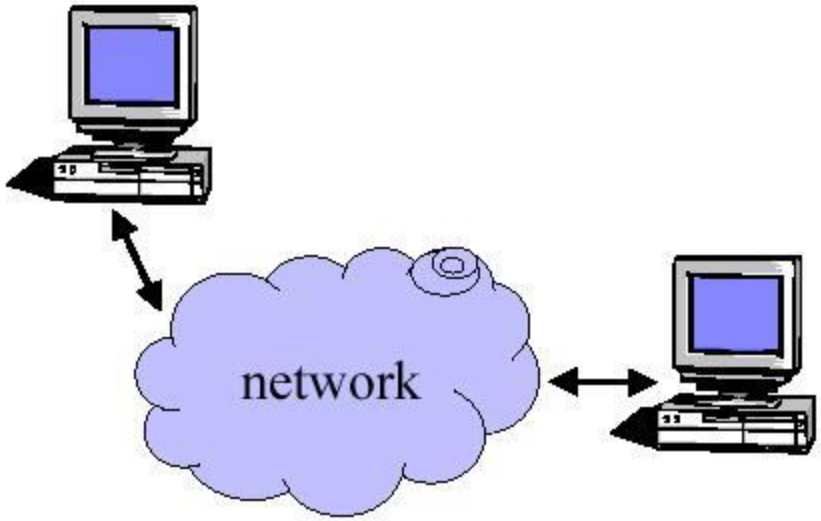
Introduction to Sockets

Jan 2015

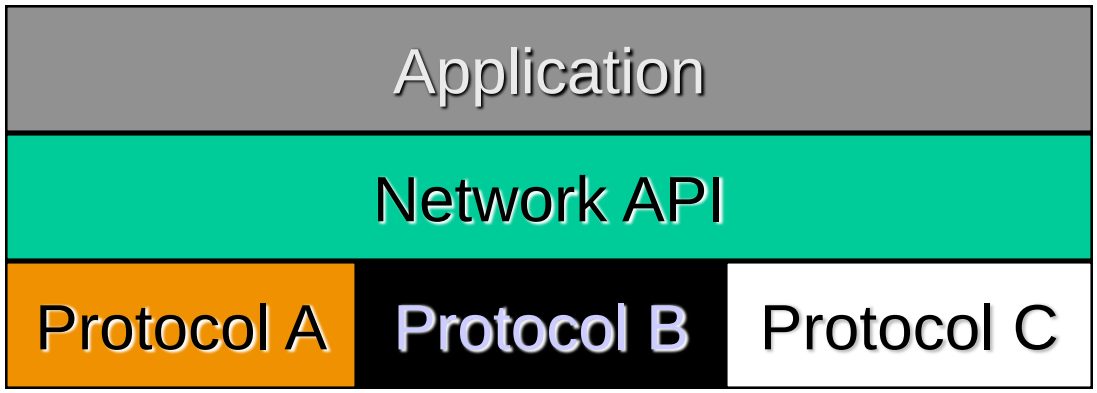
Why do we need sockets?

Provides an abstraction for interprocess communication



Definition

- The services provided (often by the operating system) that provide the interface between application and protocol software.



Functions

- Define an “end- point” for communication
- Initiate and accept a connection
- Send and receive data
- Terminate a connection gracefully

Examples

- File transfer apps (FTP), Web browsers
- (HTTP), Email (SMTP/ POP3), etc…

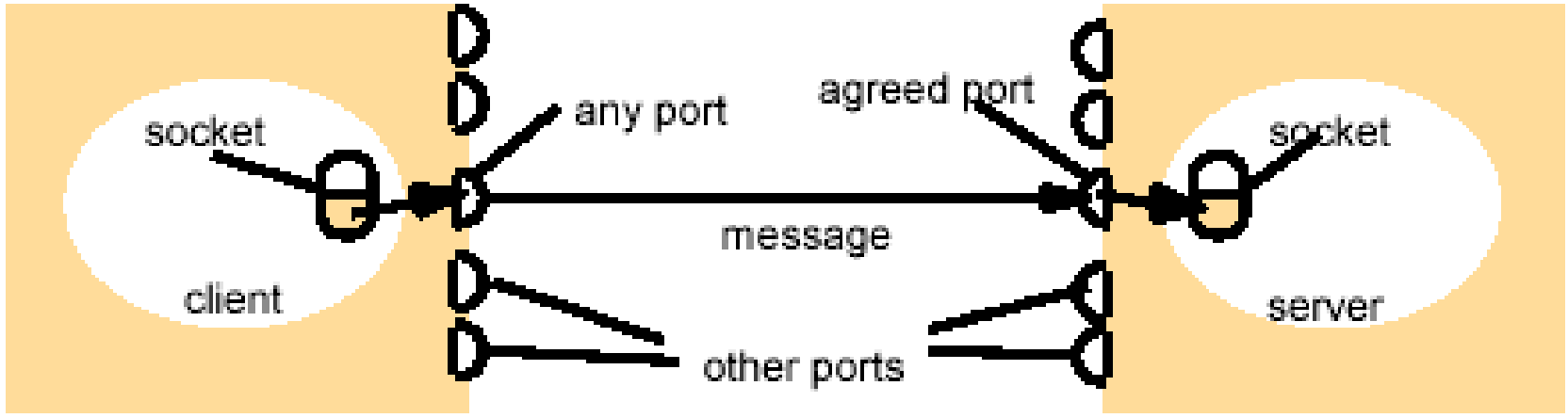
Types of Sockets

- Two different types of sockets :
  - stream vs. datagram

- Stream socket :(*a. k.* a. connection- oriented socket)
  - It provides reliable, connected networking service
  - Error free; no out- of- order packets (uses TCP)
  - applications: telnet/ ssh, http, …

- Datagram socket :(*a. k.* a. connectionless socket)
  - It provides unreliable, best- effort networking service
  - Packets may be lost; may arrive out of order (uses UDP)
  - applications: streaming audio/ video (realplayer), …

Addressing

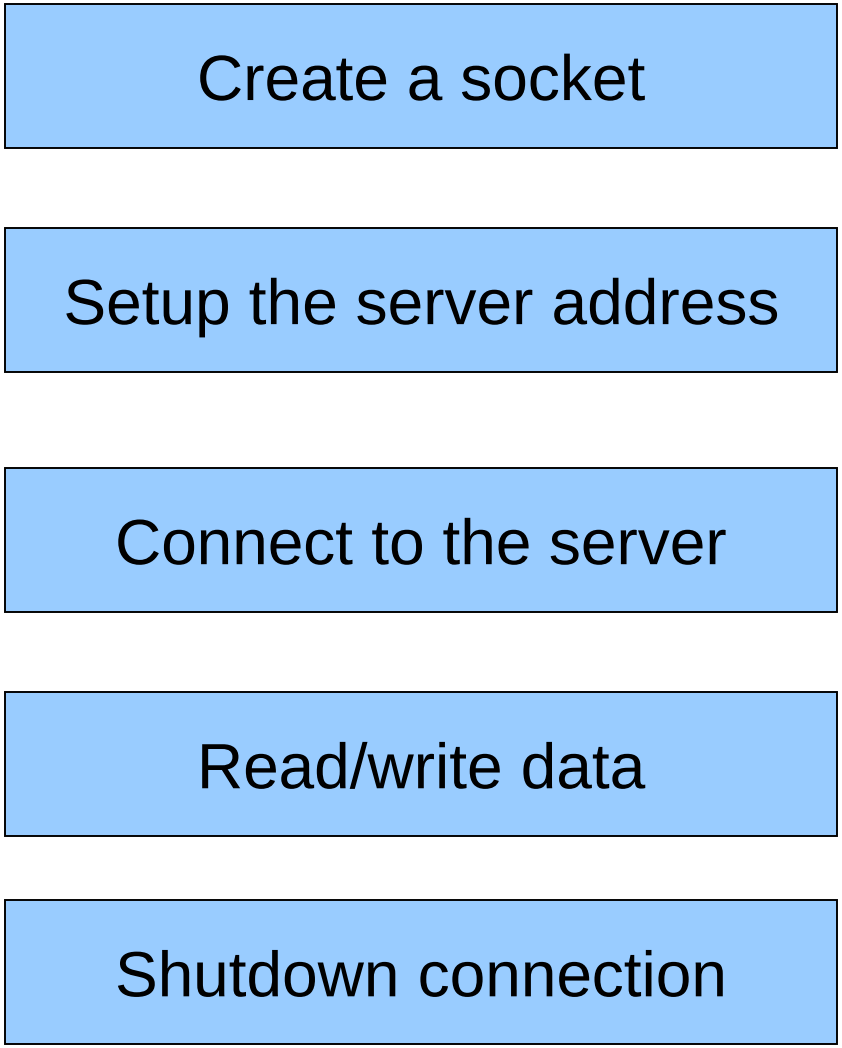


Client ⟷ Server

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox

Client – high level view



```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SOCK_ STREAM, 0);
        if (sock == -1)
                return sock;
        host = gethostbyname( hostname);
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, S
        if (sock == -1)
                return sock;
        host = gethostbyname( hos
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

```
Ipv4 socket address structure
struct socketaddr_in{
  uint8_t          sin_len;    /*length of the structure (16)*/
  sa_falimily_t    sin_family /* AF_INT*/
  in_port_t        sin_port    /* 16 bit TCP or UDP port number*/
  struct in_addr   sin_addr    /* 32 bit Ipv4 address */
  char             sin_zero(8)/* unused*/
}
```

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SO
        if (sock == -1)
                return sock;
        host = gethostbyname( hostn
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

Hostent structure

```
struct hostent{
  char *    h_name        /*official name of host*/
  char **   h_aliases;    /* pointer ot array of\
                             pointers to alias name*/
  int       h_addrtype    /* host address type*/
  int       h_length      /* length of address */
  char **   h_addr_list   /*prt to array of ptrs with \
                            IPv4 or IPv6 address*/
}
```

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SOCK_ STREAM, 0);
        if (sock == -1)
                return sock;
        host = gethostbyname( hostnam
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

Make the socket

Socket(int family , int type, in t protocol);
return nonnegative value for OK, -1 for error

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SOCK_ STREAM, 0);
        if (sock == -1)
                return sock;
        host = gethostbyname( hostname);
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

Resolve the host

```
struct hostent *gethostbyname( const char *hostname);
/*Return nonnull pointer if OK, NULL on error */
```

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SOCK_ STREAM, 0);
        if (sock == -1)
                return sock;
        host = gethostbyname( hostname);
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

unit16_t htons(unit16_t host16bitvaule)
/*Change the port number from host byte order to network byte order */

Setup up the struct

```
int connect_ socket( char *hostname, int port) {
        int sock;
        struct sockaddr_in sin;
        struct hostent *host;
        sock = socket( AF_ INET, SOCK_ STREAM, 0);
        if (sock == -1)
                return sock;
        host = gethostbyname( hostname);
        if (host == NULL) {
                close( sock);
                return -1;
        }
        memset (& sin, 0, sizeof( sin));
        sin. sin_ family = AF_ INET;
        sin. sin_ port = htons( port);
        sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[ 0];
        if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
                close (sock);
                return -1;
        }
        return sock;
}
```

Connect

connect(int socketfd, const struct sockaddr * servaddr,
socket_t addrlen)
/*Perform the TCP three way handshaking*/

Server – high level view

Create a socket

Bind the socket

Listen for connections

Accept new client connections

Read/write to client connections

Shutdown connection

Listening on a port (TCP)

```
int make_ listen_ socket( int port) {
    struct sockaddr_ in sin;
    int sock;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock < 0)
        return -1;
    memset(& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ addr. s_ addr = htonl( INADDR_ ANY);
    sin. sin_ port = htons( port);
    if (bind( sock, (struct sockaddr *) &sin, sizeof( sin)) < 0)
        return -1;
    return sock;
}
```

Listening on a port (TCP)

```
int make_ listen_ socket( int port) {
    struct sockaddr_ in sin;
    int sock;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock < 0)
        return -1;
    memset(& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ addr. s_ addr = htonl( INADDR_ ANY);
    sin. sin_ port = htons( port);
    if (bind( sock, (struct sockaddr *) &sin, sizeof( sin)) < 0)
        return -1;
    return sock;
}
```

Make the socket

Listening on a port (TCP)

```
int make_ listen_ socket( int port) {
    struct sockaddr_ in sin;
    int sock;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock < 0)
        return -1;
    memset(& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ addr. s_ addr = htonl( INADDR_ ANY);
    sin. sin_ port = htons( port);
    if (bind( sock, (struct sockaddr *) &sin, sizeof( sin)) < 0)
        return -1;
    return sock;
}
```

Setup up the struct

Listening on a port (TCP)

```
int make_ listen_ socket( int port) {
    struct sockaddr_ in sin;
    int sock;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock < 0)
        return -1;
    memset(& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ addr. s_ addr = htonl( INADDR_ ANY);
    sin. sin_ port = htons( port);
    if (bind( sock, (struct sockaddr *) &sin, sizeof( sin)) < 0)
        return -1;
    return sock;
}
```

Bind

```
bind(int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);
/* return 0 if OK, -1 on error
        assigns a local protocol adress to a socket*/
```

accepting a client connection (TCP)

```
int get_ client_ socket( int listen_ socket) {
    struct sockaddr_ in sin;
    int sock;
    int sin_ len;
    memset(& sin, 0, sizeof( sin));
    sin_ len = sizeof( sin);
    sock = accept( listen_ socket, (struct sockaddr *) &sin, &sin_ len);
    return sock;
}
```

accepting a client connection (TCP)

```
int get_ client_ socket( int listen_ socket) {
    struct sockaddr_ in sin;
    int sock;
    int sin_ len;
    memset(& sin, 0, sizeof( sin));
    sin_ len = sizeof( sin);
    sock = accept( listen_ socket, (struct sockaddr *) &sin, &sin_ len);
    return sock;
}
```

Setup up the struct

accepting a client connection (TCP)

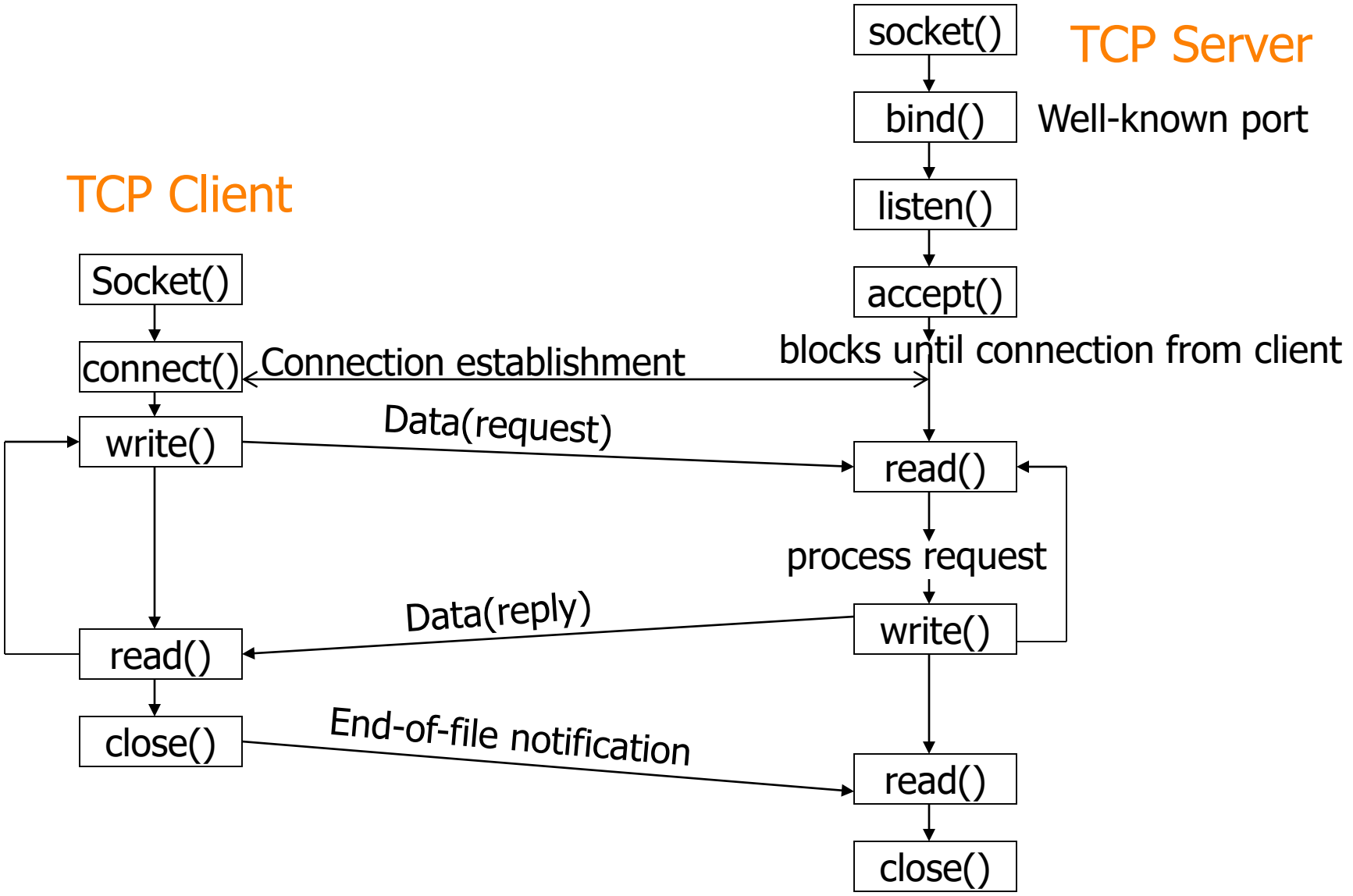
```
int get_ client_ socket( int listen_ socket) {
    struct sockaddr_ in sin;
    int sock;
    int sin_ len;
    memset(& sin, 0, sizeof( sin));
    sin_ len = sizeof( sin);
    sock = accept( listen_ socket, (struct sockaddr *) &sin, &sin_ len);
    return sock;
}
```

Accept the client connection

```
accept(int sockefd, struct sockaddr * claddr, socklen_t * addrlen)
/* return nonnegative descriptor if OK, -1 on error
    return the next completed connection from the front of the
     completed connection queue.
    if the queue is empty,
                the process is put to sleep(assuming blocking socket)*/
```

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]


TCP Server
socket()
bind()
Well-known port
listen()
accept()
blocks until connection from client
TCP Client
Socket()
connect()
Connection establishment
write()
Data(request)
read()
process request
write()
Data(reply)
read()
close()
End-of-file notification
read()
close()

Dealing with blocking calls

- Many functions block
  - accept(), connect(),
  - All recv()
- For simple programs this is fine
- What about complex connection routines
  - Multiple connections
  - Simultaneous sends and receives
  - Simultaneously doing non-networking processing

Dealing with blocking (cont..)

- Options
  - Create multi-process or multi-threaded code
  - Turn off blocking feature (*fcntl()* system call)
  - Use the *select()* function
- What does select() do?
  - Can be permanent blocking, time-limited blocking or non-blocking
  - Input: a set of file descriptors
  - Output: info on the file-descriptors' status
  - Therefore, can identify sockets that are "ready for use": calls involving that socket will return immediately

select function call

- int status = select()
  - Status: # of ready objects, -1 if error
  - nfds: 1 +largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - Timeout: time after which select returns