Network Programming

- Network Programming as Programming across Machine Boundaries
- The Sockets API
- Reliable Communication Channels: TCP
- Dangerous at any Speed; connectionless communication and UDP
- Server Design

- Reading: R&R, Ch 18

Naming in a Networked Environment

Reminder:

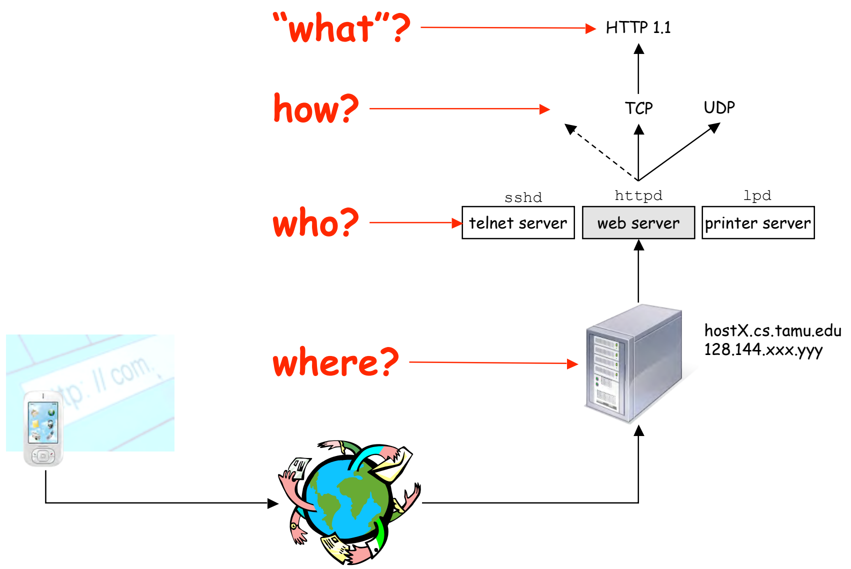
- Naming within a **single address space**?
  - addresses, duh!

- Naming **across address spaces**?
  - file descriptors
  - filenames (use file system as name space)
  - keys

New:

- Naming **across machine boundaries**?

Naming in a Networked Environment (II)



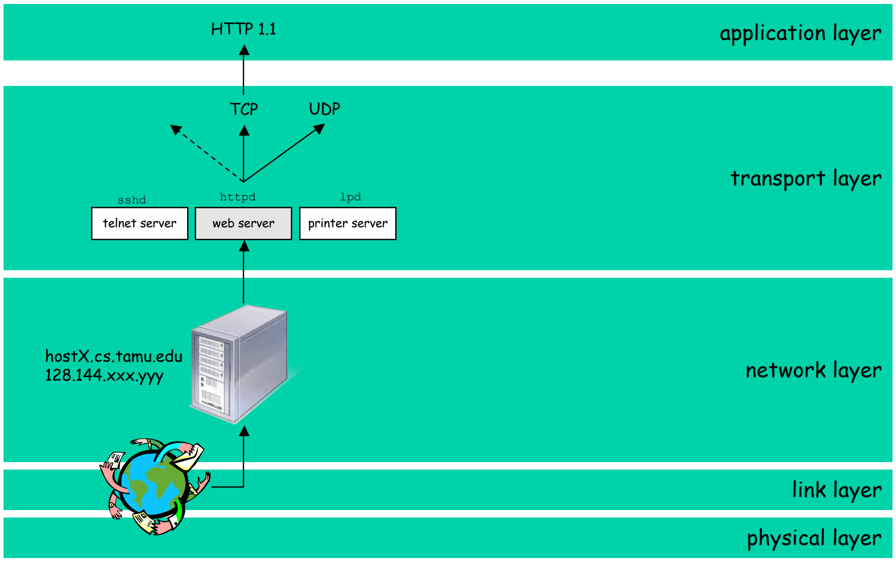
"what"? HTTP 1.1

how? TCP UDP

who?
sshd telnet server
httpd web server
lpd printer server

where?
hostX.cs.tamu.edu
128.144.xxx.yyy

Protocols for a Networked Environment



application layer
HTTP 1.1

transport layer
TCP UDP
sshd telnet server
httpd web server
lpd printer server

network layer
hostX.cs.tamu.edu
128.144.xxx.yyy

link layer

physical layer

The Socket API

- What does an API need to support? [Comer&Stevens]
  - allocate local resources (buffers)
  - specify local and remote communication endpoints
  - initiate a connection
  - wait for an incoming connection
  - send or receive data
  - determine when data arrives
  - generate urgent data
  - handle incoming urgent data
  - terminate a connection gracefully
  - handle connection termination from the remote site
  - abort communication
  - handle error conditions or a connection abort
  - release local resources when communication finishes

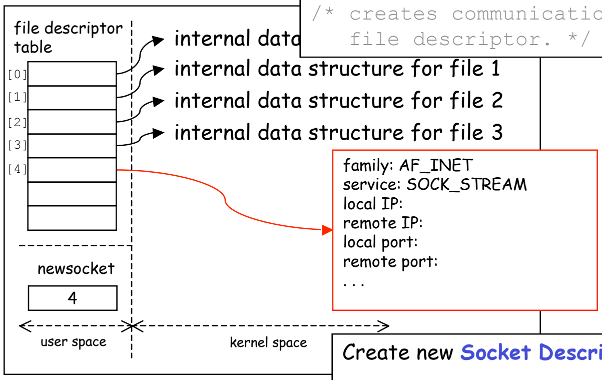
- Existing TCP/IP APIs:
  - **Berkeley Socket** API
    - aka socket API, socket interface, sockets
    - adapted by Linux and others
  - **Windows Sockets**
  - **System V Unix TLI** (Transport Layer Interface)

Specifying a Protocol Interface

- Reality Check: “All” networks today are based on TCP/IP.
- How to define a network API, then?!
  - **Approach 1:** Define functions that specifically support TCP/IP communication.
    - e.g. `makeTCPconnection(int32 host, int16 portno);`
  - **Approach 2:** Define functions that support network communication in general, and use parameters to handle TCP/IP as a special case.
- The socket API provides generalized functions that support network communication using many possible protocols.
- The programmer specifies the type of service required rather than the name of a specific network protocol.

Sockets and File Descriptors

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

int socket(int family, int type, int protocol)
/* creates communication endpoint and returns
   file descriptor. */
```



Create new **Socket Descriptor**:

```
int newsocket;
newsocket = socket(AF_INET, SOCK_STREAM, 0);
```

TCP? UDP?!!

- **Connection-oriented Style** (SOCK_STREAM) of communication
- Implemented in TCP/IP as the **Transport Control Protocol (TCP).**
- TCP provides full **reliability**:
  - verifies that data arrives, with **automatic retransmission**
  - computes **checksums** to detect corruption
  - uses sequence numbers to **guarantee ordering** of received packets
  - automatically **eliminates duplicated packets**
  - provides **flow control** (ensures that sender does not send more packets than the receiver can handle)
  - informs sender if network becomes inoperable for any reason
- TCP **protects network resources**:
  - provides **congestion control** (throttles transmission when it detects that network is congested)
- What is the **cost** of all this?! Connection establishment overhead.

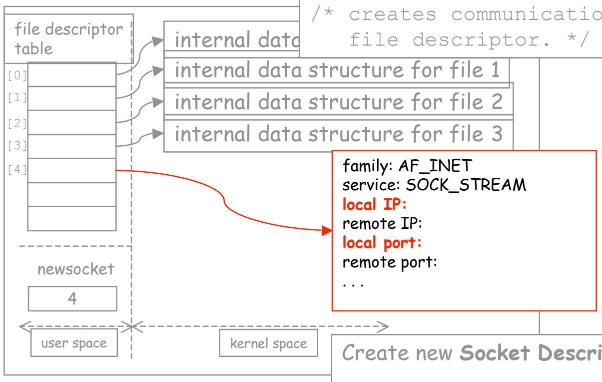
TCP? UDP?!!

- **Connectionless Style** (SOCK_DGRAM) of communication
- Implemented in TCP/IP as the **User Datagram Protocol (UDP).**
- UDP provides **no guarantee about reliable delivery**:
  - packets can be **lost**, **duplicated**, **delayed**, or **delivered out of order**
- UDP works well if the underlying network works well, e.g. local network

- **In practice**, programmers use UDP only when:
  1. The application requires that UDP must be used. (The application has been designed to handle reliability and delivery errors.)
  2. The application relies on hardware broadcast or multicast for delivery.
  3. Application runs in reliable, local environment, and overhead for reliability is unnecessary.

The Socket's Local Address

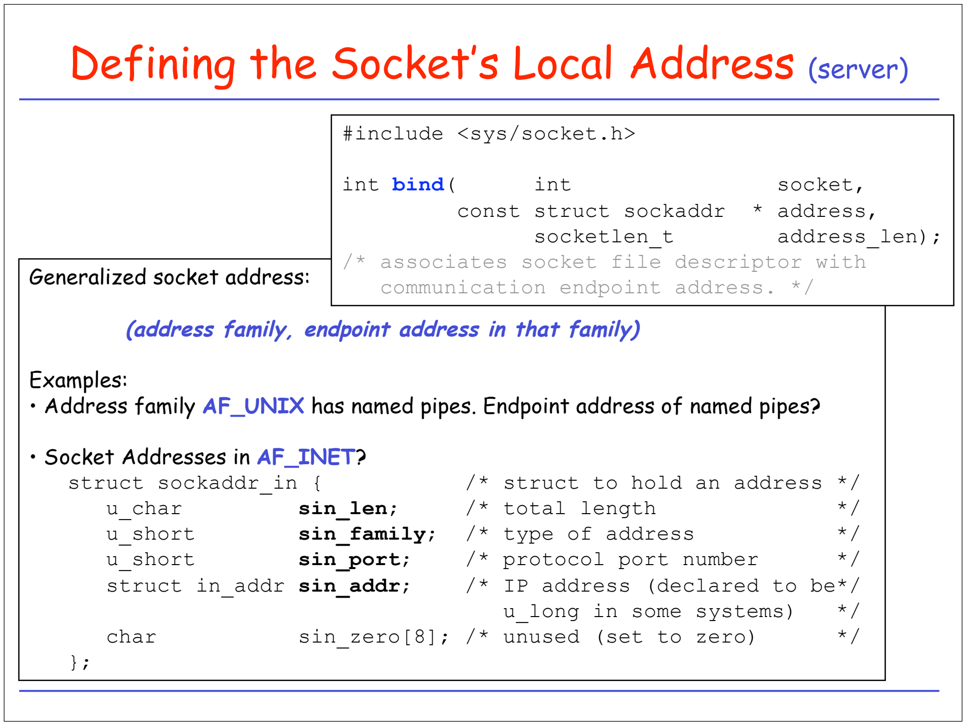
```
#include <sys/socket.h>

int socket(int family, int type, int protocol)
/* creates communication endpoint and returns
   file descriptor. */
```



family: AF_INET
service: SOCK_STREAM
local IP:
remote IP:
local port:
remote port:
. . .

Create new **Socket Descriptor**:

```
int newsocket;
newsocket = socket(AF_INET, SOCK_STREAM, 0);
```



Defining the Socket's Local Address (server)

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

int bind(      int                   socket,
         const struct sockaddr  * address,
               socketlen_t           address_len);
/* associates socket file descriptor with
   communication endpoint address. */
```

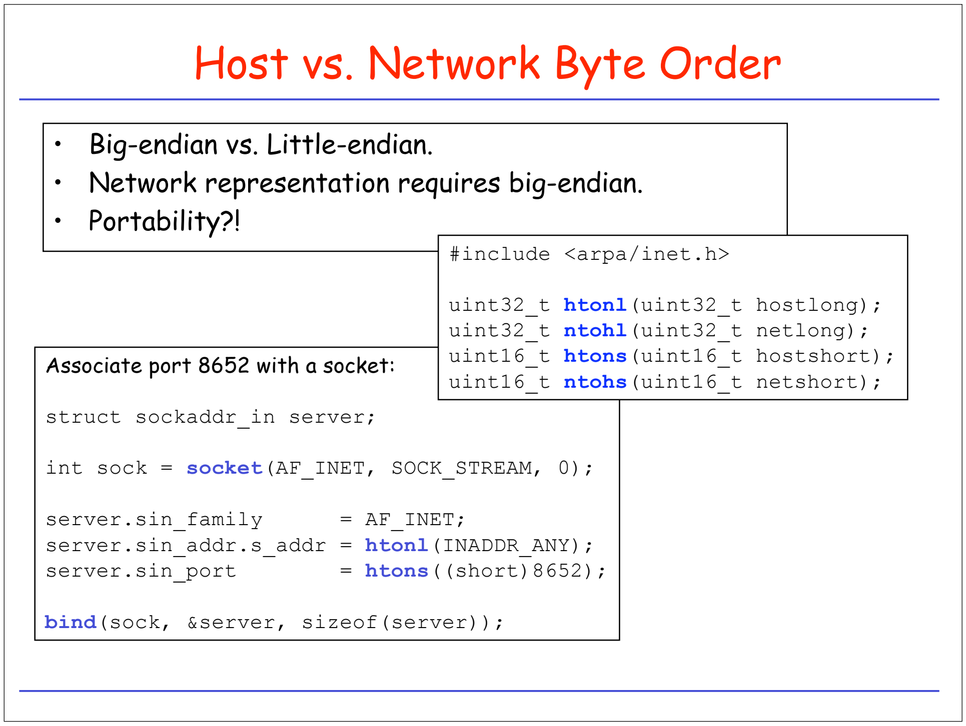
Generalized socket address:

(address family, endpoint address in that family)

Examples:

- Address family **AF_UNIX** has named pipes. Endpoint address of named pipes?
- Socket Addresses in **AF_INET**?

```
   struct sockaddr_in {              /* struct to hold an address */
      u_char          sin_len;       /* total length              */
      u_short         sin_family;    /* type of address           */
      u_short         sin_port;      /* protocol port number      */
      struct in_addr  sin_addr;      /* IP address (declared to be*/
                                        u_long in some systems)   */
      char            sin_zero[8];   /* unused (set to zero)      */
   };
```



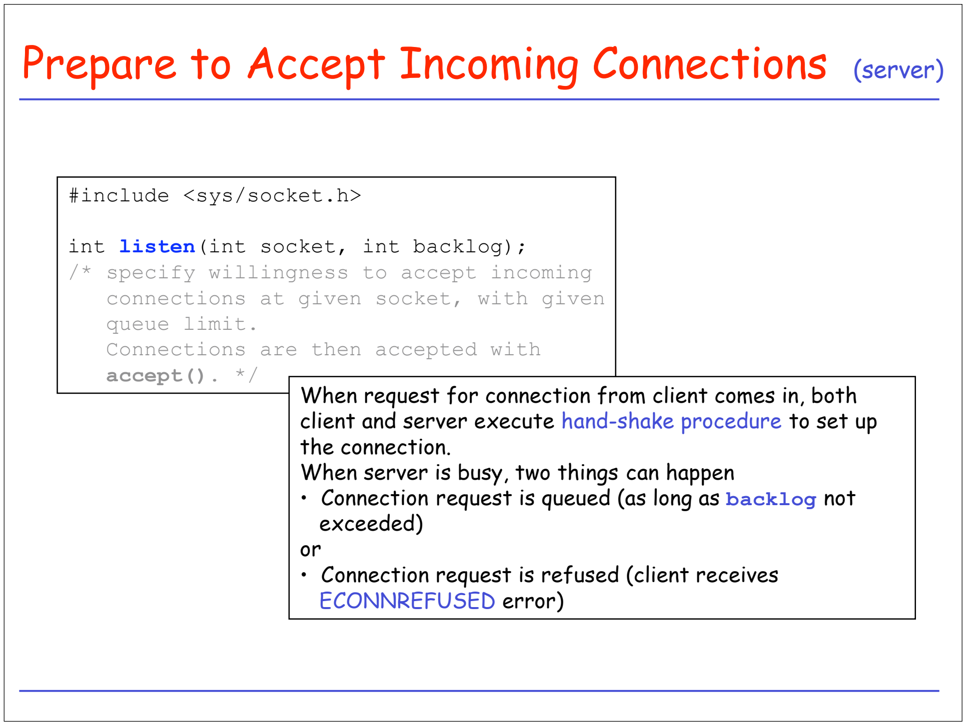
Host vs. Network Byte Order

- Big-endian vs. Little-endian.
- Network representation requires big-endian.
- Portability?!

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);
```

Associate port 8652 with a socket:

```
struct sockaddr_in server;

int sock = socket(AF_INET, SOCK_STREAM, 0);

server.sin_family       = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port         = htons((short)8652);

bind(sock, &server, sizeof(server));
```

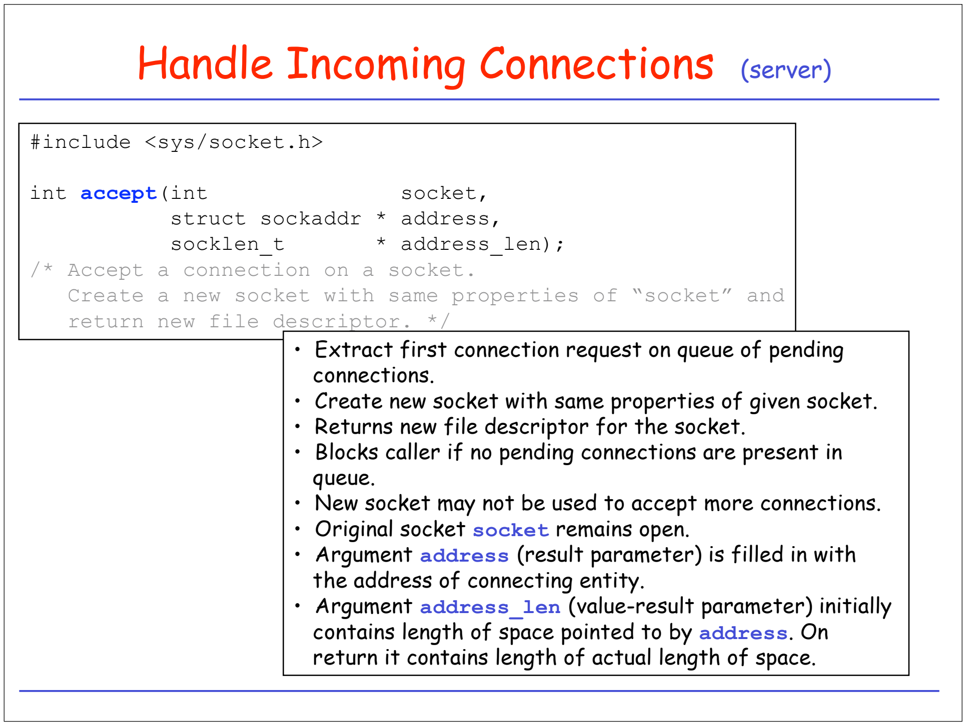


Prepare to Accept Incoming Connections (server)

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

int listen(int socket, int backlog);
/* specify willingness to accept incoming
   connections at given socket, with given
   queue limit.
   Connections are then accepted with
   accept(). */
```

When request for connection from client comes in, both client and server execute hand-shake procedure to set up the connection.
When server is busy, two things can happen
- Connection request is queued (as long as `backlog` not exceeded)

or
- Connection request is refused (client receives ECONNREFUSED error)



Handle Incoming Connections (server)

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

int accept(int               socket,
           struct sockaddr * address,
           socklen_t       * address_len);
/* Accept a connection on a socket.
   Create a new socket with same properties of "socket" and
   return new file descriptor. */
```

- Extract first connection request on queue of pending connections.
- Create new socket with same properties of given socket.
- Returns new file descriptor for the socket.
- Blocks caller if no pending connections are present in queue.
- New socket may not be used to accept more connections.
- Original socket `socket` remains open.
- Argument `address` (result parameter) is filled in with the address of connecting entity.
- Argument `address_len` (value-result parameter) initially contains length of space pointed to by `address`. On return it contains length of actual length of space.

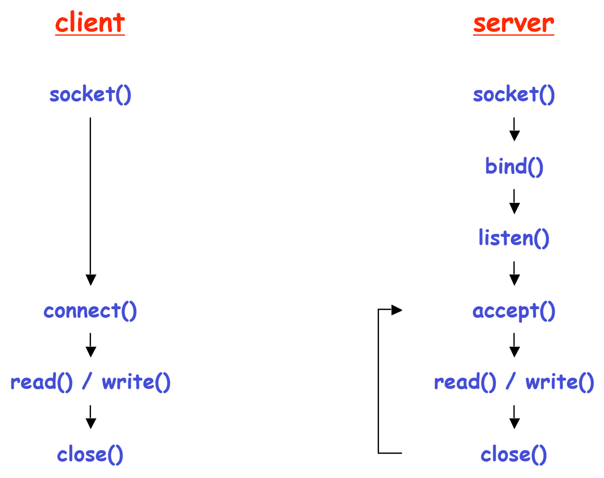
In the meantime, at the Client's End (client)

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

int connect(int                socket,
            struct sockaddr *  address,
            socklen_t          address_len);
/* Initiate a connection on a socket.
   Structure address specifies the
   other endpoint of the connection. */
```

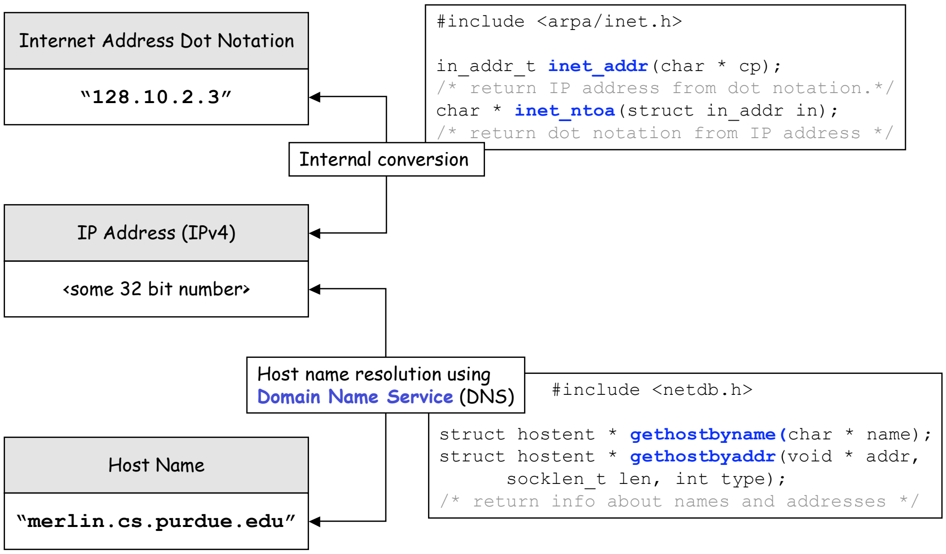
Note difference between SOCK_DGRAM and SOCK_STREAM sockets:

- For SOCK_STREAM sockets, `connect` attempts to establish connection with other socket.
  - Generally, stream sockets may connect only once.
- For SOCK_DGRAM sockets, `connect` specifies the peer with which to associate socket.
  - Datagram sockets may dissolve association by connecting to invalid address, such as null address.
- For `address`, fill in family, address, and port number.

Connection Establishment: Summary



Addressing: IP Address vs. Dotted vs. Name



```
#include <arpa/inet.h>

in_addr_t inet_addr(char * cp);
/* return IP address from dot notation.*/
char * inet_ntoa(struct in_addr in);
/* return dot notation from IP address */
```

```
#include <netdb.h>

struct hostent * gethostbyname(char * name);
struct hostent * gethostbyaddr(void * addr,
      socklen_t len, int type);
/* return info about names and addresses */
```

Host Names and IP Addresses

Host Information:

```
#include <netdb.h>

struct hostent {
  char  * h_name;       /* canonical name of host */
  char ** h_aliases;    /* alias list */
  int     h_addrtype;   /* host address type */
  int     h_length;     /* length of address */
  char ** h_addr_list;  /* list of addresses */
}
```

Example: Translate a host name into IP address for use in `connect()` call:

```
struct hostent * hp;
struct sockaddr_in the_server;

if ((hp = gethostbyname("www.cs.tamu.edu")) == NULL)
  fprintf(stderr, "Failed to resolve host name\n");
else
  memcpy((char*)&the_server.sin_addr.s_addr,
         hp->h_addr_list[0], hp->h_length);
```

Example TCP Client: DAYTIME client [Comer]

```
#define LINELEN 128

/* forward */
int connectTCP(const char * host, const char * service);

/* main program */
int main(argc, char * argv) {

  char * host    = “localhost”;/* use local host if none supplied */
  char * service = “daytime”;  /* default service port */

  if (argc > 1) host    = argv[1];
  if (argc > 2) service = argv[2];

  int s = connectTCP(host, service);

  while ( (int n = read(s, buf, LINELEN)) > 0) {
    buf[n] = ‘\0’;                /* ensure null terminated */
    (void) fputs(buf, stdout);
  }
}
```

Example TCP Client: (cont…)

```
#define LINELEN 128

/* forward
int connec

/* main pr
int main(a

  char * h
  char * s

  if (argc
  if (argc

  int s =

  while (
    buf[n]
    (void)
  }
}
```

```
int connectTCP(const char * host , const char * service) {
  struct sockaddr_in sin; /* Internet endpoint address */
  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;

  /* Map service name to port number */
  if (struct servent * pse = getservbyname(service, “tcp”) )
      sin.sin_port = pse->s_port;
  else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
      errexit(“can’t get <%s> service entry\n”, service);

  /* Map host name to IP address, allowing for dotted decimal */
  if (struct hostent * phe = gethostbyname(host) )
      memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
  else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
      errexit(“can’t get <%s> host entry\n”, host);

  /* Allocate socket */
  int s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0) errexit(“can’t create socket: %s\n”, strerror(errno));

  /* Connect the socket */
  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      errexit(“can’t connect to %s.%s: %s\n”, host, service, strerror(errno));

  return s;
}
```

Server Software Design: Issues [Comer]

- **Concurrent** vs. **Iterative** Servers:

  The term **concurrent server** refers to whether the server permits multiple requests to proceed concurrently, **not** to whether the underlying implementation uses multiple, concurrent threads of execution.

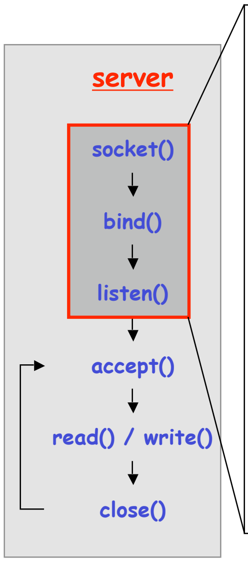
  Iterative server implementations are easier to build and understand, but may result in poor performance because they make clients wait for service.
- **Connection-Oriented** vs. **Connectionless** Access:

  Connection-oriented (TCP, typically) servers are easier to implement, but have resources bound to connections.

  Reliable communication over UDP is not easy!
- **Stateful** vs. **Stateless** Servers:

  How much information should the server maintain about clients? (What if clients crash, and server does not know?)

Example: **Iterative**, Connection-Oriented Server

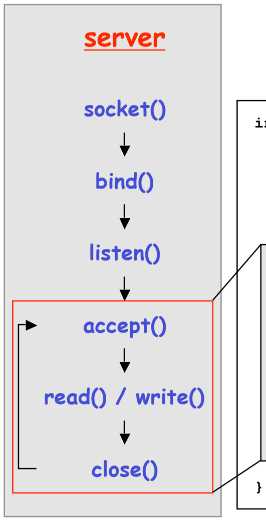


server

socket() → bind() → listen() → accept() → read() / write() → close()

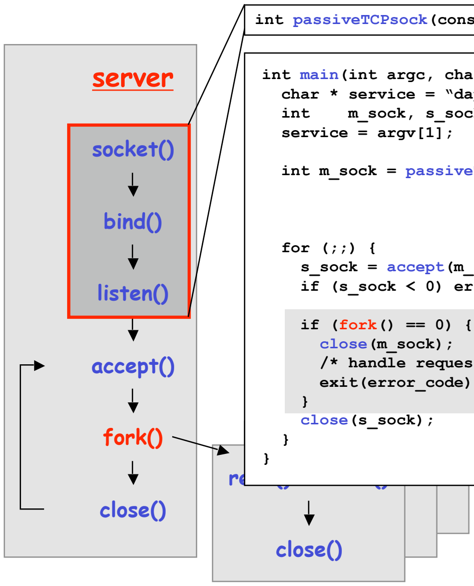
```
int passiveTCPsock(const char * service, int backlog) {

  struct sockaddr_in sin;          /* Internet endpoint address */
  memset(&sin, 0, sizeof(sin));    /* Zero out address */
  sin.sin_family      = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;

  /* Map service name to port number */
  if (struct servent * pse = getservbyname(service, "tcp") )
      sin.sin_port = pse->s_port;
  else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
      errexit("can't get <%s> service entry\n", service);

  /* Allocate socket */
  int s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0) errexit("can't create socket: %s\n", strerror(errno));

  /* Bind the socket */
  if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
       errexit("can't bind to …\n");

  /* Listen on socket */
  if (listen(s, backlog) < 0)
       errexit("can't listen on …\n")

  return s;
}
```

Example: **Iterative**, Connection-Oriented Server



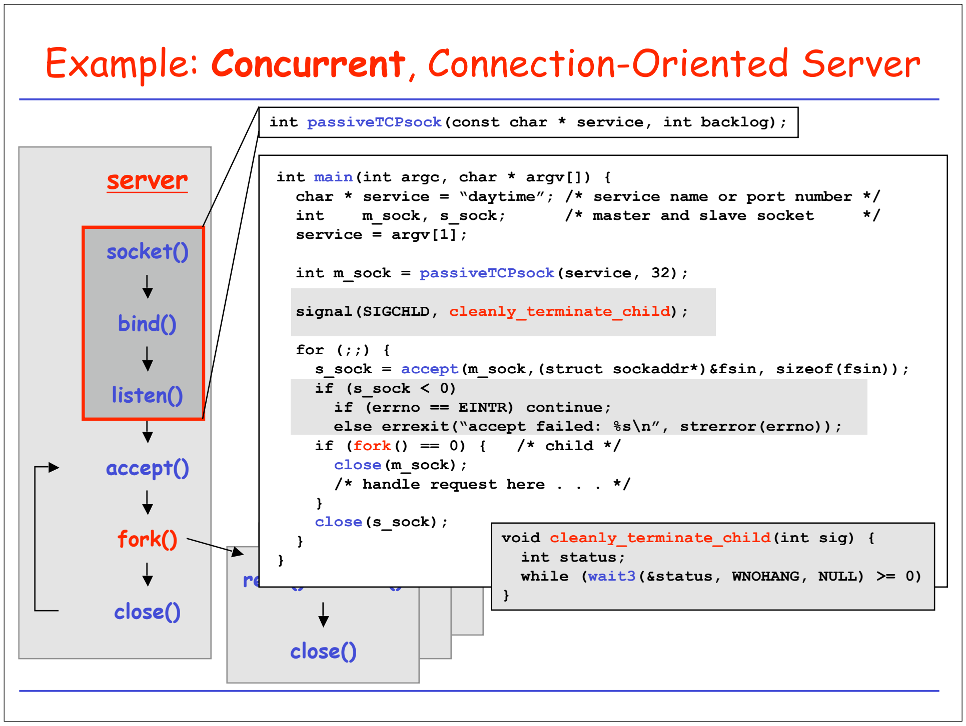
```
int main(int argc, char * argv[]) {
  char * service = "daytime"; /* service name or port number */
  int    m_sock, s_sock;      /* master and slave socket     */
  service = argv[1];

  int m_sock = passiveTCPsock(service, 32);

  for (;;) {
    s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
    if (s_sock < 0) errexit("accept failed: %s\n", strerror(errno));

    time_t now;
    time(&now);
    char * pts = ctime(&now);
    write(s_sock, pts, strlen(pts));

    close(s_sock);
  }

}
```

Example: **Concurrent**, Connection-Oriented Server



```
int passiveTCPsock(const char * service, int backlog);
```

```
int main(int argc, char * argv[]) {
  char * service = "daytime"; /* service name or port number */
  int    m_sock, s_sock;      /* master and slave socket     */
  service = argv[1];

  int m_sock = passiveTCPsock(service, 32);


  for (;;) {
    s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
    if (s_sock < 0) errexit("accept failed: %s\n", strerror(errno));

    if (fork() == 0) {   /* child */
      close(m_sock);
      /* handle request here . . . */
      exit(error_code);
    }
    close(s_sock);
  }
}
```



Example: **Concurrent**, Connection-Oriented Server

server

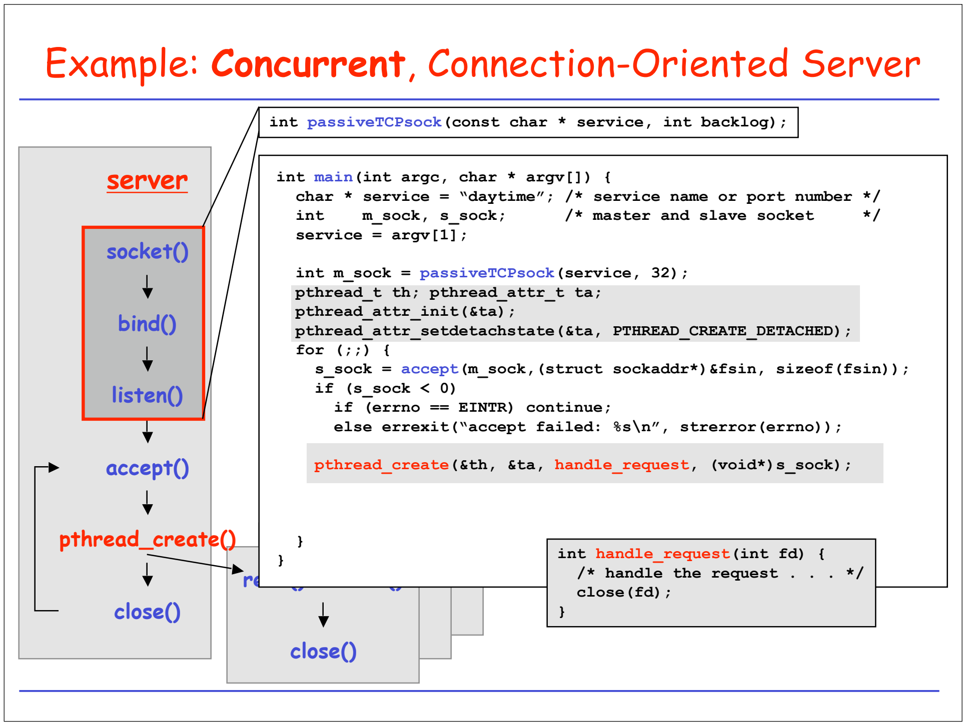
socket()
bind()
listen()
accept()
fork()
close()

close()

```
int passiveTCPsock(const char * service, int backlog);
```

```
int main(int argc, char * argv[]) {
  char * service = “daytime”; /* service name or port number */
  int    m_sock, s_sock;      /* master and slave socket     */
  service = argv[1];

  int m_sock = passiveTCPsock(service, 32);

  signal(SIGCHLD, cleanly_terminate_child);

  for (;;) {
    s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
    if (s_sock < 0)
      if (errno == EINTR) continue;
      else errexit(“accept failed: %s\n”, strerror(errno));
    if (fork() == 0) {    /* child */
      close(m_sock);
      /* handle request here . . . */
    }
    close(s_sock);
  }
}
```

```
void cleanly_terminate_child(int sig) {
  int status;
  while (wait3(&status, WNOHANG, NULL) >= 0)
}
```



Example: **Concurrent**, Connection-Oriented Server

server

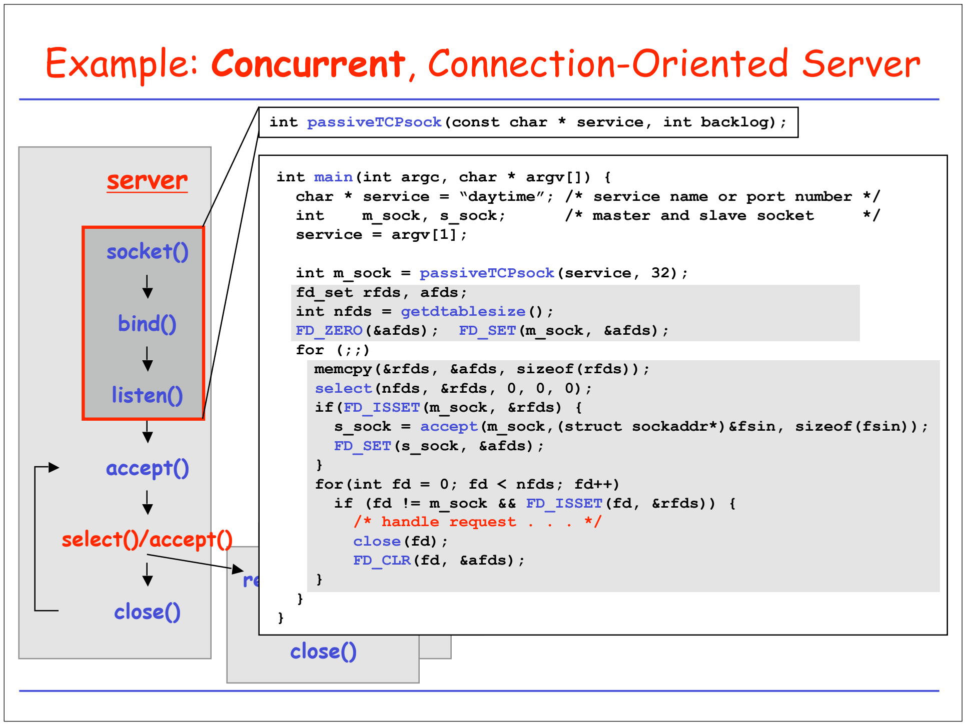
socket()
bind()
listen()
accept()
pthread_create()
close()

close()

```
int passiveTCPsock(const char * service, int backlog);
```

```
int main(int argc, char * argv[]) {
  char * service = “daytime”; /* service name or port number */
  int    m_sock, s_sock;      /* master and slave socket     */
  service = argv[1];

  int m_sock = passiveTCPsock(service, 32);
  pthread_t th; pthread_attr_t ta;
  pthread_attr_init(&ta);
  pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
  for (;;) {
    s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
    if (s_sock < 0)
      if (errno == EINTR) continue;
      else errexit(“accept failed: %s\n”, strerror(errno));

     pthread_create(&th, &ta, handle_request, (void*)s_sock);


  }
}
```

```
int handle_request(int fd) {
  /* handle the request . . . */
  close(fd);
}
```



Example: **Concurrent**, Connection-Oriented Server

server

socket()

bind()

listen()

accept()

select()/accept()

close()

re

close()

```
int passiveTCPsock(const char * service, int backlog);
```

```
int main(int argc, char * argv[]) {
  char * service = “daytime”; /* service name or port number */
  int    m_sock, s_sock;      /* master and slave socket     */
  service = argv[1];

  int m_sock = passiveTCPsock(service, 32);
  fd_set rfds, afds;
  int nfds = getdtablesize();
  FD_ZERO(&afds);  FD_SET(m_sock, &afds);
  for (;;)
    memcpy(&rfds, &afds, sizeof(rfds));
    select(nfds, &rfds, 0, 0, 0);
    if(FD_ISSET(m_sock, &rfds) {
      s_sock = accept(m_sock,(struct sockaddr*)&fsin, sizeof(fsin));
      FD_SET(s_sock, &afds);
    }
    for(int fd = 0; fd < nfds; fd++)
      if (fd != m_sock && FD_ISSET(fd, &rfds)) {
        /* handle request . . . */
        close(fd);
        FD_CLR(fd, &afds);
    }
  }
}
```