Recursos do Sistema Linux

Sistemas Embarcados

Conteúdo

1. Acesso a arquivos
2. Processos e Sinais
3. Threads POSIX e biblioteca *pthreads*
4. Comunicação e Sincronismo entre Processos
5. **Programação usando *sockets***
6. Device Drivers

Sockets

- **Histórico**
  - Os ***sockets*** tem origem em 1983 na Universidade de Berkeley, também conhecido como a *BSD socket API*.
  - Sua primeira versão foi implementada no BSD 4.2 Unix operating system (1983) como uma API.
  - Em 1989 a UC Berkeley lança a público a versão de seu UNIX (BSD) com a API de redes livre de licenças.

Sockets

- **Função**
  - O socket é uma extensão do conceito de ***pipe*** com a possibilidade de comunicação através de rede de computadores.
  - Um processo pode utilizar socket para se comunicar com outro processo utilizando um modelo cliente/servidor tanto através da rede quanto internamente em uma mesma máquina.

Sockets

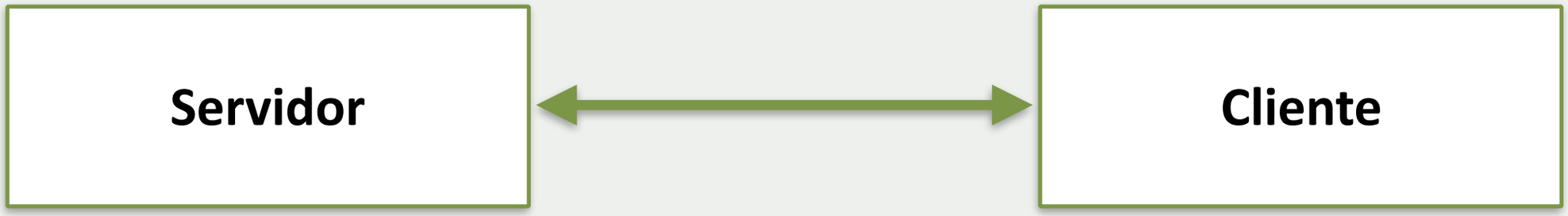
- **Suporte a Sockets**
  - No UNIX as funções de ***sockets*** são parte do sistema operacional.
  - Em outros SOs as funcionalidades de sockets são fornecidas através de bibliotecas.

Sockets

- **Suporte a Sockets**
  - Na programação de socket diferente da E/S convencional um aplicativo deve escolher um protocolo de transporte em particular, fornecer o endereço de protocolo de uma máquina remota e especificar se o aplicativo é um cliente ou um servidor. Cada socket tem diversos parâmetros e opções.
- A API de sockets define várias funções específicas para definir cada uma dos detalhes em seguida está apto a enviar e receber dados.

Sockets

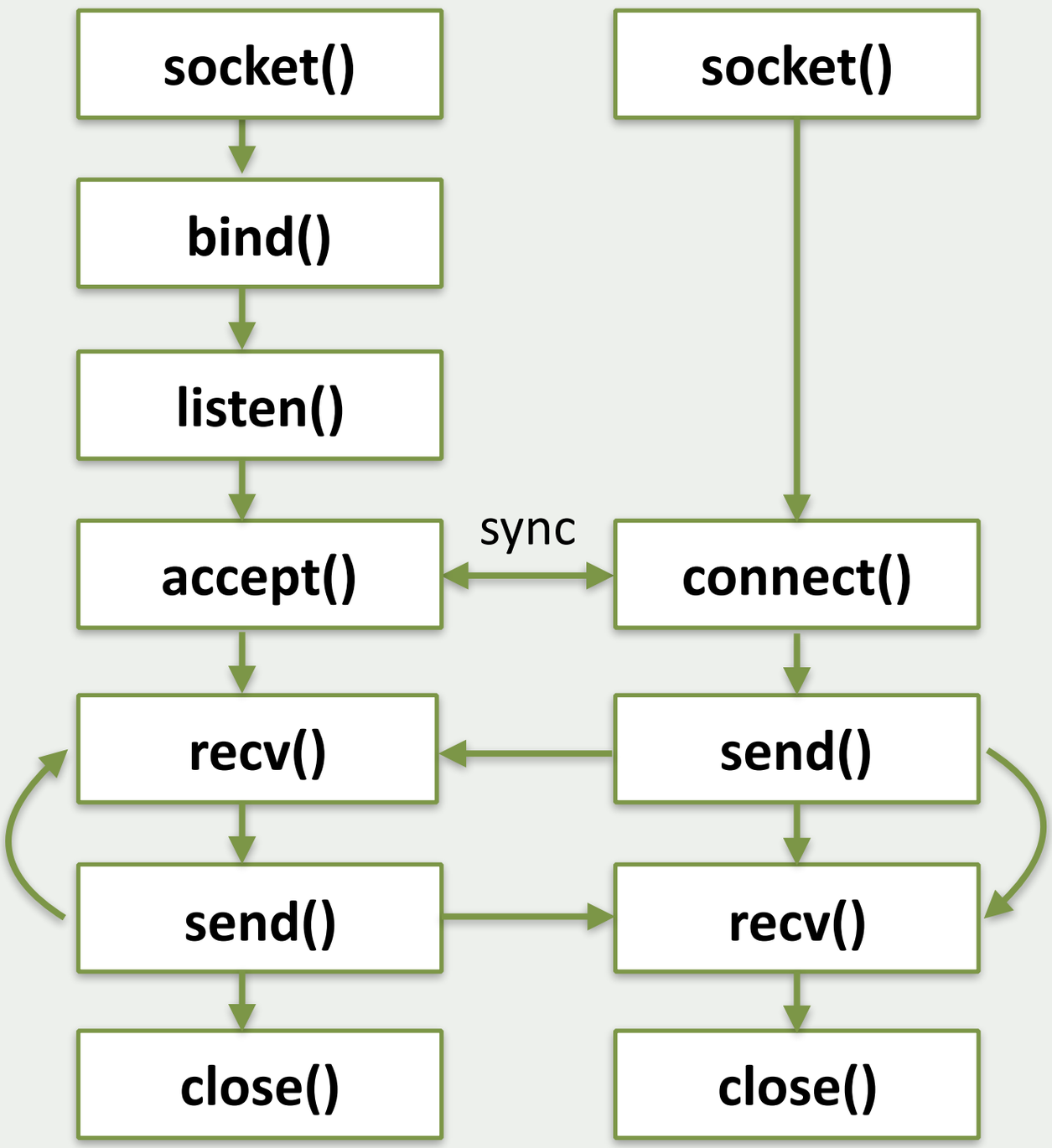
- **Comunicação Cliente-Servidor**



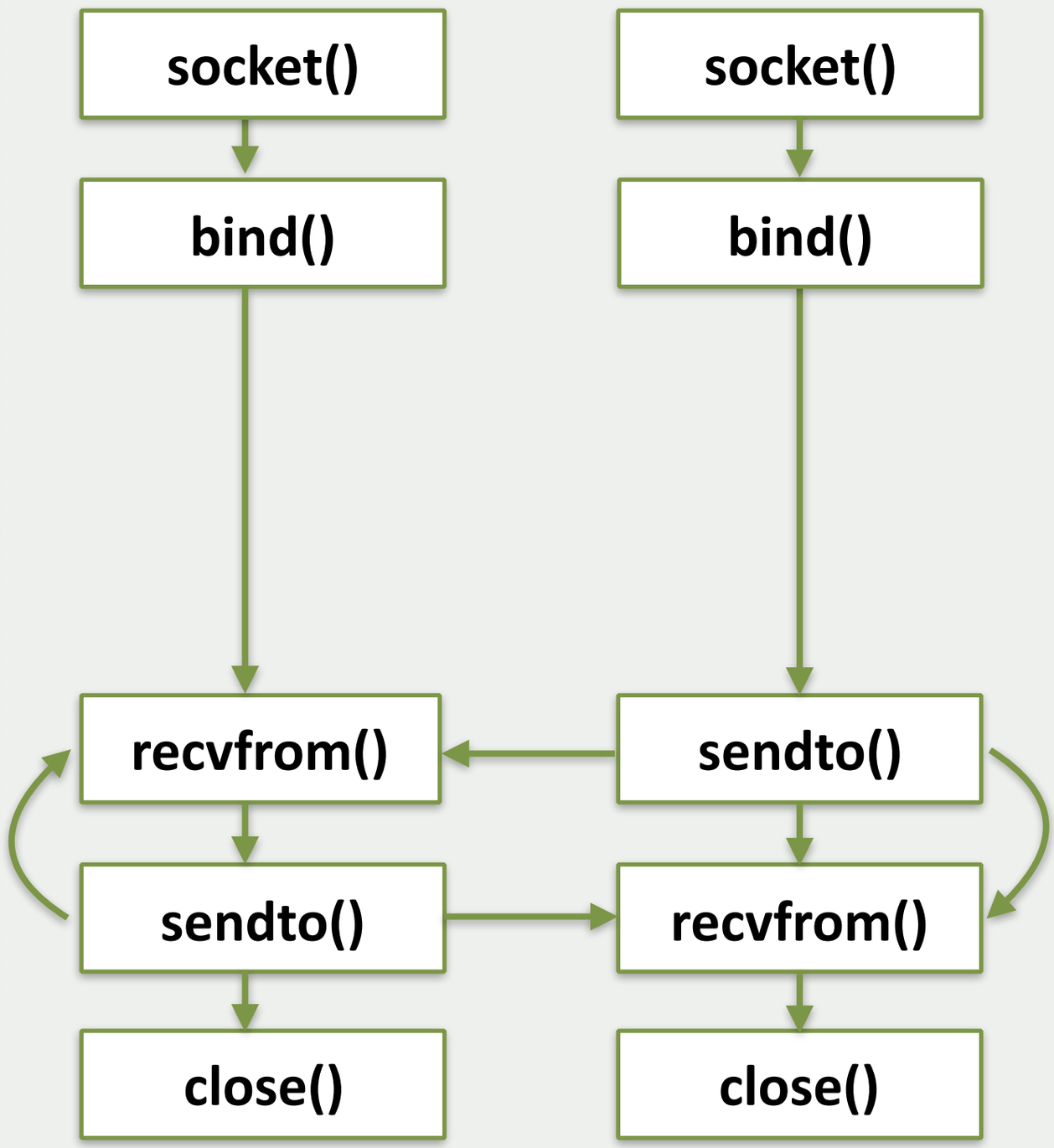
- Aguarda passivamente
- Responde aos clientes
- ***Socket passivo***

- Inicia a comunicação
- Deve saber o endereço e a porta do servidor
- ***Socket ativo***

Stream (TCP)

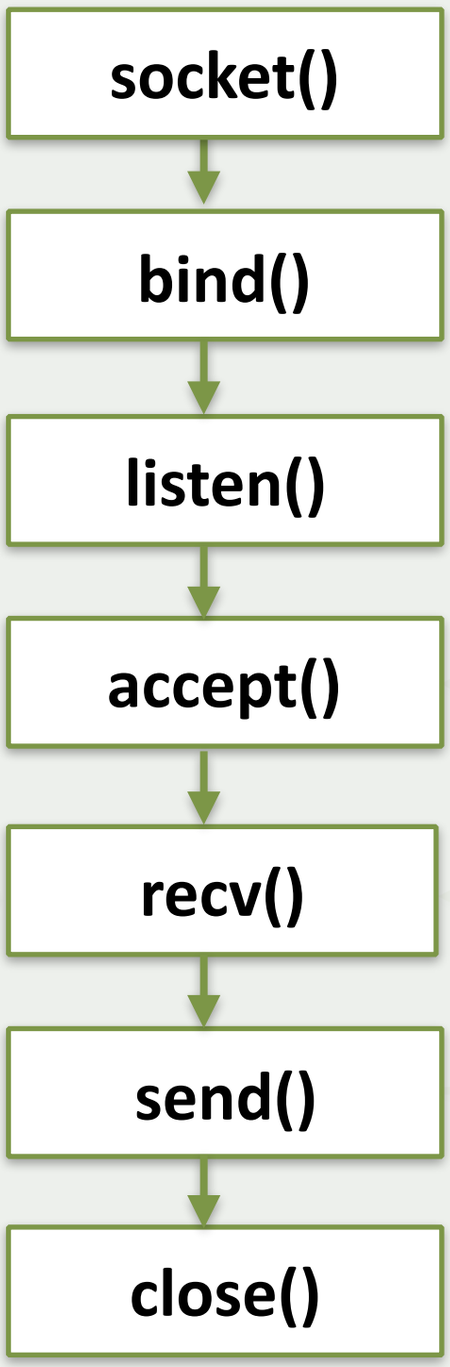


Datagram (UDP)



Stream (TCP)

Servidor



cria um socket

associa um socket de um servidor a um endereço

configura um socket para aceitar conexões

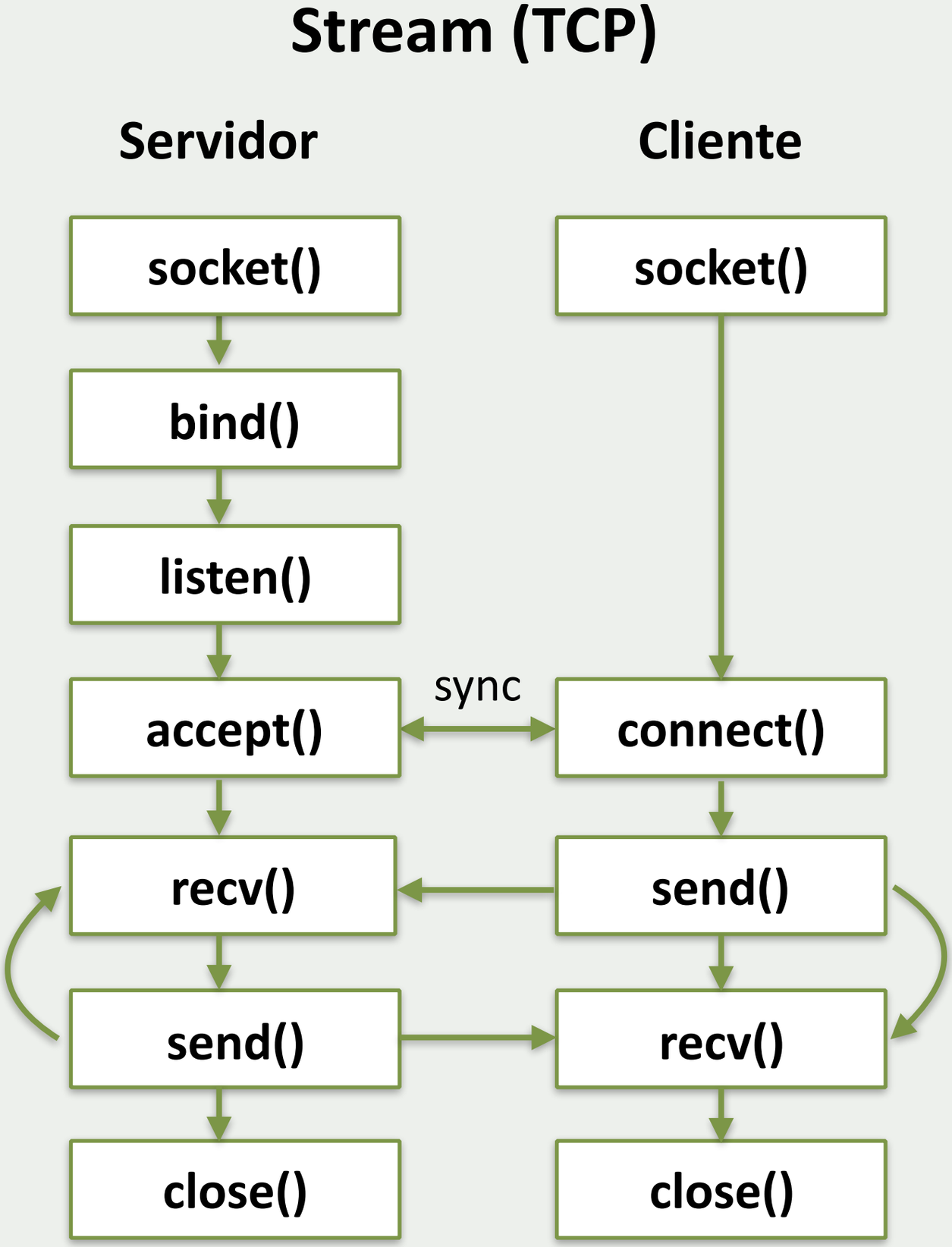
aceita uma conexão e cria um socket específico para a conexão

recebe dados (com conexão estabelecida)

envia dados (com conexão estabelecida)

destroy um socket

Stream (TCP)



cria um socket

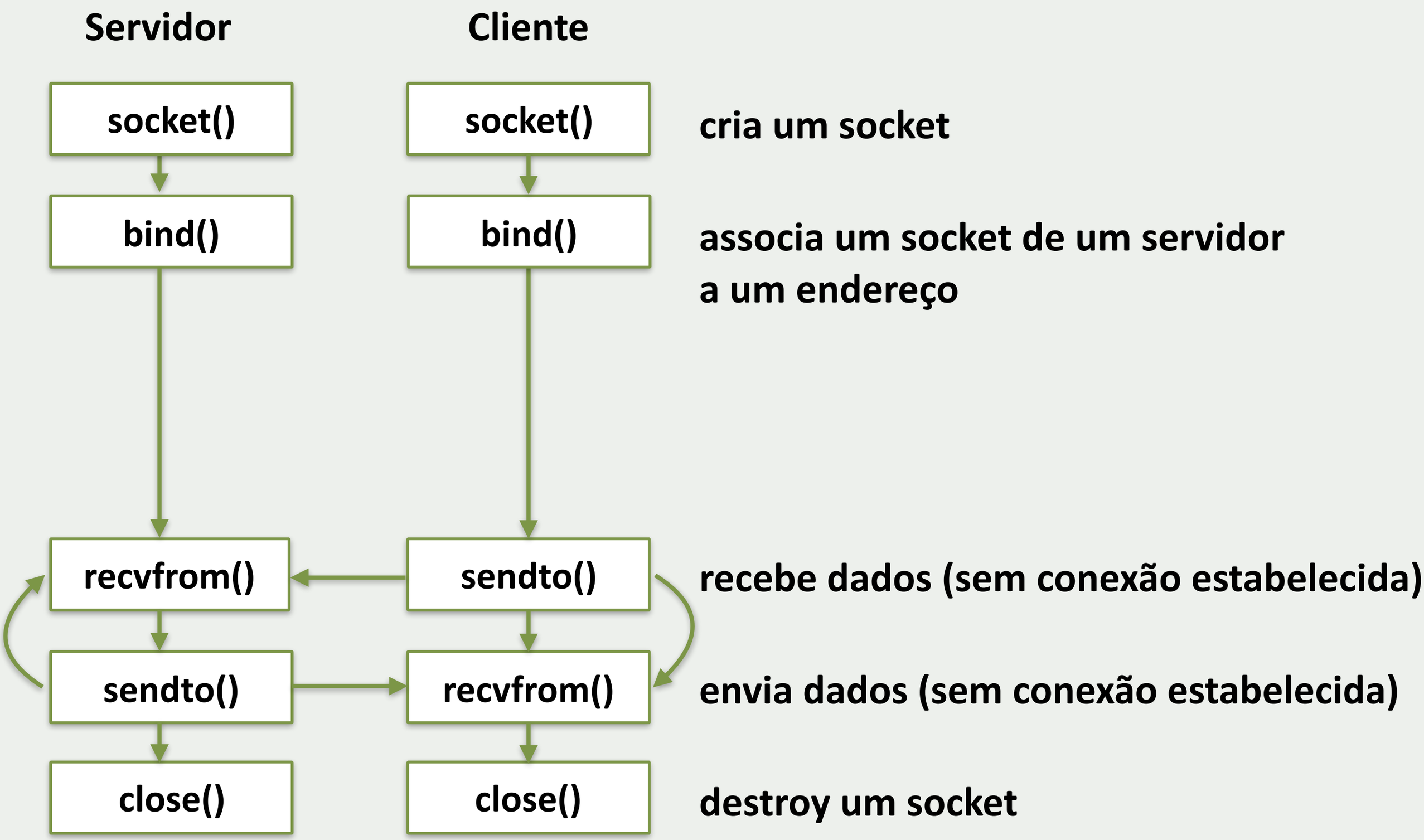
cria uma conexão entre dois sockets

envia dados (com conexão estabelecida)

recebe dados (com conexão estabelecida)

destroy um socket

Datagram (UDP)



Sockets - *socket()*

- `int socket_id socket(familia, tipo, protocolo);`
  - **socket_id**: descritor de arquivo (retorno da função)
  - **familia**: (PF protocol families) especifica a família de protocolos.
    - **PF_INET**: TCP/IP, IPv4 protocols, Internet addresses
    - **PF_UNIX** / **PF_LOCAL**: Comunicação Local, Endereço de Arquivos
    - **PF_DECnet**: protocolos da Digital Equipment Corporation

Sockets - *socket()*

- `int socket_id socket(familia, tipo, protocolo);`

  - **tipo**: tipo de comunicação
    - **SOCK_STREAM**: confiável, 2-vias, serviço baseado em conexão
    - **SOCK_DGRAM**: não-confiável, sem conexão, mensagens de tamanho máximo.
  - **protocolo**: especifica o protocolo de transporte a ser usado
    - **IPPROTO_TCP**;
    - **IPPROTO_UDP**;
    - **0**, protocolo padrão

Sockets - *close()*

- **`close`**`(socket_id);`
  - **socket_id**: descritor do socket.
- O procedimento ***close*** informa ao sistema para terminar o uso de um socket.
- Se o socket está usando um protocolo de transporte orientado à conexão, o close termina a conexão antes de fechar o socket. O fechamento de um socket imediatamente termina seu uso. O descritor é liberado, impedindo que o aplicativo envie mais dados, e o protocolo de transporte para de aceitar mensagens recebidas direcionadas para o socket, impedindo que o aplicativo receba mais dados.

Sockets - *bind()*

- **bind**(socket_id, localaddr, addrlen);
  - **socket_id**: descritor de um socket que foi criado, mas não previamente amarrado (com bind); a chamada é uma requisição que, ao socket, seja atribuído um número de porta de protocolo particular.
  - **localaddr**: estrutura que especifica o endereço local a ser atribuído ao socket (no formato sockaddr, sockaddr_in)
  - **addrlen**: inteiro que especifica o comprimento do endereço (depende do protocolo).
- Quando criado, um socket não tem um endereço local e nem um endereço remoto. Um servidor usa o procedimento ***bind*** para prover um número de porta de protocolo em que o servidor esperará por contato.

Sockets - *bind()*

- **`bind`**`(socket_id, localaddr, addrlen);`

  - Formato genérico do endereço ***sockaddr***

```
struct sockaddr{
    u_char sa_len;      /* comprimento total
                           do endereço */
    u_char sa_family; /* família do
                           endereço */
    char sa_data[14]; /* o endereço
                           propriamente dito */
};
```

Sockets - *bind()*

- **`bind`**`(socket_id, localaddr, addrlen);`

  - Para TCP/IP utiliza-se a estrutura ***sockaddr_in***

```
struct in_addr {
    unsigned long s_addr; /* Internet address (32 bits) */

}
struct sockaddr_in {
    u_char  sin_len;    /* comprimento total do endereço*/
    u_char  sin_family; /* família do endereço */
    u_short sin_port;   /* número de porta de protocolo */
    struct  in_addr sin_addr; /* endereço IP */
    char    sin_zero[8]; /* não usado (inicializado com
                            zero) */
};
```

Sockets - *bind()*

- **`bind`**`(socket_id, localaddr, addrlen);`
  - Um ***servidor*** chama ***bind*** para especificar o número da porta de protocolo em que aceitará um contato. Porém, além de um número de porta de protocolo, a estrutura ***sockaddr_in*** contém um campo para um endereço IP.
- Embora um servidor possa escolher preencher o endereço IP ao especificar um endereço, fazer isso causa problemas quando um *host* tiver múltiplas interfaces (*multihomed*) porque significa que o servidor aceita apenas requisições enviadas a um endereço específico.
- Para permitir que um servidor opere em um host com múltiplas interfaces, a API de sockets inclui uma constante simbólica especial, **INADDR_ANY**, que permite a um servidor usar uma porta específica em quaisquer dos endereços IP do computador.

Sockets - *listen()*

- `listen(socket_id, queuesize);`
  - **socket_id**: descritor de um socket que foi criado,
  - **queuesize**: comprimento para a fila de requisição do socket
- O **listen** é usado para que o sistema operacional coloque o socket em **modo passivo** aguardando o contato de clientes.
- O sistema operacional cria uma fila vazia de requisição separada para cada socket.
- À medida que chegam requisições de clientes, elas são inseridas na fila; quando o servidor pede para recuperar uma requisição recebida do socket, o sistema retorna a próxima requisição da fila.
- Se a fila está cheia quando chega uma requisição, o sistema rejeita a requisição. Ter uma fila de requisições permite que o sistema mantenha novas requisições que chegam enquanto o servidor está ocupado tratando de uma requisição anterior. O argumento ***queuesize*** permite que cada servidor escolha um tamanho máximo de fila que é apropriado para o serviço esperado.

Sockets - *accept()*

- **`newsock = accept(`**`socket_id, caddress, caddresslen)`
  - **socket_id**: descritor de um socket que após o **bind**
  - **caddersss**: endereço de uma estrutura do tipo **sockaddr**
  - **caddresslen**: ponteiro para um inteiro
- O ***accept*** preenche os campos do argumento caddress e caddresslen .
- Em seguida, cria um **novo socket** para a conexão e retorna o descritor do novo socket para quem chamou.
- O servidor usa o novo socket para se comunicar com o cliente e então fecha o socket quando termina.
- O **socket original** do servidor permanece inalterado - depois de terminar a comunicação com um cliente, o servidor usa o socket original para aceitar a próxima conexão de um cliente.

Sockets - *send()*

- `int` **`send`**`(socket_id, msg, msgLen, flags)`
  - **socket_id**: descritor do um socket após o connect/accept
  - **msg:** const void[], mensagem a ser transmitida
  - **msgLen:** int, comprimento da mensagem em bytes
  - **flags:** int, opções de configuração (normalmente 0)
  - **retorno**: quantidade de bytes enviados ou -1 para erro
- A chamada é blocante, ou seja, só continua após enviar todos os bytes

Sockets - *recv()*

- `int recv(socket_id, recBuf, bufLen, flags)`
  - **socket_id**: descritor do um socket após o connect/accept
  - **recBuf:** void[], local para armazenar bytes recebidos
  - **msgLen:** int, número de bytes recebidos
  - **flags:** int, opções de configuração (normalmente 0)
  - **retorno**: quantidade de bytes recebidos ou -1 para erro
- A chamada é blocante, ou seja, só continua após enviar todos os bytes

Sockets - *sendto()*

- `int sendto(socket_id, data, length, flags, &foreignAddr, addrlen)`
  - **socket_id**: descritor do um socket após o connect/accept
  - **msg:** *const void[],* mensagem a ser transmitida
  - **msgLen:** *int*, comprimento da mensagem em bytes
  - **flags:** *int*, opções de configuração (normalmente 0)
  - **foreignAddr**: *struct sockaddr*, endereço do destinatário
  - **addrlen**: tamanho (bytes) do foreignAddr
  - **retorno**: quantidade de bytes enviados ou -1 para erro
- A chamada é blocante, ou seja, só continua após enviar todos os bytes

Sockets - *recv()*

- `int recvfrom(socket_id, recBuf, bufLen, flags, &clientAddr, addrLen)`
  - **socket_id**: descritor do um socket após o connect/accept
  - **recBuf:** void[], local para armazenar bytes recebidos
  - **msgLen:** int, número de bytes recebidos
  - **flags:** int, opções de configuração (normalmente 0)
  - **clientAddr**: *struct sockaddr*, endereço do cliente
  - **addrlen**: tamanho (bytes) do clientAddr
  - **retorno**: quantidade de bytes recebidos ou -1 para erro
- A chamada é blocante, ou seja, só continua após enviar todos os bytes

Sockets

- Todos os servidores iniciam chamando socket para criar um socket e bind para especificar um número de porta de protocolo.
- Depois de executar as duas chamadas, um servidor que usa um protocolo de transporte sem conexão está pronto para aceitar mensagens.
- Porém, um servidor que usa um protocolo de transporte orientado à conexão exige passos adicionais antes de poder receber mensagens: o servidor deve chamar listen para colocar o socket em modo passivo, e deve então aceitar uma requisição de conexão.

Sockets

- Uma vez que uma conexão tenha sido aceita, o servidor pode usar a conexão para se comunicar com um cliente. Depois de terminar a comunicação, o servidor fecha a conexão.
- Um servidor que usa transporte orientado à conexão deve chamar o procedimento accept para aceitar a próxima requisição de conexão. Se uma requisição está presente na fila, accept retorna imediatamente; se nenhuma requisição chegou, o sistema bloqueia o servidor até que um cliente forme uma conexão.