

# Socket Programming

# Socket Programming

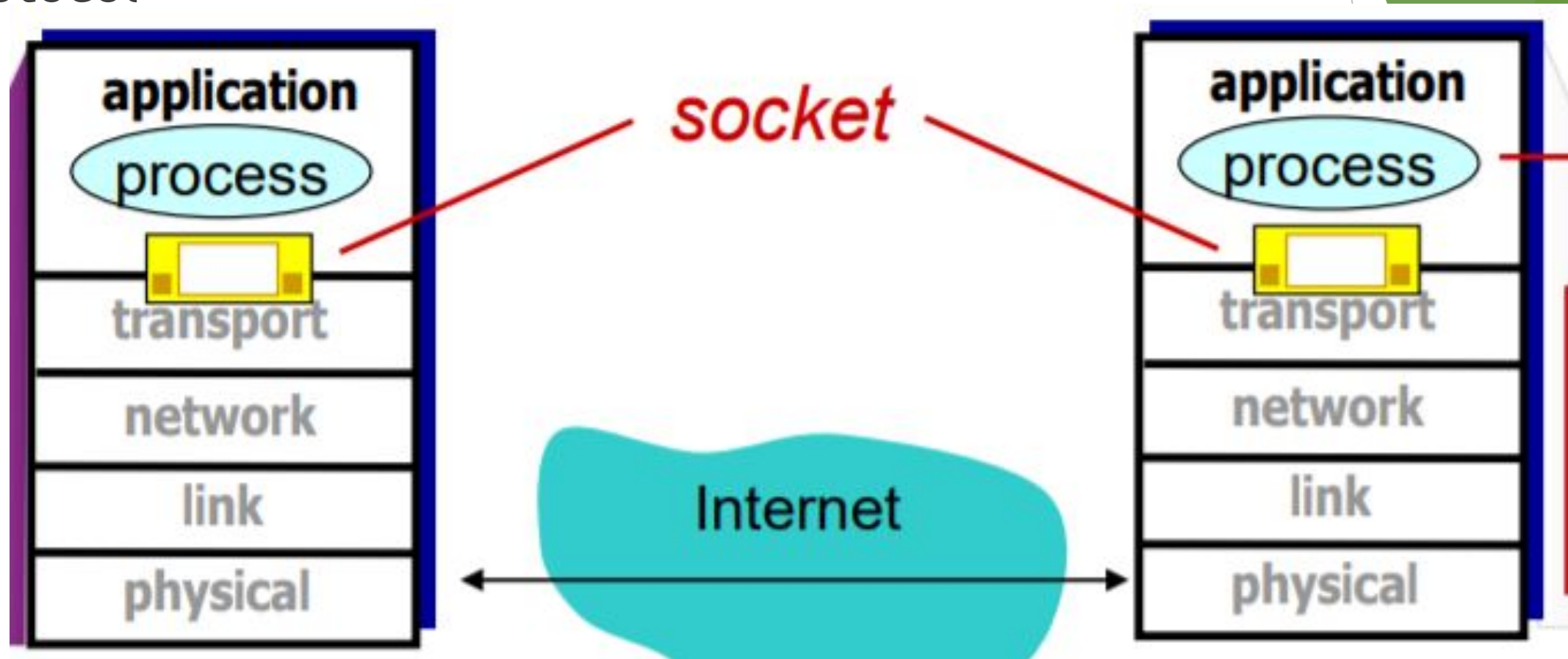
- ▶ **Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.**

# Socket Programming

- ▶ **End to end programming (chatting, file transfer, calculation , data processing etc.)**
- ▶ **process sends/receives messages to/from its socket**
- ▶ **A socket can be uniquely identified by**
  - ▶ **An Internet Address**
  - ▶ **An End to End Protocol (TCP or UDP)**
  - ▶ **A Port Number**

# Socket Programming

- socket: door between application process and end-end- transport protocol



# Socket Programming

- ▶ **Two types of (TCP/IP) sockets**
  - ▶ **Stream sockets (e.g. uses TCP)**
    - ▶ **provide reliable byte-stream service**
- ▶ **Datagram sockets (e.g. uses UDP)**
  - ▶ **provide best-effort datagram service**
  - ▶ **messages up to 65,500 bytes**

# Client-Server Communication

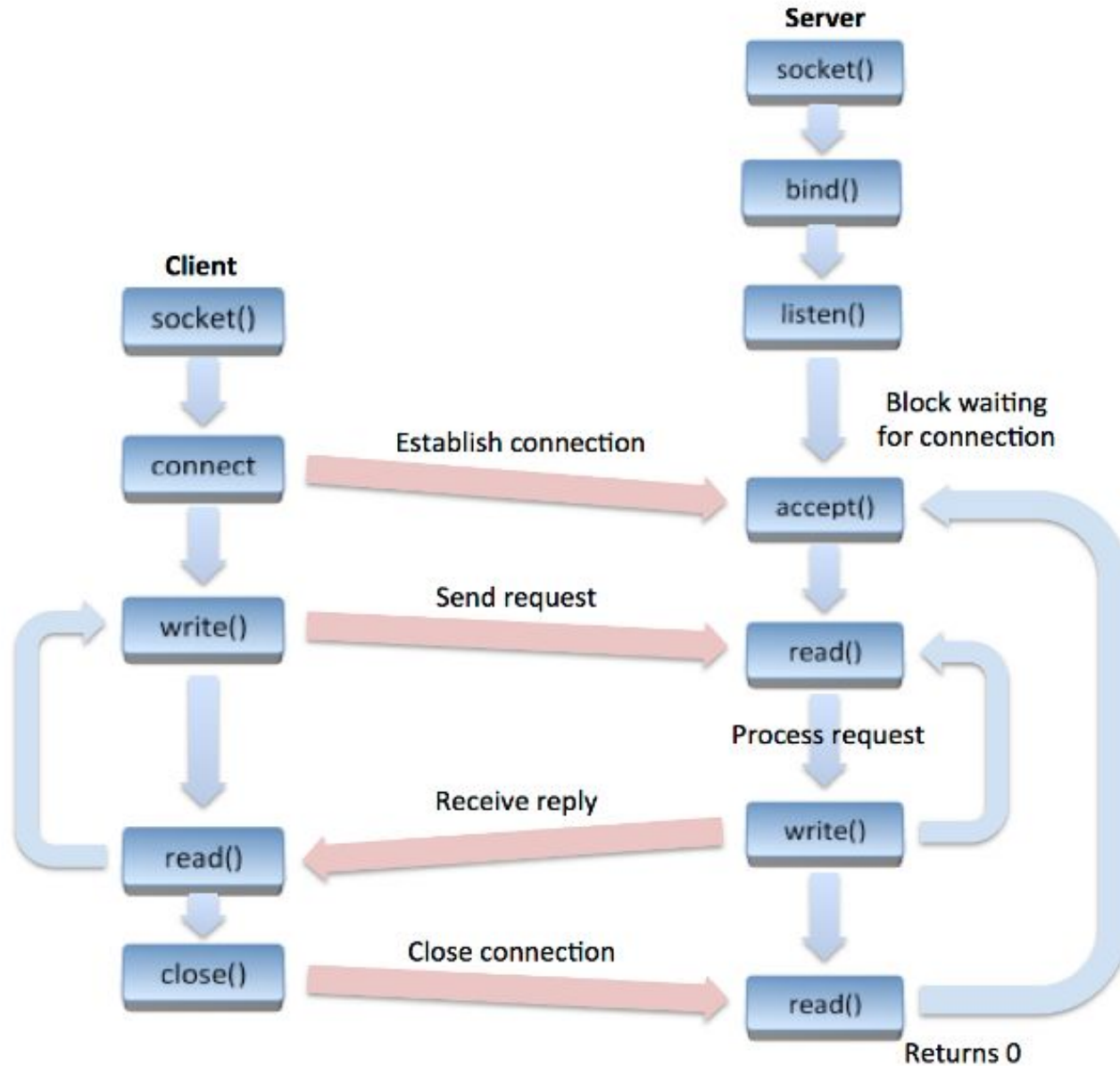
## ► Server

- passively waits for and responds to clients
- passive socket

## ► Client

- initiates the communication
- must know the address and the port of the server
- active socket

# Socket Programming TCP



abc

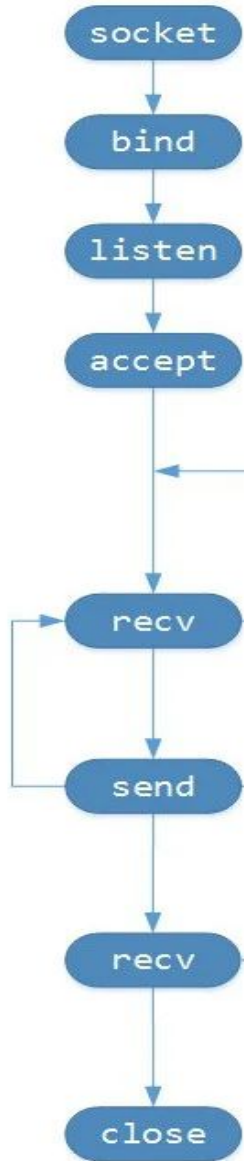
# Socket Programming TCP

- ▶ **client must contact server**
  - ▶ server process must first be running
  - ▶ server must have created socket (door) that welcomes client's contact
- ▶ **client contacts server by:**
  - ▶ Creating TCP socket, specifying IP address, port number of server process.
- ▶ **when client creates socket:** client TCP establishes connection to server TCP
- ▶ **when contacted by client,** server TCP creates new socket for server process to communicate with that particular client
  - ▶ allows server to talk with multiple clients
  - ▶ source port numbers used to distinguish clients



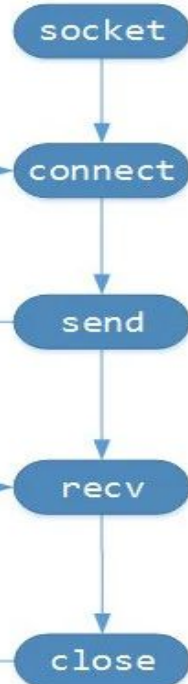
# Socket Programming TCP

## Server



Server creating listening socket

## Client



Establishing connection,  
three-way handshake

Client sending data,  
server receiving data

Server sending data,  
client receiving data

Client sending close message

# Socket Programming TCP

**Socket:** Create a new communication endpoint

**Bind:** Attach a local address to a socket

**Listen:** Announce willingness to accept connections

**Accept:** Block caller until a connection request arrives

**Connect:** Actively attempt to establish a connection

**Send:** Send some data over the connection

**Receive:** Receive some data over the connection

Read/write

**Close:** Release the connection

# Socket Programming TCP

Clients and servers communicate with each other by reading from and writing to socket descriptors.

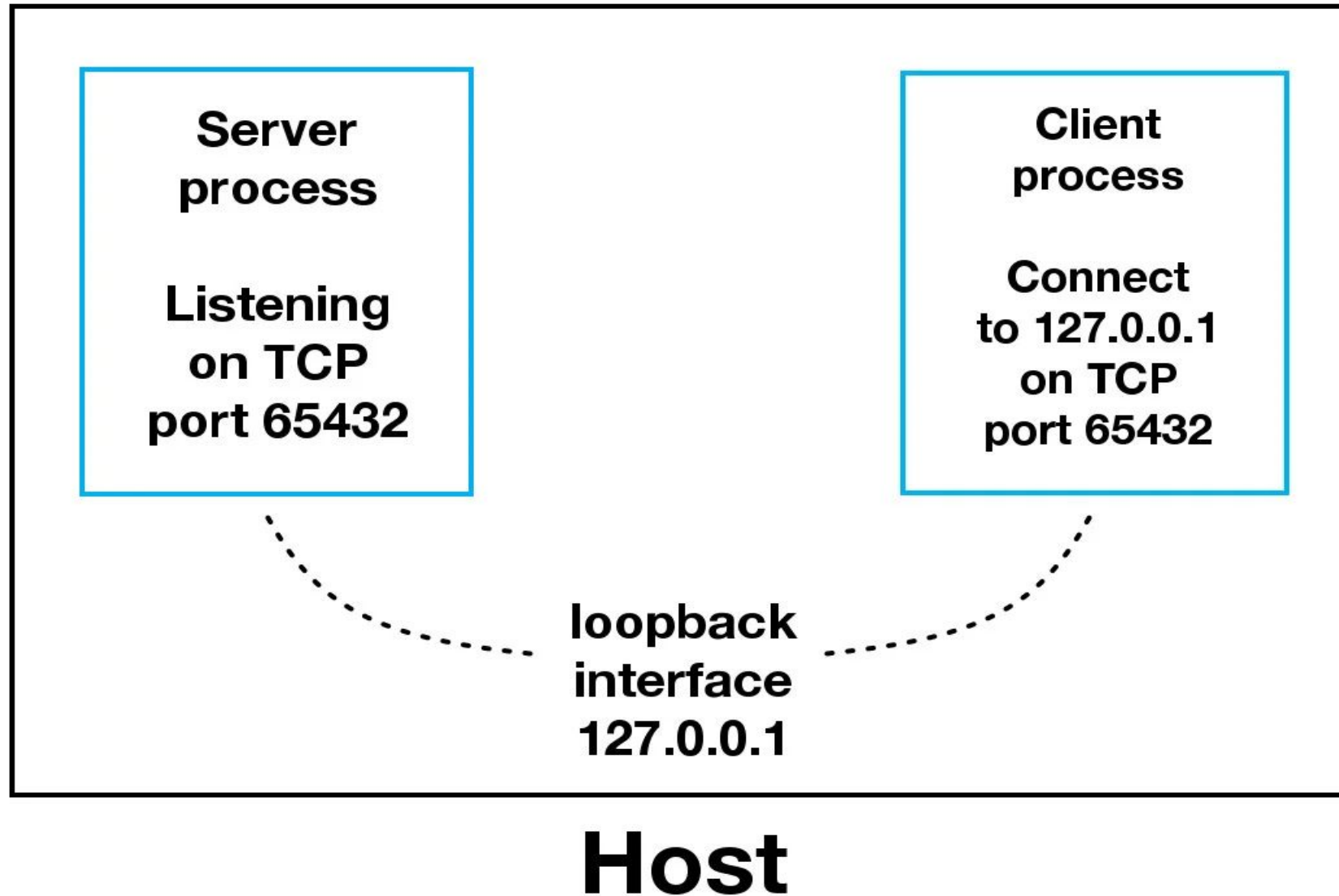
Server creates a socket and binds address/port

- Server creates a socket, just like the client does
- Server associates the socket with the port number

Create a socket -

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

# Socket Programming TCP



# Socket Programming TCP

**/usr/include/netinet/in.h**

Defines Internet constants and structures.

**/usr/include/arpa/nameser.h**

Contains Internet name server information.

**/usr/include/netdb.h**

Contains data definitions for socket subroutines.

**/usr/include/resolv.h**

Contains resolver global definitions and variables.


**/usr/include/sys/socket.h**

Contains data definitions and socket structures.

**/usr/include/sys/types.h**

Contains data type definitions.

# Socket Programming TCP

- ▶ `Int sockfd`      `#socket Descriptor`
- ▶ `Int newsockfd`      `#after successful connection we will assign it as socket fd so that a server can handle multiple client.`
- ▶ `ip`      `"127.0.0.1" / gethostbyname(argv[1]); {character pointer}`
- ▶ `Port`      `"<1024" / atoi(argv[2]); {int}`
- ▶ `struct`
  - ▶ `sockaddr_in`, : **Socket /server binding**
  - ▶ `serv_addr`, : **address specifications :**
  - ▶ `cli_addr`; after succesfull connections client address will be linked with it. 

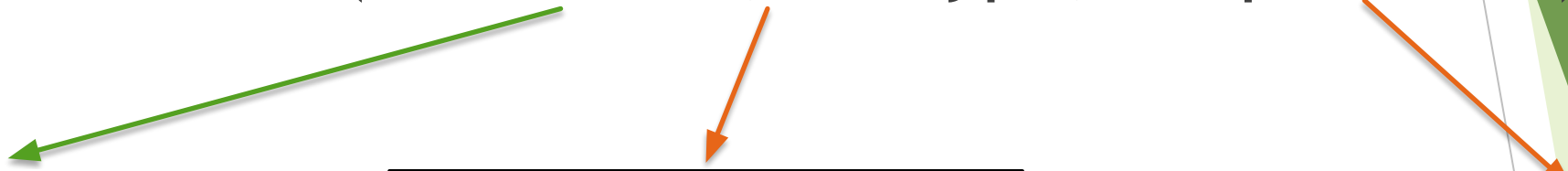
The `<netinet/in.h>` header shall define the **`sockaddr_in`** structure that includes at least the following members:

<code>sa_family_t</code>	<code>sin_family</code>	<code>AF_INET.</code>
<code>in_port_t</code>	<code>sin_port</code>	Port number.
<code>struct in_addr</code>	<code>sin_addr</code>	IP address.

# Socket Programming TCP

Create a socket -

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



Communication Domain  
:Address family  
AF\_INET=> IPV4

Sock\_Stream -> TCP  
Sock\_DGRAM -> UDP

For TCP it is 0 by default

sockfd => socket(int domain, int type, int protocol )  
is the usual socket file descriptor from the socket() system call

# Socket Programming TCP

Bind socket to the local address and port number -

```
int bind(int sockfd, struct sockaddr *my_addr, sizeof(serv_addr)  
)
```

File descriptor

Assign address specified by my\_addr  
Which is second parameter

Specify the size in Byte of address  
structure provided by addr

Bind function return 0 for successful binding  
-1 for failure



# Socket Programming TCP

Listen function:

```
#include <sys/socket.h>
```

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

`sockfd` => is the usual socket file descriptor from the `socket()` system call

`queue_size` => maximum size of the queue of pending (incomplete) connections

# Socket Programming TCP

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

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

- ▶ `addr` will usually be a pointer to a local struct `sockaddr_in`. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port).
- ▶ `addrlen` is a local integer variable that should be set to `sizeof(struct sockaddr_in)` before its address is passed to `accept()`. `Accept` will not put more than that many bytes into `addr`. If it puts fewer in, it'll change the value of `addrlen` to reflect that.

# Socket Programming TCP

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

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

```
sizeof(serv_addr)
```

# Socket Programming TCP

The `send()` call:

```
int send(int sockfd, const void *msg, int len, int flags);
```

`msg` is a pointer to the data you want to send, and `len` is the length of that data in bytes.

The `recv()` call is similar in many respects:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

`buf` is the buffer to read the information into,

`len` is the maximum length of the buffer, and `flags` can again be set to 0.

`recv()` returns the number of bytes actually read into the buffer, or -1 on error (with `errno` set, accordingly.)

Wait! `recv()` can return 0. This can mean only one thing: the remote side has closed the connection.

# Socket Programming TCP

The read() call:

```
int read(int sockfd, string buff, int len);
```

msg is a pointer to the data you want to send, and len is the length of that data in bytes.

```
n = read(sockfd,buffer,255);
```

The write() call is similar in many respects:

```
int write(int sockfd, string buff, int len);
```

buf is the buffer to read the information into,

```
n = write(sockfd,buffer,strlen(buffer));
```

# Socket Programming TCP

```
write(sockfd, &numbervar, sizeof(int));  
read(newsockfd, & numbervar, sizeof(int));
```

```
read(sockfd,buffer,255);  
write(sockfd,buffer,strlen(buffer));
```

```
int send(int sockfd, const void *msg, int len, int flags);  
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

# Socket Programming TCP

`close()` => just to close the connection

`close(sockfd)`

`close(newsockfd)`

# Socket Programming TCP

## Server Side:

Create socket > bind() > listen() > accept & newsocket > *(read/write)/(send/receive)*  
> *fulfill the operation/request/function* > *(read/write)/(send/receive)* > close

## Client Side:

Create socket > connect > *(read/write)/(send/receive)* > *fulfill the operation/request/function* > *(read/write)/(send/receive)* > close

<https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html>



# Socket Programming TCP

## Server Side:

Create socket > bind() > listen() > accept & newsocket > *(read/write)/(send/receive)*  
> *fulfill the operation/request/function* > *(read/write)/(send/receive)* > close

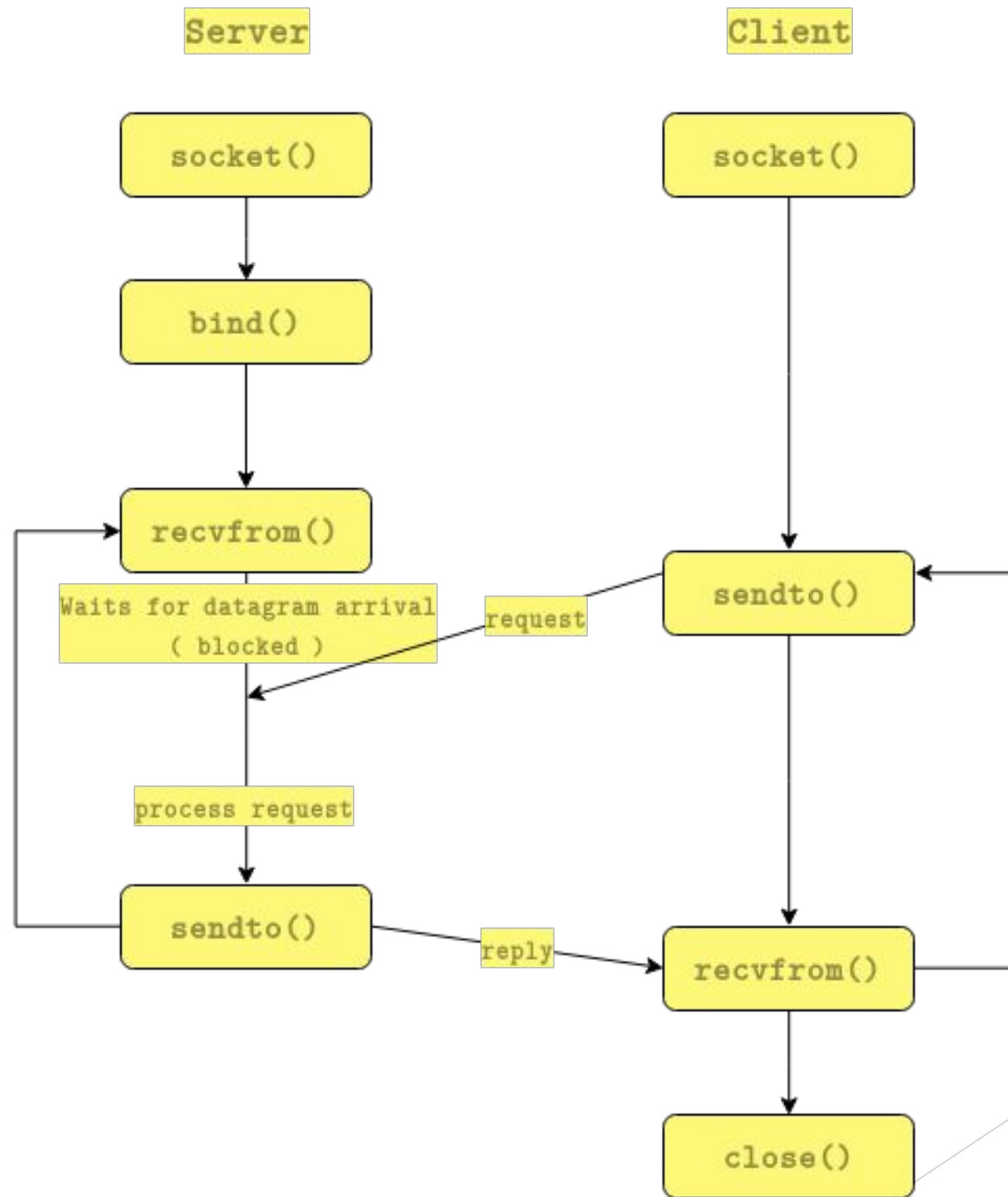
## Client Side:

Create socket > connect > *(read/write)/(send/receive)* > *fulfill the operation/request/function* > *(read/write)/(send/receive)* > close

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect. The shapes are concentrated on the left and right sides of the frame, leaving a large white central area.

UDP

# UDP



# UDP

- ▶ `sockfd = socket(AF_INET, SOCK_DGRAM, 0);`
- ▶ `DatagramSocket serverSocket = new DatagramSocket(5555);`

# Server side programming:



# Evaluation

<https://meet.google.com/lookup/cvosmsi6sf>