POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Programação Avançada – Licenciatura em Eng.ª Informática

Ficha 6 — Sockets UDP

Versão 2.4

Duração: 2 aulas

© **2025**: {patricio.domingues, vitor.carreira, gustavo.reis, carlos.grilo, carlos.machado, gabriel.silva, miguel.negrao}@ipleiria.pt

Sumário

1 Introdução

Os *sockets* são uma forma de comunicação entre processos que se distinguem das restantes (memória partilhada, *pipes*, filas de mensagens, ...) pelo facto de suportarem comunicação através da rede. Quando se pretende iniciar uma ligação entre processos através de *sockets* é necessário ter em conta os seguintes aspetos:

- O papel que cada processo desempenha — cliente ou servidor;
- O tipo de servidor — iterativo ou concorrente;
- O tipo de ligação — orientada à ligação (*connection oriented*) ou sem ligação permanente (*connectionless*);
- O protocolo de comunicação usado — internet, XNS, SNA, etc.

Nesta ficha vamos abordar os *connectionless sockets*, também conhecidos por *datagram sockets*, cujo protocolo de comunicação é o UDP. Os *datagram sockets* (UDP) não garantem a entrega da informação ao destinatário, ao contrário do que sucede com os *stream sockets*. No entanto, são muito utilizados porque requerem menos processamento e geram menos tráfego na rede. As aplicações que usam este tipo de *sockets* são aquelas em que a perda de um pacote de informação não prejudica de forma significativa o seu funcionamento. Por exemplo, num serviço de distribuição de hora para sincronizar os computadores duma rede (e.g. Network Time Protocol – NTP – RFC 5905), caso um pacote de pedido para obtenção da data atual seja perdido (ou a resposta do servidor), basta ao cliente efetuar novo pedido ao servidor. Outro tipo de aplicação que usa o UDP é a *voz sobre IP* (VoIP), porque é preferível perder um pacote de informação de voz do que recebê-lo com atraso.

2 Endereços IPv4 e IPv6

2.1 Endereços IPv4

Os endereços IPv4 são representados por uma cadeia de carateres composta por quatro grupos de números decimais separados por pontos (*dotted-decimal*), por exemplo: `192.168.234.244`. Este formato é útil para facilitar a sua leitura e memorização, mas tem de ser convertido para o seu equivalente binário (4 *bytes*, 32 *bits*) em **formato de rede**, para ser usado na estrutura de endereço de *sockets*. O formato de rede é *big endian*, e determina a ordem dos *bytes* num valor com vários *bytes*.

2.2 Endereços IPv6

Os endereços IPv6 são representados por 16 *bytes* (128 *bits*), agrupados em 8 blocos, cada um com dois bytes. Dado que é empregue a representação hexadecimal, cada bloco de dois *bytes* tem valores entre 0 e 0xFFFF (os zeros à esquerda podem ser omitidos). Os 4 *bytes*

menos significativos de um endereço IPv6 podem ser um endereço IPv4 em formato *dotted-decimal* (por exemplo, `1:2:3:4:5:6:192.193.194.195`). Um endereço IPv6 pode ainda fazer uso de representação abreviada, em que um ou mais blocos de zero *bytes* é abreviado por `::` (duas vezes dois pontos), sendo contudo apenas permitida a existência de um bloco `::`. Um exemplo de endereço IPv6 passível de ser abreviado é `1:2:0:0:0:0:7:8` que pode ser representado como `1:2::7:8`.

2.3 Endereço local (localhost)

O endereço dito de *localhost* é `127.0.0.1` em formato IPv4 e `::1` em formato IPv6.

2.4 Funções para a conversão de endereços

```c
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);

char *inet_ntop(int af, const void *src, char *dst, socklen_t size)
```

A função `inet_pton` faz a conversão de um endereço em formato texto (no caso de IPv4 em *dotted-decimal*) para o seu equivalente binário.

Recebe como parâmetros:

- `af` — família de endereços (IPv4 = AF_INET, IPv6 = AF_INET6);
- `src` — o endereço em formato texto (no caso de AF_INET em *dotted-decimal*);
- `dst` — o endereço onde a representação binária do endereço deve ser guardada (no caso de AF_INET deverá ser do tipo `struct in_addr`);

Valores de retorno

- **Sucesso** — devolve 1.
- **Insucesso** — devolve 0 se `src` não contém um endereço válido ou -1 se `af` não contém uma família de endereços válida.

A função `inet_ntop` faz o inverso, ou seja, faz a conversão da representação binária para o formato texto.

- `af` — família de endereços (IPv4 = AF_INET, IPv6 = AF_INET6);
- `src` — o endereço IP em formato binário (no caso de AF_INET deverá ser do tipo `struct in_addr`);
- `dst` — o endereço da string onde será colocado o texto (No caso de AF_INET deverá ter tamanho pelo menos `INET_ADDRSTRLEN` bytes, e a string gerada estará em formato *dotted-decimal*);

Valores de retorno

- **Sucesso** — ponteiro para `dst`.
- **Insucesso** — `NULL`.

2.5 Lab 1

Elabore o programa `IPv4_addr` que deve receber um endereço IPv4 em formato texto (p.ex., `192.168.14.11`) através da linha de comando e convertê-lo através da função `inet_pton`. Crie uma função `memory_dump` para mostrar cada um dos *bytes* da zona de memória ocupada pela variável que recebeu o valor devolvido por `inet_pton`.

> **Nota**: o parâmetro endereço IPv4 deve ser enviado a partir da linha de comandos, usando, para isso, a ferramenta gengetopt.

3 *Sockets* — estruturas, constantes e funções auxiliares

De seguida são apresentadas as estruturas de dados empregues com *sockets*, bem como algumas funções auxiliares para inicialização destas estruturas de dados.

3.1 Estruturas e constantes

A estrutura `struct sockaddr` representa o conceito de endereço de forma genérica, independente do protocolo. Esta estrutura é usada em várias funções como, por exemplo, `recvfrom`, `sendto`, `bind`, `accept`, `connect` e serve para uniformizar a interface das funções relacionadas com os *sockets*, por isso, nunca é usada diretamente. Assim, são usadas estruturas específicas para cada protocolo, sendo depois feito um *cast* para a estrutura genérica.

```
#include <sys/socket.h>
struct sockaddr {
    unsigned short sa_family;   /* address family: AF_XXXX value */
    char           sa_data[14]; /* protocol-specific address     */
}
```

Para representar um endereço *socket* Internet IPv4 deve ser usada a estrutura específica `struct sockaddr_in`. Por sua vez, para representar um endereço *socket* Internet IPv6 deve ser empregue a estrutura `struct sockaddr_in6`.

```
#include <arpa/inet.h>
struct sockaddr_in {
sa_family_t    sin_family; /* address family: AF_INET */
in_port_t      sin_port; /* port in network byte order (usar htons)*/
struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
uint32_t  s_addr;     /* address in network byte order (usar htonl)*/
};
```

```
12
13 #include <arpa/inet.h>
14 struct sockaddr_in6 {
15 sa_family_t     sin6_family;   /* AF_INET6 */
16 in_port_t       sin6_port;   /* port number, in network byte order */
17 uint32_t        sin6_flowinfo; /* IPv6 flow information */
18 struct in6_addr sin6_addr;     /* IPv6 address */
19 uint32_t        sin6_scope_id; /* Scope ID */
20 };
21
22 struct in6_addr {
23     uint8_t s6_addr[16];  /* IPv6 address, in network byte order */
24 };
```

Estas estruturas apresentam os seguintes campos:

- `sin_family` — indica o nome da família de protocolos. Deve ser usado AF_INET para *sockets* IPv4 e AF_INET6 para *sockets* IPv6.
- `sin_port` — serve para guardar o porto em formato de rede/*big-endian* (*byte ordered* — ver funções `htons` e `ntohs` mais abaixo).
- `sin_addr` — serve para especificar o endereço IP, também em formato de rede (ver função `inet_pton`).

Quando se regista um *socket* no sistema local (função `bind`), e caso se pretenda que um *socket* aceite ligações independentemente da interface de rede (por exemplo, num computador com duas placas de rede), é necessário colocar a constante `INADDR_ANY` no campo do endereço IP para o caso do IPv4. Para IPv6, faz-se uso da constante `in6addr_any`. Esta situação é usada normalmente quando pretendemos criar um *socket* nas aplicações servidoras. Por outro lado, se a constante `INADDR_ANY` for usada no campo do porto, estamos a indicar ao sistema operativo que pode escolher um porto disponível.

3.2 Funções de ordenamento dos *bytes*

Por forma a permitir a comunicação de processos entre sistemas com diferentes *endianess* — *little endian* (por exemplo x86 e x86-64) e *big endian* (por exemplo, sistemas com arquitetura SPARC) — existem funções para transformar os dados do formato específico da máquina (*host*) para o formato de rede (*net*, que é *big endian*) e vice-versa. A seguir, são apresentados os protótipos dessas funções:

```
1 #include <arpa/inet.h>
2 uint32_t htonl(uint32_t hostlong);
3 uint32_t ntohl(uint32_t netlong);
4 uint16_t htons(uint16_t hostshort);
5 uint16_t ntohs(uint16_t netshort);
```

Os nomes destas funções derivam da junção das abreviaturas das seguintes palavras:

- h — *host* (máquina);
- n — *network* (rede);
- l — *long* (32 bits);

- s — *short* (16 bits).

Assim, a função `htonl` (*host to network long*) converte o parâmetro de entrada, um *long int* (32 *bits*), do formato da máquina para o formato da rede. A descrição das restantes funções é facilmente dedutível.

3.3 Lab 2

Recorrendo a uma variável do tipo **`short`** e às funções de conversão do formato local / formato de rede, elabore o programa `determina_endianess` que deve indicar se a máquina local é *little endian* ou *big endian*. O programa deve validar que o tipo de dados **`short`** ocupa dois octetos.

3.4 Função de preenchimento `memset`

Antes de atribuir valores às variáveis do tipo `struct sockaddr_in`, devemos iniciá-las com todos os valores a zero (mesmo quando iremos preencher seguidamente todos os seus campos). Para atingir este objetivo usa-se a função seguinte:

```
1 #include <string.h>
2
3 void *memset(void *dest, int c, size_t n);
```

A função memset preenche cada um dos `n` primeiros bytes do endereço `dest` com o valor especificado pelo parâmetro `c`. Para "zerar" uma zona de memória, especifica-se o valor 0 (zero) para o parâmetro `c`.

Valores de retorno

Devolve um ponteiro para a zona de memória `dest`.

3.5 Lab 3

Elabore o programa `zera_addr` que:

1. Mostra o tamanho da estrutura `struct sockaddr_in`;
2. Fazendo uso da função `memset`, coloca a zero a zona de memória ocupada por uma variável do tipo `struct sockaddr_in` (`my_addr_IPv4`);
3. Após a chamada à função `memset`, usa a função `memory_dump` criada no lab1 para mostrar cada um dos *bytes* da zona de memória ocupada pela variável `my_addr_IPv4`, confirmando que cada *byte* tem o valor zero;

   Sugestão: iterar a zona de memória da variável do tipo `struct sockaddr_in` com um ponteiro para `unsigned char` (`unsigned char *`).

4. Fazendo uso da função `memset`, coloca o valor 4 em cada um dos primeiros 6 bytes da variável `my_addr_IPv4`;
5. Utiliza a função `memory_dump` para voltar a mostrar o estado atual da zona de memória da variável `my_addr_IPv4`.

4 *Sockets* UDP

A Figura 1 apresenta um esquema com os passos a realizar no estabelecimento de um sistema cliente/servidor que comunica via *datagram socket* (UDP).

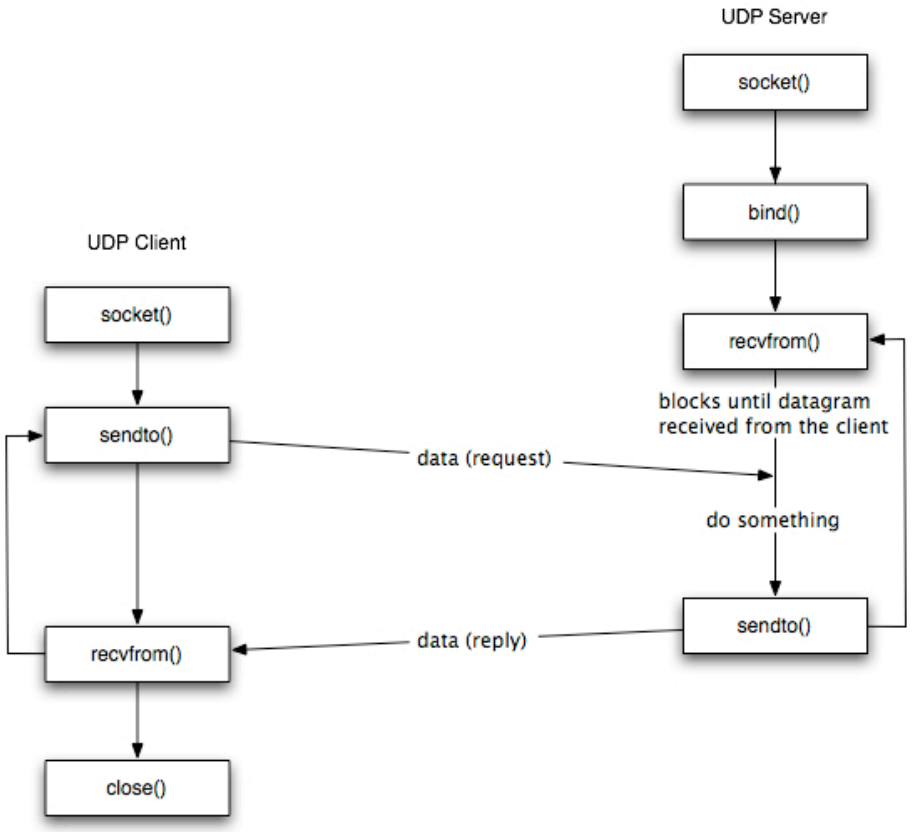

Figura 1: Esquema da criação de processos

4.1 *Socket*

Inicialmente, o servidor e o cliente devem criar um identificador para a ligação através da função `socket`.

4.1.1 Iniciar um *socket*

```c
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

A função `socket` cria e devolve um descritor para um *socket*.

Recebe como parâmetros:

- `domain` — família de endereços usada para a comunicação (por exemplo, `AF_INET` para IPv4).
- `type` — especifica o tipo de comunicação (por exemplo, `SOCK_STREAM`, `SOCK_DGRAM`, ou `SOCK_RAW`).
- `protocol` — especifica o protocolo a ser utilizado. Caso se pretenda utilizar o protocolo por omissão (TCP para `SOCK_STREAM`, UDP para `SOCK_DGRAM`, ...), passa-se como argumento o valor 0 (zero).

Valores de retorno

- **Sucesso** — devolve um descritor para o *socket*.
- **Insucesso** — devolve -1.

4.1.2 Fechar um *socket*

```c
#include <unistd.h>

int close(int fd);
```

A função `close` fecha um descritor de um ficheiro especificado pelo parâmetro `fd`, de modo a que não possa ser mais utilizado, e liberta os recursos associados, caso `fd` seja a última referência para o ficheiro em questão. Em contexto de *sockets*, podemos utilizar esta função para fechar e libertar os recursos de um *socket*, se o seu descritor for especificado como parâmetro `fd`.

Valores de retorno

- **Sucesso** — devolve 0.
- **Insucesso** — devolve -1.

4.2 Lab 4

Elabore o programa `cria_socket`, o qual deve:

1. Criar um *socket* UDP / IPv4;
2. Criar um *socket* UDP / IPv6;
3. Validar a correta criação de cada *socket*, detetando situações de erro. Caso a criação do *socket* seja bem-sucedida, deve ser mostrado o valor inteiro do descritor atribuído ao *socket*;

4. Fechar ambos os descritores.

4.3 Servidor

O servidor deve ainda registar o *socket* num porto pré-definido usando a função `bind`. É usual um programa servidor baseado em UDP implementar um serviço iterativo, dado ser muito frequente o serviço ser do tipo *"um pedido, uma resposta"*. Nesta configuração, o programa servidor está bloqueado no *socket* a aguardar um pedido de um qualquer cliente (para tal, efetuou a chamada à função `recvfrom`). Quando chega um pedido (p. ex., "qual é a data corrente?"), o programa servidor processa o pedido, enviando de seguida a resposta para o cliente que efetuou o pedido. Importante notar que esta configuração permite que o programa servidor atenda apenas um cliente de cada vez, situação aceitável se o processamento do pedido do cliente for rápido (p. ex., devolver a data corrente). Caso o processamento do pedido e elaboração da respetiva resposta possa demorar, então torna-se necessário implementar um servidor concorrente que recorra, por exemplo, a múltiplas *threads*/processos.

4.3.1 `bind`

```c
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

A função `bind` permite mapear um *socket* para um determinado porto e interface de rede local.

Recebe como parâmetros:

- `sockfd` — descritor do *socket* a mapear.
- `addr` — estrutura com o endereço a ser associado ao *socket* `sockfd`. No caso de um socket de rede deverá ser passado um endereço de uma estrutura `struct sockaddr_in` ou `struct sockaddr_in6` ao qual é aplicado um *cast*. O IP colocado nessa estrutura deverá ser o IP da interface de rede a utilizar ou `INADDR_ANY` caso se queira permitir pedidos de todas as interfaces de rede disponíveis. Esta estrutura irá também especificar o porto a usar.
- `addrlen` — especifica o tamanho, em *bytes*, da estrutura apontada por `addr`.

Valores de retorno

- **Sucesso** — devolve 0.
- **Insucesso** — devolve -1.

4.4 Lab 5

Elabore o programa `regista_UDP_lab` que deve criar e registar um *socket* UDP / IPv4 para todas as interfaces da máquina local. O porto a registar deve ser especificado através da linha de comando. O programa deve validar a criação do *socket*, bem como o registo local (`bind`).

1. O que sucede quando é indicado um porto entre 1 e 1023 (inclusive)?
2. O que sucede quando se tenta registar um porto que já está registado?

Nota: é possível obter uma lista dos portos UDP registados no sistema através do utilitário `ss`, executado da seguinte forma: `ss -unlp`. Para mais informações acerca deste comando, deve consultar a respetiva página de manual (`man ss`).

4.5 Cliente

Logo que uma aplicação cliente UDP tenha criado o *datagram socket* (através da chamada à função socket, que é idêntica à realizada pelo servidor), poderá enviar *datagrams* para qualquer outra entidade remota. Para tal, deve fazer uso da função `sendto`, indicando o endereço IP e porto da entidade remota para a qual pretende enviar o *datagram*. É ainda usual, após o envio do *datagram*, que a aplicação cliente aguarde pela resposta do servidor, recorrendo para o efeito à função `recvfrom`.

4.6 Comunicação entre cliente e servidor

As funções de leitura e escrita usadas na comunicação através de *sockets datagram* são `recvfrom` e `sendto`, respetivamente.

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3
4  ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr
       *src_addr, socklen_t *addrlen);
5
6  ssize_t sendto(int sockfd, void *buf, size_t len, int flags, const struct
       sockaddr *dest_addr, socklen_t addrlen);
```

A função `recvfrom` permite receber mensagens ou dados através um *socket* e a função `sendto` permite enviar mensagens ou dados através de um *socket*.

Recebem como parâmetros:

- `sockfd` — descritor do *socket* .
- `buf` — endereço no qual o conteúdo do *datagram* é escrito no caso da função `recvfrom`, ou lido, no caso da função `sendto`.
- `len` — tamanho da zona de memória de `buf`. É necessário ter em conta que o tamanho máximo teórico de um *datagram* é de 64 KiB, por isso, o tamanho do parâmetro `buf`

deve ser igual ou inferior a este valor (o valor mais apropriado seria o do tamanho da MTU [1] da rede para minimizar a perda de pacotes).

- `flags` — permite aceder a modos alternativos das funções. Por exemplo, a opção `MSG_DONTWAIT` pode ser usada na função `recvfrom` para que esta deixe de ser bloqueante.
- `src_addr` e `dest_addr` — estrutura que indica o endereço associado ao *datagram* (ver descrição na Secção 3). Na operação de leitura, o parâmetro contém o endereço e porto de origem do *datagram* recebido. O endereço pode depois ser usado, por exemplo, para o servidor enviar a resposta ao cliente. Na operação de escrita, o parâmetro especifica o endereço e porto de destino para onde se pretende enviar o *datagram*.
- `addrlen` — corresponde ao tamanho do endereço (`src_addr` ou `dest_addr`). No caso da função `recvfrom`, este parâmetro é um ponteiro que é preenchido pela função com o tamanho do endereço. Nesta função **o parâmetro** `addrlen` **é um parâmetro valor/-resultado**, significando isso que deve ser iniciado com o tamanho da estrutura de endereço antes da chamada à função, caso contrário ocorre o erro do tipo `EINVAL` (argumento inválido). Na função `sendto`, este parâmetro, que não é um ponteiro, deve ser igualmente preenchido com o tamanho da estrutura de endereço.

Valores de retorno

- **Sucesso** — devolve o número de bytes recebidos, no caso da função recvfrom, ou o número de bytes enviados, no caso da função sendto.
- **Insucesso** — devolve -1.

> **Nota**: De acordo com a Figura 1, a ordem das operações de leitura e escrita são relevantes. O servidor tem de começar sempre com uma leitura para obter o endereço ao qual vai responder ao pedido. Consequentemente, o cliente é obrigado a começar sempre com uma escrita. Lembra-se que, por omissão, a função `recvfrom` bloqueia o próprio processo até que seja recebido um *datagram*.

> **Nota**: Ao enviar uma *string* não é necessário enviar o caracter `'\0'`. Por outro lado, ao receber uma *string*, a função `recvfrom` irá devolver o número de *bytes* recebidos, que corresponde ao número de caracteres da *string*. Este valor deverá depois ser usado para terminar corretamente a *string*: `buf[n] = '\0';`.

4.7 Lab 6

Elabore o programa cliente `envia_pacote_UDP` e o programa servidor `recebe_pacote_UDP`, utilizando para isso a *template* **"EmptyProject-Client-Server-Template"**. Os programas devem ser executados e testados em terminais distintos.

[1] Maximum Transmission Unit

O programa servidor deve criar e registar um *socket* UDP / IPv4 para o endereço local, num porto predefinido (8899), receber um número inteiro de 16 *bits* sem sinal e devolver a raiz quadrada desse número em formato *string*, terminando de seguida.

O programa cliente deve ligar-se ao servidor, enviar-lhe um número inteiro aleatório de 16 *bits* sem sinal (entre 1 e 100), receber o valor da raiz quadrada desse número em formato *string* e mostrar o valor calculado pelo servidor.

> **Nota**: use o ficheiro **"common.h"** para definir o porto a usar por ambos os programas e não se esqueça de libertar, corretamente, todos os recursos utilizados.

5 Espera limitada no tempo no `recvfrom`

Por omissão, a função `recvfrom` é bloqueante, o que leva a que o processo/*thread* chamante fique bloqueado até que ocorra a receção de dados. Contudo, em certas situações, a receção de dados poderá nunca ocorrer. Considere-se, por exemplo, uma aplicação cliente que enviou um datagrama UDP com um pedido de serviço a um servidor, chamando de seguida a função `recvfrom` com o intuito de receber a resposta do servidor. Contudo, tanto o pedido do cliente como a resposta do servidor podem perder-se, nunca sendo entregue aos respetivos destinatários. Neste cenário, a aplicação cliente ficará bloqueada indefinidamente. Para evitar o bloqueio sem limite temporal da chamada, a API *socket* apresenta duas possibilidades: i) configurar o *socket* através da função `setsockopt` estabelecendo um limite máximo de espera em operações de `recv`/`recvfrom`; ii) efetuar uma chamada `recv`/`recvfrom` não bloqueante, que retorna imediatamente, indicando se foram ou não recebidos dados. Estas duas possibilidades são analisadas de seguida.

5.1 `setsockopt`

A função `setsockopt` permite configurar várias propriedades de um *socket*, nomeadamente estabelecer um tempo máximo de espera numa operação de receção, seja através da função `recv` ou da função `recvfrom`.

O protótipo da função `setsockopt` é:

```c
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int optname, const void *optval,
    socklen_t optlen);
```

A configuração de um temporizador para operações de receção `recv`/`recvfrom` faz-se indicando `SOL_SOCKET` para o parâmetro `level`, `SO_RCVTIMEO` para o parâmetro `optname` e o endereço de uma estrutura do tipo `struct timeval`. É nesta estrutura que é indicado o tempo máximo de espera. A estrutura `struct timeval` tem a seguinte definição:

```c
#include <sys/time.h>

struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

A Listagem 1 exemplifica o uso da função `setsockopt` para configurar um tempo máximo de espera de dois segundos.

Listagem 1: Uso da função setsockopt

```c
#include <sys/time.h>

struct timeval timeout;
timeout.tv_sec = 2;
timeout.tv_usec = 0;
int setsock_ret = setsockopt(udp_client_socket,SOL_SOCKET, SO_RCVTIMEO, &
    timeout, sizeof(timeout));
if(setsock_ret == -1 ){
    ERROR(EXIT_FAILURE,"Cannot setsockopt 'SO_RCVTIMEO'");
}
```

Configurado o *socket* com a função `setsockopt`, a chamada à função `recvfrom` faz-se normalmente (ver Listagem 2). Caso ocorra uma espera para além do tempo máximo de espera, a função `recvfrom` termina, devolvendo -1, com a mensagem de erro associada "Resource temporarily unavailable".

Listagem 2: Uso normal da função recvfrom

```c
ssize_t udp_read_bytes = recvfrom(udp_client_socket, S, sizeof(S)-1,0, (struct
    sockaddr *) &udp_server_addr, &udp_server_addr_len);
if( udp_read_bytes == -1 ){
    ERROR(EXIT_FAILURE,"Cannot recvfrom server");
}
```

5.2 Chamada `recv`/`recvfrom` não bloqueante

Uma alternativa à função `setsockopt` é o uso da constante `MSG_DONTWAIT` no parâmetro `flag` das funções `recv`/`recvfrom` de modo a que as funções não bloqueiem, mesmo que não existam dados para receber. Concretamente, a chamada à função `recvfrom` com a indicação `MSG_DONTWAIT`, leva a que quando não existem dados para receber, a função devolve o valor -1 e que o valor de `errno` seja `EWOULDBLOCK`.

A listagem ilustra uma leitura com `recvfrom` não bloqueante. Para o efeito é empregue um ciclo *while*, dentro do qual é efetuada a chamada ao `recvfrom`. Caso a chamada falha com -1 e `errno` a `EWOULDBLOCK`, é efetuada uma espera de um segundo (`sleep(1)`), que é seguida por nova chamada a função `recvfrom`. No total é efetuado um máximo de três chamadas não bloqueantes, após o qual se considera que não será possível receber nenhum datagrama.

```c
#define MAX_ATTEMPTS (3)
```

```
int received = 0;
int num_attempts = 0;
while( ! received ){
    num_attempts++;
    udp_read_bytes = recvfrom(udp_client_socket, S, sizeof(S)-1,
    MSG_DONTWAIT, (struct sockaddr *) &udp_server_addr,
                            &udp_server_addr_len);
    if( udp_read_bytes == -1 ){
        if ( errno == EWOULDBLOCK ){
            if( num_attempts >= MAX_ATTEMPTS ){
                    fprintf(stderr,"[CLIENT] Timeout at "
                        "recvfrom (%d attempts)\n", num_attempts);
                    exit(EXIT_FAILURE);
            }
            else{
                printf("[CLIENT] attempt %d\n", num_attempts);
                sleep(1);
            }
        }else{
            ERROR(EXIT_FAILURE, "Cannot recv server");
    }else{
        // successful read
        received = 1;
        break;
}//while
```

6 Modo ligado UDP

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *svc_addr, int addrlen);
```

Um *socket* UDP pode ser colocado em modo dito *ligado*. Para tal, faz-se uso da função `connect`, indicando um endereço remoto (endereço IP/porto). O efeito resultante é que o *socket* apenas pode comunicar com a entidade remota (endereço IP/porto) especificada na função `connect`. Quando o cliente pretende comunicar com apenas um servidor deverá ser utilizada esta função, por forma a garantir que apenas respostas provenientes do servidor são aceites.

Num *socket* em modo ligado, ocorre notificação de erro quando se envia um pacote para um endereço IP:Porto no qual não está nenhum servidor à escuta. Contudo, um *socket* UDP em modo ligado continua a não garantir a entrega dos pacotes, tal como um *connectionless socket*. O cliente só é notificado do erro quando tenta receber informação do servidor para o qual enviou a mensagem, ou seja, aquando da chamada à função `recvfrom`. Em caso de erro, a função `recvfrom` devolve -1 e o código de erro é guardado na variável `errno`.

Habitualmente, com *sockets* UDP, empregam-se as funções `recvfrom` e `sendto` para, respetivamente, receber e enviar um *datagram* UDP. No entanto, quando se está a usar um *socket* UDP em modo ligado, as funções `recvfrom` e `sendto` podem ser substituídas pelas funções `recv` e `send`. Estas funções possuem menos parâmetros dado não ser necessário especificar

o endereço de origem/destino do *datagram*. Importa ainda notar que as funções `recv` e `send` são também empregues com *stream sockets* (e.g., TCP), constituindo nesse caso uma alternativa às funções `read` e `write`, respetivamente.

6.1 Lab 7

Modifique o código do Lab 6 de modo que se utilize o protocolo UDP em modo ligado. Compile e execute os programas e observe os resultados.

6.2 Seleção da interface/porto num *socket* UDP cliente

Usualmente, a atribuição do porto do *socket* cliente é feita automaticamente pelo sistema local. Contudo, é possível selecionar a interface local e o respetivo porto para um *socket* UDP cliente. Para o efeito, faz-se uso da função `bind`, especificando a interface pretendida e o respetivo porto. Note-se que o pedido falhará caso o porto indicado já se encontre em uso. Acresce-se que esta funcionalidade é raramente empregue.

7 Exemplo

O exemplo seguinte ilustra a implementação do jogo "*Adivinha Número*" usando *sockets* UDP. Ao iniciar, o servidor gera um número aleatório. Em cada iteração recebe um número de um cliente e compara-o com o número aleatório gerado. De seguida, indica ao cliente, através de constantes, se o número é igual, menor ou maior que o número gerado. No entanto, devido ao facto dos *sockets* UDP serem *sem ligação (connectionless)*, o número a adivinhar pelos vários clientes será o mesmo.

Listagem 3: Ficheiro common.h

```h
#ifndef __COMUM_H__
#define __COMUM_H__

#define IGUAL 0
#define MENOR 1
#define MAIOR 2

#endif /* __COMUM_H */
```

Listagem 4: Servidor

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```c
#include <fcntl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <time.h>

#include "debug.h"
#include "common.h"
#include "server_args.h"

int processa_pedido(int fd, uint16_t n_serv);

int main(int argc, char *argv[]) {
    /* Estrutura gerada pelo utilitario gengetopt para
     guardar os parametros de entrada */
    struct gengetopt_args_info args_info;

    /* Processa os parametros da linha de comando */
    if (cmdline_parser(argc, argv, &args_info) != 0) {
        return 1;
    }

    int sock_fd;
    /* Cria o socket */
    if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        ERROR(2, "Criaçao do socket");
    }

    struct sockaddr_in ser_addr;
    /* Preenche a estrutura */
    memset(&ser_addr, 0, sizeof(ser_addr));
    ser_addr.sin_family = AF_INET;
    ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    ser_addr.sin_port = htons(args_info.port_arg);

    /* Efetua o registo */
    if (bind(sock_fd, (struct sockaddr *)&ser_addr, sizeof(ser_addr)) == -1) {
        ERROR(3, "bind server");
    }

    srand(time(NULL));
    while (1) {
        uint16_t gerado = 1 + (uint16_t)(rand() % 100);
#ifdef SHOW_DEBUG
        DEBUG("Numero aleatório = %hu", gerado);
#endif
        /* Apenas sai do ciclo se o numero foi adivinhado */
        while (!processa_pedido(sock_fd, gerado))
            ;
    }
    return 0;
}

int processa_pedido(int fd, uint16_t n_serv) {
    uint16_t n_cli;
    struct sockaddr_in cli_addr;
    socklen_t len = sizeof(cli_addr);
    if (recvfrom(fd, &n_cli, sizeof(n_cli), 0, (struct sockaddr *)&cli_addr,
                 &len) == -1) {
        ERROR(4, "recvfrom");
    }
#ifdef SHOW_DEBUG
```

```
    char ip[20];
    DEBUG("cliente [%s@%d]",
          inet_ntop(AF_INET, &cli_addr.sin_addr, ip, sizeof(ip)),
          htons(cli_addr.sin_port));
#endif

    n_cli = ntohs(n_cli);

    uint16_t res;
    if (n_cli == n_serv) {
        res = IGUAL;
    } else if (n_cli < n_serv) {
        res = MENOR;
    } else {
        res = MAIOR;
    }

    res = htons(res);

    if (sendto(fd, &res, sizeof(res), 0, (struct sockaddr *)&cli_addr, len) <
        0) {
        ERROR(5, "sendto");
    }

    return n_cli == n_serv;
}
```

Listagem 5: Cliente

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <time.h>

#include "debug.h"
#include "common.h"
#include "client_args.h"

void adivinha_num(int fd, struct sockaddr_in ser_addr);

int main(int argc, char *argv[]) {
    /* Estrutura gerada pelo utilitario gengetopt */
    struct gengetopt_args_info args_info;

    /* Processa os parametros da linha de comando */
    if (cmdline_parser(argc, argv, &args_info) != 0) {
        return 1;
    }

    int sock_fd;
    /* Cria o socket */
    if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

```c
        ERROR(2, "Criacao do socket");
    }

    struct sockaddr_in ser_addr;
    /* Preenche a estrutura */
    memset(&ser_addr, 0, sizeof(ser_addr));
    ser_addr.sin_family = AF_INET;
    ser_addr.sin_port = htons(args_info.port_arg);

    /* Utiliza a função inet_pton para preencher o endereço */
    switch (inet_pton(AF_INET, args_info.ip_arg, &ser_addr.sin_addr)) {
    case 0:
        printf("O endereço IP %s não e' valido\n\n", args_info.ip_arg);
        cmdline_parser_print_help();
        return 2;
    case -1:
        printf("Endereço IP desconhecido: %s\n\n", args_info.ip_arg);
        cmdline_parser_print_help();
        return 2;
    }

    adivinha_num(sock_fd, ser_addr);

    close(sock_fd);
    return 0;
}

void adivinha_num(int fd, struct sockaddr_in ser_addr) {
    uint16_t num, res;
    do {
        printf("\nIntroduza um numero entre 1 e 100: ");
        scanf("%hu", &num);

        num = htons(num);

        socklen_t len = sizeof(struct sockaddr_in);

        if (sendto(fd, &num, sizeof(num), 0, (struct sockaddr *)&ser_addr,
                   len) < 0) {
            ERROR(3, "sendto");
        }

        if (recvfrom(fd, &res, sizeof(res), 0, (struct sockaddr *)&ser_addr,
                     &len) < 0) {
            ERROR(4, "recvfrom");
        }

        res = ntohs(res);

        if (res == IGUAL) {
            printf("Parabens! Acertou\n\n");
        } else if (res == MENOR) {
            printf("O numero do servidor e' MAIOR\n");
        } else {
            printf("O numero do servidor e' MENOR\n");
        }

    } while (res != IGUAL);
}
```

7.1 Lab 8

Analise o código do servidor e do cliente, do exemplo acima apresentado, e responda às questões seguintes. Depois, compile e execute este projeto de modo a verificar as respostas.

1. O que faz a linha de código `srand(time(NULL));` empregue no servidor?
2. Qual é a gama de números aleatórios gerados pela seguinte linha de código (código servidor)? `uint16_t gerado = 1 + (uint16_t)(rand()% 100);`
3. Caso exista mais de um cliente, o que acontece aos restantes se um deles acertar no número?

> **Nota:** tenha em atenção que o código servidor utiliza, via *gengetopt* o parâmetro `--porto/-p <porto>`, e que o código cliente utiliza, para além da opção `--porto/-p`, a opção `--ip/-i <IP_servidor>`. Esta última opção serve para especificar o endereço IP do servidor com o qual o cliente pretende interagir.

8 Exercícios

8.1 Aula

1. Elabore, em C com *sockets* UDP, o programa servidor `servidorEco` que recebe como parâmetro de entrada o porto onde vai ficar à escuta. O servidor deve receber uma mensagem enviada por um cliente, mostrá-la no *stdout* e enviá-la novamente ao cliente. Implemente também o cliente `clienteEco`, que recebe como argumentos de entrada o endereço IP e o porto do servidor. O cliente deve pedir ao utilizador as mensagens a enviar e terminar quando a mensagem for igual a "fim".

> **Info:** teste o servidor UDP através da ferramenta `nc` (*netcat*).
>
> ```
> 1 nc -u IP_servidor Porto_servidor
> ```

8.2 Extra aula

2. Modifique o cliente do *Lab 8* para que este desista de esperar pela resposta do servidor caso este demore mais do que um segundo a responder. Utilize a função `setsockopt` para definir um tempo máximo de espera na operação de leitura. Caso a resposta não chegue antes do tempo definido, a função `recvfrom` devolve -1 sendo o código de erro `EWOULDBLOCK` colocado na variável `errno`.

> **Nota**: para testar altere também o servidor, de modo a que este faça uma pausa suficientemente grande antes de chamar a função `sendto`.

3. Elabore o programa servidor `TimestampServerUdp` que, para cada pedido de ligação, responde com uma *string* indicando o tempo em microssegundos desde 00:00:00 de 1 janeiro de 1970 (*Epoch*). Quando iniciado, o servidor deve apresentar uma mensagem com o nome do programa e o porto onde se encontra o serviço. Na ocorrência de um pedido, o servidor deve apresentar a identificação do cliente (endereço e porto IP) no terminal e enviar ao cliente o número de microssegundos desde 1 janeiro de 1970. Implemente também o cliente `checkTimestamp` a fim de testar o servidor `TimestampServerUdp`.

4. Elabore o programa cliente `enviar&receber_random_UDP` que deve criar e registar um *socket* UDP / IPv4 para o endereço local, num porto que recebe como parâmetro de entrada. Este programa deverá escutar até 3 mensagens através deste *socket* e criar uma *thread* adicional que lhe envie um número aleatório entre 0 e 10, de 2 em 2 segundos. Este precisará, ainda, usar UDP em modo ligado.

   > **Nota**: as duas *threads* envolvidas devem utilizar descritores de *sockets* diferentes e após a receber 3 mensagens a *thread* principal deverá fechar o seu *socket*, utilizando a função close.

5. Implemente o programa servidor `StatServerUdp` que responde com dados estatísticos de um ficheiro indicado pelo cliente. Assim que o cliente estabelece a ligação, deve enviar o caminho absoluto de um ficheiro existente no servidor. Na ocorrência de um pedido, o servidor deve tentar obter os dados estatísticos do ficheiro e enviar a seguinte informação: o tamanho, a data do último acesso, a data da última modificação e a data da última alteração dos meta-dados (`man 2 stat`). O formato da resposta deve corresponder ao seguinte:

   ```
   Size: <BYTES> bytes
   Last Access: <DATA>
   Last Modified: <DATA>
   Last Changed: <DATA>
   ```

   Em caso de erro (por exemplo, o ficheiro não existe), o servidor deve devolver a mensagem correspondente (`man strerror`).

   O programa servidor recebe como parâmetro de entrada o porto onde vai ficar à escuta (`--port` ou `-p`). Investigue como pode utilizar o comando `nc` (ou *netcat*) por forma a testar o servidor, evitando assim a implementação de um programa cliente para o efeito.

6. Como sabe, por força do protocolo UDP, o tamanho de um *datagram* UDP encontra-se teoricamente limitado a 2^{16} bytes, ou seja, 65536 bytes. Construa a aplicação cliente-servidor `max_udp`, composta por um servidor `max_udp_s` e um cliente `max_udp_c`, onde ocorre a troca de mensagens entre o cliente e o servidor. O cliente deve suportar a opção `--tam <Tamanho>` onde é indicado o tamanho do *datagram* UDP a ser enviado ao servidor. Teste a sua aplicação correndo o servidor numa máquina e o cliente noutra.

1. Qual é o tamanho máximo do *datagram* suportado entre o cliente e o servidor? Verifique as diferenças quando o cliente e o servidor estão ligados por uma ligação *wireless* vs ligação fixa.
2. O que sucede se especificar ao cliente o endereço (i.e. IP + porto) de um sistema em que o servidor não se encontra a correr?