ISEP – LEEC - Sistemas Computacionais

Ficha 6 – Comunicações com a API de *sockets*

Extraia o conteúdo do arquivo `ficha-sockets-ficheiros.zip` para um diretório à sua escolha.

Exercício 1

a) Analise os programas `clt_exemplo.c` e `srv_exemplo.c`.
   1) Que dados deverão ser passados através da linha de comandos (vetor `argv[]`) para cada um dos programas? (procure as ocorrências de `argv`)
   2) Que dados são trocados entre os dois programas?
   3) Que serviço é fornecido pelo servidor `srv_exemplo`?

b) Abra uma *shell* e compile os programas.

c) Execute o programa `srv_exemplo`. O número do porto a indicar na linha de comando deverá ser superior a 1023 (ver nota de rodapé[1]).

d) Abra uma nova *shell* e execute o programa `clt_exemplo`. Como o servidor (`srv_exemplo`) está a ser executado na mesma máquina, o cliente pode indicar o endereço IP 127.0.0.1 (ou *localhost*) para aceder ao servidor.

e) Altere o servidor, recorrendo ao mecanismo de criação de processos (função `fork`), de modo que seja possível atender simultaneamente ligações de vários clientes. Para tal, deverá criar um novo processo por cada ligação aceite. Evite a acumulação de processos *zombie* através da chamada a `signal(SIGCHLD, SIG_IGN)`.

Exercício 2

a) Implemente uma aplicação baseada no modelo cliente/servidor, com comunicação TCP/IP, em que o cliente faz o envio do conteúdo de um ficheiro, cujo nome é indicado através da linha de comando, para o servidor. O servidor, por sua vez, deverá guardar os dados recebidos (i.e., o conteúdo do ficheiro enviado pelo cliente) no ficheiro criado e aberto pela seguinte sequência de instruções:

```
char filename[11];
strcpy(filename, "fileXXXXXX");
int fd = mkstemp(filename);
```

   A função `mkstemp` cria um ficheiro, cujo nome é criado aleatoriamente com base no formato especificado em `filename`, e abre-o em modo de leitura e escrita. O valor de retorno do `mkstemp` é o respetivo descritor de ficheiro.

   Se pretender usar as funções de entrada/saída da biblioteca standard de C, pode aplicar a função `fdopen` ao descritor de ficheiro `fd`.

   Para implementar a leitura e escrita dos ficheiros, pode basear-se no programa fornecido no ficheiro `simple_clone.c`.

[1] Só o utilizador *root* tem permissão para fazer o *bind* aos portos 1 a 1023. Estes portos são reservados para serviços tipicamente encontrados nos servidores e estações de trabalho (servidor *Web,* serviço de sistema de ficheiros através da rede, etc.)

b) Altere o servidor, recorrendo ao mecanismo de criação de processos (função `fork`), de modo que seja possível receber simultaneamente ficheiros de vários clientes. Aplique a estratégia de criar um novo processo para cada ligação aceite. Do lado do servidor, faça uma chamada à função `sleep` antes de cada leitura, de modo a poder observar mais facilmente que é possível ter dois clientes a transferir ficheiros em simultâneo. Lance um cliente em cada terminal, especificando um ficheiro para envio com um tamanho que obrigue o servidor a executar várias operações de leitura.

Exercício 3 – Transmissão em formato de texto, com linhas terminadas em `'\n'`

Considere o seguinte protocolo de comunicação definido para uma aplicação baseada no modelo cliente/servidor, com comunicação sobre TCP/IP:

– Após o estabelecimento da ligação, o cliente deverá enviar para o servidor a representação em modo de texto de um número de estudante no ISEP, terminada com '\n', seguida duma mensagem introduzida pelo utilizador, que também deverá estar terminada com '\n'.
Esta sequência de caracteres deverá ser enviada em não mais do que 5 segundos após o estabelecimento de ligação. Após esse período, o servidor interromperá a ligação, caso esta ainda esteja ativa, não enviando qualquer resposta.
– No caso de receber um pedido válido, o servidor deverá responder através de duas linhas de texto. A primeira linha de texto da resposta do servidor deverá ser a conversão para maiúsculas da segunda linha de texto enviada pelo cliente. A segunda linha deverá ser o nome do estudante, exatamente como apresentado no portal do ISEP.
– Em nenhum dos casos deverá ser transmitido o carácter `'\0'` (indicador de fim de *string* da linguagem C).

Ou seja, num cenário em que um estudante com o nome "Jorge Estrela da Silva" e número "1230001" introduz o texto `"A minha mensagem!\n"`, o cliente deverá enviar a sequência

```
1230001\nA minha mensagem!\n
```

e o servidor deverá responder com

```
A MINHA MENSAGEM!\nJorge Estrela da Silva\n
```

a) ´Pretende-se implementar um programa (`ex3_clt.c`) que se comporte como cliente desta aplicação, tendo em atenção as seguintes especificações adicionais:

– O número de estudante a enviar deverá ser o seu. O número deverá estar codificado no próprio programa ("*hard coded*"). Não deverá ser passado como argumento de linha de comando nem solicitado pelo programa.
– A mensagem a ser enviada ao servidor, após o número do estudante, deverá ser lida do dispositivo standard de entrada (*stdin*) e deverá ter um tamanho inferior a 2000 bytes.
– Adicionalmente, antes de terminar, o programa deverá fazer a seguinte impressão no ecrã:

```
printf("Mensagem: %s\n""Nome: %s\n""Total: %d\n", linha1, linha2, nbytes);
```

onde `linha1` e `linha2` correspondem às duas linhas de texto da resposta do servidor e `nbytes` indica o número de bytes da resposta do servidor.
– O endereço IP e o porto do servidor deverão ser passados como argumentos de linha de comando.

Implemente o programa usando um dos métodos descritos abaixo. Teste o programa implementado usando o servidor disponível no porto **2000** do endereço **193.136.63.4**. Tenha em atenção que esta versão do servidor responde sempre com `"Instituto Superior de Engenharia do Porto\n"` na segunda linha, independentemente do número de estudante especificado. Tenha também em conta que os portos **2000** do endereço **193.136.63.4** apenas está acessível a partir de máquinas ligadas à rede do ISEP (o mesmo se aplicando aos portos 2001 e 2002, mencionados nos exercícios seguintes).

Sugestões de resolução:

A leitura de cada linha pode ser feita recorrendo a uma função tal como a apresentada abaixo:

```
int myReadLine1(int s, char *buf, int count){
  int r, n=0;

  if(count<=0)
    return 0;
  else if(count==1) {
      buf[0] = 0;
      return 0;
    } else
      --count; //leave space for '\0'

  do {
    r = read(s, buf+n, 1);
    if(r==1)
      ++n;
  } while( r==1 && n < count && buf[n-1]!='\n');

  buf[n] = '\0';

  return n;
}
```

Contudo, convém frisar que a utilização da função `read` para ler dados byte a byte é um procedimento ineficiente do ponto de vista de utilização de processador, uma vez que cada chamada à função `read` envolve uma chamada ao *kernel* do sistema operativo.

De seguida apresenta-se uma implementação alternativa (função `myReadLine2`). Neste caso, cada chamada à função `read` tenta ler vários bytes, de modo a minimizar o número de chamadas à função `read`. Desta forma, os dados podem ser analisados na mesma byte a byte (para detetar o fim de linha), mas esse processamento é feito ao nível do processo. Contudo, desta forma, cada chamada à função `read` pode ler dados correspondentes a várias linhas de texto. Para que esses dados possam ser processados nas chamadas seguintes à função `myReadLine2`, ficam armazenados num *buffer* persistente entre chamadas à função (observe que as variáveis `lbuf` e `lbuf_n` são declaradas como *static*).

```
int myReadLine2(int s, char *buf, int count) {
  static int lbuf_n = 0; //number of bytes stored in lbuf
  static char lbuf[4096];
  int nproc, n = 0;

  if(count<=0)
    return 0;
  else if(count==1) {
      buf[0] = 0;
      return 0;
    } else
      --count; //leave space for '\0'

  do {
    nproc=0;
    if(lbuf_n) {
      do {
        buf[n] = lbuf[nproc];
        ++n;
        ++nproc;
      } while( nproc!=lbuf_n && n < count && buf[n-1]!='\n');

      lbuf_n = lbuf_n - nproc;
      //move the remaining bytes in lbuf to the beginning
      if(lbuf_n)
        memcpy(lbuf, lbuf+nproc, lbuf_n);

      if( n == count || buf[n-1]=='\n' )
        break;
    }

    lbuf_n = read(s, lbuf, sizeof(lbuf));

  } while(lbuf_n>0);

  buf[n] = 0;

  return n;
}
```

A função `myReadLine2` apresenta, contudo, uma limitação que condiciona o seu uso noutras aplicações: não pode ser usada para fazer leituras alternadamente de múltiplas fontes de dados no mesmo processo, uma vez que os dados ficariam misturados no *buffer* local (variável *static* `lbuf`). Naturalmente essa limitação poderia ser contornada recorrendo a utilização de um *buffer* individual para cada ficheiro aberto, que é precisamente o que é feito nas funções da biblioteca standard do C.

Nesse sentido, como terceira alternativa, é possível recorrer à função `fdopen` e usar a função `fgets`, tal como exemplificado no seguinte extrato de código do servidor:

```
ns = accept(s, &clt_addr, &addrlen);
(…)
  FILE *fp = fdopen(ns, "r+");
  char *r = fgets(buffer, sizeof(buffer), fp);
  if(r==NULL) {
    printf("Connection broken\n");
    //...
```

A função `fgets`, de modo análogo à função `myReadLine2`, usa também um *buffer* intermédio com o objetivo de minimizar o número de chamadas à função `read`. Dessa forma, tem uma eficiência semelhante à da função `myReadline2`, mas não padece das suas limitações.

b) Implemente o servidor (`ex3_srv.c`), com a capacidade de atender vários clientes em simultâneo. Teste a comunicação entre os dois programas que implementou.

Exercício 4 - Transmissão de sequências de bytes de comprimento conhecido

Considere os seguintes tipos de dados:

```
typedef struct {
  char student_id[7];
  char text[2000]; //should be '\0' terminated
} msg1_t;

typedef struct {
  char text[2000]; //should be '\0' terminated
  char student_name[100]; //should be '\0' terminated
} msg2_t;
```

Pretende-se implementar uma aplicação cliente/servidor com funcionalidade idêntica à do exercício anterior, também baseada em *sockets* AF_INET/SOCK_STREAM, mas recorrendo a um diferente protocolo de aplicação.

Após o estabelecimento de uma ligação, o cliente deverá enviar ao servidor uma sequência de bytes correspondente à representação em memória de uma variável do tipo `msg1_t`. Em resposta, o servidor deverá enviar ao cliente uma sequência de bytes correspondente à representação em memória de uma variável do tipo `msg2_t`. Os campos dessas variáveis deverão conter os dados descritos no exercício anterior:

- Campo `student_id` de `msg1_t`: número de estudante
- Campo `text` de `msg1_t`: linha de texto introduzida pelo utilizador
- Campo `text` de `msg2_t`: linha de texto introduzida pelo utilizador convertida para maiúsculas
- Campo `student_name` de `msg2_t`: nome de estudante conforme portal

a) Escreva o cliente da aplicação (`ex4_clt.c`) recorrendo a um dos métodos sugeridos abaixo. Teste o programa implementado usando o servidor disponível no porto **2001** do endereço **193.136.63.4**.

Sugestões de resolução:

Uma vez que os dados são transmitidos em blocos de tamanho fixo (`sizeof(msg1_t)` e `sizeof(msg2_t)`), a estratégia de leitura de dados deverá ser adaptada para tal. De forma análoga ao Exercício 3, considere as abordagens seguintes:

– Chamadas sucessivas à função `read` até concluir a leitura do número de bytes solicitado. Em cada chamada à função `read` é solicitado o número de bytes ainda em falta. Note-se que, neste caso, não há necessidade de fazer leitura byte a byte, uma vez que não é necessário detetar nenhum carácter especial. Desta forma, é possível implementar de forma eficiente uma função que, ao contrário da função `read`, irá aguardar até que sejam recebidos exatamente `count` bytes (ou que a ligação seja terminada):

```
int myReadBlock(int s, void *buf, int count) {
  int r, nread = 0;

  while( nread < count ) {
    r = read(s, buf+nread, count-nread);
    if(r <= 0)
      break;
    else
      nread += r;
  }

  return nread;
}
```

– Alternativamente, é possível obter um comportamento semelhante ao da função `myReadBlock` recorrendo à função `fread` da biblioteca standard do C. Por exemplo:

```
msg2_t msg;
FILE * fp = fdopen(s, "r+");
int nbytes = fread(&msg, 1, sizeof(msg2_t), fp);
```

b) Implemente o servidor desta aplicação (`ex4_srv.c`). Teste a comunicação entre os dois programas que implementou.

Exercício 5

Considere uma versão alternativa da aplicação cliente/servidor descrita no exercício anterior em que as variáveis do tipo `msg1_t` são enviadas no seguinte formato:

– Campo `student_id` é enviado como uma sequência de sete caracteres, sem qualquer tipo de terminação.

– O envio da *string* contida no campo `text` deve ser precedida de um valor, representado em modo de texto e terminada em '\n', correspondente ao número de bytes dessa *string*, (ou seja, uma linha de texto a indicar o número de bytes que serão enviados de seguida).

– Envio dos caracteres da *string* contida em `text`. Sublinhe-se que, ao contrário do que acontece no Exercício 3, neste caso não existe nenhuma garantia de que a sequência de caracteres termine com `'\n'` ou qualquer outro tipo de delimitador (inclusive o `'\0'`). O número de bytes a ler deverá ser o indicado na linha de texto que precede o envio da *string* propriamente dita.

No caso das variáveis do tipo `msg2_t`, ambos os campos devem ser enviados do modo descrito para o campo `text` das variáveis do tipo `msg1_t` (linha de texto a indicar o número de bytes da *string*, seguida dos caracteres da *string*).

Ou seja, num cenário em que um estudante com o nome "Jorge Estrela da Silva" e número "1230001" introduz o texto `"A minha mensagem!\n"`, o cliente deverá enviar a sequência

```
123000118\nA minha mensagem!\n
```

e o servidor deverá responder com

```
18\nA MINHA MENSAGEM!\n22\nJorge Estrela da Silva
```

a) Escreva o cliente da aplicação (`ex5_clt.c`) recorrendo a um dos métodos sugeridos abaixo. Teste o programa implementado usando o servidor disponível no porto **2002** do endereço **193.136.63.4**.

Sugestões de resolução:

É possível fazer a receção dos dados referentes aos campos `text` e `student_name` recorrendo à função my`ReadLine1` (para ler a linha com o número de bytes da *string*) seguida uma chamada à função `myReadBlock`, para ler o número de bytes indicado.

Neste caso, não é possível recorrer à à função `myReadLine2`, pois haveria o risco de esta ler antecipadamente, para o seu *buffer* interno (`lbuf`), dados que deveriam ser lidos pela função `myReadBlock`. Para evitar este problema, as variáveis `lbuf` e `lbuf_n` teriam que passar a ser variáveis acessíveis a ambas as funções (passadas como argumentos em ambas as funções, por exemplo).

No caso da biblioteca standard C, as funções de E/S partilham o mesmo *buffer*, logo não há problema em alternar o uso das funções:

```
msg2_t msg2;
char txt_nbytes[16];
FILE * fp = fdopen(s, "r+");
char * r = fgets(txt_nbytes, sizeof(txt_nbytes), fp);
if(r==NULL) {
  return 1;
}
int nbytes = atoi(txt_nbytes);
if(nbytes > sizeof(msg2.text)-1){
  nbytes = sizeof(msg2.text)-1;
}
int nbytes_rcvd = fread(msg2.text, 1, nbytes, fp);
if(nbytes_rcvd!=nbytes) {
  return 1;
}
msg2.text[nbytes_rcvd] = 0;
```

b) Implemente o servidor desta aplicação (`ex5_srv.c`). Teste a comunicação entre os dois programas que implementou.

Exercício 6

O *Hypertext Transfer Protocol* (HTTP) é o protocolo de aplicação responsável pelas trocas de dados na *World Wide Web* (WWW), tais como o acesso a recursos *web* (páginas, imagens, etc.). Um recurso *web* é identificado através do respetivo *universal resource locator* (URL) (e.g., `http://ave.dee.isep.ipp.pt/~jes/sopcp/ola.html`). O URL indica o protocolo de acesso (`http`, no exemplo acima), o nome do servidor que disponibiliza o recurso (`ave.dee.isep.ipp.pt`, no exemplo acima) e o recurso propriamente dito[2] (`/~jes/sopcp/ola.html`, no exemplo acima).

O HTTP assenta tipicamente sobre o TCP/IP. O porto padrão para os servidores HTTP (também designados servidores *web*) é o 80. Após estabelecer a ligação TCP/IP com o servidor HTTP, o cliente (e.g., *web browser*) faz o pedido do recurso *web* através de uma mensagem semelhante à seguinte:

```
GET ~jes/sopcp/ola.html HTTP/1.1
Host: ave.dee.isep.ipp.pt

```

(incluir linha em branco, ou seja, `"\r\n"`)

A resposta do servidor deverá ter um conteúdo semelhante ao apresentado abaixo:

```
HTTP/1.1 200 OK
Date: Mon, 06 Feb 2017 09:27:29 GMT
Server: Apache/2.2.21 (Fedora)
Last-Modified: Wed, 03 Feb 2016 16:55:35 GMT
ETag: "a40df1-2e-52ae07c990fc0"
Accept-Ranges: bytes
Content-Length: 46
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Language: pt

<html>
<body>
Olá mundo!<br>
</body>
</html>
```

a) O programa `http_get.c` implementa a transação descrita acima. Analise o código do programa.
   1) Que tipo de comunicação (SOCK_STREAM/SOCK_DGRAM) é implementado neste programa?
   2) Em que tipo de aplicação este código seria normalmente usado, num *web browser* ou num servidor *web*?

b) Compile e execute o programa. Adicionalmente, aceda, num *web browser*, ao endereço `http://ave.dee.isep.ipp.pt/~jes/sistc/pl/ola.html`. Compare os resultados com o *output* do programa `http_get`. Ambos os programas (`http_get` e *browser*) recebem a mesma resposta do servidor. No entanto, o *browser* interpreta o código HTML para fazer a apresentação, devidamente formatada, do conteúdo da página.

[2] Para mais detalhes, consultar https://tools.ietf.org/html/rfc7230#section-2.7

c) Altere o programa de modo a fazer apenas a impressão do código HTML (ou seja, deverá omitir o cabeçalho da mensagem HTTP). Para tal, só deverá iniciar a impressão dos carateres após a receção de uma linha em branco (“\r\n”).
Uma forma de abordar o problema será fazer a leitura linha a linha da resposta HTTP até que seja detetada a linha “em branco” (`strcmp(buffer, “\r\n”)`).

Exercício 7

Atualmente existem diversas implementações de servidores HTTP para as mais diversas finalidades, pelo que a implementação de raiz de um servidor HTTP raramente é necessária. O programa fornecido no ficheiro `dummy_webserver.c` emula o funcionamento básico de um servidor HTTP, implementando um serviço rudimentar que consiste em devolver ao cliente uma página HTML com a reprodução da mensagem HTTP enviada inicialmente pelo cliente.

a) Numa *shell*, compile e execute o programa:

```
make dummy_webserver
./dummy_webserver 8080
```

b) Aceda, num *web browser*, ao endereço `http://localhost:8080/ola.html` e observe a informação apresentada, nomeadamente a linha com o pedido GET. Relacione o resultado com o código do ficheiro `dummy_webserver.c`.

Exercício 8

a) Implemente um programa que permita a receção de mensagens de texto através porto UDP 3000 e faça a sua impressão, assim como do endereço IP e porto do emissor, no ecrã.
b) Implemente um programa que permita enviar mensagens de texto, lidas do teclado, para o programa descrito na alínea anterior, até que seja terminado (através de CTRL+C, por exemplo). O endereço IP de destino das mensagens deverá ser especificado através da linha de comando.
c) Verifique que o programa inicial pode receber sequências de mensagens de diferentes clientes sem ter de ser reiniciado e sem recorrer a qualquer tipo de mecanismo multi-tarefa (criação de novos processos ou *threads*). Para tal, abra um terminal adicional, de modo a poder executar duas instâncias do programa criado na alínea b e enviar alternadamente mensagens a partir de cada um dos respetivos terminais.
d) O tamanho máximo dos pacotes IPv4 é 65536 bytes. No entanto, em geral, a camada de ligação (e.g., a rede *ethernet*) não permite o envio de tramas dessa dimensão. O protocolo IPv4 suporta um mecanismo de fragmentação de forma a superar essa limitação. Contudo, esse mecanismo tem algumas desvantagens pelo que, em Linux, está desativado por omissão para o UDP. Verifique, por tentativa e erro (também pode basear-se na informação fornecida pelo comando `ifconfig`), qual o tamanho máximo que as mensagens podem ter para serem enviadas com sucesso.

Histórico

- 2024-03-27 – Anterior Exercício 3 substituído por Exercícios 3, 4 e 5 (jes@isep.ipp.pt)
- 2020-02-01 – Exercício 3 (representação nativa vs em modo de texto) (jes@isep.ipp.pt)
- 2018-02-01 – Exercício SOCK_DGRAM (jes@isep.ipp.pt)
- 2017-02-01 – Exercício 2 e Exercício "dummy_webserver" (jes@isep.ipp.pt)
- 2016-02-01 – Exercício "http_get" (jes@isep.ipp.pt)
- 2013-02-01 – Exercício 1 (jes@isep.ipp.pt)