U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

Redes de Computadores

3ºAno MIEIC

10 de novembro de 2020

Rafael Cristino, up201806680@fe.up.pt

Rita Peixoto, up201806257@fe.up.pt

Grupo 3, Turma 2

Índice

Sumário

Este trabalho laboratorial, desenvolvido no âmbito da unidade curricular de Redes de Computadores, teve como objetivo a implementação de um protocolo de dados e teste deste com uma aplicação simples de transferência de ficheiros entre dois computadores através de uma porta série.

Pode-se concluir que o protocolo implementado cumpriu os seus objetivos, com uma eficiência, para os valores predefinidos de *FRAME_SIZE* (= 512 bytes) e de *BAUDRATE* (= B38400), de 78%, como é possível analisar nas estatísticas geradas, sendo capaz de transferir ficheiros entre dois computadores sem perda de dados.

Introdução

Neste trabalho laboratorial, o objetivo foi o desenvolvimento de uma aplicação de transferência de ficheiros entre dois computadores através de uma porta série, com a implementação de um protocolo de dados, ambos seguindo uma especificação descrita no guião do trabalho, em que se garantia a integridade dos dados transferidos, inclusive nos casos de eventuais falhas de ligação ou interferências.

Este relatório pretende examinar a implementação deste trabalho laboratorial, descrevendo os seus diferentes componentes e como estes se complementam. O relatório organiza-se na seguinte estrutura: **arquitetura**, descrição dos blocos funcionais e interfaces; **estrutura do código**, descrição e análise das *API*s, principais estruturas de dados, principais funções e a sua relação com a arquitetura; **casos de uso principais**, identificação dos principais casos de uso e das sequências de chamadas de funções; **protocolo da ligação lógica,** identificação dos principais aspetos funcionais com descrição da estratégia de implementação destes aspetos; **protocolo de aplicação,** identificação dos principais aspetos funcionais com descrição da estratégia de implementação destes aspetos; **validação**, descrição dos testes efetuados e apresentação quantificada dos seus resultados; **eficiência do protocolo de ligação de dados**, caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido; **conclusões,** síntese da informação apresentada nas secções anteriores e reflexão acerca dos objetivos de aprendizagem alcançados; **anexos,** código fonte, tabelas, entre outros.

Arquitetura

Existem dois grandes blocos funcionais neste trabalho laboratorial: o protocolo de ligação de dados e a aplicação, que representam uma arquitetura em camadas fundamentada no princípio de independência entre camadas, sendo responsáveis por realizar tarefas encapsuladas.

Protocolo de ligação de dados

O protocolo de ligação de dados é responsável pela transmissão e receção de tramas de informação (I), Supervisão (S) e não numeradas (U), fiável, entre dois sistemas ligados por um cabo série. Esta camada configura a porta série, estabelece e termina a ligação, garante o sincronismo e a transparência das tramas, e possui mecanismo de tratamento de erros.

O protocolo de dados inclui a *link layer*, que é uma estrutura de dados que permite uma melhor encapsulação de informações críticas pertencentes ao protocolo, como por exemplo o tamanho da trama. O acesso (escrita/leitura) aos conteúdos desta estrutura é completamente encapsulado.

Aplicação

A aplicação está integrada no que concerne a interface com o utilizador. Esta pode tomar o papel de transmissor, ***transmitter***, encarregue do envio do ficheiro, que é dividido em pacotes de dados, especificado ou não (neste caso é utilizado um por omissão), ou o papel de recetor, ***receiver***, encarregue da receção dos pacotes de dados e da sua concatenação.

Estes dois blocos estão interligados de forma unilateral, no sentido em que a aplicação utiliza as funções da interface com o protocolo de ligação de dados de forma a conseguir estabelecer conexão, enviar pacotes de dados e controlo, no caso do transmissor, e receber, no caso do recetor, e terminar a conexão.

Estrutura do código

O código encontra-se dividido entre diferentes ficheiros para melhor organização, sendo que a cada ficheiro de código de fonte (.c) corresponde um ficheiro *header* (.h). Na pasta principal está também presente um ficheiro *Makefile* de forma a facilitar a compilação.

Protocolo

Interface protocolo-aplicação (ficheiros protocol/iPA.c e protocol/iPA.h)

- **llopen** - estabelece a ligação entre o transmissor e o recetor.
- **llread** - recebe os pacotes e preenche um buffer com os dados recebidos.
- **llwrite** - transmites os pacotes de dados e controlo
- **llclose** - termina a ligação entre o transmissor e o recetor.
- **llset** - altera a *baudrate* ou o tamanho das tramas. Uma função não especificada no guião de trabalho, mas que se mostrou útil para permitir à aplicação alterar estes parâmetros, o que possibilitou incluir esses argumentos na chamada do programa e facilitou a sua alteração para o cálculo da eficiência com variação de valores.

Link Layer (ficheiros protocol/linkLayer.c e protocol/linkLayer.h)

Nestes ficheiros estão descritas todas as funções necessárias para aceder à estrutura de dados *linkLayer*, bem como a sua definição, tal como descrita no anexo 1.

Assim como mencionado na secção 'Arquitetura', o acesso (leitura e escrita) a esta estrutura é completamente encapsulado. Para tal, é necessário recorrer às seguintes funções:

- Métodos *GET* para cada um dos seus parâmetros.
- Métodos *SET* para os parâmetros que os necessitam.
- **incrementSequenceNumber** - incrementa o número de sequência das tramas I, sendo *nextSequenceNumber* = (*prevSequenceNumber* + 1) % 2

Máquina de Estados (ficheiros protocol/stateMachine.c e protocol/stateMachine.h)

Para a interpretação dos conjuntos de bytes recebidos e a sua organização em tramas, foi implementada uma máquina de estados. A máquina de estados é responsável por, por cada byte recebido, interpretar o seu valor no contexto que o antecedeu e efetuar a operação de *destuffing*. A mesma máquina de estados é utilizada tanto para tramas de informação como para tramas de supervisão.

A operação de *destuffing* é da seguinte forma realizada na máquina de estados: 1) receção do octeto de escape (0x7d); 2) ignorar este byte; 3) avaliação do próximo byte tendo em conta o byte anterior, e a sua substituição pelo valor correto; 4) avaliação deste byte pela parte restante da máquina de estados como se fosse um byte normal.

- **checkState** - realiza as funções acima descritas.
- **checkDestuffedBCC** - calcula o BCC de um *buffer* de dados e compara-o com o BCC recebido.

Funções gerais (ficheiros protocol/protocol.c e protocol/protocol.h)

- **openConfigureSP** – abre e configura a porta série. Inclui uma chamada *atexit*, que faz com que a função *closeSP* seja executada sempre no final, para o caso em que o programa é terminado por uma chamada *exit ()*, ou seja, sem chamar *llclose*.
- **writeToSP** – escreve um *buffer* de dados para a porta série.
- **readFromSP** – lê uma trama da porta série. Retorna a ação a tomar ou erro. Essa ação pode ser RR ou REJ, que são interpretados diferentemente pelo recetor e transmissor. Por exemplo, REJ para o recetor significa enviar uma trama de supervisão REJ, enquanto que para o transmissor significa reenviar a trama anterior.
- **closeSP** – fecha a porta série. Inclui uma condição que verifica se o *file descriptor* fornecido é inválido, para o caso em que esta função já foi executada.
- **constructSupervisionMessage** – constrói uma trama de supervisão tendo em conta os argumentos.
- **constructInformationMessage** – constrói uma trama de informação tendo em conta os argumentos.
- **byteStuffing** – aplica *stuffing* à trama especificada nos argumentos.

Funções do recetor (ficheiros protocol/receiverFuncs.c e protocol/receiverFuncs.h)

- **receiverConnect** – executa a rotina de conexão do recetor (receção de SET e envio de UA).
- **receiverRead** – executa a rotina de leitura do recetor: chamada da função *readFromSP* e ação de acordo com o seu retorno.
- **receiverDisconnect** – executa a rotina de desconexão do recetor (receção de DISC, envio de DISC e receção de UA.

Funções do transmissor (ficheiros protocol/transmitterFuncs.c e protocol/transmitterFuncs.h)

- **stopAndWait** – implementa o mecanismo de stop&wait: quando uma trama é enviada é acionado um alarme com *TIMEOUT* segundos, se a confirmação de receção correta não chegar antes do tempo de o alarme terminar, então a trama é retransmitida. No entanto, esta retransmissão só pode acontecer no máximo *NUM_TRANSMISSIONS* vezes, após isso o programa termina com uma mensagem de erro no envio da mensagem.
- **logicConnectionFunction** – rotina de conexão lógica do transmissor (Envio de SET e receção de UA).
- **disconnectionFunction** – rotina de desconexão do transmissor (Envio de DISC, receção de DISC e envio de UA).
- **sendDataFunction** – função que se encarrega de enviar dados.
- **transmitterConnect, transmitterWrite e transmitterDisconnect** – encarregam-se de chamar a função *stopAndWait* com cada uma das três funções acima como argumento, respetivamente, e de preparar as tramas de forma a serem utilizadas por essas funções.

Constantes (ficheiro protocol/defines.h)

Contém definições de constantes úteis. Melhoria de leitura do código e facilidade de alteração dos valores.

Aplicação

Argumentos do programa (ficheiros application/application.c e application/application.h)

- **printUsage** – imprime a forma de uso dos argumentos da aplicação no terminal.
- **checkCmdArgs** – verifica se os argumentos são válidos e chama *llset*.

Máquina de estados (ficheiros application/stateMachine.c e application/stateMachine.h)

- **checkStateReception** – recebe o pacote e verifica os seus conteúdos, retornando o estado final, ou erro. Serve tanto para pacotes de controlo como de informação.

Recetor (ficheiros application/receiver.c e application/receiver.h)

- **receiver** – utiliza as funções da interface protocolo-aplicação no *flow* descrito na secção 'Casos de uso principais'. Verifica a validade do pacote recebido utilizando a função *checkStateReception* e age de acordo com o resultado. Escreve os pacotes de dados recebidos para o ficheiro de destino.

Transmissor (ficheiros application/transmitter.c e application/transmitter.h)

- **transmitter** – utiliza as funções da interface protocolo-aplicação no *flow* descrito na secção 'Casos de uso principais'. Envia o pacote de controlo *Start* com os tamanho e nome do ficheiro de destino. Lê o ficheiro a enviar e divide em pacotes. Quando enviar todos os pacotes do ficheiro, envia o pacote de controlo *End* com a repetição dos dados do ficheiro de destino.
- **constructControlPacket** – Constrói um pacote de controlo, recebendo como argumentos o buffer de destino (*ret*), o campo de controlo a incluir (*ctrl*), o nome de ficheiro a incluir e o tamanho desse ficheiro.
- **constructDataPacket** – Constrói um pacote de dados, recebendo como argumentos o buffer de destino (*ret*), os dados a incluir (*data*), o tamanho dos dados e o número de sequência da mensagem (módulo 255).

Constantes (ficheiro application/defines.h)

Contém definições de constantes úteis. Melhoria de leitura do código e facilidade de alteração dos valores.

Utilitários (pasta utilities/)

Descreve funções úteis tanto para o protocolo como para a aplicação, como por exemplo a função *isUnsignedNumber* que verifica que uma string é um número inteiro sem sinal.

Testes de eficiência (pasta tests/)

Descreve as funções que permitiram a realização dos testes de eficiência do protocolo.

Casos de uso principais

O principal caso de uso deste trabalho é a transferência de ficheiros entre dois computadores através de uma porta série, possível através da interface com o utilizador que foi desenvolvida.

O fluxo normal do uso da aplicação segue a seguinte ordem:

1. Compilação do programa utilizando o MakeFile incluído na pasta do código fonte, executando o comando make na pasta *src*.
2. Execução de duas instâncias da aplicação, uma em cada um dos computadores que se pretendem ligar, efetuando a escolha da porta série, seguida da escolha do papel a desempenhar pela instância da aplicação a ser executada, isto é, ou transmissor, ***TRANSMITTER***, ou recetor, ***RECEIVER***. No caso do papel de ser transmissor, é necessário referir qual o ficheiro a ser enviado e, opcionalmente, o nome com que esse ficheiro será guardado no lado do recetor que, de modo a obter o resultado esperado deve conter a mesma extensão.

   De modo a utilizar valores diferentes dos definidos por omissão, é possível especificar o tamanho da trama (*FRAME_SIZE*) e a capacidade da ligação (*BAUDRATE*) através dos argumentos na chamada de execução do programa, cujos valores devem ser iguais tanto no emissor como no recetor para permitir um bom funcionamento do programa. Um exemplo de chamada de execução da aplicação encontra-se descrito no anexo 2.

3. No caso do transmissor, é realizada a leitura do ficheiro e envio dos dados através da interface protocolo-aplicação; no caso do recetor, é realizada a receção dos dados, através da interface protocolo-aplicação e posterior escrita num novo ficheiro.

Encontra-se disponível um exemplo de execução da aplicação, no anexo 3.

O fluxo normal da utilização da interface protocolo-aplicação é o seguinte:

1. Chamada opcional a **llset** de forma a alterar os valores predefinidos da *BAUDRATE* ou do *FRAME_SIZE*.
2. Estabelecimento da conexão entre os dois computadores através da porta série com a chamada à função **llopen**.
3. Utilização da função **llread**, no caso do *receiver*, para ler mensagens recebidas ou da função **llwrite**, no lado do *transmitter*, para enviar mensagens.
4. Terminação da conexão entre os dois computadores com a chamada à função **llclose**.

Protocolo de ligação lógica

O protocolo de ligação lógica integra as seguintes componentes: o estabelecimento de ligação entre os dois computadores, o envio e receção de tramas e a terminação da conexão lógica.

Estabelecimento da ligação entre o transmissor e o recetor – llopen

```
int llopen (int porta, char * role); // role is TRANSMITTER | RECEIVER
```

Aqui é estabelecida a ligação entre o transmissor e o recetor, após a configuração e abertura da porta série (*openConfigureSP*). O transmissor envia uma trama de supervisão SET e aguarda a confirmação não numerada da correta receção desta trama por parte do recetor, que o faz enviando uma trama de supervisão UA.

Para efetuar esta conexão lógica, no lado do transmissor é utilizada a função *transmitterConnect*. No lado do recetor é utilizada a função *receiverConnect*.

Escrita das tramas - llwrite

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

Esta função é responsável pelo envio de tramas do tipo I, receção das respostas por parte do recetor e por agir consoante a resposta do recetor, recebendo como argumento os dados a enviar.

Começa por construir a trama de informação, fazendo o framing da mensagem, com o devido *stuffing*, se necessário, e cálculo do BCC2 (também com stuffing), e chama a função *transmitterWrite* para proceder ao envio

da mensagem. A função *transmitterWrite* recorre também à função *stopAndWait* para fazer o envio da trama, uma vez que utiliza o mesmo mecanismo de retransmissão referido anteriormente. A função *stopAndWait*, por sua vez, chama a função *sendDataFunction*, que envia a trama, lê a resposta do recetor e, recorrendo à mesma máquina de estados utilizada em *llopen*, interpreta-a, enviando de volta para a função *stopAndWait* a necessidade ou não de retransmissão.

Leitura das tramas - llread

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

A função *llread* trata de ler as tramas enviadas em *llwrite* e de responder, de acordo com o que o leu, com uma tramas de supervisão RR (acknowledgment) ou REJ (reject).

Recorre diretamente à chamada da função *receiverRead*, que começa por ler a mensagem recebida byte a byte (caracter a caracter), procedendo ao seu *destuffing* e validação na *state machine* integrada na função *readFromSP*. Dependendo da resposta de estado final retornado por essa *state machine*, constrói a trama de supervisão de resposta, de forma a notificar corretamente o transmissor acerca do estado da mensagem recebida, isto é, RR no caso de ser uma trama correta ou repetida, pedindo o envio da próxima e REJ no caso de haver necessidade de retransmissão e envia-a.

Terminação da conexão lógica - llclose

```
int llclose (int fd);
```

A desconexão dá-se com o envio de uma trama de supervisão DISC da parte do transmissor, resposta de uma trama de supervisão DISC da parte do recetor e termina com o transmissor a enviar uma mensagem de supervisão UA. No caso do transmissor, é feita a chamada da função *transmitterDisconnect* e no caso do recetor é utilizada a função *receiverDisconnect*.

Após a verificação deste protocolo, são repostas as configurações iniciais e fechada a ligação.

Protocolo de aplicação

O protocolo de aplicação implementado tem como objetivo a construção, envio e receção de pacotes, de controlo ou de dados, do nível de aplicação, de forma a transferir um ficheiro entre dois computadores, utilizando a interface protocolo-aplicação.

Tendo em conta este objetivo, foram desenvolvidas as seguintes funções:

- Construção e interpretação de pacotes de controlo
- Envio e receção de pacotes de controlo, *START* e *END*, que sinalizam respetivamente o início e o final da transferência e contêm informação sobre o ficheiro, como o tamanho em *bytes* e o nome do ficheiro de destino.
- Do lado do transmissor: leitura do ficheiro e divisão do seu conteúdo, que será encapsulado com um cabeçalho para serem enviados formando pacotes de dados.
- Do lado do recetor: escrita do ficheiro, através da concatenação dos fragmentos recebidos após o desencapsulamento dos pacotes.

Recetor

checkStateReception

```
enum checkReceptionState checkStateReception(char * buffer, int bufferSize, size_t *
fileSize, char ** fileName, int * dataAmount);
```

Esta função é responsável pela análise do pacote recebido, ou seja, interpretar os dados que o constituem. Para isso usa uma estrutura de máquina de estados, sendo a transição entre estados efetuada *byte* a *byte*. O enum de retorno está descrito no anexo 4.

Se o pacote recebido for um pacote de controlo *Start*, a função retorna o estado *START_RECEIVED*. Se for um pacote de controlo *End*, retorna o estado *END_RECEIVED*. Se for um pacote de dados, retorna o estado *DATA_FINISHED*. Se esta identificar um erro, retorna *ERROR*. Se não identificar nenhum erro, mas o estado final não for um dos mencionados anteriormente, retorna o estado atual.

receiver

```
void receiver (int serialPort);
```

Esta é a função principal do recetor. Este estabelece conexão chamando *llopen* e entra num *loop* de leitura utilizando *llread* para ler os pacotes. Tendo em conta os pacotes que recebe, a ordem de receção deve ser sempre um pacote de controlo *Start*, seguido de dados e terminando com um pacote *End*. Se os pacotes não forem lidos nesta ordem, dá erro.

Assim que é lido um pacote *Start*, a aplicação abre o ficheiro de destino para escrita com o nome de ficheiro recebido neste pacote. No caso em que já existe um ficheiro com esse nome, a aplicação procura encontrar um nome não utilizado, acrescentando um número à frente.

À medida que o recetor vai lendo os pacotes recebidos, vai utilizando a função *checkStateReception* para os analisar, e age de acordo com o seu retorno, procurando manter esta ordem de operações. Se receber um pacote inválido, dá erro. Os pacotes válidos são escritos para o ficheiro de destino.

No final, ao receber o pacote *End*, verifica se recebeu a quantidade de bytes especificada. Se não, dá erro. Se sim, então termina o *loop* de leitura e chama a função *llclose* de forma a terminar a ligação.

Transmissor

transmitter

```
void transmitter (int serialPort, char * fileToSend, char * destFile);
```

Esta é a função principal do transmissor. Este estabelece conexão chamando *llopen* e após sucesso, tenta abrir o ficheiro a enviar. Se o ficheiro não existir, dá erro. Caso contrário, obtém o seu tamanho e obtém um pacote *Start* chamando *constructControlPacket*, e envia-o utilizando *llwrite*.

Após iniciar a transmissão do ficheiro, entra num *loop* de leitura do ficheiro que se encontra no computador conjugado com o envio de pacotes de dados pela porta série, usando novamente *llwrite*, construídos a partir da informação lida utilizando *constructDataPacket*.

Após chegar ao *EOF* (*end of file)* do ficheiro que está a enviar, termina o envio do ficheiro e prepara um pacote de controlo *End*, novamente usando *constructControlPacket* e a função llwrite. Para terminar, chama a função *llclose*.

Validação

De forma a verificar a implementação deste trabalho laboratorial e a sua robustez, comprovando o funcionamento de acordo com o especificado, foram desenvolvidos alguns testes.

Testes com indução de erros nas tramas a enviar

Desenvolvimento de um módulo de testes que permitiu rapidamente testar o protocolo, independente da aplicação desenvolvida para demonstração.

Este módulo foi desenvolvido de forma a induzir erros nas tramas, sendo estes no cabeçalho ou nos dados, de forma a avaliar a deteção de erros no cabeçalho ou dados por parte do protocolo, bem como a remoção destes, através do reenvio de tramas (quer seja devido a *timeout* ou à receção de uma trama de controlo REJ, por parte do transmissor).

Para além disso, foi seguida a sugestão do guião do trabalho, que consiste em gerar de forma aleatória erros em tramas de informação, sendo que foram gerados erros tanto no cabeçalho como no campo de dados, ambos com probabilidades predefinidas e independentes, e gerar um atraso de propagação simulado induzido no final do processamento de cada trama.

Testes com introdução de interferências (ruído)

- *Simulando o desligar do cabo através do uso do ficheiro* "cable.c"*, fornecido pelos docentes.*
- *Desligando fisicamente o cabo no hardware dos laboratórios, tendo esta interrupção o intervalo de um ou dois TIME_OUT para obrigar à necessidade de retransmissão da trama.*
- *Utilizando materiais metálicos de forma a induzir ruído na ligação.*

Testes de variação de parâmetros

- *Testes com variação do tamanho do ficheiro transmitido com ficheiros*

Foram efetuados testes com ficheiros de: 11KiB, 106KiB, 239KiB, 493KiB, 7.54 MiB, sendo que este último apenas foi efetuado em ambiente de simulador.

- *Testes com variação do tamanho máximo das tramas de informação*

Os testes efetuados foram com os seguintes valores: 64, 128, 256, 512, 1024, entre outros, de forma experimental, como 2000 e 5000.

- *Testes com variação da capacidade de ligação (*"baudrate"*)*

O valor do campo *BAUDRATE*, localizado na *link layer* foi alterado de modo a permitir a inserção de interferências, uma vez que diminuir este valor faz com que se diminua a taxa a que os bits são transmitidos, sendo a transmissão mais lenta, tendo sido os valores utilizados foram 38400 e 9600.

Conclusão dos testes

Todos estes testes obtiveram um resultado positivo, sendo que o programa no estado atual age consoante esperado em todas essas ocasiões. Isto significa que o ficheiro é enviado de um computador para o outro, sem erros, com o mesmo número de bytes, e com uma visualização igual, independentemente de qual destes testes é realizado.

Eficiência do protocolo de ligação de dados

Os testes referidos no tópico anterior foram utilizados para avaliar a eficiência do programa. A eficiência (S) é calculada pela fórmula $S = R / C$, sendo R o débito observado e C a capacidade da ligação.

Desta forma, foram realizados os testes de forma a variar diferentes parâmetros, sendo que para as mesmas condições realizámos dois testes e calculámos a sua média, de modo a reduzir o desvio dos dados. As tabelas com todos os dados obtidos estão apresentadas no anexo 5, em maior detalhe. Os testes realizados permitiram elaborar os gráficos e as conclusões apresentadas de seguida.

Variação da capacidade da ligação (C)

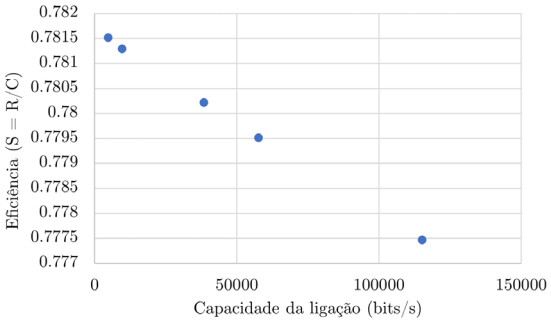


FIGURA 1: VARIAÇÃO DA EFICIÊNCIA COM A CAPACIDADE DE LIGAÇÃO

Com este gráfico, é possível concluir que com o aumento da capacidade de ligação a eficiência diminui linearmente, embora seja uma diminuição muito reduzida. Com estas amostras a diminuição mais acentuada é de cerca de 0.4% (entre 78.15% e 77.75%).

Variação do tamanho máximo das tramas de informação

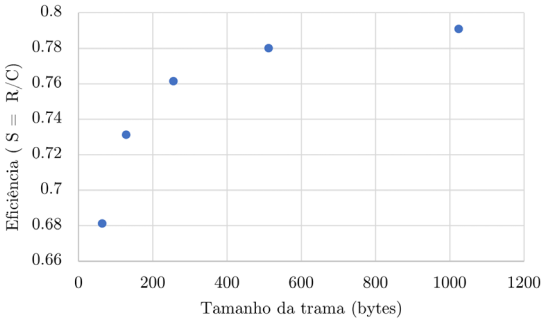
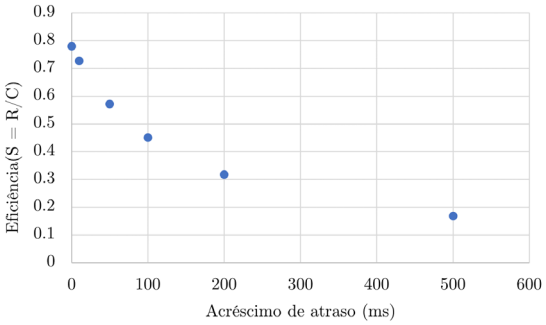


FIGURA 2: VARIAÇÃO DA EFICIÊNCIA COM O TAMANHO DAS TRAMAS I

Neste gráfico, verifica-se que quanto maior o tamanho da trama I, maior a eficiência. Isto deve-se a, quanto maior for a trama, menor é a quantidade de dados enviada pois é necessário menor *overhead* para a encapsulação dos dados. No entanto, apesar de aumentar com o tamanho da trama, a partir dos 500 bytes estabiliza, apresentando valores com pouca variância para tamanhos superiores.

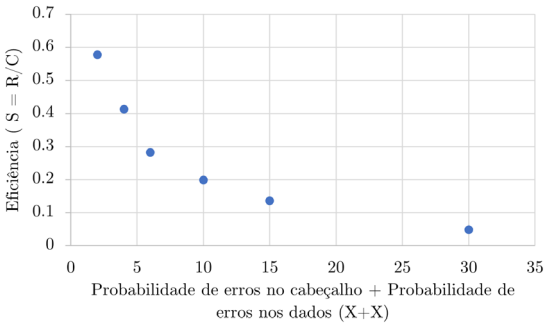
Variação do acréscimo de atraso no tempo de propagação



É possível comprovar que quanto maior o acréscimo de atraso no tempo de propagação, menor a eficiência do protocolo. Estas duas variáveis têm uma relação muito forte, sendo que quanto mais tempo demorar a propagação de dados, menor será o débito observado, o que faz com que a eficiência diminua.

FIGURA 3: VARIAÇÃO DA EFICIÊNCIA COM O ACRÉSCIMO DO ATRASO NO TEMPO DE PROPAGAÇÃO

Variação do FER

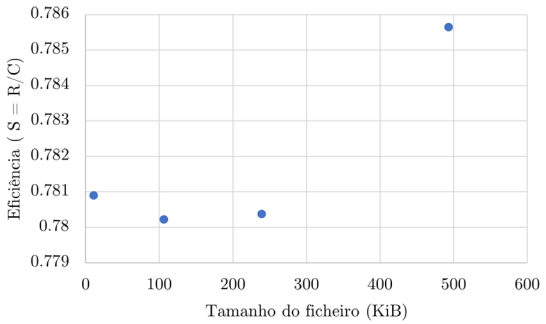


Neste gráfico, verifica-se que a ocorrência de erros impacta muito a eficiência do programa. Adicionalmente, tem-se em conta que os erros de cabeçalho impactam muito mais a eficiência do que os erros dados, pois um erro no cabeçalho não origina resposta, ou seja, o transmissor aguarda pelo *timeout* para retransmitir a trama, enquanto que um erro nos dados origina uma resposta REJ, que faz com que o transmissor reenvie a trama assim que a recebe. Este facto torna-se mais evidente nas tabelas do anexo 5.

Nota: o eixo x representa a probabilidade X+X; por exemplo, X = 10, representa a probabilidade 10+10.

FIGURA 4: VARIAÇÃO DA EFICIÊNCIA COM A VARIAÇÃO DO FER

Variação do tamanho do ficheiro



Por análise deste gráfico é possível retirar que a eficiência não parece ter uma relação muito definida com o tamanho do ficheiro. Nesta amostra varia entre 78% e 78.6%, tendo um ligeiro aumento com o ficheiro de 500KiB.

FIGURA 5: VARIAÇÃO DA EFICIÊNCIA COM A VARIAÇÃO DO TAMANHO DO FICHEIRO

Conclusões

O trabalho laboratorial permitiu compreender com clareza as aplicações práticas dos conceitos abordados nas aulas teóricas desta unidade curricular. De entre destes conceitos, é relevante salientar: *framing* e posteriores *stuffing* e *destuffing*, a deteção de erros a partir da introdução de redundância com verificação de paridade e mecanismos de retransmissão, sendo neste caso específico implementado o mecanismo de *stop&wait*. Por fim, destacamos, acima de tudo, a necessidade de garantir independência entre camadas, sendo que as camadas desconhecem os detalhes de implementação uma da outra, tendo apenas o conhecimento da forma de acesso ao serviço.

A eficiência do protocolo implementado ronda os 78%, para os valores predefinidos de *FRAME_SIZE* (= 512 bytes) e de *BAUDRATE* (= B38400).

Para terminar, realça-se que o trabalho foi concluído com sucesso, cumprindo todos os objetivos propostos: a aplicação e protocolo de ligação de dados são capazes de enviar um ficheiro entre duas máquinas conectadas através de uma porta série, conseguindo lidar com eventuais erros e interferências mantendo a integridade dos dados transferidos.

Anexos

Anexo 1 – Definição da estrutura de dados *linkLayer*

```
struct linkLayer {
    speed_t BAUDRATE;
    unsigned FRAME_SIZE,
        MAX_DATA_PACKET_SIZE,
        MAX_FRAME_BUFFER_SIZE,
        TIMEOUT,
        NUM_TRANSMISSIONS,
        nextSequenceNumber;
    int fd;  // serial port file descriptor
    struct termios oldtio;
};
```

Anexo 2 – Exemplo de chamadas de execução da aplicação

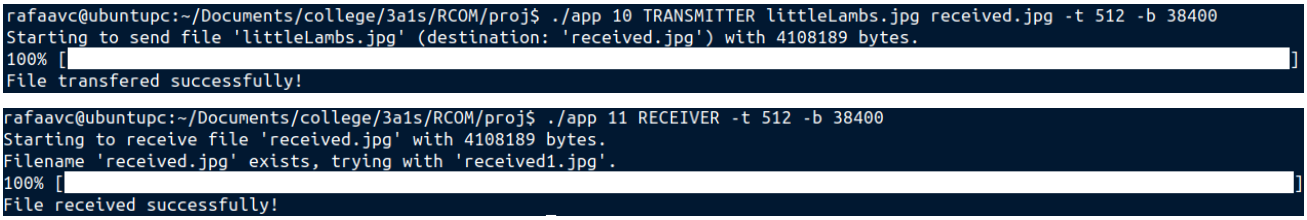
Computador 1

```
./app 10 RECEIVER [-t FRAME_SIZE] [-b BAUDRATE]
```

Computador 2

```
./app 11 TRANSMITTER littleLambs.jpg received.jpg [-t FRAME_SIZE] [-b BAUDRATE]
```

Anexo 3 – Exemplo de execução da aplicação



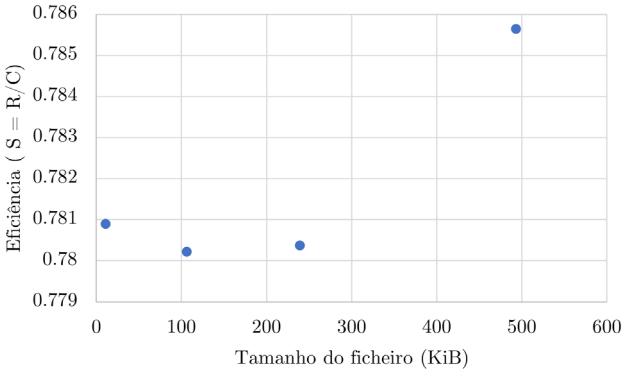
Anexo 4 – Enums

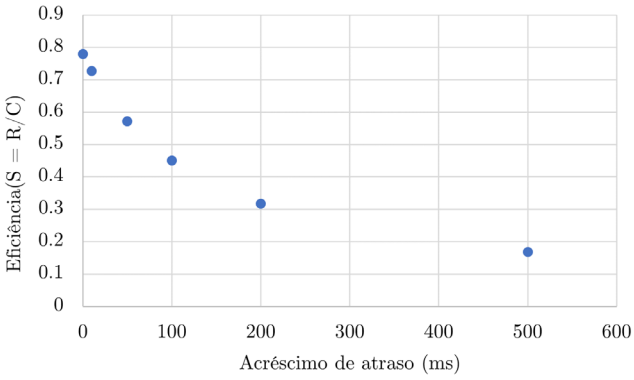
```
enum checkReceptionState {CTRL, T, L, V, RECEIVING_DATA, DATA_N, DATA_L1, DATA_L2,
DATA_FINISHED, END_RECEIVED, START_RECEIVED, ERROR};
enum stateMachine { START, FLAG_RCV, A_RCV, C_RCV, BCC_HEAD_OK, DATA, DATA_OK,
BCC_DATA_OK, DONE_S_U, DONE_I };
enum destuffingState { DESTUFF_OK, DESTUFF_WAITING, DESTUFF_VIEWING };
enum checkStateRET {
    StateOK,
    IGNORE_CHAR,
    HEAD_INVALID,
    DATA_INVALID
};
enum readFromSPRet { RR, REJ, SAVE, READ_ERROR, STOPPED_OR_SU };
```

Anexo 5 – Tabelas de dados obtidas experimentalmente

Variação do tamanho do ficheiro

Tamanho do ficheiro (KiB)	Tempo (s)	Débito Recebido (R) (bits/s)	Baudrate (C) (bits/s)	Eficiência (S) (R/C)	Tempo médio (T)	Débito médio (R)	Eficiência média (S)
11	3.002971	29986.30597	38400	0.780893	3.002943	29986.5849	0.7809005
	3.002915	29986.86382		0.780908			
106	29.762306	29960.58125		0.780223	29.7622525	29960.63545	0.7802245
	29.762199	29960.68964		0.780226			
239	67.012222	29966.47386		0.780377	67.0123545	29966.41461	0.780377
	67.012487	29966.35536		0.780377			
493	136.548109	30169.00081		0.785651	136.548782	30168.85205	0.785647
	136.549455	30168.7033		0.785643			



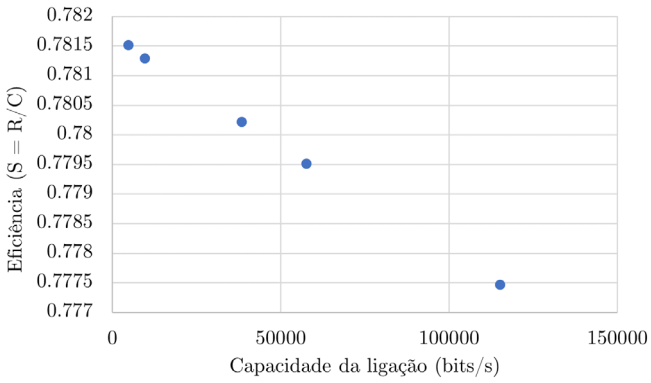


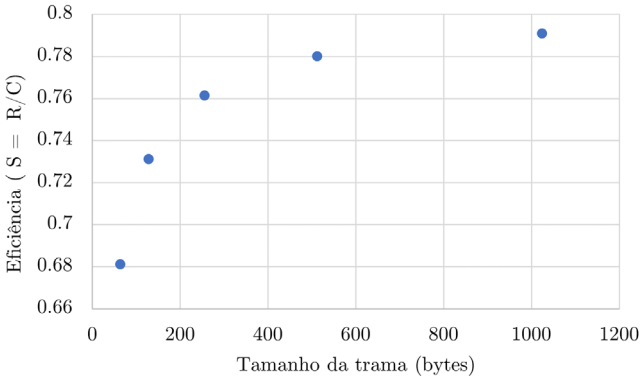
Variação do acréscimo de atraso

Acréscimo de atraso (ms)	Tempo (s)	Débito Recebido (R) (bits/s)	Baudrate (C) (bits/s)	Eficiência (S) (R/C)	Tempo médio (T)	Débito médio (R)	Eficiência média (S)
0	29.762932	29959.95182	38400	0.780207	29.762772	29960.11283	0.780211
	29.762612	29960.27383		0.780215			
10	31.946253	27912.38179		0.726885	31.946393	27912.2595	0.726882
	31.946533	27912.13722		0.726879			
50	40.626258	21948.7604		0.571582	40.6267355	21948.50226	0.5715755
	40.627213	21948.24413		0.571569			
100	51.476352	17322.43968		0.451105	51.4763225	17322.44951	0.4511055
	51.476293	17322.45934		0.451106			
200	73.176854	12185.49239		0.317331	73.1764635	12185.55745	0.3173325
	73.176073	12185.62252		0.317334			
500	138.276477	6448.645618		0.167933	138.2762255	6448.657362	0.1679335
	138.275974	6448.669105		0.167934			

Variação da capacidade de ligação

Capacidade da ligação (C) (bits/s)	Tempo (s)	Débito Recebido (R) bits/s	Eficiência (S) (R/C)	Tempo médio (T)	Débito médio (R)	Eficiência média (S)
4800	237.70489	3751.273278	0.781515	237.7052085	3751.268248	0.781514
	237.705527	3751.263217	0.781513			
9600	118.887122	7500.358173	0.781287	118.886758	7500.381144	0.7812895
	118.886394	7500.404114	0.781292			
38400	29.76247	29960.41623	0.780219	29.762528	29960.35782	0.7802175
	29.762586	29960.29942	0.780216			
57600	19.859668	44899.84383	0.779511	19.85959	44900.02024	0.779514
	19.859512	44900.19666	0.779517			
115200	9.955959	89564.04536	0.777466	9.955974	89563.91311	0.7774645
	9.955989	89563.78087	0.777463			



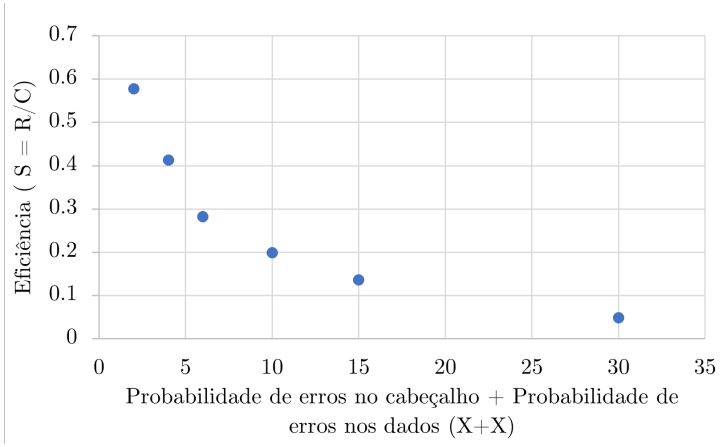


Variação do tamanho da trama

Tamanho de cada trama (bytes)	Tempo (s)	Débito Recebido (R) bits/s	Baudrate (C) (bits/s)	Eficiência (S) (R/C)	Tempo médio (T)	Débito médio (R)	Eficiência média (S)
64	39.581067	26156.74795	38400	0.681165	39.580655	26157.0204	0.6811725
	39.580243	26157.29285		0.68118			
128	33.765295	28076.63931		0.731162	33.7651385	28076.76913	0.7311655
	33.764982	28076.89894		0.731169			
256	31.10722	29243.88642		0.76156	31.106855	29244.2292	0.7615685
	31.10649	29244.57198		0.761577			
512	29.762294	29960.59364		0.780224	29.7627945	29960.08975	0.780211
	29.763295	29959.58586		0.780198			
1024	29.072697	30370.48793		0.790898	29.0729635	30370.20949	0.790891
	29.07323	30369.93105		0.790884			

Variação do FER

Probabilidade de erro no cabeçalho + Probabilidade de erro no campo de dados	Erros no cabeçalho	Erros no campo de dados	Tempo (s)	Débito Recebido (R) bits/s	Eficiência (S) (R/C)	Baudrate (C) (bits/s)	Média de Erros no Cabeçalho	Média de Erros nos Dados	Tempo médio (T)	Débito médio (R)	Eficiência média (S)
2+2	4	3	42.177259	21141.62971	0.550563	38400	3.333333333	4.333333333	40.358602	22174.64758	0.577464667
	2	9	36.998916	24100.59813	0.62762						
	4	1	41.899631	21281.7149	0.554211						
4+4	11	5	63.449774	14053.57254	0.365978		9	6.666666667	57.58783767	15868.60349	0.413244667
	11	8	63.724022	13993.09036	0.364403						
	5	7	45.589717	19559.14757	0.509353						
6+6	19	14	85.693383	10405.65759	0.270981		17.66666667	13	82.55369133	10832.53892	0.282097333
	16	12	76.412838	11669.45279	0.303892						
	18	13	85.554853	10422.50637	0.271419						
10+10	24	13	103.552281	8611.07056	0.224247		29	19.66666667	117.468942	7654.597785	0.199338667
	29	22	119.792219	7443.688799	0.193846						
	34	24	129.062326	6909.033996	0.179923						
15+15	43	43	161.326227	5527.284788	0.14394		46.33333333	44.66666667	172.741818	5218.081854	0.135887667
	55	53	198.911402	4482.880276	0.116742						
	41	38	157.987825	5644.080497	0.146981						
30+30	143	117	465.378496	1916.066187	0.049898		149	106	482.1074965	1851.808915	0.0482245
	155	95	498.836497	1787.551643	0.046551						



Anexo 6 – Código fonte

Protocolo (pasta *protocol*)

iPA.c

```
#include "receiverFuncs.h"
#include "iPA.h"

char role;
static bool usedLLSet = false;

int llopen(int porta, char * r){
    if (strcmp(r, "RECEIVER") == 0) role = RECEIVER;
    else if (strcmp(r, "TRANSMITTER") == 0) role = TRANSMITTER;
    else {
        printError("Invalid role given to llopen\n");
        return -1;
    }

    srand(time(NULL));

    if (!usedLLSet) createLinkLayer();

    char portString[12];

    sprintf(portString, "/dev/ttyS%d", porta);

    int fd = openConfigureSP(portString);

    if (fd == -1) return -1;

    if(role == RECEIVER){
        if (!receiverConnect()) return -1; //establishing connection with TRANSMITTER
    }
    else if (role == TRANSMITTER){
        if (signal(SIGALRM, alarmHandler) < 0) {  // Installs the handler for the alarm
interruption
            perror("Alarm handler wasn't installed");
            exit(EXIT_FAILURE);
        }
        siginterrupt(SIGALRM, true);  // SIGALARM interrupts system calls (returns -1
and errno is se to EINTR)
        if (!transmitterConnect()) return -1; //establishing connection with RECEIVER
    }
    else return -1;

    return fd;
}

int llwrite(int fd, char * buffer, int length){
    char * msg = myMalloc(getMaxFrameBufferSize());
    bzero(msg, getMaxFrameBufferSize());
    size_t s = length;

    constructInformationMessage(msg, buffer, &s);

    int res = transmitterWrite(msg,s);
```

```
    free(msg);
    if (res < 0) return -1;// in case of error

    return res;
}

int llread(int fd, char * buffer){
    return receiverRead(buffer);
}

int llclose(int fd){
    if(role == TRANSMITTER){
        if(!transmitterDisconnect()) //disconnecting in the TRANSMITTER side
            return -1;
    }
    else if(role == RECEIVER){ //disconnecting in the RECEIVER side
       if(receiverDisconnect() == -1)//if there was an error sending/receiving the
disconnection messages
            return -1;;
    }
    else return -1;

    sleep(1);

    if(closeSP() != 0){
        return -1;
    }

    return 0;
}

void llset(int baudrateArg, int frameSizeArg) {
    usedLLSet = true;
    createLinkLayer();
    if (baudrateArg != -1) {
        speed_t BAUDRATE;
        switch(baudrateArg) {
            case 4800:
                BAUDRATE = B4800;
                break;
            case 9600:
                BAUDRATE = B9600;
                break;
            case 19200:
                BAUDRATE = B19200;
                break;
            case 38400:
                BAUDRATE = B38400;
                break;
            case 57600:
                BAUDRATE = B57600;
                break;
            case 115200:
                BAUDRATE = B115200;
                break;
            case 230400:
                BAUDRATE = B230400;
```

```
                break;
            default:
                printError("The specified baudrate isn't valid. Using default.
Available:\n4800\n9600\n19200\n38400\n57600\n115200\n230400\n");
                break;
        }
        setBaudrate(BAUDRATE);
    }

    if (frameSizeArg != -1) {
        setFrameSize(frameSizeArg);
    }
}
```

linkLayer.c

```
#include "defines.h"  // incluidos aqui de forma à aplicação não ter acesso aos defines
do protocolo (porque o linkLayer é incluído pelo iPA.h)
#include "linkLayer.h"

static bool createdLayer = false;

struct linkLayer { // put here so other files don't know anything about this struct
    speed_t BAUDRATE;
    unsigned FRAME_SIZE,
        MAX_DATA_PACKET_SIZE,
        MAX_FRAME_BUFFER_SIZE,
        TIMEOUT,
        NUM_TRANSMISSIONS,
        nextSequenceNumber;
    int fd;  // serial port file descriptor
    struct termios oldtio;
};

static struct linkLayer llayer;


speed_t getBaudrate() {
    return llayer.BAUDRATE;
}
unsigned getFrameSize() {
    return llayer.FRAME_SIZE;
}
unsigned getMaxDataPacketSize() {
    return llayer.MAX_DATA_PACKET_SIZE;
}
unsigned getMaxFrameBufferSize() {
    return llayer.MAX_FRAME_BUFFER_SIZE;
}
unsigned getTimeout() {
    return llayer.TIMEOUT;
}
unsigned getNumTransmissions() {
    return llayer.NUM_TRANSMISSIONS;
}
struct termios * getOldTIO() {
    return &llayer.oldtio;
```

```
}
unsigned getNextSequenceNumber() {
    return llayer.nextSequenceNumber;
}
int getFD() {
    return llayer.fd;
}


void incrementNextSequenceNumber() {
    llayer.nextSequenceNumber = (llayer.nextSequenceNumber+1)%2;
}

void setFD(int fd) {
    llayer.fd = fd;
}

void setTimeout(unsigned timeout) {
    llayer.TIMEOUT = timeout;
    //printf("Set timeout = %u\n", timeout);
}

void setNumTransmissions(unsigned numTransmissions) {
    llayer.NUM_TRANSMISSIONS = numTransmissions;
    //printf("Set num transmissions = %u\n", numTransmissions);
}

void setBaudrate(speed_t baudrate) {
    llayer.BAUDRATE = baudrate;
    //printf("Set baudrate = %u\n", baudrate);
}

void setFrameSize(unsigned frameSize) {
    llayer.FRAME_SIZE = frameSize;
    llayer.MAX_DATA_PACKET_SIZE = llayer.FRAME_SIZE - SUPERVISION_MSG_SIZE - 1;
    llayer.MAX_FRAME_BUFFER_SIZE = llayer.FRAME_SIZE * 2;
    //printf("Set frame size = %u\n", frameSize);
}

void createLinkLayer() {
    if (createdLayer) return;
    createdLayer = true;
    setFrameSize(512);
    setBaudrate(B38400);
    setNumTransmissions(5);
    setTimeout(3);
    setFD(-1);
    llayer.nextSequenceNumber = 0;
}
```

protocol.c

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

```
#ifdef DEBUG_STATE_MACHINE
static char * stateNames[] = { "Start", "FLAG_RCV", "A_RCV", "C_RCV", "BCC_HEAD_OK",
"DATA", "DATA_OK", "BCC_DATA_OK", "DONE_S_U", "DONE_I" };
#endif

volatile bool stopAndWaitFlag = false;

int openConfigureSP(char* port) {
    /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
    */

    struct termios newtio;

    setFD(open(port, O_RDWR | O_NOCTTY));

    if (getFD() < 0) { perror(port); return -1; }

    if (tcgetattr(getFD(), getOldTIO()) == -1) { /* save current port settings */
        perror("Error on tcgetattr");
        return -1;
    }

    atexit((void *)&closeSP); //resets configuration when program terminates with
exit(EXIT_FAILURE)

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

    /* set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;
    /*
        VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
        leitura do(s) pr�ximo(s) caracter(es)
        t = TIME * 0.1s
    */
    newtio.c_cc[VTIME]    = 0;   // if read only blocks for VTIME*0.1 seconds, or until
a character is received
    newtio.c_cc[VMIN]     = 1;   // blocks until a character is read

    tcflush(getFD(), TCIOFLUSH);   // discards from the queue data received but not read
and data written but not transmitted

    if (tcsetattr(getFD(), TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
        return -1;
    }

    debugMessage("[SP] OPENED AND CONFIGURED");

    return getFD();
}
```

```c
size_t writeToSP(char* message, size_t messageSize) {
    return write(getFD(), message, messageSize*sizeof(message[0]));
}

enum readFromSPRet readFromSP(char * buf, enum stateMachine *state, ssize_t *
stringSize, char addressField, char controlField) {
    char reading;
    int counter = 0;

    #ifdef DEBUG
    static int sendREJ = 0;   // Simulates sending REJ (if > 0, sends REJ)
    #endif

    bool STOP = false;
    *state = START;

    char bcc[2];

    static int pS = 1;
    //reads from the serial port
    while(!STOP) {
        int readRet = read(getFD(), &reading, 1);

        if (stopAndWaitFlag) STOP = true; // if the alarm interrupts
        if (readRet < 0 && errno != EINTR) {
            perror("Unsuccessful read");
            return READ_ERROR;
        }
        else if (readRet == 0) continue; // if didn't read anything

        // if read is successful
        switch (checkState(state, bcc, &reading, buf, addressField, controlField)) {
            case HEAD_INVALID: // IGNORE ALL, can start reading next one
                #ifdef DEBUG_STATE_MACHINE
                debugMessage("HEAD_INVALID");
                #endif
                counter = 0;
                continue;
            case DATA_INVALID: // Evaluate head && act accordingly
                #ifdef DEBUG_STATE_MACHINE
                debugMessage("DATA_INVALID");
                #endif
                STOP = true;
                int s = getS(buf[CTRL_IDX]);
                if(s == pS){  //send RR, confirm reception
                    //printf("S == pS (RR): %d == %d\n", s, pS);
                    return RR;
                }
                else{  //send REJ, needs retransmission
                    //printf("S != pS (REJ): %d == %d\n", s, pS);
                    return REJ;
                }
            case IGNORE_CHAR:
                #ifdef DEBUG_STATE_MACHINE
                debugMessage("IGNORE_CHAR");
                #endif
                continue;
```

```
            case StateOK:   // adds the byte to the buffer
                #ifdef DEBUG_STATE_MACHINE
                debugMessage("StateOK");
                #endif
            default: break;

        }

        #ifdef DEBUG_STATE_MACHINE
        printf("state after checkstate: %s, counter: %d\n", stateNames[*state],
counter);
        #endif
        buf[counter++] = reading;
        if (isAcceptanceState(state)) break;
    }

    *stringSize = counter;
    if (isI(state)) {
        int s = getS(buf[CTRL_IDX]);
        //printf("Received s: %d, pS: %d\n", s, pS);

        if(s == pS){  // s == previousS -> send RR & discard
            //printf("COUNTER/SIZE S == PS: %d\n", counter);
            return RR;
        }
        else{ // send RR and accept data
            //printf("COUNTER/SIZE ELSE: %d\n", counter);
            //printf("COUNTER/SIZE: %d\n", counter);
            #ifdef DEBUG   // simulates REJ
            if (sendREJ > 0) {
                sendREJ--;
                return REJ;
            }
            #endif
            pS = s; // updates s
            return SAVE;
        }
    } else if (isSU(state) && (isRR(buf[CTRL_IDX])|| isREJ(buf[CTRL_IDX]))) {   //
Received ack / nack  -> this is the transmitter instance
        int r = getR(buf[CTRL_IDX]);
        //printf("Received r: %d, nextS: %d\n", r, getNextSequenceNumber());

        if (r != (getNextSequenceNumber())) { // Repeated control -> R needs to be
always equal to the S of the next information msg
            // RR (ignore)
            return RR;
        } else { // if it isREJ or RR && new control (r == nextS)
            // act upon the result
            return isRR(buf[CTRL_IDX]) ? RR : REJ;
        }
    }

    // This means it was stopped by the stop and wait alarm OR it is either UA, SET or
DISC
    return STOPPED_OR_SU;
}
```

```
int closeSP() {
    if (fcntl(getFD(), F_GETFD) != 0 && errno == EBADF) return -1; // verifies if fd is
valid
    sleep(1);
    if (tcsetattr(getFD(), TCSANOW, getOldTIO()) == -1) {
      perror("Error on tcsetattr");
      return -1;
    }

    if(close(getFD()) != 0){
        perror("Error on close");
        return -1;
    }
    return 0;
}

void constructSupervisionMessage(char * ret, char addr, char ctrl) {
    ret[BEGIN_FLAG_IDX] = MSG_FLAG;
    ret[ADDR_IDX] = addr;
    ret[CTRL_IDX] = ctrl;
    ret[BCC1_IDX] = BCC(addr, ctrl);
    ret[BCC1_IDX+1] = MSG_FLAG;
}

void constructInformationMessage(char* ret ,char* data, size_t * dataSize) {//ret size =
6 + dataSize OR NOT (if stuffing actually replaces bytes)
    if(*dataSize == 0) {
        printError("Tried to construct an information message without any data.\n");
        return;
    }
    char bcc = 0;

    ret[BEGIN_FLAG_IDX] = MSG_FLAG;
    ret[ADDR_IDX] = ADDR_SENT_EM;
    ret[CTRL_IDX] = CTRL_S(getNextSequenceNumber());
    ret[BCC1_IDX] = BCC(ret[2], ret[1]);

    for (size_t i = 0; i < *dataSize; i++) {  // calculates the data bcc (between all
data bytes)
        bcc = BCC(bcc, data[i]);
        ret[BCC1_IDX+1+i] = data[i];
    }
    ret[BCC1_IDX+*dataSize+1] = bcc;
    ret[BCC1_IDX+*dataSize+2] = MSG_FLAG;

    *dataSize += 6;
    incrementNextSequenceNumber();

    byteStuffing(ret, dataSize); // dataSize is updated in the byteStuffing function
}

void byteStuffing(char * ret, size_t * retSize){
    char * buf = myMalloc(getMaxFrameBufferSize()*sizeof(char));
    buf[0] = MSG_FLAG;
    unsigned bufferIdx = 1;

    for (unsigned i = bufferIdx; i < (*retSize)-1; i++){
```

```
        if(ret[i] == 0x7E){// case it is equal to the flag pattern
            buf[bufferIdx++] = 0x7D;
            buf[bufferIdx++] = 0x5E;
        }
        else if (ret[i] == 0x7D){// case it is equal to the escape pattern
            buf[bufferIdx++] = 0x7D;
            buf[bufferIdx++] = 0x5D;
        } else {
            buf[bufferIdx++] = ret[i];
        }
    }
    buf[bufferIdx] = MSG_FLAG;
    *retSize = bufferIdx+1;
    memcpy(ret, buf, *retSize);
    free(buf);
}
```

receiverFuncs.c

```
#include "receiverFuncs.h"

static bool receivedDataFlag = false;  // flag that says whether at least a data message
has been received

size_t receiverRead(char * buffer) {
    ssize_t size;
    enum stateMachine state;
    enum readFromSPRet res;

    while (true) {
        char * ret = myMalloc(getMaxFrameBufferSize());


        res = readFromSP(ret, &state, &size, ADDR_SENT_EM, ANY_VALUE);

        if (res == READ_ERROR) return -1;
        // readFromSP only returns acceptance state
        char *buf = (char*)myMalloc(SUPERVISION_MSG_SIZE*sizeof(char));

        // checks if it read from a valid information message
        // OR has an RR or REJ to send (SAVE only happens if msg is valid, so it is
implicit)
        // otherwise an error occurred
        if (isI(&state) || res == RR || res == REJ) {
            //printf("Received I\n");
            #ifdef EFFICIENCY_TEST
            delayGenerator();
            #endif

            if (!receivedDataFlag) receivedDataFlag = true;
            int s = getS(ret[CTRL_IDX]);
            if(res == REJ) {
                //debugMessage("Sending REJ");
                constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_REJ((s+1)%2));
                writeToSP(buf, SUPERVISION_MSG_SIZE);
                free(buf);
                return 0;
```

```
            }
            else { // SAVE or RR
                //debugMessage("Sending RR");
                constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_RR((s+1)%2));
                writeToSP(buf, SUPERVISION_MSG_SIZE);
                free(buf);
                if(res == SAVE) {
                    //debugMessage("Saving Message");
                    //printf("Saving message, size: %ld\n", size);
                    size_t dataLength = size - 6;
                    memcpy(buffer, &ret[BCC1_IDX+1], dataLength);
                    free(ret);
                    return dataLength;
                } else { // case of repeated packet
                    return 0;
                }
            }
            free(ret);
            // (REJ == STOPPED_OR_SU)
        } else { // in this case it means that we received an SU msg
            if (!receivedDataFlag && isSU(&state) && ret[CTRL_IDX] == CTRL_SET) { // it
means that the ack of the SET in llopen (sent by the receiver) didn't reach the
transmitter
                constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_UA);
                writeToSP(buf, SUPERVISION_MSG_SIZE);
                free(buf);
            } else {
                printError("Received invalid data (SU message) while running llread.
Protocol reader state: %u\n", state);
                free(ret);
                return -1;
            }
        }
    }
}


bool receiverConnect() {
    ssize_t size;
    enum stateMachine state;
    char *buf = (char*)myMalloc(SUPERVISION_MSG_SIZE*sizeof(char));

    while (true) {
        char * ret = myMalloc(getMaxFrameBufferSize());
        if (readFromSP(ret, &state, &size, ADDR_SENT_EM, CTRL_SET) == READ_ERROR) break;
        debugMessage("RECEIVED SET");
        free(ret);
        constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_UA);
        writeToSP(buf, SUPERVISION_MSG_SIZE);
        free(buf);
        return true;
    }
    return false;
}


size_t receiverDisconnect() {
```

```
    ssize_t size;
    enum stateMachine state;
    char *buf = (char*)myMalloc(SUPERVISION_MSG_SIZE*sizeof(char));

    while (true) {
        char * ret = myMalloc(getMaxFrameBufferSize());
        if (readFromSP(ret, &state, &size, ADDR_SENT_EM, CTRL_DISC) == READ_ERROR)
return -1;

        if (isAcceptanceState(&state)) {
            debugMessage("RECEIVED DISC");
            constructSupervisionMessage(buf, ADDR_SENT_RCV, CTRL_DISC);
            writeToSP(buf, SUPERVISION_MSG_SIZE);
            free(buf);
        }
        else{
            free(buf);
            return -1;
        }

        if (readFromSP(ret, &state, &size, ADDR_SENT_RCV, CTRL_UA) == READ_ERROR) return
-1;

        free(ret);
        if (isAcceptanceState(&state)) {
            debugMessage("RECEIVED UA");
            return 0;
        }
    }
    return -1;

}
```

stateMachine.c

```
#include "stateMachine.h"

static char prevByte;
extern char role;

bool isAcceptanceState(enum stateMachine *state) {
    return *state == DONE_I || *state == DONE_S_U;
}

bool isI(enum stateMachine *state) {
    return *state == DONE_I;
}

bool isSU(enum stateMachine *state) {
    return *state == DONE_S_U;
}

bool isRR(unsigned char ctrl) {
    return ctrl == 0x85 || ctrl == 0x05;  // depending on the value of r
}

bool isREJ(unsigned char ctrl) {
```

```
    return ctrl == 0x81 || ctrl == 0x01;  // depending on the value of r
}

int getS(unsigned char ctrl) {
    return ctrl >> 6;
}

int getR(unsigned char ctrl) {
    return ctrl >> 7;
}

bool checkDestuffedBCC(char* buf, char bcc, size_t dataCount){
    #ifdef EFFICIENCY_TEST
    if(role ==  RECEIVER) generateDataError(buf,dataCount);
    #endif


    char aux = 0x0;
    if (dataCount == 0) return false;

    // Can be done like this because dataCount is the number of data bytes received and
that will always be true

    #ifdef DEBUG_STATE_MACHINE
    printf("BCC calculated with: ");
    #endif
    for(size_t i = 4; i < dataCount + 4; i++){
        #ifdef DEBUG_STATE_MACHINE
        printf("%x ", (unsigned char)buf[i]);
        #endif
        aux ^= (unsigned char)buf[i];
    }

    #ifdef DEBUG_STATE_MACHINE
    printf("\nCalculated BCC: %x, real BCC: %x\n", aux, bcc);
    #endif

    if(aux == bcc) return true;
    else return false;

}

bool receivedMessageFlag(char * byte, enum destuffingState destuffing) {
    return *byte == MSG_FLAG && destuffing == DESTUFF_OK;
}

void goBackToStart(enum stateMachine * state, enum destuffingState * destuffing) {
    *state = START;
    *destuffing = DESTUFF_OK;
}

void goBackToFLAG_RCV(enum stateMachine * state, enum destuffingState * destuffing) {
    *state = FLAG_RCV;
    *destuffing = DESTUFF_OK;
}
```

```
enum checkStateRET checkState(enum stateMachine *state, char * bcc, char * byte,
char*buf, char addressField, char controlField) {
    static unsigned dataCount = 4;
    static bool dataBCC = 0;
    static enum destuffingState destuffing = DESTUFF_OK;

    // Destuffs the message while it's reading it
    switch (destuffing) {
        case DESTUFF_VIEWING:
            destuffing = DESTUFF_OK;
            // break is missing intentionally
        case DESTUFF_OK:
            if (*byte == 0x7D){
                destuffing = DESTUFF_WAITING;
                return IGNORE_CHAR;
            }
            break;
        case DESTUFF_WAITING:
            if (*byte == 0x5E){
                *byte = 0x7E;
            } else if (*byte == 0x5D){
                *byte = 0x7D;
            } else {
                printf("Error while destuffing in checkstate!\n");
                //exit(EXIT_FAILURE);
            }
            destuffing = DESTUFF_VIEWING;
            break;
        default:
            break;
    }

    #ifdef DEBUG_STATE_MACHINE
    printf("\nSM: byte received: %x; currentState: %s\n", *byte, stateNames[*state]);
    #endif

    switch (*state){
    case START:
        // Advances from Start when flag id received
        if(receivedMessageFlag(byte, destuffing)){
            *state = FLAG_RCV;
        } else {
            return HEAD_INVALID;
        }
        break;

    case FLAG_RCV:
        // Only advances when a valid address field is received
        if(!receivedMessageFlag(byte, destuffing)
            && (*byte == addressField || addressField == ANY_VALUE)) {
            *state = A_RCV;
            bcc[0] = *byte;
        }
        else if(!receivedMessageFlag(byte, destuffing)){ // If it receives a flag, it
doesn't change state
            goBackToStart(state, &destuffing);
            return HEAD_INVALID;
```

```
        } else {
            return IGNORE_CHAR; // Ignores the flag in case it had received one
imediately before
        }
        break;

    case A_RCV:
        // Only advances when a valid control field is received
        if (!receivedMessageFlag(byte, destuffing) &&
            (*byte == controlField || controlField == ANY_VALUE)) {
            *state = C_RCV;
            bcc[1] = *byte;
        }
        else { // INVALID
            receivedMessageFlag(byte, destuffing) ?
                goBackToFLAG_RCV(state, &destuffing) :
                goBackToStart(state, &destuffing);
            return HEAD_INVALID;
        }
        break;

    case C_RCV:
        // Only advances if BCC is correct
        #ifdef EFFICIENCY_TEST
            if(role == RECEIVER && (prevByte == 0x00 || prevByte == 0x40))
generateHeadError(bcc);  // if it's the receiver and is receiving an I message
        #endif
        if (*byte == BCC(bcc[0], bcc[1]) && !receivedMessageFlag(byte, destuffing)) {
            *state = BCC_HEAD_OK;
        }
        else { // INVALID
            receivedMessageFlag(byte, destuffing) ?
                goBackToFLAG_RCV(state, &destuffing) :
                goBackToStart(state, &destuffing);
            return HEAD_INVALID;
        }
        break;

    case BCC_HEAD_OK:
        if (receivedMessageFlag(byte, destuffing)) {
            *state = DONE_S_U; // S and U
        }
        else {
            *state = DATA; // I
            dataCount = 1;
        }
        break;

    case DATA:
        if(!receivedMessageFlag(byte, destuffing)){
            dataCount++;
            if (dataCount >= getMaxDataPacketSize()) { *state = DATA_OK;}
            // if it doesn't receive a flag, then it only looks for the bcc when the max
data packet length has been used
            break;
        } else {  // receives a flag; the bcc is the previous byte
            dataCount--; // discounts the BCC from the data count
```

```
            dataBCC = checkDestuffedBCC(buf, prevByte, dataCount);
            if (dataBCC){
                *state = DONE_I;
            }
            else {  // BCC is wrong
                goBackToFLAG_RCV(state, &destuffing);
                return DATA_INVALID;
            }
        }
        break;
    case DATA_OK:
        // verify BCC
        if (!receivedMessageFlag(byte, destuffing)) {
            //printf("ERROR. datacount: %d, byte: %x, prevByte: %x\n", dataCount,
(unsigned char) *byte, (unsigned char) prevByte);

            dataBCC = checkDestuffedBCC(buf, *byte, dataCount);

            if(dataBCC){
                *state = BCC_DATA_OK;
            }
            else{
                goBackToStart(state, &destuffing);
                return DATA_INVALID;
            }
        } else {   // if it receives a flag when dataOk, it means that the bcc is the
previous byte
            dataBCC = checkDestuffedBCC(buf, prevByte, dataCount);
            dataCount--;  // removes the bcc from the dataCount
            if (dataBCC){
                *state = DONE_I;
            }
            else {  // BCC is wrong
                goBackToFLAG_RCV(state, &destuffing);
                return DATA_INVALID;
            }
        }
        break;

    case BCC_DATA_OK:
        if (receivedMessageFlag(byte, destuffing)) {  // receives the end flag
            *state = DONE_I;
        } else {
            goBackToStart(state, &destuffing);
            return DATA_INVALID;
        }
        break;

    case DONE_I:
    case DONE_S_U:
        destuffing = DESTUFF_OK;
    default:
        break;
    }
    prevByte = *byte;
```

```
    return StateOK;
}
```

transmitterFuncs.c

```
#include "transmitterFuncs.h"

extern bool stopAndWaitFlag;
static bool rejFlag = false;

void alarmHandler(int signo) {
    if (signo == SIGALRM) {
        stopAndWaitFlag = true;
        debugMessage("[SIG HANDLER] SIGALRM");
    }
}

bool stopAndWait(bool (*functionToExec)(char*,size_t, size_t*), char * msgToWrite,
size_t msgSize, size_t  *res) {
    unsigned counter = 0;
    bool first = true;

    #ifdef DEBUG
    struct myTimer timer;
    initTimer(&timer, "SIGALARM");
    #endif

    stopAndWaitFlag = true;   // used by the alarm

    while(counter < getNumTransmissions()) { // sends the message NO_TRIES times
        if(stopAndWaitFlag) {
            stopAndWaitFlag = false;
            counter++;

            alarm(getTimeout());
            //printf("sending message with s = %d and size = %d\n", nextS, msgSize);

            #ifdef DEBUG
            stopTimer(&timer, !rejFlag);
            rejFlag = false;

            startTimer(&timer);
            #endif
            if (functionToExec(msgToWrite, msgSize, res)) { // RR, sends the next one
                alarm(0); // unset alarm
                return true;
            }

            /* REJ is a retransmission request, so everytime it receives a REJ resends
the data
            NO_TRIES is only for retransmission tries due to timeout*/
            if (rejFlag || first) counter--; // if it's sending the first time or
because of REJ
            first = false;
        }
    }
```

```
    printError("Error sending message (retried %d times with no/invalid response)\n",
getNumTransmissions());
    printCharArray(msgToWrite, msgSize);
    alarm(0);

    exit(EXIT_FAILURE);
}

bool logicConnectionFunction(char * msg, size_t msgSize, size_t *res ) {
    ssize_t size;
    char *ret = (char*)myMalloc(getMaxFrameBufferSize()*sizeof(char));

    *res = writeToSP(msg, SUPERVISION_MSG_SIZE);
    if (*res == -1) {
        perror("Error writing to SP");
        return false;
    } else if (*res != msgSize) { //verifies if it was written correctly
        printError("Error while writing to SP (size doesn't match)\n");
        return false;
    }

    enum stateMachine state;
    if (readFromSP(ret, &state, &size, ADDR_SENT_EM, CTRL_UA) == READ_ERROR) return
false;

    free(ret);

    if(isAcceptanceState(&state)) {
        debugMessage("[LOGIC CONNECTION] SUCCESS");
        return true;
    }
    return false;
}

bool transmitterConnect() {
    char *buf = (char*)myMalloc(sizeof(char)*SUPERVISION_MSG_SIZE); // the message to
send
    size_t res;

    constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_SET);

    return stopAndWait(&logicConnectionFunction, buf, SUPERVISION_MSG_SIZE, &res); //
writes to the serial port, trying to connect
}


bool disconnectionFunction(char * msg, size_t msgSize, size_t *res ) {
    ssize_t size;
    char *ret = (char*)myMalloc(getMaxFrameBufferSize()*sizeof(char));

    //writes to serial port, trying to connect

    *res = writeToSP(msg, SUPERVISION_MSG_SIZE);//write DISC

    //verifies if it was written correctly
    if (*res == -1) {
        perror("Error writing to SP");
```

```
        return false;
    }
    else if (*res != msgSize) {
        printError("Error while writing to SP (size doesn't match)\n");
        return false;
    }

    enum stateMachine state;

    //tries to read the message back from the serialPort
    readFromSP(ret, &state, &size, ADDR_SENT_RCV, CTRL_DISC);//read DISC

    free(ret);
    if(isAcceptanceState(&state))
        return true;

    return false;
}

bool transmitterDisconnect() {
    char *buf = (char*) myMalloc(SUPERVISION_MSG_SIZE*sizeof(char));
    size_t res;
    constructSupervisionMessage(buf, ADDR_SENT_EM, CTRL_DISC);
    if (stopAndWait(&disconnectionFunction, buf, SUPERVISION_MSG_SIZE, &res)) {
        constructSupervisionMessage(buf, ADDR_SENT_RCV, CTRL_UA);
        writeToSP(buf, SUPERVISION_MSG_SIZE); //write UA
        free(buf);
        debugMessage("[DISC] SUCCESS");
        return true;
    }

    return false;
}

bool sendDataFunction(char* msg, size_t msgSize, size_t *res ){
    ssize_t size;
    char *ret = (char*)myMalloc(getMaxFrameBufferSize()*sizeof(char));

    *res = writeToSP(msg, msgSize);
    if (*res == -1) {
        perror("Error writing to SP");
        return false;
    } else if (*res != msgSize) { //verifies if it was written correctly
        printError("Error while writing to SP (size doesn't match)\n");
        return false;
    }

    enum stateMachine state;
    enum readFromSPRet result;
    result = readFromSP(ret, &state, &size, ADDR_SENT_EM, ANY_VALUE);
    /*printf("Returned message: ");
    fwrite(ret, sizeof(char), size, stdout);
    printf("\n");*/
    if (result == READ_ERROR) return false;

    free(ret);
```

```
    if(!isAcceptanceState(&state)) {
        debugMessage("[SENDING DATA] Didn't receive a valid response.\n");
        return false;
    }
    if(result == REJ){
        debugMessage("[SENDING DATA] Received REJ.\n");
        stopAndWaitFlag = true;
        rejFlag = true;
        return false;
    }

    return true;
}

size_t transmitterWrite(char* msg, size_t msgSize) {
    size_t bytesWritten;
    if (!stopAndWait(&sendDataFunction, msg, msgSize, &bytesWritten)) return -1;
    return bytesWritten;
}
```

defines.h

```
#ifndef _DEFINES_H_
#define _DEFINES_H_


#define TRANSMITTER     0
#define RECEIVER        1

#define BIT(n)      (0x01 << (n))

#define MAX_DEBUG_MSG_LENGTH 100

/* Message */
#define MSG_FLAG    0x7E /*all messages are delimited by this flag*/


/* Address */
#define ADDR_SENT_EM    0x03 /*Commands sent by transmitter and answers sent by
receiver*/
#define ADDR_SENT_RCV   0x01 /*Commands sent by receiver and answers sent by
transmitter*/


/* Control Protocol*/
#define CTRL_SET        0x03 /*Set up control flag*/
#define CTRL_UA         0x07 /*Unnumbered acknowledgment control flag*/
#define CTRL_DISC       0x0B /*Disconnect control flag*/
#define CTRL_S(s)       (s<<6) /*0S000000 S = N(s)*/
#define CTRL_RR(r)      ((r%2)<<7) | BIT(2) | BIT(0) /*Receiver ready / positive ACK
control flag*/
#define CTRL_REJ(r)      ((r%2)<<7) | BIT(0) /*Reject / negative ACK control flag */


/* Indexes */
#define BEGIN_FLAG_IDX  0
#define ADDR_IDX        1
```

```
#define CTRL_IDX        2
#define BCC1_IDX        3

#define BCC(a,c) (a^c) /*Calculation of BCC = XOR*/


/* Fixed Constants */
#define SUPERVISION_MSG_SIZE 5
#define ANY_VALUE       -1
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define printError(args...) fprintf(stderr, ##args)

#endif /* _DEFINES_H_*/
```

Aplicação (pasta *application*)

main.c (pasta src)

```
#include "application/application.h"

void applicationLoop(int argc, char ** argv) {
    checkCmdArgs(argc, argv);

    if (strcmp(argv[2], TRANSMITTER_STRING) == 0) { // TRANSMITTER
        transmitter(atoi(argv[1]), (argc >= 4 ? argv[3] : NULL), (argc >= 5 ? argv[4] :
NULL));
    }
    else { // RECEIVER
        receiver(atoi(argv[1]));
    }

}

int main(int argc, char ** argv) {
    applicationLoop(argc, argv);
    return 0;
}
```

application.c

```
#include "application.h"

extern char *optarg;

int MAX_DATA_PACKET_DATA_SIZE;

void printUsage() {
    printf("Usage:\n  ./app <serial port> <actor>\nWhere <serial port> is the 'x'
in:\n/dev/ttySx\nAnd <actor> is either RECEIVER or TRANSMITTER.\nWhen <actor> is
TRANSMITTER you can add two optional arguments: <fileToSend> <destFileName>, which need
to come in this order after the mandatory arguments.\nYou can also specify frame size (-
t) and baudrate (-b). Be sure that these values match in the RECEIVER and the
TRANSMITTER.\nBaudrate can be one of the
following:\n4800\n9600\n19200\n38400\n57600\n115200\n230400\n");
    exit(EXIT_FAILURE);
}
```

```
void checkCmdArgs(int argc, char ** argv) {
    if (argc < 3) {
        printUsage();
    }
    else {
        bool isValidSP = isUnsignedNumber(argv[1]);
        if ((strcmp(argv[2], TRANSMITTER_STRING) != 0 && strcmp(argv[2],
RECEIVER_STRING) != 0) || !isValidSP) {
            printUsage();
        }

        int frameSize = -1, baudrate = -1;


        int option;
        while ((option = getopt(argc, argv, "-t:b:")) != -1) {
            switch(option) {
                case 't':
                    if (!isUnsignedNumber(optarg)) {
                        printError("Frame size (-t) has to be an unsigned number.\n");
                    } else {
                        frameSize = atoi(optarg);
                    }
                    break;
                case 'b':
                    if (!isUnsignedNumber(optarg)) {
                        printError("Baudrate (-b) has to be an unsigned number.
Available:\n4800\n9600\n19200\n38400\n57600\n115200\n230400\n");
                    } else {
                        baudrate = atoi(optarg);
                    }
                    break;
                case 1:
                default:
                    break;
            }
        }

        llset(baudrate, frameSize);

        MAX_DATA_PACKET_DATA_SIZE = getMaxDataPacketSize() - 4;

    }
}
```

receiver.c

```
#include "receiver.h"
#include "../tests/efficiency.h"


void recAlarmHandler(int signo) {
    if (signo == SIGALRM) {
        printError("Turning off due to inactivity.\n");
        exit(EXIT_FAILURE);
    }
}
```

```
void receiver(int serialPort){
    if (signal(SIGALRM, recAlarmHandler) < 0) {  // Installs the handler for the alarm
interruption
            perror("Alarm handler wasn't installed");
            exit(EXIT_FAILURE);
    }
    siginterrupt(SIGALRM, true);  // SIGALARM interrupts system calls (returns -1 and
errno is se to EINTR)
    alarm(INACTIVITY_TIME);
    int fd;
    if ((fd = llopen(serialPort, RECEIVER_STRING)) == -1) {// Establishes communication
with transmitter
        printError("Wasn't able to establish logic connection!\n");
        exit(EXIT_FAILURE);
    }


    char *buffer = (char*) myMalloc(getMaxDataPacketSize()*sizeof(char));
    char* fileName = NULL;
    size_t fileSize;
    bool receivedStart = false;
    FILE * fileToSave;


    unsigned bytesReceived = 0;
    while(true){
        int dataRead = llread(fd,buffer);
        if (dataRead < 0 && errno != EINTR) { //case of error
            printError("Error reading the file.\n");
            exit(EXIT_FAILURE);
        }
        else if (dataRead == 0) continue; //case of REJ sent or repeated data received
        else {
            alarm(INACTIVITY_TIME);
            int dataAmount = -1;
            enum checkReceptionState receptionRet = checkStateReception(buffer,
dataRead, &fileSize, &fileName, &dataAmount);
            //printf("Checking the following buffer, with size %d.\n", dataRead);
            //printCharArray(buffer, dataRead);
            if(receptionRet == START_RECEIVED && !receivedStart){
                printf("Starting to receive file '%s' with %ld bytes.\n", fileName,
fileSize);
                FILE * aux;
                unsigned tries = 1;
                char fileNameCurrent[256];
                char fileNameNext[257]; // the added 1 is in case the received string
has exactly 256 characters (counting \0)

                strcpy(fileNameCurrent, fileName);

                while ((aux = fopen(fileNameCurrent, "rb")) != NULL){
                    fclose(aux);
                    if (tries > 5) {
                        printError("Conflicts with destination filenames! Tried to find
an unused name 5 times.\n");
                        exit(EXIT_FAILURE);
```

```
                    }
                    addIntToFilename(fileNameNext, fileName, tries);
                    printError("Filename '%s' exists, trying with '%s'.\n",
fileNameCurrent, fileNameNext);
                    strcpy(fileNameCurrent, fileNameNext);
                    tries++;
                }

                fileToSave = fopen(fileNameCurrent, "wb");
                if (ferror(fileToSave)) {
                    perror("Error opening file");
                    exit(EXIT_FAILURE);
                }

                receivedStart = true;
                // check current receiver state (whether it has already received start,
if it has, error)
            } else if(receptionRet == DATA_FINISHED && receivedStart){ //it has received
the end of the end control packry
                // check if receiver has received start, it it didn't, it's error
                if (dataAmount > 0){
                    fwrite(&buffer[4], sizeof(char), (size_t)dataAmount, fileToSave);
                    bytesReceived += dataAmount;
                } else if (dataAmount == 0) {
                    printError("Amount of data received is 0.\n");
                    exit(EXIT_FAILURE);
                } else if (dataAmount < 0) {
                    printError("Amount of data received is negative.\n");
                    exit(EXIT_FAILURE);
                }
                printProgressBar(fileSize, bytesReceived);
            } else if (receptionRet == END_RECEIVED && receivedStart) {
                fclose(fileToSave);
                if (fileSize == bytesReceived) {
                    printf("File received successfully!\n");
                    break;
                } else {
                    printError("Specified file size (%ld) and received number of bytes
(%d) do not match.\n", fileSize, bytesReceived);
                    break;
                }
            } else if (receptionRet == ERROR){
                printError("\nThere was an error in the reception, received invalid
data. Terminating connection.\n");
                exit(EXIT_FAILURE);
            } else {
                printError("Received invalid data package! State: %d\n", receptionRet);
                exit(EXIT_FAILURE);
            }
            bzero(buffer, getMaxDataPacketSize());
        }

    }

    free(buffer);
    free(fileName);
```

```
    #ifdef EFFICIENCY_TEST
    errorsGenerated();
    #endif

    /* After receiving and end control packet, it has received the full file so it's
going to disconnect from the transmitter*/
    if (llclose(fd) != 0) {
        printError("Wasn't able to disconnect!\n");
        exit(EXIT_FAILURE);
    }
}
```

stateMachine.c

```
#include "stateMachine.h"

static int sequenceNumber = 0;

enum checkReceptionState checkStateReception(char * buffer, int bufferSize, size_t *
fileSize, char ** fileName, int * dataAmount){
    int totalAmountOfChars = -1;
    int currentAmountOfChars = -1;
    int idx = 1;
    bool end = false;
    size_t auxFileSize = 0x0;
    char * auxFileName = NULL;
    enum checkReceptionState state = CTRL;

    for (int i = 0; i < bufferSize; i++) {
        unsigned char byte = buffer[i];
        switch(state){
            case(CTRL):
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("CTRL");
                #endif
                if(byte == START_CTRL){
                    state = T;
                }
                else if (byte == DATA_CTRL) {
                    state = DATA_N;
                }
                else if(byte == END_CTRL){
                    end = true;
                    state = T;
                }
                else {
                    printError("Error in package SM: Received invalid CTRL\n");
                    state = ERROR;
                }
                break;
            case(T):
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("T");
                #endif
                //printf("Received %u in T\n", byte);
                if ((byte == APP_FILE_T_SIZE && idx == 1) || (byte == APP_FILE_T_NAME &&
idx == 2)){
```

```
                    if (!end && idx == 1) *fileSize = 0x0;
                    state = L;
                } else {
                    printError("Error in package SM: Received invalid T\n");
                    state = ERROR;
                }
                break;
            case(L):
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("L");
                #endif
                if ((idx == 1) || (idx == 2)){
                    totalAmountOfChars = byte;
                    currentAmountOfChars = 0;
                    state = V;
                    //printf("Receiving string with size %d\n", totalAmountOfChars);
                    if (idx == 2) {//we already know the size of the file so we can
allocate memory for it's data
                        if (!end)
                            *fileName = (char*) myMalloc(sizeof(char) *
(totalAmountOfChars + 1)); // the +1 is because of the \0 end character
                        else
                            auxFileName = (char*) myMalloc(sizeof(char) *
(totalAmountOfChars + 1));
                    }
                } else {
                    printError("Error in package SM: Received invalid L\n");
                    state = ERROR;
                }
                break;
            case(V):
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("V");
                #endif
                if(currentAmountOfChars < totalAmountOfChars && idx == 1){//fileSize
                    if(!end){
                        *fileSize = (*fileSize << 8) | (unsigned char)byte;
                    }
                    else{
                        auxFileSize = (auxFileSize << 8) | (unsigned char)byte;
                    }
                    //printf("Value of byte: %x\n", (unsigned char) byte);
                    //printf("Value of fileSize: %lx\n", (unsigned char) *fileSize);
                    currentAmountOfChars++;
                    if(currentAmountOfChars == totalAmountOfChars){
                        //printf("Received file size\n");
                        if(!end){
                            state = T; //going to receive the fileName
                            idx = 2;
                        }
                        else{
                            //printf("Received file size - end. %ld, %ld\n",
auxFileSize, *fileSize);
                            if(auxFileSize != *fileSize){
                                printError("Error in package SM: Received invalid file
size in end package\n");
                                state = ERROR;
```

```
                        }
                        else {
                            //printf("changed to t\n");
                            state = T;
                            idx = 2;
                        }
                    }
                }
            }
            else if(currentAmountOfChars < totalAmountOfChars && idx ==
2){//fileName
                if(!end)
                    (*fileName)[currentAmountOfChars++] = byte;
                else
                    auxFileName[currentAmountOfChars++]= byte;
                /*printf("Current string: ");
                fwrite(*fileName, sizeof(char), currentAmountOfChars-1, stdout);
                printf("\n");*/
                if(currentAmountOfChars == totalAmountOfChars){
                    if(!end){
                        (*fileName)[currentAmountOfChars] = '\0';
                        state = START_RECEIVED;
                    }
                    else{
                        if(strcmp(*fileName, auxFileName) != 0){
                            state = ERROR;
                            printError(" V: End packet with wrong filename\n");
                        }
                        else
                            state = END_RECEIVED;
                    }
                }
            }
            else{
                state = ERROR;
                printError("V: Not filename or filesize\n");
            }
            break;
        case DATA_N:
            #ifdef DEBUG_APP_STATE_MACHINE
            debugMessage("CTRL_DATA");
            #endif
            if((unsigned)byte >= 0 && (unsigned)byte < 255){
                if((unsigned)byte == sequenceNumber){
                    sequenceNumber = (sequenceNumber+1) % 255;
                    state = DATA_L2;
                }
                else{
                    state = ERROR;//sequence number not valid
                    printError("DataN: Sequence number not valid 1\n");
                }

            }
            else{
                state = ERROR;
                printError("Data N: Sequence number not valid 2\n");
            }
```

```
                break;
            case DATA_L2:
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("SEQUENCE_NUMBER");
                #endif
                totalAmountOfChars = (unsigned)((byte << 8) & (unsigned)0xFF00);//number
of bytes to read to complete the data
                state = DATA_L1;
                break;
            case DATA_L1:
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("L_DATA");
                #endif
                totalAmountOfChars |= (byte & 0x000FF);
                currentAmountOfChars = 0;
                //printf("Receiving %x bytes of data\n", totalAmountOfChars);
                if(totalAmountOfChars >= 0){ // we store the data
                    state = RECEIVING_DATA;
                }
                else{
                    state = ERROR;
                     printError("Data L1: Invalid total amount of chars\n");
                }
                break;

            case RECEIVING_DATA:
                #ifdef DEBUG_APP_STATE_MACHINE
                debugMessage("DATA_PACKET_FINISH");
                #endif
                *dataAmount = totalAmountOfChars;
                state = DATA_FINISHED;
                break;
            case DATA_FINISHED: case END_RECEIVED: case START_RECEIVED:
            case ERROR:
                break;
        }
    }
    return state;
}
```

// transmitter.c

```
#include "transmitter.h"
#include "../tests/efficiency.h"

extern unsigned MAX_DATA_PACKET_DATA_SIZE;

void transmitter(int serialPort, char * fileToSend, char * destFile){

    int fd = llopen(serialPort, TRANSMITTER_STRING); // Establish communication with
receiver

    if (fd == -1) {
        printError("Wasn't able to establish logic connection!\n");
        exit(EXIT_FAILURE);
    }
```

```
    /*Starts to write all information frames, keeping in mind the need to resend, etc.
    While there is info to write, writes with the stop&wait mechanism as in other
situations*/

    char* filename = fileToSend != NULL ? fileToSend : "littleLambs.jpg";
    char* destFilename = destFile != NULL ? destFile : "receivedFile.jpg";
    FILE * file = fopen(filename, "rb");

    if(file == NULL){
        printError("File %s does not exist.\n", filename);
        exit(EXIT_FAILURE);
    }

    struct stat st;
    stat(filename, &st);
    size_t fileSize = st.st_size;

    char *buffer = (char*)myMalloc(getMaxDataPacketSize()*sizeof(char));
    char *ret = (char*)myMalloc(getMaxDataPacketSize()*sizeof(char));

    if(feof(file)){
        printError("The file is empty, nothing to send.\n");
        exit(EXIT_FAILURE);
    }
    else{//the file is not empty
        int packetSize = constructControlPacket(ret, START_CTRL, destFilename,
fileSize);
        if(packetSize == -1){
            exit(EXIT_FAILURE);
        }
        //printCharArray(ret, packetSize);
        if(llwrite(fd,ret, packetSize*sizeof(char)) == -1){  //sending start control
packet
            printError("Error sending the start control packet.\n");
            exit(EXIT_FAILURE);
        }
    }

    printf("Starting to send file '%s' (destination: '%s') with %ld bytes.\n", filename,
destFilename, fileSize);

    int msgNr = 0;
    char *dataPacket = (char*) myMalloc(getMaxDataPacketSize()*sizeof(char));
    size_t amountTransfered = 0;

    #ifdef EFFICIENCY_TEST
    struct myTimer timer;
    initTimer(&timer, "Efficiency timer");
    #endif

    while (!feof(file)) {
        size_t amount = fread(buffer, sizeof(char), MAX_DATA_PACKET_DATA_SIZE, file);
        constructDataPacket(dataPacket, buffer, amount, msgNr);
        //printCharArray(dataPacket, amount+4);

        #ifdef EFFICIENCY_TEST
```

```
        startTimer(&timer);
        #endif

        size_t ret = llwrite(fd, dataPacket, amount+4); // second arg has a maximum size
of MAX_DATA_PACKET_SIZE
        if (ret == -1) {
            printError("Error in llwrite\n");
            exit(EXIT_FAILURE);
        }
        if (ferror(file)) {
            perror("Error reading file");
            exit(EXIT_FAILURE);
        }

        #ifdef EFFICIENCY_TEST
        double elapsedTime = stopTimer(&timer, false);
        rateValuesUpdate(ret, elapsedTime);
        #endif


        msgNr++;
        amountTransfered += amount;
        printProgressBar(fileSize, amountTransfered);
    }

    #ifdef EFFICIENCY_TEST
    calculateEfficiency();
    #endif

    if (amountTransfered == fileSize) {
        printf("File transfered successfully!\n");
    } else {
        printError("Was supposed to transfer %ld bytes, only transfered %ld\n",
fileSize, amountTransfered);
    }

    fclose(file);

    int endSize = constructControlPacket(ret, END_CTRL,destFilename, fileSize);

    if(llwrite(fd,ret, endSize*sizeof(char)) == -1){ // sending end control packet,
sends the same packet as start control packet with the difference in the control
        printError("Error sending the end control packet\n");
        exit(EXIT_FAILURE);
    }

    free(ret);
    free(buffer);
    free(dataPacket);

    /*
        When there is no more information to write, it's going to disconnect: sends a
DISC frame, receives a DISC frame
        and sends back an UA frame, ending the program execution
    */
    debugMessage("SENDING DISC...");
```

```
    if (llclose(fd) != 0) {
        printError("Wasn't able to disconnect!\n");
        exit(EXIT_FAILURE);
    }
}


int constructControlPacket(char * ret, char ctrl, char* fileName, size_t fileSize){
    size_t fileNameSize = strlen(fileName)+1;
    int fileSizeLength = sizeof(size_t);

    if(((fileSizeLength + fileNameSize + 5) * sizeof(char)) >
getMaxDataPacketSize()*sizeof(char)) {
        printError("File name size too large for the defined
getMaxDataPacketSize()!\n");
        return -1;
    }

    if(fileNameSize > 255){
        printError("File name size can not be larger than 255 characters!\n");
        return -1;
    }

    ret[APP_CTRL_IDX] = ctrl;
    ret[APP_FILE_T_SIZE_IDX] = APP_FILE_T_SIZE;
    ret[APP_FILE_L_SIZE_IDX] = (unsigned char)fileSizeLength;

    for (int i = fileSizeLength-1; i >= 0; i--) {
        ret[APP_FILE_V_SIZE_IDX + ((fileSizeLength-1) - i)] = (unsigned char)((fileSize
>> i*8) & 0x0FF);
        //printf("File size: %lx; File size current: %x\n", fileSize, ((unsigned
char)fileSize) >> i);
    }

    unsigned nameTPosition = APP_FILE_T_NAME_IDX + fileSizeLength-1;
    unsigned nameLPosition = APP_FILE_L_NAME_IDX + fileSizeLength-1;

    ret[nameTPosition] = APP_FILE_T_NAME;
    ret[nameLPosition] = (unsigned char)strlen(fileName);

    for (int i = 0; i < strlen(fileName); i++) {
        ret[nameLPosition + 1 + i] = (unsigned char)fileName[i];
    }

    return CTRL_PACKET_SIZE + fileSizeLength-1 + strlen(fileName);
}

//K = 256*L2 +L1 = 2⁸ * L2 + L1 = L2<<8 +L1, L2 is the most significant byte of dataSize
and L1 the least signifcant byte
void constructDataPacket(char * ret, char* data, size_t dataSize, int msgNr){
    ret[APP_CTRL_IDX] = DATA_CTRL;
    ret[APP_SEQUENCE_NUM_IDX] = msgNr % 255;
    ret[APP_L2_IDX] = (dataSize & 0x0FF00) >> 8;
    ret[APP_L1_IDX] = dataSize & 0x00FF;

    memcpy(ret + APP_DATA_START_IDX, data, dataSize);//inserts the data in the
DataPacket at its right index    }
```

defines.h

```
/* Control Packet*/
#define START_CTRL      0x02
#define END_CTRL        0x03

#define printError(args...) fprintf(stderr, ##args)

/* Indexes */
#define APP_CTRL_IDX            0
#define APP_FILE_T_SIZE_IDX     1
#define APP_FILE_L_SIZE_IDX     2
#define APP_FILE_V_SIZE_IDX     3
#define APP_FILE_T_NAME_IDX     4
#define APP_FILE_L_NAME_IDX     5
#define APP_FILE_V_NAME_IDX     6
#define APP_FILE_T_SIZE         0
#define APP_FILE_T_NAME         1
#define APP_SEQUENCE_NUM_IDX    1
#define APP_L2_IDX              2
#define APP_L1_IDX              3
#define APP_DATA_START_IDX      4

/* Data Packet */
#define DATA_CTRL       0x01

#define INACTIVITY_TIME 60

#define CTRL_PACKET_SIZE    7
#define APP_SEQ_NUM_SIZE    256

#define PORT_AMOUNT 2
#define TRANSMITTER_STRING "TRANSMITTER"
#define RECEIVER_STRING "RECEIVER"
```

Testes de eficiência (pasta *tests*)

efficiency.c

```
#include "efficiency.h"

static unsigned n_Packets = 0;
static double totalTime = 0;
static size_t totalSize = 0;

static unsigned dataErrorCount = 0;
static unsigned headErrorCount = 0;


void rateValuesUpdate(size_t packetSize, double time){
    n_Packets++;
    totalTime += time;
    totalSize += packetSize;
}

double getAverageRate(){
    return (totalSize*8) / totalTime;
}
```

```
void calculateEfficiency(){
    unsigned baudrateNum = 0;
    switch(getBaudrate()) {
        case B4800:
            baudrateNum = 4800;
            break;
        case B9600:
            baudrateNum = 9600;
            break;
        case B19200:
            baudrateNum = 19200;
            break;
        case B38400:
            baudrateNum = 38400;
            break;
        case B57600:
            baudrateNum = 57600;
            break;
        case B115200:
            baudrateNum = 115200;
            break;
        case B230400:
            baudrateNum = 230400;
            break;
        default:
            printError("The specified baudrate isn't valid.
Available:\n4800\n9600\n19200\n38400\n57600\n115200\n230400\n");
            return;
    }
    printf("\nProbability head error: %u%%\nProbability data error: %u%%\n\n# Error
generation\nGenerated %u head errors.\nGenerated %u data errors.\n\n",
PROBABILITY_HEAD, PROBABILITY_DATA, headErrorCount, dataErrorCount);
    printf("T_prop: %u\n", DELAY);
    printf("Baudrate: B%u\n", baudrateNum);
    printf("Frame size: %u\n\n", getFrameSize());
    printf("Total time of transfer: %f\n", totalTime);
    printf("Total size of transfer: %lu, %u packets\n", totalSize, n_Packets);
    printf("Average rate: %lf bits/s\n", getAverageRate());
    double efficiency = getAverageRate()/(double)baudrateNum;
    printf("-- Efficiency: %lf\n", efficiency);
}

void errorsGenerated() {
    printf("Probability head error: %u%%\nProbability data error: %u%%\n\n\n# Error
generation\nGenerated %u head errors.\nGenerated %u data errors.\n\n",
PROBABILITY_HEAD, PROBABILITY_DATA, headErrorCount, dataErrorCount);
}

void delayGenerator(){
    if (DELAY != 0) usleep(DELAY*1e3);  // DELAY in miliseconds
}


void generateDataError(char *frame, size_t frameSize){
    int probability = (rand() % 100) + 1;
```

```
    if(probability <= PROBABILITY_DATA){
        int randomByte = (rand()% (frameSize - 5)) + 4;
        char randomByteValue = (rand()%177);

        frame[randomByte] = randomByteValue;
        printf("\nGenerate BCC2 with errors\n");

        dataErrorCount++;

    }

}

void generateHeadError(char *buffer){
    int probability = (rand() % 100) + 1;

    if(probability <= PROBABILITY_HEAD){
        int randomByte = (rand() % 2);
        char randomByteValue = (rand()%177);

        buffer[randomByte] = randomByteValue;
        printf("\nGenerate BCC1 with errors\n");

        headErrorCount++;

    }
}
```

Utilitários (pasta *utilities*)

// utilities.c

```
#include "utilities.h"
#include <math.h>

void debugMessage(char * msg) {
    #ifndef DEBUG
    return;
    #endif
    printf("- %s\n", msg);
}

void * myMalloc(size_t size) {
    void * ret = malloc(size);
    if (ret == NULL) {
        perror("Error alocating memory.");
        exit(EXIT_FAILURE);
    }
    return ret;
}

bool isUnsignedNumber(char * str) {
    for(int i = 0; str[i] != '\0'; i++) {
        if(!(str[i] >= '0' && str[i] <= '9')) {
            return false;
        }
    }
```

```
    return true;
}

void printCharArray(char * arr, size_t arrSize) {
    int count = 0;
    for (size_t i = 0; i < arrSize; i++) {
        count++;
        printf("%x ", (unsigned char)arr[i]);
    }
    printf("\ntotal: %d, dados: %d\n", count, count-6);
    printf("\n");
}

void initTimer(struct myTimer * timer, char * name) {
    timer->started = false;
    timer->timerName = name;
}

void startTimer(struct myTimer * timer) {
    timer->started = true;
    clock_gettime(CLOCK_REALTIME, &(timer->startTime));
}

double stopTimer(struct myTimer * timer, bool verbose) {
    if (!timer->started) return -1;
    timer->started = false;

    struct timespec endTime;
    clock_gettime(CLOCK_REALTIME, &endTime);


    time_t sElapsed  = endTime.tv_sec - timer->startTime.tv_sec;
    time_t nsElapsed = endTime.tv_nsec - timer->startTime.tv_nsec;

    double timeInSeconds = sElapsed + nsElapsed/1.0e9;

    if (verbose) {
        printf("Timer %s took %lf seconds\n", timer->timerName, timeInSeconds);
    }

    return timeInSeconds;
}


void addIntToFilename(char * fileNameNext, char * fileName, int tries) {
    unsigned numberOfDigits = floor(log10(abs(tries)) + 1);
    int fileNameSize = strlen(fileName);
    int lastDotIdx = -1;
    for (unsigned i = 0; i < fileNameSize; i++) {
        if (fileName[i] == '.') {
            lastDotIdx = i;
        }
    }
    if (lastDotIdx != -1) {
        char * fileNameHolder = malloc(strlen(fileName)*sizeof(char));
        strcpy(fileNameHolder, fileName);
        fileNameHolder[lastDotIdx] = '\0';
```

```
        sprintf(fileNameNext, "%s%d", fileNameHolder, tries);
        for (int i = lastDotIdx; i < fileNameSize; i++) {
            fileNameNext[i + numberOfDigits] = fileName[i];
        }
        fileNameNext[fileNameSize+numberOfDigits] = '\0';
        free(fileNameHolder);
    } else {
        sprintf(fileNameNext, "%s%d", fileName, tries);
    }

}
```

ui.c

```
#include "ui.h"

void printProgressBar(size_t goal, size_t current) {
    #ifndef LOADING_UI
    return;
    #endif
    struct winsize size;
    ioctl(1, TIOCGWINSZ, &size);

    /* size.ws_row is the number of rows, size.ws_col is the number of columns. */
    int numCol = size.ws_col;

    double ratio = (double)current / (double) goal;

    int blockAmount = numCol - 7;

    int filledBlocks = ratio * blockAmount;

    printf("%03i%% [", (int)(ratio*100));
    for (int i = 0; i < filledBlocks; i++) {
        printf("\u2588");
    }
    for (int i = 0; i < blockAmount - filledBlocks; i++) {
        printf(" ");
    }
    printf("]\r");

    if (ratio == 1) {
        printf("\n");
    }

}
```

globalDefines.h

```
//#define DEBUG
#define LOADING_UI
//#define DEBUG_STATE_MACHINE
//#define DEBUG_APP_STATE_MACHINE

//#define EFFICIENCY_TEST
```