



# AMPLIACIÓN DE SISTEMAS OPERATIVOS Y REDES

*Grado en Ingeniería Informática / Doble Grado*

*Universidad Complutense de Madrid*

---

## TEMA 2.4. Programación con Sockets

### PROFESORES:

Rubén Santiago Montero

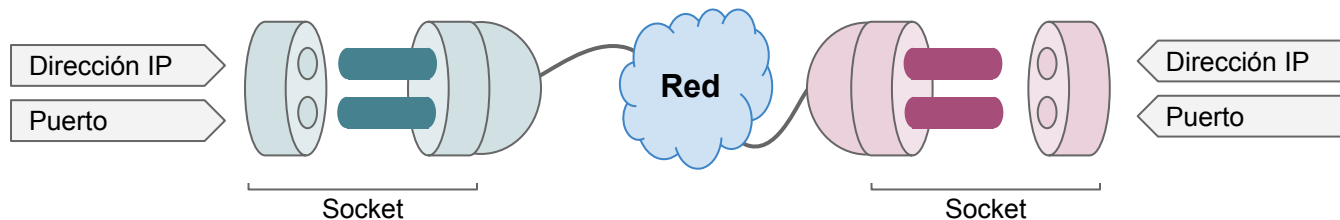
Eduardo Huedo Cuesta

Luis M. Costero Valero

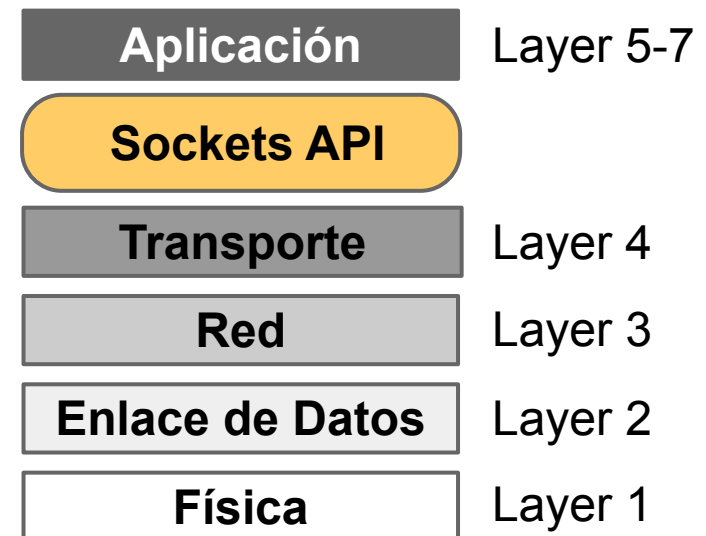
# Introducción

## Sockets

- Cada extremo del canal de comunicación establecido entre el cliente y el servidor se denomina “socket” (enchufe)
- El socket permite el intercambio de datos bidireccional entre cliente y servidor
- Cada aplicación servidor o cliente está identificada por un número de puerto



- APIs de programación con sockets:
  - **BSD Sockets API** o Sockets de Berkeley
  - WinSock API, equivalente al de BSD
  - Bindings disponibles en todos los lenguajes



# Tipos de Sockets

- Especifican la semántica de la comunicación:
  - SOCK\_STREAM
    - Flujo de bytes orientado a conexión, con entrega ordenada, fiable y bidireccional
    - Debe establecerse la conexión para poder enviar o recibir datos
    - Similar a una tubería, se envía la señal SIGPIPE si un proceso envía en un flujo interrumpido y se detecta el cierre de la conexión al recibir 0 bytes
    - Los límites de los mensajes en los datagramas entrantes no se conservan, por lo que debe marcarse el inicio y fin del mensaje (ej. `\n`, `<HTML>...</HTML>`, `{"msg": {...}}`)
  - SOCK\_DGRAM
    - Datagramas (mensajes de longitud máxima fija sin conexión y no fiables)
  - SOCK\_RAW
    - Acceso directo a los protocolos de red o de transporte, evitando el procesamiento normal de TCP/IP, lo que permite implementar nuevos protocolos en el espacio de usuario
    - Orientado a datagrama

# Protocolos de Soporte

---

- La abstracción ofrecida por los diferentes tipos de sockets se apoyan en las diferentes familias y protocolos de red
- Un **dominio** de comunicación es una familia de protocolos usados para comunicación que comparten un esquema de direccionamiento
  - **AF\_INET, AF\_INET6**: Protocolos de Internet sobre IPv4 e IPv6
  - **AF\_UNIX**: Comunicación local entre procesos de un mismo sistema
  - Otros: **AF\_IPX, AF\_X25, AF\_APPLETALK, AF\_PACKET**
- Cada **tipo** de socket se implementa usando la funcionalidad de un dominio de comunicación
  - Algunos tipos de sockets pueden no estar soportados por todos los dominios
- Se usa un **protocolo** particular de la familia para implementar cada tipo de socket
  - Normalmente, el tipo de socket determina el protocolo dentro de un dominio

# Creación de Sockets

<sys/types.h>  
<sys/socket.h>

POSIX+BSD

- Crear un socket:

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

- domain es la familia de protocolos
- type es el tipo de socket
- protocol se suele dejar a 0 o, en AF\_INET y AF\_INET6, puede ser:
  - IPPROTO\_TCP para SOCK\_STREAM
  - IPPROTO\_UDP para SOCK\_DGRAM
- Devuelve un descriptor de fichero para el socket

- Creación de sockets IPv4:

```
tcp_sd = socket(AF_INET, SOCK_STREAM, 0);  
udp_sd = socket(AF_INET, SOCK_DGRAM, 0);
```

- Creación de sockets IPv6:

```
tcp6_sd = socket(AF_INET6, SOCK_STREAM, 0);  
udp6_sd = socket(AF_INET6, SOCK_DGRAM, 0);
```

- La implementación de IPv6 es compatible casi totalmente con IPv4

# Direcciones de Sockets

- Direcciones de sockets IPv4:

```
struct sockaddr_in {  
    sa_family_t    sin_family; // Familia: AF_INET  
    in_port_t      sin_port;   // Puerto  
    struct in_addr sin_addr;    // Dirección IPv4  
};  
→ struct in_addr {  
    uint32_t        s_addr;     // 32 bits dirección IP  
};
```

- **Puerto (sin\_port)**
  - Por debajo de 1024 sólo para procesos con privilegio y suelen ser bien conocidos
  - Asociados a los protocolos superiores TCP y UDP
- **Dirección IPv4 (sin\_addr)**
  - Dirección de red local o remota
  - Se puede inicializar o asignar con las constantes INADDR\_ANY (0.0.0.0) e INADDR\_LOOPBACK (127.0.0.1)

# Direcciones de Sockets

- Direcciones de sockets IPv6:

```
struct sockaddr_in6 {  
    sa_family_t    sin6_family;    // Familia: AF_INET6  
    in_port_t      sin6_port;      // Número de puerto  
    uint32_t       sin6_flowinfo;  // Id del flujo  
    struct in6_addr sin6_addr;      // Dirección IPv6  
    uint32_t       sin6_scope_id;  // Índ. de zona (link-local)  
};  
→ struct in6_addr {  
    unsigned char  s6_addr[16];    // Dir. IPv6 de 128 bits  
};
```

- La dirección IPv6 (`sin6_addr`) se puede inicializar con las constantes `IN6ADDR_ANY_INIT` e `IN6ADDR_LOOPBACK_INIT`, o asignar a las variables `in6addr_any (:::)` e `in6addr_loopback (:::1)`

# Gestión de Direcciones

```
<sys/types.h>  
<sys/socket.h>  
<netdb.h>
```

POSIX

- Traducción de nombres a direcciones:

```
int getaddrinfo(const char *node, const char *service,  
               const struct addrinfo *hints, struct addrinfo **res);
```

- node hace referencia al host, y puede ser:
  - Un nombre de host, que se resuelve usando gethostbyname(3)
  - Una dirección IPv4 en notación decimal de punto (ej. “192.168.0.1”)
  - Una dirección IPv6 en notación hexadecimal abreviada (ej. “fe80::1:2”)
  - NULL, para especificar el host local
- service hace referencia al puerto, y puede ser:
  - Un nombre del servicio, según /etc/services (ej. “http”)
  - Un número entero en decimal (ej. “80”)
  - NULL, para no especificar ninguno
- hints establece algunos criterios de búsqueda
- res se usa para devolver una lista de direcciones de socket
  - El host tiene varios interfaces o soporta varios protocolos (ej. IPv4 e IPv6)
  - El servicio soporta varios protocolos (ej. telnet → tcp/23 y udp/23)

Comprobar valor de retorno y códigos de error: getaddrinfo(3) → gai\_strerror()



# Gestión de Direcciones

```
struct addrinfo {  
    int          ai_flags;      // Opciones para filtrado (hints)  
    int          ai_family;  
    int          ai_socktype;  
    int          ai_protocol;  
    socklen_t    ai_addrlen;    // Resultado (res)  
    struct sockaddr *ai_addr;  
    char         *ai_canonname;  
    struct addrinfo *ai_next;  
};
```

- Opciones de filtrado (hints, el resto de campos deben ser 0 o NULL):
  - ai\_family: AF\_INET para IPv4, AF\_INET6 para IPv6 o AF\_UNSPEC para ambos
  - ai\_socktype y ai\_protocol: Tipo de socket y protocolo
  - ai\_flags: Opciones, ej. AI\_PASSIVE para devolver 0.0.0.0 ó :: si node=NULL (si no, devuelve 127.0.0.1 ó ::1)
- Resultado (res):
  - ai\_addr y ai\_addrlen: Puntero a la dirección y tamaño en bytes
  - ai\_canonname: Nombre oficial del host si AI\_CANONNAME en ai\_flags
  - ai\_next: Puntero al siguiente resultado (lista enlazada)

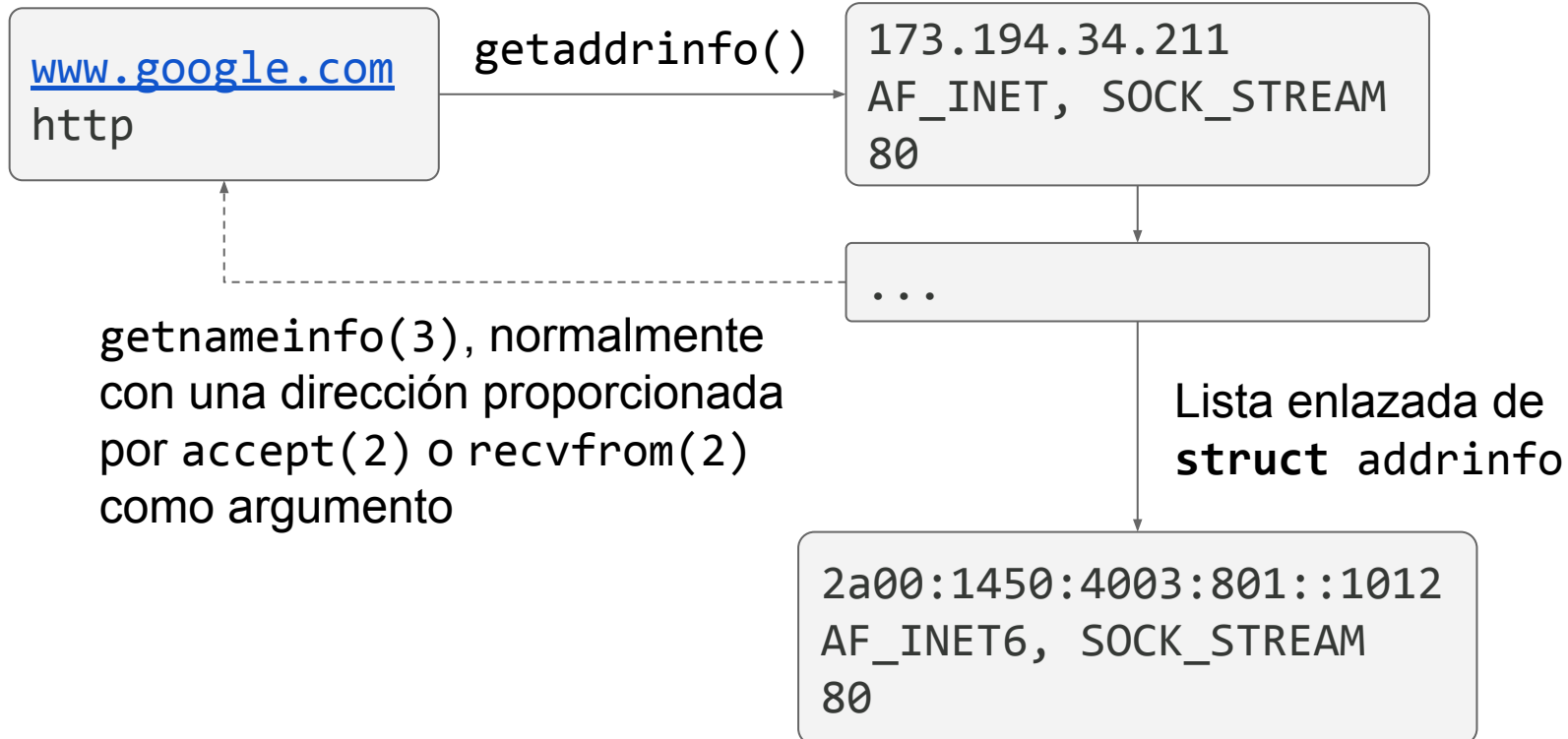
# Gestión de Direcciones

```
<sys/types.h>  
<sys/socket.h>  
<netdb.h>
```

POSIX

- Traducción de direcciones a nombres:

```
int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,  
               char *host, socklen_t hostlen,  
               char *serv, socklen_t servlen, int flags)
```



`getnameinfo(3)`, normalmente  
con una dirección proporcionada  
por `accept(2)` o `recvfrom(2)`  
como argumento

# Conversión de Direcciones y Valores

<arpa/inet.h>

POSIX

- Convertir direcciones entre formato binario y de texto:

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

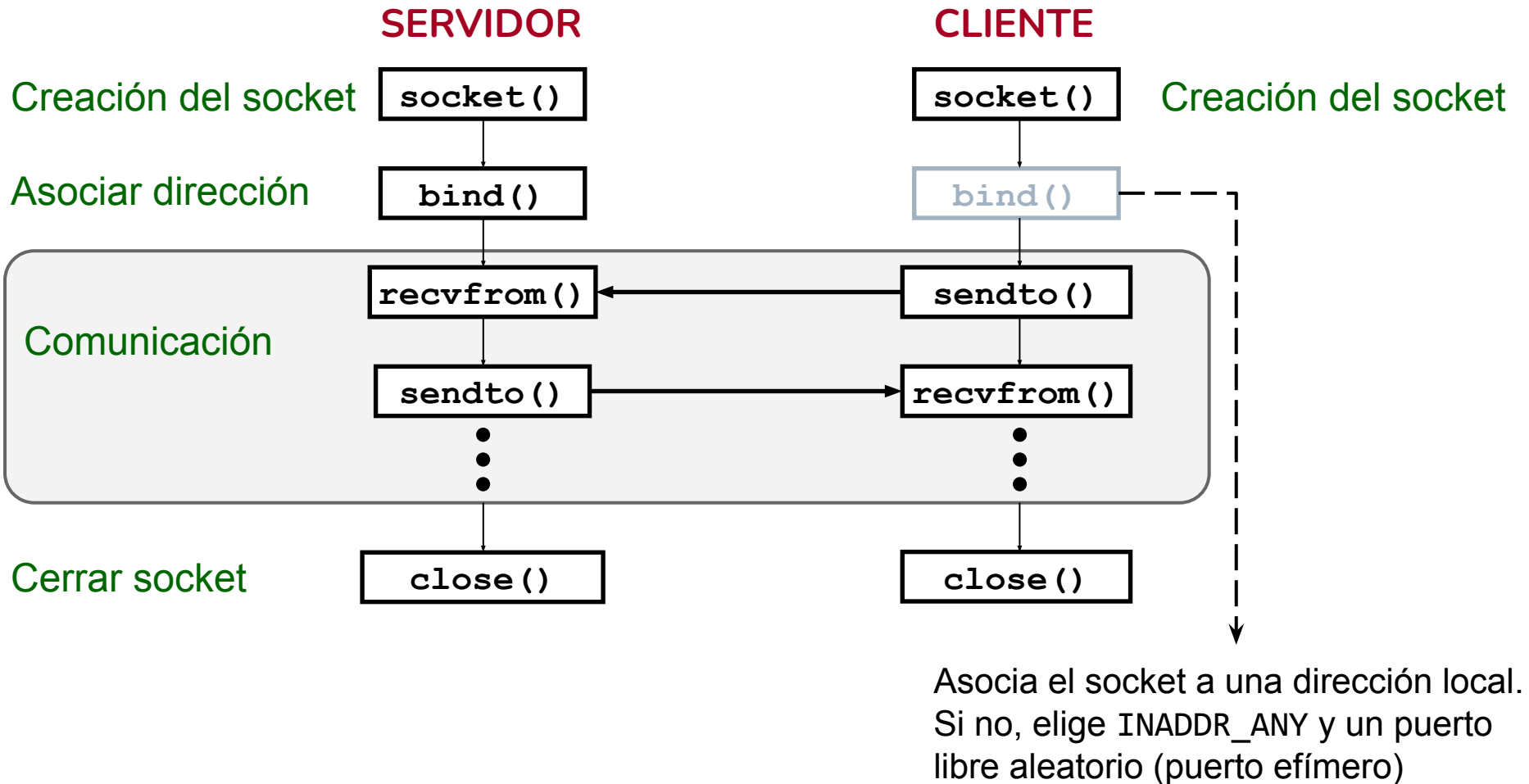
- af es una familia de protocolos (AF\_INET o AF\_INET6)
- Los argumentos de tipo **void \*** contienen la estructura de la dirección en binario (**struct in\_addr** o **struct in6\_addr**)
- Los argumentos de tipo **char \*** contienen la representación de la dirección como texto (representación decimal de punto o hexadecimal abreviada)
- Convertir valores entre orden de byte de red y de host:

```
uint32_t htonl(uint32_t hostlong);  
uint16_t htons(uint16_t hostshort);  
uint32_t ntohl(uint32_t netlong);  
uint16_t ntohs(uint16_t netshort);
```

  - Los datos se envían en orden de byte de red (*big-endian*), por lo que puede ser necesario convertirlos al orden de la arquitectura del procesador
  - Las direcciones y puertos se almacenan en orden de byte de red

# Sockets UDP: Patrón de Comunicación

- Sockets tipo SOCK\_DGRAM para la familia AF\_INET y AF\_INET6



**NOTA:** Este patrón de comunicaciones es también válido para AF\_UNIX

# Asignación de Direcciones

<sys/socket.h>

POSIX

- Asignar una dirección local a un socket:

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

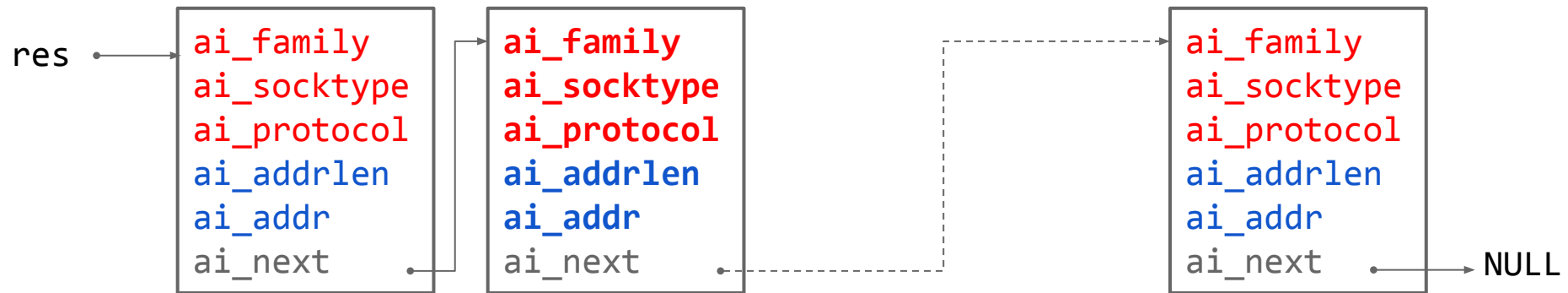
- Obligatorio en el servidor, para poder recibir
  - sd es el descriptor creado con socket(2)
  - addr es de tipo genérico para acomodar la dirección de cada familia
  - addrlen es la longitud de la dirección, que depende del tipo (usar **sizeof()**)
- Asignar una dirección remota a un socket:

```
int connect(int sd, const struct sockaddr *addr, socklen_t addrlen);
```

- Con SOCK\_STREAM es obligatorio en el cliente para iniciar la conexión
- Con SOCK\_DGRAM asigna la dirección a la que se envían los datagramas por defecto y la única desde la que se reciben los datagramas

# Asignación de Direcciones

```
getaddrinfo(node, service, &hints, &res);
```



```
sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

```
// Servidor
```

```
bind(sock, (struct sockaddr *) res->ai_addr, res->ai_addrlen);
```

```
// Cliente (opcional para SOCK_DGRAM (UDP))
```

```
connect(sock, (struct sockaddr *) res->ai_addr, res->ai_addrlen);
```

# Envío y Recepción de Datos

<sys/socket.h>

POSIX

- Enviar y recibir datos:

```
ssize_t sendto(int sd, const void *buf, size_t len, int flags,  
               const struct sockaddr *addr, socklen_t addrlen);
```

```
ssize_t recvfrom(int sd, void *buf, size_t len, int flags,  
                 struct sockaddr *addr, socklen_t *addrlen);
```

- Se usan con SOCK\_DGRAM para especificar u obtener la dirección del otro extremo en addr (con SOCK\_STREAM se ignora), de tamaño addrlen
- sendto(2) envía len bytes de buf
- recvfrom(2) recibe hasta len bytes en buf
  - addrlen es un argumento valor-resultado, que inicialmente contiene el tamaño de la variable addr y se modifica para indicar el tamaño real de la dirección devuelta
  - Se bloquea si no hay mensajes disponibles: Se puede usar select(2) o el modo no bloqueante (flag MSG\_DONTWAIT)

# Almacenamiento de Direcciones

- Para escribir aplicaciones compatibles con IPv4 e IPv6, deben eliminarse las dependencias en el formato de las direcciones
  - **struct** `sockaddr` solo evita advertencias del compilador (tiene el mismo tamaño que **struct** `sockaddr_in`)
  - La nueva **struct** `sockaddr_storage` permite almacenar tanto **struct** `sockaddr_in` como **struct** `sockaddr_in6`
- Ejemplo:

```
struct sockaddr_storage addr;  
socklen_t addrlen = sizeof(addr);  
b = recvfrom(sd, buf, 80, 0, (struct sockaddr *) &addr, &addrlen);
```



# Resumen: Esquema Servidor UDP

```
hints.ai_flags      = 0;
hints.ai_family     = AF_UNSPEC;    // IPv4 or IPv6
hints.ai_socktype   = SOCK_DGRAM;

rc = getaddrinfo(argv[1], argv[2], &hints, &result);

sd = socket(result->ai_family, result->ai_socktype, 0);

bind(sd, (struct sockaddr *) result->ai_addr, result->ai_addrlen);

while (1) {
    addrlen = sizeof(addr);
    c = recvfrom(sd, buf, 80, 0, (struct sockaddr *) &addr, &addrlen);

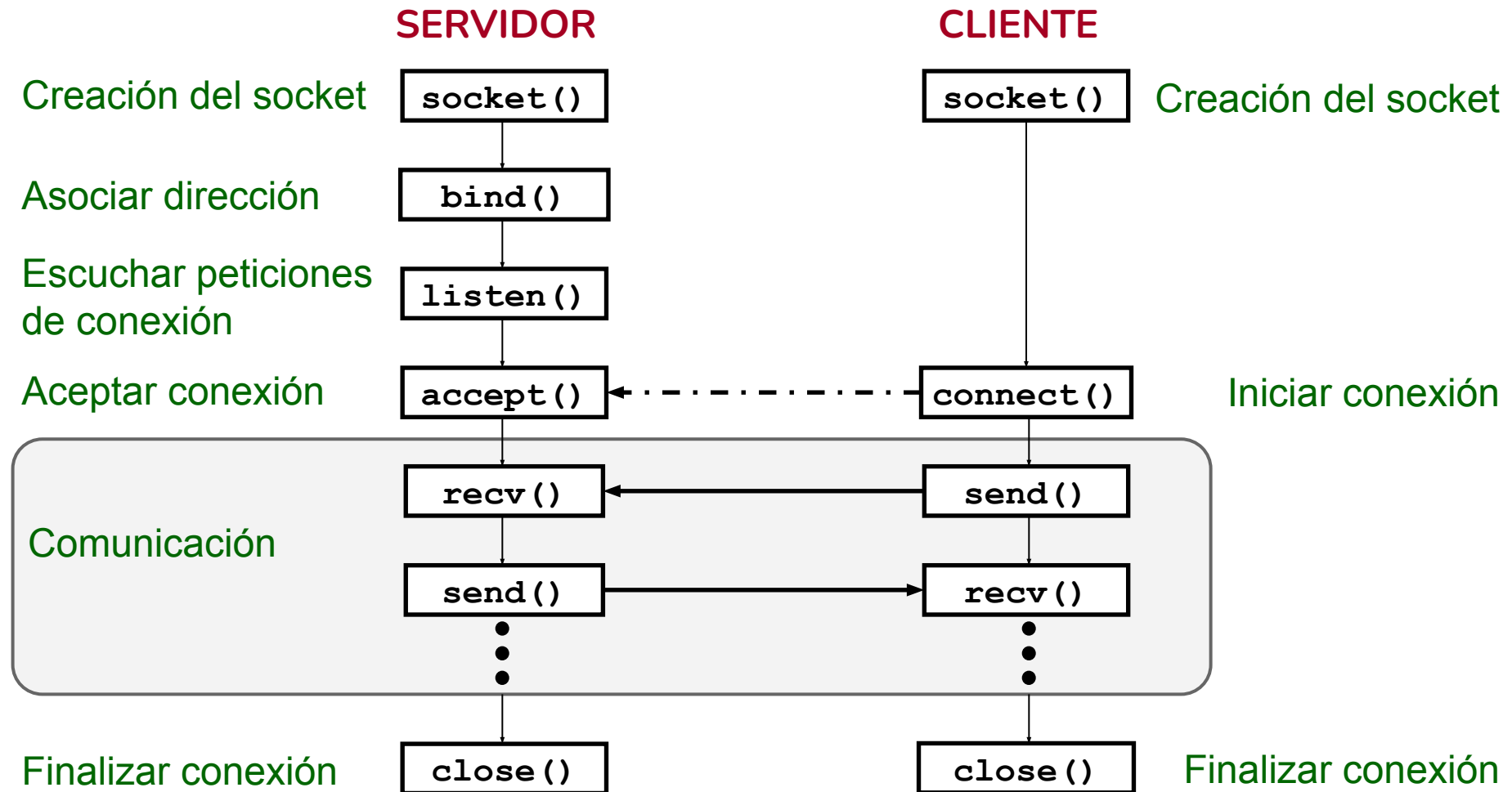
    getnameinfo((struct sockaddr *) &addr, addrlen, host, NI_MAXHOST,
                serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);

    printf("Received %d bytes from %s:%s\n", c, host, serv);

    sendto(sd, buf, c, 0, (struct sockaddr *) &addr, addrlen);
}
```

# Sockets TCP: Patrón de Comunicación

- Sockets de tipo `SOCK_STREAM` para la familia `AF_INET` y `AF_INET6`



**NOTA:** Este patrón de comunicaciones es también válido para `AF_UNIX`

# Gestión de la Conexión

<sys/socket.h>

POSIX

- Escuchar conexiones en un socket:

```
int listen(int sd, int backlog);
```

- Con SOCK\_STREAM se usa en el servidor para poner el socket en modo de escucha para aceptar peticiones de conexión
- backlog es el tamaño de la cola *accept* (conexiones completamente establecidas esperando ser aceptadas)
  - No confundir con la cola SYN (conexiones incompletas)

- Aceptar una conexión en un socket:

```
int accept(int sd, struct sockaddr *addr, socklen_t *addrlen);
```

- Con SOCK\_STREAM se usa en el servidor para aceptar una conexión establecida y crear un nuevo socket conectado
- Se bloquea si no hay conexiones pendientes
  - Se puede usar select(2) o el modo no bloqueante
- addr devuelve la dirección del cliente que se ha conectado
- addrlen es un argumento valor-resultado
- Devuelve un nuevo descriptor de socket para gestionar la conexión

# Envío y Recepción de Datos

<sys/socket.h>

POSIX

- Enviar y recibir datos:

```
ssize_t send(int sd, const void *buff, size_t len, int flags);  
ssize_t recv(int sd, void *buff, size_t len, int flags);
```

- Se usan normalmente con sockets SOCK\_STREAM
- send(2) envía len bytes de buff
  - Con SOCK\_DGRAM hay que usar connect(2) previamente
  - Si el mensaje es demasiado grande, no se envían datos (EMSGSIZE)
- recv(2) recibe hasta len bytes en buff
  - Con SOCK\_DGRAM, el mensaje se debe leer en una sola operación (tamaño del búfer) para no perder datos
- Ambas pueden bloquearse
  - send(2) se bloquea si el mensaje no cabe en el búfer de envío
  - recv(2) se bloquea si no hay mensajes disponibles en el socket
  - Se puede usar select(2) o el modo no bloqueante (flag MSG\_DONTWAIT)

# Resumen: Esquema Servidor TCP

```
hints.ai_flags      = 0;
hints.ai_family     = AF_UNSPEC;    // IPv4 or IPv6
hints.ai_socktype   = SOCK_STREAM;

rc = getaddrinfo(argv[1], argv[2], &hints, &result);

sd = socket(result->ai_family, result->ai_socktype, 0);
bind(sd, (struct sockaddr *) result->ai_addr, result->ai_addrlen);
listen(sd, 5);

while (1) {
    addrlen = sizeof(addr);
    clisd = accept(sd, (struct sockaddr *) &addr, &addrlen);

    getnameinfo((struct sockaddr *) &addr, addrlen, host, NI_MAXHOST,
                serv, NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV);
    printf("Connection from %s:%s\n", host, serv);

    while (c = recv(clisd, buf, 80, 0)) { // Check message!
        send(clisd, buf, c, 0);
    }
    close(clisd);
}
```

# Opciones de Sockets

<sys/socket.h>

POSIX

- Pueden fijarse y consultarse diversas opciones en el socket:

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

- `level` especifica el nivel de la capa de protocolos donde aplica la opción:
  - API de sockets: `SOL_SOCKET`
  - Protocolo: `IPPROTO_IP`, `IPPROTO_IPV6`, `IPPROTO_TCP`, `IPPROTO_UDP`
- `optname` especifica la opción, que acepta un valor `optval` de un tipo específico (`void *`) y tamaño `optlen`
- Muchas opciones se pueden configurar vía `sysctl` o `/proc`

# Opciones para Sockets

- Nivel de API de sockets (SOL\_SOCKET):
  - SO\_KEEPALIVE: Activa el mecanismo de *keepalive* en sockets SOCK\_STREAM
  - SO\_BROADCAST: Permite a sockets SOCK\_DGRAM usar direcciones de *broadcast*
  - SO\_REUSEADDR: Activa la reutilización de direcciones locales en TIME\_WAIT
  - SO\_SNDBUF y SO\_RCVBUF: Obtienen o establecen el tamaño de los buffers de envío y recepción (actualmente se autoajusta en función de la latencia y el ancho de banda)
- Nivel de protocolo TCP (IPPROTO\_TCP):
  - TCP\_NODELAY: Desactiva el algoritmo de Nagle
  - TCP\_QUICKACK: Desactiva los ACKs retrasados
- Nivel de protocolo IPv4 (IPPROTO\_IP):
  - IP\_ADD\_MEMBERSHIP e IP\_DROP\_MEMBERSHIP: Gestión de grupos multicast
  - IP\_MTU: Obtiene el MTU de la ruta
  - IP\_MTU\_DISCOVER: Activa el algoritmo Path MTU Discovery
  - IP\_OPTIONS, IP\_TTL e IP\_TOS: Obtienen o establecen campos del datagrama

socket(7), tcp(7), ip(7)

# Soporte para Clientes IPv4 e IPv6

- Alternativas para servidores que soporten clientes IPv4 e IPv6
- Crear **un único socket IPv6 para ambas versiones** (*dual stack*):
  - Deshabilitar la opción IPV6\_V6ONLY en el socket (su valor se define en el parámetro `net.ipv6.bindv6only`, que por defecto está deshabilitado)

```
int on = 0;
setsockopt(sd, IPPROTO_IPV6, IPV6_V6ONLY, (void *) &on,
           sizeof(on));
```
  - Asociar (con `bind(2)`) a `:: (in6addr_any)`
    - Usa direcciones IPv6 mapeadas a IPv4 (`192.168.0.1` ⇒ `::FFFF:192.168.0.1`)
    - No está soportado en todos los sistemas
- Crear **dos sockets, uno para cada versión**:
  - Habilitar IPV6\_V6ONLY en el socket IPv6 si se va a asociar a `::`
  - Obtener las direcciones válidas para crear un socket con cada versión



# Servidores Concurrentes

---

## Necesidad

- El servidor debe atender a varios clientes concurrentemente
- En general, las llamadas son bloqueantes
  - `accept(2)` espera a que se establezcan conexiones de clientes
  - `recv(2)` y `recvfrom(2)` esperan a que lleguen de datos
  - `send(2)` espera si el mensaje no cabe en el *buffer* de envío

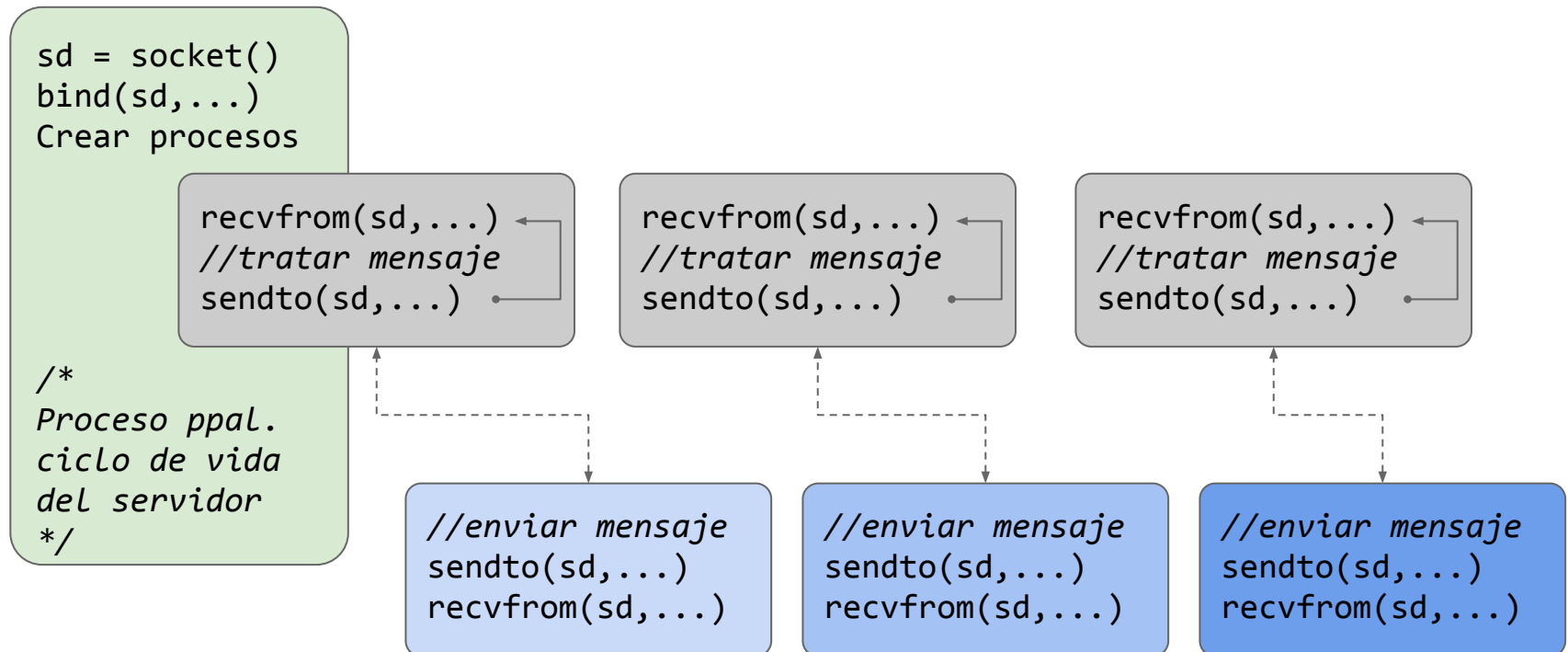
## Herramientas

- Los hilos comparten un espacio de direcciones y los descriptores (sockets) y los procesos heredan los descriptores (sockets)
- Las operaciones son concurrentes sobre descriptores de socket
  - Múltiples hilos pueden llamar a `accept(2)` para aceptar una conexión
  - Múltiples hilos pueden llamar a `recvfrom(2)` para recibir datos
  - Todos los hilos se bloquearán en la llamada y uno de ellos se desbloqueará cuando se establezca una conexión o lleguen datos
- También se puede usar multiplexación de E/S síncrona (`select(2)`), pero la lógica del programa es mucho más compleja

# Servidores Concurrentes: SOCK\_DGRAM

## Patrón

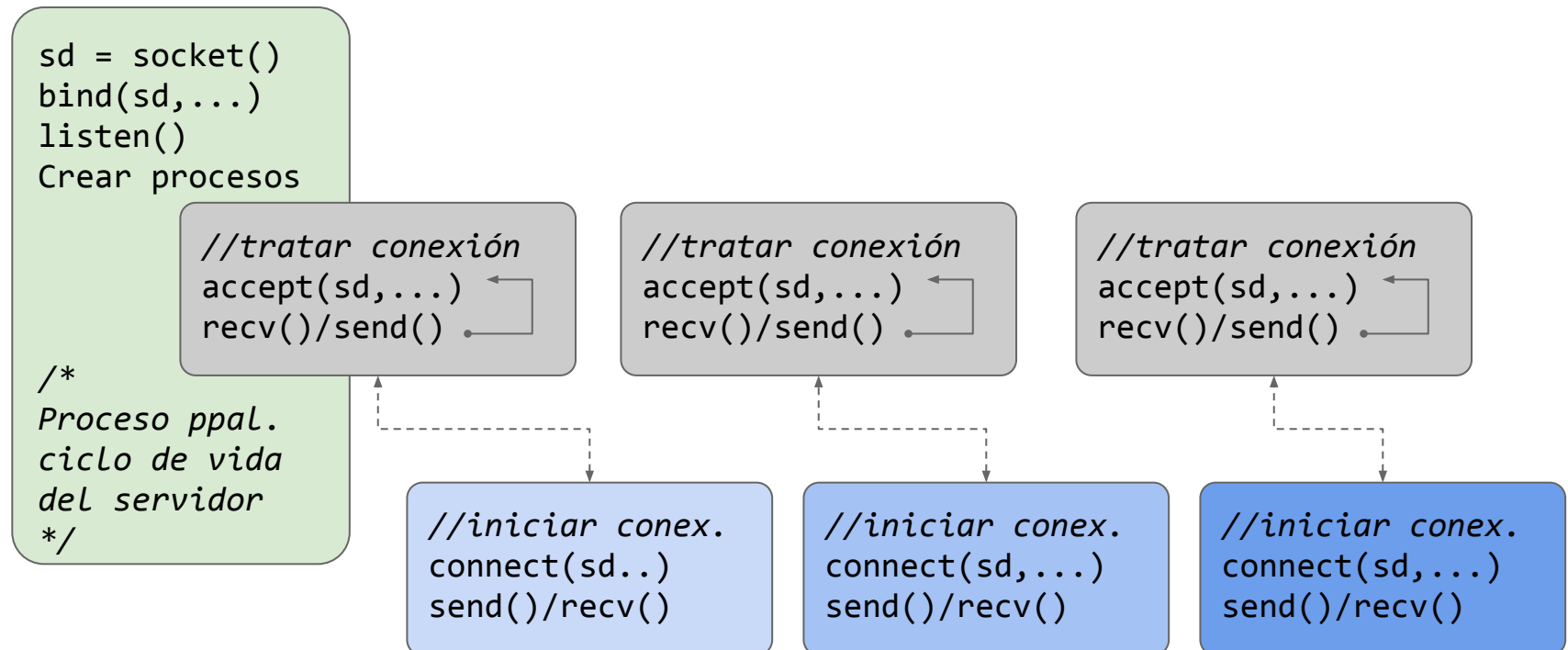
- Recepción concurrente de mensajes
- El servidor crea un conjunto de procesos/hilos para procesar los mensajes recibidos con `recvfrom(2)`
- Concurrencia a nivel de mensajes



# Servidores Concurrentes: SOCK\_STREAM

## Patrón *pre-fork*

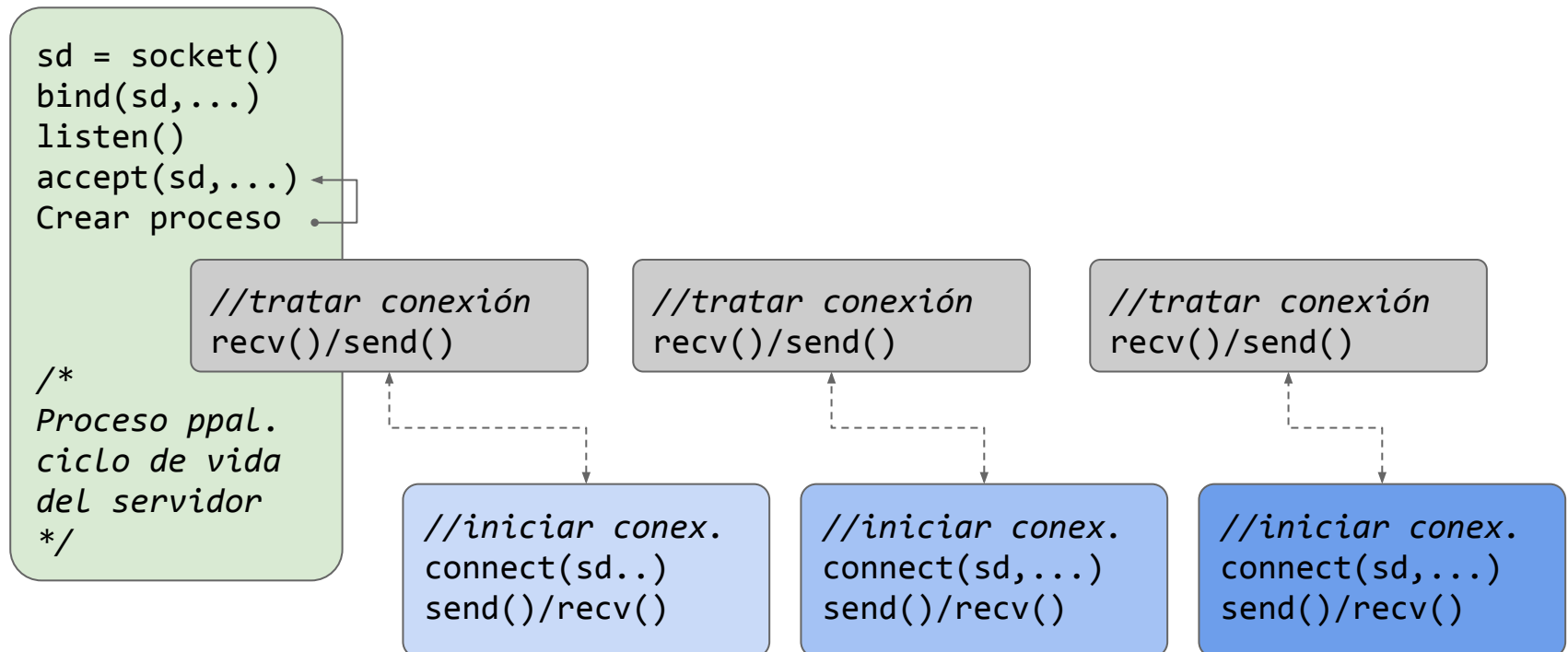
- Gestión concurrente de conexiones
- El servidor crea un conjunto de procesos/hilos que aceptan conexiones con `accept(2)` y las gestionan
- Concurrencia a nivel de conexiones



# Servidores Concurrentes: SOCK\_STREAM

## Patrón *accept-and-fork*

- Gestión concurrente de conexiones
- El servidor acepta conexiones con `accept(2)` y crea un proceso/hilo para procesar cada una
- Concurrencia a nivel de conexiones



# Ejemplos de Preguntas Teóricas

¿Generan algún mensaje de red socket(2), bind(2), listen(2) y accept(2)?

- ☐ No.
- ☐ Si.
- ☐ Depende.

¿Qué ocurre si no se usa bind(2) en un socket cliente?

- ☐ Que la siguiente llamada dará error.
- ☐ El resultado es indefinido, porque es obligatorio usarla.
- ☐ Que se asocia a la dirección local INADDR\_ANY y a un puerto libre aleatorio.