

Relatório Final



Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Grupo:

Duarte Moraes - ei12176
João Isaías - up201305893
Pedro Carvalho - up201306506

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

Índice

1. Índice
2. Sumário:
3. Arquitetura
4. Estrutura do código
 - a. Ligação de Dados
 - b. Aplicação
5. Protocolo de Ligação Lógica
 - a. Stuffing
 - b. BCC
 - c. Máquinas de Estado
 - d. Envio de RR/RREJ
6. Protocolo de Aplicação
 - a. Empacotar e Desempacotar Pacotes de Dados
 - b. Empacotar e Desempacotar Pacotes de Controlo
 - c. Chunking
7. Validação
 - a. Primeiro teste - Ligação Normal
 - b. Segundo e terceiro teste - Ligação com a porta de série retirada e ligação com erros
8. Elementos de Valorização
9. Anexo - Código fonte

Sumário:

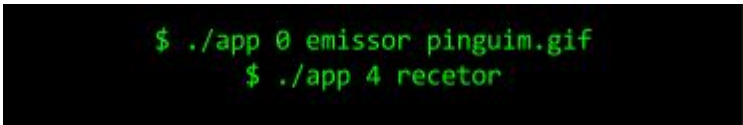
No âmbito da unidade curricular de Redes de Computadores foi-nos pedido o desenvolvimento de uma aplicação de transferência de dados. Aplicando o conhecimento das aulas teóricas, realçamos que a elaboração deste trabalho destaca a *Application Layer*, a *Data Link Layer* e *Physical Layer*. Dito isto, o trabalho baseia-se em desenvolver uma aplicação de transferência de ficheiros entre dois computadores através das portas de série de cada máquina.

Com este relatório, pretendemos não só demonstrar o processo a transferência de dados, como a garantia de uma transferência sem fugas.

Arquitetura

O trabalho divide-se em dois grupos independentes de funções. Por um lado temos funções dedicadas à camada de ligação de dados, estas estão descritas e definidas nos ficheiros `linklayer.c` e `h`, e no ficheiro `aux.h`. A outra parte simula uma aplicação de transferência de ficheiros encontra-se definida nos ficheiros `appfunc.c` e `h`, e `app.c`.

A interface com o utilizador ocorre ao nível da aplicação. Este tem de enviar argumentos na linha de comandos que indiquem se a aplicação é do tipo utilizador emissor ou utilizador recetor de ficheiros, qual o numero da porta de série da máquina por onde se faz a comunicação, e no caso do emissor, qual o caminho para o ficheiro a enviar.



```
$ ./app 0 emissor pinguim.gif
$ ./app 4 recetor
```

Durante o restante funcionamento da aplicação, apenas é mostrada a informação ao utilizador, como o progresso do envio/recepção dos pacotes, tamanho do ficheiro, nome do ficheiro esperado, outros. Não é necessário mais input.

Está incluído também uma Makefile para a compilação do código.

Estrutura do código

a) Ligação de Dados

A ligação de dados tem 4 funções principais a serem usadas pela aplicação. Estas são as que se apresentam no ficheiro `linkfunc.h`: `int llopen()`; `int llclose()`; `int llwrite()`; `int llread()`; Estas utilizam a estrutura de dados única `LinkLayer{}` para guardar informações.

- llopen():

Tem dois objectivos. **1º** Procura a porta de série escolhida na máquina e abre-a, retornando um descritor de ficheiro correspondente. Encontra-se encapsulada em `initialize()`. **2º** Envia ou recebe, consoante o tipo de utilizador, através da porta de série, tramas de comandos `SET` ou `UA`. As duas funções `fazer_trama_supervisao()`; `fazer_trama_resposta()`; fazem a construção das tramas, as duas funções `envia_e_espera_superv()`;

espera_e_responde_superv(); implementam máquinas de estado que assimilam os seus envios e verificações de receção.

- llclose():

Tem também dois objectivos. 1º Com as mesmas funções que para *llopen()*, constroi tramas de supervisão e envia/recebe-as e verifica-as. 2º Fecha os descritores de ficheiros abertos em *initialize()*, desta vez por via da função *finalize()*.

- llwrite():

Recebe um endereço de memória, dados, e envia pela porta de série. Estes dados tem que ser tratados antes de ser enviados por duas operações: O **Stuffing** é necessário para remover potencial ambiguidade entre os dados e informação na trama -

byte_stuffing_encode(); O **Framing** dos dados é o acrescentar no início e no final bytes de comando das tramas que ajudem o receptor a identificar os dados de uma trama de outras, e a eventualidade de erros - *Fazer_trama()*; . Após fazer o envio, o *llread()* corre uma máquina de estados que fique à espera de uma resposta, re-enviando a trama de dados se tal não ocorrer ou a resposta for de rejeição.

- llread():

O *llread()* é a função irmã do *llwrite()*. Espera e recebe dados enviados pela porta de dados, e aplica-lhes as operações inversas: '**Destuffing**' e '**Deframing**' - *Desfazer_trama()*; *de_stuffing()*; . Se a verificação dos dados (BCC) for correcta, então o *llread()* envia uma trama de supervisão do tipo **RR**, caso contrário, a de rejeição **RREJ** - *enviar_RR_REJ()*.

b) Aplicação

A aplicação corre a maioria da sua lógica interna partir de duas funções principais: *int Logic_Emissor()*; *int Logic_Recetor()*; Estas duas efectuam a totalidade das operações *llread()* e *llwrite()* da aplicação. Existem operações comuns antes e após efectuar a lógica, correspondentes ao 1º objectivo de *llopen()* e ao 2º de *llclose()*. O main de app portanto chama sequencialmente *llread()*, *Logic_()*, *llopen()*. A estrutura onde se guardam os dados é a *applicationLayer{}*;

Logic_Emissor() e *Logic_Recetor()* são funções irmãs com três tempos de funcionalidade. O primeiro e o último tempo correspondem ao envio e recepção de pacotes de comando. O pacote de comando transmite duas informações: O tamanho e o nome do ficheiro a enviar. Este é construído com a função *packup_control()*; e verificado por *unpack_control()*; . O segundo tempo de função é o envio sequencial de todos os dados do ficheiro. Os ficheiros são enviados por múltiplos pacotes, sendo que a cada pedaço dos dados chama-se Chunk, e a operação que separa os dados '**Chunking**'. No lado do emissor é necessário então chamar para cada pacote a função *get_chunk()*; . Os pacotes de dados, por sua vez, são construídos e desempacotados pelas funções *packup_data()*; *unpack_data()*;

Protocolo de Ligação Lógica

Os elementos da camada de ligação implementam um esquema do tipo **Stop and Wait**. A informação é transmitida uma trama de cada vez sendo esperado o reconhecimento da trama por parte do receptor. Nas três situações em que tal não ocorre: Envio incorrecto, Resposta Incorrecta ou Timeout, a informação anterior tem de voltar a ser transmitida.

Stuffing

Como já foi mencionado, é feito o Framing e o Stuffing dos dados. A flag de inicio e fim das tramas é a 0x7E, o stuffing é feito por bytes, e o byte de conversão da flag é o 0x7D.

```
int byte_stuffing_encode(char * trama, char * res, int size){
    int i, j=0;
    int count = 0;

    for(i = 0; i < size; i++, j++)
    {
        if (trama[i] == 0x7E)
        {
            res[j] = 0x7D;
            j++; count++;
            res[j] = 0x5E;
        }
        else if(trama[i] == 0x7D)
        {
            res[j] = 0x7D;
            j++; count++;
            res[j] = 0x5D;
        }
        else{
            res[j] = trama[i];
        }
    }
    return count;
}

int de_stuffing(char * trama, char * res, int size){
    int i, j=0;
    int count = 0;

    for(i = 0; i < size; i++, j++)
    {
        if (trama[i] == 0x7D && trama[i+1] == 0x5E)
        {
            res[j] = 0x7E;
            i++; count++;
        }
        else if(trama[i] == 0x7D && trama[i+1] == 0x5D)
        {
            res[j] = 0x7D;
            i++; count++;
        }
        else
        {
            res[j] = trama[i];
        }
    }
    return count;
}
```

BCC

Os BCC - **Block Check Character** - são bytes incluídos nas tramas que ajudam a proteger o envio de erros aleatórios que possam aparecer. Aqui, dois BCC são usados, formados por operação XOR de cada um dos bits em cada elemento dos bytes nas tramas. O BCC1 para os bytes de comando, o BCC2 para os bytes de dados. Na imagem vemos a verificação do BCC2 em lread().

```
char bcc = dados_destuffed[0];
int i;
for (i = 1; i < tamanho_destuffed; i++)
    bcc = (bcc ^ dados_destuffed[i]);

if (bcc != BCC2)
{
    return -1;
}
```

Máquinas de Estado

```
int espera_e_responde_superv(int port, char * msg, char * res){  
    unsigned char pak;  
    int state = 0;  
    while (STOP==FALSE)  
    {  
        read(port,&pak,1);  
  
        switch (state)  
        {  
            case 0: //Espera FLAG - F  
                if (pak == msg[0])  
                {  
                    state++;  
                }  
                break;  
            case 1: //Espera Endreço - A  
                if (pak == msg[1])  
                    state++;  
                else if (pak == msg[0])  
                    ;  
                else  
                    state = 0;  
                break;  
            case 2: // Espera Controlo - C  
                if (pak == msg[2])  
                    state++;  
                else if (pak == msg[0])  
                    state = 1;  
                else  
                    state = 0;  
                break;  
            case 3: // Espera de BCC  
                if (pak == msg[3] )  
                    state++;  
                else if (pak == msg[0])  
                    state = 1;  
                else  
                    state = 0;  
                break;  
            case 4: // Espera Flag - F  
                if (pak == msg[4])  
                {  
                    STOP = TRUE;  
                }  
                else  
                    state = 0;  
                break;  
        }  
    }  
}
```

As máquinas de estados, no exemplo uma máquina de leitura de tramas de supervisão, lêem os bytes provenientes da porta de série um a um e avançam nos estados apenas se o byte recebido corresponder a um byte do tipo esperado para esse estado. Se um aparecer fora de ordem, a máquina recomeça.

Envio de RR/RREJ

Os RR (Receiver Ready) e RREJ (Receiver Rejection) são tramas de comando de “acknowledgement” enviadas pelo recetor ao emissor. Estas avisam o emissor do sucesso ou insucesso da ultima tentativa de envio de trama.

```

void enviar_RR_REJ(int sucesso){
    char trama_resposta[5];
    if (sucesso == 0)
    {
        if (Linkdata.ALTERNATING == 0) Linkdata.ALTERNATING = 1; else Linkdata.ALTERNATING = 0;
        fazer_trama_supervisao(trama_resposta, TYPE_RR, EMISSOR, Linkdata.ALTERNATING);
    }
    else
    {
        fazer_trama_supervisao(trama_resposta, TYPE_REJ, EMISSOR, Linkdata.ALTERNATING);
    }

    write(Linkdata.portfd, trama_resposta, 5);
}

```

Protocolo de Aplicação

Os pacotes são estruturas que as aplicações usam para comunicar umas com as outras, independentes do modo de funcionamento da camada de ligação de dados. Estes pacotes são cabeçalhos de bytes que se incorporam como prefixos aos dados, ou são unicos (de comando) e enviam mensagens.

Empacotar e Desempacotar Pacotes de Dados

Os pacotes de dados devem guardar o tamanho em bytes de dados que vão enviar. Esse tamanho é descrito por dois bytes L1 e L2 , tal que o tamanho= $L1 \cdot 256 + L2$. Os pacotes de dados também tem um numero de sequencia que provoca erro no receptor se ocorrer envio fora de ordem.

unpack_data

```

int unpack_data(char * res, uint8_t n_seq, char * data)
{
    if(data[0] != 0x00)
    {
        printf("unpack_data(): This wasn't a Data Packet\n");
        return -1;
    }
    if( (unsigned char) data[1] != n_seq)
    {
        printf("unpack_data(): Wrong sequence number\n");
        return -1;
    }

    int read_size = 256 * data[2] + data[3];

    memcpy(&res[0], &data[4], read_size);

    return 0;
}

```

packup_data

```

int packup_data(char * res, int n_seq, char * data, int data_size)
{
    if(data_size > 512)
    {
        printf("packup_data(): Data read after stuffing was larger than maximum 512 byte\n");
        return -1;
    }

    int L1 = data_size % 256;
    int L2 = data_size / 256;

    int i = 0;
    for(i=0; i < 4; i++){
        switch (i)
        {
            case 0: //C
                res[i] = 0;
                break;
            case 1: //N
                res[i] = n_seq;
                break;
            case 2: //L1
                res[i] = (unsigned char) L2;
                break;
            case 3: //L2
                res[i] = (unsigned char) L1;
                break;
        }
    }

    memcpy(&res[4], &data[0], data_size);
    return data_size+4;
}

```

Empacotar e Desempacotar Pacotes de Controlo

O pacote de dados envia duas informações: Tamanho do ficheiro em pacotes, e nome do ficheiro a ser enviado. O tamanho do ficheiro é também descrito pela função $\text{tamanho} = 256 * V1 + V2$.

unpack_control

```
int unpack_control(char * pak, int command, char * file_name)
{
    if (command != pak[0] )
    {
        printf("unpack_control(): Wrong expected C value\n");
        return -1;
    }

    int pack_amount = 256 * (uint8_t) pak[3] + (uint8_t) pak[4];

    int str_length = (uint8_t) pak[6];

    memcpy(file_name, &pak[7], str_length);

    if (strlen(file_name) != str_length)
    {
        printf("unpack_control(): String and its length don't match\n");
        return -1;
    }

    return pack_amount;
}
```

packup_control

```
int packup_control(char * res, int command)
{
    if (! (command == 1 || command == 2) )
    {
        printf("packup_control(): Invalid Command number, try 1 or 2\n");
        return -1;
    }
    res[0] = command;

    res[1] = 0; //T1 File size
    res[2] = 2; //T1 Amount of V1 octets
    res[3] = Appdata.total_number_packets / 256;
    res[4] = Appdata.total_number_packets % 256; //V1

    res[5] = 1; //T2 File size
    res[6] = strlen(Appdata.filename);

    memcpy(&res[7], Appdata.filename, strlen(Appdata.filename));

    res[7 + strlen(Appdata.filename)] = 0x00;
    res[8 + strlen(Appdata.filename)] = 0x00;

    return (7 + strlen(Appdata.filename));
}
```

Chunking

O chunking é uma operação bastante simples. Para um ficheiro, ler apenas um tamanho de bytes que a começar desde o índice onde terminou a última leitura, ou até se atingir o final do ficheiro a ser enviado. Esses bytes são o Chunk que pode ser empacotado pela aplicação.

```
int get_chunk(char * res, int chunk_size, int offset)
{
    fseek(Appdata.fileDescriptor, offset, SEEK_SET);
    size_t amount = chunk_size;
    if (fread(res, sizeof(char), amount, Appdata.fileDescriptor) == 0 )
        printf("file_to_buffer(): Erro a ler ficheiro \n");
    return amount;
}
```


Validação

Nesta secção são abordados os testes realizados com a nossa aplicação:

Primeiro teste - Ligação Normal

Execução do programa sem interrupção do envio dos dados.

Emissor:

```
User: 0
New termios structure set
llopen(): SUCESSO
bytes: 13568, chunks: 53
Emissor(): SUCESSO
llclose(): SUCESSO
gnu63:~/Desktop/trabalho#
```

Recetor:

```
User: 1
New termios structure set
llopen(): SUCESSO
N pacotes: 53, Nome Ficheiro: tux.gif
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Completo
Recetor(): SUCESSO
llclose(): SUCESSO
up201305893@linus80:~/Desktop/trabalho$
```

Segundo e terceiro teste - Ligação com a porta de série retirada e ligação com erros

Execução do programa mas com a remoção do do cabo que liga as portas de série a meio do envio e o encaixe de novo mais após um periodo de tempo. No terceiro teste, fez-se também a introdução de erros.

Emissor:

User: 0	User: 0
New termios structure set	New termios structure set
llopen(): SUCESSO	llopen(): SUCESSO
bytes: 13568, chunks: 53	bytes: 13568, chunks: 53
Ocorreu time out	Ocorreu time out
Ocorreu time out	Ocorreu time out
Ocorreu time out	Ocorreu time out
	Ocorreu time out
	Ocorreu time out
	Emissor(): SUCESSO
	llclose(): SUCESSO
	gnu63:~/Desktop/trabalho#

Recetor:

```
User: 1
New termios structure set
llopen(): SUCESSO
N pacotes: 53, Nome Ficheiro: tux.gif
|||||||||||||||||
User: 1
New termios structure set
llopen(): SUCESSO
N pacotes: 53, Nome Ficheiro: tux.gif
|||||||||||||||||
Completo
Recetor(): SUCESSO
llclose(): SUCESSO
up201305893@linus80:~/Desktop/trabalho$
```

Os resultados de ambos os testes são iguais. À esquerda de cada imagem está a fase em que se desliga o cabo. O emissor fica a avisar da ocorrência de timeouts enquanto o cabo não for colocado de novo. Entretanto o recetor simplesmente aguarda e recupera silenciosamente do erro quando for rencaixado o cabo.

Elementos de Valorização

O nosso único elemento de valorização é a comparação entre o tamanho do ficheiro recebido real e o valor indicado nos pacotes de controlo. No nosso código temos essa comparação aplicada na função `unpack_control()`.

Anexo - Código fonte

Os ficheiros do código estão incluídos no arquivo `FileTransfer.zip`