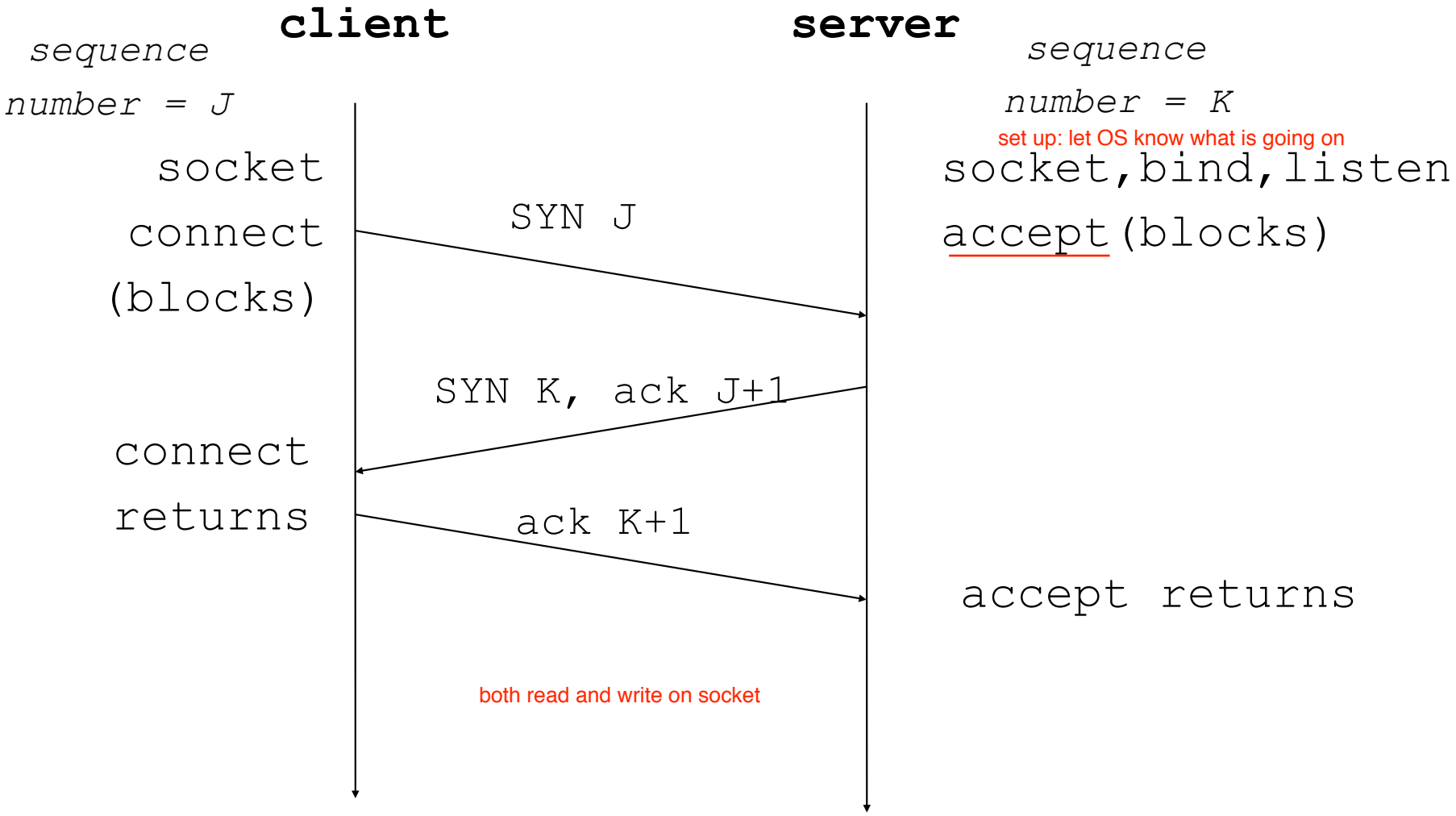
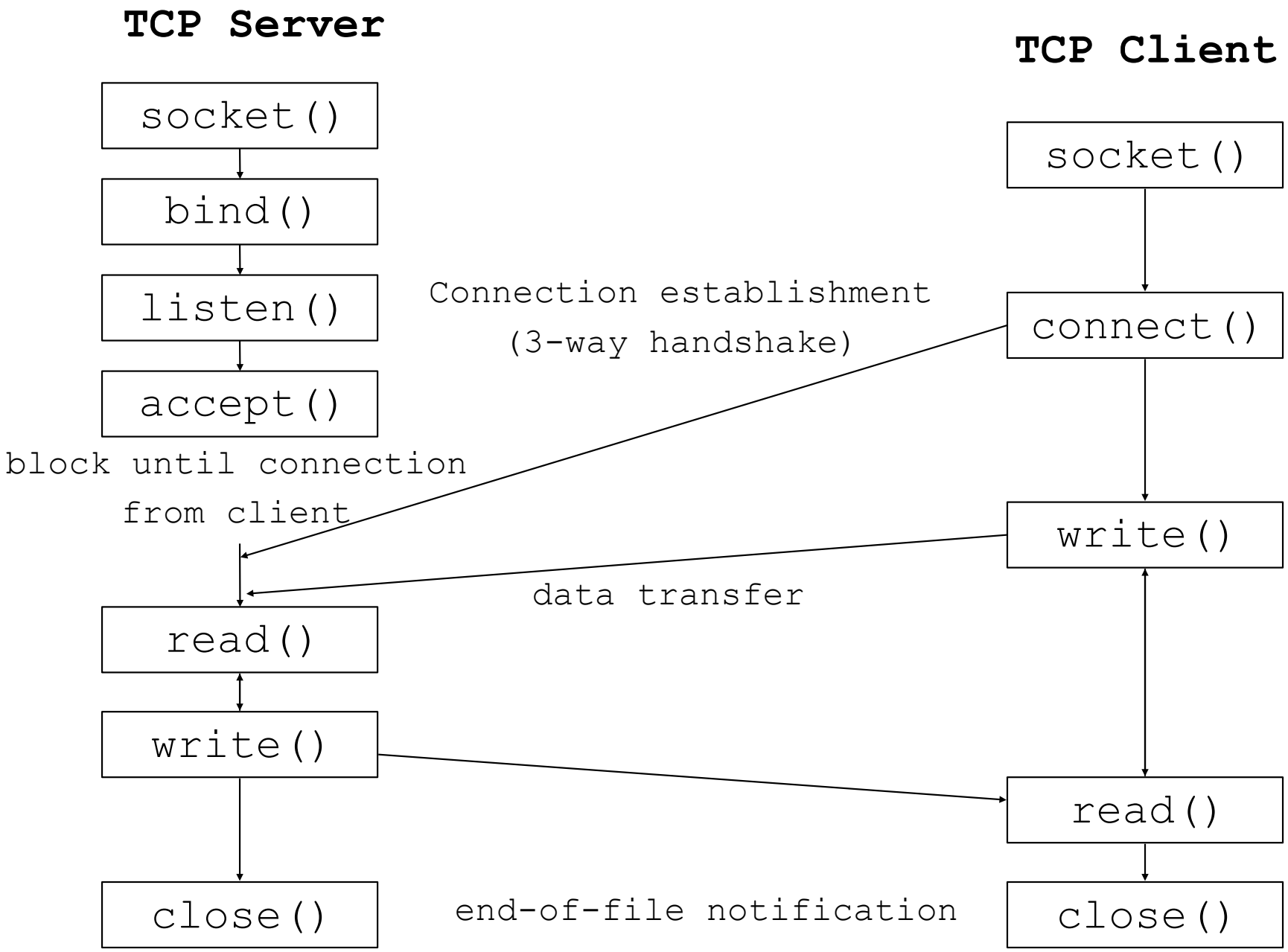
# Programming with Sockets in C

- The slides by themselves will not be sufficient to learn how to write socket code.
- If you did not attend class, then you will want to review the relevant chapters in Kerrisk (ch 56, 59, 60, 61.1 63.1, 63.2(. 1))
- The provided example code is also a good starting point.

# TCP: Three-way handshake




TCP Server
TCP Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
write()
read()
write()
close()
Connection establishment
(3-way handshake)
data transfer
read()
close()
end-of-file notification

# Connection-Oriented

## Server

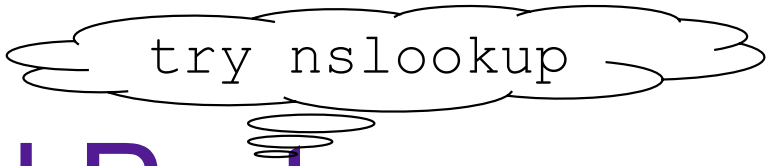
- Create a socket: `socket()`
- Assign a name to a socket: `bind()`
- Establish a queue for connections: `listen()`
- Get a connection from the queue: `accept()`

## Client

- Create a socket: `socket()`
- Initiate a connection: `connect()`

# Socket Types

- Two main categories of sockets
  - UNIX domain: both processes on the same machine performace better
  - INET domain: processes on different machines
- Three main types of sockets:
  - `SOCK_STREAM`: the one we will use for connection oriented stream
  - `SOCK_DGRAM`: for connectionless sockets
  - `SOCK_RAW` for DIY socket

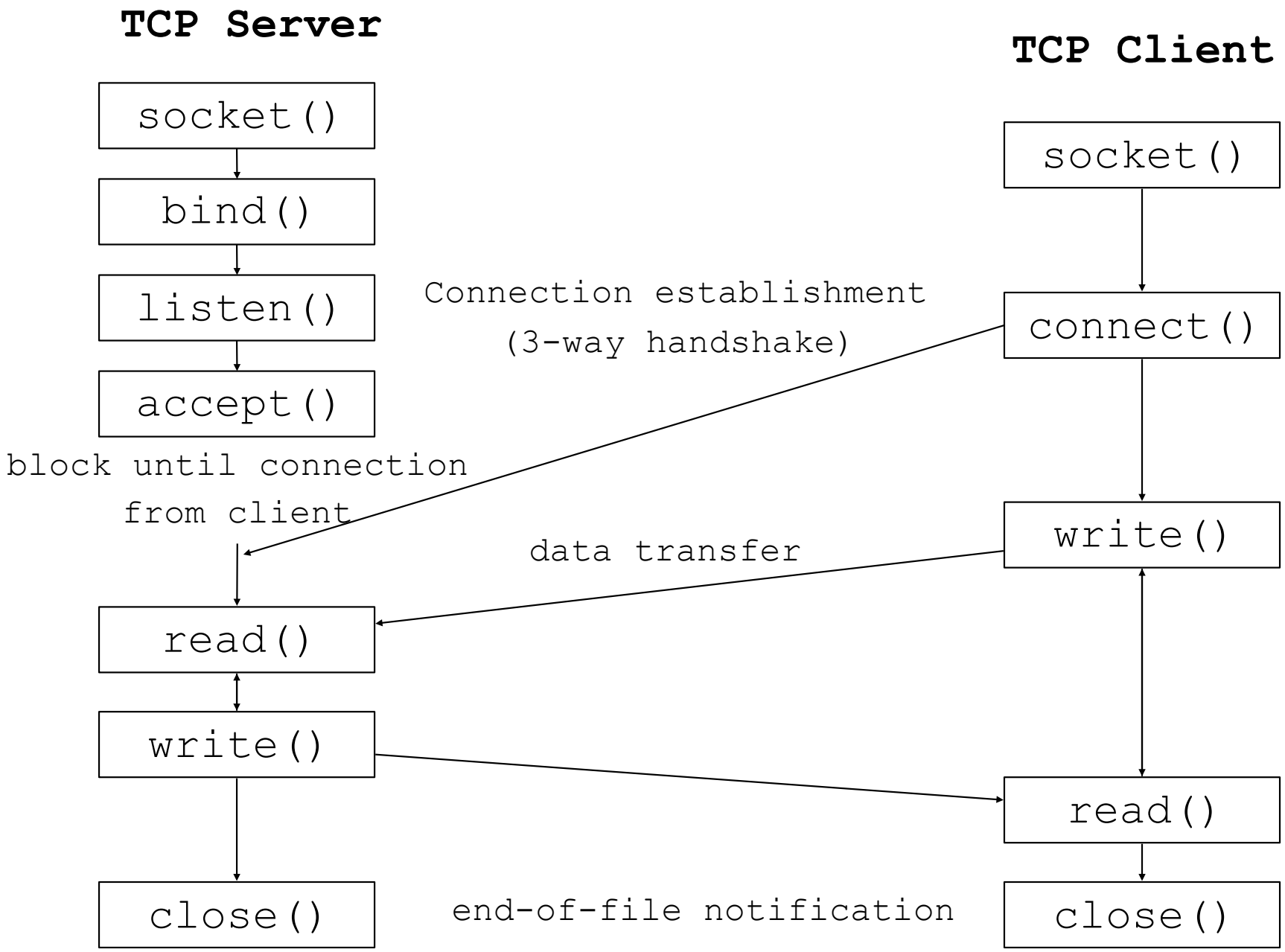



# Addresses and Ports

- A socket pair is the two endpoints of the connection.
- An endpoint is identified by an IP address and a port.
- IPv4 addresses are 4 8-bit numbers: 0 ~ 255
  - 128.100.31.198 = greywolf
  - 128.100.31.203 = redwolf
- Ports
  - because multiple processes can communicate with a single machine we need another identifier.

# More on Ports


- Well-known ports: 0-1023
  - 80 = http
  - 22 = ssh
  - 23 = telnet
  - 21 = ftp
  - 25 = smtp (mail)
  - 194 = irc
- Registered ports: 1024-49151
  - 2709 = supermon
  - 26000 = quake
  - 3724 = world of warcraft
- Dynamic (private) ports: 49152-65535


TCP Server
TCP Client
socket()
bind()
listen()
accept()
block until connection
from client
Connection establishment
(3-way handshake)
data transfer
read()
write()
close()
socket()
connect()
write()
read()
close()
end-of-file notification

# Server side

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

- family specifies protocol family:
  - `PF_INET` – IPv4
  - `PF_LOCAL` – Unix domain
- type
  - `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
- protocol
  - set to 0 except for RAW sockets
- returns a socket descriptor

# bind to a name

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

- sockfd – returned by socket
- `struct sockaddr_in {`

```
    short           sin_family; /*PF_INET */
    u_short         sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8]; /*filling*/
};
```

- `sin_addr` can be set to `INADDR_ANY` to communicate on any network interface.

cheap way of getting differnet socket, gives padding -> for different socket type

# Set up queue in kernel

creates holding area…

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

- after calling `listen`, a socket is ready to accept connections

the queue holds a queue of request

- prepares a queue in the kernel where partially completed connections wait to be accepted.
- `backlog` is the maximum number of partially completed connections that the kernel should queue.

max length of queue

# Complete the connection

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

- blocks waiting for a connection (from the queue) process is suspended, is not going to resume until conditions satisfied
- returns a new descriptor which refers to the TCP connection with the client specifically the socket descriptor of same type as `sockfd`
- `sockfd` is the listening socket socket created with socket() in local process
- `cliaddr` is the address of the client incoming connection
- reads and writes on the connection will use the socket returned by accept

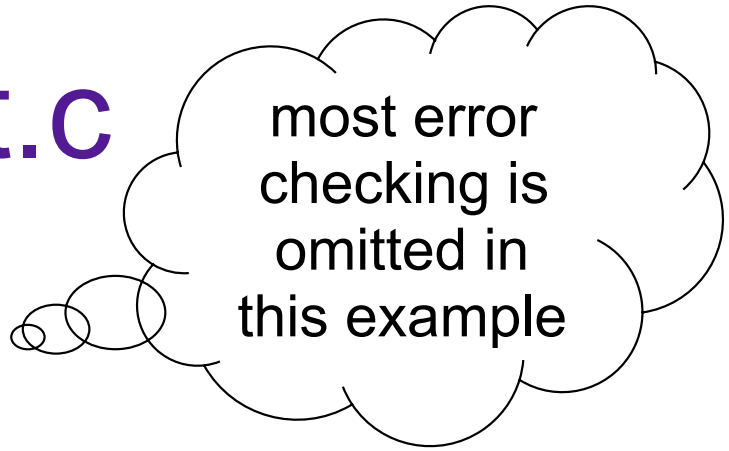
# Client side

- `socket()` – same as server, to say "how" we are going to talk

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

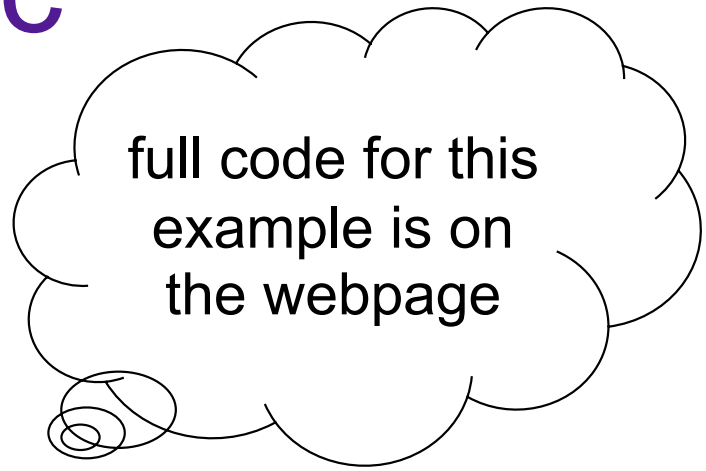
- the kernel will choose a dynamic port and source IP address.
- returns 0 on success and -1 on failure setting errno.
- initiates the three-way handshake.

# inetclient.c




```
int soc;
struct hostent *hp;
struct sockaddr_in peer;


peer.sin_family = AF_INET;
peer.sin_port = htons(PORT);
/* fill in peer address */
hp = gethostbyname(argv[1]);
peer.sin_addr = *((struct in_addr *)hp->h_addr);
/* create socket */
soc = socket(PF_INET, SOCK_STREAM, 0);
/* request connection to server */
if (connect(soc, (struct sockaddr *)&peer, sizeof(peer))
       == -1) {
  perror("client:connect"); close(soc); exit(1);
}
write(soc, "Hello Internet\n", 16);
read(soc, buf, sizeof(buf));
printf("SERVER SAID: %s\n", buf);
close(soc);
```

# inetserver.c




```
struct sockaddr_in peer;
struct sockaddr_in self;
int soc, ns, k;
int peer_len = sizeof(peer);


self.sin_family = PF_INET;
self.sin_port = htons(PORT);
self.sin_addr.s_addr = INADDR_ANY;
bzero(&(self.sin_zero), 8);


peer.sin_family = PF_INET;
/* set up listening socket soc */
soc = socket(PF_INET, SOCK_STREAM, 0);
if (soc < 0) {
  perror("server:socket"); exit(1);
}


if (bind(soc, (struct sockaddr *)&self, sizeof(self))== -1){
  perror("server:bind"); close(soc); exit(1);
}
listen(soc, 1);
...
```

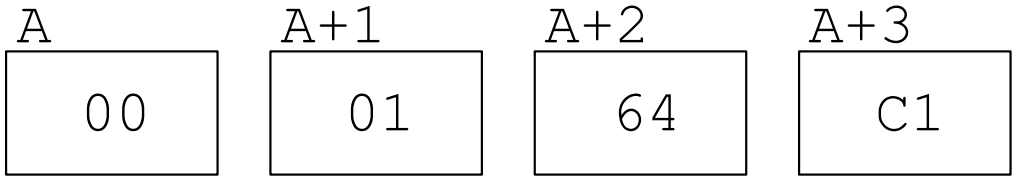
# inetserver.c (concluded)

```
/* ... repeated from previous slide ...
  soc = socket(PF_INET, SOCK_STREAM, 0);
  bind(soc, (struct sockaddr *)&self, sizeof(self))== -1){
    perror("server:bind"); close(soc); exit(1);
  }
  listen(soc, 1);
... and now continuing ... */

  /* accept connection request */
  ns = accept(soc, (struct sockaddr *)&peer, &peer_len);
  if (ns < 0) {
    perror("server:accept"); close(soc); exit(1);
  }
  /* data transfer on connected socket ns */
  k = read(ns, buf, sizeof(buf));
  printf("SERVER RECEIVED: %s\n", buf);
  write(ns, buf, k);


  close(ns);
  close(soc);
```

# Byte order

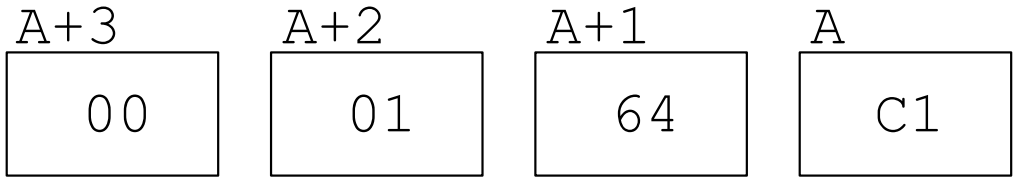
- Big-endian

91,329 =




- Little-endian

91,329 =




- Intel is little-endian, and Sparc is big-endian

data may need to order properly over different byte order machines in order to interpret them properly
Need to set up a network byte order to solve this problem

# Network byte order

array of char is same on differenrt byte order machine

- To communicate between machines with unknown or different “endian-ness” we convert numbers to network byte order (big-endian) before we send them.
- There are functions provided to do this:
  - `unsigned long htonl(unsigned long)`

host to network long/short

  - `unsigned short htons(unsigned short)`
  - `unsigned long ntohl(unsigned long)`

network to host long / short

  - `unsigned short ntohs(unsigned short)`

# Sending and Receiving Data

- `read` and `write` calls work on sockets, but sometimes we want more control
- `ssize_t send(int fd, const void *buf, size_t len, int flags);`
  - works like `write` if `flags==0`
  - flags: `MSG_OOB`, `MSG_DONTROUTE`, `MSG_DONTWAIT`
- `ssize_t recv(int fd, void *buf, size_t len, int flags);`
  - flags: `MSG_OOB`, `MSG_WAITALL`, `MSG_PEEK`