

# Sockets



# Sockets

- Permitem comunicação *full-duplex* entre programas
  - Programas podem residir em diferentes máquinas
- Comunicação entre máquinas diferentes envolve a utilização das redes de dados e protocolos correspondentes (e.g., protocolo IP).



# Redes IP

- Rede de dados composta, ao nível físico, por nós de reencaminhamento (*routers*) e ligações entre estes.
  - Ligações têm diferentes larguras de banda.
  - O tráfego em cada ligação é variável.
  - Configuração física da rede pode variar
    - Avarias ou manutenção de ligações ou *routers*
    - Adição de novas ligações ou *routers*



# Redes IP

- Dados são transmitidos em “pacotes” (*packets*)
  - Dados a transmitir são segmentados (partidos em “bocados”) pelas camadas protocolares superiores.
  - Aumenta-se o potencial de otimização de utilização das capacidades da rede.
    - Em cada momento, os routers podem escolher a melhor rota para cada pacote.
  - Gestão de erros (nas camadas protocolares superiores) pode ser orientada ao pacote:
    - Falha num pacote não obriga a retransmitir a totalidade dos dados.



# Redes IP

- Transmissões orientadas ao “pacote”
  - Pacote = cabeçalho + *payload*
  - Informação no cabeçalho do pacote IP (e.g.):
    - Endereços IP de origem e destino
    - Comprimento
    - *Checksum*
    - *Time to live*
    - Protocolo (TCP, UDP, ICMP, etc.)
  - Pacotes podem ser perdidos e chegar por ordem diferente da do envio. Porquê?



# Redes IP

- Perdas de pacotes
  - Um *router* em sobrecarga poderá não conseguir processar todos os pacotes
  - Avaria numa ligação durante o trânsito de pacotes
  - Corrupção de dados no pacote.
- Pacotes fora de ordem
  - Rota pode ser alterada durante a transmissão de uma sequência de pacotes => pacotes mais recentes podem seguir por uma rota mais “rápida” e chegar primeiro ao destino
- Avarias e falhas de equipamento podem gerar pacotes duplicados.



# Redes IP

- Protocolos de “transporte” (exemplos: TCP, UDP):
  - Trabalham “sobre” o IP (i.e., usam as suas funcionalidades)
  - Segmentação dos dados (divisão por pacotes)
  - Oferecem o **conceito de porto** (multiplexagem).
    - Uma forma lógica de gerir vários fluxos de dados através da mesma interface de rede (e.g., bittorrent + messenger + browser).
  - Podem implementar o envio fiável de dados, através da numeração de pacotes e reenvio dos pacotes perdidos. Exemplo: TCP.



# Sockets

- A utilização de sockets tem duas fase principais:
  - Criação de um socket em cada programa que pretenda enviar/receber dados.
    - Envolve a definição do tipo de comunicação, protocolos e endereços.
    - No caso mais frequente, existe também o estabelecimento de uma ligação entre 2 pontos (2 processos).
  - O envio/receção dos dados
    - Semelhante a trabalhar com ficheiros de dados, mas com algumas particularidades.
      - A leitura dos dados é feita em série
        - Não existe o conceito de seek
      - Cada byte de informação só poderá ser lido uma vez (as leituras “consomem” os dados).





# Criação de *sockets*

- Três aspetos a definir na criação de um socket
  - Domínio/família de protocolos (PF\_INET, PF\_UNIX, PF\_IPX, etc.)
  - Tipo de comunicação (SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW, etc.)
  - Protocolo (TCP, UDP, etc.)
- Atualmente, os casos de maior interesse são:
  - PF\_INET, SOCK\_STREAM => protocolos TCP/IP
  - PF\_INET, SOCK\_DGRAM => protocolos UDP/IP



# Sockets – Domínio

- Define a família de protocolos a utilizar
  - PF\_INET
    - Para comunicações dentro da mesma máquina ou entre máquinas ligadas por uma rede *Internet Protocol* (IP).
  - PF\_UNIX (ou PF\_LOCAL)
    - Apenas para comunicações dentro da mesma máquina
  - etc.



# Sockets – Domínio

- Define um espaço de nomes
  - PF\_INET: endereço IP (xxx.xxx.xxx.xxx) + porto
    - Um endereço IP tem associado vários portos
    - Uma máquina pode ter vários endereços IP.
    - Domain Name Service (DNS) – permite associar nomes aos endereços IP. Exemplo (usar comando nslookup): portal.isep.ipp.pt => 193.136.60.7
  - PF\_UNIX: nomes de ficheiros (com permissões)



# Sockets – Tipos de comunicação

- SOCK\_STREAM - Com ligação (connection-based)
  - Garantia de entrega de dados sem perdas e mantendo a mesma sequência de envio.
  - Fluxo de bytes, não existe delimitação dos dados enviados.
  - Após o estabelecimento da ligação, os dados podem ser enviados em ambos os sentidos sem necessidade de se especificar\* o endereço de qualquer um dos extremos.
    - Análogo a uma comunicação telefónica.
    - (\*) i.e., em termos de programação
  - Tipicamente segue a arquitectura cliente-servidor:
    - Servidor aguarda ligações.
    - Cliente inicia ligação.



# Sockets – Tipos de comunicação

- SOCK\_DGRAM - Sem ligação (connectionless)
  - Orientado ao datagrama. Cada escrita gera um datagrama, cada leitura só recebe um datagrama.
    - Tamanho do datagrama é limitado
  - Datagramas podem ser perdidos ou chegar em duplicado.
  - Ordem de entrega dos datagramas pode ser alterada
  - Menor “overhead” do que no SOCK\_STREAM
  - Análogo ao envio postal
    - O mesmo socket pode receber mensagens de diferentes fontes: funciona como uma caixa de correio.
    - Podemos enviar mensagens para diferentes destinos usando o mesmo socket



# Criação de sockets

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

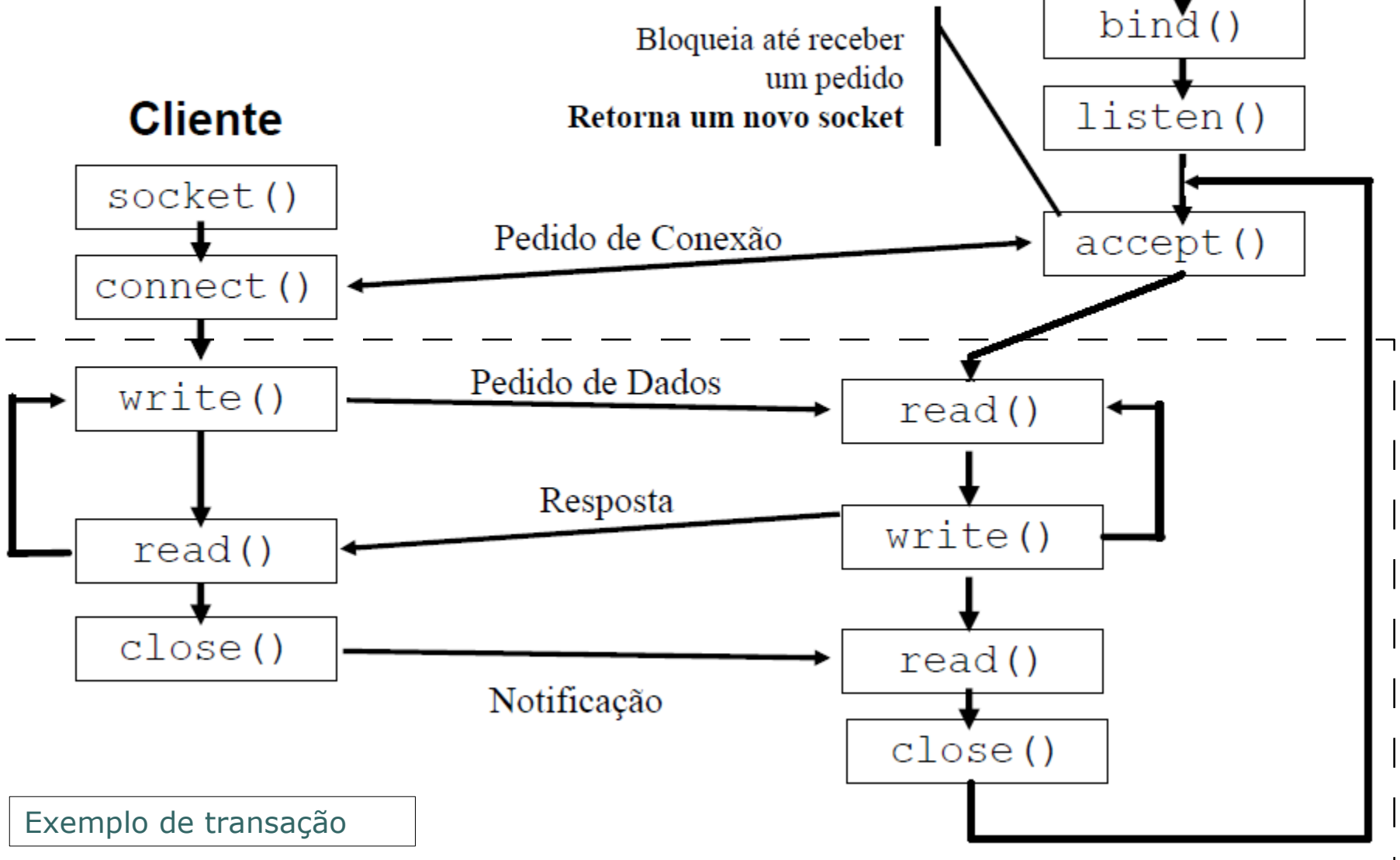
- Nos casos mais típicos, o domínio e o tipo de ligação já determinam o protocolo usado (usar 0 para escolher o default). Exemplo: PF\_INET, SOCK\_STREAM => TCP/IP
- Em caso de sucesso, retorna um descritor (identificador) do *socket*.
  - *File descriptor* - forma alternativa de identificar ficheiros abertos em sistemas UNIX.
    - Standard input: *file descriptor* 0
    - Standard output: *file descriptor* 1
    - Standard error: *file descriptor* 2



# Endereçamento do *socket*

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

- Associa o *socket* identificado por `sockfd` a um endereço (“dá um nome ao *socket*”).
- O endereço deve ser escolhido de acordo com o domínio do *socket*.
- Caso o `bind` não seja usado, o sistema atribui automaticamente um endereço ao *socket* quando este é utilizado
  - Pelas funções `connect` e `sendto`, a ser vistas mais à frente.







## Comunicação SOCK\_STREAM: *listen*, *accept*

- `int listen(int sockfd, int backlog);`
  - Indica que o *socket* será usado para atender ligações.
- `int accept(int socket,  
struct sockaddr *address,  
socklen_t *address_len);`
  - Usada pelo servidor para atender pedidos de ligação (em resposta ao *connect*) => estabelece a ligação.
  - Retorna um *socket descriptor*, para comunicação com o cliente.



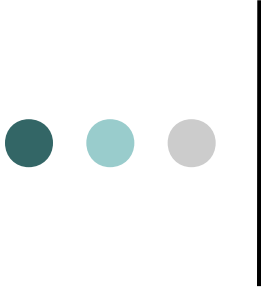
## Comunicação SOCK\_STREAM: *connect*

- `int connect(int sockfd,  
          const struct sockaddr *addr,  
          socklen_t addrlen);`
  - Usada pelo cliente para fazer o pedido de ligação.



# Operações de E/S com descritores de ficheiros

- Funções de E/S da norma POSIX:
  - `ssize_t read(int fd, void *buf, size_t count);`
    - retorna número de bytes lidos; no caso dos sockets, o valor de retorno 0 indica que a ligação foi terminada.
  - `ssize_t write(int fd, void *buf, size_t count);`
    - retorna número de bytes escritos
  - `int close(int fd);`
- Os descritores de ficheiros podem referir-se a ficheiros de dados “normais” ou a ficheiros especiais (como os sockets).
- Abertura de um ficheiro através do seu nome (pathname):
  - `int open(const char *pathname, int flags);`
    - flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, etc.



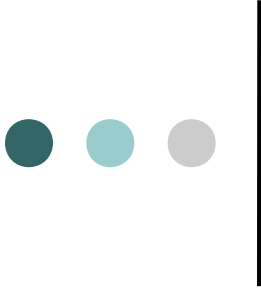
# Exemplo 1a – servidor (PF\_INET)

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

int main(int argc, char *argv[]){
    int s, ns, clilen;
    struct sockaddr_in serv_addr;

    s = socket(PF_INET, SOCK_STREAM, 0);

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    //liga-se a todas as interfaces locais, no porto 80
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(80);
    bind(s, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```



# Exemplo 1a (cont)

`//listen` e o `accept` só são usadas em `SOCK_STREAM`

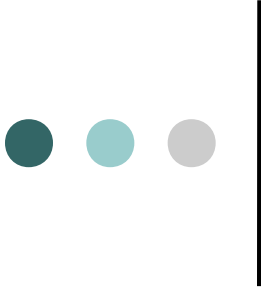
```
listen(s, 5); //no máximo 5 ligações pendentes
```

```
ns = accept(s, /* <= socket inicial */  
           NULL,  
           NULL);
```

```
... // Entrada/Saída de dados usando ns  
}
```

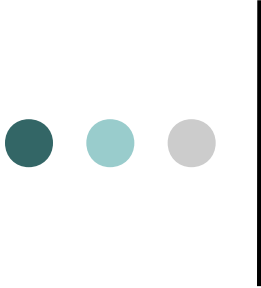
- Como detectar ligações pendentes:

- Bloquear o processo com a chamada à função `accept` (exemplo acima)
  - Pode-se criar uma nova tarefa para cada ligação.
- Alternativas: `select()`, `SIGIO`



# Exemplo 1b – cliente (PF\_INET)

```
int main (int argc, char* const argv[]){  
    char *servername = ...  
    char *port = ...  
  
    //get server address  
    struct addrinfo hints, *addrs;  
    memset(&hints, 0, sizeof(struct addrinfo));  
    hints.ai_family = AF_INET;  
    getaddrinfo(servername, port, &hints, &addrs);  
  
    int s = socket (PF_INET, SOCK_STREAM, 0);  
    connect (s, addrs->ai_addr, addrs->ai_addrlen);  
}
```



## Exemplo 2 (PF\_UNIX)

```
#include <sys/un.h>

int main(int argc, char *argv[]){
    int s, ns, clilen;
    struct sockaddr_un serv_addr, cli_addr;
    s = socket(PF_UNIX, SOCK_STREAM, 0);
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sun_family = AF_UNIX;
    strcpy(serv_addr.sun_path, "/mysock");
    bind(s, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
    listen(s,5);
    clilen = sizeof(cli_addr);
    ns = accept(s, (struct sockaddr *) &cli_addr, &clilen);
    ...
}
```



# Funções adicionais

- **gethostbyname**
  - Tradicionalmente usada para conversão de nomes para endereços.
  - Marcada como obsoleta e substituída pela **getaddrinfo**.
- **inet\_ntoa / inet\_aton** – conversão entre representação binária e textual de endereços IP.





# SOCK\_STREAM: transferências de dados

```
//cliente
```

```
char texto[100];
```

```
(...)
```

```
while(1) {  
    fgets(texto, 100, stdin)  
    if(strlen(texto)==1)  
        break;  
    write(s,  
        texto,  
        strlen(texto));  
}  
close(s);
```

```
//servidor
```

```
int n;
```

```
char buf[1000];
```

```
(...)
```

```
while(1) {  
    ns = accept(s, NULL, 0);  
    while(1) {  
        n = read(ns, buf, 1000);  
        if(n<=0)  
            break;  
        fwrite(buf, n, 1, stdout);  
    }  
    close(ns);  
}
```



# SOCK\_STREAM: transferências de dados

- Dados são enviados como uma sequência de bytes
  - A sequência de escritas não impões nenhuma delimitação nos dados recebidos no outro extremo.
    - Exemplo 1: `write(,,1476)+write(,,2)+write(,,10)`
      - Um primeiro `read()` pode ler (e.g.) 1480 bytes de uma vez e para os restantes 8 bytes terá que haver um segundo `read`;
    - Exemplo 2: `write(,,30000)`
      - `read(,,30000)` pode ler apenas alguns bytes.



# fdopen

- `FILE *fdopen(int fd, const char *mode);`
  - Associa um *stream* (`FILE *`) ao descritor de ficheiro `fd`
  - Tal como para ficheiros, o *stream* funciona como um *buffer* intermédio (com *full buffering*)
  - fazer o `fflush()` ou o `fclose()` após a escrita de todos os dados, caso contrário os dados poderão ficar no *buffer* e não ser enviados.
  - `fread` só retorna quando lê os bytes pedidos ou quando a ligação é terminada.
- *Conforming to* (extraído da página `man`):
  - The `open()` function conforms to POSIX.1-2001
  - The `fopen()` and `freopen()` functions conform to C89.
  - The `fdopen()` function conforms to POSIX.1-1990.



# Associar um *stream* a um *file descriptor*

```
//servidor
```

```
int n;  
char buf[1000];  
FILE *fp;  
(...)
```

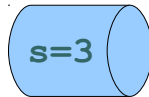
```
while(1) {  
    ns = accept(s, NULL, 0);  
    fp = fdopen(ns, "r");  
    while(1) {  
        if(fgets(buf, 1000, fp) == NULL)  
            break;  
        puts(buf);  
    }  
    fclose(fp);  
}
```

# Sockets SOCK\_STREAM

- Criação dos sockets

- Cliente

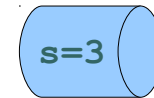
```
s = socket(..., SOCK_STREAM, 0);
```



/

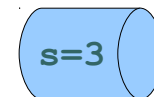
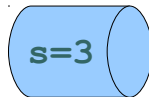
- Servidor

```
s = socket(..., SOCK_STREAM, 0);  
bind(s, ..., ...);  
listen(s, ...);
```



- Pedido de ligação

```
connect(s, ..., ...);
```



# Sockets SOCK\_STREAM

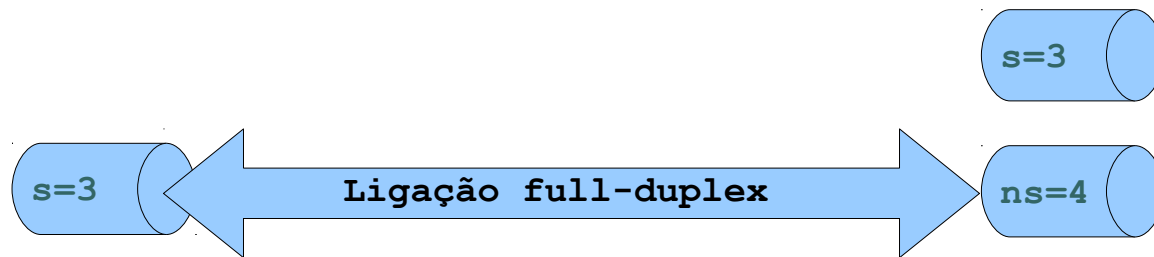
- Atendimento de ligação

- Cliente

/

Servidor

```
ns = accept(s, ..., ...);
```



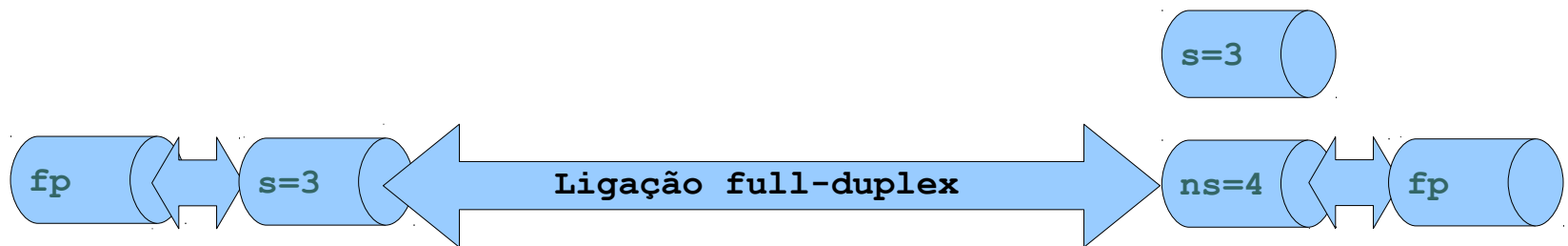
# Sockets SOCK\_STREAM

- Associação de *streams* aos descritores de *socket*

• Cliente / Servidor

```
FILE *fp = fdopen(s, "r+");
```

```
FILE *fp = fdopen(ns, "r+");
```



# Sockets – notas adicionais

- TCP (PF\_INET, SOCK\_STREAM)

- Os portos TCP associados a sockets com a função *bind* ficam indisponíveis durante algum tempo depois do processo terminar.

```
$ netstat -ln
```

```
Active Internet connections (servers and established)
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:80	127.0.0.1:59530	TIME_WAIT

- SOCK\_STREAM

- Escritas para um socket cuja ligação foi terminada geram um SIGPIPE

- PF\_INET

- Portos 0-1023 são reservados. Apenas processos do utilizador “root” podem usar esses portos.





# Servidor multiprocesso

```
int main() {  
    //Descomentar esta linha para criar um daemon  
    //daemon(0,0);  
  
    signal(SIGCHLD, SIG_IGN); //para evitar processos zombie.  
  
    //int s = socket, bind, listen  
    ...  
  
    while(1) {  
        int ns = accept(s, NULL, NULL);  
  
        int r = fork();  
        if(r == 0) {  
            //inserir código para atendimento do cliente  
  
            exit(0); //termina processo filho  
        }  
  
        close(ns);  
    }  
}
```



# Sockets

## Funções de E/S especiais

- `send()` = `write()` + flags;
- `sendto()` = `send()` + destino
  - `ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);`
  - Útil apenas com *sockets* `SOCK_DGRAM`; no caso `SOCK_STREAM`, os dois últimos parâmetros são ignorados.
- Nota: A função `connect` pode ser usado em comunicações do tipo `SOCK_DGRAM`, para evitar a necessidade de usar a função `sendto`.
  - Nesse caso, a função `connect` estabelece um endereço default para os dados enviados através do *socket*.

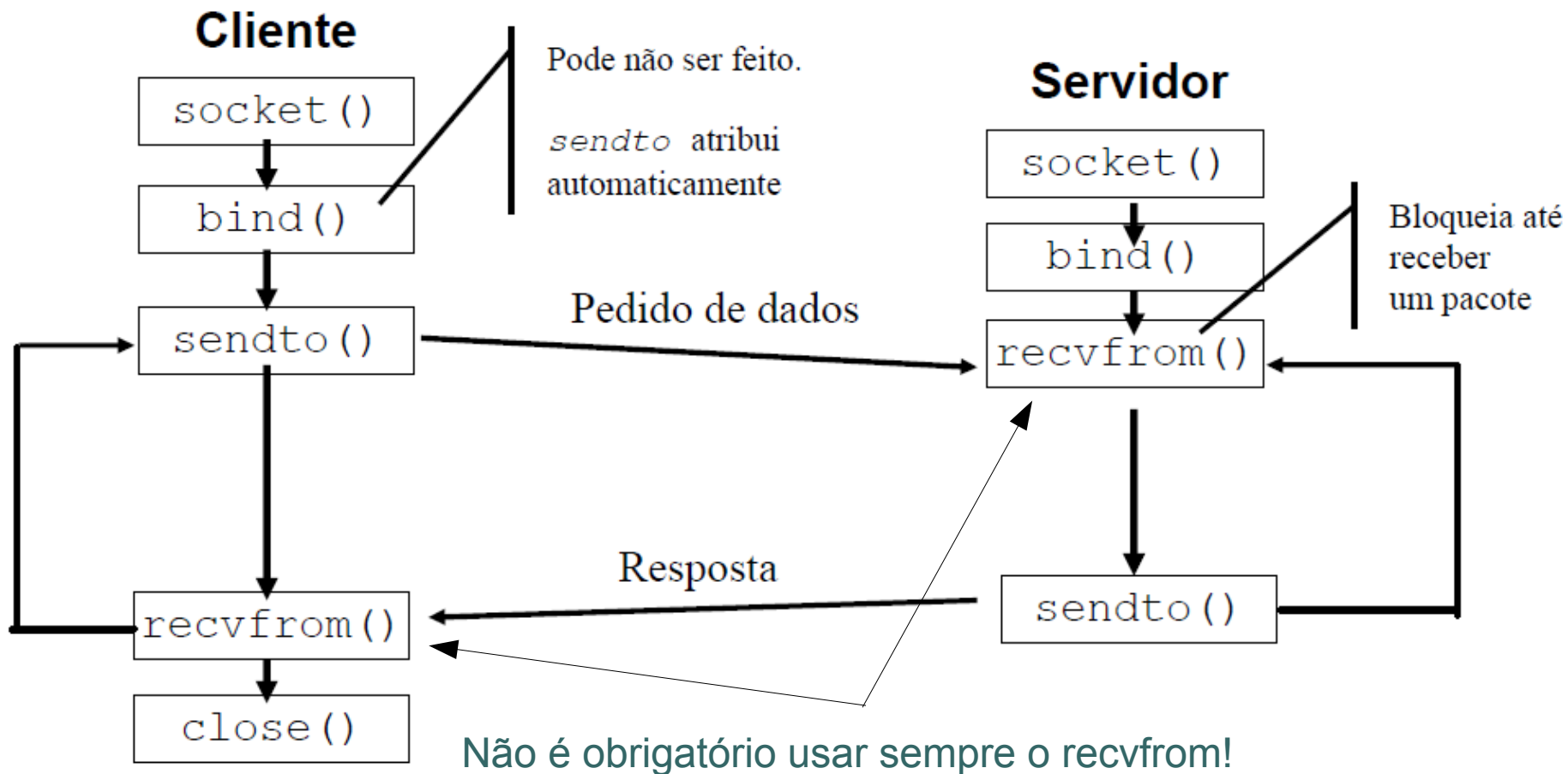


# Sockets

## Funções de E/S especiais

- `recv()` = `read()` + flags;
- `recvfrom()` = `recv()` + recepção do endereço do emissor.
  - `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen);`
  - Retorna número de bytes lidos, ou -1 em caso de erro.
  - Necessária para sockets `SOCK_DGRAM`, caso seja necessário identificar o endereço do emissor.
  - Alternativas para sockets `SOCK_STREAM`:
    - `accept` **ou** `getpeername`

# SOCK\_DGRAM





# Sockets – referências na *shell*

```
$ man 2 socket  
$ man 7 socket  
$ man 7 unix  
$ man 7 ip  
$ man 7 udp  
$ man 7 tcp  
$ man 3 gethostbyname  
$ man 2 bind  
$ man 2 listen  
$ man 2 accept  
$ man 2 connect  
$ man 2 send
```