RCOM - 1º Projeto

Diogo Alexandre Soares Martins (202108883)
Tomás Figueiredo Marques Palma (202108880)

1. Introdução

O trabalho de redes de computadores teve como objetivo a construção de um protocolo de dados de baixo nível para a transmissão de bytes, assim como a construção de uma aplicação para transferência de ficheiros que utiliza a camada anteriormente referida, servindo este relatório como meio de documentação das camadas desenvolvidas, assim como um documento com conclusões sobre a eficiência e sobre o quão corretos os móduluos desenvolvidos são.

No relatório, primeiramente ir-se-á encontrar documentado detalhes sobre a arquiterura, a estrutura do código e os casos de uso principais das camadas desenvolvidas e de seguida falar-se-à com detalhe acerca de cada um dos protocolos desenvolvidos (o da ligação lógica e o da aplicação), sendo, depois, por fim, abordados a correção e a eficiência do programa.

2. Arquitetura

Será descrito os diferentes módulos e como interagem entre si.

2.1. Módulo Porta Série (`serial_port.c`)

Este módulo é responsável pela configuração inicial da porta série de modo a ficar separado das outras camadas de modo a que saibam o mínimo de informações internas, como o `baudrate`, aumentando a modularidade.

2.2. Módulo de ligação de dados (`link_layer.c`)

Este módulo está acima do módulo da porta série, sendo este o responsável por invocar a API da porta série, de modo a começar a configuração inicial da mesma, assim como o responsável por ter o conhecimento acerca de como transferir e receber uma trama quer seja ela de informação ou de supervisão, fazendo a verificação de erros e obrigando a uma retransmissão em caso de falha.

2.3. Módulo da aplicação (`application_layer.c`)

Este módulo implementa a funcionalidade quer de transferência como de receção de ficheiros de entre duas portas série, invocando a API exposta pelo protocolo de ligação de dados, sem nunca ter conhecimento interno de como transferir uma trama de um lado para o outro, simplesmente apenas tendo o conhecimento de como pedir ao módulo de ligação de dados para transferir pacotes, assim como codificar e descodificar informação do ficheiro como o seu nome, tamanho e conteúdo de modo a que as duas partes estejam sincronizadas num mesmo protocolo.

2.4. Módulo de *utils* (`utils.c`)

2.4.1. Relação com a arquitetura

Este módulo implementa uma API com funcionalidades fora da responsabilidade de outros módulos que em termos tal como teóricos poderiam ser usados em qualquer módulo.

2.4.2. Principais funções (API)

```
int get_size_of_file(FILE *file);
int get_no_of_bits(int n);
```

3. Estrutura do código

3.1. Módulo porta série

Este módulo é responsável pela configuração inicial da porta série.

```
#define BAUDRATE 115200

int setup_port(int fd);
```

3.2. Módulo da ligação de dados

3.2.1. Relação com a arquitetura

Este módulo é o responsável pela transmissão e receção de bytes de uma porta série, sem ter o conhecimento do que é que os bytes, para além dos de controlo especificados pelo guião do trabalho, significam.

3.2.2. Principais funções (API)

```
int llopen(LinkLayer connectionParameters);
int llwrite(const unsigned char *buf, int bufSize);
int llread(unsigned char *packet);
int llclose(LinkLayerRole role, int showStatistics);
int show_statistics(LinkLayerRole role);
int llclose_transmitter();
int llclose_receiver();
```

3.2.3. Principais estruturas de dados

```
typedef enum {
  LlTx,
  LlRx,
} LinkLayerRole;
```

- É utilizado como uma forma mais amigável ao programador de identificar se o programa está a ser executado como recetor ou transmissor.

```
typedef struct {
  char serialPort[50];
  LinkLayerRole role;
  int baudRate;
  int nRetransmissions;
```

```
  int timeout;
} LinkLayer;
```

- É utilizado como um aglomerado de informação acerca da configuração de como o protocolo de ligação de dados irá funcionar e com que tipo de configuração da porta série.

3.3. Módulo da aplicação

3.3.1. Relação com a arquitetura

É o módulo responsável pela funcionalidade de transmitir e receber ficheiros.

3.3.2. Principais funções (API)

```
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);
int transmitter_application_layer(const char *filename);
int receiver_application_layer();
unsigned char *read_control_frame(int app_layer_control, int *file_size);
int read_file(int file_size, unsigned char *filename,
    FILE *file, int *current_size);
int send_file(FILE *file, int file_size);
int send_control_frame(const char *filename, int file_size,
    int app_layer_control);
```

3.3.3. Principais estruturas de dados

Foram criadas as seguintes estruturas de dados auxiliares:

```
typedef enum {
  READ_CONTROL_START,
  READ_FILESIZE,
  READ_FILENAME,
} READ_CONTROL_STATE;

typedef enum {
  READ_CONTROL_DATA,
  READ_DATA_L1,
  READ_DATA_L2,
  READ_DATA
} READ_FILE_STATE;
```

São estados utilizados na máquinas de estados de leitura do ficheiro.

```
int get_size_of_file(FILE *file);
int get_no_of_bits(int n);
```

4. Casos de usos principais

Em cada um dos casos estará descrito o fluxo lógico de execução.

4.1. Receber um ficheiro

A lógica inicia-se na função `receiver_application_layer` do ficheiro `application_layer.c`.

1. Receber a trama de conrole de início a partir da rotina `read_control_frame(CONTROL_START, &file_size_start)` que vai atribuir o tamanho do ficheiro e o nome do ficheiro a variáveis.
2. Abrir of ficheiro onde se vai escrever com `fopen`.
3. Depois de recebida a trama de controlo inicial, vai-se recebendo tramas de informação com a função `read_file`, até termos recebido o número de bytes igual ao indicado pela trama de controlo.
4. Por fim, recebemos a trama de controlo de fim a partir da rotina `read_control_frame(CONTROL_END, &file_size_stop)`.

4.2. Enviar um ficheiro

A lógica inicia-se na função `transmitter_application_layer` do ficheiro `application_layer.c`.

1. Ler o ficheiro para memória a partir com o `fread`.
2. Para além do nome do ficheiro que recebe a partir da função `main`, é necessário obter o tamanho do ficheiro aberto pelo `fread`, que se faz a partir da função `get_size_of_file` do `utils.c`.
3. De seguida, é construída e enviada a trama de controlo de início cuja especificação vai de encontro ao guião do trabalho, a partir da rotina `send_control_frame(filename, file_size, CONTROL_START)`.
4. Depois de transferida a trama de controlo inicial, vai-se enviando `MAX_DATAFIELD_SIZE` bytes do ficheiro de cada vez, a partir da função `send_file` execeto quando se chega ao momento em que a quantidade de bytes que resta enviar seja inferior a `MAX_DATAFIELD_SIZE`, circunstâncias essas onde se envia `Tamanho restante do ficheiro` bytes.
5. Por fim, após a transferência do ficheiro, enviamos a trama de controlo de fim que repete a mesma informação que a que a trama de controlo de início continha, a partir da rotina `send_control_frame(filename, file_size, CONTROL_START)`.

5. Protocolo de ligação de dados

Esta camada é a camada implementada por nós que é a de mais baixo nível, sendo ela a utilizada pelo protocolo de aplicação para transferir e receber ficheiros.

5.1. `llopen()`

Uma das responsabilidades desta função é a preparação da porta série para transmissão e receção, configurando-a com os parâmetros pretendidos, sendo de seguida iniciado o processo de estabelecimento de comunicação entre as duas partes como descrito no guião do trabalho, em que o transmissor envia uma trama de supervisão `SET` e fica à espera de receber um `UA` do recetor, caso em que depois permite a execução da camada de aplicação.

5.2. `llwrite()`

É responsável pela escrita de tramas de informação para a porta série recetora, processo no qual efetua um mecanismo de *byte stuffing*, caso existam bytes nos ficheiros a transferir que tenham significado para o protocolo de ligação de dados tal como é o caso do `0x7e` que indica o início e fim de uma trama, assim

comocalcula o *bcc* dos bytes enviados, não considerando os bytes adicionais resultantes do *byte stuffing*, colocando-os na trama de informação e, de seguida, fica à espera da resposta do recetor, retransmitindo a trama caso o recetor indique que detetou erro, ao calcular o bcc de novo.

5.3. `lread()`

É responsável pela leitura de tramas de informação recebidas pelo transmissor, tendo de preoceder a um *destuffing* dos bytes, assim como verificando se o bcc calculado ao ler os bytes sem contar com os bytes adicionais do *stuffing* é igual ao que foi enviado pelo transmissor na trama de informação e, caso não seja, tem de enviar uma trama de supervisão de rejeição `REJ(nr)` para que o transmissor volte a reenviar. Caso cálculo do *bcc* bata certo, envia uma trama de supervisão de aceitação `RR(nr)`.

5.4. `llclose()`

É responsável pelo fecho da conexão, onde o transmissor envia um `DISC` para o recetor, ficando à espera de receber um `DISC` de volta do recetor, de seguida enviando um `UA`, fechando a conexão. Por sua vez, o recetor apenas envia um `DISC` após receber o primeiro `DISC` do transmissor e depois fecha a conexão com a porta série.

6. Protocolo de aplicação

Esta camada é a camada de mais alto nível implementada por nós, sendo utilizada para interagir com o utilizador e com o ficheiro a ser transferido, utilizando a API do protocolo de ligação de dados.

6.1. `receiver_application_layer`

É responsável por receber, primeiramente, a trama de controlo de início, que contém informacação sobre o nome e tamanho do ficheiro, seguida das tramas de informação, com as quais preenche um novo ficheiro (o ficheiro transferido), e, por fim, a trama de controlo de fim, que repete os dados transferidos na trama de controlo de início.

6.2. `transmitter_application_layer`

É responsável por enviar o ficheiro desejado para o outro computador, começando pela trama de controlo de início, seguida das tramas de informação, que são geradas dividindo o ficheiro de modo que sejam o menor número possível de tramas, em que todas exceto a última têm tamamho `MAX_DATAFIELD_SIZE`, e, por fim, a trama de controlo de fim.

7. Validação

Este programa foi testado quer em ambiente de laboratório na FEUP quer virtualmente a partir do `cable.c` fornecido pela equipa docente.

Em caso de erro, o programa consegue recuperar, o que significa que o transmissor recebe a trama de rejeição do recetor, não enviando mais nenhuma trama a seguir, tentando simplesmente reenviar a trama que chegou ao recetor com erro, até o número de tentativas chegar a `MAX_TRIES` de `TIMEOUT` em `TIMEOUT` segundos.

Em caso de desconexão espontânea, o mecanismo de retransmissão comprovou-se fiável, sendo que após uma desconexão de mais que `TIMEOUT` segundos, o programa consegue recuperar.

Os mecanismos de retransmissão foram também testados a correr o transmissor antes do recetor, comprovando que o transmissor, após não receber uma resposta após 4 tentativas, aborta a conexão.

O programa também sucedeu com ficheiros de diferentes tamanhos:

- O ficheiro fornecido (penguin.gif) de 10968 bytes
- Um outro ficheiro introduzido por nós de 31802 bytes

8. Eficiência do protocolo de ligação de dados

Devido ao tamanho das tabelas e figuras, para melhor respeitar o limite de páginas imposto, estas encontram-se no anexo.

8.1. Variar `Frame Error Ratio`

```
if((rand % 10000) * 100 < FRAME_ERROR) {
    frame[INFORMATION_HEADER_SIZE + 5] ^= ba;
}
llwrite(frame);
// Restaurar sanidade da trama
frame[INFORMATION_HEADER_SIZE + 5] ^= ba;
```

Ao introduzir aleatoriamente geração do *flip* de um byte de dados da trama de informação, observou-se que quanto maior o valor do frame error ratio, o programa vai estar durante mais tempo a reenviar tramas uma vez que o recetor envia um sinal de rejeição mais frequentemente.

É interessante notar que os valores tendem a aumentar muito nos primeiros 10%, sendo a taxa de aumento reduzida entre mais 10 e 50% e depois mais acentuada até aos 75%.

8.2. Variar tempo de propagação

De um modo geral, a aumentar o tempo de propagação a eficiência sofre uma penalização, uma vez que fica a demorar mais um pacote chegar ao outro lado.

Contudo, pela figura 4, observa-se que aumentos de 0.5 não apresentam alterações significativas de perda de eficiência, sendo ao contrário para aumentos de mais do que 1 segundo.

8.3. Variar capacidade de ligação

Ao aumentar a capacidade de ligação, aumentamos o número de bits por segundo que conseguem estar a ser transmitidos através da porta série, o que significa que até um certo limite, ao aumentarmos a capacidade de ligação vamos estar a aumentar a performance.

Para valores abaixo ou equivalente a 1200 de baudrate, o programa não corre nas devidas condições devido ao tempo que um pacote vai demorar a ser enviado.

8.4. Variar tamanho das tramas

Quanto maior o número de tramas menor será o número de transmissões, o que significa que num contexto em que não ocorre erros a transmissão demorará menos tempo, apesar de que entre os valores testados a diferença não é assim tão significativa, como pode ser observado na figura 2.

9. Conclusões

Em suma, o nosso programa tem duas camadas principais, isoladas entre si em que a de ligação de dados expõe uma API para a camada da aplicação, de modo a que consigamos efetuar a transferência de um ficheiro de uma porta série para outra.

Para além disso, em termos de aprendizagem, este trabalho foi bastante útil para uma melhor compreensão acerca de *byte stuffing*, assim como o mecanismo de transmissão e controlo de erros `Stop and Wait`, tal como uma primeira experiência real com um dispositivo físico onde nem tudo funciona tão bem como num meio virtual.

10. Anexos

10.1. Código

Encontra-se na paste `/code`.

10.2. Figuras

FER (%)	Tempo (s)	Bitrate (Bit/s)	Eficiência (%)
0	12.964149	6768.203605	70.50
10	16.108527	5447.0529	56.74
20	17.303611	5070.849084	52.82
25	18.497586	4743.537886	49.41
50	19.521349	4494.771339	46.82
75	23.270911	3770.544264	39.27

Table 1: Eficiência por Frame Error Ratio

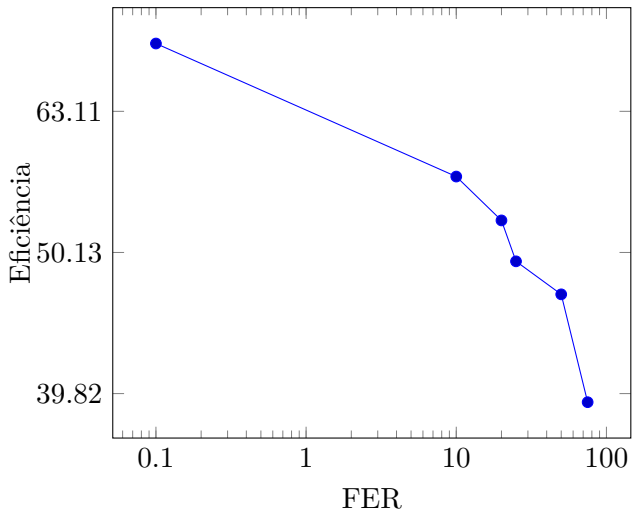

Figure 1: Eficiência / Frame Error Ratio

Tamanho dos pacotes (*Bytes*)	Tempo (*s*)	Bitrate (Bit/s)	Eficiência (%)
512	13.043	6727.203605	70.08
700	12.919	6791.856955	70.75
1024	12.864	6820.895522	71.05
2048	12.531	7002.154656	72.94

Table 2: Eficiência tamanho do pacote

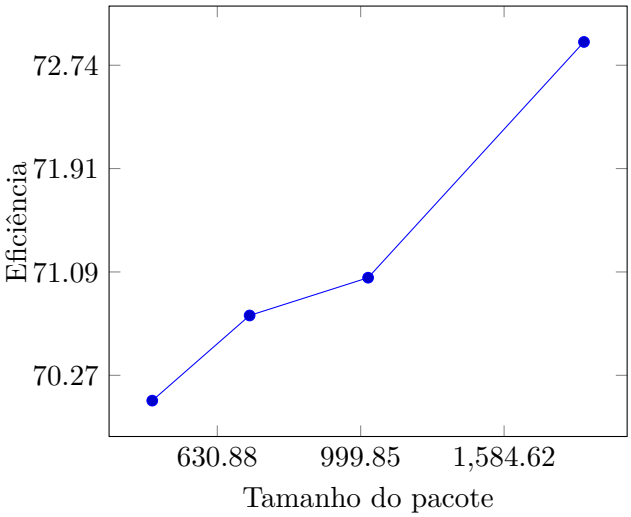

Figure 2: Eficiência / Tamanho do pacote

Baudrate ($Baud/s$)	Tempo (s)	Bitrate (Bit/s)	Eficiência (%)
2400	14.024969	6256.27051	56.17
9600	12.864	6820.895522	71.05
38600	11.693	7503.78550	78.16
115200	11.670	7518.93156	78.32
256000	11.687	7507.82921	78.21

Table 3: Eficiência por baud rate

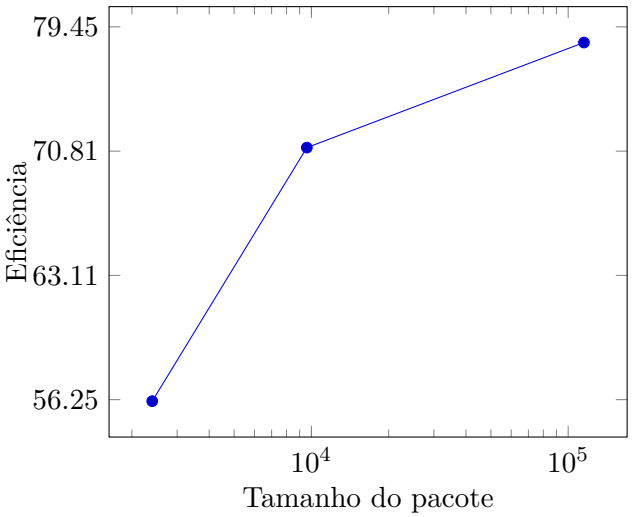

Figure 3: Eficiência / Baudrate

TProp (s)	Tempo (s)	Bitrate (Bit/s)	Eficiência (%)
0	12.864	6820.8955	71.05
0.2	13.0124	6742.1066	70.02
0.5	13.2100	6642.2407	69.19
1	14.916138	5882.58781	61.28
1.5	14.921172	5880.50322	61.26
2	26.008912	3373.61286	35.14

Table 4: Eficiência por tempo de propagação

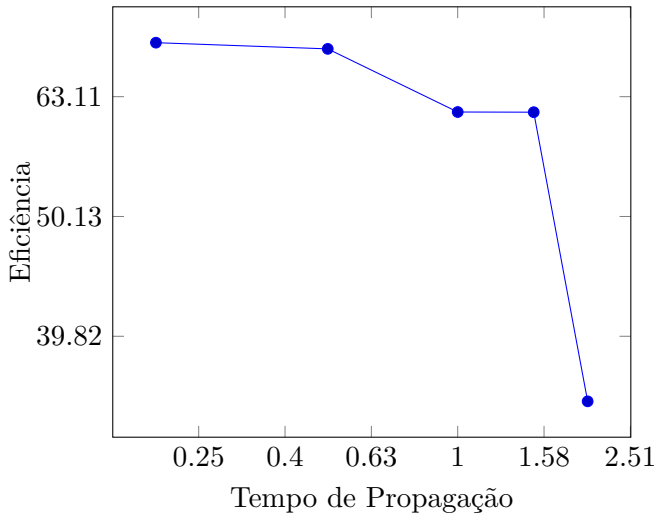

Figure 4: Eficiência / Tempo de Propagação