Socket Programming

Data types and structures for writing client-server programs

Introduction

- Transport layer and layers below
  - Basic communication
  - reliability
- Application Layer Functionality
  - Abstractions
  - Names:
    - define symbolic names to identify both physical and abstract resources available on an internet

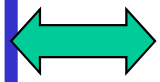
- Network
  - transfers bits
  - operates at application's request
- Application determines
  - what/when/where to send
  - Meaning of bits
    => Application programs are the entities that communicate with each other, not the computers or users.
- Important point: For 2 application programs to communicate with each other, one application initiates communication and the other accepts.

Client/Server Model

How 2 application programs make contact

Server

- Starts first
- Passively waits for contact from a client at a prearranged location
- Responds to requests



Client

- Starts second
- Actively contacts a server with a request
- Waits for response from server

• **Client-server paradigm**: form of communication used by all network applications

Characteristics of a Client

- Arbitrary application program
- Becomes client temporarily
- Can also perform other computations
- Invoked directly by user
- Runs locally on user's computer
- Actively initiates contact with a server
- Contacts one server at a time

Characteristics of a Server

- Special-purpose, privileged program
- Dedicated to providing one service
- Can handle multiple remote clients simultaneously
- Invoked automatically when system boots
- Executes forever
- Needs powerful computer and operating system
- Waits passively for client contact
- Accepts requests from arbitrary clients

Terminology

- Server
  - An executing program that accepts contact over the network
- server-class computer
  - Hardware sufficient to execute a server
- Informally
  - Term “server” often applied to computer

Direction of Data Flow

- Data can flow
  - from client to server only
  - from server to client only
  - in both directions
- Application protocol determines flow
- Typical scenario
  - Client sends request(s)
  - Server sends responses(s)

Server CPU use

- Facts
  - Server operates like other applications
    - uses CPU to execute instructions
    - Performs I/O operations
  - Waiting for data to arrive over a network does not require CPU time
- Consequence
  - Server program uses only CPU when servicing a request

The Socket Interface

- The ***Berkeley Sockets API***
  - Originally developed as part of BSD Unix (under gov't grant)
    - BSD = Berkeley Software Distribution
    - API=Application Program Interface
  - Now the most popular API for C/C++ programmers writing applications over TCP/IP
    - Also emulated in other languages: Perl, Tcl/Tk, etc.
    - Also emulated on other operating systems: Windows, etc.

The Socket Interface

- The basic ideas:
  - a ***socket*** is like a file:
    - you can read/write to/from the network just like you would a file
  - For connection-oriented communication (e.g. TCP)
    - servers (passive open) do **listen** and **accept** operations
    - clients (active open) do **connect** operations
    - both sides can then do **read** and/or **write** (or **send** and **recv)**
    - then each side must **close**
    - There are more details, but those are the most important ideas
  - Connectionless (e.g. UDP): uses **sendto** and **recvfrom**

Sockets And Socket Libraries

- **On some other systems, socket procedures are *not* part of the OS**
  - instead, they are implemented as a library, linked into the application object code (e.g. a DLL under Windows)
  - Typically, this DLL makes calls to similar procedures that ***are*** part of the native operating system.
  - This is what the Comer text calls a ***socket library***
    - A socket library simulates Berkeley sockets on OS's where the underlying OS networking calls are different from Berkeley sockets

Some definitions

- **Data types**

int8_t	signed 8-bit integer
int16_t	signed 16-bit integer
int32_t	signed 32-bit integer
uint8_t	unsigned 8-bit integer
uint16_t	unsigned 16-bit integer
uint32_t	unsigned 32-bit integer
u_char	Unsigned 8-bit character
u_short	Unsigned 16-bit integer
u_long	Unsigned 32-bit integer

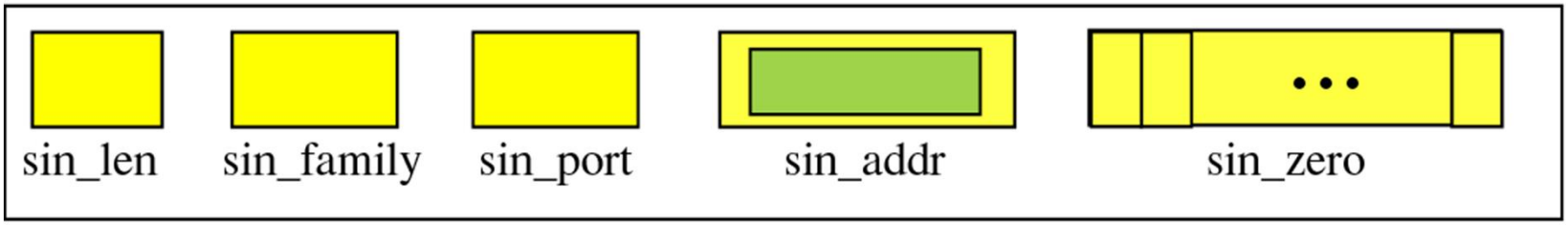
More Definitions

- Internet Address Structure

```
struct in_addr
{
        in_addr_t      s_addr;
};
```

```
struct     in_addr
{
                u_long   s_addr ;
} ;
```

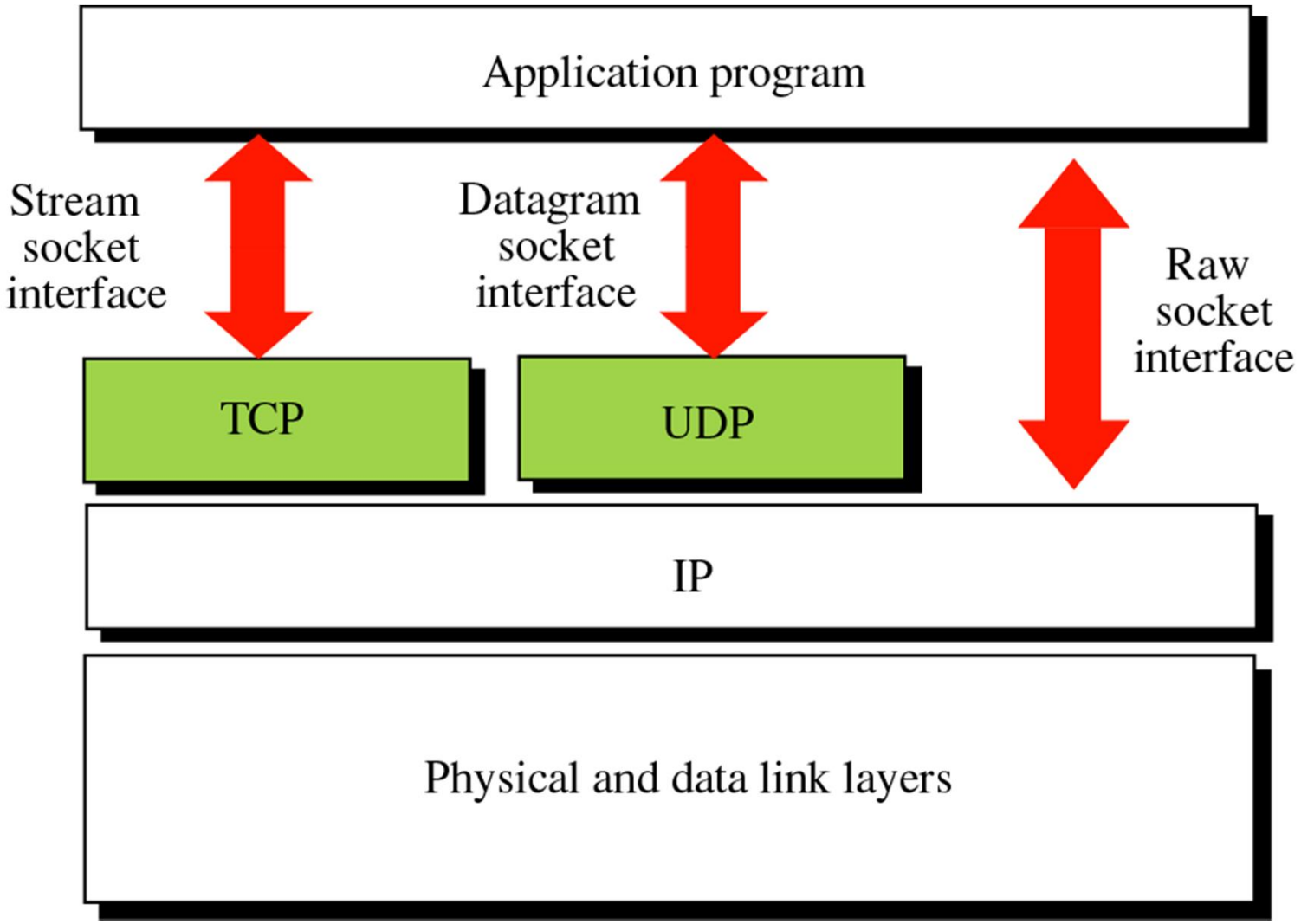
Socket address structure



sockaddr_in

```
struct    sockaddr_in
{
        u_char              sin_len ;
        u_short             sin_family ;
        u_short             sin_port ;
        struct  in_addr     sin_addr ;
        char                sin_zero [8] ;
} ;
```

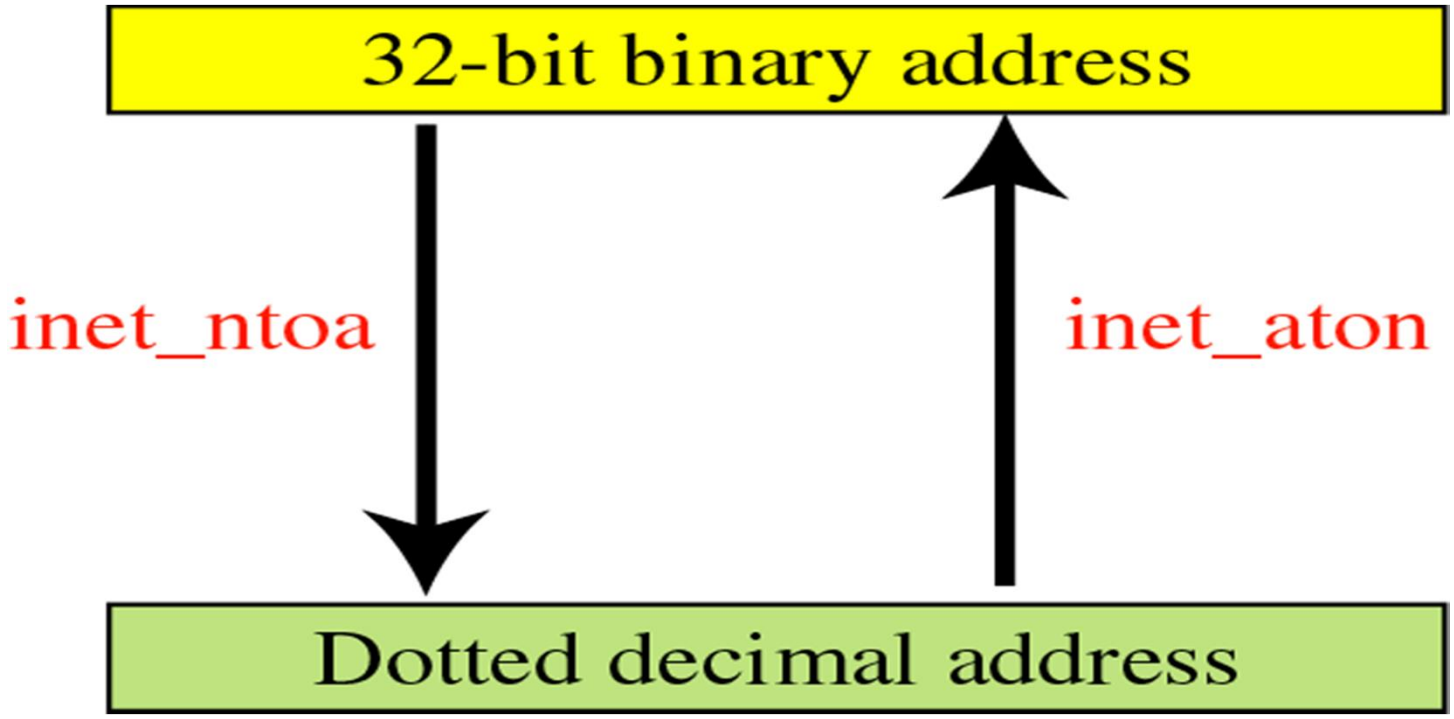
Socket Types



Address Transformation

int **inet_aton** (const char ***strptr** , struct in_addr ***addrptr**) ;

char ***inet_ntoa** (struct in_addr ***inaddr**) ;



Byte-Manipulation Functions

- In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields.
  - Cannot use string functions (strcpy, strcmp, …) which assume null character termination.

```
void  *memset  ( void *dest , int  chr , int len ) ;

void  *memcpy  ( void *dest , const void  *src , int len ) ;

int     memcmp  ( const void *first , const void  *second , int len ) ;
```

Procedures That Implement The Socket API

Creating and Deleting Sockets

- `fd=`**`socket`**`(protofamily, type, protocol)`
  Creates a new socket. Returns a file descriptor (fd). Must specify:
  - the protocol family (e.g. TCP/IP)
  - the type of service (e.g. STREAM or DGRAM)
  - the protocol (e.g. TCP or UDP)
- **`close`**(fd)
  Deletes socket.
  For connected STREAM sockets, sends EOF to close connection.

Procedures That Implement The Socket API

- **`bind`**(fd)
  Used by server to establish port to listen on.
  When server has >1 IP addrs, can specify "ANY", or a specific one
- **`listen`** (fd, queuesize)
  Used by connection-oriented servers only, to put server "on the air"
  Queuesize parameter: how many pending connections can be waiting
- afd = **`accept`** (lfd, caddress, caddresslen)
  Used by connection-oriented servers to accept one new connection
  - There must already be a listening socket (lfd)
  - Returns afd, a new socket for the new connection, and
  - The address of the caller (e.g. for security, log keeping. etc.)

Procedures That Implement The Socket API

How Clients Communicate with Servers

- **`connect`** (fd, saddress, saddreslen)

  Used by connection-oriented clients to connect to server
  - There must already be a socket bound to a connection-oriented service on the fd
  - There must already be a listening socket on the server
  - You pass in the address (IP address, and port number) of the server.

  Used by connectionless clients to specify a "default send to address"
  - Subsequent "writes" or "sends" don't have to specify a destination address
  - BUT, there really ISN'T any connection established… this is a bad choice of names!

Procedures That Implement The Socket API

How Clients Communicate with Servers

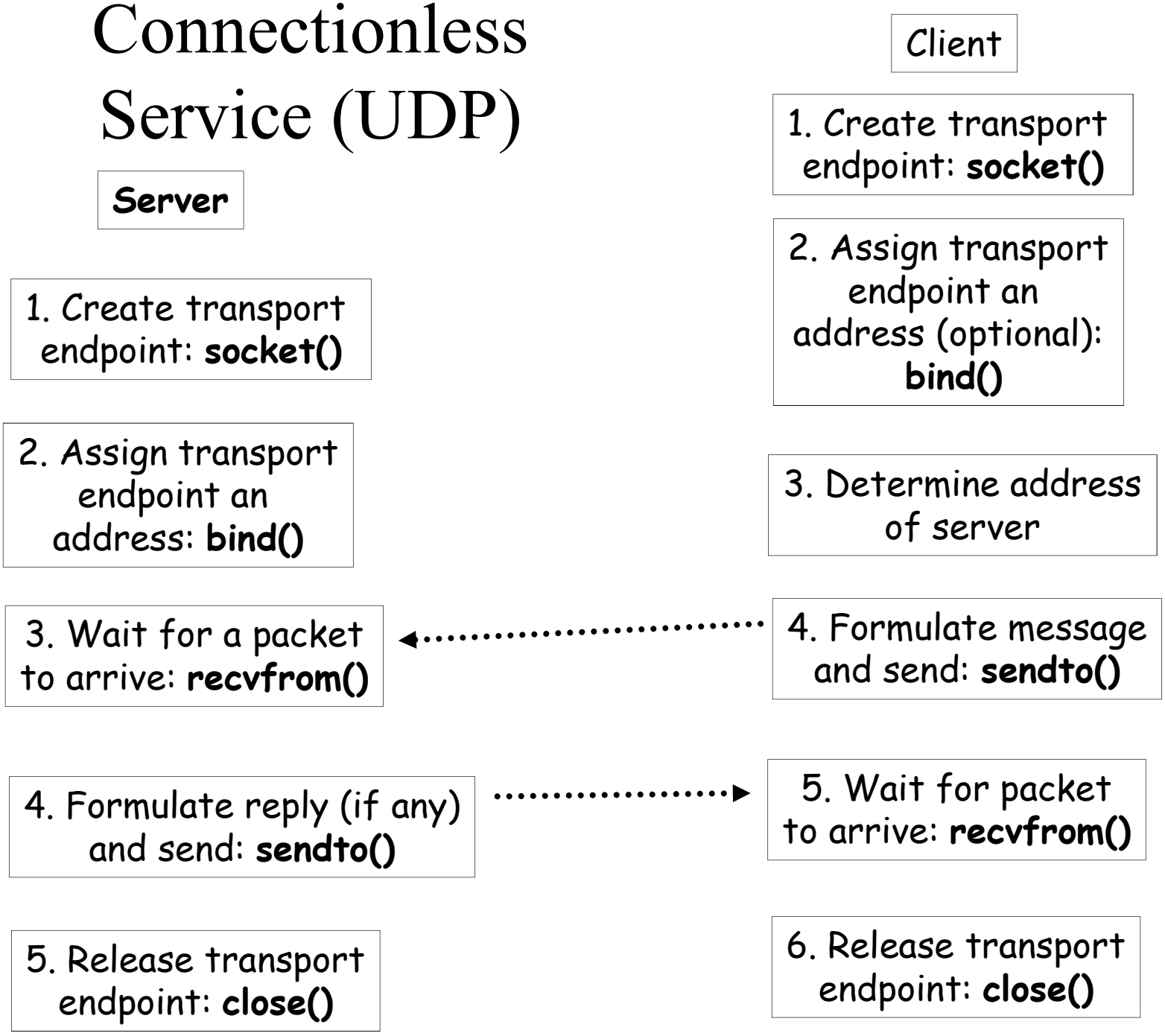
- **`send`** (fd, data, length, flags)
  `sendto` (fd, data, length, flags, destaddress, addresslen)
  `sendmsg` (fd, msgstruct, flags)
  `write` (fd, data, length)

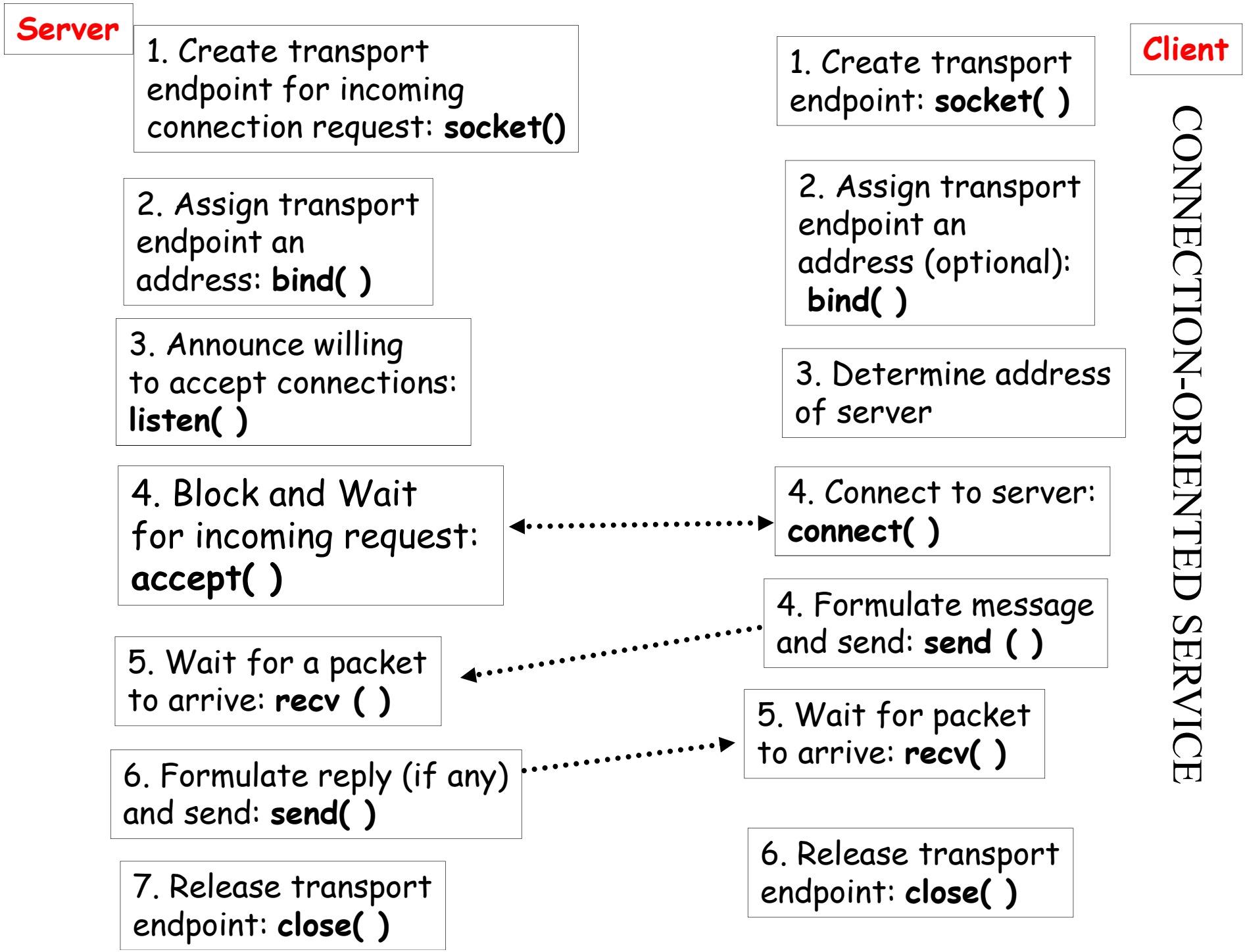
  Used to send data.
  - **send** requires a connection (or for UDP, default send address) be already established
  - **sendto** used when we need to specify the dest address (for UDP only)
  - sendmsg is an alternative version of sendto that uses a struct to pass parameters
  - write is the "normal" write function; can be used with both files and sockets
- **`recv`** (...) **`recvfrom`** (...) **`recvmsg`** (...) **`read`** (...)

  Used to receive data… parameters are similar, but in reverse

  (destination => source, etc…)

Connectionless Service (UDP)




Server
1. Create transport endpoint for incoming connection request: socket()
2. Assign transport endpoint an address: bind( )
3. Announce willing to accept connections: listen( )
4. Block and Wait for incoming request: accept( )
5. Wait for a packet to arrive: recv ( )
6. Formulate reply (if any) and send: send( )
7. Release transport endpoint: close( )
Client
1. Create transport endpoint: socket( )
2. Assign transport endpoint an address (optional): bind( )
3. Determine address of server
4. Connect to server: connect( )
4. Formulate message and send: send ( )
5. Wait for packet to arrive: recv( )
6. Release transport endpoint: close( )
CONNECTION-ORIENTED SERVICE