# Computer Networks

EDA387/DIT663

Socket API

Part 3

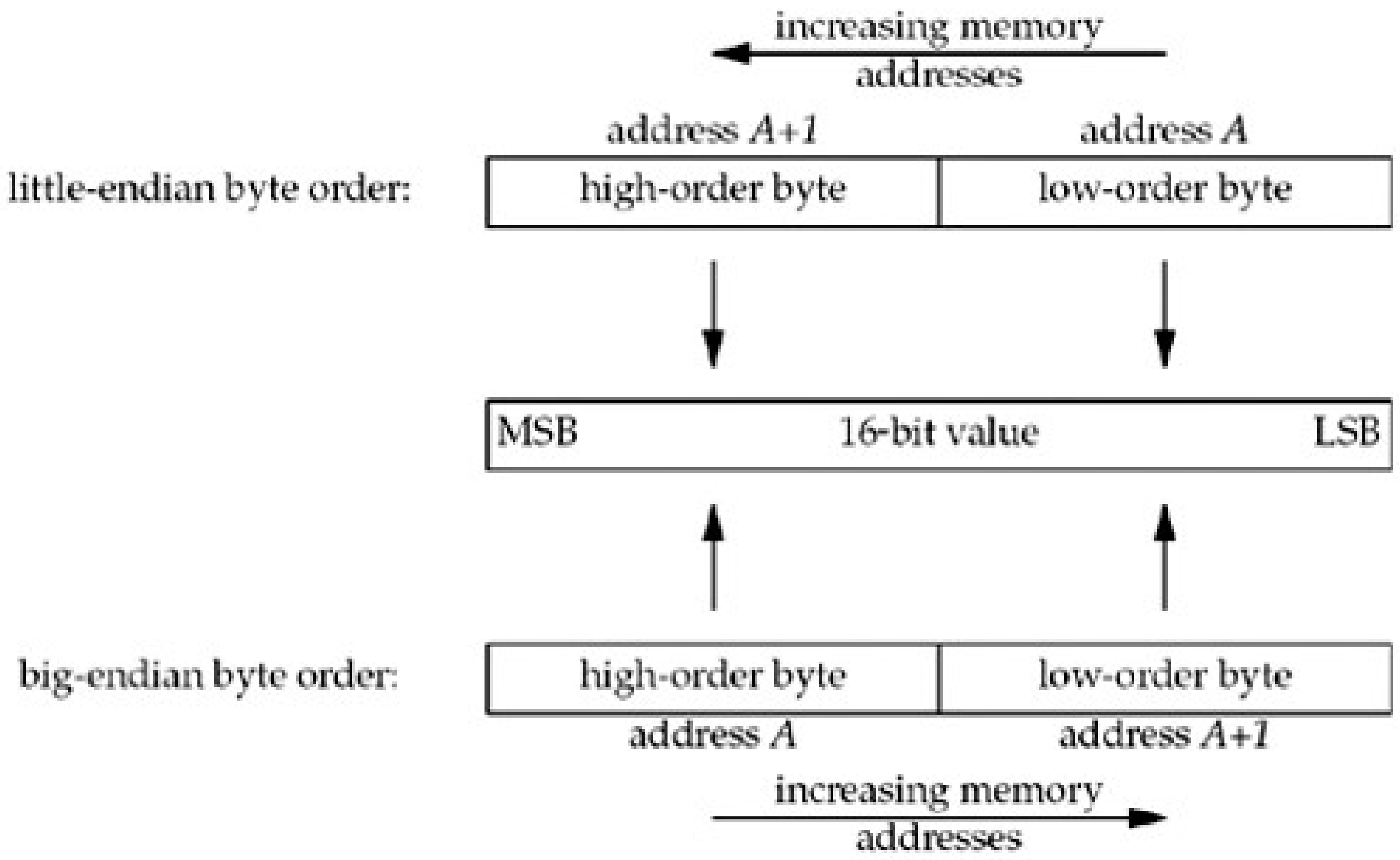
# Outline

- Byte Ordering (review on your own)
- IP addresses (review on your own)
- The Echo Server (fast)
- Elementary TCP Sockets
- Summary

# Byte Ordering

Self-study

- 16-bit integers are made up of 2 bytes
- 2 ways to store the 2 bytes in memory



## Outline

- Byte Ordering (review on your own)
- IP addresses (review on your own)
- The Echo Server
- Elementary TCP Sockets
- Summary

# Programmers' View

Self-study

- 1. Hosts are mapped to a set of 32-bit *IP addresses*.
  - 128.2.203.179
- 2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*
  - 128.2.203.179 is mapped to www.cs.cmu.edu
- 3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

# IP Addresses

Self-study

- 32-bit IP addresses are stored in an *IP address struct*
  - IP addresses are always stored in memory in network byte order (big-endian byte order)
  - True in general for any integer transferred in a packet header from one machine to another.
    - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

Handy network byte-order conversion functions:

| | |
|---|---|
| `htonl`: | convert `long int` from host to network byte order. |
| `htons`: | convert `short int` from host to network byte order. |
| `ntohl`: | convert `long int` from network to host byte order. |
| `ntohs`: | convert `short int` from network to host byte order. |

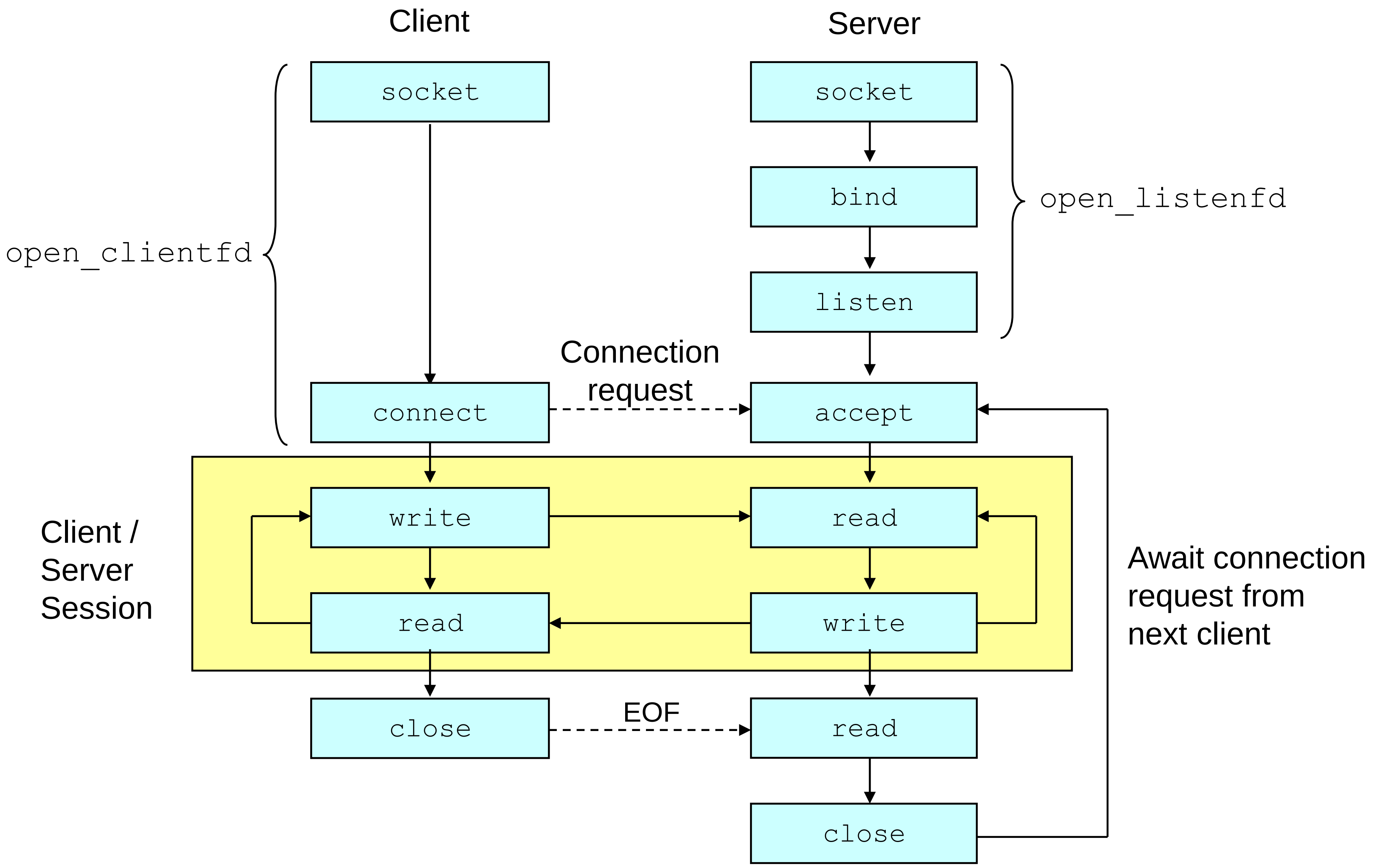
## Outline

- Byte Ordering (review on your own)
- IP addresses (review on your own)
- The Echo Server
- Elementary TCP Sockets
- Summary

## Elementary TCP Sockets

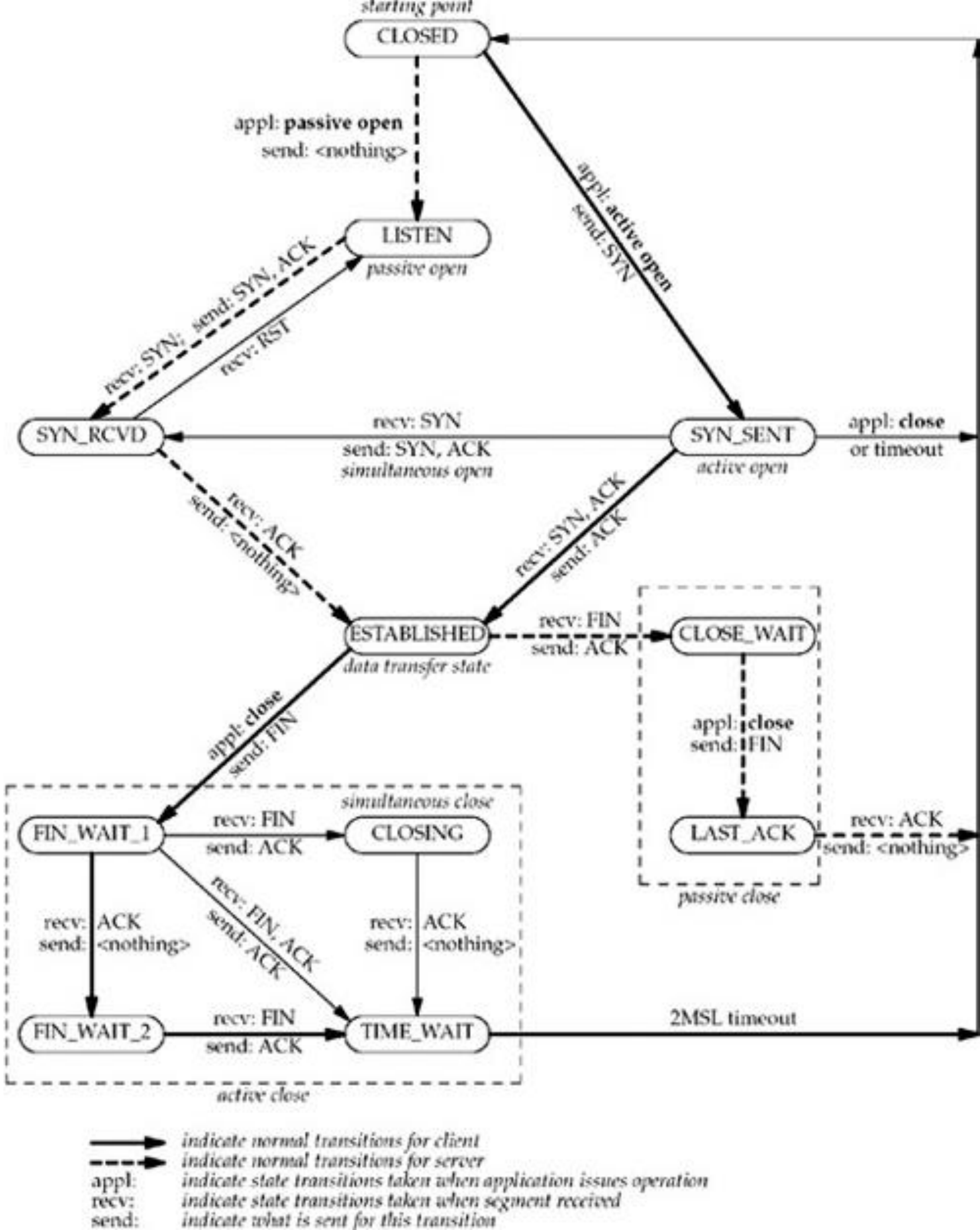
- We are now ready to describe the elementary socket API in more details by revising our TCP client and server programs
- We first consider each of the system calls and enhance our examples

# Overview of BSD Sockets Interface



# TCP State Transition

- CLOSED is the state in which a socket begins when it is created by the socket function



# `connect` **Function**

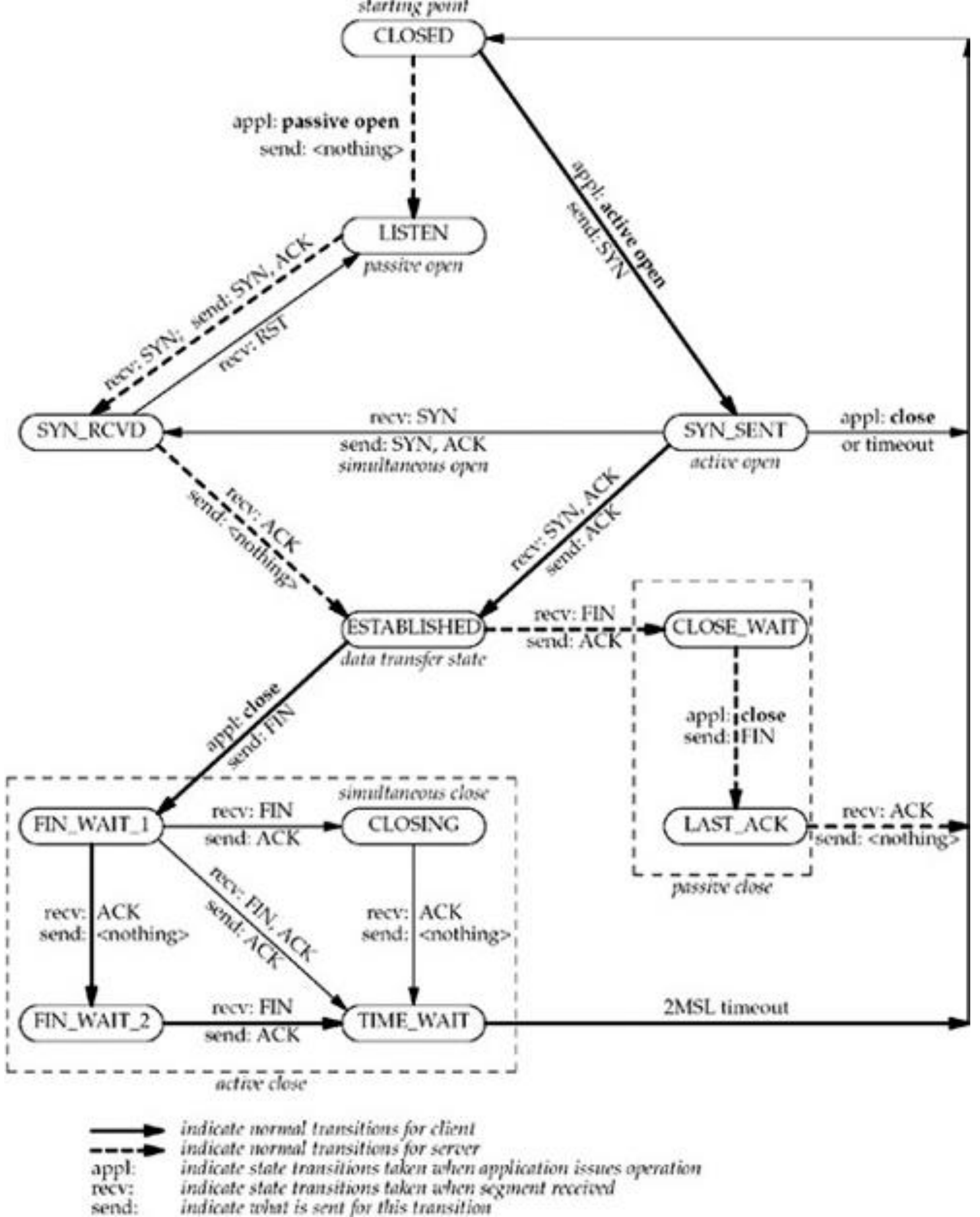
- TCP client establishes a connection with its server

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

- Client does not have to call bind before calling connect:
  - kernel will choose both an ephemeral port and the source IP address if necessary
- For TCP sockets, connect initiates TCP's three-way handshake and returns only when the connection is established or an error occurs

## TCP State Transition

- moves from CLOSED to SYN_SENT, and on success, to STABLISHED
- On failure, the socket is no longer usable (must close)
- Cannot call connect again on the socket



## `bind` **Function**

- assigns a local protocol address to a socket
  - 32-bit IPv4/128-bit IPv6 address and 16-bit TCP/UDP port
- `int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);`

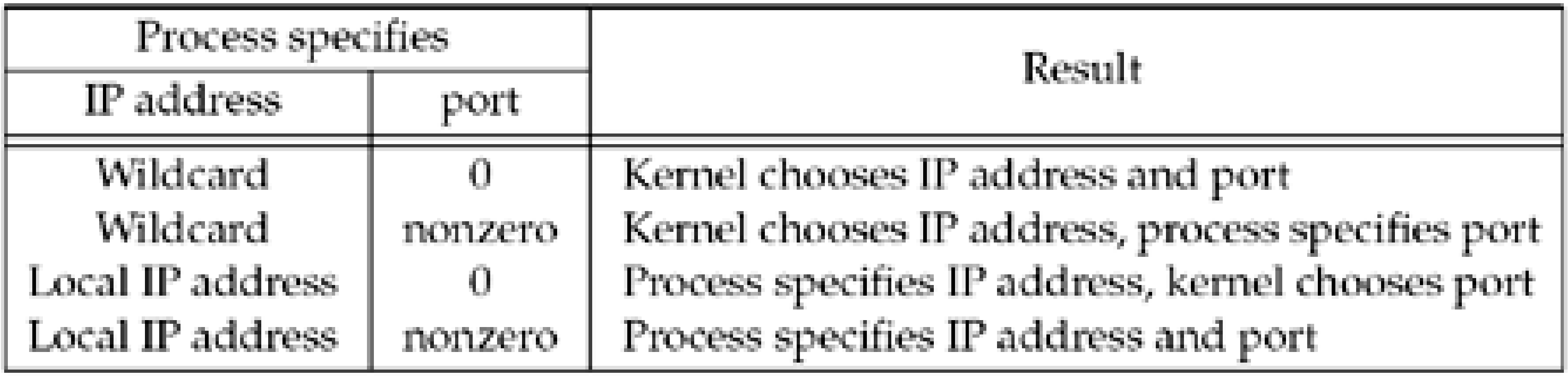
- Servers bind their well-known port when they start
  - kernel chooses ephemeral ports for the socket after calling connect or listen for unbounded sockets (client or server)

  - normal TCP clients rare for TCP servers

## `bind` **Function**

Self-study

- Result when specifying IP address and/or port number to bind



| Process specifies | | Result |
|---|---|---|
| IP address | port | |
| Wildcard | 0 | Kernel chooses IP address and port |
| Wildcard | nonzero | Kernel chooses IP address, process specifies port |
| Local IP address | 0 | Process specifies IP address, kernel chooses port |
| Local IP address | nonzero | Process specifies IP address and port |

- kernel chooses an ephemeral port when bind is called
- when specifying a wildcard IP address, the kernel does not choose the local IP address until the socket is connected

## `listen` **Function**

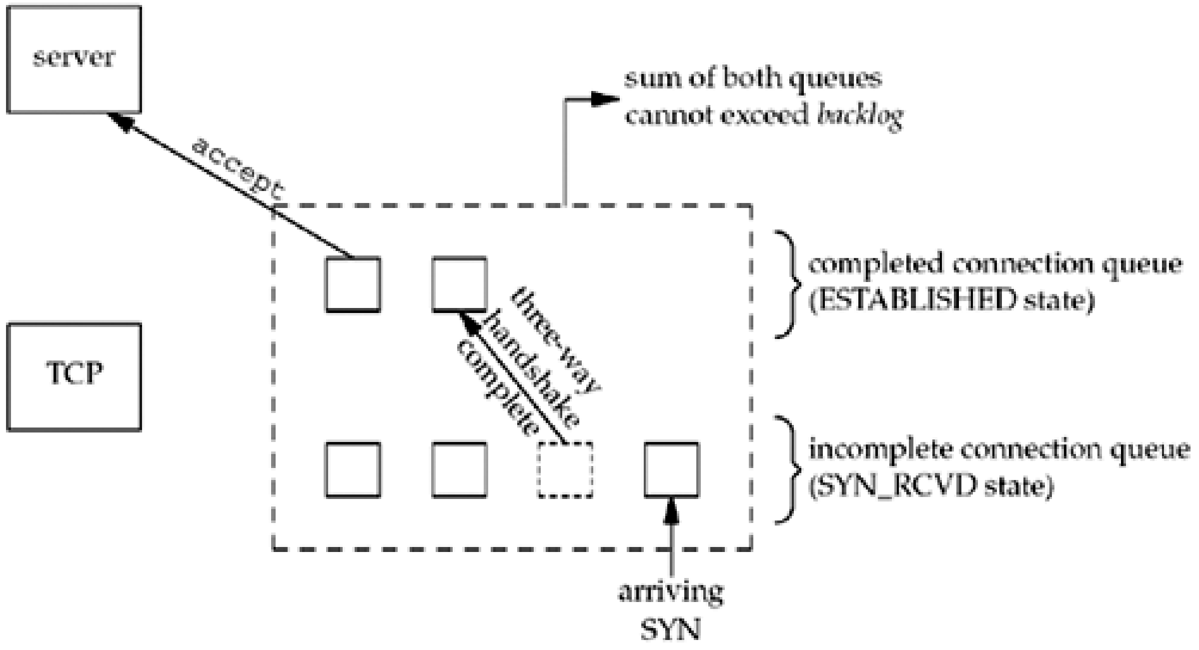
- Called only by a TCP server
  - Converts an unconnected socket, which is assume to be an active socket, into a passive socket
  - kernel should accept incoming connection requests directed to this socket
  - moves the socket from the CLOSED state to the LISTEN state
  - specifies the maximum number of connections the kernel should queue for this socket

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

- must be called before `accept`
  - normally called after both `socket` and `bind`

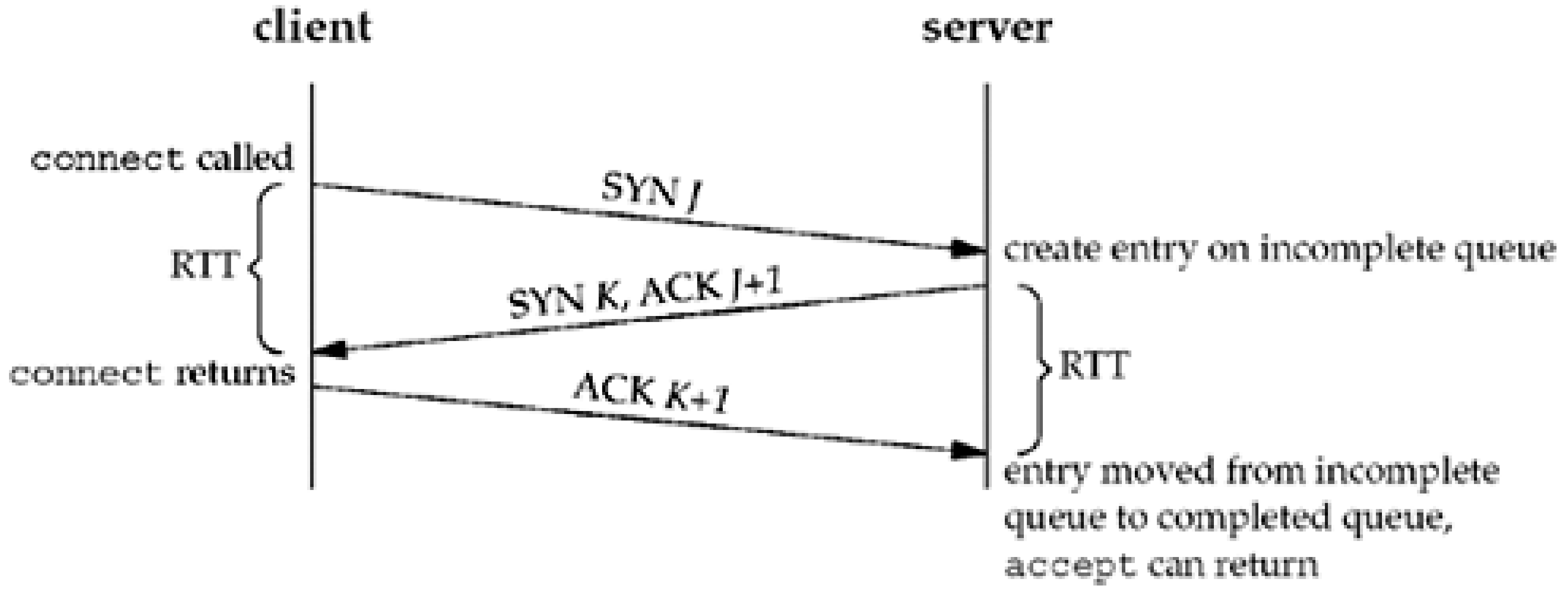
# `listen` Function

- Kernel maintains two queues:
  - Incomplete connection queue (SYN_RCVD): contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP 3-way handshake
  - Completed connection queue (ESTABLISHED): contains an entry for each client with whom the handshake has completed



# `listen` **Function**

- Once an entry is created on the incomplete queue, the listen socket's parameters are automatically copied over to the newly created connection
  - server process is not involved


client
server
connect called
SYN J
create entry on incomplete queue
RTT
SYN K, ACK J+1
connect returns
RTT
ACK K+1
entry moved from incomplete queue to completed queue, accept can return

## `accept` **Function**

- returns to the server the next completed connection from the front of the completed connection queue
  - If that queue is empty, the process is put to sleep (by default)

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

  - `cliaddr` and `addrlen` arguments are used to return the protocol address of the connected peer process
  - `addrlen` is a value-result argument: Before the call, `*addrlen` is the size of the structure pointed to by `cliaddr`; on return, contains the actual number of bytes stored by the kernel in the socket address structure

# Connected vs. Listening Descriptors

- Listening descriptor
  - End point for client connection requests.
  - Created once and exists for lifetime of the server.
- Connected descriptor
  - End point of the connection between client and server.
  - A new descriptor is created each time the server accepts a connection request from a client.
  - Exists only as long as it takes to service client.
- Why the distinction?
  - Allows for concurrent servers that can communicate over many client connections simultaneously.
    - E.g., Each time we receive a new request, we fork a child to handle the request.

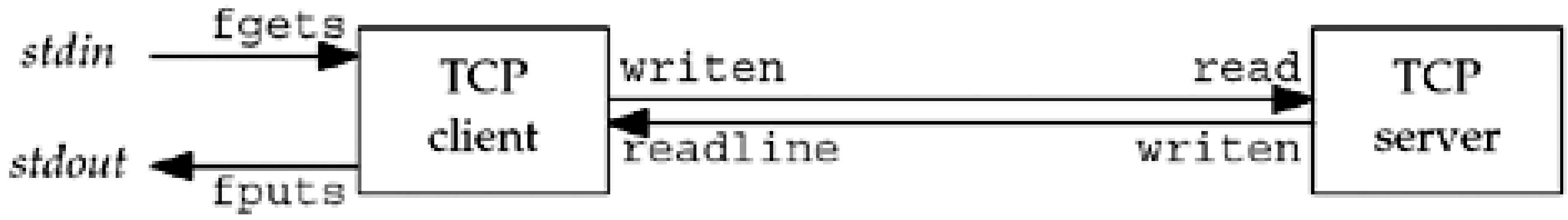
## `close` Function

- closes a socket and terminates the TCP connection

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

- mark the socket as closed and immediately returns
  - cannot be used to read or write
  - TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place
  - SO_LINGER option lets us change this default action

## Simple echo Client and Server



# Example: Echo Client and Server

## On Server

```
hive> echoserver 5000
server established connection with HIVE (128.252.21.14)
server received 4 bytes: 123
server established connection with HIVE (128.252.21.14)
server received 7 bytes: 456789
...
```

## On Client

```
hive> echoclient hive 5000
Please enter msg: 123
Echo from server: 123

hive> echoclient hive 5000
Please enter msg: 456789
Echo from server: 456789
hive>
```

## Echo Server

```
int main(int argc, char **argv) {

   int listenfd, connfd; struct sockaddr_in servaddr; char buff[MAXLINE];

   listenfd = Socket(AF_INET, SOCK_STREAM, 0);

   bzeros(&servaddr, sizeof(servaddr)); servaddr.sin_family = AF_INET;

   servaddr.sin_addr.s_addr = htonl(INADDR_ANY); servaddr.sin_port = htons(7);

  Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

  Listen(listenfd, LISTENQ);
```

## Echo Server

```
for ( ; ; ) {

    clilen = sizeof(cliaddr);

    connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

    str_echo(connfd);

    Close(connfd);

 }

}
```

# str_echo

```
void str_echo(int sockfd){

ssize_t n; char buf[MAXLINE];

again:

 while ( (n = read(sockfd, buf, MAXLINE)) > 0)

  Writen(sockfd, buf, n);

  if (n < 0 && errno == EINTR) goto again;

  else if (n < 0) err_sys("str_echo: read error");

}
```

# Echo Client

```
int main(int argc, char **argv){

int sockfd; struct sockaddr_in servaddr;

 if (argc != 2) err_quit("usage: tcpcli <IPaddress>");

 sockfd = Socket(AF_INET, SOCK_STREAM, 0); bzero(&servaddr, sizeof(servaddr));

 servaddr.sin_family = AF_INET;  servaddr.sin_port = htons(SERV_PORT);
     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

 Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

 str_cli(stdin, sockfd);     /* do it all */

 exit(0);

}
```

## str_cli

```
void str_cli(FILE *fp, int sockfd){

 char    sendline[MAXLINE], recvline[MAXLINE];

 while (Fgets(sendline, MAXLINE, fp) != NULL) {

   Writen(sockfd, sendline, strlen (sendline));

   if (Readline(sockfd, recvline, MAXLINE) == 0)

     err_quit("str_cli: server terminated prematurely");

   Fputs(recvline, stdout);

  }

}
```

# Outline

- Byte Ordering (review on your own)
- IP addresses (review on your own)
- The Echo Server
- Elementary TCP Sockets
- Summary

## Summary

- We have reviewed byte ordering issues
- We saw how it all works together in the echo server

## Next Lecture

*We are going to look into the way that IP addresses are encoded and review our knowledge about byte ordering issues.*

*Then we will look at the socket box, before learning more of what is inside the box and how the socket API interacts with the kernel.*

*Moreover, we are going to discover more about the limitations of iterative servers.*

## Review Questions

1. Find out what is the bye ordering of at least two computers that were manufactured by different vendors and use two different operating systems
2. A server and a client communicate via TCP. The server sends data to the client using the following code

# Review Questions

```
while( 1 ) {
     long measuredData = read_data_from_sensor();
  ssize_t  ret  =  send(  clientSocket,  &measuredData,
  sizeof(long), 0);
  if( -1 == ret ) handle_send_error();
}
```

## Review Questions

The client uses the following code to receive the data:

```
while( 1 ) {
  long receivedData;
  ssize_t ret = recv( serverSocket, &receivedData,
  sizeof(long), 0 );
  if( -1 == ret ) handle_recv_error();
  if( 0 == ret ) handle_disconnect();
  printf( "Received value: %ld\n", receivedData );
}
```

## Review Questions

- This system worked perfectly when both client and server were hosted on identical little-endian 32-bit machines. On these machines, the `long' data type was 32 bits. If the server read the value `12648430` (= `0xC0FFEE` in hex) from the sensor, the client would print

```
value = C0FFEE
```

- Recently, however, the server was upgraded to a 64-bit little endian machine, which caused the `long' data type to become 64-bits on the server only.

## Review Questions

- Assuming the server reads the same value `12648430` from the sensor, what is the output from the client?

(a) "`value = 0`"

(b) "`value = C0FFEE`"

(c) "`value = 0`" followed by "`value = C0FFEE`"

(d) "`value = C0FFEE`" followed by "`value = 0`"

## Review Questions

3. What happen when a server tries to bind to port number 0. How could you know which is the actually port number that is used? Explain why. Also, write the server code and show the output of a typical execution.

4. Consider a host that provides Web servers to multiple organizations. Each organization has its own domain name, such as www.organization.com. Moreover, each organization's domain name maps into a different IP address, but on the same subnet.

## Review Questions

For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All these IP addresses are then aliased onto a single network interface so that the IP layer will accept incoming datagrams destined for any of the aliased addresses.

Explain two alternatives in calling the bind system call in the above scenario. Choose the design alternative that is preferable to you and explain why do you prefer it.