe a trama *UA*, ativando o *alarm* e, assumindo que o programa Emissor recebeu a trama corretamente, retorna.

No resto da função *llclose* é reposta a configuração inicial da porta alterada em *llopen*, é fechada a porta do identificador da ligação de dados e são restaurados os valores iniciais da *struct sigaction* relativa ao sinal de alarme.

Protocolo de aplicação

No início da função *main* é feito o *parse* dos argumentos da linha de comandos e, caso haja algum erro, uma mensagem explicativa de como usar o programa é apresentada e o programa fecha.

O programa procede para a execução da função *llopen* para abrir a ligação entre o Emissor e o Recetor, após isto, dependendo do tipo de aplicação, o programa executará as funções *transmitterApp* ou *receiverApp*. No final, a função *llclose* é chamada na *main* para ambas as aplicações em execução para terminar o programa.

Na função transmitterApp é usada a função *stat* para guardar informações relativas ao ficheiro a enviar, e a partir destes dados é possível construir o pacote de controlo, com uma *flag* de *start* no primeiro byte, e enviá-lo com uso da função *llwrite*. Após este envio é

iniciado um ciclo *while* que servirá para ler parte do ficheiro, construir um pacote de dados e enviar para o Recetor, até não restarem mais bytes para ler do ficheiro. Durante esta transferência é apresentada na consola uma barra de progresso que é reescrita a cada ciclo de leitura do ficheiro com o valor em percentagem do ficheiro que já foi transferido. No final da transferência é usado o pacote de controlo inicial com o primeiro byte alterado para a *flag* de *end* para indicar o final da transferência no lado do Recetor. Para finalizar, é apresentado na consola o tempo em milissegundos que a transferência demorou e o débito transmitido antes de retornar.

A função *receiverApp* é constituída por um ciclo *while* que faz uma leitura com *llread* e processa o que leu, tendo como condição de paragem receber o pacote de controlo final. No final da função há uma chamada de sistema *close* para fechar o ficheiro na qual o conteúdo lido é guardado.

Dentro do ciclo é feita a verificação do primeiro byte de cada pacote recebido. Se o byte for o de *start* (0x02) é usada a função *parseFileInfo* para processar os dados recebidos no pacote e é criado um ficheiro para onde se escreverão os dados recebidos. Se o byte for o de *end* (0x03) o ciclo é interrompido. Se o byte for de *data* (0x01), a barra de progresso é atualizada e o conteúdo lido é adicionado ao ficheiro.

Validação

Após a transferência do ficheiro ser bem sucedida, foi necessário verificarmos a robustez do nosso programa. Nesse sentido, foi interrompida a ligação via porta série, fazendo com que ocorra um *timeout* do lado do emissor pois não é recebida a mensagem de confirmação de receção da trama. Consequentemente, o emissor imprime uma mensagem de aviso a informar que o alarme foi ativado e que irá tentar enviar a trama novamente até receber uma confirmação ou atingir o número máximo de tentativas.

Num segundo teste simularam-se erros nos bytes da trama I, aos quais o programa lida de forma correta. Os bytes *BCC1* e *BCC2* têm como função detetar esses erros e, por sua vez, rejeitar a trama. Quando ocorre um *reject*, o emissor não ativa o alarme, pois este, ao contrário do caso descrito anteriormente, recebe uma mensagem de confirmação de receção por parte do recetor com o campo de controlo preenchido com o *REJ* adequado. Como o emissor recebe uma mensagem que o informa de que a trama enviada foi rejeitada faz uma retransmissão da mesma.

Outra situação que verificamos foi a receção de uma mensagem duplicada. Através do número de sequência, o recetor é capaz de identificar se recebeu ou não uma mensagem duplicada. Caso seja duplicada, é escrita uma mensagem de aviso que informa sobre o erro no byte do campo de controlo e que a trama já foi anteriormente lida pelo recetor. Nesta situação a trama que este deve enviar como resposta tem de ser coerente com a mensagem recebida, ou seja, não troca o valor do byte do campo de controlo correspondente ao *RR* para que o emissor saiba qual a trama a enviar de seguida. Assim, não ocorre nenhum *REJ* mas sim um *RR* sendo a trama descartada.

Eficiência do protocolo de ligação de dados

Para obter a eficiência começamos por medir o tempo que o programa demora a executar o ciclo de envio de todas as tramas I na camada da aplicação, no fim é chamada a função *log_bitsPerSecond* que calcula o débito recebido (R) através do tamanho do ficheiro.

O débito (R) é dividido pela capacidade de transmissão (C) dando-nos a eficiência (S) do programa.

Sendo este um mecanismo de *Stop & Wait* é sempre necessário um *acknowledgment* a cada trama enviada, que afeta a eficiência do protocolo. Esta eficiência representa a percentagem de tempo que o protocolo consegue usar o cabo para transferir bits. Se o valor de *a* for pequeno (*Tprop* muito menor que *Tf*) a eficiência é próxima de 1, e uma vez que Tprop é diretamente proporcional à distância, o valor de *a* será tanto menor quanto menor for a distância entre o emissor e o recetor, e consequentemente, melhor será a eficiência. Como no laboratório a distância é relativamente pequena, seria expectável uma eficiência próxima de 1, o nosso protocolo obteve uma eficiência de aproximadamente 0,76.

Além da distância, existem outros fatores que influenciam a eficiência pelo que elaboramos vários gráficos que demonstram a implicação da sua variação na eficiência do protocolo **(ver anexo III).**

Conclusões

Neste relatório descrevemos a estratégia de implementação do protocolo, as funções implementadas, os testes realizados e os resultados obtidos. Foi possível verificar os benefícios aliados à implementação do protocolo no estabelecimento de comunicações, entre os quais sincronização de tramas, mecanismos de verificação e recuperação de erros.

Tivemos oportunidade de melhorar as nossas competências ao nível de compreensão, programação e implementação de protocolos de comunicação.

Relativamente aos resultados obtidos, que podem ser visualizados em anexo, a eficiência foi cerca de 76%, que não é um mau resultado, mas acreditamos que poderia ser melhorado com uma melhor implementação de algumas funções.

Consideramos que implementamos com sucesso o protocolo, provando com os testes efetuados a robustez e a flexibilidade do mesmo.

Gostaríamos de aproveitar esta oportunidade para apresentar um melhoramento possível do primeiro trabalho laboratorial possível de acrescentar em anos futuros: O serviço GitHub Classroom para manutenção de repositórios remotos. Isto permite um melhor acompanhamento por parte dos professores das aulas práticas e serve como uma plataforma para guardar código, evitando uma submissão via moodle, que pode ser substituída pelo uso de *releases* e *tags*. Usar GitHub também abre a hipótese de criar um relatório final em estilo MarkDown que se integra com o resto do repositório.

Anexo I - Código fonte

application.h

```
/** \addtogroup Application
 *  @{
 */
#ifndef APP_HEADER
#define APP_HEADER

#include "dcp.h"
#include "logs.h"

/**
 * @brief Struct to save data relative to the Application
 */
struct applicationLayer{
  off_t filesize; // Size of file in bytes
  char filename[255];  // String with file name
  char destinationArg[255];  // String with path to destination message received
  char filenameArg[255]; //String with filename passed by programs arguments
  int type; // TRANSMITTER | RECEIVER
  int gate; // /dev/ttySx | gate is x
};

/**
 * @brief Função para dar parse dos argumentos da linah de comandos
 *
 * @param argc argc de main
 * @param argv argv de main
 */
void parseArgs(int argc, char** argv);

/**
 * @brief Transmitter part of the application
 *
 * @param fd file descriptor of connection
 * @return int 0 if all okay, negative otherwise
 */
int transmitterApp(int fd);

/**
 * @brief Parses the info of a Control Package to the Application Layer Struct
 *
 * @param controlpackage Control Package
 */
void parseFileInfo(unsigned char *controlpackage, int packagesize);
```

```
/**
 * @brief Receiver part of the application
 *
 * @param fd file descriptor of connection
 * @return int 0 if all okay, negative otherwise
 */
int receiverApp(int fd);

/**
 * @brief
 *
 * @param argc valor esperado: 3
 * @param argv argv[1] = receiver|transmitter argv[2]={0,1,10,11}
 * @return int
 */
int main(int argc, char** argv);

#endif // APP_HEADER
/** @}*/
```

application.c

```
/** \addtogroup Application
 *  @{
 */
#include "application.h"

static struct applicationLayer applayer;  // Struct to save data about the Application

void parseArgs(int argc, char** argv){
  if (argc != 4){
    log_error("Usage: ./application (receiver <destination> | transmitter <filename>) [gate =
{0,1,10,11}]\n");
    exit(-1);
  }

  applayer.gate = atoi(argv[3]);
  if (applayer.gate != 0 && applayer.gate != 1 && applayer.gate != 10 && applayer.gate != 11)
{
      log_caution("nPorta = {0,1,10,11}\n");
      exit(-1);
  }

  if (strcmp("receiver",argv[1])==0){
      applayer.type = RECEIVER;
      strcpy(applayer.destinationArg,argv[2]);
  }
  else if (strcmp("transmitter",argv[1])==0){
```

```
      applayer.type = TRANSMITTER;
      strcpy(applayer.filenameArg,argv[2]);
  }
  else {
      log_caution("Usage: ./application (receiver <destination> | transmitter <filename>) [gate
= {0,1,10,11}]\n");
      exit(-1);
  }
}

int transmitterApp(int fd){
  char controlPackage[255];
  char dataPackage[MAX_SIZE];

  struct timespec start_time_general;

  int sequenceNumber = 0;
  int nbytes;
  int file_fd;
  char * file_data[MAX_SIZE];

  struct stat fileInfo;
  startTime(&start_time_general);

  if (stat(applayer.filenameArg, &fileInfo)<0){
    log_error("transmitterApp() - Error obtaining file information.\n");
    return -1;
  }

  file_fd = open(applayer.filenameArg,O_RDONLY);
  if (file_fd <0){
    log_error("transmitterApp() - Error opening file.\n");
    return -1;
  }

  /*building control package*/
  controlPackage[0] = START;
  controlPackage[1] = T_SIZE;
  controlPackage[2] = sizeof(fileInfo.st_size); /* sending size of off_t */
  memcpy(&controlPackage[3],&fileInfo.st_size,sizeof(fileInfo.st_size));

  controlPackage[sizeof(fileInfo.st_size)+3] = T_NAME;
  controlPackage[sizeof(fileInfo.st_size)+4] = strlen(applayer.filenameArg);

memcpy(&controlPackage[sizeof(fileInfo.st_size)+5],applayer.filenameArg,strlen(applayer.fil
enameArg));

  if(llwrite(fd,controlPackage,sizeof(fileInfo.st_size) + 5 + strlen(applayer.filenameArg))<0){
    log_error("transmitterApp() - llwrite() failed writing Start Control Packet");
```

```
    return -1;
  }

  int progress = 0;

  while( (nbytes = read(file_fd,file_data,MAX_SIZE-4)) != 0){
    /*building data package*/
    dataPackage[0] = DATA;
    dataPackage[1] = sequenceNumber % 255;
    dataPackage[2] = nbytes / 256;
    dataPackage[3] = nbytes % 256;
    memcpy(&dataPackage[4],file_data,nbytes);

    progress+=nbytes;
    printProgressBar(progress,fileInfo.st_size);

    if (llwrite(fd,dataPackage,nbytes+4) < 0){
      log_error("transmitterApp() - llwrite() failed writing Data Packet");
      return -1;
    }
    sequenceNumber++;
    clearProgressBar();
  }

  printProgressBar(1,1);

  controlPackage[0] = END;

  if(llwrite(fd,controlPackage,sizeof(fileInfo.st_size) + 5 + strlen(applayer.filenameArg))<0){
    log_error("transmitterApp() - llwrite() failed writing End Control Packet");
    return -1;
  }

  log_elapsedTime("\n\nTotal time elapsed: ",start_time_general);
  log_bitsPerSecond(fileInfo.st_size,start_time_general);

  return 0;
}

void parseFileInfo(unsigned char *controlpackage, int packagesize){
  off_t size = 0;
  int informationsize;
  int next_tlv = 1;

  while (next_tlv != packagesize)
  {
    if (controlpackage[next_tlv]==T_SIZE){
      informationsize = controlpackage[next_tlv+1];
      for (int i = next_tlv+2, k=0; i < informationsize+next_tlv+2; i++, k++){
```

```
        size += controlpackage[i] << 8*k;
      }
      applayer.filesize = size;
      next_tlv=informationsize+3;
    }

    if (controlpackage[next_tlv]==T_NAME){
      informationsize = controlpackage[next_tlv+1];
      for (int i =next_tlv+2, k=0; i < informationsize+next_tlv+2; i++, k++){
        applayer.filename[k] = controlpackage[i];
      }
      applayer.filename[informationsize+next_tlv+2] = '\0';
      next_tlv = next_tlv+2 + informationsize;
    }
  }
}

int receiverApp(int fd){
  unsigned char package[MAX_SIZE];
  int dataPackageSize;
  int nsequence_expected=0, nsequence_expected_aux;
  int offset_need = 0;

  int file_fd, sizeread;

  int progress = 0;

  while(1){
    sizeread = llread(fd,package);

    if(sizeread<0)
      log_caution("receiverApp() - llread() error");

    if (package[0] == START){
      parseFileInfo(package, sizeread);
      strcat(applayer.destinationArg,applayer.filename);
      file_fd = open(applayer.destinationArg,O_RDWR | O_CREAT, 0777);
      continue;
    }
    else if (package[0] == END){
      break;
    }
    else if (package[0] == DATA && sizeread>0) {

      /*printing bar*/
      progress+=sizeread;
      printProgressBar(progress,applayer.filesize);
```

```
      dataPackageSize = package[3] + 256* package[2];

      if (package[1] != nsequence_expected){
        off_t offset = (package[1] + offset_need) * (MAX_SIZE-4) ;
        lseek(file_fd, offset, SEEK_SET);
      }

      write(file_fd,&package[4], dataPackageSize);

      if (package[1] != nsequence_expected){
        lseek(file_fd, 0 , SEEK_HOLE);
      }

      if (package[1]== nsequence_expected){
        nsequence_expected++;
      }

      nsequence_expected_aux = nsequence_expected;
      nsequence_expected %= 255;

      if (nsequence_expected % 255 == 0 && nsequence_expected_aux!= 0){
        offset_need+=255;
      }
    }

    clearProgressBar();
  }

  printProgressBar(1,1);

  if (close(file_fd)<0){
    log_error("Error closing file\n");
  }

  return fd;
}

int main(int argc, char** argv) {
  printf("Started App\n");

  int fd;
  parseArgs(argc, argv);

  fd = llopen(applayer.gate, applayer.type);

  if (fd < 0) {
    log_error("main() - Unable to establish connection. Exiting..  ");
    return -1;
  } else log_success("Connection established.");
```

```
  if (applayer.type == TRANSMITTER)
  {
    if( transmitterApp(fd) < 0)
      log_error("main() - transmitterApp error");
  }
  else if (applayer.type == RECEIVER) {
    if (receiverApp(fd) <0)
      log_error("main() - receiverApp error");
  }

  printf("Closing Connection..\n");
  if (llclose(fd) < 0){
    log_error("main() - Error on llclose()");
    return -1;
  }
  return 0;
}
/** @}*/
```

dcp.h

```
#ifndef PLA_HEADER
#define PLA_HEADER
#define _GNU_SOURCE
/** \addtogroup Data_Connection_Protocol
 *  @{
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <termios.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

#include "macros.h"
#include "logs.h"
#include "statemachine.h"
#include "dcp_spec.h"

/**
 * @brief Struct to hold information about the Link Layer
```

```
 */
struct linkLayer {
  char port[20]; /* Device /dev/ttySx, x = 0, 1*/
  int baudRate; /*Transmission Speed*/
  unsigned char sequenceNumber;   /*Frame Sequence Number: 0x00, 0x01*/
  unsigned int timeout; /*Alarm Timeout: x s*/
  unsigned int numTransmissions; /*Number of tries in case of failure*/
  unsigned char frame[MAX_SIZE_AFT_STUFF]; /*Frame with double the size of
MAX_SIZE to accomodate for byte_stuffing increase*/
  unsigned int status; /*TRANSMITTER | RECEIVER*/
};


struct linkLayer linkLayer;  ///< Data relative to the link layer
struct termios oldtio; ///< used in llclose to reset termios
volatile int failed; ///< used in the alarm handler


/**
 * @brief Opens a data connection with the serial port
 *
 * @param porta number of the port x in  "/dev/ttySx"
 * @param type TRANSMITTER|RECEIVER
 * @return int idata connection identifier or -1 in case of error
 */
int llopen(int porta, int type);

/**
 * @brief Writes the content of buffer to fd
 *
 * @param fd data connection identifier
 * @param buffer character array of data to send
 * @param lenght array's size
 * @return int amount of characters read or -1 in case of error
 */
int llwrite(int fd, char *buffer, int lenght);

/**
 * @brief Reads from fd to buffer
 *
 * @param fd data connection identifier
 * @param buffer character array of received data
 * @return int array's size or -1 in case of error
 */
int llread(int fd, unsigned char *buffer);

/**
 * @brief Closes the connection to the serial port
 *
```

```
 * @param fd data connection identifier
 * @return int positive value in case of sucess, -1 in case of error
 */
int llclose(int fd);

#endif // PLA_HEADER
/** @}*/
```

dcp.c

```
/** \addtogroup Data_Connection_Protocol
 *  @{
 */
#include "dcp.h"

static struct sigaction old_action; ///< sigaction to restore SIGALRN handler
int testSend=0;

int llopen(int porta, int type){
  int fd;
  char port[12]; // "/dev/ttyS11\0" <- 12 chars
  snprintf(port, 12, "/dev/ttyS%d", porta);

  if (initConnection(&fd, port) < 0){
    log_error("llopen() - Error on initConnection");
    return -1;
  }

  // Set Handler for Alarm
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = atende;
  sa.sa_flags = 0;
  if (sigaction(SIGALRM, &sa, &old_action)<0){
    log_error("llopen() - error on sigaction");
    return -1;
  }

  switch (type)
  {
  case TRANSMITTER:
    linkLayer.status = TRANSMITTER;
    return transmitter_SET(fd);
    break;
  case RECEIVER:
    linkLayer.status = RECEIVER;
    return receiver_UA(fd);
    break;
```

```
  default:
    log_error("llopen() - Wrong value for variable type");
    return -1;
    break;
  }

}

int llwrite(int fd, char *buffer, int lenght){
  int currentLenght = lenght;
  unsigned char buf1[4] = {FLAG, A_ER, C_I(linkLayer.sequenceNumber), BCC(A_ER,
C_I(linkLayer.sequenceNumber))};
  unsigned char *dataBuffer = (unsigned char *)malloc(lenght);

  if (lenght > MAX_SIZE){
    log_error("llwrite() - Message size greater than MAX_SIZE");
    return -1;
  }

  /*building trama I*/
  unsigned char BCC2 = buffer[0];
  for (int i = 1; i<lenght; i++){
    BCC2 = BCC2 ^ buffer[i];
  }

  unsigned char *buf2 = (unsigned char *)malloc(2);
  int buf2Size = 2;
  // Byte Stuffing (BCC2 on buf2)
  if (BCC2 == 0x7E || BCC2 == 0x7D){
    buf2 = (unsigned char *) realloc(buf2, 3);
    buf2[0] = 0x7D;
    buf2[1] = BCC2 ^ 0x20;
    buf2[2] = FLAG;
    buf2Size=3;
  }
  else{
    buf2[0] = BCC2;
    buf2[1] = FLAG;
  }

  // Byte Stuffing (data buffer)
  for (int i = 0, k=0; i<lenght; i++, k++){
    if (buffer[i] == 0x7E || buffer[i] == 0x7D){
      currentLenght++;
      dataBuffer = (unsigned char *) realloc(dataBuffer, currentLenght);

      dataBuffer[k+1] = buffer[i] ^ 0x20;
      dataBuffer[k] = 0x7D;
      k++;
```

```
    }
    else{
      dataBuffer[k] = buffer[i];
    }
  }

  unsigned char finalBuffer[currentLenght + 4 + buf2Size]; /*trama I completa*/

  fillFinalBuffer(finalBuffer, buf1, buf2, buf2Size, dataBuffer, currentLenght);

  stateMachineSetUp(C_RR(linkLayer.sequenceNumber^0x01), A_ER, Start, Write);

  if (writeCycle(writeR, fd, finalBuffer, sizeof(finalBuffer)) <0){
    log_error("llwrite() - failed writing");
    return -1;
  }
  linkLayer.sequenceNumber ^= 0x01;
  free(dataBuffer);
  free(buf2);
  return 0;
}

int llread(int fd, unsigned char *buffer){
  unsigned char *dataBuf;
  int retBufferSize = 0;
  stateMachineSetUp(C_I(linkLayer.sequenceNumber), A_ER, Start, Read);

  unsigned char c;

  int reading_r = readingCycle(readR, fd, &c, &dataBuf, &retBufferSize);
  if ( reading_r == -1){
    log_error("llread() - failed reading");
    tcflush(fd, TCIFLUSH);
    return -1;
  }

  unsigned char replyBuf[5] = {FLAG, A_ER, c, BCC(A_ER, c), FLAG};

  tcflush(fd, TCOFLUSH);
  int res = write(fd,replyBuf,5);
  if (res == -1) {
    log_error("llread() - Failed writing response to buffer.");
    return -1;}

  if (reading_r != -2){
    for (int i = 0; i < retBufferSize; i++)
      buffer[i] = dataBuf[i];

    free(dataBuf);
```

```
  }

  return retBufferSize;
}

int llclose(int fd){
  int returnValue = fd;

  switch (linkLayer.status)
  {
  case TRANSMITTER:

    if (transmitter_DISC_UA(fd) <0)
      returnValue = -1;
    break;
  case RECEIVER:
    if (receiver_DISC_UA(fd) <0)
      returnValue = -1;
    break;
  }

  if (tcsetattr(fd,TCSANOW,&oldtio) != 0){
    log_error("llclose() - Error on tcsetattr()");
    return -1;
  }
  if (close(fd) != 0){
    log_error("llclose() - Error on close()");
    return -1;
  }
  if (sigaction(SIGALRM, &old_action, NULL)<0){
    log_error("llclose() - error on sigaction");
    return -1;
  }

  if (returnValue>0)
    log_success("Closing Successful");

  return returnValue;
}
/** @}*/
```

dcp_spec.h

```
/** \addtogroup Data_Connection_Protocol_Specification
 *  @{
 */
#ifndef PLD_SPEC_HEADER
#define PLD_SPEC_HEADER
```

```
#include "dcp.h"

enum readingType {openR, readR, closeDISC, closeUA}; // Enum of possible modifications to readingCycle()
enum writingType {trans_SET, writeR, trans_DISC_UA}; // Enum of possible modifications to writeCycle()

/**
 * @brief Function to read a byte from fd
 *
 * @param type variable to distinguish warning messages
 * @param fd file descriptor
 * @param c controll byte, used with type readR
 * @param dataBuf buffer to read data, used with type readR
 * @param retBufferSize variable to store dataBuf size, used with type readR
 * @return int
 */
int readingCycle(enum readingType type, int fd, unsigned char *c, unsigned char **dataBuf, int *retBufferSize);
/**
 * @brief Function used to write to fd
 *
 * @param type variable to distinguish warning messages
 * @param fd file descriptor
 * @param buf buffer of content to write
 * @param bufsize lenght of buffer in bytes
 * @return int negative in case of errors, 0 otherwise
 */
int writeCycle(enum writingType type, int fd, unsigned char *buf, int bufsize);

/**
 * @brief Estabelece ligação ao cabo e cria fd
 *
 * @param fd file descriptor da ligação
 * @param port "/dev/ttySx"
 */
int initConnection(int *fd, char *port);
/**
 * @brief Função handler do sinal de alarme
 *
 */
void atende();
/**
 * @brief Função que envia Trama SET e recebe trama UA
 *
 * @param fd identificador da ligação de dados
 * @return int identificador da ligação de dados OU -1 em caso de erro
 */
```

```
int transmitter_SET(int fd);
/**
 * @brief Função que recebe trama SET e envia trama UA
 *
 * @param fd identificador da ligação de dados
 * @return int identificador da ligação de dados OU -1 em caso de erro
 */
int receiver_UA(int fd);

/**
 * @brief Stuffs the final buffer with the Information Packet
 *
 * @param finalBuffer pointer to the Information buffer
 * @param headerBuf pointer to the header buffer [FLAG, A, C, BCC1]
 * @param footerBuf pointer to the footer buffer [BCC2, FLAG]
 * @param footerBufSize size of footer buffer
 * @param dataBuffer pointer to the Data Buffer [D1, D2, ..., Dn]
 * @param dataSize size of Data Buffer
 * @return int size of finalBuffer
 */
int fillFinalBuffer(unsigned char* finalBuffer, unsigned char* headerBuf, unsigned char*
footerBuf, int footerBufSize, unsigned char* dataBuffer, int dataSize);

/**
 * @brief Transmitter sequence to Disconnect
 *
 * @param fd file descriptor ot the connection
 * @return int -1 in case of errors, 0 otherwise
 */
int transmitter_DISC_UA(int fd);

/**
 * @brief Receiver sequence to Disconnect
 *
 * @param fd file descriptor ot the connection
 * @return int -1 in case of errors, 0 otherwise
 */
int receiver_DISC_UA(int fd);

/**
 * @brief Generates a error on BCC2 based on PROBABILITY_BCC2
 *
 * @param frame frame to generate error on
 * @param frameSize size of frame
 */
void generateErrorBCC2(unsigned char *frame, int frameSize);

/**
 * @brief Generates a error on BCC based on PROBABILITY_BCC2
```

```
 *
 * @param checkBuffer Buffer with Address Byte and Control Byte to generate error on
 */
void generateErrorBCC1(unsigned int *checkBuffer);

#endif // PLD_SPEC_HEADER
/** @}*/
```

dcp_spec.c

```
/** \addtogroup Data_Connection_Protocol_Specification
 *  @{
 */
#include "dcp_spec.h"
int testSend1=0;

int readingCycle(enum readingType type, int fd, unsigned char *c, unsigned char
**dataBuf, int *retBufferSize){
  volatile int STOP=FALSE;
  int res;
  unsigned char buf[1];

  while (STOP==FALSE) {       /* loop for input */
    res = read(fd,buf,1);   /* returns after 1 char has been input */
    if (res == 0)
      continue;

    if (type == closeUA)
    {
      if (res == -1 && errno == EINTR) {  /*returns -1 when interrupted by SIGALRM and sets
errno to EINTR*/
      log_caution("readingCycle() - Failed reading UA from transmitter.");
      return -1;
      } else if (res == -1){
      log_error("readingCycle() - Failed reading UA from buffer.");
      return -1;}
    }
    else
    {
      if (res == -1) {
        switch (type)
        {
        case openR:
          log_error("readingCycle() failed reading SET from buffer.");
          break;
        case readR:
          log_error("readingCycle() - Failed reading frame from buffer.");
          break;
```

```
        case closeDISC:
          log_error("readingCycle() - Failed reading DISC from buffer.");
          break;
        default:
          log_error("readingCycle() - Unknown type");
          break;
        }
      log_error("readingCycle() - Error on read()");
      return -1;
      }
    }

    if (type == readR)
    {
      int retStateMachine = stateMachine(buf[0], dataBuf, retBufferSize);
      if (retStateMachine == -1){
        *c = C_REJ(linkLayer.sequenceNumber);
        log_error("readingCycle() - Error in BCC");
        return -1;
      }
      if (retStateMachine == -2){
        *c = C_RR(linkLayer.sequenceNumber);
        log_caution("readingCycle() - Error in C, wrong sequence number");
        return -2;
      }
      *c = C_RR(linkLayer.sequenceNumber);
    }
    else
      stateMachine(buf[0], NULL, NULL);

    if (state_machine.state == DONE) STOP=TRUE;
  }
  return 0;
}

int writeCycle(enum writingType type, int fd, unsigned char *buf, int bufsize){
  int attempt = 0, res;
  volatile int STOP=FALSE;

  do{
    attempt++;
    tcflush(fd,TCIFLUSH);
    res = write(fd,buf,bufsize);
    if (res == -1) {
      switch (type)
      {
      case trans_SET:
        log_error("writeCycle() - Failed writing SET to buffer.");
        break;
```

```
      case writeR:
        log_error("writeCycle() - Failed writing data to buffer.");
        break;
      case trans_DISC_UA:
        log_error("writeCycle() - Failed writing DISC to buffer.");
        break;
      default:
        log_error("writeCycle() - Unknown type");
        return -1;
        break;
      }
      log_error("writeCycle() - read error");
      return -1;
    }

    alarm(linkLayer.timeout);
    failed = FALSE;
    state_machine.state = Start;
    unsigned char buf_r[1]={};

    while (STOP==FALSE) {
      res = read(fd,buf_r,1);   /* returns after 1 char has been input */

      if (res == 0)
        continue;

      if (res == -1 && errno == EINTR) {  /*returns -1 when interrupted by SIGALRM and sets
errno to EINTR*/
        switch (type)
        {
        case trans_SET:
          log_caution("writeCycle() - Failed reading UA from receiver.");
          break;
        case writeR:
          log_caution("writeCycle() - failed reading RR from receiver.");
          break;
        case trans_DISC_UA:
          log_caution("writeCycle() - Failed reading DISC from receiver.");
          break;
        default:
          log_error("writeCycle() - Unknown type");
          return -1;
          break;
        }
        if (attempt < linkLayer.numTransmissions)
          log_caution("Trying again...");
        else{
          log_error("writeCycle() - Exceded number of attempts");
          return -1;
```

```
        }
        break;
      } else if (res == -1){
        switch (type)
        {
        case trans_SET:
          log_error("writeCycle() - Failed reading UA from buffer.");
          break;
        case writeR:
          log_error("writeCycle() - Failed reading RR from buffer.");
          break;
        case trans_DISC_UA:
          log_error("writeCycle() - Failed reading DISC from buffer.");
          break;
        default:
          log_error("writeCycle() - Unknown type");
          return -1;
          break;
        }
        log_error("writeCycle() - Error on read()");
        return -1;
      }

      if (type == writeR){
        if (stateMachine(buf_r[0], NULL, NULL) < 0){
          log_error("llwrite() - Error while receiving RR or REJ");
          failed = TRUE;
          alarm(0);
          break;
        }
      } else {
        stateMachine(buf_r[0], NULL, NULL);
      }

      if (state_machine.state == DONE || failed) STOP=TRUE;
    }
    tcflush(fd, TCIOFLUSH);
  }while (attempt < linkLayer.numTransmissions && failed);

  alarm(0);
  return 0;
}


int initConnection(int *fd, char *port){
  struct termios newtio;

  *fd = open(port, O_RDWR | O_NOCTTY );
  if (fd < 0) {perror(port); exit(-1); }
```

```
  if ( tcgetattr(*fd,&oldtio) == -1) { /* save current port settings */
    log_error("initConnection() - Error on tcgetattr - check cable connection or port number");
    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");
    exit(-1);
  }

  strcpy(linkLayer.port,port);
  linkLayer.baudRate = BAUDRATE;
  linkLayer.sequenceNumber = 0x00;
  linkLayer.timeout = TIME_OUT;
  linkLayer.numTransmissions = ATTEMPTS;

  log_success("New termios structure set");
  return 0;
}

void atende(){
  if (state_machine.state != DONE){
    failed = TRUE;
    log_caution("\nAlarm Activated");
    return;
  }
}

int transmitter_SET(int fd){
  unsigned char buf[SU_FRAME_SIZE] = {FLAG, A_ER, C_SET, BCC(A_ER, C_SET),
FLAG};

  stateMachineSetUp(C_UA, A_ER, Start, Supervision);

  writeCycle(trans_SET, fd, buf, SU_FRAME_SIZE);

  if(failed == TRUE){
```

```
    log_error("transmitter_SET() - Failed all attempts");
    return -1;
  }
  return fd;
}

int receiver_UA(int fd){
  stateMachineSetUp(C_SET, A_ER, Start, Supervision);


  if (readingCycle(openR, fd, NULL, NULL, NULL) < 0){
    log_error("receiver_UA() - Error on reading Cycle");
    return -1;
  }

  unsigned char replyBuf[SU_FRAME_SIZE] = {FLAG, A_ER, C_UA, BCC(A_ER, C_UA),
FLAG};

  int res = write(fd,replyBuf,SU_FRAME_SIZE); //+1 para enviar o \0
  if (res == -1) {
    log_error("receiver_UA() - Failed writing UA to buffer.");
    return -1;}

  return fd;
}


int fillFinalBuffer(unsigned char* finalBuffer, unsigned char* headerBuf, unsigned char*
footerBuf, int footerBufSize, unsigned char* dataBuffer, int dataSize){
  int finalIndex = 0, dataIndex = 0, footerBufIndex=0;

  while (finalIndex < 4){
  finalBuffer[finalIndex] = headerBuf[finalIndex];
  finalIndex++;
  }
  while (dataIndex < dataSize){
    finalBuffer[finalIndex] = dataBuffer[dataIndex];
    finalIndex++; dataIndex++;
  }
  while (footerBufIndex < footerBufSize){
    finalBuffer[finalIndex] = footerBuf[footerBufIndex];
    finalIndex++; footerBufIndex++;
  }

  return finalIndex;
}


int transmitter_DISC_UA(int fd){
```

```
  unsigned char buf[SU_FRAME_SIZE] = {FLAG, A_ER, C_DISC, BCC(A_ER, C_DISC),
FLAG};

  stateMachineSetUp(C_DISC, A_RE, Start, Supervision);

  writeCycle(trans_DISC_UA, fd, buf, SU_FRAME_SIZE);

  if(failed == TRUE){
    log_error("transmitter_DISC_UA() - Failed all attempts");
    return -1;
  }

  unsigned char buf1[SU_FRAME_SIZE] = {FLAG, A_RE, C_UA, BCC(A_RE, C_UA),
FLAG};

  int res = write(fd,buf1,SU_FRAME_SIZE);
  if (res == -1) {
    log_error("transmitter_DISC_UA() - Failed writing UA to buffer.");
    return -1;
  }

  return fd;
}

int receiver_DISC_UA(int fd){
  stateMachineSetUp(C_DISC, A_ER, Start, Supervision);

  /* parse DISC*/
  readingCycle(closeDISC, fd, NULL, NULL, NULL);

  unsigned char replyBuf[SU_FRAME_SIZE] = {FLAG, A_RE, C_DISC, BCC(A_RE,
C_DISC), FLAG};

  int res = write(fd,replyBuf,SU_FRAME_SIZE);
  if (res == -1) {
    log_error("receiver_DISC_UA() - Failed writing DISC to buffer.");
    return -1;
  }

  stateMachineSetUp(C_UA, A_RE, Start, Supervision);

  alarm(linkLayer.timeout+1); /* waits a limited time for UA response from Transmitter */
  /* parse UA*/
  if (readingCycle(closeDISC, fd, NULL, NULL, NULL) < 0){
    log_error("receiver_DISC_UA() - error on reading Cycle");
    return -1;
  }

  alarm(0);
```

```
  return fd;
}


void generateErrorBCC2(unsigned char *frame, int frameSize){
  int prob = (rand() % 100) + 1;

  if (prob <= PROBABILITY_BCC2){
    int i = (rand() % (frameSize - 5)) + 4; /* only considering data and BCC2*/
    unsigned char randomAscii = (unsigned char)((rand() % 177));
    frame[i] = randomAscii;
    log_message("\nGenerate BCC2 with errors.\n");
  }
}

void generateErrorBCC1(unsigned int *checkBuffer){
  int prob = (rand() % 100) + 1;
  if (prob <= PROBABILITY_BCC1)
  {
    int i = (rand() % 2);
    unsigned char randomAscii = (unsigned char)((rand() % 177));
    checkBuffer[i] = randomAscii;
    log_message("\nGenerate BCC1 with errors.");
  }
}
/** @}*/
```

statemachine.h

```
/** \addtogroup State_Machine
 *  @{
 */
#ifndef SM_HEADER
#define SM_HEADER

#include "macros.h"
#include "logs.h"
#include "dcp.h"

enum stateMachineType {Supervision, Write, Read};   // Type of State Machine
enum stateMachineState {Start, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DONE};  // State
of the State Machine

/**
 * @brief List of Parameters to Personalize the State Machine
 */
struct stateMachineParams {
  unsigned char control; // Control Byte that is suppossed to be received
```

```
  unsigned char address; // Address Byte that is suppossed to be received
  enum stateMachineState state;
  enum stateMachineType type;
};

struct stateMachineParams state_machine; // Instance of the stateMachineParams struct

/**
 * @brief Function to easily change the State Machine
 *
 * @param control Control Byte that is suppossed to be received
 * @param address Address Byte that is suppossed to be received
 * @param state State of the State Machine
 * @param type Type of State Machine
 */
void stateMachineSetUp(unsigned char control, unsigned char address, enum
stateMachineState state, enum stateMachineType type);
/**
 * @brief The StateMachine function that processes a byte and updates it's state
 *
 * @param byte byte to process
 * @param buf buffer to put data if stateMachineType is Read
 * @param size variable to keeo size of buf if stateMachineType is Read
 * @return int negative value in case of error, 0 otherwise
 */
int stateMachine(unsigned char byte, unsigned char **buf, int *size);

/**
 * @brief Function to process a byte in case the state is Start
 *
 * @param byte byte to be processed
 * @return int negative value in case of error, 0 otherwise
 */
int processStart(unsigned char byte);
/**
 * @brief Function to process a byte in case the state is FLAG_RCV
 *
 * @param byte byte to be processed
 * @return int negative value in case of error, 0 otherwise
 */
int processFLAG_RCV(unsigned char byte);
/**
 * @brief Function to process a byte in case the state is A_RCV
 *
 * @param byte byte to be processed
 * @return int negative value in case of error, 0 otherwise
 */
int processA_RCV(unsigned char byte);
/**
```

```
 * @brief Function to process a byte in case the state is C_RCV
 *
 * @param byte byte to be processed
 * @return int negative value in case of error, 0 otherwise
 */
int processC_RCV(unsigned char byte);
/**
 * @brief Function to process a byte in case the state is BCC_OK
 *
 * @param byte byte to be processed
 * @param buffer buffer to put data if stateMachineType is Read
 * @param buffersize variable to keeo size of buf if stateMachineType is Read
 * @return int negative value in case of error, 0 otherwise
 */
int processBCC_OK(unsigned char byte, unsigned char **buffer, int *buffersize);

#endif // PLA_HEADER
/** @}*/
```

statemachine.c

```
/** \addtogroup State_Machine
 *  @{
 */
#include "statemachine.h"

static unsigned checkBuffer[2]; // Buffer to hold Address and Control Bytes to check BCC1
static int frameIndex, wrongC;

void stateMachineSetUp(unsigned char control, unsigned char address, enum
stateMachineState state, enum stateMachineType type){
  state_machine.control = control;
  state_machine.address = address;
  state_machine.state = state;
  state_machine.type = type;
}

int stateMachine(unsigned char byte, unsigned char **buf, int *size){
  if (state_machine.type == Read) linkLayer.frame[frameIndex] = byte;

  switch (state_machine.state)
  {
  case Start:
    return processStart(byte);
    break;
  case FLAG_RCV:
    return processFLAG_RCV(byte);
    break;
```

```
  case A_RCV:
    return processA_RCV(byte);
    break;
  case C_RCV:
    return processC_RCV(byte);
    break;
  case BCC_OK:
    return processBCC_OK(byte, buf, size);
    break;
  case DONE:
    return 0;
    break;
  default:
    log_error("stateMachine() - unknown state value");
    return -1;
    break;
  }
  return 0;
}

int processStart(unsigned char byte){
  if (state_machine.type == Read)
  {
    frameIndex = 0;
    wrongC = FALSE;
    if (byte == FLAG){
      state_machine.state = FLAG_RCV;
      frameIndex++;
    }
  }
  else
  {
    if (byte == FLAG)
      state_machine.state = FLAG_RCV;
  }
  return 0;
}

int processFLAG_RCV(unsigned char byte){
  if (byte == state_machine.address) {
      state_machine.state = A_RCV;
      checkBuffer[0] = byte;
      if (state_machine.type == Read)
        frameIndex++;
    }
    else if (byte != FLAG)
      state_machine.state = Start;
  return 0;
}
```

```
int processA_RCV(unsigned char byte){
  switch (state_machine.type)
  {
  case Supervision:
    if (byte == state_machine.control) {
      state_machine.state = C_RCV;
      checkBuffer[1] = byte;
      }
    else if (byte == FLAG)
      state_machine.state = FLAG_RCV;
    else
      state_machine.state = Start;
    break;

  case Write:
    if (byte == C_REJ(linkLayer.sequenceNumber^0x01)){
      log_error("processA_RCV() - Reject Control Byte Received");
      return -1;
    }

    if (byte == state_machine.control) { // Control esperado
      state_machine.state = C_RCV;
      checkBuffer[1] = byte;
      }
    else if (byte == FLAG)
      state_machine.state = FLAG_RCV;
    else{
      state_machine.state = Start;
    }
    break;

  case Read:
    if (byte == C_I(linkLayer.sequenceNumber ^ 1)) {
      log_caution("processA_RCV() - Control Byte with wrong sequence number, need to
check BCC");
      wrongC = TRUE;
      state_machine.state = C_RCV;
      checkBuffer[1] = byte;
      frameIndex++;
    }
    else if (byte == state_machine.control) {  // Control esperado
      state_machine.state = C_RCV;
      checkBuffer[1] = byte;
      frameIndex++;
    } else if (byte == FLAG){
      state_machine.state = FLAG_RCV;
      frameIndex = 1;
    } else
```

```
      state_machine.state = Start;
    break;
  default:
    log_error("processA_RCV() - unknown type value");
    return -1;
    break;
  }
  return 0;
}

int processC_RCV(unsigned char byte){

  if(PROBABILITY_BCC1 != 0 && state_machine.type == Read){
      generateErrorBCC1(checkBuffer);
  }

  if (byte == BCC(checkBuffer[0],checkBuffer[1])){
    if (state_machine.type == Read && wrongC == TRUE){
      log_caution("processC_RCV() - Received already read Packet");
      return -2;
    }
    state_machine.state = BCC_OK;
    if (state_machine.type == Read)
      frameIndex++;
  }
  else if (byte == FLAG)
  {
    state_machine.state = FLAG_RCV;
    if (state_machine.type == Read)
      frameIndex = 1;
  }
  else
  {
    switch (state_machine.type)
    {
    case Supervision:
      state_machine.state = Start;
      break;
    case Write:
      log_error("processC_RCV() - Error in BCC");
      return -1;
      break;
    case Read:
      log_error("processC_RCV() - BCC received with Errors");
      return -1;
      break;
    default:
      log_error("processC_RCV() - unknown type value");
      return -1;
```

```
      break;
    }
  }
  return 0;
}

int processBCC_OK(unsigned char byte, unsigned char **buffer, int *buffersize){
  if (state_machine.type == Supervision || state_machine.type == Write)
  {
    if (byte == FLAG)
      state_machine.state = DONE;
    else
      state_machine.state = Start;
  }
  else if (state_machine.type == Read)
  {
    frameIndex++;
    if (byte == FLAG){

      if(PROBABILITY_BCC2 != 0){
          generateErrorBCC2(linkLayer.frame,frameIndex+1);
      }

      sleep(T_PROP_DELAY);

      *buffer = (unsigned char *)malloc((frameIndex-6));
      *buffersize = 0;

      unsigned char BCC2_destufed;
      int decrement=2;
      //BCC2 De-stuffing
      if (linkLayer.frame[frameIndex-3] != 0x7D)
          BCC2_destufed = linkLayer.frame[frameIndex-2];
      else{
        BCC2_destufed = linkLayer.frame[frameIndex-2] ^ 0x20;
        decrement=3;
      }

      // Byte De-stuffing
      for (int i = 4; i < frameIndex - decrement; i++)
      {
        if (linkLayer.frame[i] != 0x7D)
          (*buffer)[*buffersize] = linkLayer.frame[i];
        else{
          (*buffer)[*buffersize] = linkLayer.frame[i+1] ^ 0x20;
          i++;
        }
        (*buffersize)++;
      }
```

```
      *buffer = (unsigned char *) realloc(*buffer, (*buffersize));

      // Formação de BCC2
      unsigned char BCC2 = (*buffer)[0];
      for (int i = 1; i < *buffersize; i++)
        BCC2 = BCC2 ^ (*buffer)[i];

      if (BCC2_destufed==BCC2){
        linkLayer.sequenceNumber = linkLayer.sequenceNumber ^ 1;
        state_machine.state = DONE;
      }
      else{
        log_error("processBCC_OK() - BCC2 received with Errors");
        return -1;
      }
    }
  }
  return 0;
}
/** @}*/
```

logs.h

```
/** \addtogroup Logs
 *  @{
 */
#ifndef LOGS_HEADER
#define LOGS_HEADER

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


#define RESET   "\033[0m"
#define BLACK   "\033[30m"      /* Black */
#define RED     "\033[31m"      /* Red */
#define GREEN   "\033[32m"      /* Green */
#define YELLOW  "\033[33m"      /* Yellow */
#define BLUE    "\033[34m"      /* Blue */
#define MAGENTA "\033[35m"      /* Magenta */
#define CYAN    "\033[36m"      /* Cyan */
#define WHITE   "\033[37m"      /* White */
#define BOLDBLACK   "\033[1m\033[30m"      /* Bold Black */
#define BOLDRED     "\033[1m\033[31m"      /* Bold Red */
#define BOLDGREEN   "\033[1m\033[32m"      /* Bold Green */
```

```
#define BOLDYELLOW  "\033[1m\033[33m"      /* Bold Yellow */
#define BOLDBLUE    "\033[1m\033[34m"      /* Bold Blue */
#define BOLDMAGENTA "\033[1m\033[35m"      /* Bold Magenta */
#define BOLDCYAN    "\033[1m\033[36m"      /* Bold Cyan */
#define BOLDWHITE   "\033[1m\033[37m"      /* Bold White */


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

/**
 * @brief Outputs a message in red
 *
 * @param arr message String
 */
void log_error(char *arr);

/**
 * @brief Outputs a message in yellow
 *
 * @param arr message String
 */
void log_caution(char *arr);

/**
 * @brief Outputs a message in green
 *
 * @param arr message String
 */
void log_success(char *arr);

/**
 * @brief Outputs a message in white
 *
 * @param arr message String
 */
void log_message(char *arr);

/**
 * @brief Outputs a message in the form "number - message"
 *
 * @param arr message String
 * @param numb number
 */
void log_message_number(char *arr, int numb);

/**
```

```
 * @brief Outputs a message in the form "Content: 0xHH"
 *
 * @param a Byte
 */
void log_hexa(unsigned char a);

/**
 * @brief Clears progress bar from terminal
 *
 */
void clearProgressBar();

/**
 * @brief Prints progress bar on terminal
 *
 * @param progress nbytes already sent
 * @param total number of bytes to send
 */
void printProgressBar(int progress, int total);

/**
 * @brief starts counting time
 *
 */
void startTime(struct timespec *start_time);

/**
 * @brief starts counting time
 *
 */
void log_elapsedTime(char * message, struct timespec start_time);

void log_bitsPerSecond(double nbytes, struct timespec start_time);

#endif // LOGS_HEADER
/** @}*/
```

logs.c

```
/** \addtogroup Logs
 *  @{
 */
#define _POSIX_C_SOURCE 199309L
#include "logs.h"

void log_error(char *arr){
  char msg[255];
  sprintf(msg, RED "%s\n" RESET, arr);
```

```
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_caution(char *arr){
  char msg[255];
  sprintf(msg, YELLOW "%s\n" RESET, arr);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_success(char *arr){
  char msg[255];
  sprintf(msg, GREEN "%s\n" RESET, arr);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_message(char *arr){
  char msg[255];
  sprintf(msg, "%s\n" , arr);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_message_number(char *arr, int numb){
  char msg[255];
  sprintf(msg, "%d - %s" , numb,  arr);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_hexa(unsigned char a){
  char msg[255];
  sprintf(msg, "Content: %#4.2x\n" , a);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void clearProgressBar() {
    int i;
    for (i = 0; i < NUM_BACKSPACES+1; ++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);
}

void startTime(struct timespec *start_time){
  if(clock_gettime(CLOCK_MONOTONIC, start_time) == -1){perror("startTime");}
}

void log_elapsedTime(char * message, struct timespec start_time){
  struct timespec current_time;
  if(clock_gettime(CLOCK_MONOTONIC, &current_time) == -1){perror("elapsedTime");}
  double elapsed = (current_time.tv_sec-start_time.tv_sec)*1000+((current_time.tv_nsec-
start_time.tv_nsec)/10e6);

  char msg[255];
  sprintf(msg,"%s %f ms\n", message,elapsed);
  write(STDOUT_FILENO, msg, strlen(msg));
}

void log_bitsPerSecond(double nbytes, struct timespec start_time){
  struct timespec current_time;
  if(clock_gettime(CLOCK_MONOTONIC, &current_time) == -1){perror("elapsedTime");}
  double elapsed = (current_time.tv_sec-start_time.tv_sec)*1000+((current_time.tv_nsec-
start_time.tv_nsec)/10e6);

  double bits = (nbytes * 8) / (elapsed/1000);

  char msg[255];
  sprintf(msg,"%s %f \n", "[R] Bits per second: ", bits);
  write(STDOUT_FILENO, msg, strlen(msg));
}
/** @}*/
```

macros.h

```
/** \addtogroup Macros
 *  @{
 */
#ifndef MACROS_HEADER
#define MACROS_HEADER

/*-----------Variables-----------*/

#define BAUDRATE            B50
#define TIME_OUT            3
#define ATTEMPTS            5
#define MAX_SIZE            1024 // Tamanho máximo em bytes dos dados da trama I ===
Tamanho máximo do pacote de dados
```

```
// entre 24-inf

#define PROBABILITY_BCC2      0 //percentage
#define PROBABILITY_BCC1      0 //percentage
#define T_PROP_DELAY          0

/**
 * @brief Enum com Valores para a Máquina de Estados de SET-UA
 */

#define SU_FRAME_SIZE         5  ///< tamanho em bytes das tramas de Supervisão e Não Numeradas

#define FLAG                  0b01111110  ///< (0x7E) flag de inicio e fim

#define A_ER                  0b00000011  ///< (0x03) Campo de Endereço (A) de commandos do Emissor, resposta do Receptor
#define A_RE                  0b00000001  ///< (0x01) Campo de Endereço (A) de commandos do Receptor, resposta do Emissor

#define C_SET                 0b00000011 ///< (0x03) Campo de Controlo - SET (set up)
#define C_DISC                0b00001011 ///< (0x0B) Campo de Controlo - DISC (disconnect)
#define C_UA                  0b00000111 ///< (0x07) Campo de Controlo - UA (Unnumbered Acknowledgement)
#define C_RR(r)               ((0b00000101) ^ (r) << (7)) ///< (0x05 OU 0x85) Campo de Controlo - RR (receiver ready / positive ACK))
#define C_REJ(r)              ((0b00000001) ^ (r) << (7)) ///< (0x01 OU 0x81) Campo de Controlo - REJ (reject / negative ACK))
#define C_I(r)                ((0b01000000) & (r) << (6)) ///< (0x00 0x40) Campo de Controlo - Tramas I

#define BCC(a,c)              (a ^ c) ///< XOR entre a e c


#define FALSE                 0
#define TRUE                  1

// valores de type usados em application.c
#define TRANSMITTER           1
#define RECEIVER              0

/*-------package control--------*/

//C flag
#define DATA                  0x01
#define START                 0x02
#define END                   0x03

//T flag
```

```
#define T_SIZE              0x00
#define T_NAME               0x01


#define MAX_SIZE_AFT_STUFF    2*MAX_SIZE
#endif // MACROS_HEADER
/** @}*/
```

Anexo II - Fragmentos de código

ApplicationLayer

```
struct applicationLayer{
  off_t filesize; // Size of file in bytes
  char filename[255];  // String with file name
  char destinationArg[255];  // String with path to destination message received
  char filenameArg[255]; //String with filename passed by programs arguments
  int type; // TRANSMITTER | RECEIVER
  int gate; // /dev/ttySx | gate is x
};
```

LinkLayer

```
struct linkLayer {
  char port[20]; /* Device /dev/ttySx, x = 0, 1*/
  int baudRate; /*Transmission Speed*/
  unsigned char sequenceNumber;   /*Frame Sequence Number: 0x00, 0x01*/
  unsigned int timeout; /*Alarm Timeout: x s*/
  unsigned int numTransmissions; /*Number of tries in case of failure*/
  unsigned char frame[MAX_SIZE_AFT_STUFF]; /*Frame with double the size of MAX_SIZE
to accomodate for byte_stuffing increase*/
  unsigned int status; /*TRANSMITTER | RECEIVER*/
};
```

stateMachineParams

```
enum stateMachineType {Supervision, Write, Read};   // Type of State Machine
enum stateMachineState {Start, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DONE};  // State of
the State Machine

/**
 * @brief List of Parameters to Personalize the State Machine
 */
struct stateMachineParams {
  unsigned char control; // Control Byte that is supposed to be received
  unsigned char address; // Address Byte that is supposed to be received
```

```
  enum stateMachineState state;
  enum stateMachineType type;
};

struct stateMachineParams state_machine; // Instance of the stateMachineParams struct
```

stateMachine

```
int stateMachine(unsigned char byte, unsigned char **buf, int *size){
  if (state_machine.type == Read) linkLayer.frame[frameIndex] = byte;

  switch (state_machine.state)
  {
  case Start:
    return processStart(byte);
    break;
  case FLAG_RCV:
    return processFLAG_RCV(byte);
    break;
  case A_RCV:
    return processA_RCV(byte);
    break;
  case C_RCV:
    return processC_RCV(byte);
    break;
  case BCC_OK:
    return processBCC_OK(byte, buf, size);
    break;
  case DONE:
    return 0;
    break;
  default:
    log_error("stateMachine() - unknown state value");
    return -1;
    break;
  }
  return 0;
}
```

transmitter_SET

```
int transmitter_SET(int fd){
  unsigned char buf[SU_FRAME_SIZE] = {FLAG, A_ER, C_SET, BCC(A_ER, C_SET),
FLAG};

  stateMachineSetUp(C_UA, A_ER, Start, Supervision);

  writeCycle(trans_SET, fd, buf, SU_FRAME_SIZE);
```

```
  if(failed == TRUE){
    log_error("transmitter_SET() - Failed all attempts");
    return -1;
  }
  return fd;
}
```

receiver_UA

```
int receiver_UA(int fd){
  stateMachineSetUp(C_SET, A_ER, Start, Supervision);


  if (readingCycle(openR, fd, NULL, NULL, NULL) < 0){
    log_error("receiver_UA() - Error on reading Cycle");
    return -1;
  }

  unsigned char replyBuf[SU_FRAME_SIZE] = {FLAG, A_ER, C_UA, BCC(A_ER, C_UA),
FLAG};

  int res = write(fd,replyBuf,SU_FRAME_SIZE); //+1 para enviar o \0
  if (res == -1) {
    log_error("receiver_UA() - Failed writing UA to buffer.");
    return -1;}

  return fd;
}
```

transmitter_DISC_UA

```
int transmitter_DISC_UA(int fd){
  unsigned char buf[SU_FRAME_SIZE] = {FLAG, A_ER, C_DISC, BCC(A_ER, C_DISC),
FLAG};

  stateMachineSetUp(C_DISC, A_RE, Start, Supervision);

  writeCycle(trans_DISC_UA, fd, buf, SU_FRAME_SIZE);

  if(failed == TRUE){
    log_error("transmitter_DISC_UA() - Failed all attempts");
    return -1;
  }

  unsigned char buf1[SU_FRAME_SIZE] = {FLAG, A_RE, C_UA, BCC(A_RE, C_UA),
FLAG};
```

```
  int res = write(fd,buf1,SU_FRAME_SIZE);
  if (res == -1) {
    log_error("transmitter_DISC_UA() - Failed writing UA to buffer.");
    return -1;
  }

  return fd;
}
```

receiver_DISC_UA

```
int receiver_DISC_UA(int fd){
  stateMachineSetUp(C_DISC, A_ER, Start, Supervision);

  /* parse DISC*/
  readingCycle(closeDISC, fd, NULL, NULL, NULL);

  unsigned char replyBuf[SU_FRAME_SIZE] = {FLAG, A_RE, C_DISC, BCC(A_RE,
C_DISC), FLAG};

  int res = write(fd,replyBuf,SU_FRAME_SIZE);
  if (res == -1) {
    log_error("receiver_DISC_UA() - Failed writing DISC to buffer.");
    return -1;
  }

  stateMachineSetUp(C_UA, A_RE, Start, Supervision);

  alarm(linkLayer.timeout+1); /* waits a limited time for UA response from Transmitter */
  /* parse UA*/
  if (readingCycle(closeDISC, fd, NULL, NULL, NULL) < 0){
    log_error("receiver_DISC_UA() - error on reading Cycle");
    return -1;
  }


  alarm(0);
  return fd;
}
```

Anexo III - Dados

Os gráficos seguintes apresentam os resultados obtidos nos vários testes realizados utilizando ssh para nos conectarmos aos computadores do laboratório.

Os dois primeiros gráficos apresentam os resultados correspondentes aos testes realizados às probabilidades de erro geradas individualmente, utilizando o **pinguim.gif (10.5 kb)**. Apesar do aumento da probabilidade de erro, não é garantido que serão gerados mais erros em relação à probabilidade de erro anterior. Na verdade, para obtermos valores de eficiência mais verossímeis, seria necessário realizar milhares de testes para cada probabilidade e, de seguida, fazer uma média destes testes, devido ao fator aleatório que a geração de erros tem.

Os resultados são os esperados, uma vez que a eficiência é facilmente afetada com o aumento da probabilidade de erro e número de retransmissões.

Tabela 1 - Dados recolhidos para o gráfico da Eficiência (S) dependente da prob. de erros detectados no BCC2.

Prob. BCC1	Prob. BCC2	Atraso de prop. (s)	Baudrate (C) bits/s	Nº de Retransmissões	Tempo total (ms)	Bit rate recebido (R) bits/s	Tamanho trama (L) bytes	Eficiência (S)
0%	0%	0	38400	0	2992.946	29316.915	1024	76,346%
0%	10%	0	38400	0	2993.448	29311.985	1024	76,333%
0%	20%	0	38400	1	5993.545	14639.743	1024	38,124%
0%	25%	0	38400	2	89922.434	9756.448	1024	25,407%
0%	30%	0	38400	7	23993.468	3656.994	1024	9,523%
0%	40%	0	38400	7	23993.493	3656.991	1024	9,523%
0%	50%	0	38400	8	26993.471	3250.563	1024	8,465%

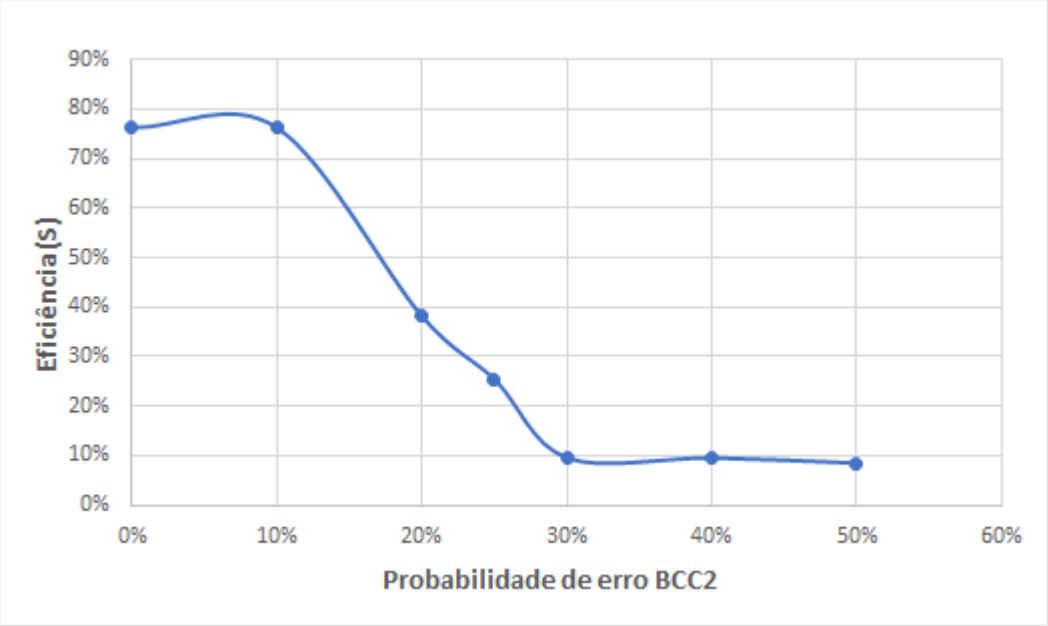


Figura 1 - Gráfico da Eficiência (S) dependente da probabilidade de erros detectados no BCC2.

Tabela 2 - Dados recolhidos para o gráfico da Eficiência (S) dependente da prob. de erros detectados no BCC1.

Prob. BCC1	Prob. BCC2	Atraso de prop. (s)	Baudrate (C) bits/s	Nº de Retransmissões	Tempo total (ms)	Bit rate recebido (R) bits/s	Tamanho trama (L) bytes	Eficiência (S)
0%	0%	0	38400	0	2992.946	29316.915	1024	76,346%
10%	0%	0	38400	0	2993.453	29311.943	1024	76,333%
20%	0%	0	38400	1	5093.414	17226.942	1024	44,861%
25%	0%	0	38400	2	8993.420	9756.463	1024	25,407%
30%	0%	0	38400	7	23993.470	3656.994	1024	9,523%
40%	0%	0	38400	7	23993.499	3656.99	1024	9,523%
50%	0%	0	38400	8	26993.425	3250.569	1024	8,465%

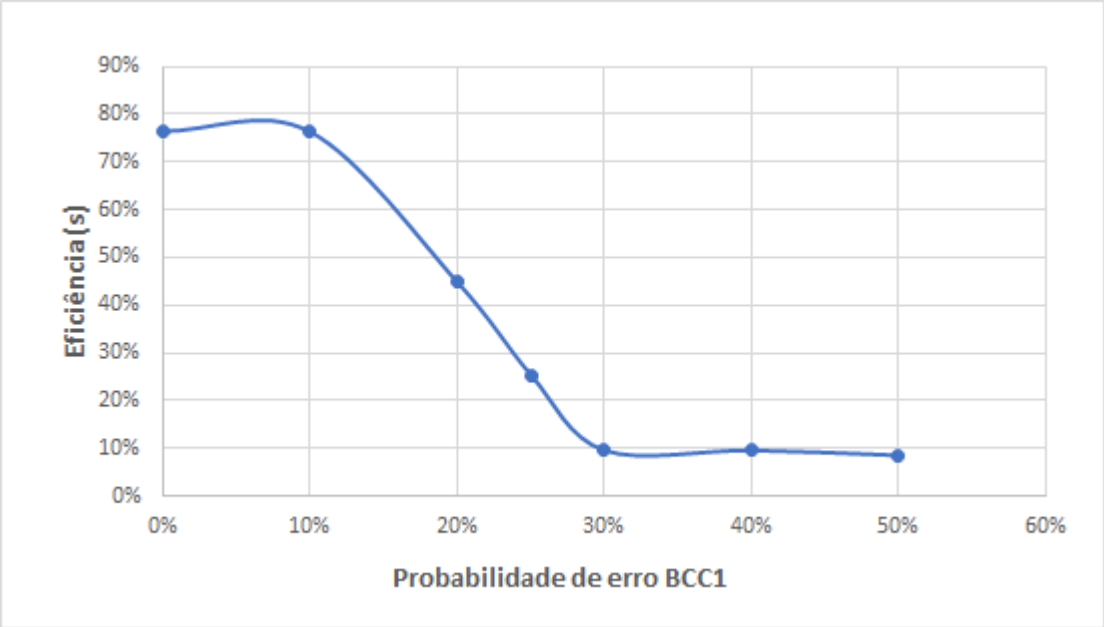


Figura 2 - Gráfico da Eficiência (S) dependente da probabilidade de erros detectados no BCC1.

Na tabela seguinte encontram-se as variações de eficiência de acordo com um atraso (em segundos) no processamento de cada trama recebida.

Prob. BCC1	Prob. BCC2	Atraso de prop. (s)	Baudrate (C) bits/s	Nº de Retransmissões	Tempo total (ms)	Bit rate recebido (R) bits/s	Tamanho trama (L) bytes	Eficiência (S)
0%	0%	1	38400	0	15993.443	5486.247	1024	14,287%
0%	0%	2	38400	0	28993.455	3026.337	1024	7,881%

O gráfico seguinte corresponde à variação do tamanho máximo da trama I.

Uma vez que o tamanho do pinguim.gif é de apenas 10.5 kb, o tempo de transferência é bastante reduzido pelo que não nos permitiu retirar dados conclusivos, porque a variação do tempo de transferência de bits por segundo era praticamente nula com a alteração do tamanho máximo da trama. Assim, decidimos usar para esta variação a **img.jpeg (60.1 kb)**.

Tabela 3 - Dados recolhidos para o gráfico da Eficiência (S) dependente da .variação do tamanho da trama I.

Prob. BCC1	Prob. BCC2	Atraso de prop. (s)	Baudrate (C) bits/s	Nº de Retransmissões	Tempo total (ms)	Bit rate recebido (R) bits/s	Tamanho trama (L) bytes	Eficiência (S)
0%	0%	0	38400	0	16030.978	30725.008	2048	80,013%
0%	0%	0	38400	0	16049.979	30688.635	1024	79,918%
0%	0%	0	38400	0	16991.495	28988.145	512	75,490%
0%	0%	0	38400	0	17069.533	28855.619	256	75,145%
0%	0%	0	38400	0	19029.635	25883.415	128	67,405%

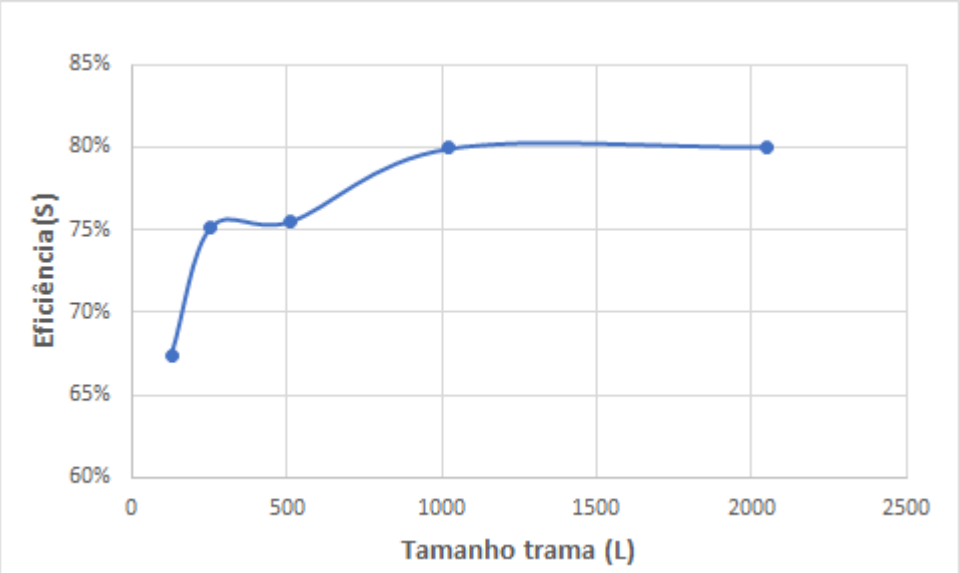


Figura 3 - Gráfico da Eficiência (S) dependente da variação do tamanho da trama I.

Os valores estão de acordo com o esperado, dado que quanto maior o número de bytes da trama mais eficiente é o envio, uma vez que resulta numa menor perda de tempo a fazer parsing de respostas de acknowledgment, e a construir tramas I com as devidas flags e bytes de address e controlo.

Tal como nos dados anteriores, na variação do Baudrate também foi usada a **img.jpeg (60.1 kb)**, que nos permitiu resultados mais conclusivos em relação ao pinguim.gif onde a eficiência praticamente não variou.

Tabela 4 - Dados recolhidos para o gráfico da Eficiência (S) dependente da variação do Baudrate.

Prob. BCC1	Prob. BCC2	Atraso de prop. (s)	Baudrate (C) bits/s	Nº de Retransmissões	Tempo total (ms)	Bit rate recebido (R) bits/s	Tamanho trama (L) bytes	Eficiência (S)
0%	0%	0	115200	0	5953.715	82730.163	1024	71,814%
0%	0%	0	38400	0	16950.004	29059.104	1024	75,675%
0%	0%	0	19200	0	32997.650	14926.880	1024	77,744%
0%	0%	0	9600	0	65992.636	7463.741	1024	77,747%
0%	0%	0	4800	0	131982.327	3731.954	1024	77,749%

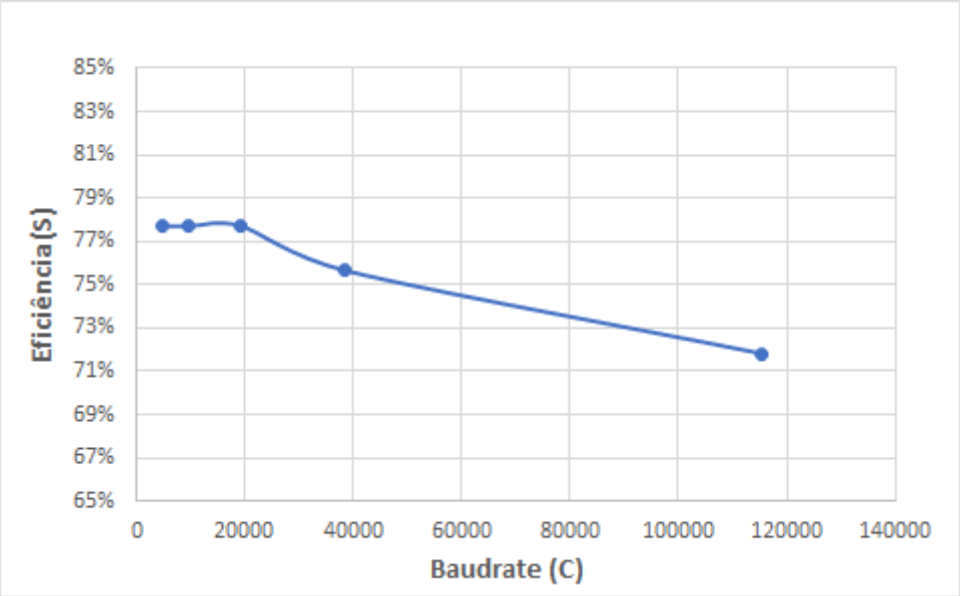


Figura 4 - Gráfico da Eficiência (S) dependente da variação do Baudrate.

Conclui-se com estes dados que a eficiência é ligeiramente melhor com Baudrates menores, o que faz sentido dado que em Baudrates maiores o tempo gasto em processamento de tramas e envio/receção de mensagens de *acknowledgment* tem mais influência, porque nesse espaço de tempo poderíamos estar a enviar um grande número de bytes, mas em vez disso estamos a processar a trama enviada.

Em Baurates menores o tempo gasto a processar a trama tem menos influência na eficiência porque não se passariam tantos bytes nesse espaço de tempo devido ao limite do Baudrate, caso o envio fosse contínuo (i.e. não existir mensagens de *acknowledgment* nem *stuffing*/*de-stuffing*).