Protocolo de Ligação de Dados

Relatório do 1º Projeto de Redes de Computadores

Licenciatura em Engenharia Informática e Computação

Turma 7, Grupo 1

Ana Matilde Guedes Perez da Silva Barra (up20194795)
Sérgio Miguel Rosa Estêvão (up201905680)
Sérgio Rodrigues da Gama (up201906690)

11 de Dezembro 2021

Conteúdo

1 Sumário

No âmbito da disciplina de Redes de Computadores, do 3ºAno da Licenciatura em Engenharia Informática e Computadores, foi-nos proposto o desafio de elaborar uma aplicação apoiada num protocolo de ligação de dados, que realiza a transferência de ficheiros entre dois computadores, através de uma porta série assíncrona.

A realização deste projeto foi muito útil no entendimento de como as redes de computadores funcionam, dado que estas dependem de bons e eficientes protocolos. Este trabalho foi um desafio cativante, uma vez que foi possível observar, de forma relativamente direta, o impacto do código criado por nós na ligação estabelecida e as consequências de determinadas ações na visualização dos dados enviados.

2 Introdução

O objetivo deste trabalho era implementar um protocolo de ligação de dados, de modo a permitir uma comunicação estável entre dois computadores ligados através de uma porta série. Posteriormente, foi também necessário desenvolver um protocolo de aplicação para existir uma camada que simplifica a interação com as funções de baixo nível desenvolvidas para a transferência de dados.

Com este relatório pretendemos expor o trabalho desenvolvido ao longo das últimas semanas, numa perspetiva mais teórica, enfatizando os métodos utilizados para alcançar os objetivos propostos no guião do trabalho e também as conclusões alcançadas.

Para isso, este relatório está estruturado da seguinte forma:

- **Arquitetura** - Indicação dos blocos funcionais e interfaces.
- **Estrutura de código** - Exposição das *APIs*, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais** - Identificação dos casos de uso principais e das sequências de chamadas de funções a eles associadas.
- **Protocolo de ligação lógica** - Identificação dos principais aspetos funcionais deste protocolo e descrição da estratégia de implementação desses aspetos.
- **Protocolo de aplicação** - Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
- **Validação** - Descrição dos testes efetuados com apresentação quantificada dos resultados.

- **Eficiência do protocolo de ligação de dados** - Caraterização estatística da eficiência do protocolo desenvolvido, efetuada recorrendo a medidas sobre o código elaborado.
- **Conclusão** - Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura

Para a criação e testagem deste protocolo, o projeto dividiu-se em duas grandes vertentes: uma de baixo nível, na camada de ligação de dados e uma de alto nível, na camada de aplicação. Estas demonstram uma espécie de relação de dependência unidirecional, dado que pelo menos uma das vertentes, a camada de aplicação, depende diretamente do bom funcionamento da camada de mais baixo nível, ligação de dados.

Desta forma, a camada de ligação de dados trata de definir métodos para a abertura e fecho de uma conexão, como também métodos para o envio e receção de dados. Nestes métodos são utilizados todos os mecanismos necessários para que a transferência de dados ocorra com sucesso, como por exemplo *timeouts* adequados, *bytestuffing* e criação de cabeçalhos com dados de verificação de erros, entre outros.

Os métodos disponibilizados pela camada mais baixa (*llopen*, *llread*, *llwrite* e *llclose*) são depois utilizados por um recetor e um transmissor na camada da aplicação que enviam pacotes de dados entre eles. Por fim, também existem vários tipos de pacotes, com diferentes finalidades, definidos na camada da aplicação.

4 Estrutura de código

Para organizar o nosso código, optámos por dividir *header files* e *source files* por pastas diferentes, incluir pastas para os ficheiros de teste e *object files*.

4.1 Source (src)

Neste diretório encontram-se todos os ficheiros *source* do nosso projeto. Estão aqui presentes os ficheiros *api.c*, *api_receiver.c*, *api_transmitter.c*, *app_receiver.c*, *app_transmitter.c*, *buffer_utils.c*, *byte_stuffing.c*, *file.c*, *receiver.c*, *trasmitter.c*.

Nos ficheiros *receiver.c* e *transmitter.c* encontram-se as funções *main* dos dois modos de execução possíveis, **recetor** ou **transmissor**.

Nos ficheiros *app_receiver.c* e *app_transmitter.c* encontra-se a implementação da camada de aplicação.

```
// Funções principais de app_receiver.c e app_transmitter.c

//app_receiver.c
int checkControlPacket(enum packet_id id, unsigned char* packet);
unsigned long receiveStartPacket(int fd, unsigned char* name,
 unsigned int delay, unsigned int genErrors);

//app_transmitter.c
int sendControlPacket(int fd, enum packet_id id,
  ControlPacket control_p);
int sendDataPacket(int fd, FileInfo* file_info,
  unsigned int delay);
ControlPacket createControlPacket(FileInfo* file_info);
```

Nos ficheiros *api.c*, *api_receiver.c*, *api_transmitter.c* encontra-se a implementação da camada de ligação de dados (*llopen*, *llread*, *llwrite* e *llclose*).

```
// Funções principais de api.c, api_receiver.c e api_transmitter.c

//api.c
int llopen(char* port, enum status stat, int* fid);
int llread(int fd, unsigned char* buffer, unsigned int delay,
  unsigned int genErrors);
int llwrite(int fd, unsigned char* buffer, int length);
int llclose(int fd, enum status stat);

//Estrutura de Dados - Application Layer
typedef struct ApplicationLayer {
    int fileDescriptor; //file descriptor correspondant to the serial port
    enum status status; // TRANSMITTER or RECEIVER
} ApplicationLayer;

//api_receiver.c
int llopen_receiver(char* port, int* fid);
int llclose_receiver(int fd);

//Verificação de frames recebidas
int checkSETByteRecieved(unsigned char byte_recieved, int idx);
int checkDiscEByteRecieved(unsigned char byte_recieved, int idx);
int checkUA_E_ByteRecieved(unsigned char byte_recieved, int idx);
int checkDataFrame(unsigned char* frame, int Nr, int size);

void handleIFrameState(char c, int* state);

//api_transmitter.c
int llopen_transmitter(char* port, int* fid);
int llclose_transmitter(int fd);

//Verificação de frames recebidas
int checkRRByteRecieved(unsigned char* buf, int index, int Ns);
int isRej(unsigned char c, int index,  int Ns);
int checkUAByteRecieved(unsigned char byte_recieved, int idx);
int checkDiscRByteRecieved(unsigned char byte_recieved, int idx);
```

Por fim, nos ficheiros *buffer_utils.c*, *byte_stuffing.c*, *file.c* temos funções utilitárias para *print* de tramas recebidas na consola, **Byte Stuffing** e **Destuffing** e também para simplificar a interação com os ficheiros a transferir.

```
// Funções principais de buffer_utils.c, byte_stuffing.c e file.c

//buffer_utils.c
int writeData(int fd, unsigned char *buf, int size);
void printData(unsigned char *trama, int size, int read);
void clean_buf(void *buf, int size);

//byte_stuffing.c
int byteStuffing(size_t size, unsigned char data[],
  unsigned char* stuffed);
int reverseByteStuffing(int *size, unsigned char stuffed[],
  unsigned char* original);

//file.c
int get_file_size(FileInfo* fi, FILE * file);
int read_file(FileInfo* fi, FILE *file);
int fillInfo(FileInfo* fi, FILE *file, char *file_name);
unsigned char* dataChunk(unsigned char* data, int start_index,
  int quantity);
```

4.2 Includes

Neste diretório estão incluídos os *header files* que definem todas as funções do nosso programa e também o ficheiro *macrosLD.h* que define as *macros* e *enums* a utilizar.

4.3 Files

Nesta pasta existem dois subdiretórios, **transfer** e **receive**, que incluem, respetivamente, os ficheiros a enviar e os ficheiros recebidos.

4.4 Out

Pasta que contem todos *object files* criados pelo compilador a partir de cada *source*, que vão ser ligados para criar o executável final quando se faz *make* (que vai utilizar o *Makefile* criado por nós).

5 Casos de uso principais

A nossa aplicação recebe apenas input do utilizador na chamada inicial para execução:

- No caso de estar a ser executado o programa **transmitter**, é recebido como *input* o *nserial* da *serial port* a utilizar, o nome do ficheiro a enviar e o *nsecs* para ser causado um delay nos sleeps existentes no código.
- Por outro lado, estando a executar o programa **receiver**, é pedido como *input* o *nserial* da *serial port*, o *nsecs* para o delay e também um valor entre 0 a 1000 que vai ser a chance de erros a serem introduzidos na leitura (dividindo o valor por 10 obtém-se a percentagem).

As sequências de chamadas a funções numa execução típica de **transmitter** e **receiver** serão referidas nas duas secções seguintes.

6 Protocolo de ligação lógica

6.1 llopen()

Esta função é usada para estabelecer a conexão entre as duas partes, através de ***llopen_transmitter()*** ou ***llopen_receiver()***, consoante o programa que a invoca.

```
int llopen(char* port, enum status stat, int* fid){
    if(stat == TRANSMITTER){
       if ((strcmp("/dev/ttyS0", port)!=0) &&
         (strcmp("/dev/ttyS1", port)!=0) &&
         (strcmp("/dev/ttyS10", port)!=0)) {
           printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS10\n");
           return ERROR;
       }
       return llopen_transmitter(port, fid);
    }
    else if(stat == RECEIVER){
       if ((strcmp("/dev/ttyS0", port)!=0) &&
         (strcmp("/dev/ttyS1", port)!=0) &&
         (strcmp("/dev/ttyS11", port)!=0)){
           printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS11\n");
           return ERROR;
       }
       return llopen_receiver(port, fid);
    }
    else{
     perror("\nError: unknown status on llopen function call\n");
     return ERROR;
    }
}
```

O procedimento adotado para estabelecer a conexão foi o proposto, sendo que a função ***llopen_transmitter()*** envia uma trama de controlo (**SET**) e espera pela respetiva resposta (**UA**), enviada pela função ***llopen_receiver()***. Assim que esta última envia o (**UA**) e este é recebido, a conexão está estabelecida.

De modo a garantir a correta receção das tramas de controlo na sua generalidade, tanto na função ***llopen()*** como na ***llclose()***, foram desenvolvidas várias funções que se encarregam de verificar a estrutura destas primeiras, ao longo deste processo. Com efeito, não era viável colocar todos os *checks* aqui, seguindo-se o código seguinte, que verifica um **DISC** do emissor (a título de exemplo).

Com este exemplo podemos verificar o tipo de estrutura usado para a criação das tramas de controlo, que foi, de facto como o proposto.

```
int checkDiscEByteRecieved(unsigned char byte_recieved, int idx){
  int is_OK = FALSE;
  if((idx == 0 || idx == 4) && byte_recieved == FLAG){
    is_OK = TRUE;
  }
  else if (idx == 1 && byte_recieved == A_EE){
    is_OK = TRUE;
  }
  else if (idx == 2 && byte_recieved == C_DISC){
    is_OK = TRUE;
  }
  else if (idx == 3 && byte_recieved == BCC(A_EE, C_DISC)){
    is_OK = TRUE;
  }
  return is_OK;
}
```

Finalmente, foi também implementado um *timeout* nas funções do transmissor, tanto em ***llopen()*** como em ***llclose()***. Assim, quando o transmissor entra em espera ativa por uma resposta, espera apenas um determinado número de segundos até desistir e terminar a execução. Este período encontra-se definido como 5 segundos, já que com menos segundos era complicado de realizar certos testes.

Esta implementação dos *timeouts*, também está presente em ***llwrite()***, sendo o mecanismo adotado semelhante. Este passou por utilizar a *library signal.h* da linguagem c. Segue-se o seguinte excerto de código como demonstração.

```
...
signal(SIGALRM, atende);
siginterrupt(SIGALRM, 1);
...
if (connect_attempt > MAX_ATTEMPS){
    printf("Sender gave up, attempts exceded\n");
    return ERROR;
}
...
alarm(ALARM_SECONDS);
...
```

6.2 llclose()

Esta função está encarregue do fecho da conexão, sendo que para isso também foram desenvolvidas duas subfunções: ***llclose_transmitter()*** e ***llclose_receiver()***, sendo a primeira chamada pelo transmissor e a segunda pelo recetor. Deste modo, a função principal ***llclose()*** apresenta-se semelhante à ***llopen()***, tal como se pode ver no seguinte excerto de código.

```
int llclose(int fd, enum status stat){
    if(stat == TRANSMITTER){
        if(llclose_transmitter(fd) <0){
            printf("Failed to disconnect transmitter\n");
            return ERROR;
        }
```

```
        return 0;
    }
    else if(stat == RECEIVER){
        if(llclose_receiver(fd) < 0){
            printf("Failed to disconnect receiver\n");
            return ERROR;
        }
        return 0;
    }
    else{
        perror("\nError: unknown status on llclose function call\n");
        return ERROR;
    }
}
```

O procedimento que foi adotado para a desconexão também foi o proposto no guião. O transmissor, através da função ***llclose_transmitter()***, envia uma trama de controlo (**DISC**), que é recebida pelo recetor, na função ***llclose_receiver()***. Após a receção desta trama, esta última função envia também um **DISC**, recebido pelo transmissor, que envia um **UA** como resposta e fecha o seu programa. No final, ao receber esta trama, o recetor também fecha, terminando assim a conexão.

6.3 llwrite()

Esta função tem o papel de envio de dados, sendo que para isso, envia uma trama de informação, através de uma *system call* (*write()*), devidamente codificada (através de ***byte stuffing***), de forma a garantir que as suas ***FLAG's*** delimitadoras, não são confundidas com dados. De seguida, fica ativamente à espera de resposta do recetor. No caso de o recetor não enviar nenhuma resposta, a função acaba por terminar, devido ao *timeout*, explicado anteriormente.

A criação de tramas de informação segue também a estrutura proposta, sendo delimitadas por duas **FLAG**'s e compostas pelo campo do endereço **A**, campo de controlo **C** e os dois BCC's, **BCC1** após **C** e **BCC2** no final, antes da *flag* final.

```
    unsigned char* frame = (unsigned char*) malloc(I_FRAME_SIZE);

    frame[0] = FLAG;
    frame[1] = A_EE;

    if(!Ns){
        frame[2] = C_NS0;
    }else{
        frame[2] = C_NS1;
    }
    frame[3] = BCC(A_EE, frame[2]);

    unsigned short BCC2 = 0;

    for(int i = 0; i < length; i++){
        frame[4 + i] = buffer[i];
```

```
        BCC2 = buffer[i]^BCC2;
    }

    frame[length + 4] = BCC2;
    frame[length + 5] = FLAG;
```

6.4 llread()

Esta função tem o papel de receção de dados, ficando em espera ativa através de *system call*'s (read()). Após receção de uma trama completa de dados, esta função envia uma resposta ao transmissor, mediante o resultado das verificações efetuadas aos mesmos. Caso os dados estejam corrompidos a função envia um ***REJ***. Por outro lado, estando os dados corretos, a função envia um ***RR***.

Para a correta receção dos dados começámos por implementar uma ***state machine*** que verificava tudo o enviado entre duas ***FLAG***'s. Isto revelou-se não estar correto, pois era possível 'enganar' o recetor quando a conexão parava a meio. Assim, esta máquina de estados tornou-se mais complexa, no sentido em que não só verifica os campos das ***FLAG***'s, como também os restantes. Segue-se o código que ilustra este mecanismo.

```
    void handleIFrameState(char c, int* state){
        switch(*state){
            case 0:{
                if(c==FLAG) *state = 1;
                break;
            }
            case 1:{
                if(c==A_EE) *state = 2;
                else *state = 0;
                break;
            }
            case 2:{
                if(c==C_NS0 || c==C_NS1) *state = 3;
                else *state = 0;
                break;
            }
            case 3:{
                if(c==A_EE^C_NS0 || c==A_EE^C_NS1) *state = 4;
                else *state = 0;
                break;
            }
            case 4:{
                if(c==FLAG) *state = 5;
                break;
            }
        }
    }
```

É necessário referir que antes de efetuar qualquer verificação sobre os dados recebidos, é necessário descodificar os dados que tinham sido submetidos a *byte stuffing*.

7 Protocolo de aplicação

Esta parte do projeto visava a criação da camada de alto nível, o protocolo de aplicação. Assim, foi criado um conjunto de funções para manipular pacotes, que são a principal estrutura de transferência de dados a este nível de abstração.

7.1 Pacotes

No excerto de código seguinte podemos observar o encapsulamento dos pacotes, com informação sobre o seu tamanho, facilitando a sua receção.

```
typedef struct ControlPacket {
    unsigned char *packet;      // array to be sent
    unsigned int size;          // size of the array
} ControlPacket;
```

Existem efetivamente dois tipos de pacotes: controlo (*start* e *end packets*) e dados.

```
enum packet_id {START, DATA, END};
```

Ambos os pacotes começam por um byte de controlo com valor 1 para dados e 2 para *start* e *end packets*.

Os pacotes de dados possuem dois campos adicionais no cabeçalho: o número de sequência e o número de *bytes* no campo de dados. Estes guardam também o próprio campo de dados que pretendemos enviar.

Por outro lado, os pacotes de controlo são codificados através do formato TLV (*Type*, *Length*, *Value*). Desta forma incluímos dois conjuntos neste formato, o tamanho do ficheiro de dados total (T = 0x0) e o nome do ficheiro (T = 0x1). Estes são criados alocando dinamicamente memória consoante o espaço necessário, como é visível no excerto de código abaixo. É de notar que todos os pacotes de controlo criados pela função ***createControlPacket()*** são *start packets* por omissão, sendo apenas necessário trocar o byte inicial para criar um *end packet*.

```
...
control.packet = (unsigned char *)malloc(i + 1);
control.packet[i++] = CP_START;
control.packet = (unsigned char *)realloc(control.packet, (i + 1));
...
```

7.2 Transmissor

Uma vez criados os pacotes, é necessário enviá-los para um recetor. Com efeito, antes de iniciarem comunicação, o transmissor e o recetor invocam a

função ***llopen()*** do protocolo de ligação de lógica. Após um retorno com sucesso dessa função, podem prosseguir ao envio e receção de informação.

Após aberta a conexão, o transmissor utiliza a função ***sendControlPacket()*** para enviar os dados, tirando partido da função ***llwrite()*** para enviar o pacote de início com a informação necessária para o recetor começar a receção dos dados. Depois utiliza a função ***sendDataPacket()***, que chama a mesma função de baixo nível múltiplas vezes com partes sequenciais dos dados a enviar até o envio estar completo. Por fim, é enviado o pacote de controlo final através da primeira função referida, para sinalizar o final do envio.

```
...
sendControlPacket(fd, START, control_p);

sendDataPacket(fd, &file_info, delay);

sendControlPacket(fd, END, control_p);
...
```

7.3 Recetor

Do lado do recetor, é recebido e verificado o pacote de controlo inicial, de modo a obter o nome para o ficheiro a ser reconstruído. Com esse nome é criado e aberto o ficheiro para escrita. De seguida, é iniciado um *loop* de receção de dados, terminando apenas ao receber um pacote de controlo final. O recetor utiliza a função ***llread()*** do protocolo de ligação lógica, para ler os pacotes de dados e de controlo.

Quando termina o envio e receção de dados, é fechada a conexão de ambos os lados, através da função ***llclose()***.

```
while(NOT_END){

    clean_buf(packet, I_FRAME_SIZE);
    clean_buf(data, DATA_SIZE);

    llread(fd, packet, delay, genErrors);
    ...
```

8 Validação

Para validar o nosso protocolo realizámos vários testes com ficheiros de diversos tipos e tamanhos: *pinguim.gif* - (10.7 KB); *cat_flower.gif* (3.05 MB); *this_is_fine.gif* (198 KB); *risitas.gif* (837 KB); *dog_fire.jpeg* (10.5 KB); *cube.jpg* (62.1 KB) e *Mario.mp3* (42.6 KB).

Todos os testes que corremos com estes ficheiros terminaram na execução normal do programa, sendo verificável a correta visualização dos ficheiros.

9 Eficiência do protocolo de ligação de dados

Para testar a eficiência do protocolo foram analisados os tempos de execução do programa com diferentes chances de geração de erros na leitura de tramas de informação e com diferentes intervalos de propagação.

Com um **Baudrate** de 38400 e a enviar um total de 128 *bytes* de dados por *data packet*, foram obtidos os seguintes dados ao enviar um ficheiro de 10968 *bytes*:

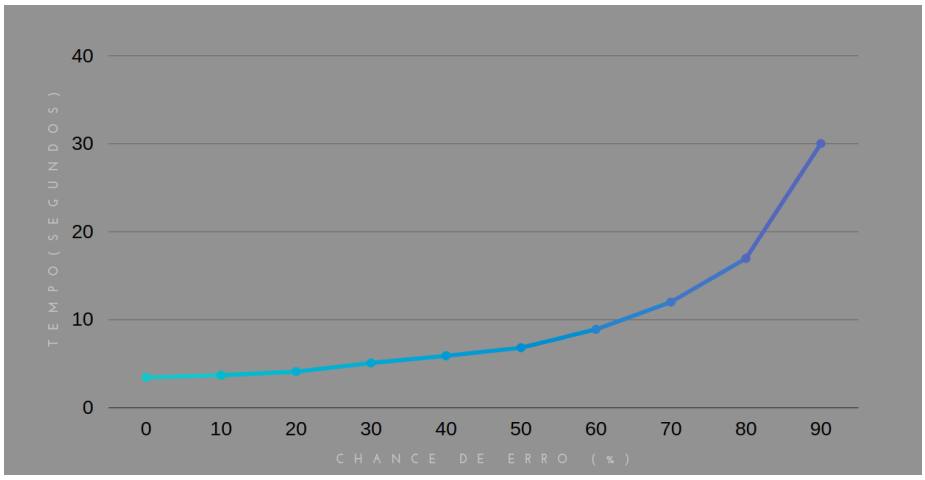

Fig 1 - Variação do tempo de execução - Chance de erro na trama de dados

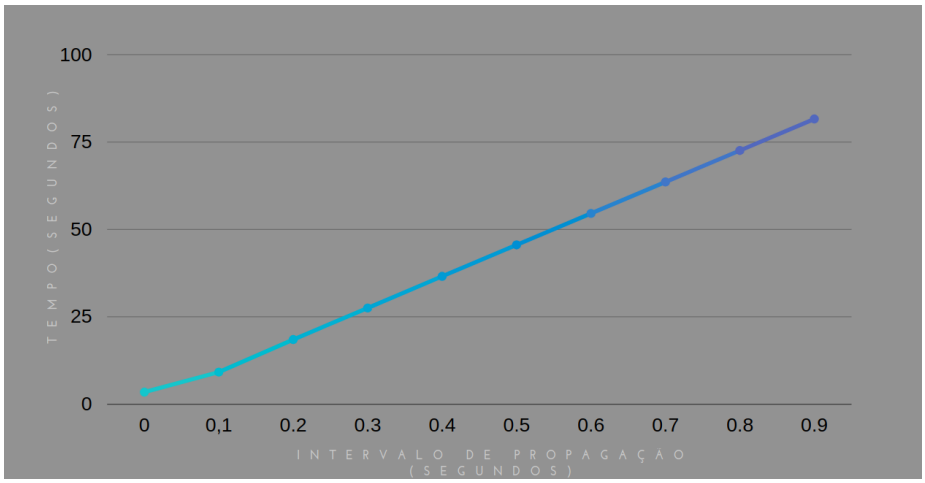

Fig 2 - Variação do tempo de execução - Aumento do tempo de propagação

10 Conclusão

Concluindo, neste projeto levámos a cabo o desenvolvimento de um protocolo de ligação de dados, do início ao fim, criando tanto as partes de baixo nível, com as de mais alto nível. Assim, tivemos de ultrapassar vários contratempos, como por exemplo a criação de uma *state machine* para a correta receção de tramas de informação, de modo a obter um produto final completamente funcional. Apesar disso, a experiência foi muito positiva e permitiu aprofundar o nosso conhecimento acerca do funcionamento destes protocolos e também de como transferência da dados é efetuada em geral.

Por fim, este foi um projeto crucial para o entendimento de como as redes de computadores funcionam, dado que estas dependem de protocolos bons e eficientes. O projeto tornou-se um verdadeiro desafio, ao tentar tornar o nosso protocolo o mais robusto possível. Por outro lado, também foi cativante, uma vez que efetuávamos experiências com ficheiros à nossa escolha e o *feedback* era bastante interativo.

11 Anexo I - Código fonte

O código fonte encontra-se no zip submetido em conjunto com este relatório.