Socket Programming

Giulio Grassi
giulio.grassi@lip6.fr

Introduction

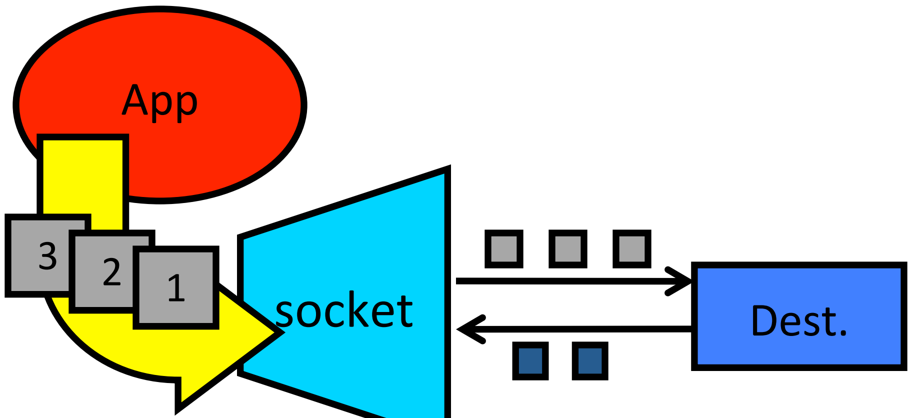
- Why Sockets?
  - Used for Interprocess communication
- What are Sockets?
  - End-point of interprocess communication.
  - An interface through which processes can send / receive information
- Once configured, the application can
  - pass data to the socket for network (or interprocess) transmission
  - receive data from the socket (transmitted through the network by some other host or by some process/thread running in the same machine)

Introduction

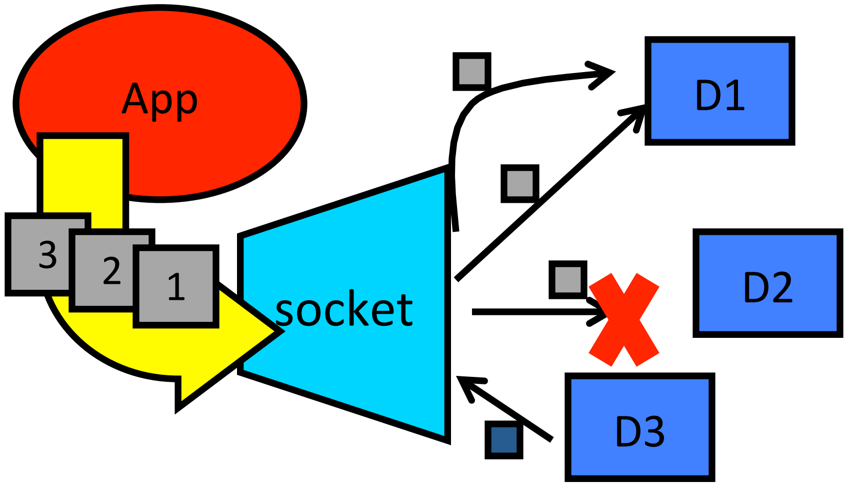
- Sockets can be either
  - **connection based** or **connectionless**: Is a connection established before communication or does each packet describe the destination?
  - **packet based** or **streams based**: Are there message boundaries or is it one stream?
  - **reliable** or **unreliable**. Can messages be lost, duplicated, reordered, or corrupted?

Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional



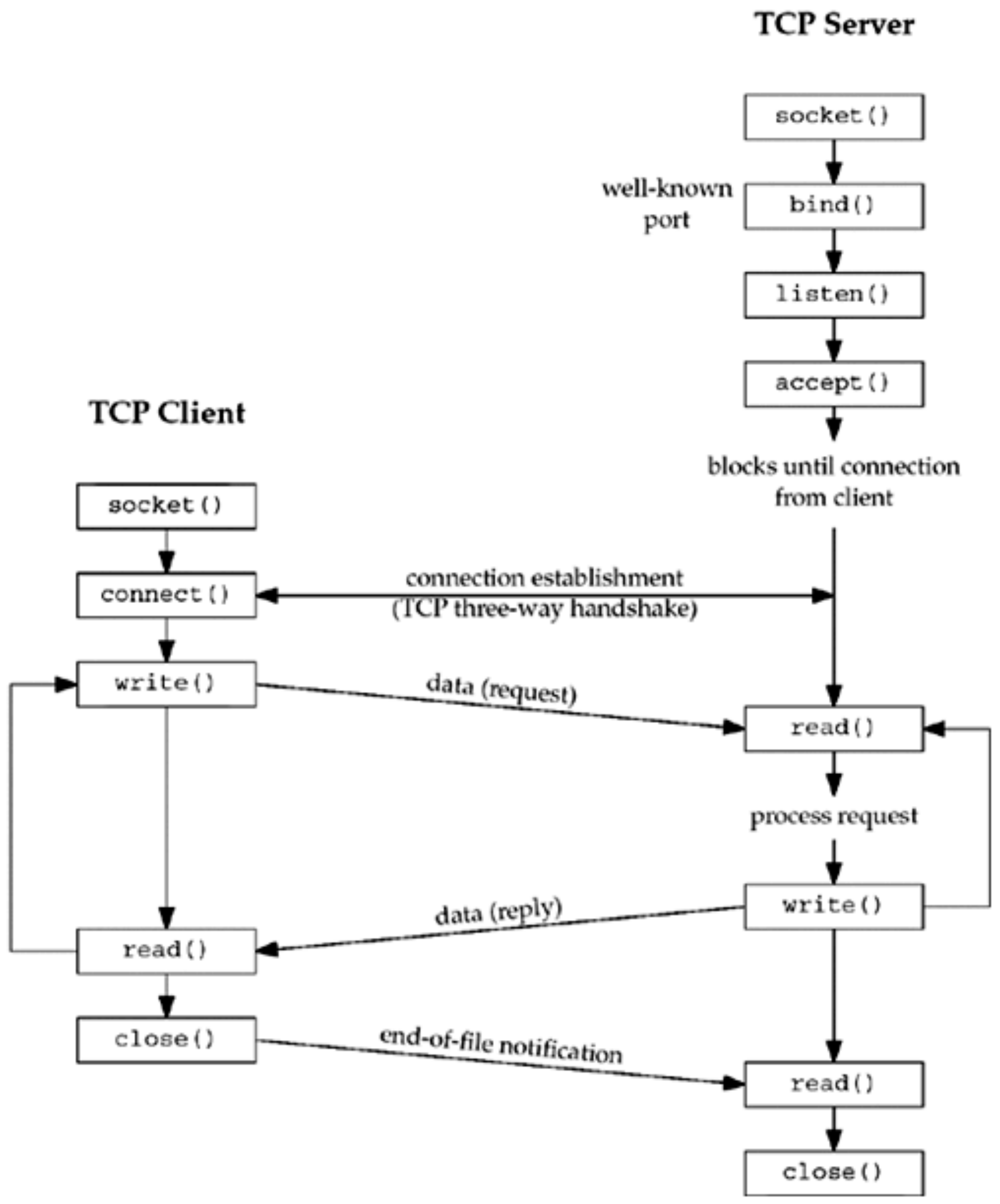
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" – app indicates dest. for each packet
  - can send or receive



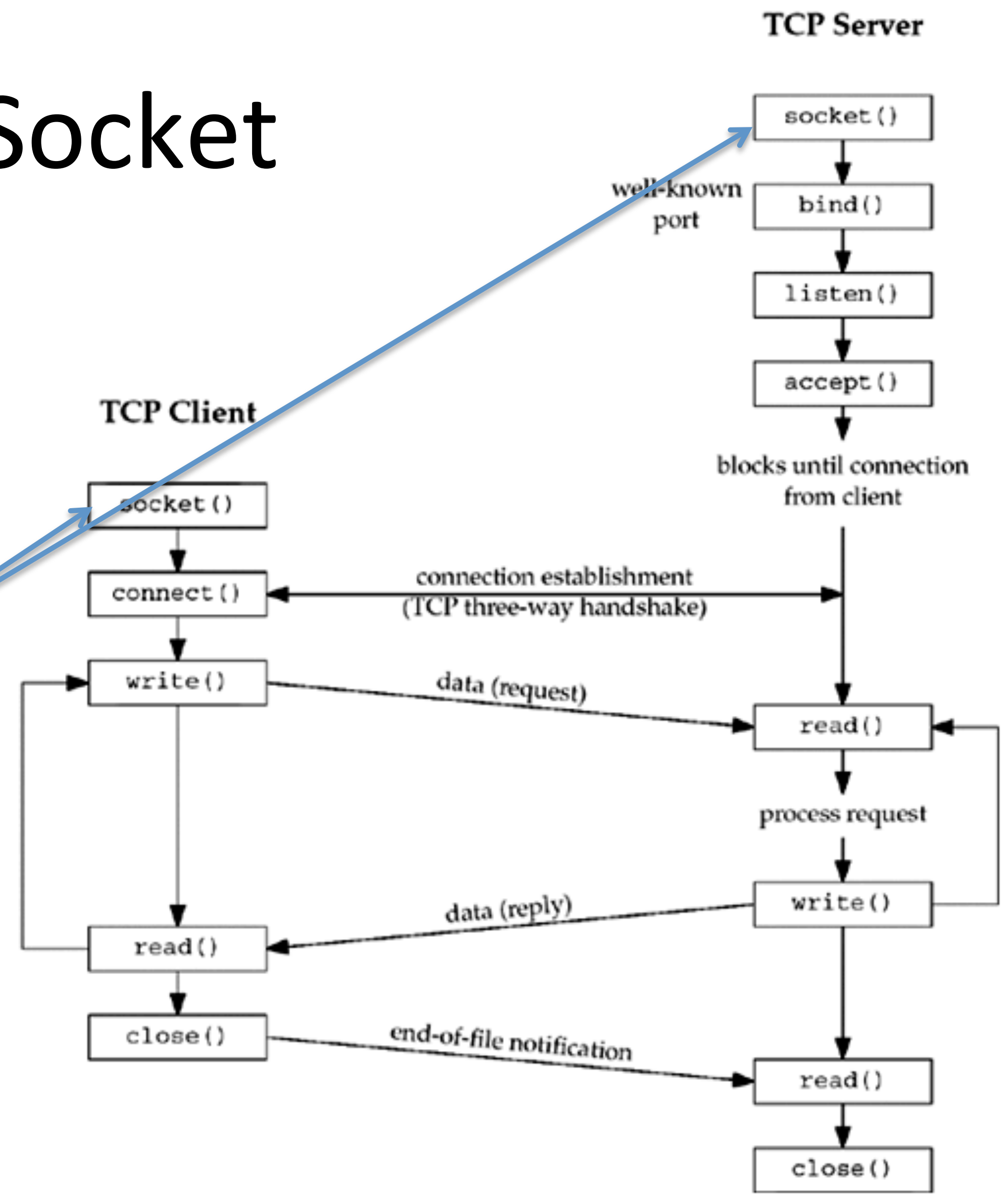
TCP socket

Models

- The Client-Server model
  - Most interprocess communication uses client-server model
  - Client & Server are two processes that wants to communicate with each other
  - The Client process connects to the Server process, to make a request for information/services own by the Server.
  - Once the connection is established between Client process and Server process, they can start sending / receiving information.


TCP Server
socket()
well-known port
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
connection establishment (TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

Socket



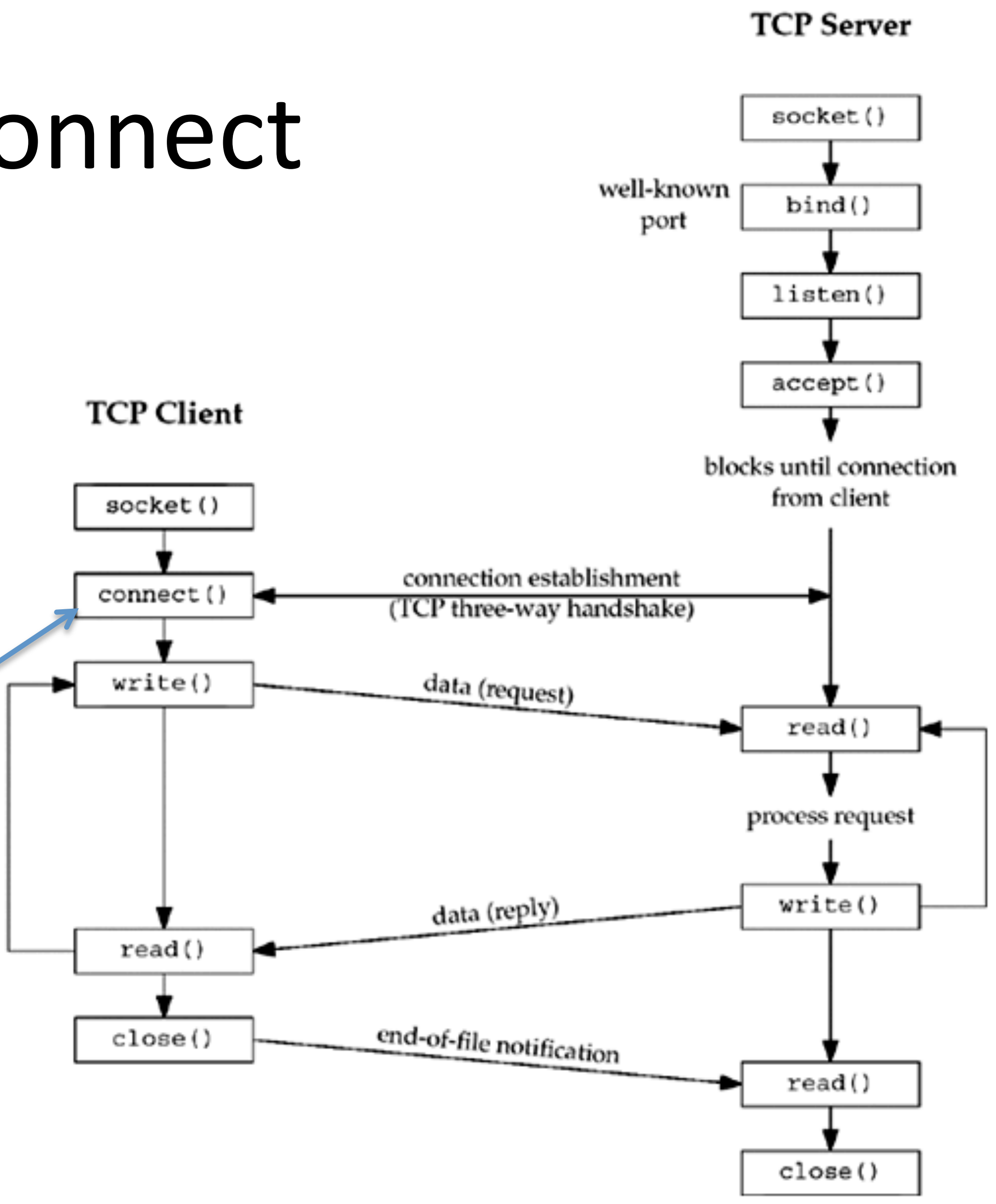
Creates the socket, the endpoint of the communication

Connect

It establishes the communication among client and server.

A TCP syn packet will be sent to the server to initiate the communication (TCP three-way handshake)

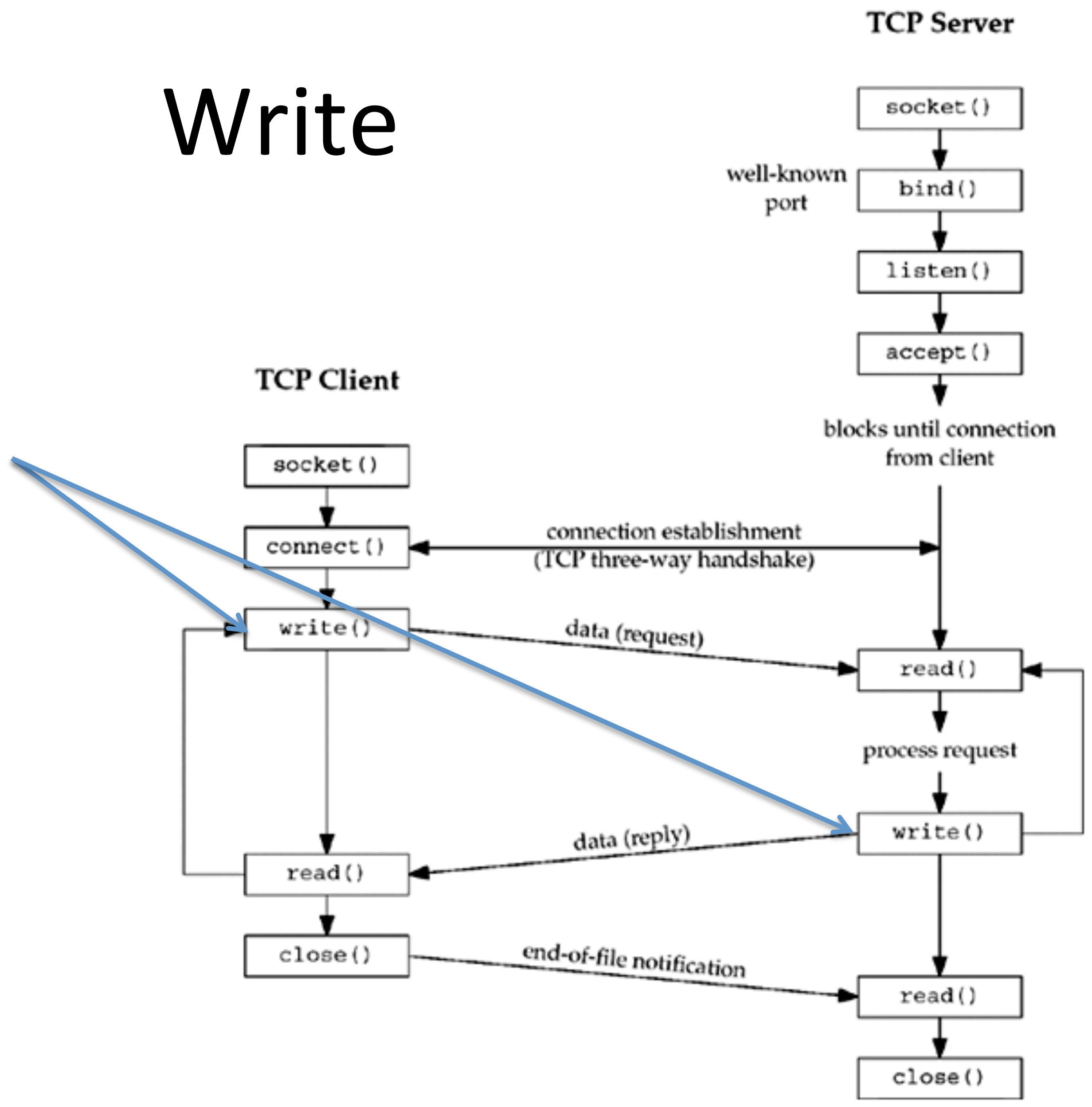
Before completing the connection, no data can be sent/received over the socket



Write

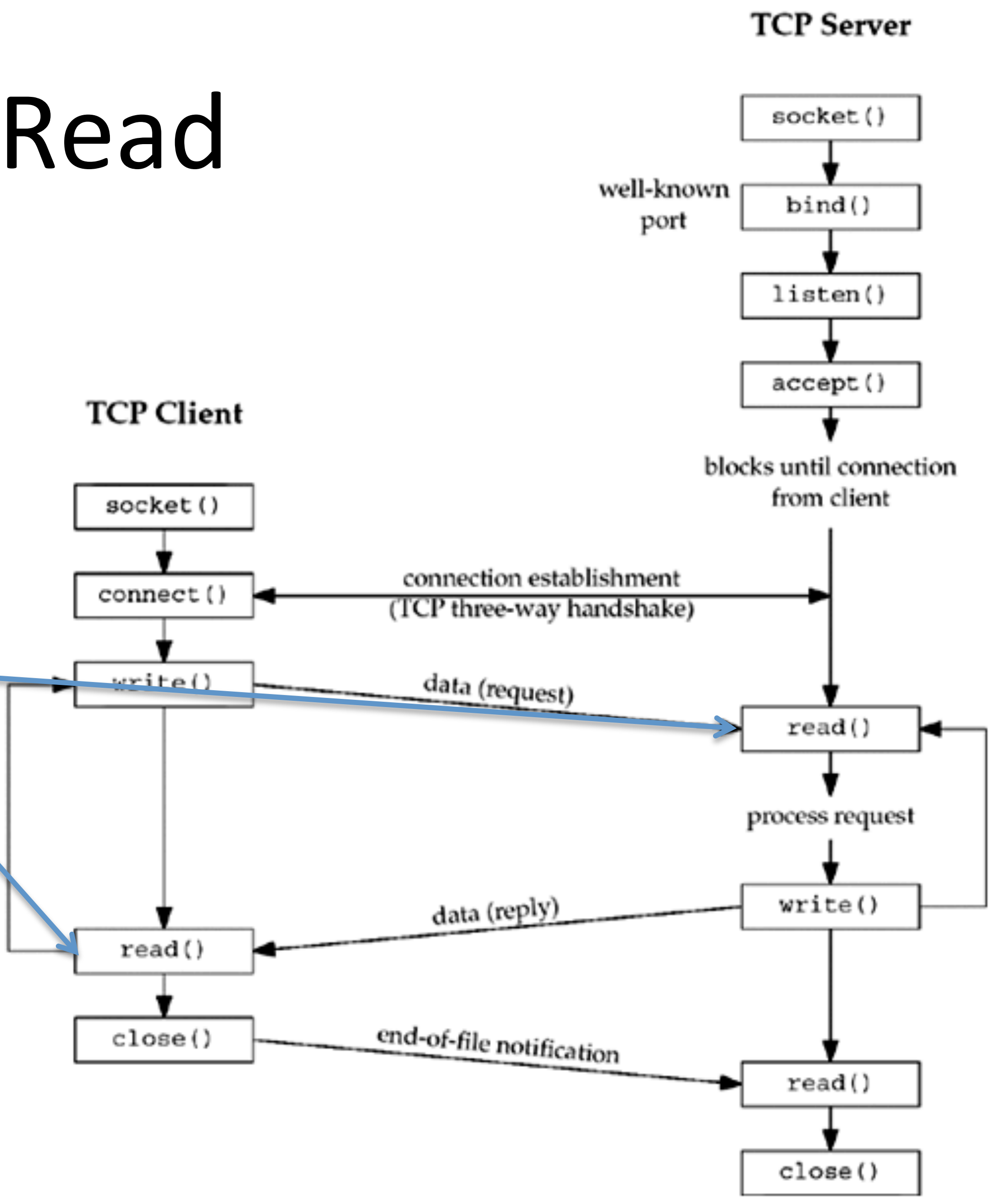
It sends data to the other endpoint of the communication

The communication must be already established (see connect)



Read

It receives data sent by the other endpoint of the communication



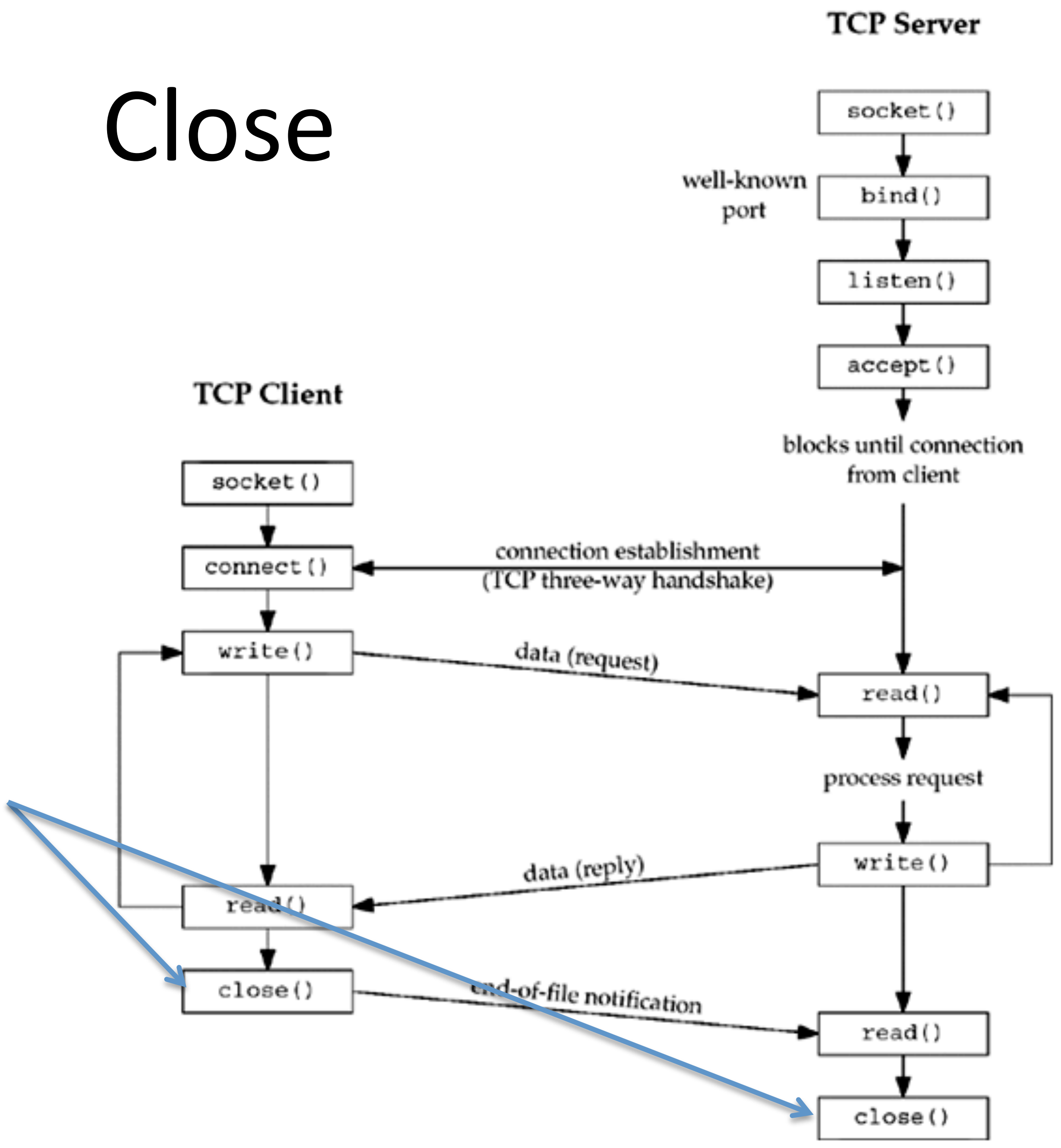
Close

It closes the socket and the communication.

It won't be possible to utilize the closing socket anymore

The O.S. will release the data structures utilized by the kernel for the communication

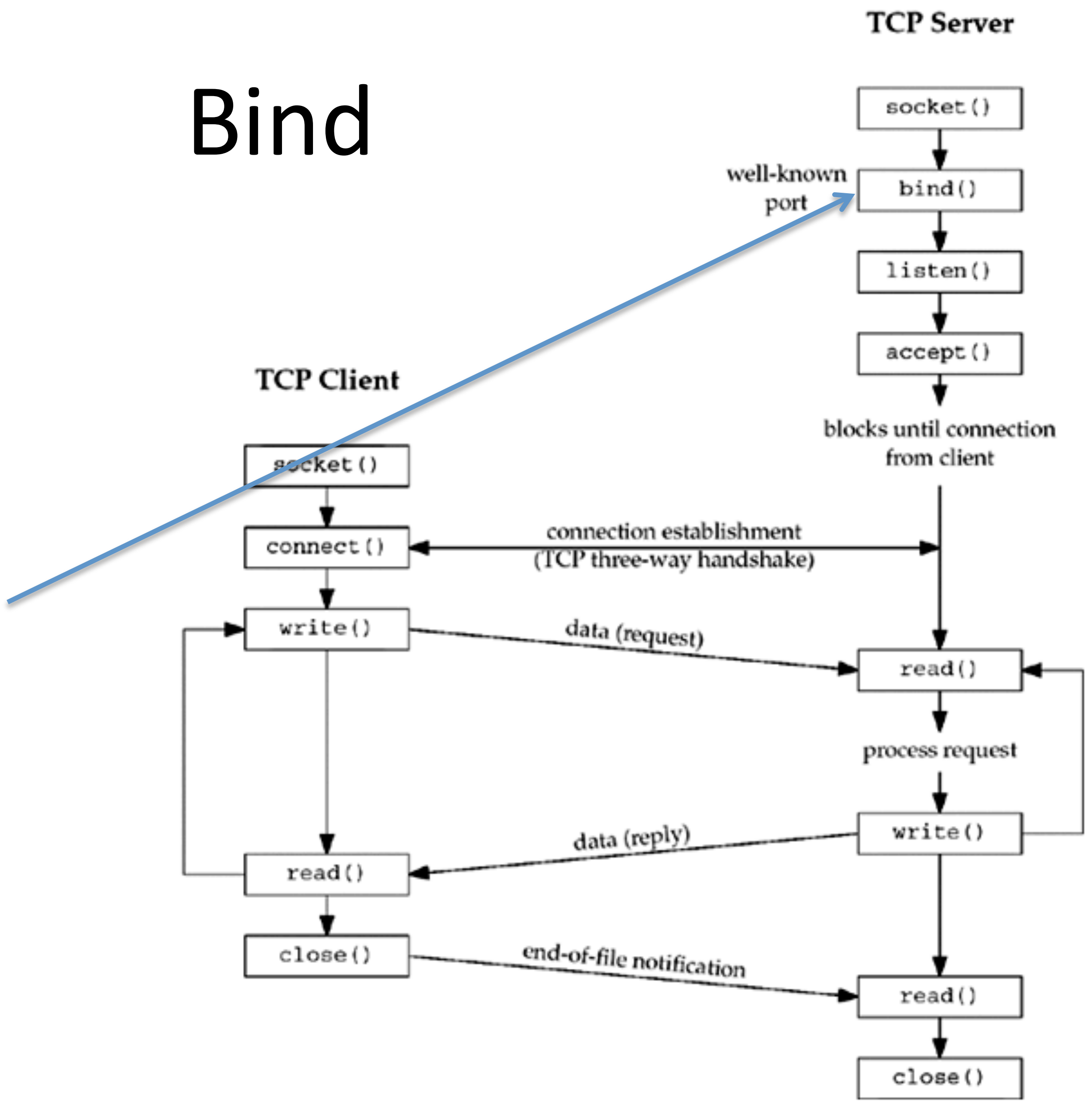
If there is still data waiting to be transmitted over the connection, normally close tries to complete this transmission



Bind

It associates an address (IP address and port number) to an open socket.

If not specified, address and port number will be chosen by the O.S. The client must know the server address to initiate the communication, therefore the server must bind the socket to a well known port

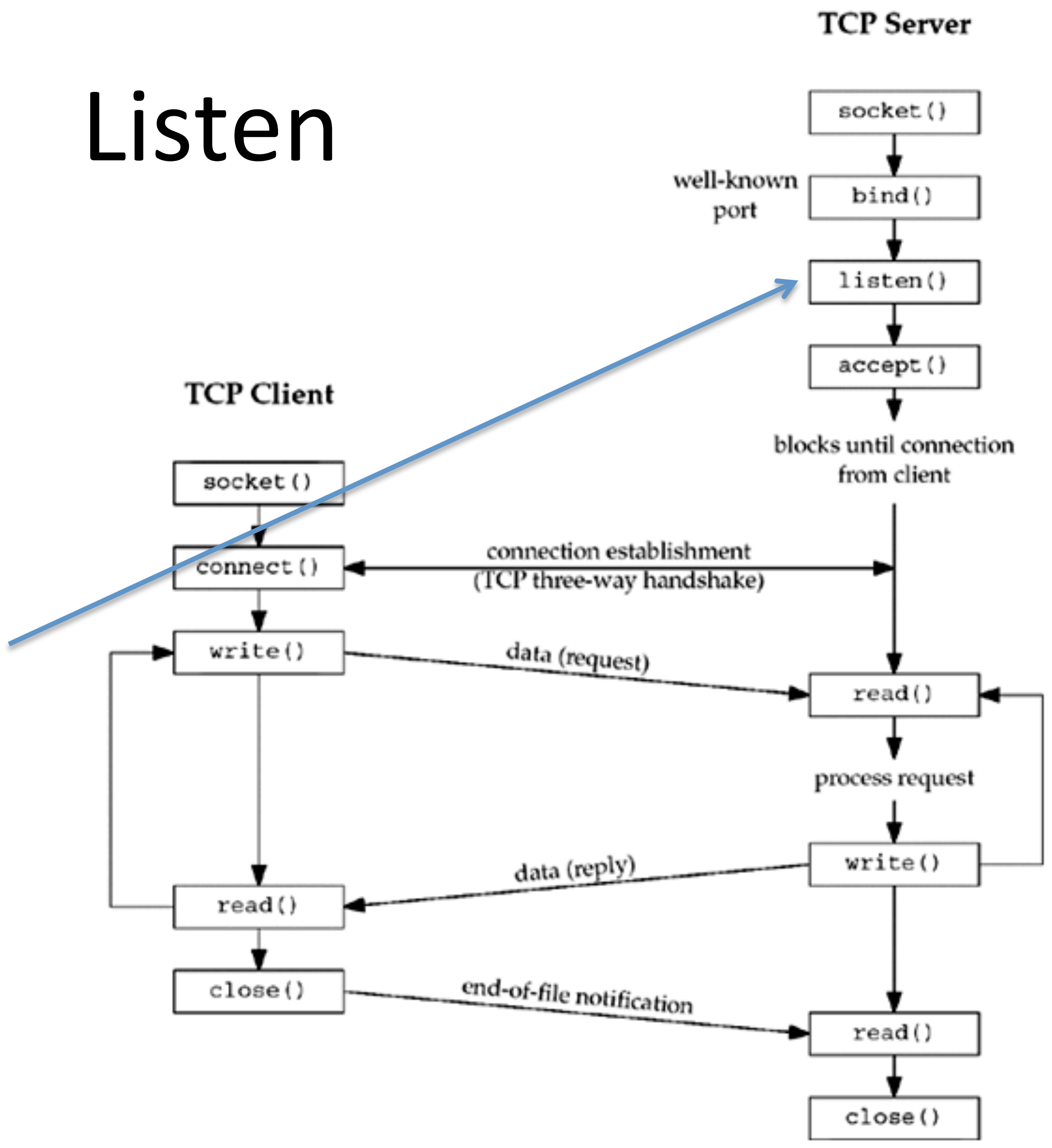


Listen

It enables connection requests on the socket – it tells the O.S. that the application is willing to accept connection request over that socket

It specifies the maximum number of connection requests that can be pending for this process Further requests will be dropped

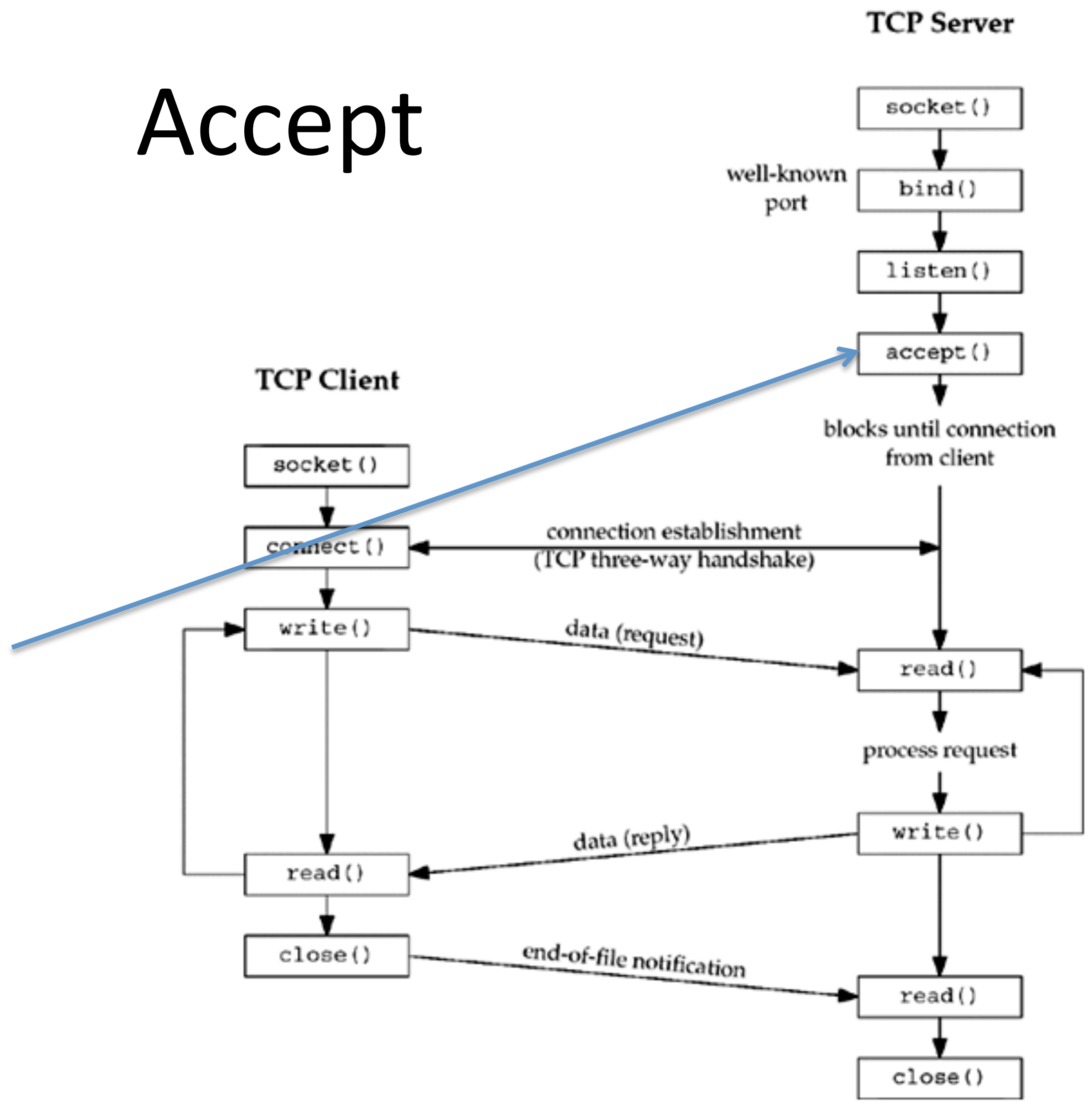
The listen function is not allowed for connectionless socket.



Accept

Establishes the connection with a specific client

It waits for a SYN packet sent by the client (using connect), accepts the connection request and notifies the client the success (TCP three way handshake)



Programming TCP Client in C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg){    perror(msg);    exit(0);}

int main(int argc, char *argv[]){
        int sockfd, portno, n;
        struct sockaddr_in serv_addr;
        struct hostent *server;
        char buffer[256];

        if (argc < 3) {
                fprintf(stderr,"usage %s hostname port
                exit(0);
        }
        portno = atoi(argv[2]);

        sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sockfd < 0)         error("ERROR opening socket");
```

```
/* a structure to contain an internet address
defined in the include file <netinet/in.h> */
struct sockaddr_in {
        short   sin_family; /* should be AF_INET */
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8]; /* not used, must be zero
*/
};

struct in_addr {
   unsigned long s_addr;
};
```

Programming TCP Client in C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg){    perror(msg);    exit(0);}

int main(int argc, char *argv[]){
        int sockfd, portno, n;
        struct sockaddr_in serv_addr;
        struct hostent *server;
        char buffer[256];

        if (argc < 3) {
                fprintf(stderr,"usage %s hostname port\n", a
                exit(0);
        }
        portno = atoi(argv[2]);

        sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sockfd < 0)        error("ERROR opening socket");
```

__Socket System Call__ – create an end point for communication

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

int socket(int *domain*, int *type*, int *protocol*);

Returns a descriptor

domain: selects protocol family
e.g. AF_UNIX, AF_INET …

type: specifies communication semantics
e.g. SOCK_DGRAM, SOCK_RAW

protocol: specifies a particular protocol to be used
e.g. IPPROTO_UDP, PPROTO_ICMP

Programming TCP Client in C

```
        server = gethostbyname(argv[1]);
        if (server == NULL) {  fprintf(stderr,"ERROR, no such host\n"); exit(0);    }
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr
                , server->h_length);
        serv_addr.sin_port = htons(portno);

        if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
                error("ERROR connecting");

        printf("Please enter the message: ");
        bzero(buffer,256);    fgets(buffer,255,stdin);
        n = send(sockfd,buffer,strlen(buffer),0);
        if (n < 0)          error("ERROR writing to socket
        bzero(buffer,256);
        n = recv(sockfd,buffer,255,0);
        if (n < 0)
                 error("ERROR reading from socket");
        printf("%s\n",buffer);
        close(sockfd);
        return 0;
}
```

__Connect System Call__ – initiates a connection on a socket

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

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

Returns 0 on success

sockfd: descriptor that must refer to a socket

serv_addr: address to which we want to connect

addrlen: length of *serv_addr*

Programming TCP Client in C

```
        server = gethostbyname(argv[1]);
        if (server == NULL) {  fprintf(stderr,"ERROR, no such host\n"); exit(0);    }
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr
                , server->h_length);
        serv_addr.sin_port = htons(portno);

        if (connect(sockfd,&serv_addr,sizeof(serv_add
                error("ERROR connecting");

        printf("Please enter the message: ");
        bzero(buffer,256);    fgets(buffer,255,stdin);
        n = send(sockfd,buffer,strlen(buffer),0);
        if (n < 0)          error("ERROR writing to socket
        bzero(buffer,256);
        n = recv(sockfd,buffer,255,0);
        if (n < 0)
                 error("ERROR reading from socket");
        printf("%s\n",buffer);
        close(sockfd);
        return 0;
}
```

Send System Call – send a message to a socket

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

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

Returns number of characters sent on success

s: descriptor that must refer to a socket in connected state

msg: data that we want to send

len: length of data

flags: use default 0. MSG_OOB, MSG_DONTWAIT

Programming TCP Client in C

```
        server = gethostbyname(argv[1]);
        if (server == NULL) {  fprintf(stderr,"ERROR, no such host\n"); exit(0);    }
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr
                , server->h_length);
        serv_addr.sin_port = htons(portno);

        if (connect(sockfd,&serv_addr,sizeof(serv_addr)) <
                error("ERROR connecting");

        printf("Please enter the message: ");
        bzero(buffer,256);    fgets(buffer,255,stdin);
        n = send(sockfd,buffer,strlen(buffer),0);
        if (n < 0)          error("ERROR writing to socket");
        bzero(buffer,256);
        n = recv(sockfd,buffer,255,0);
        if (n < 0)
                 error("ERROR reading from socket");
        printf("%s\n",buffer);
        close(sockfd);
        return 0;
}
```

Recv System Call – receive a message from a socket

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

int recv( int s, const void *buff, size_t len,
        int flags);
```

Returns number of bytes received on success

s: descriptor that must refer to a socket in connected state

buff: data that we want to receive

len: length of data

flags: use default 0. MSG_OOB, MSG_DONTWAIT

Programming TCP Client in C

```
        server = gethostbyname(argv[1]);
        if (server == NULL) {  fprintf(stderr,"ERROR, no such host\n"); exit(0);    }
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr
                , server->h_length);
        serv_addr.sin_port = htons(portno);

        if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
                error("ERROR connecting");

        printf("Please enter the message: ");
        bzero(buffer,256);    fgets(buffer,255,stdin);
        n = send(sockfd,buffer,strlen(buffer),0);
        if (n < 0)          error("ERROR writing to socket");
        bzero(buffer,256);
        n = recv(sockfd,buffer,255,0);
        if (n < 0)
                 error("ERROR reading from socket");
        printf("%s\n",buffer);
        close(sockfd);
        return 0;
}
```

Close System Call – close a socket descriptor

#include <unistd.h>

int close(int *s*);

Returns 0 on success
s: descriptor to be closed

Programming TCP Server in C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg){    perror(msg);    exit(0);}

int main(int argc, char *argv[]){
        int sockfd, newsockfd, portno, clilen;
        char buffer[256];
        struct sockaddr_in serv_addr, cli_addr;
        int n;
        if (argc < 2) { fprintf(stderr,"ERROR, no port provided\n"); exit(1); }
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) error("ERROR opening socket");
        bzero((char *) &serv_addr, sizeof(serv_addr));
        portno = atoi(argv[1]);
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        serv_addr.sin_port = htons(portno);
```

Programming TCP Server in C

```
        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");
        listen(sockfd,5);
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) error("ERROR on accept");
        bzero(buffer,256);
        n = recv(newsockfd,buffer,255,0);
        if (n < 0) error("ERROR reading fron
        printf("Here is the message: %s\n",b
        n = send(newsockfd,"I got your mes
        if (n < 0) error("ERROR writing to sc
        close(newsockfd);
        close(sockfd);
        return 0;
}
```

Bind System Call – bind a name to a socket

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

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

Returns 0 on success

sockfd: descriptor that must refer to a socket

serv_addr: address we want to use (INADDR_ANY)

addrlen: length of *serv_addr*

Programming TCP Server in C

```
        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");
        listen(sockfd,5);
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) error("ERROR on accept");
        bzero(buffer,256);
        n = recv(newsockfd,buffer,255,0);
        if (n < 0) error("ERROR reading from socket");
        printf("Here is the message: %s\n",buffer);
        n = send(newsockfd,"I got your message",18,0);
        if (n < 0) error("ERROR writing to socket");
        close(newsockfd);
        close(sockfd);
        return 0;
}
```

Listen System Call – listen for connections on a socket

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

int listen(int *s*, int *backlog*);

Returns 0 on success
s: descriptor that must refer to a socket
backlog: maximum length the queue for completely established sockets waiting to be accepted

Programming TCP Server in C

```
        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");
        listen(sockfd,5);
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) error("ERROR on accept");
        bzero(buffer,256);
        n = recv(newsockfd,buffer,255,0);
        if (n < 0) error("ERROR reading from socket");
        printf("Here is the message: %s\n",buffer);
        n = send(newsockfd,"I got your message",18,0
        if (n < 0) error("ERROR writing to socket");
        close(newsockfd);
        close(sockfd);
        return 0;
}
```

Accept System Call – accepts a connection on a socket

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

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

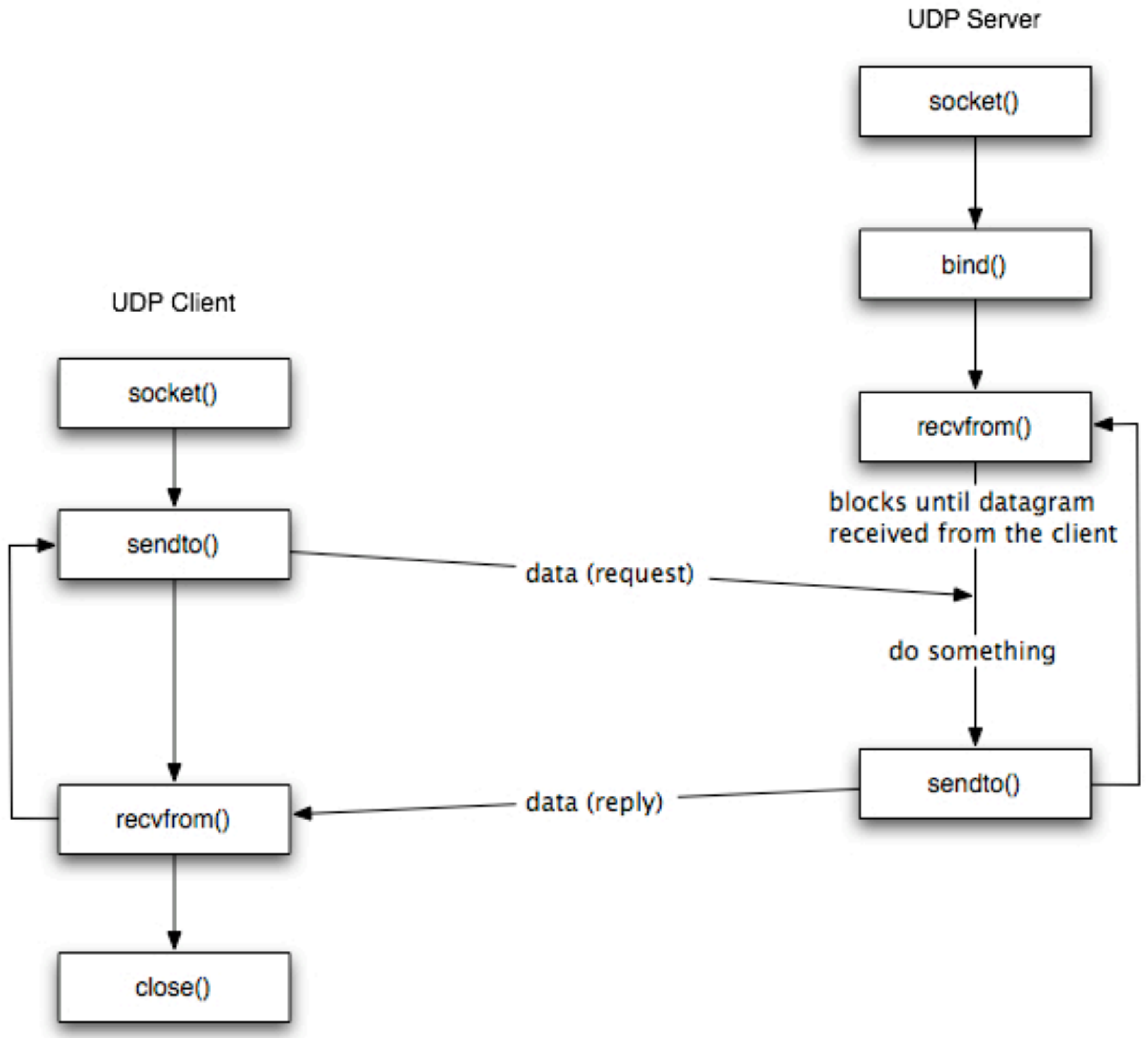
Returns a non-negative descriptor on success

sockfd: descriptor that must refer to a socket

addr: filled with address of connecting entity

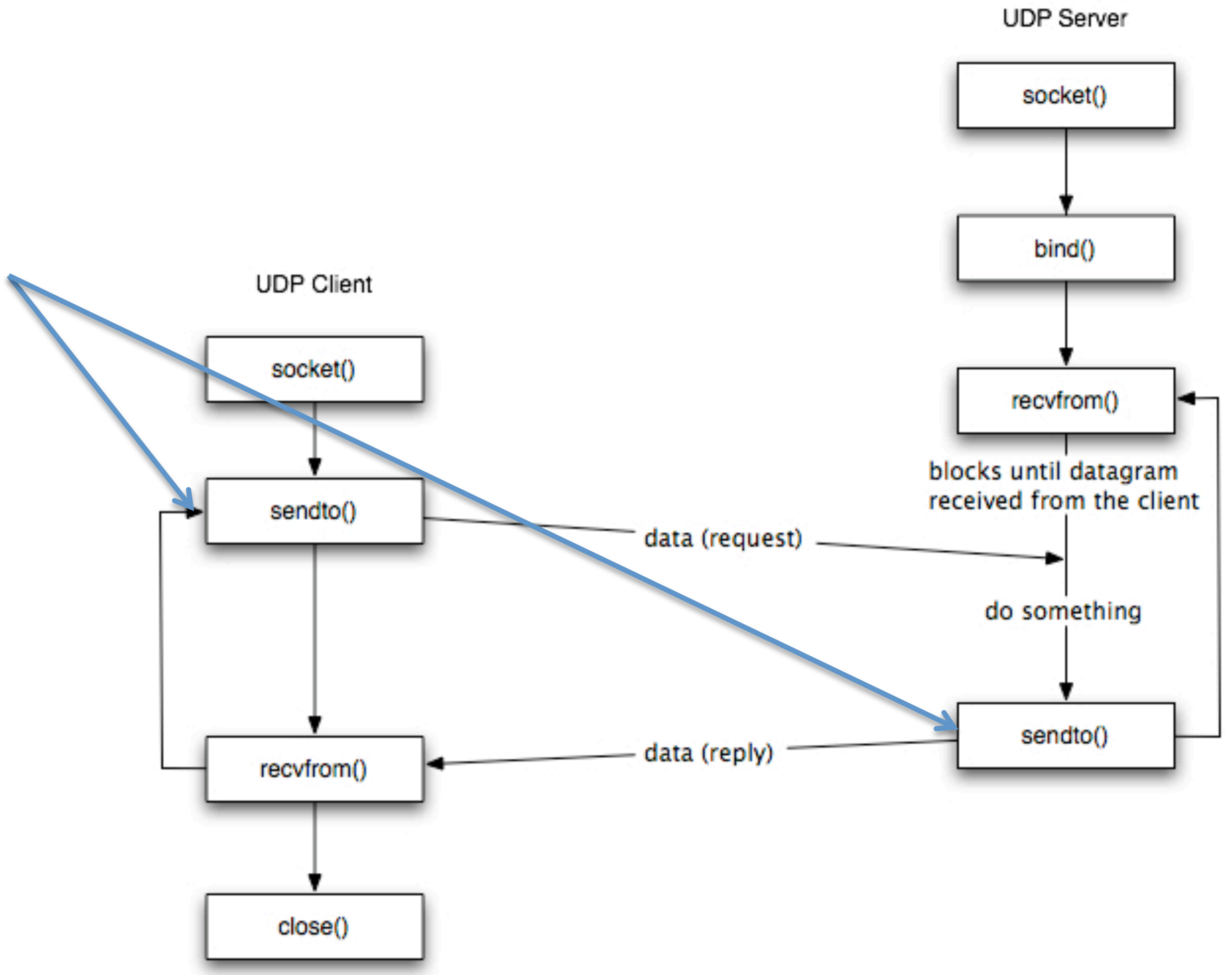
addrlen: length of *addr*

UDP socket


UDP Server
socket()
bind()
recvfrom()
blocks until datagram received from the client
do something
sendto()
UDP Client
socket()
sendto()
recvfrom()
close()
data (request)
data (reply)

Sendto

Sends a message to the destination specified as parameter

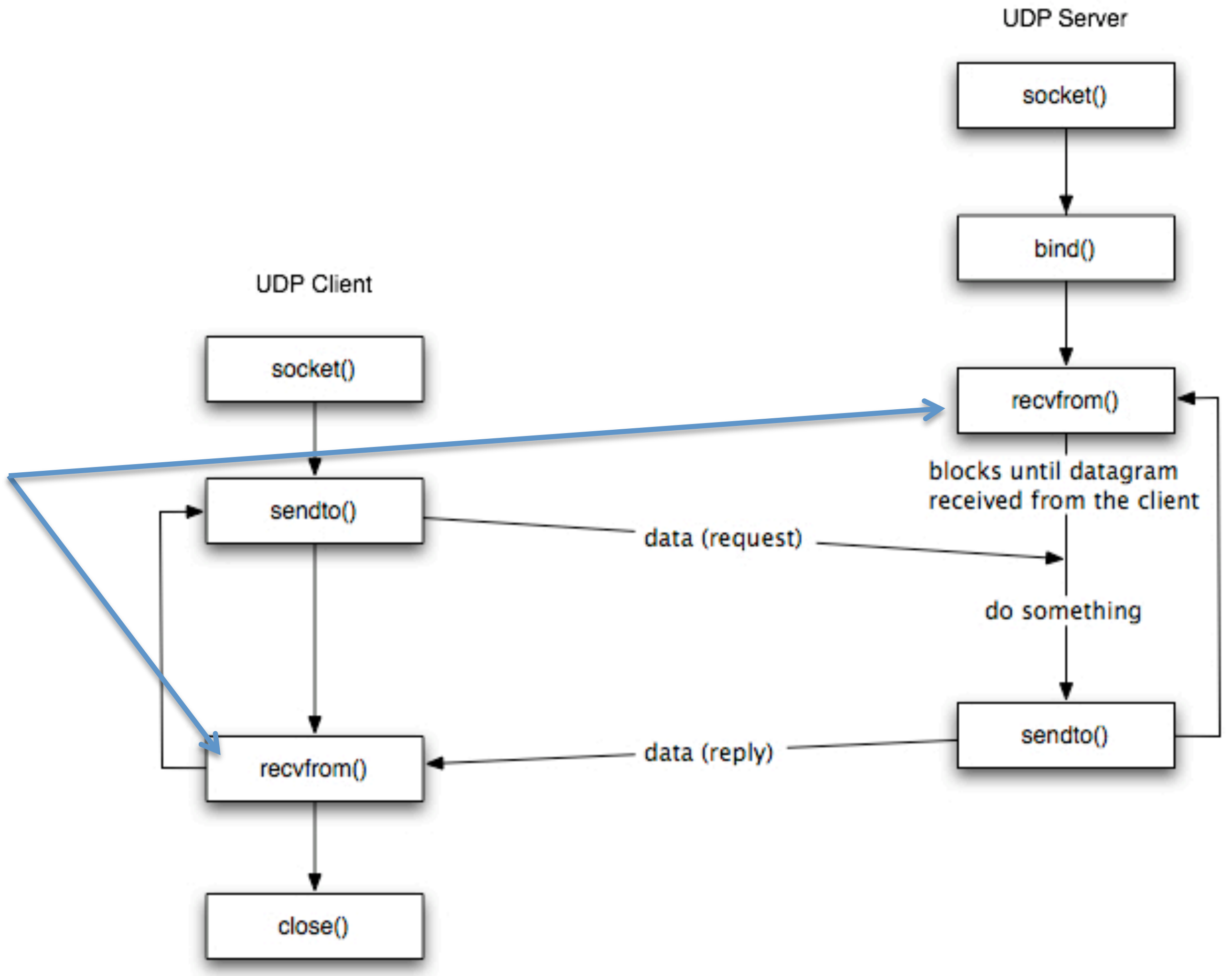


Recvfrom

Receives a message sent to the address bound to the socket and tells the application the address of the sender

The application can then use the sender address to send a reply (if needed)

In case of datagram communication, the function will read the entire payload of the datagram (read everything or nothing)



UDP client

The client code for a datagram socket client is the same as that for a stream socket with the following differences:

- the socket system call has
  - SOCK_DGRAM instead of SOCK_STREAM as its second argument
  - IPPROTO_UDP instead of IPPROTO_TCP as its third argument.
- connect() is not needed.
- instead of send() and recv(), the client uses sendto() and recvfrom()
  - If connect has been used, the application can use send() instead of sendto(). The datagrams will be sent to the destination specified by connect()

```
•       sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

•       len = sizeof(struct sockaddr_in);
•       while (1) {
•       /* write */
•               n = sendto(sock,“Got your message\n",17, 0,(struct sockaddr *) &server, len);
•               f (n < 0) error("sendto");
•       /* read */
•               n = recvfrom(sock,buf,1024,0,(struct sockaddr *)&from, len);
                if (n < 0) error("recvfrom");
•       }
```

UDP server

- Server code with a datagram socket is similar to the stream socket code with following differences.
  - Servers using datagram sockets do not use the listen() or the accept() system calls.
  - After a socket has been bound to an address, the program calls recvfrom() to read a message or sendto() to send a message.

```
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

len = sizeof(struct sockaddr_in);
while (1) {
/* read */
  n = recvfrom(sock,buf,1024,0,(struct sockaddr *)&from, len);
  if (n < 0) error("recvfrom");

/* write */
  n = sendto(sock,"Got your message\n",17, 0,(struct sockaddr *)&from, len);
  if (n < 0) error("sendto");
}
```

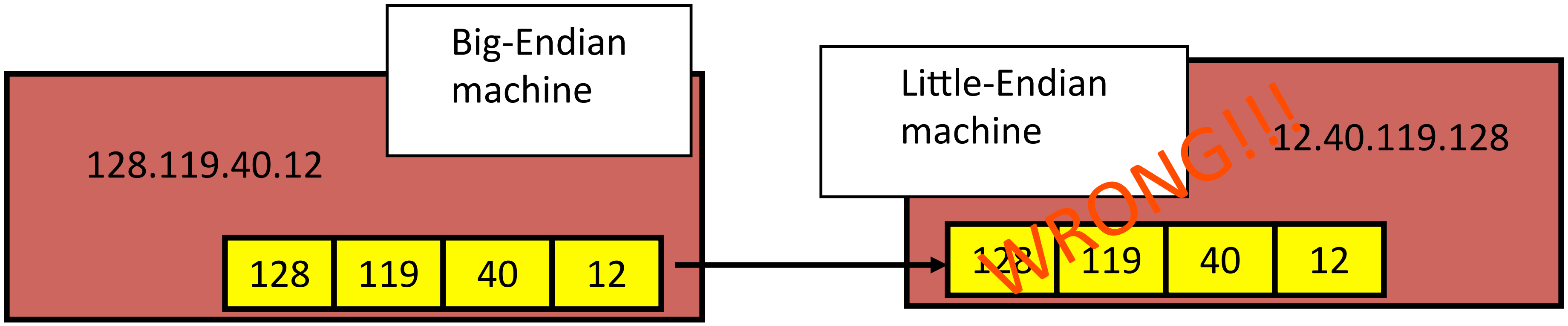
Byte Ordering

Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

❒ Problem:
- ❍ different machines / OS's use different word orderings
  - little-endian: lower bytes first
  - big-endian: higher bytes first
- ❍ these machines may communicate with one another over the network



Solution: Network Byte-Ordering

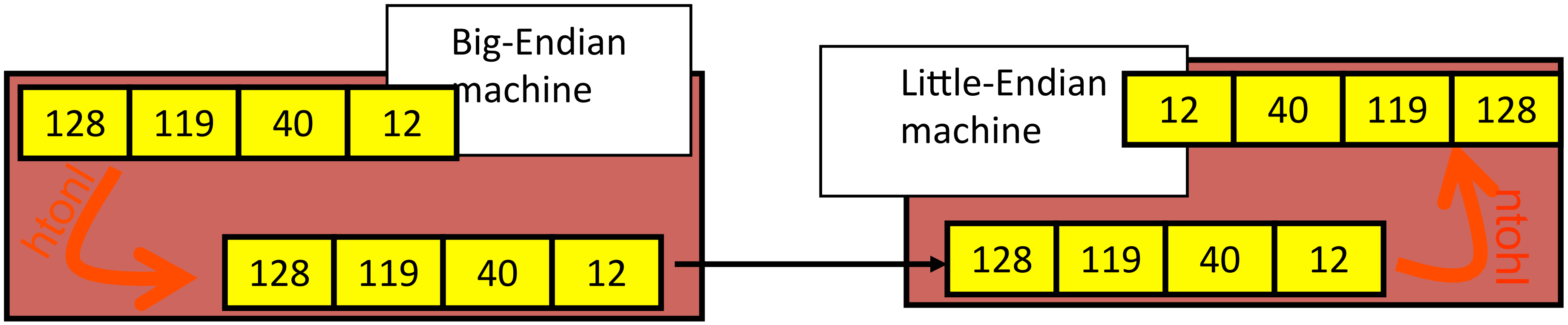
- Defs:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

UNIX' s byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

❒ On big-endian machines, these routines do nothing

❒ On little-endian machines, they reverse the byte order



Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing w/ blocking (cont' d)

- Options:
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.
- What does select do?
  - waits on multiple file descriptors and timeout
  - can be permanent blocking, time-limited blocking or non-blocking
  - returns when any file descriptor
    - is ready to be read or
    - written or
    - indicate an error, or
    - timeout exceeded

select function call

- int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - 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, even if nothing ready - can be 0 or ∞
    (point timeout parameter to NULL for ∞)

To be used with select:

- Recall select uses a structure, struct fd_set
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- After calling select:
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"

Example: Server Programming

- create stream socket (*socket()*)
- Bind port to socket (*bind()*)
- Listen for new client (*listen()*)
- While
  - Wait for (*select()*)

(depending on which file descriptors are ready)

  - accept user connection and create a new socket (*accept()*)
  - data arrives from client (*recv()*)
  - data has to be send to client (*send()*)

Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

Exercise

- Not mandatory. **Highly suggested**, especially for those who never used socket

Exercise 1

- TCP server wait for messages from a client
- TCP Client receives input from user (a string) and sends the message to the server
- The server prints the message (remember: you might need multiple read to receive the entire message), reverts the message (optional) and sends back the message to the client
- The client prints the message

Exercise 2

- Try this out with UDP communication too
- Try Exercise 1 and 2 with and without select

Readings

- Man pages in Linux
  Accessible through following command
  - man 2 <system_call_name>
  - E.g. man 2 socket
  - You can find the content also online (i.e. http://linux.die.net/man/)
- Book "Unix network programming" by Richard Stevens
- Beej' s guide to Network Programming
  http://beej.us/guide/bgnet/

- Tutorials:
  - http://www.linuxhowtos.org/C_C++/socket.htm

- Examples:
  - http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/textcode.html

- If you need more, on Internet you can find plenty of tutorial, books, slides on socket programming