ADVANCED OPERATING SYSTEMS AND NETWORKS

Computer Science Engineering

Universidad Complutense de Madrid

2.4. Socket Programming

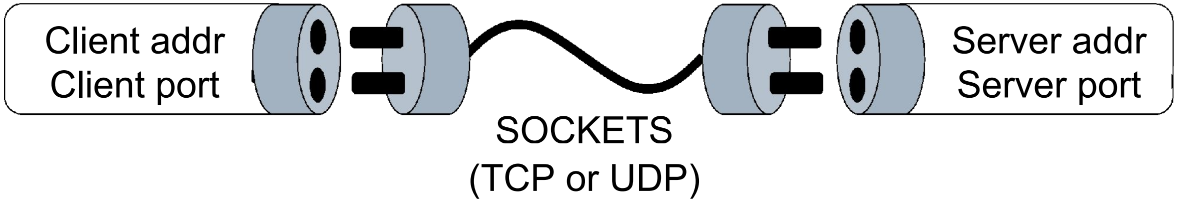
PROFESSORS:
Eduardo Huedo Cuesta
Rubén Santiago Montero

OTHER AUTHORS:
Rafael Moreno Vozmediano
Juan Carlos Fabero Jiménez

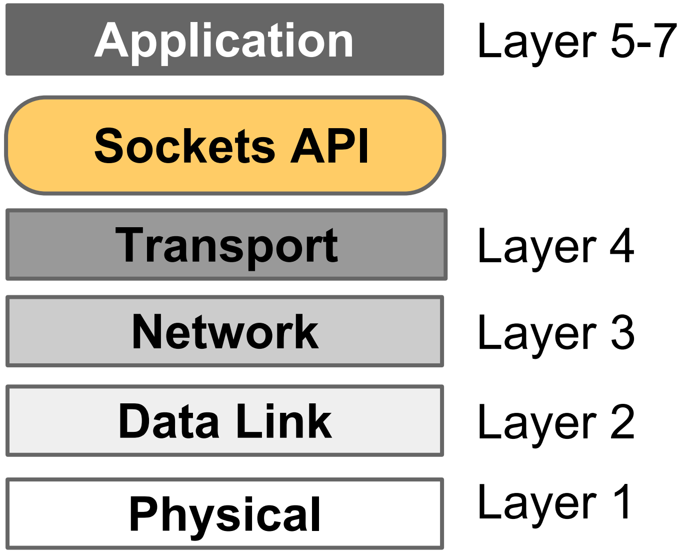
Introduction

Sockets

- A **socket** is an endpoint for communication
- Sockets allow the bidirectional data exchange between a client and a server
- Each server or client application is (normally) identified by a **port** number



- **Socket APIs:** Uniform interface between the user process and the network protocol stacks in the kernel
  - **BSD Sockets API** or Berkeley Sockets
  - WinSock API equivalent to BSD
  - Bindings available for all languages



Sockets Types

- **`SOCK_STREAM`**
  - Sequenced, reliable, two-way, connection-based byte streams
  - An out-of-band data transmission mechanism may be supported
  - Similarly to a pipe, a connection must be established before any data may be sent or received and the `SIGPIPE` signal is raised if a process sends or receives on a broken stream
  - Message boundaries in incoming datagrams are not preserved, so the beginning and end of the message must be marked (e.g. `<HTML></HTML>`, `{“msg”: {...}}`)
- **`SOCK_DGRAM`**
  - Datagrams (connectionless, unreliable messages of a fixed maximum length)
- **`SOCK_RAW`**
  - Raw network protocol access (e.g. to implement protocol modules in user space)

Socket Creation

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

POSIX+BSD

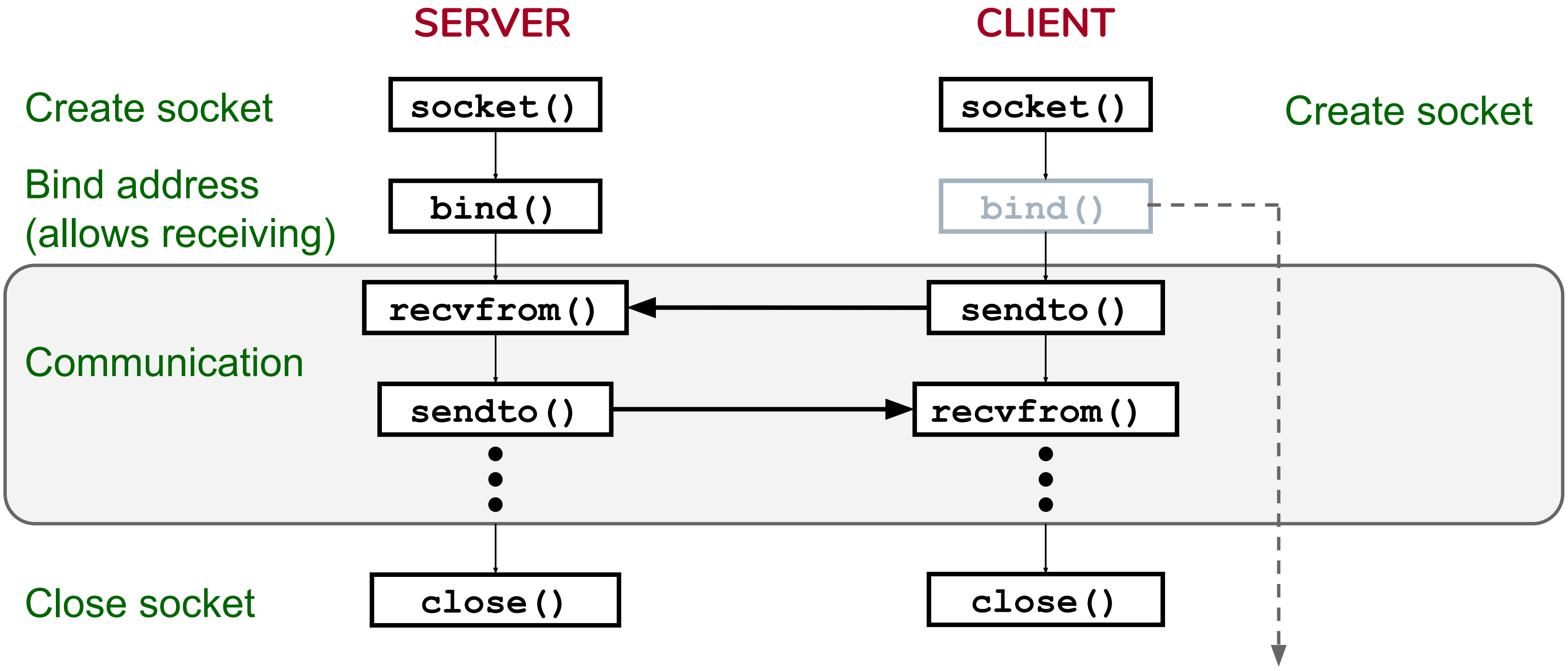
- Create an endpoint for communication:

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

    - `domain` is a protocol family, sharing some address scheme
        - `AF_UNIX`: Local communication between processes in the same system
        - **`AF_INET`**, **`AF_INET6`**: Internet protocols, over IPv4 or IPv6
        - Others: `AF_IPX`, `AF_X25`, `AF_APPLETALK`, `AF_PACKET`...
    - `type` is a socket type, specifying the communication semantics
        - **`SOCK_STREAM`**, **`SOCK_DGRAM`** or `SOCK_RAW`
    - `protocol` is the particular protocol of the family to be used to implement the socket type
        - Normally, only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0
        - Some socket types may not be implemented by all protocol families

UDP Sockets: Communication Pattern

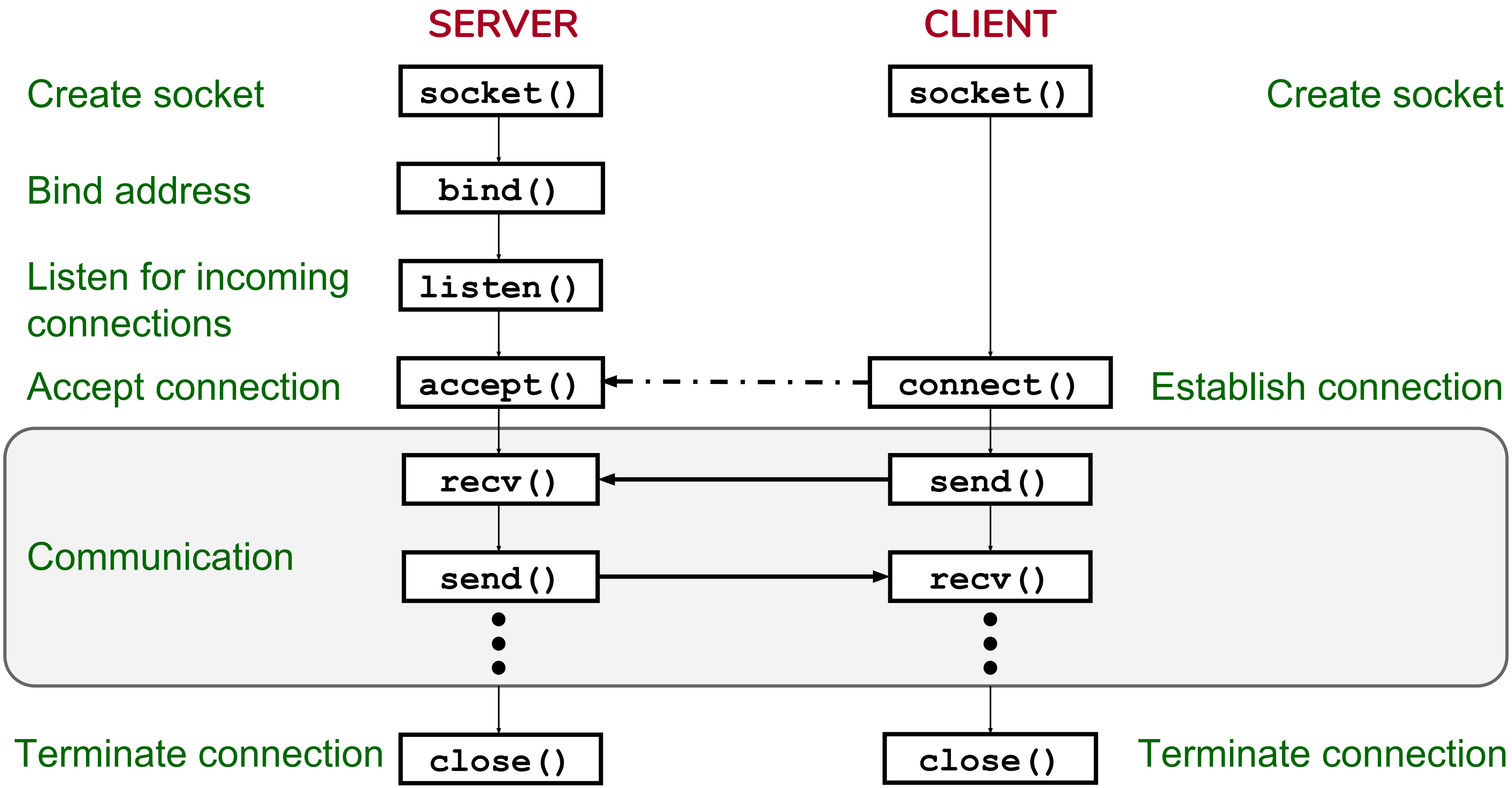
- Sockets of type `SOCK_DGRAM`



To bind the socket to a local address and port. Otherwise, it chooses `INADDR_ANY` and a random free port (ephemeral port)

TCP Sockets: Communication Pattern

- Sockets of type `SOCK_STREAM`



Sockets Creation: IPv4 Sockets

```
tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET, SOCK_RAW, protocol);
```

- The protocol can be:
  - `IPPROTO_TCP` for `SOCK_STREAM` ⇒ always use 0
  - `IPPROTO_UDP` for `SOCK_DGRAM` ⇒ always use 0
  - A valid IANA IP protocol number (RFC 1700) for `SOCK_RAW`

- **Socket address:** Defined by a 32-bit IP address (network-level address) and a 16-bit TCP/UDP port (transport-level address)

```
struct sockaddr_in {
   sa_family_t    sin_family; // address family: AF_INET
   in_port_t      sin_port;   // port number in network byte order
   struct in_addr sin_addr;   // IP address
};

struct in_addr {
   uint32_t      s_addr; // 32-bit IP address in network byte order
};
```

Sockets Creation: IPv4 Sockets

Ports (`in_port_t`)

- Privileged (usually, also well-known) port numbers, below 1024, can only be used by privileged processes
- Associated to transport layer protocols: TCP and UDP
- On RAW sockets it is set to the Protocol field in the IP header

Addresses (`struct in_addr`)

- Address associated to the interface
- To use broadcast addresses, option `SO_BROADCAST` must be specified in the socket
- Constants:
  - `INADDR_ANY` (`0.0.0.0`)
  - `INADDR_BROADCAST` (`255.255.255.255`)
    - For historical reasons, it has the same effect on `bind()` as `INADDR_ANY`
  - `INADDR_LOOPBACK` (`127.0.0.1`)

Management of Network Addresses and Names

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

POSIX

- Name-to-address translation in protocol-independent manner:

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

  - node specifies the host, and it could be:
    - A host name, which is resolved using `gethostbyname(3)`
    - An IPv4 address in dotted-decimal notation, e.g. “`192.168.0.1`”
    - An IPv6 address in hexadecimal abbreviated notation, e.g. “`fe80::1:2`”
    - `NULL`, which refers to the local host
  - `service` specifies the port, and it could be:
    - A service name from `/etc/services` (see `services(5)`), e.g. “`http`”
    - A decimal integer, e.g. “`80`”
    - `NULL`, to not set the port
  - `hints` sets some search criteria
  - `res` is used to return the list of socket addresses
    - The host have multiple network interfaces or supports several protocols (e.g. IPv4 and IPv6)
    - The service supports several protocols (e.g., telnet tcp/23 and udp/23)

Management of Network Addresses and Names

```
struct addrinfo {
    int              ai_flags;      // Filtering options (hints)
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;    // Result (res)
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

- Filtering options (`hints` argument, other fields must be set to 0 or `NULL`):
  - `ai_family`: `AF_INET` for IPv4, `AF_INET6` for IPv6, or `AF_UNSPEC` for both
  - `ai_socktype` and `ai_protocol`: Type and protocol as in `socket(2)`
  - `ai_flags`: Additional options, e.g. `AI_PASSIVE` to return `0.0.0.0` or `::` with `node == NULL` (`127.0.0.1` or `::1` otherwise)
- Result (`res` argument):
  - `ai_addr` and `ai_addrlen`: Pointer to a socket address and its length in bytes
  - `ai_canonname`: Official name of the host if `AI_CANONNAME` set in `ai_flags`
  - `ai_next`: Pointer to next result (linked list)

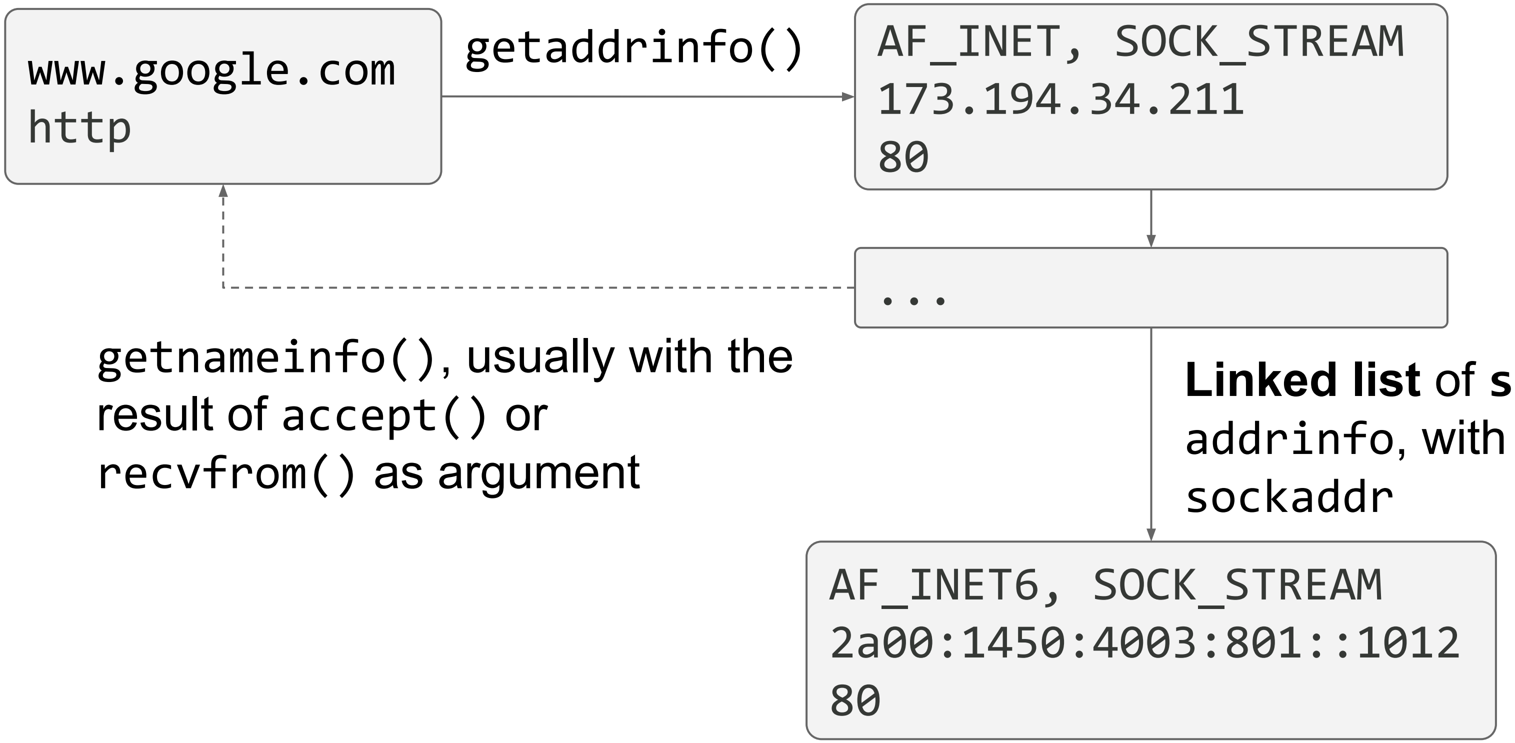
Management of Network Addresses and Names

- Address-to-name translation in protocol-independent manner:

```
int getnameinfo(const struct sockaddr *sa,
       socklen_t salen, char *host,
       socklen_t hostlen, char *serv,
       socklen_t servlen, int flags);
```

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

POSIX



getnameinfo(), usually with the result of accept() or recvfrom() as argument

Linked list of **struct** addrinfo, with **struct** sockaddr

Management of Network Addresses and Names

<arpa/inet.h>

POSIX

- Convert network addresses from binary to text form and from text to binary form:

```
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` is the address family
    - `AF_INET`: dotted decimal representation (“`d.d.d.d`”)
    - `AF_INET6`: abbreviated hexadecimal representation
  - Arguments of type `void *` contain the address structure in binary form
    - `AF_INET`: `struct in_addr`
    - `AF_INET6`: `struct in6_addr`
  - Arguments of type `char *` contain the address representation in text form

Connection Management

<sys/socket.h>
POSIX

- Manage connections:

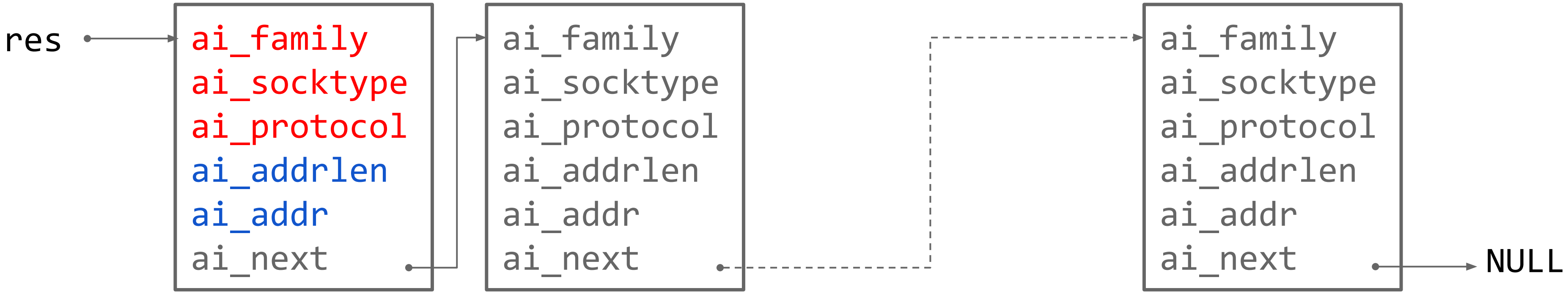
```
int bind(int sd, const struct sockaddr *addr,
        socklen_t len);
int connect(int sd, const struct sockaddr *addr, socklen_t len);
int listen(int sd, int backlog);
int accept(int sd, struct sockaddr *addr, socklen_t *len);
```

  - `sd` is a socket descriptor created with a call to `socket()`
  - `addr` is an address of generic type to accommodate each family address
    - In `bind()`, it is the local socket address to be assigned to the socket
    - In `connect()`, it is the address of the socket to connect to
    - In `accept()`, it stores the address of the peer socket after the call returns
  - `len` is the address length, which depends on the address type (use `sizeof()`)
  - `backlog` is the maximum queue length of connections waiting to be accepted

- `listen()` and `accept()` are only used for `SOCK_STREAM`
  - `listen()` marks the socket as a passive socket, that is, as a socket that will be used to accept incoming connection requests using `accept()`
  - `accept()` returns a new socket for the established connection
    - If no pending connections are present on the queue and the socket is not marked as nonblocking, it blocks (`select()` can be used)

Connection Management

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



res

ai_family
ai_socktype
ai_protocol
ai_addrlen
ai_addr
ai_next

ai_family
ai_socktype
ai_protocol
ai_addrlen
ai_addr
ai_next

ai_family
ai_socktype
ai_protocol
ai_addrlen
ai_addr
ai_next

NULL

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

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

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

Sending and Receiving Data

<sys/socket.h>

POSIX

- Send and receive data:

```
ssize_t send(int sock, const void *buf,
        size_t length, int flags);
ssize_t recv(int sock, void *buf, size_t length, int flags);
```

  - Typically used for `SOCK_STREAM`
  - `send()` will send `length` bytes from `buf`
    - For `SOCK_DGRAM`, `connect()` must be used first to set the peer address
  - `recv()` will receive up to `length` bytes in `buf`
    - For `SOCK_DGRAM`, the message should be read in a single `recv()` operation (buffer size) in order to not losing data
  - Both calls can block:
    - Use `select()` or `fcntl(sd, F_SETFL, O_NONBLOCK)`
  - Errors:
    - `EMSGSIZE`: Message is too big (the message is not transmitted)

Sending and Receiving Data

<sys/socket.h>

POSIX

- Send and receive data:

```
ssize_t sendto(int socket, const void *buf,
       size_t length, int flags,
       const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int socket, void *buf,
       size_t length, int flags,
       struct sockaddr *src_addr, socklen_t *addrlen);
```

  - Typically used for `SOCK_DGRAM`, to specify or obtain the peer address
- Data is sent in network order, i.e. big-endian, so it may be necessary to convert it to the processor architecture
- Convert values between host and network byte order:

<arpa/inet.h>

POSIX

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

IP Sockets: Options

<sys/socket.h>
POSIX

- Get and set options on sockets:

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

  - `level` specifies where the option is applied:
    - Sockets API: `SOL_SOCKET`
    - Protocol: `IPPROTO_IP`, `IPPROTO_IPV6`, `IPPROTO_TCP` or `IPPROTO_UDP`
  - Each option (`optname`) has a specific value (`void *optval`) and length (`optlen`). Examples:
    - Socket API: `SO_KEEPALIVE`, `SO_BROADCAST`, `SO_REUSEADDR`
    - IP protocol:
      - Multicast groups: `IP_ADD_MEMBERSHIP`, `IP_DROP_MEMBERSHIP`…
      - MTU: `IP_MTU`, `IP_MTU_DISCOVER` (Path MTU Discovery)
      - IP datagram fields: `IP_OPTIONS`, `IP_TTL`, `IP_TOS`
- Options can also be changed with `sysctl` or `/proc` filesystem (e.g. `ip_default_ttl`…)

TCP Sockets

- Options SO_SNDBUF and SO_RCVBUF (Socket API level)
  - Size of send and receive buffers
  - Bigger sizes allow for more efficient management of the TCP window (timestamps, window scaling for flow control…)
  - Available via sysctl or /proc (net.ipv4.tcp_rmem and net.ipv4.tcp_wmem)
- Other options (TCP protocol level)
  - TCP_NODELAY disables the Nagle algorithm
  - TCP_QUICKACK disables delayed ACKs
  - Available via sysctl or /proc
- Urgent messages (URG flag in TCP header)
  - Add flag MSG_OOB (out-of-band) in send()
  - The process or process group set as the socket's owner will receive SIGURG:

```
fcntl(socket, F_SETOWN, pid);
```

IPv6 Sockets

```
tcp_socket = socket(AF_INET6, SOCK_STREAM, 0);
udp_socket = socket(AF_INET6, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET6, SOCK_RAW, protocol);
```

- As with `AF_INET`, `SOCK_STREAM` is based on TCP and `SOCK_DGRAM` on UDP
- IPv6 is almost fully compatible with IPv4 implementation
- Socket address:

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   // Address family: AF_INET6
    in_port_t       sin6_port;     // Port number
    uint32_t        sin6_flowinfo; // Flow ID
    struct in6_addr sin6_addr;     // IPv6 address
    uint32_t        sin6_scope_id; // only if link-local
};

struct in6_addr {
    unsigned char   s6_addr[16];   // 128-bit IPv6 address
};
```

IPv6 Sockets

- The address (**struct** in6_addr) can be instantiated using the following variables:
  - in6addr_any (similar to INADDR_ANY)
  - in6addr_loopback (similar to INADDR_LOOPBACK)
- To develop applications compatible with IPv4 and IPv6, dependencies with address format must be avoided
  - **struct** sockaddr is defined just to avoid compiler warnings (same size as **struct** sockaddr_in)
  - **struct** sockaddr_storage is defined to store both **struct** sockaddr_in and **struct** sockaddr_in6
- Example:

```
struct sockaddr_storage addr;
socklen_t addrlen = sizeof(addr);
accept(sd, (struct sockaddr *) &addr, &addrlen);
```

  - addr contains an IPv4 or IPv6 address
  - addrlen contains the length of the returned structure

IPv4 and IPv6 Servers

- A single socket for IPv6 and IPv4 (dual stack)
  - Disable option `IPV6_V6ONLY` in the socket
    - The default value for this option is defined in `net.ipv6.bindv6only` (disabled by default)
  - Call `bind()` with `in6addr_any` (i.e., `::`)
  - Use of mapped IPv4 addresses (`192.168.0.1` → `::FFFF:192.168.0.1`)
  - Not supported in all systems
- Two sockets, one for each stack
  - Enable option `IPV6_V6ONLY` in the IPv6 socket
  - Get valid addresses to create a socket with each stack

Summary: UDP Server Scheme

```
hints.ai_flags    = AI_PASSIVE;  // Return 0.0.0.0 or ::
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) {
  bytes = recvfrom(sd, buf, 80, 0, (struct sockaddr *) &client, &len);
  buf[bytes] = '\0';

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

  printf("Message from %s:%s: %s\n", host, serv, buf);

  sendto(sd, buf, bytes, 0, (struct sockaddr *) &client, len);
}
```

Summary: TCP Server Scheme

```
hints.ai_flags    = AI_PASSIVE;  // Return 0.0.0.0 or ::
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) {
  clisd = accept(sd, (struct sockaddr *) &client, &len);

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

  while (bytes = recv(clisd, buf, 80, 0)) { // Check message!
    buf[bytes] = ‘\0’;
    printf("\tMessage: %s\n", buf);
    send(clisd, buf, bytes, 0);
  }
}
```

Concurrent Servers

Need

- The server must attend several clients concurrently
- In general, calls are blocking
  - `accept()` waits for the client connection
  - `recv()` and `recvfrom()` wait for data to arrive
  - `send()` and `sendto()` wait if the send buffer is full

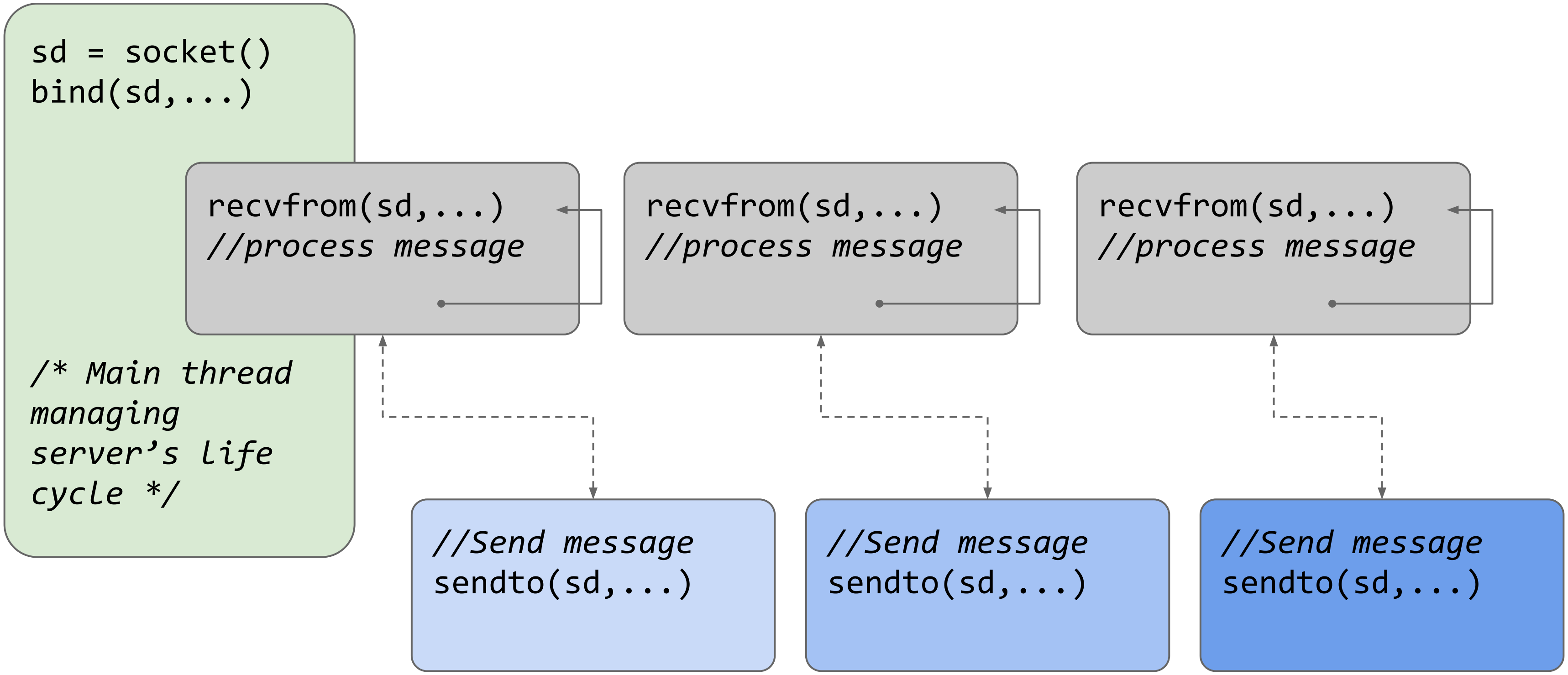
Tools

- Operations are concurrent over socket descriptors
  - Multiple threads can call `accept()` to establish a connection
  - Multiple threads can call `recvfrom()` to receive data
- All threads will block on the call and only one of them will be unblocked when a connection request or data is received
- Threads share address space and file descriptors (sockets)
- Processes inherit file descriptors (sockets)
- Synchronous I/O multiplexing can be also used, but the program logic is much more complex

Concurrent Servers: `SOCK_DGRAM`

Pre-fork Pattern

- Concurrent reception of messages
- The server creates a set of processes/threads to receive messages with `recvfrom()` and process them
- Concurrency is at the message level



```
sd = socket()
bind(sd,...)

/* Main thread
managing
server's life
cycle */
```

```
recvfrom(sd,...)
//process message
```

```
recvfrom(sd,...)
//process message
```

```
recvfrom(sd,...)
//process message
```

```
//Send message
sendto(sd,...)
```

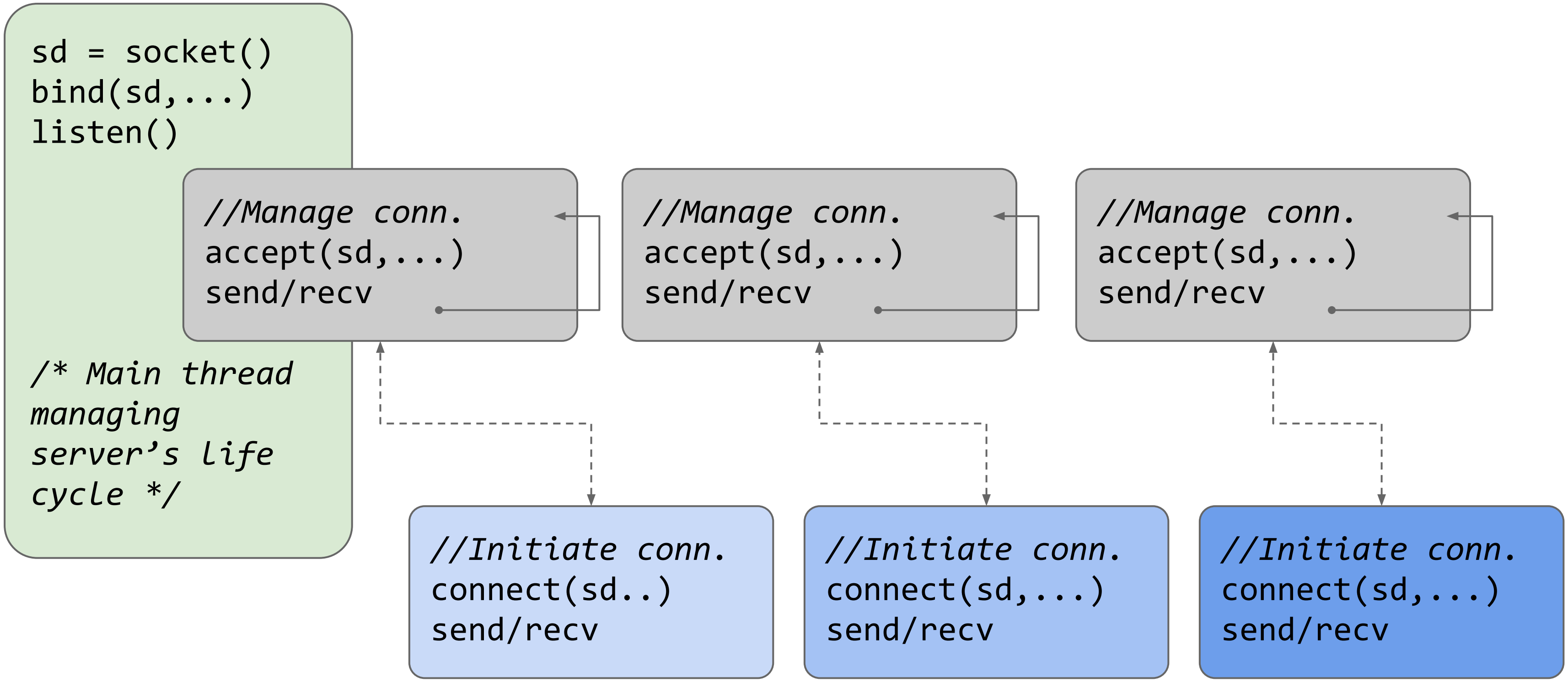
```
//Send message
sendto(sd,...)
```

```
//Send message
sendto(sd,...)
```

Concurrent servers: `SOCK_STREAM`

Pre-fork Pattern

- Concurrent connection management
- The server creates set of processes/threads to manage each connection with `accept()`
- Concurrency is at the connection level



```
sd = socket()
bind(sd,...)
listen()
```

```
/* Main thread
managing
server's life
cycle */
```

```
//Manage conn.
accept(sd,...)
send/recv
```

```
//Manage conn.
accept(sd,...)
send/recv
```

```
//Manage conn.
accept(sd,...)
send/recv
```

```
//Initiate conn.
connect(sd..)
send/recv
```

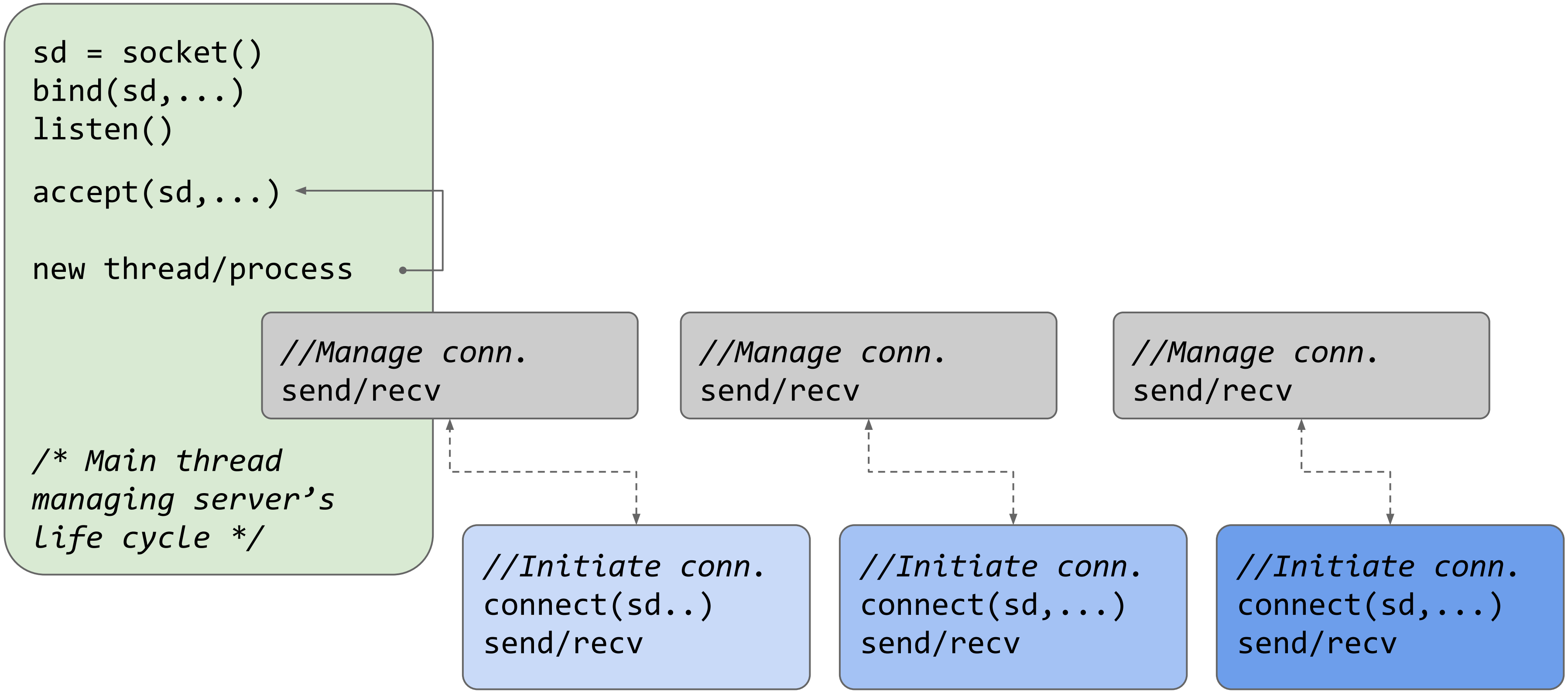
```
//Initiate conn.
connect(sd,...)
send/recv
```

```
//Initiate conn.
connect(sd,...)
send/recv
```

Concurrent Servers: `SOCK_STREAM`

Accept-and-fork pattern

- Concurrent connection management
- Main process/thread establishes connections with `accept()` and creates a process/thread to manage each one
- Concurrency is at the connection level



```
sd = socket()
bind(sd,...)
listen()

accept(sd,...)

new thread/process
```

```
/* Main thread
managing server's
life cycle */
```

```
//Manage conn.
send/recv
```

```
//Manage conn.
send/recv
```

```
//Manage conn.
send/recv
```

```
//Initiate conn.
connect(sd..)
send/recv
```

```
//Initiate conn.
connect(sd,...)
send/recv
```

```
//Initiate conn.
connect(sd,...)
send/recv
```