Operating Systems

Sockets

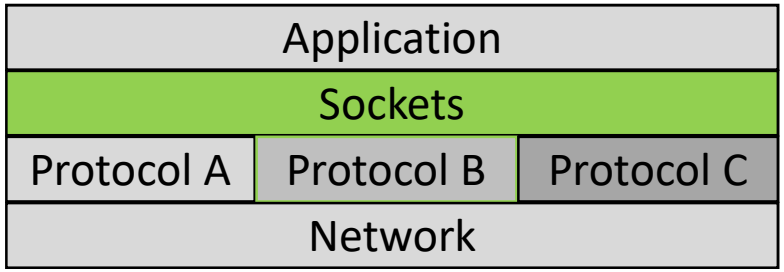
ENCE 360

Outline

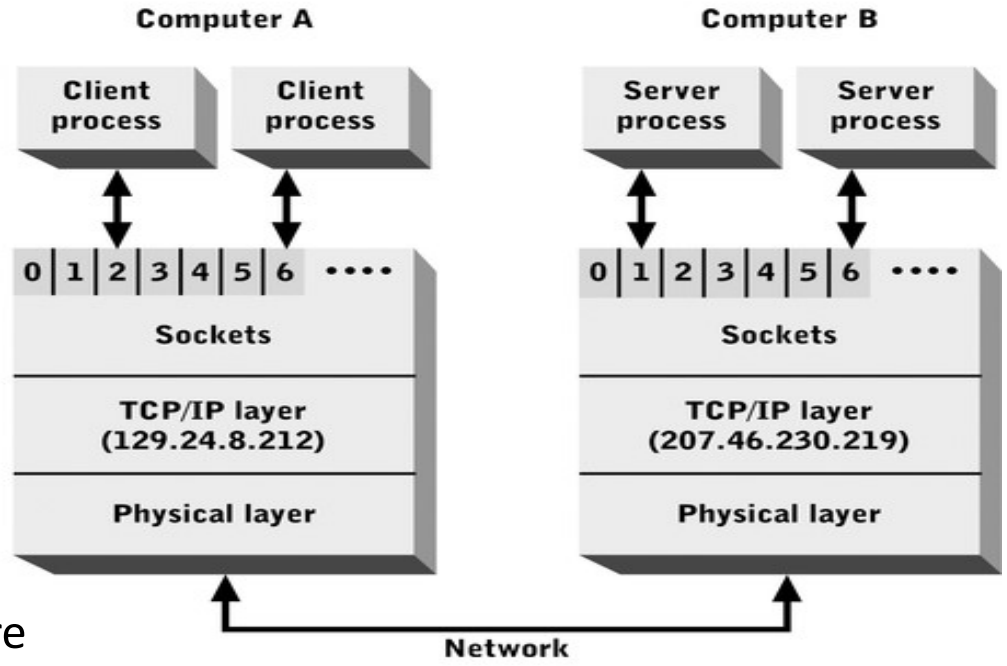
- Introduction
- Details
- Example code
- Socket options

Socket Overview

- Socket - An end-point for connection to another process (remote or local)
  - What application layer "plugs into"



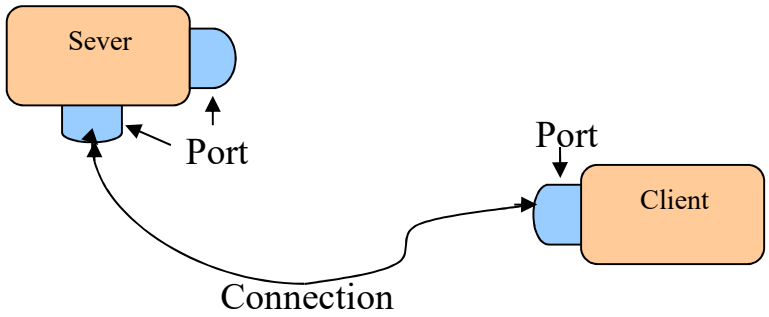
- User sees descriptor - integer index/handle
  - Like: file index from `open()`
  - Returned by `socket()` call (more later)
  - Programmer cares about Application Programming Interface (API) → similar to file I/O

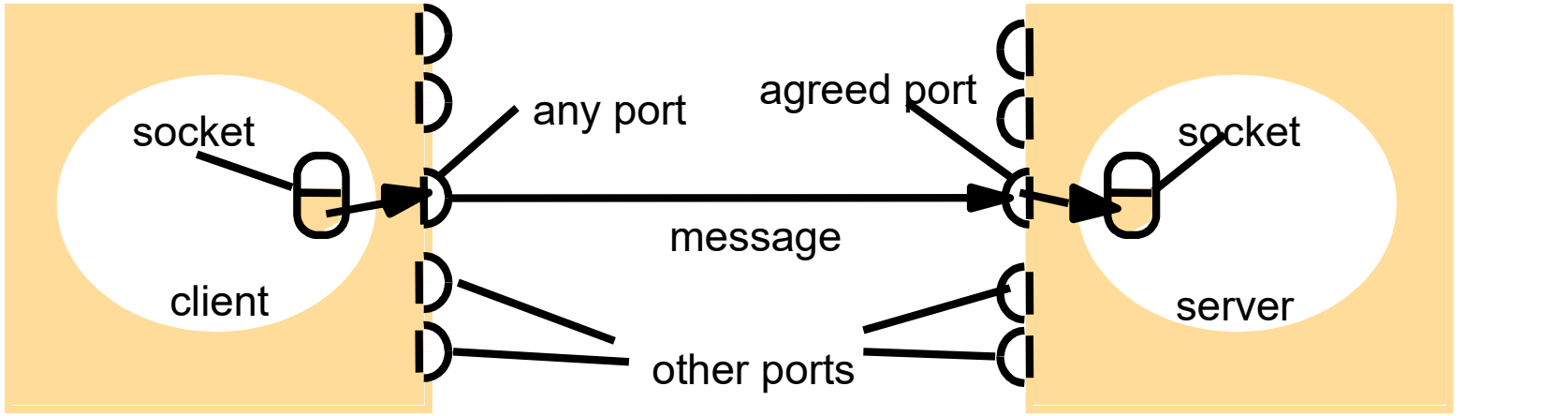


(TCP=Transport Control Protocol, IP=Internet Protocol)

Connection Endpoints

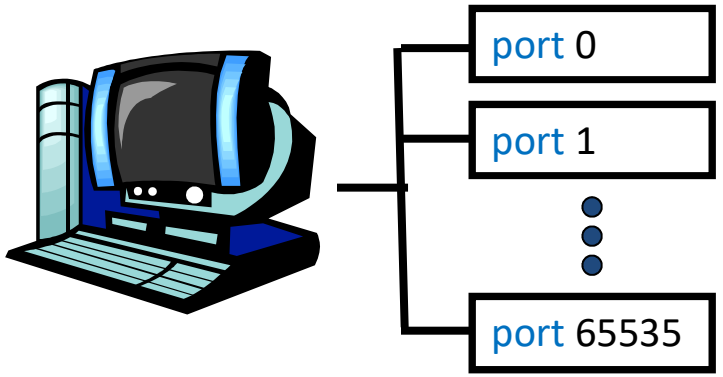
- End point determined by two things:
  - Host address: e.g., IP address
  - Port number
- Two end-points determine connection → socket pair

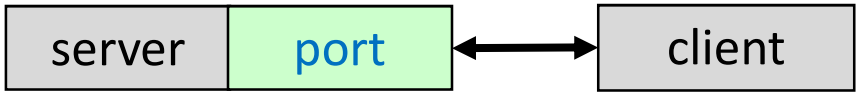


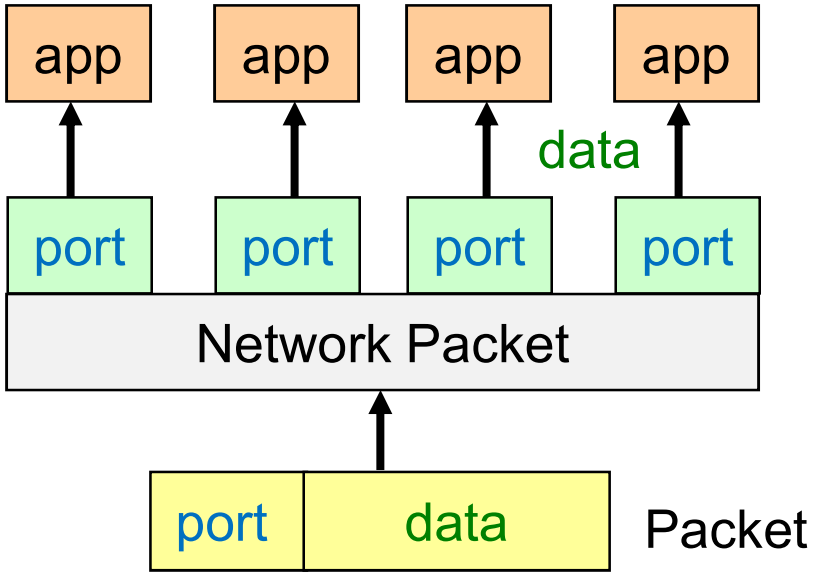


Ports

- Each host has 65,536 ports
  - 16-bit integer
- Some ports are reserved for specific apps (`/etc/services`)
  - FTP 20,21
  - Telnet 23
  - HTTP 80
- Ports below 1024 are reserved
  - User level 1024+
- Ports 1024-5000 ephemeral
  - Assigned in outgoing connection
- Ports 5001+ services

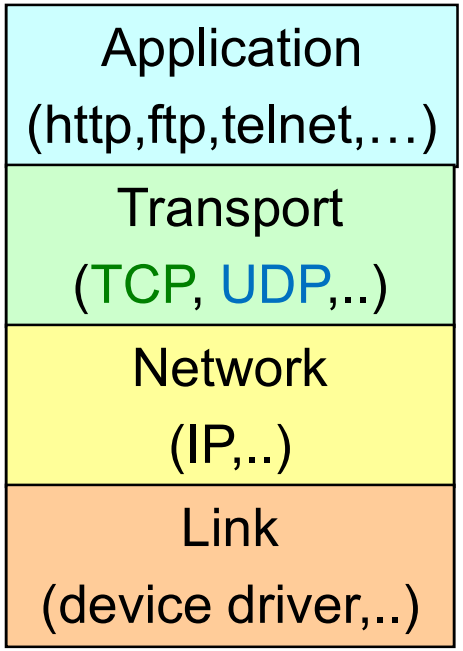






Two Main Network Transport Protocols Today

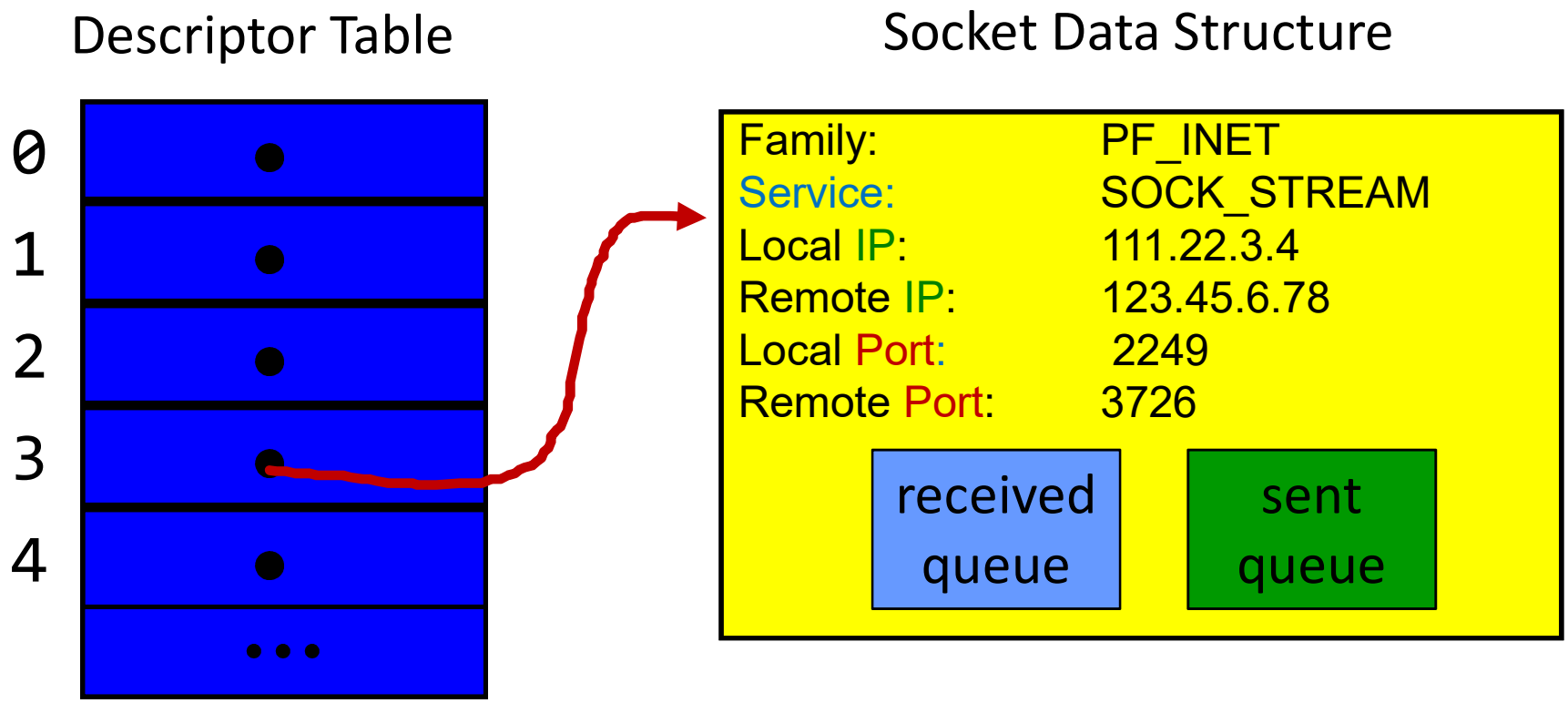
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - Connectionless
  - SOCK_DGRAM
- TCP: Transmission Control Protocol
  - reliable (in order, all arrive, no duplicates)
  - flow control
  - connection-based
  - SOCK_STREAM



TCP ~95% of all flows and packets on Internet (What applications may use UDP?)

More in a networks course!

Socket Descriptor Data Structure



Service is the Transport Protocol
IP (Internet Protocol) - address of computer
Port - specifies which process on computer

Outline

- Introduction (done)
- Details (next)
- Example code
- Socket options

Unix Network Programming, W. Richard Stevens, 2nd edition, ©1998, Prentice Hall

Beej's Guide to Network Programming, Brian Hall, ©2015, self-published, *http://beej.us/guide/bgnet/*

Addresses and Sockets

- Structure to hold address information
- Functions pass info (e.g., address) from user to OS

  `bind()`

  `connect()`

  `sendto()`
- Functions pass info (e.g., address) from OS to user

  `accept()`

  `recvfrom()`

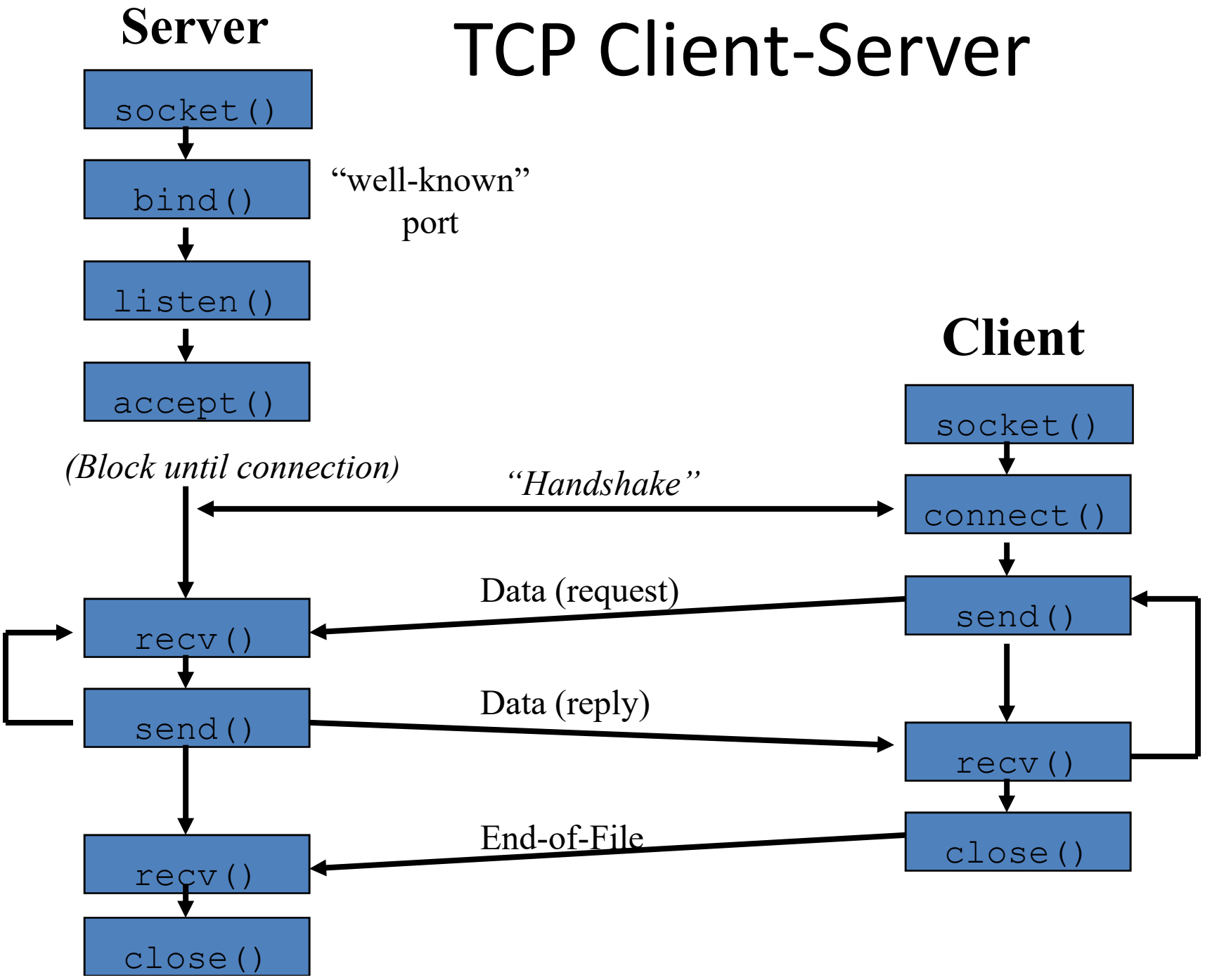
Socket Address Structure

```
struct in_addr {
  in_addr_t       s_addr;      /* 32-bit IPv4 addresses */
};

struct sockaddr_in {
  unit8_t         sin_len;     /* length of structure */
  sa_family_t     sin_family;  /* AF_INET */
  in_port_t       sin_port;    /* TCP/UDP port number */
  struct in_addr  sin_addr;    /* IPv4 address (above) */
  char            sin_zero[8];/* unused */
};
```

Also “generic” and “IPv6” socket structures

TCP Client-Server



socket()

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

Create socket, giving access to transport layer service

- *family* is one of
  - AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
  - AF_ROUTE (access to routing tables), AF_KEY (for encryption)
- *type* is one of
  - SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
  - SOCK_RAW (for special IP packets, PING, etc. Must be root)
    - setuid bit (`-rwsr-xr-x root 2014 /sbin/ping*`)
- *protocol* is 0 (used for some raw socket options)
- upon success returns socket descriptor
  - Integer, like file descriptor → index used internally
  - Return -1 if failure

bind()

```
int bind(int sockfd, const struct sockaddr *myaddr,
         socklen_t addrlen);
```

Assign local protocol address ("name") to socket

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is pointer to address `struct` with:
  - *port number* and *IP address*
  - if port is 0, then host will pick *ephemeral* port
    - not usually for server (exception RPC port-map)
  - IP address == `INADDR_ANY` (unless multiple nics)
- *addrlen* is length of structure
- returns 0 if ok, -1 on error
  - `EADDRINUSE` ("Address already in use")

`listen()`

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

Change socket state (to passive) for TCP server

- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of *incomplete* connections
  - historically 5
  - rarely above 15 even on moderately busy Web server!
- sockets default to active (for client)
  - change to passive so OS will accept connection

accept()

```
int accept(int sockfd, struct sockaddr
      *cliaddr, socklen_t *addrlen);
```

Return next completed connection

- blocking call (by default)
- *sockfd* is socket descriptor from `socket()`
- *cliaddr* and *addrlen* return protocol address from client
- returns brand new descriptor, created by OS
- note, if create new process or thread, can create concurrent server

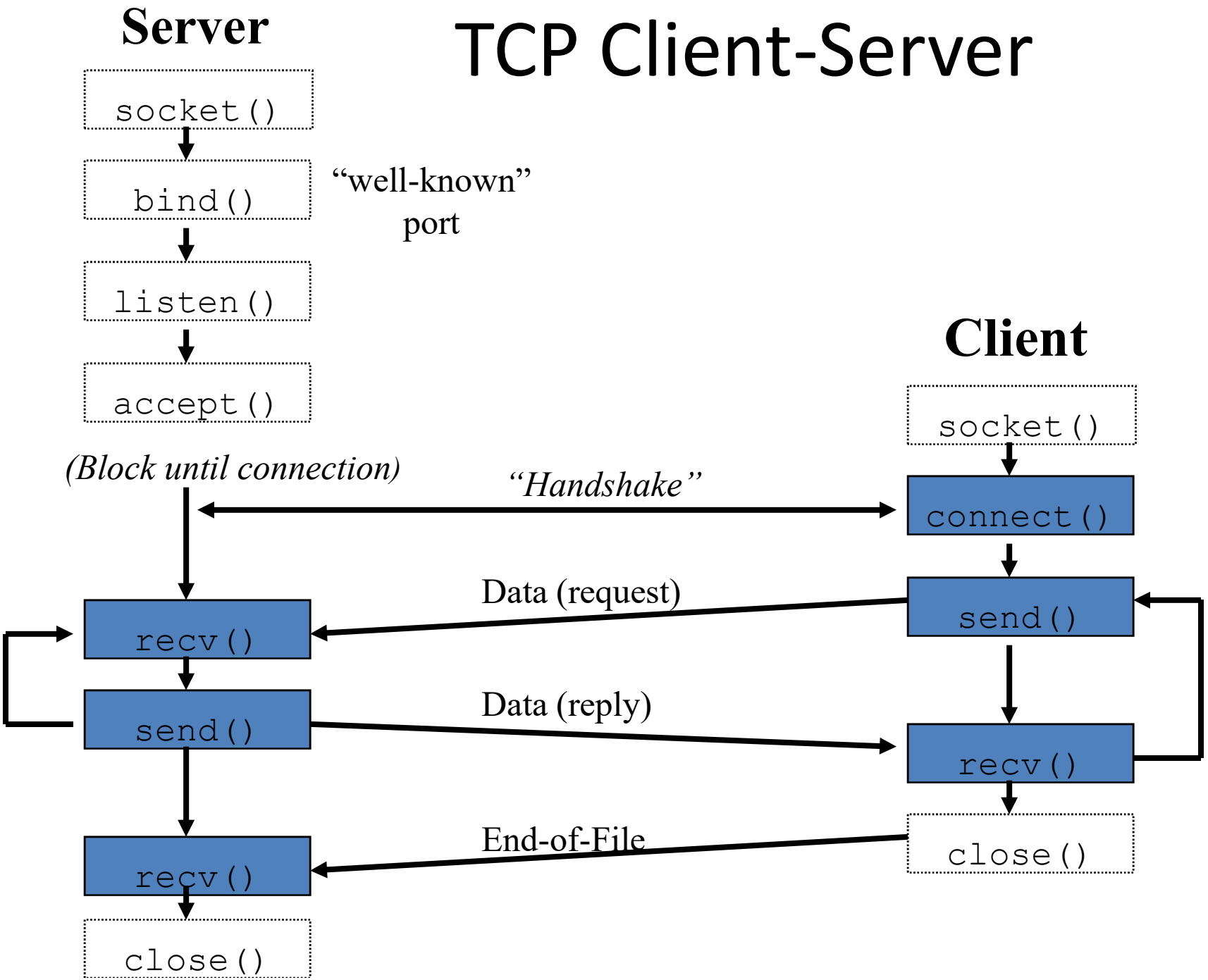
close()

```
int close(int sockfd);
```

Close socket for use

- *sockfd* is socket descriptor from `socket()`
- closes socket for reading/writing
  - returns (doesn't block)
  - attempts to send any unsent data
  - socket option SO_LINGER
    - block until data sent
    - or discard any remaining data
  - returns -1 if error

TCP Client-Server



connect()

```
int connect(int sockfd, const struct
      sockaddr *servaddr, socklen_t addrlen);
```

Connect to server

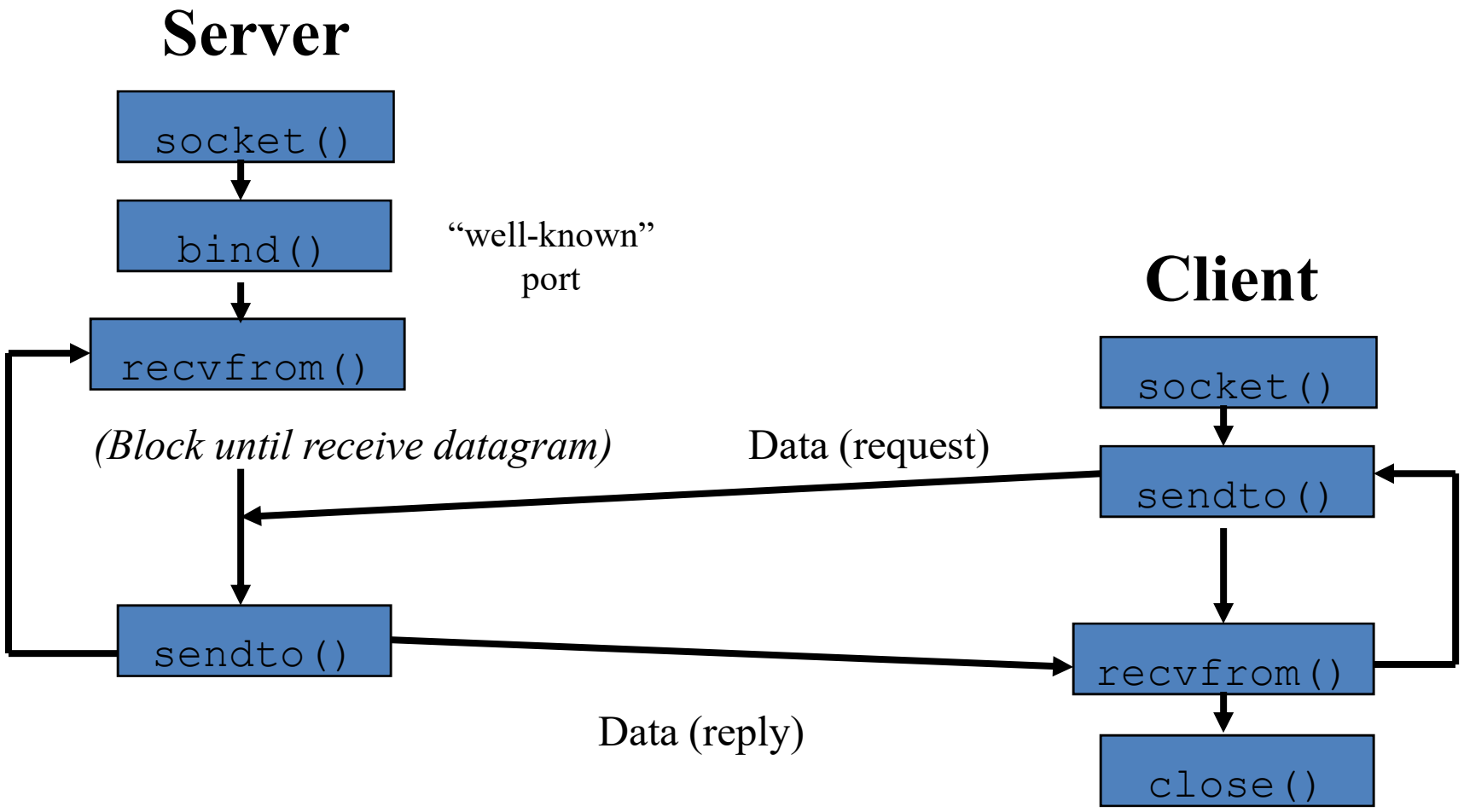
- *sockfd* is socket descriptor from `socket()`
- *servaddr* is pointer to structure with:
  - *port number* and *IP address*
  - must be specified (unlike `bind()`)
- *addrlen* is length of structure
- client doesn't need `bind()`
  - OS will pick ephemeral port
- returns socket descriptor if ok, -1 on error

Sending and Receiving

```
int recv(int sockfd, void *buff, size_t
  mbytes, int flags);
int send(int sockfd, void *buff, size_t
  mbytes, int flags);
```

- Same as `read()` and `write()` but with *flags*
  - MSG_DONTWAIT (this send non-blocking)
  - MSG_OOB (out of band data, 1 byte sent ahead)
  - MSG_PEEK (look, but don't remove)
  - MSG_WAITALL (don't return less than `mbytes`)
  - MSG_DONTROUTE (bypass routing table)

UDP Client-Server



- No "connection", no "handshake"
- No simultaneous close

Sending and Receiving

```
int recvfrom(int sockfd, void *buff, size_t mbytes, int
   flags, struct sockaddr *from, socklen_t *addrlen);
int sendto(int sockfd, void *buff, size_t mbytes, int
   flags, const struct sockaddr *to, socklen_t addrlen);
```

- Same as `recv()` and `send()` but with *addr*
  - `recvfrom` fills in address of where packet came from
  - `sendto` requires address of where sending packet to

Can `connect()` with UDP

- Record address and port of peer
  - Datagrams to/from others are not allowed
  - Does not do three way handshake, or connection
  - So, "connect" a misnomer, here. Should be `setpeername`()
- Use `send`() instead of `sendto`()
- Use `recv`() instead of `recvfrom`()
- Can change connect or unconnect by repeating `connect`() call
- (Can do similar with `bind()` on receiver)

Outline

- Introduction (done)
- Details (done)
- Example code (next)
- Socket options

Example Code

Server

```
% listen-tcp
listen-tcp - server to accept TCP connections
usage: listen-tcp <port>
        <port>  - port to listen on
% listen-tcp 7500
Listen activating.
Trying to create socket at port 7500...
Socket ready to go! Accepting connections....

received: 'Hello, world!'
received: 'Networking is awesome!'
server exiting
```

See:
"listen-tcp.c"
"talk-tcp.c"

Client

```
% talk-tcp
talk-tcp - client to try TCP connection to server
usage: talk-tcp <host> <port>
      <host>  - Internet name of server host
      <port>  - port
% talk-tcp localhost 7500
Talk activated.

Trying to connect to server localhost at port 7500...
Looking up localhost...
Found it.  Setting port connection to 7500...
Done. Creating socket...
Created. Trying connection to server...
Connection established!
Type in messages to send to server.
Hello, world!
sending: 'Hello, world!'
Networking is awesome!
sending: 'Networking is awesome!'
```

Outline

- Introduction (done)
- Details (done)
- Example code (done)
- Socket options (next)

Socket Options (General)

- `setsockopt()`,`getsockopt()`
- SO_LINGER
  - Upon close, discard data or block until sent
- SO_RCVBUF, SO_SNDBUF
  - Change buffer sizes
  - For TCP is “pipeline”, for UDP is “discard”
- SO_RCVLOWAT, SO_SNDLOWAT
  - How much data before “readable” via `select`()
- SO_RCVTIMEO, SO_SNDTIMEO
  - Timeouts

Socket Options (TCP)

- TCP_KEEPALIVE
  - Idle time before close (2 hours, default)
- TCP_MAXRT
  - Set timeout value
- TCP_NODELAY
  - Disable Nagle's Algorithm
  - Won't buffer data for larger chunk, but sends immediately

`fcntl()`

- ‘File control’ but used for sockets, too
- Set socket owner
- Get socket owner
- Set socket non-blocking

```
flags = fcntl(sockfd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(sockfd, F_SETFL, flags);
```

- Beware not getting flags before setting!

Outline

- Introduction (done)
- Details (done)
- Example code (done)
- Socket options (done)