TEMA 2.4. Programación con Sockets

PROFESORES:

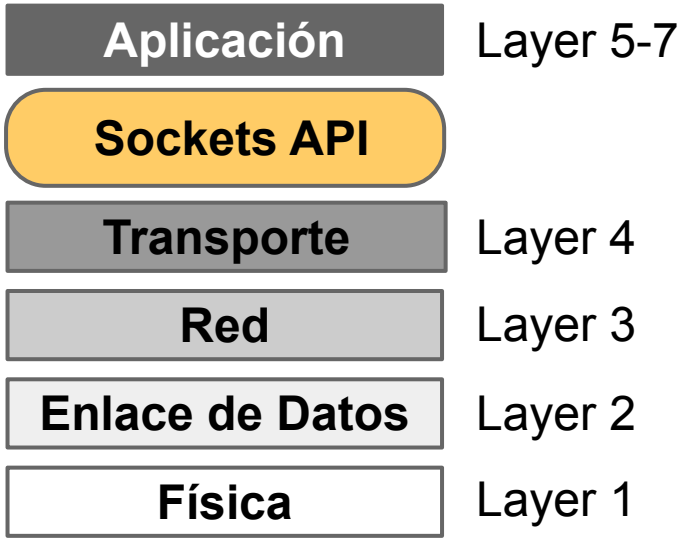
Rubén Santiago Montero
Eduardo Huedo Cuesta
Rafael Rodríguez Sánchez

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 (normalmente) 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
    - 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 procesado 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
- 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
- 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`
- 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

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 (`in_port_t`)**
    - Privilegiados (*well-known*) < 1024, sólo para procesos con privilegio
    - Asociados a los protocolos superiores TCP y UDP
  - **Dirección (`struct in_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 (**`struct`** `in6_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/socket.h>
<netdb.h>
<sys/types.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)

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)

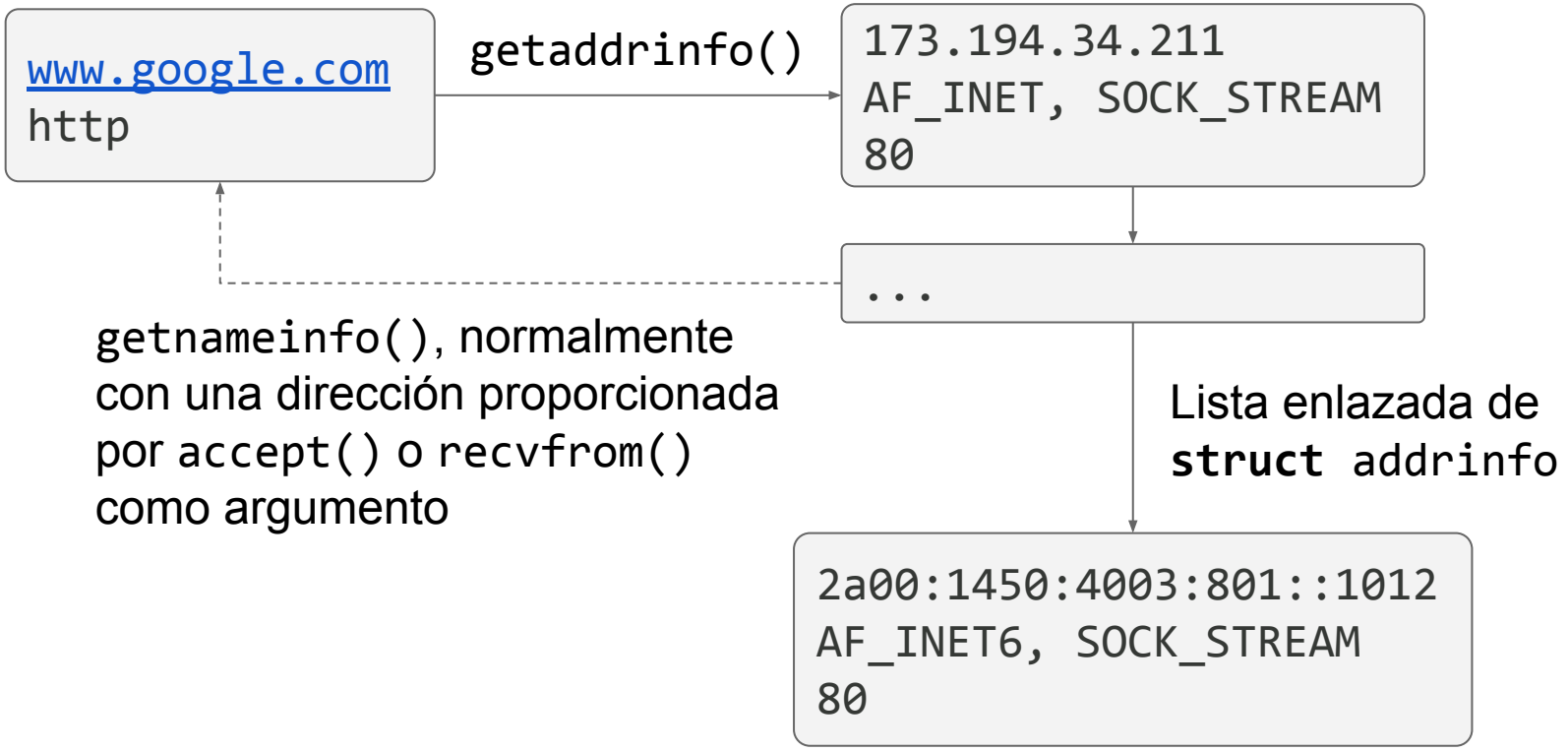
Gestión de Direcciones

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



Conversión de Direcciones y Valores

POSIX
<arpa/inet.h>

- 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

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` puede ser:
    - `IPPROTO_TCP` para `SOCK_STREAM` ⇒ Usar siempre 0
    - `IPPROTO_UDP` para `SOCK_DGRAM` ⇒ Usar siempre 0
  - 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);
```

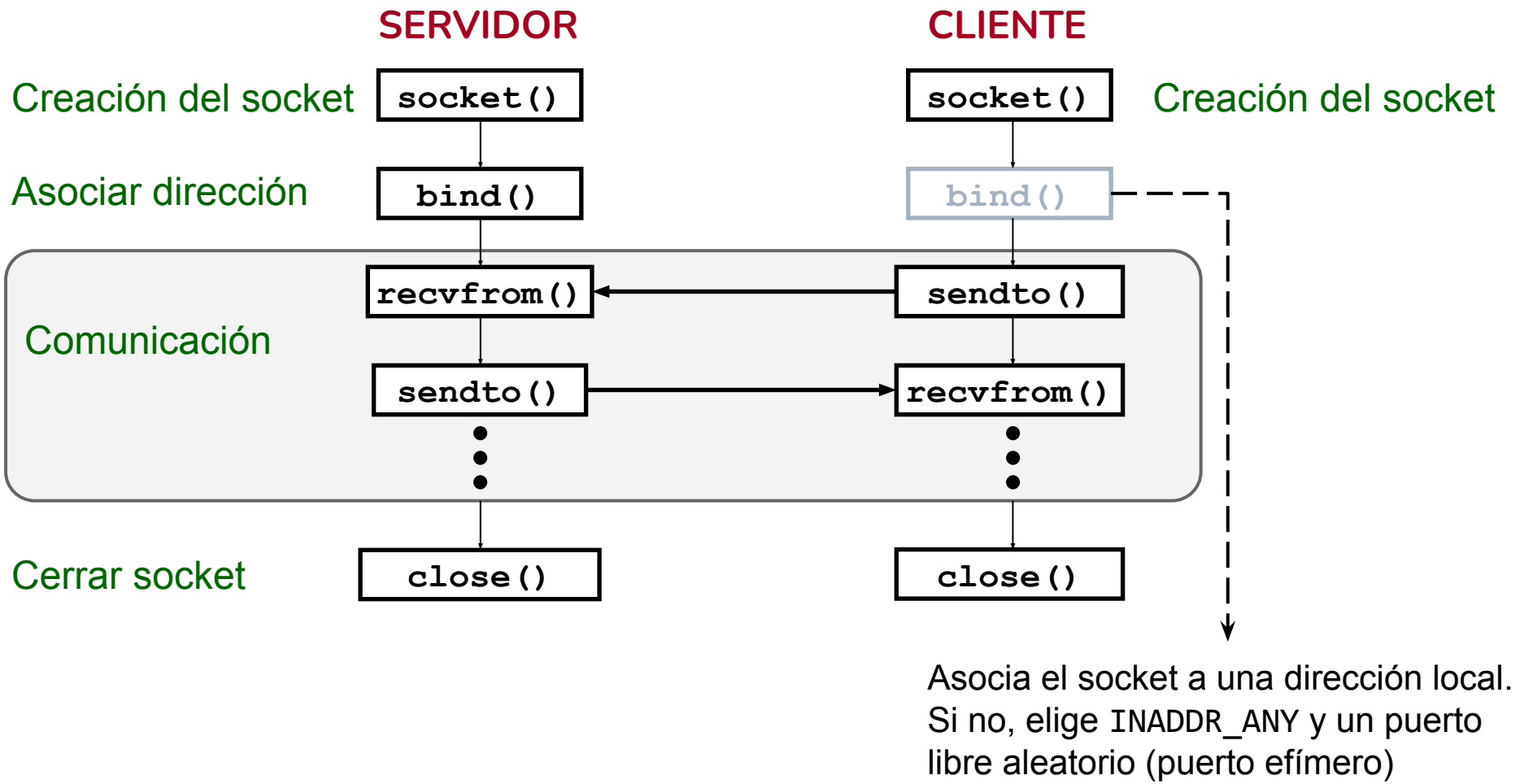
- Creación de sockets IPv6:

```
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

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

POSIX

<sys/socket.h>

- 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`**`()`)

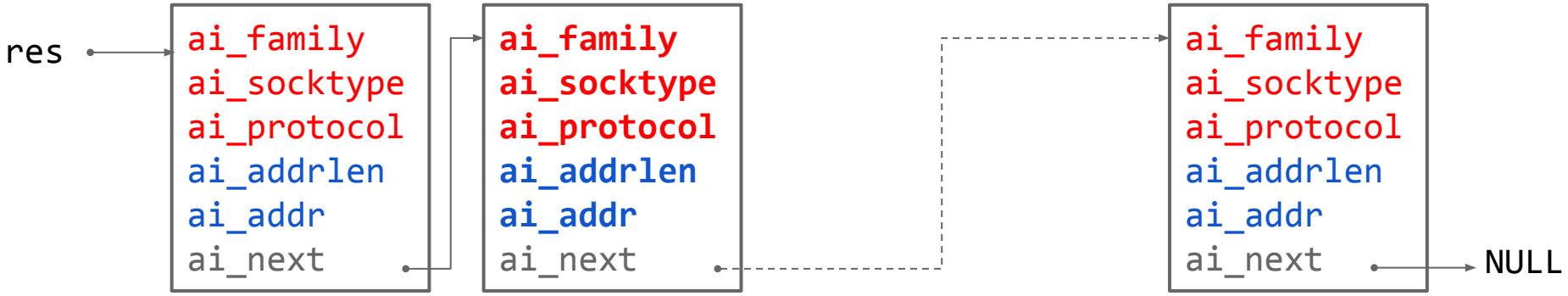
- Asignar una dirección remota a un socket:

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

Envío y Recepción de Datos

POSIX
<sys/socket.h>

- Enviar y recibir datos:

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

  - Con `SOCK_DGRAM` se usan para poder especificar u obtener la dirección del otro extremo (con `SOCK_STREAM` se ignora la dirección)
  - `recvfrom(2)` se bloquea si no hay mensajes disponibles:
    - Se puede usar `select()` 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);
```

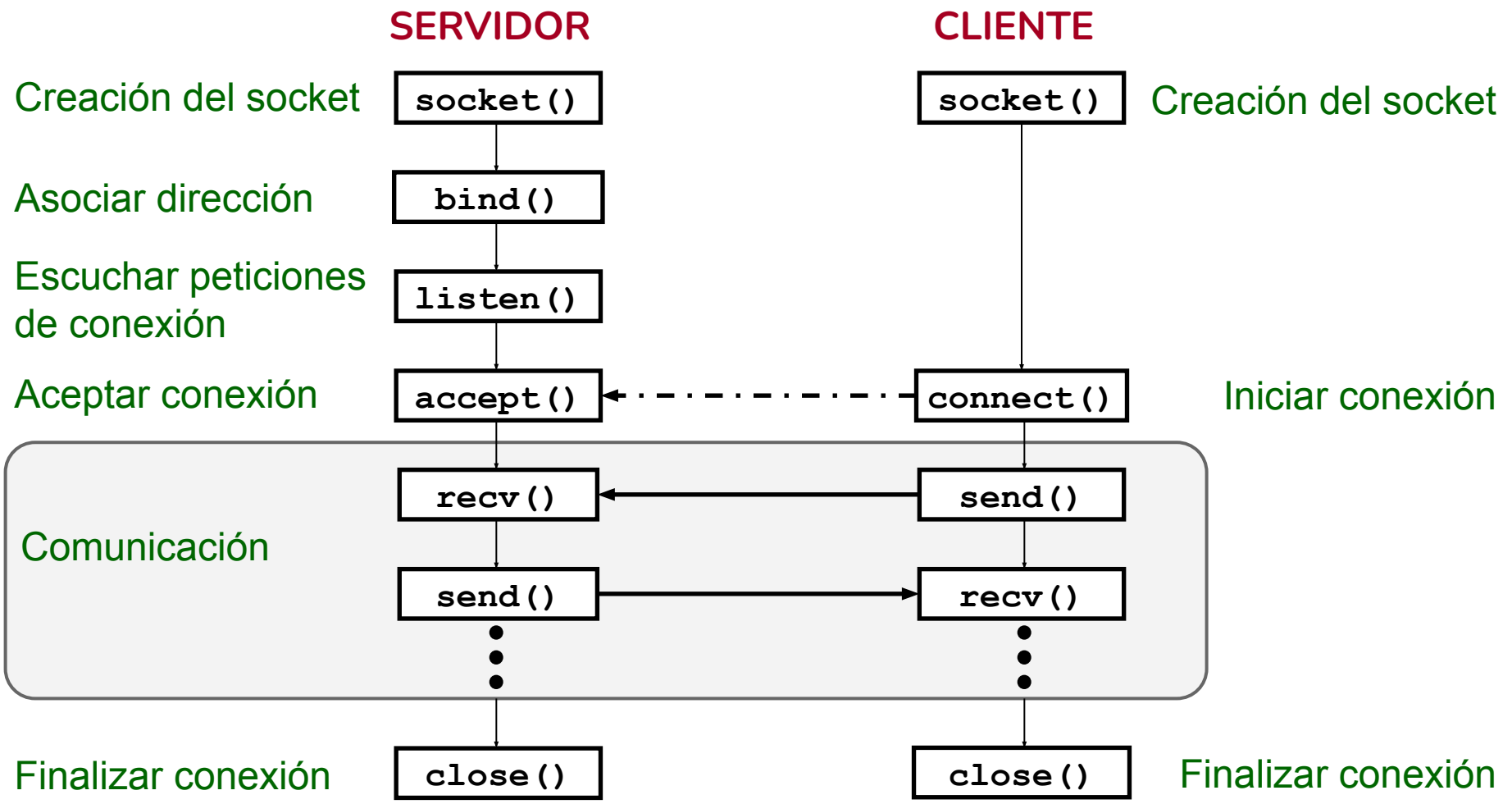
  - `addr` se usa para devolver una dirección IPv4 o IPv6
  - `addrlen` es un argumento valor-resultado, que inicialmente contiene el tamaño del búfer al que apunta `addr`, y se modifica para indicar el tamaño real de la dirección devuelta

Resumen: Esquema Servidor UDP

```
hints.ai_flags    = 0;
hints.ai_family   = AF_UNSPEC;   // IPv4 o 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("Recibidos %d bytes de %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

POSIX
<sys/socket.h>

- 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 máximo de la cola de conexiones completamente establecidas esperando ser aceptadas
    - No confundir con la cola SYN de 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()` o el modo no bloqueante
  - `addr` devuelve la dirección del cliente que se ha conectado
  - addrlen indica el tamaño del búfer y devuelve el tamaño real de la dirección
  - Devuelve un descriptor de socket para gestionar la conexión

Envío y Recepción de Datos

POSIX
<sys/socket.h>

- Enviar y recibir datos:

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

  - Se usan normalmente con sockets `SOCK_STREAM`
  - `send(2)` envía `len` bytes de `buffer`
    - 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 `buffer`
    - Con `SOCK_DGRAM`, el mensaje se debe leer en una sola operación (tamaño del buffer) para no perder datos
  - Ambas pueden bloquearse
    - `send(2)` se bloquea si el mensaje no cabe en el *buffer* de envío
    - `recv(2)` se bloquea si no hay mensajes disponibles en el socket
    - Se puede usar `select()` o el modo no bloqueante (flag `MSG_DONTWAIT`)

Resumen: Esquema Servidor TCP

```
hints.ai_flags    = 0;
hints.ai_family   = AF_UNSPEC;    // IPv4 o 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("Conexión desde %s:%s\n", host, serv);

   while (c = recv(clisd, buf, 80, 0)) { // Comprobar mensaje!
      send(clisd, buf, c, 0);
   }

   close(clisd);
}
```

Opciones de Sockets

POSIX
<sys/socket.h>

- 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

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()`) 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

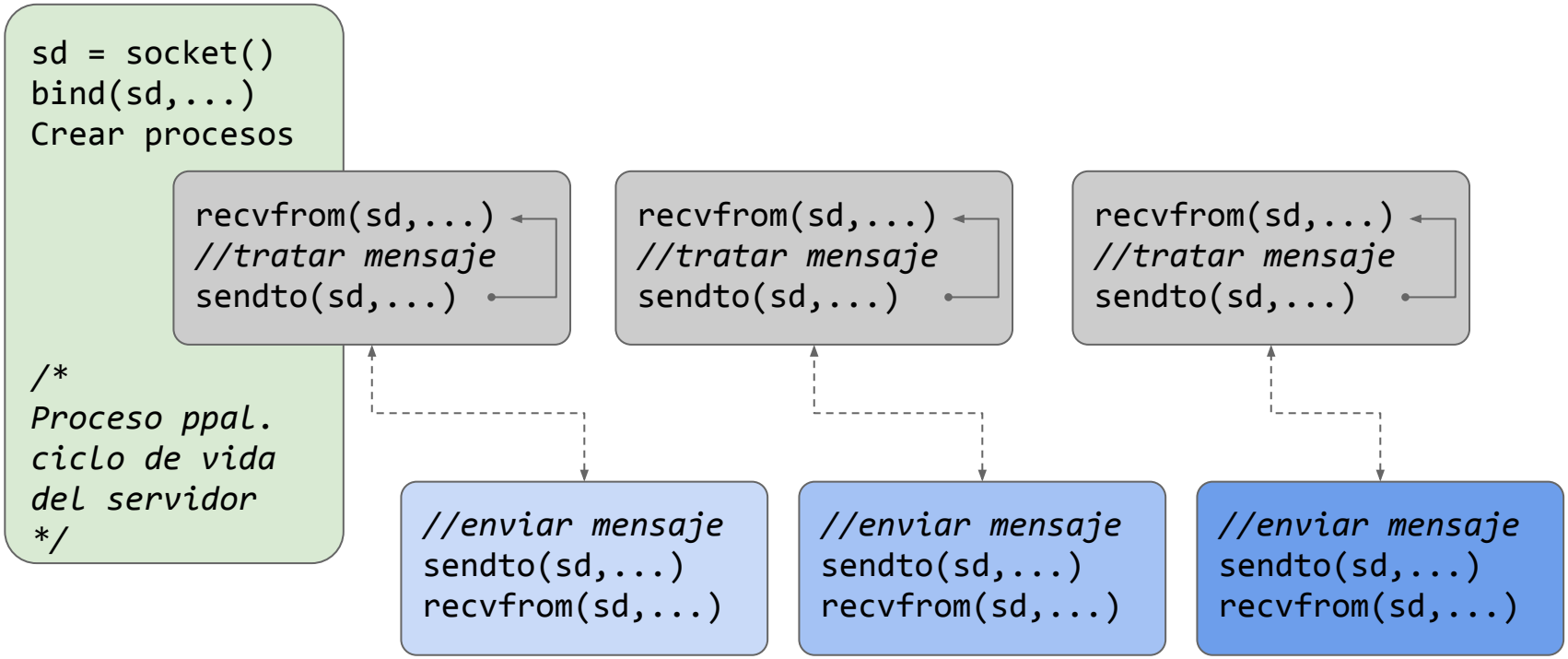
Herramientas

- Los threads 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 threads pueden llamar a `accept()` para establecer una conexión
  - Múltiples threads pueden llamar a `recvfrom()` para recibir datos
  - Todos los threads se bloquearán en la llamada y solo uno de ellos se desbloqueará cuando llegue una solicitud de conexión o datos
- También se puede usar multiplexación de E/S síncrona (`select()`), pero la lógica del programa es mucho más compleja

Servidores Concurrentes: SOCK_DGRAM

Patrón

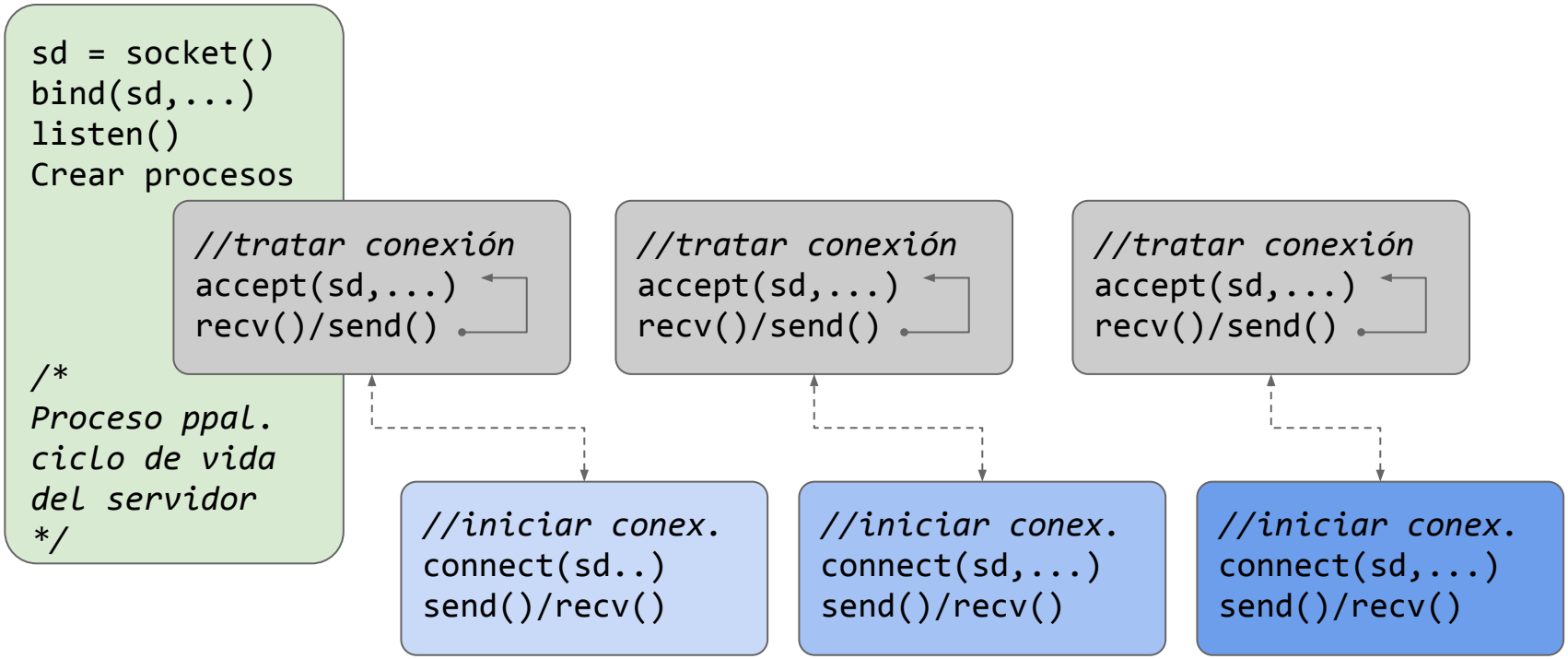
- Recepción concurrente de mensajes
- El servidor crea un conjunto de procesos/threads para procesar los mensajes recibidos con `recvfrom()`
- La concurrencia es en el nivel del mensaje



Servidores Concurrentes: SOCK_STREAM

Patrón *pre-fork*

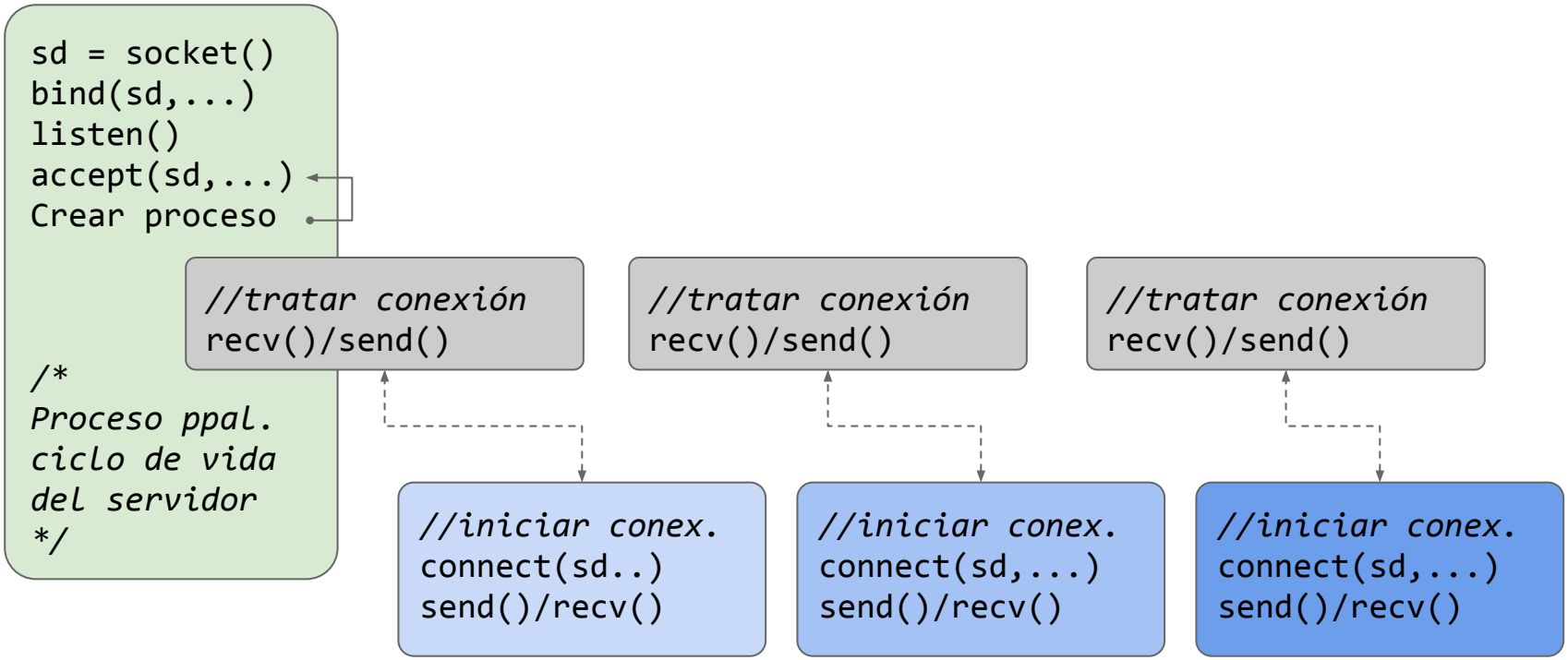
- Gestión concurrente de conexiones
- El servidor crea un conjunto de procesos/threads que aceptan conexiones con `accept()` y las gestionan
- La concurrencia es en el nivel de conexión



Servidores Concurrentes: SOCK_STREAM

Patrón *accept-and-fork*

- Gestión concurrente de conexiones
- El servidor acepta conexiones con `accept()` y crea un proceso/thread para procesar cada una
- La concurrencia es en el nivel de conexión



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 la llamada 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.