TEMA 2.4. Programación con Sockets

PROFESORES:

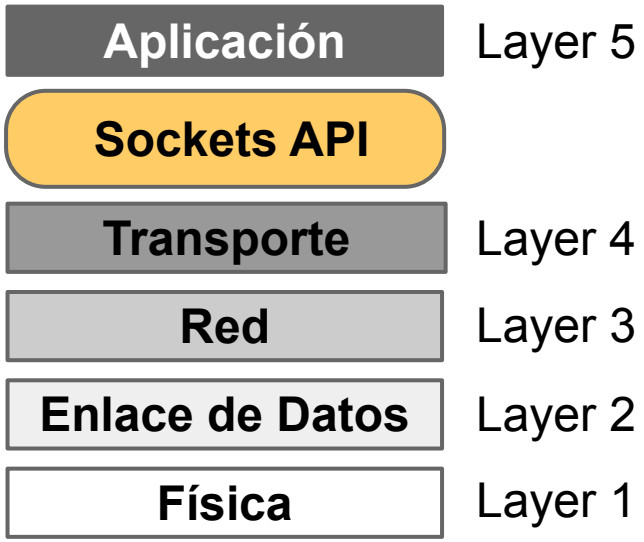
Rubén Santiago Montero
Eduardo Huedo Cuesta
Rafael Rodríguez Sánchez

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
    - Debe marcarse inicio y fin del mensaje (ej. `<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 y protocolo que se ajuste a la semántica específica del tipo
  - No todos los tipos de sockets están soportados por todos los dominios y protocolos
- **Dominio**: Familia de protocolos 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`
- **Protocolo:** Protocolo particular de la familia que se usará por el 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
};
```

  - **Puertos (`in_port_t`)**
    - Privilegiados (*well-known*) < 1024, sólo para procesos con privilegio
    - Asociados a los protocolos superiores TCP y UDP
  - **Direcciones (`struct in_addr`)**
    - Dirección de red asociada al interfaz
    - Para usar direcciones *broadcast* se debe fijar la opción `SO_BROADCAST`
    - Se puede inicializar o asignar con las constantes `INADDR_ANY` (`0.0.0.0`), `INADDR_LOOPBACK` (`127.0.0.1`) e `INADDR_BROADCAST` (`255.255.255.255`)

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`, que se refiere al 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`, no se fijará el puerto
  - `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 y protocolo, como en `socket()`
  - `ai_flags`: Opciones, ej. `AI_PASSIVE` con `node==NULL` devuelve `0.0.0.0` ó `::` (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)

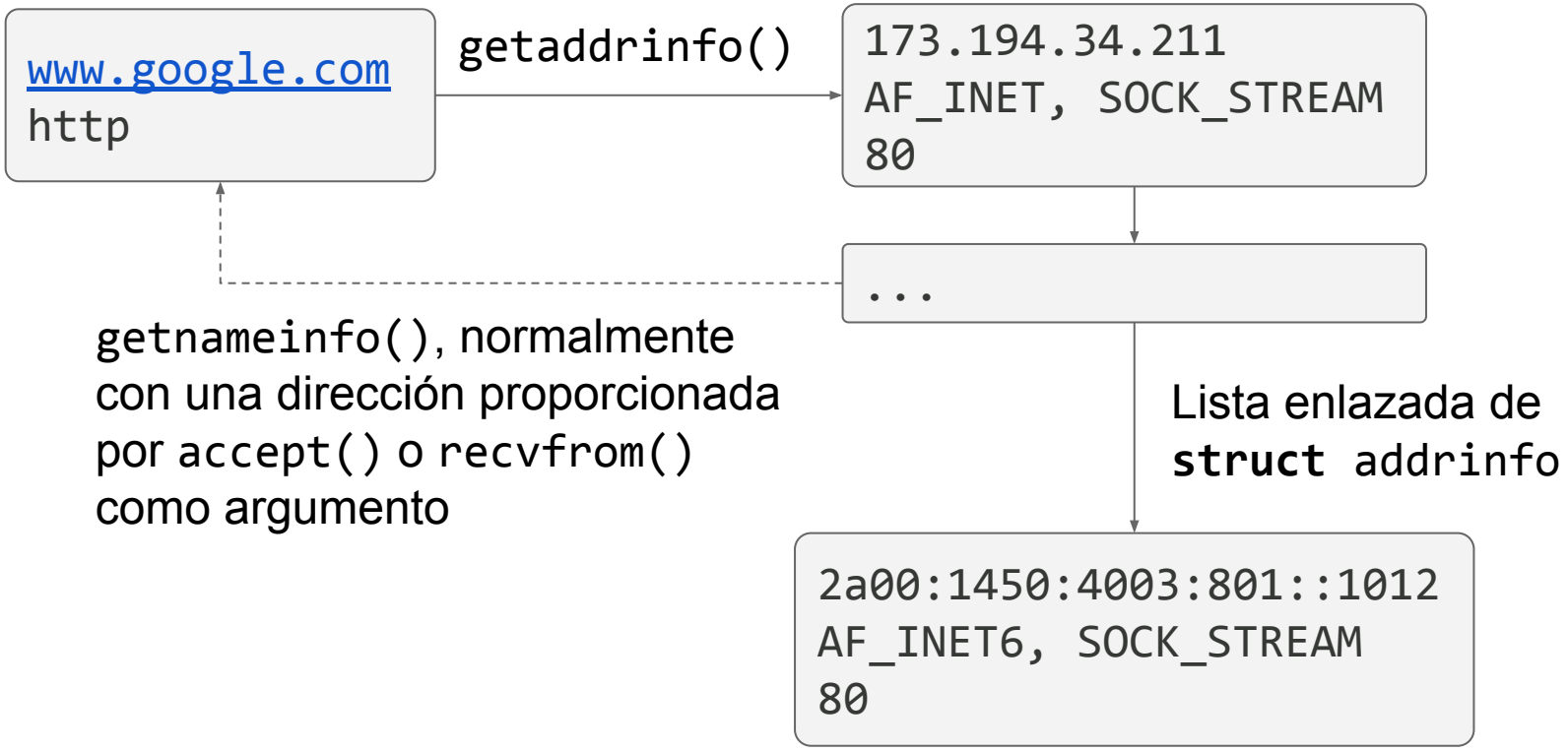
Gestión de Direcciones

- 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)
```

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

POSIX



Gestión de Direcciones

- Funciones útiles para manejar direcciones de un protocolo de red específico:

POSIX
<arpa/inet.h>

```
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
  - Los argumentos de tipo **`void *`** contienen la estructura de la dirección en binario:
    - `AF_INET`: **`struct`** `in_addr`
    - `AF_INET6`: **`struct`** `in6_addr`
  - Los argumentos de tipo **`char *`** contienen la representación de la dirección como texto
    - `AF_INET`: en representación decimal de punto
    - `AF_INET6`: en representación hexadecimal abreviada

Gestión 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 contiene una dirección IPv4 o IPv6
   addrlen contiene la longitud de la estructura devuelta */
```

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);
```

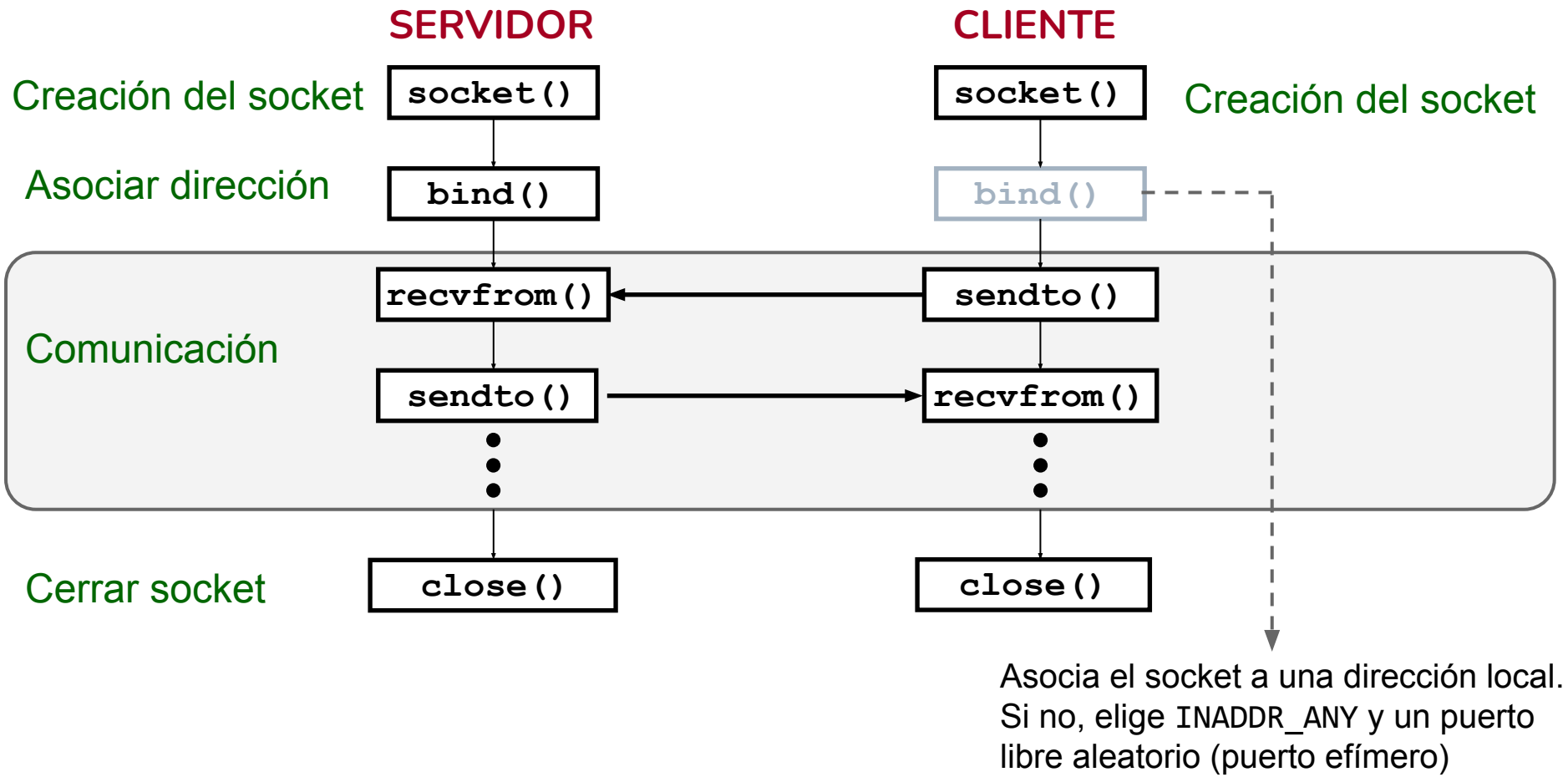
- Creación de sockets IPv6:

```
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`

Gestión de la Conexión

POSIX
<sys/socket.h>

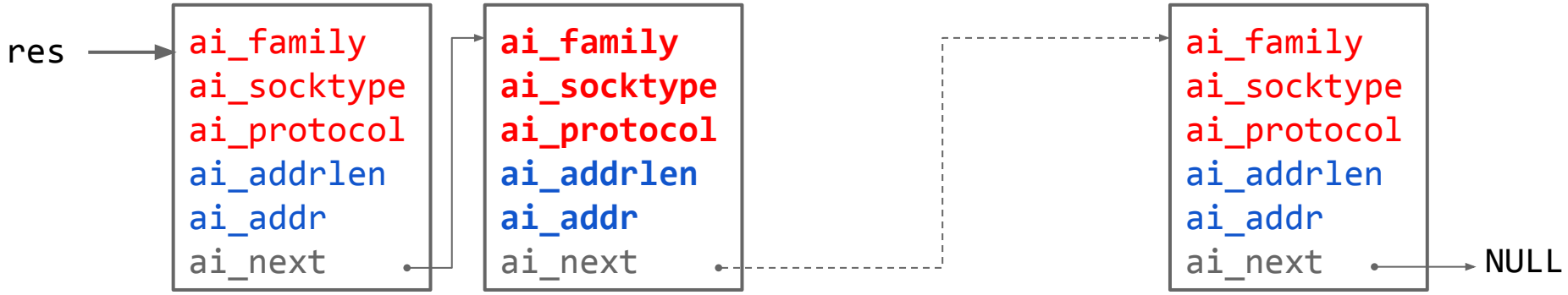
- Asociación de direcciones locales y remotas:

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

  - `bind()` establece una dirección local para el socket
  - `connect()` establece la dirección remota
    - Con `SOCK_STREAM`, inicia una conexión
  - `sd` es el descriptor creado con `socket()`
  - `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()`)

Gestión de la Conexión

```
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>

```
ssize_t sendto(int sd, const void *message, size_t length,
       int flags, const struct sockaddr *dst_addr,
       socklen_t addrlen);

ssize_t recvfrom(int sd, void *buffer, size_t length, int flags,
       struct sockaddr *src_addr, socklen_t *addrlen);
```

- Se suelen usar con SOCK_DGRAM, ya que permiten especificar u obtener la dirección del otro extremo
- Los datos se envían en el orden de red, *big-endian*, por lo que puede ser necesario convertirlos al orden de la arquitectura del procesador:

POSIX

<arpa/inet.h>

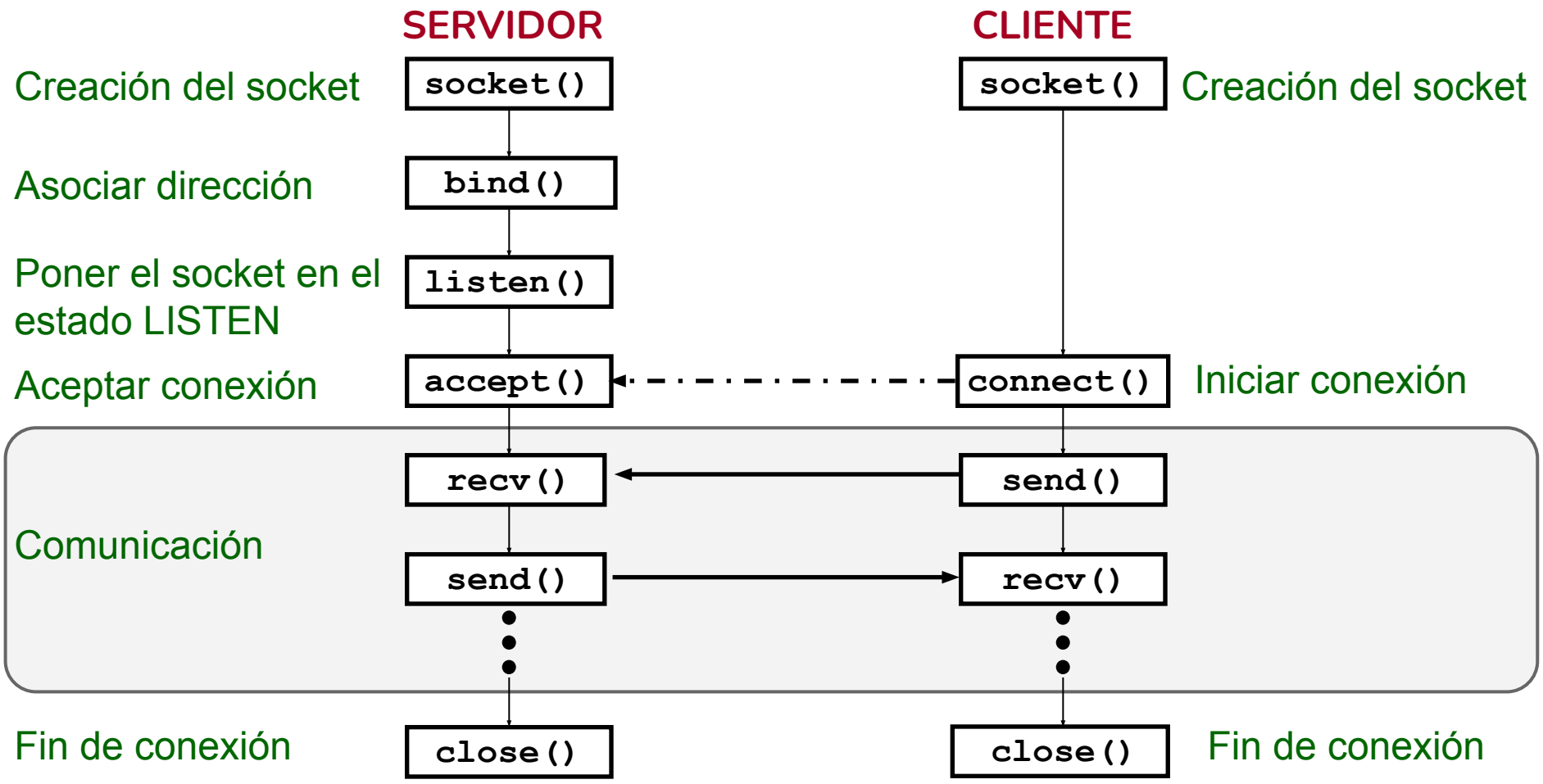
```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Resumen: Esquema Servidor UDP

```
hints.ai_flags    = AI_PASSIVE; // Devolver 0.0.0.0 o ::
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) {
  c = recvfrom(sd, buf, 80, 0, (struct sockaddr *) &addr, &addrlen);
  buf[c] = '\0';

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

  printf("Mensaje de %s:%s: %s\n", host, serv, buf);

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

Sockets TCP: Patrón de Comunicación

- Sockets tipo `SOCK_STREAM` para la familia `AF_INET` y `AF_INET6`
- El tratamiento de la conexión se puede realizar en un proceso o hilo diferente



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

Gestión de la Conexión

POSIX
<sys/socket.h>

- Gestión de conexiones:

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

  - Se usan con sockets `SOCK_STREAM`
  - `listen()` pone el socket en modo de escucha para aceptar conexiones
    - `backlog` es un número máximo de conexiones pendientes en la cola
  - `accept()` acepta una conexión establecida y devuelve un descriptor de socket para gestionarla
    - `addr` devuelve la dirección del cliente que se ha conectado
    - Se bloquea si no hay conexiones pendientes → se puede usar `select()` o el modo no bloqueante

Envío y Recepción de Datos

POSIX

<sys/socket.h>

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

- Se usan normalmente con sockets `SOCK_STREAM`
- `send()` envía `length` bytes de `buffer`
  - Con `SOCK_DGRAM` hay que usar `connect()` previamente
  - Si el mensaje es demasiado grande, no se envían datos (`EMSGSIZE`)
- `recv()` recibe hasta `length` bytes en `buffer`
  - Para `SOCK_DGRAM`, el mensaje se debe leer en una sola operación (tamaño del buffer) para no perder datos
- Ambas pueden bloquearse:
  - Se puede usar `select()` o el modo no bloqueante (flag `MSG_DONTWAIT`)

Resumen: Esquema Servidor TCP

```
hints.ai_flags    = AI_PASSIVE; // Devolver 0.0.0.0 o ::
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) {
  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 Host:%s Puerto:%s\n", host, serv);

  while (c = recv(clisd, buf, 80, 0)) { // Comprobar mensaje!
    buf[c] = '\0';
    printf("\tMensaje: %s\n", buf);
    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`
  - Ejemplos a nivel de sockets: `SO_KEEPALIVE`, `SO_BROADCAST`, **`SO_REUSEADDR`**
  - Ejemplos a nivel de protocolo IP:
    - Grupos multicast (IP): `IP_ADD_MEMBERSHIP`, `IP_DROP_MEMBERSHIP`
    - MTU: `IP_MTU`, `IP_MTU_DISCOVER`
    - Campos del datagrama IP: `IP_OPTIONS`, `IP_TTL`, `IP_TOS`

Nota: Muchas opciones se pueden configurar vía `/proc` (ej. `ip_default_ttl`…)

Opciones para Sockets TCP

- Opciones `SO_SNDBUF` y `SO_RCVBUF` (**nivel de API de sockets**)
  - Tamaño de los buffers de envío y recepción
  - Mayores tamaños permiten gestionar la ventana de TCP más eficientemente (timestamps, escalado de la ventana para control de flujo…)
  - Accesibles vía `sysctl` o `/proc` (`net.ipv4.tcp_rmem` y `net.ipv4.tcp_wmem`)
- Otras opciones (**nivel de protocolo TCP**)
  - `TCP_NODELAY` desactiva el algoritmo de Nagle
  - `TCP_QUICKACK` desactiva los ACKs retrasados
  - Otras opciones TCP son accesibles vía `sysctl` o `/proc`

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()` espera a que se establezcan conexiones de clientes
  - `recv()` y `recvfrom()` esperan a que lleguen de datos
  - `send()` y `sendto()` esperan si la ventana de envío está llena

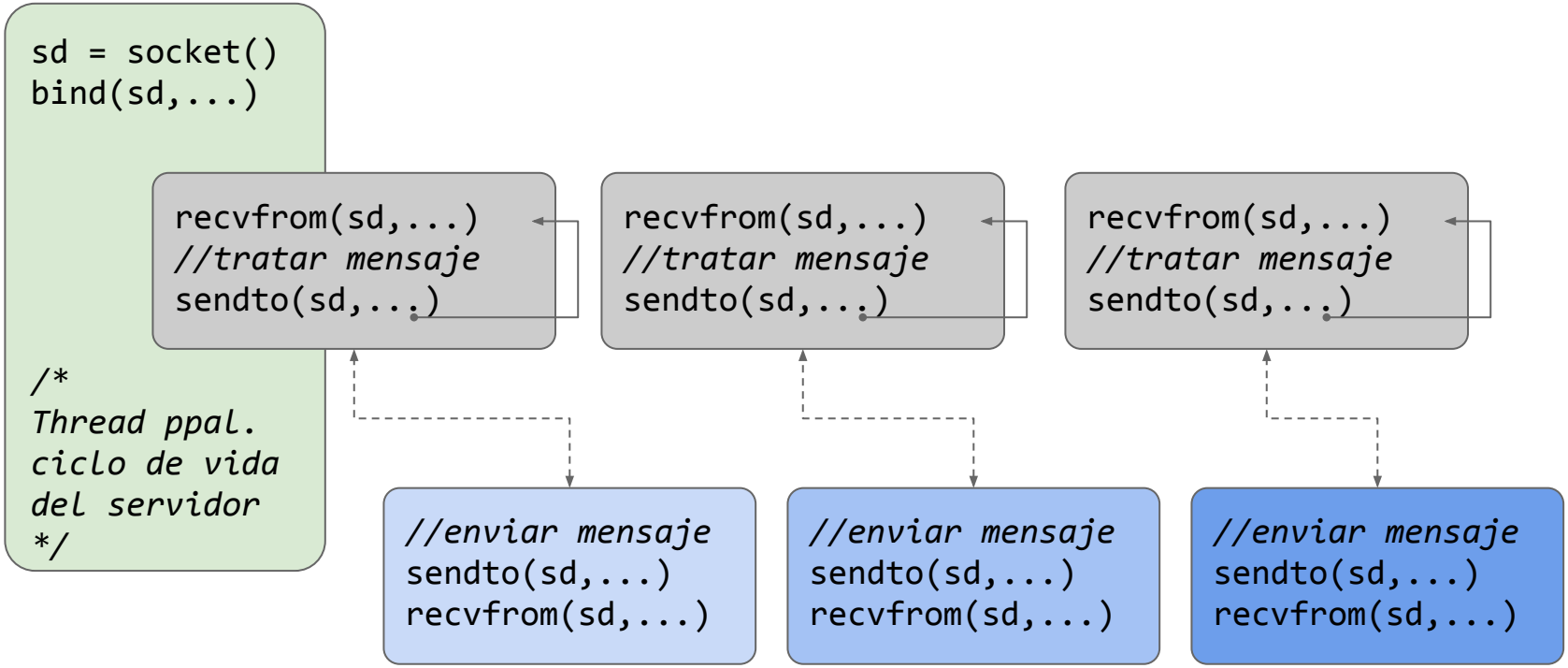
Herramientas

- 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

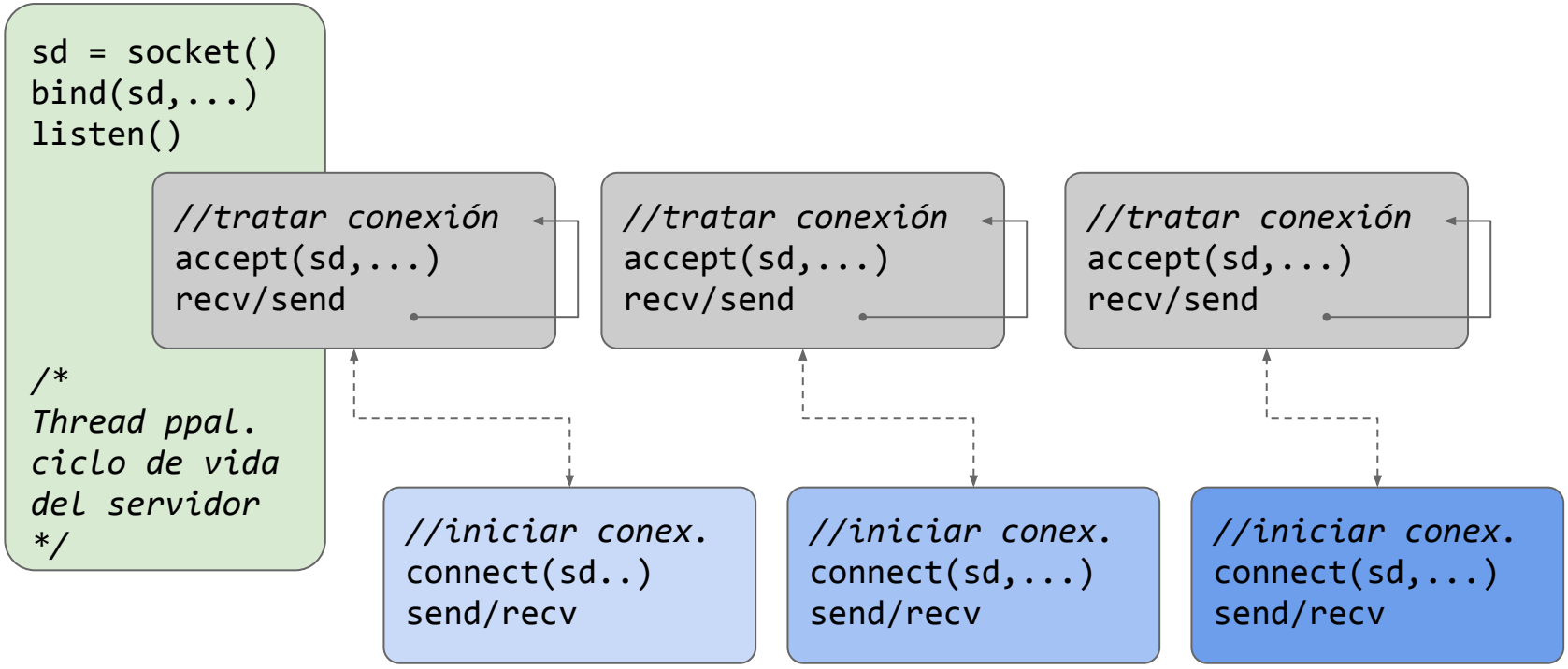
- Recepción concurrente de mensajes
- El servidor crea un conjunto de procesos/threads para procesar los mensajes recibidos con `recvfrom()`
- La concurrencia es en el nivel del mensaje



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