Sockets: Sumário

- Conceito.
- *Sockets* UDP
  - API de Java
  - **API da biblioteca C.**
  - (Comunicação *multicast*.)

Passos para Comunicação com *Sockets* UDP

1. Criar um *socket*:
   - `socket()`
2. Atribuir-lhe um *nome*:
   - `bind()`
   - se um *socket* só receber mensagens de *sockets* a que enviou mensagens, não é necessário atribuir-lhe explicitamente um nome.
3. Transferir informação:
   - `sendto()/recvfrom()`

Criar um *Socket* em C

- Em Unix/Linux, para criar um *socket* deve usar-se a chamada ao sistema:

  `int socket(int domain, int type, int protocol)`
  onde:
  `domain:` especifica a "pilha de protocolos" a usar;
  `type:` especifica as propriedades do canal;
  `protocol:` especifica qual o protocolo a usar.

- `socket()` retorna um *identificador local*, semelhante a um *file descriptor* retornado por `open()`
  - Neste ponto, o *socket* ainda não tem nome.
- A interface *socket* pode ser usada com praticamente qualquer pilha protocolar:
  - Na prática, usa-se quase exclusivamente com a família Internet (`PF_INET`)

Valores de `domain`

- Especifica a "pilha de protocolos" a usar:

  `PF_UNIX:` canal usado para comunicação entre processos no sistema Unix (Linux) – não envolve comunicação através da rede (`man 7 unix`);

  `PF_INET:` canal usado para comunicação usando os protocolos da arquitectura Internet (`man 7 ip`) – pode também ser usado para a comunicação entre processos no mesmo computador;

  `PF_PACKET:` canal usado para enviar pacotes directamente pelo *device driver* (nível interface da Internet: `man 7 packet`.)

Valores de `type`

- Especifica as propriedades do canal de comunicação (nem todas as pilhas suportam todas as possibilidades)

type	cnx	fiab	ord	msg	PF_INET
SOCK_STREAM	S	S	S	N	S
SOCK_DGRAM	N	N	N	S	S
SOCK_SEQPACKET	N	S	S	S	N
SOCK_RAW	D	D	D	D	S
SOCK_RDM	N	N	S	S	N

  Porquê `'D'` para `SOCK_RAW`?

- Para `PF_INET`, o tipo `SOCK_RAW` permite aceder directamente à interface do protocolo IP (ver `man 7 ip`);

Valores de `protocol`

- Especifica o protocolo a usar – normalmente só há um protocolo por cada par `(domain, type)`: exagero na concepção da interface de *sockets*?

- Para `PF_INET` (`man 7 ip`):

type	protocol
SOCK_STREAM	0, IPPROTO_TCP
SOCK_DGRAM	0, IPPROTO_UDP
SOCK_RAW	(IP, ver RFC 1700)

Identificação dum *socket*

- O valor retornado pela chamada ao sistema `socket()` só tem significado para o processo que o invoca;
- Para que um processo remoto possa comunicar é necessário atribuir-lhe um *nome*:

  Uma ficha telefónica não é suficiente para estabelecer uma chamada telefónica
- O nome dum *socket* depende da pilha de protocolos.

Nome dum *socket* `PF_INET`

- Consiste num par `(end. IP, porto)`:

  endereço IP: duma carta de rede (p.ex., o endereço IP de `www.fe.up.pt` é `193.136.28.205`):
  - inteiro sem sinal de 32 bits;
  - único entre todos os computadores ligados à Internet:
    - a vida real é um bocado mais complicada.

  porto: é um endereço de UDP/TCP - permite a existência simultânea de múltiplos canais de comunicação:
  - inteiro sem sinal de 16 bits;
  - único entre todos os *sockets* no computador que usam o protocolo UDP
    - no caso do protocolo TCP, pode haver mais do que um *socket* associado ao mesmo porto, no entanto só há um *socket* por extremidade dum canal.

struct sockaddr_in (man 7 ip)

```
#include <netinet/in.h>

struct sockaddr_in {
   sa_family_t    sin_family; /* address family: AF_INET */
   u_int16_t      sin_port;   /* port in network byte order */
   struct in_addr  sin_addr;  /* internet address */
};

/* Internet address. */
 struct in_addr {
    u_int32_t      s_addr;     /* address in network byte order
};
```


Network byte order

Problema: diferentes arquitecturas usam diferentes formas de representar os vários tipos de dados.

Solução: especificar uma forma de representação única para transmitir dados através da rede

Funções de conversão:

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

q **Obs:** IA32 usa LSByte primeiro, enquanto que a ordem especificada en IP é MSByte primeiro.

Inicialização do nome dum *socket*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
...
#define MYPORT          22222
...
   int s;
   struct sockaddr_in sad_loc;
   ...
   sad_loc.sin_family = AF_INET;
   sad_loc.sin_port = htons(MYPORT);
   sad_loc.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
```

- Quer o número do porto quer o endereço IP têm que ser convertidos para usar a ordem especificada nos protocolos TCP/IP;
- `INADDR_LOOPBACK` é um endereço IP que designa o computador local, usando o *loopback device*: as mensagens não circulam na rede.

Endereços *dotted decimal*

- Normalmente, usa-se a notação *dotted decimal* para representar um endereço IP. P.ex., o endereço IP de `www.fe.up.pt` é `193.136.28.31`
- As funções abaixo podem ser usadas para converter de e para o formato *dotted decimal*:

  `long inet_aton(char *, struct in_addr *)` converte do formato *dotted decimal* para o formato da Internet; correspondente em formato da Internet;

  `char *inet_ntoa(struct in_addr)` converte do formato da Internet para formato *dotted decimal* (particularmente útil para *debugging*).

`inet_aton()`: exemplo

```
/* Initializes a struct sockaddr_in, given an IP address in
 * dotted decimal format and a port number in native format */
int sin_init_addr(struct sockaddr_in *sad_in, char *dot_addr,
                  unsigned short int port) {
   sad_in->sin_family = AF_INET;
   sad_in->sin_port = htons(port);
   if( inet_aton(dot_addr, &sad_in->sin_addr) == 0 ) {
      printf("%s is not a valid IP address.\n", dot_addr);
      return -1;
   } else
      return 0;
}
```


`gethostbyname()`

Problema: Normalmente, na interface com um ser humano usa-se nomes DNS e não endereços IP, mesmo na notação *dotted decimal*. Mas o protocolo IP não entende nomes DNS.

Solução: Converter os nomes DNS em endereços IP.

- Esta conversão é realizada pelo *resolver* (`man 3 resolver`), un conjunto de funções.
- A biblioteca C oferece a função:

  ```
  struct hostent *gethostbyname(const char *name);
  ```

  E, contudo, o uso desta função não é muito conveniente. Comparem com a simplicidade dum:

  ```
  static InetAddress getByName(String name);
  ```

Atribuição dum nome a um *socket*

```
#include <sys/types.h>
#include <sys/socket.h>
int  bind(int  sockfd, struct sockaddr *my_addr,
          socklen_t addrlen);
```

`sockfd`: identificador do *socket*, i.e. valor retornado por `socket()`;

`my_addr`: endereço duma estrutura de dados com o endereço (nome) a atribuir ao *socket*;

`addrlen`: comprimento da estrutura de dados apontada pelo argumento `my_addr`.

IMP.- O programador tem de inicializar de forma apropriada `struct sockaddr my_addr`. Em particular, tem que tomar em consideração:

- possíveis diferenças na representação do endereço IP e do porto pelo computador e pela rede.

Exemplo: uso de `bind`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
...
struct sockaddr_in sia;
... /* Setup sockaddr_in */
if((s = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
   perror(''socket'');
   return -1;
}
if( bind(s, (struct sockaddr *)&sia, sizeof(sia)) == -1) {
   perror(''bind'');
   return -1;
}
...
```

Comunicação Sem Conexão: Envio

- Mensagens são enviadas independentemente umas das outras:
  - o canal de comunicação é "estabelecido" dinamicamente, para cada mensagem;
  - há que especificar as 2 extremidades desse canal, para cada mensagem.
- Chamada ao sistema `sendto()`:
  - retorna após cópia da mensagem para o sub-sistema de rede do SO
    - um processo pode bloquear temporariamente por falta de recursos;
    - em Java, `DatagramSocket.send()` também pode bloquear. Porquê?

`sendto()`

```
int  sendto(int s, const void *msg, size_t len, int flags,
            const struct sockaddr *to, socklen_t tolen);
```

`s:` identificador do *socket* remetente, i.e. extremidade local do canal de comunicação;

`msg:` endereço do *buffer* contendo a mensagem a transmitir;

`flags:` *bitmask* especificando diferentes opções;

`to:` endereço do nome do *socket* destinatário, i.e. do nome da extremidade remota do canal de comunicação;

`tolen:` comprimento da estrutura de dados com o nome do *socket* destinatário;

`sendto()` retorna o número de *bytes* transmitidos (-1 se ...)

`sendto()`: exemplo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <string.h>
...
#define MSG ‘‘What time is it?’’
...
   int                 so;
   struct sockaddr_in  loc, rem;
   char                *buf = MSG;
   size_t              len = strlen(buf);
   ...
   so = socket(PF_INET, SOCK_DGRAM, 0);
   bind(so, (struct sockaddr *)&loc, sizeof(loc));
   if( sendto(so, (void *)buf, len, 0,
              (struct sockaddr *)&rem, sizeof(rem)) == -1 ) {
      perror(‘‘sendto’’);
      return -1;
   }
```


Comunicação Sem Conexão: Recepção

- Mensagens são recebidas independentemente umas das outras:
  - o destinatário não precisa de conhecer *a priori* o remetente;
  - um mesmo *socket* pode receber mensagens enviadas por diferentes processos.
- Chamada ao sistema `recvfrom()`:
  - se não houver qualquer mensagem à espera de ser entregue, o destinatário bloqueia:
    * e se o emissor não enviar qualquer mensagem?
  - em Java, `DatagramSocket.receive()` também bloqueará.

recvfrom()

```
int  recvfrom(int  s,  void  *buf,  size_t len, int flags,
              struct sockaddr *from, socklen_t *fromlen);
```

s: identificador do *socket* receptor, i.e. do *socket* local;

buf: endereço dum *buffer* que será inicializado com a mensagem recebida;

len: comprimento de `buf` em *bytes*;

flags: *bitmask* especificando diferentes opções;

from: endereço duma `struct sockaddr` a inicializar com o nome do *socket* remetente, i.e. do *socket* remoto;

fromlen: endereço dum inteiro inicializado com o tamanho de `*from`. `recvfrom()` altera este valor para o comprimento da estrutura de dados com o nome do *socket* remetente.

`recvfrom` retorna o número de *bytes* recebidos (-1 se ...)

recvfrom(): exemplo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <string.h>
...
#define BUF_LEN 1024
...
   int                so;
   struct sockaddr_in loc, rem;
   char               buf[BUF_LEN];
   size_t             msg_sz, len = sizeof(rem);
   ...
   so = socket(PF_INET, SOCK_DGRAM, 0);
   bind(so, (struct sockaddr *)&loc, sizeof(loc));
   if( (msg_sz = recvfrom(so, (void *)buf, BUF_LEN, 0,
                          (struct sockaddr *)&rem, &len)) == -1 ) {
      perror(''recvfrom'');
      return -1;
   }
```

Comunicação *Multicast* em Java

- Comunicação *multicast* permite comunicar de 1 para *n* processos.
- IPv4 reserva os endereços `224.0.0.0` a `239.255.255.255` para comunicação *multicast*:
  - Neste caso, cada par `(endereço IP, porto)`, está associado não a um processo mas um grupo de processos, i.e. um *grupo multicast*.
- Só UDP suporta *multicast*
  - Implementar comunicação *multicast* fiável é não trivial:
    - De facto, definir *multicast* fiável é não trivial.
- Java suporta comunicação *multicast* através da classe `MulticastSocket`
  - A qual é uma subclasse de `DatagramSocket`

Classe `MulticastSocket`

- Construtores, entre outros:

  `MulticastSocket()` Cria um *socket* UDP para multicast.

  `MulticastSocket(int port)` Cria um *socket* UDP associado ao porto especificado para multicast.

- Métodos, entre outros:

  `void joinGroup(InetAddress mcastaddr)` associa o *socket* a um endereço multicast:
  - A partir de então, o *socket* receberá mensagens destinadas ao grupo *multicast* correspondente.

  `void leaveGroup(InetAddress mcastaddr)` sai do grupo *multicast*;

  `void setLoopbackMode(boolean disable)` inibe/permite *loopback* de datagramas *multicast*.

  `void setTimeToLive(int ttl)` inicializa o campo TTL dos datagramas enviados através do *socket*.

Classe `MulticastSocket`: Exemplo

```
// join a Multicast group and send it a salutation

InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(6789);
s.joinGroup(group);

// send and receive messages via the MulticastSocket,
//  just as you send and receive them via a DatagramSocket,
//  i.e. with a DatagramPacket

...

// OK, enough talk -- leave the group
s.leaveGroup(group);
```

Source: Sun

Multicast em C

- Comunicação *multicast* apareceu nos finais dos anos 80, vários anos depois da definição da interface *sockets* da BSD.
- O truque para suportar *multicast* sem a introdução duma nova chamada ao sistema foi reusar (*overload*) a chamada ao sistema:

  ```
  int setsocketopt(int s, int level, int optname,
                   void *optval, socklen_t optlen))
  ```

  onde:
  `s:` identifica *socket*;
  `level:` é nível da pilha a que esta opção se aplica;
  `optname:` é a opção a modificar;
  `optval:` aponta para o novo valor da opção;
  `optlen:` é o comprimento do valor da opção.
- A chamada ao sistema `getsockopt()` permite ler o valor duma opção.

Opções para *Multicast*

- São opções ao nível do protocolo IP (`IPPROTO_IP`) (`man 7 ip`):

opção	tipo
IP_MULTICAST_TTL	int
IP_MULTICAST_LOOP	char
IP_ADD_MEMBERSHIP	struct ip_mreqn
IP_DROP_MEMBERSHIP	struct ip_mreqn
IP_MULTICAST_IF	struct ip_mreqn

onde

```
struct ip_mreqn {
  struct in_addr imr_multiaddr; /* IP multicast group addre
  struct in_addr imr_address;   /* IP address of local inte
  int            imr_ifindex;   /* interface index */
};
```

- Para permitir vários membros dum mesmo grupo num dado computador, usa-se a opção

  `SOL_SOCKET, SO_REUSEADDR, int.`

Programação *Multicast* em C

- O receptor deve:
  1. Criar um *socket* para recepção.
     - Para que vários processos possam associar-se ao grupo, deverá permitir a opção `SO_REUSEADDR`.
  2. Atribuir-lhe um nome, com o porto do grupo *multicast*.
  3. Associá-lo ao grupo (usando `setsockopt()`);
- O transmissor deve:
  - Criar um *socket* para transmissão.
    * Para que processos no mesmo computador que pertencem ao grupo possam receber as mensagens que envia deverá permitir a opção `IP_MULTICAST_LOOP`
- A recepção/transmissão de pacotes é feita como para outros *sockets UDP*.

Programação *Multicast* em C: Exemplo

```
int sd, port;
struct ip_mreq mreq;
struct sockaddr_in addr;
char buffer[1024];
void *stack;
...
sd = socket(PF_INET, SOCK_DGRAM, 0);
if( setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) != 0)
   panic("Can't reuse address/ports");
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY;
if( bind(sd, (struct sockaddr*)&addr, sizeof(addr)) != 0 )
   panic("bind failed");
if( inet_aton(strings[1], &mreq.imr_multiaddr) == 0 )
   panic("address (%s) bad", strings[1]);
mreq.imr_interface.s_addr = INADDR_ANY;
if( setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
     &mreq, sizeof(mreq)) != 0 )
   panic("Join multicast failed");
```

Source:
http://www.cs.utah.edu/~swalton/listings/sockets/programs/part4/chap17/mcast_client.c