ALGONQUIN COLLEGE

MODULE 8: NETWORK PROGRAMMING

Professor : Dave Houtman

Office: T323

Office Hrs: Monday 15:30 – 16:00

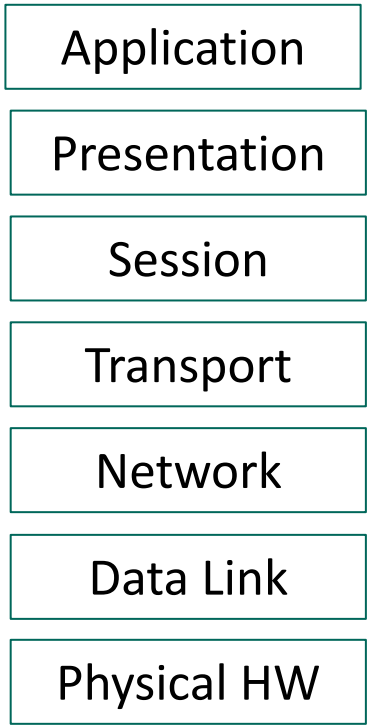
Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

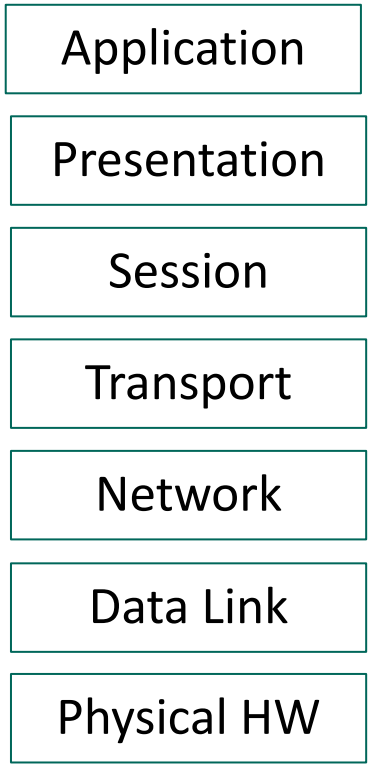
8.1 The ISO Model

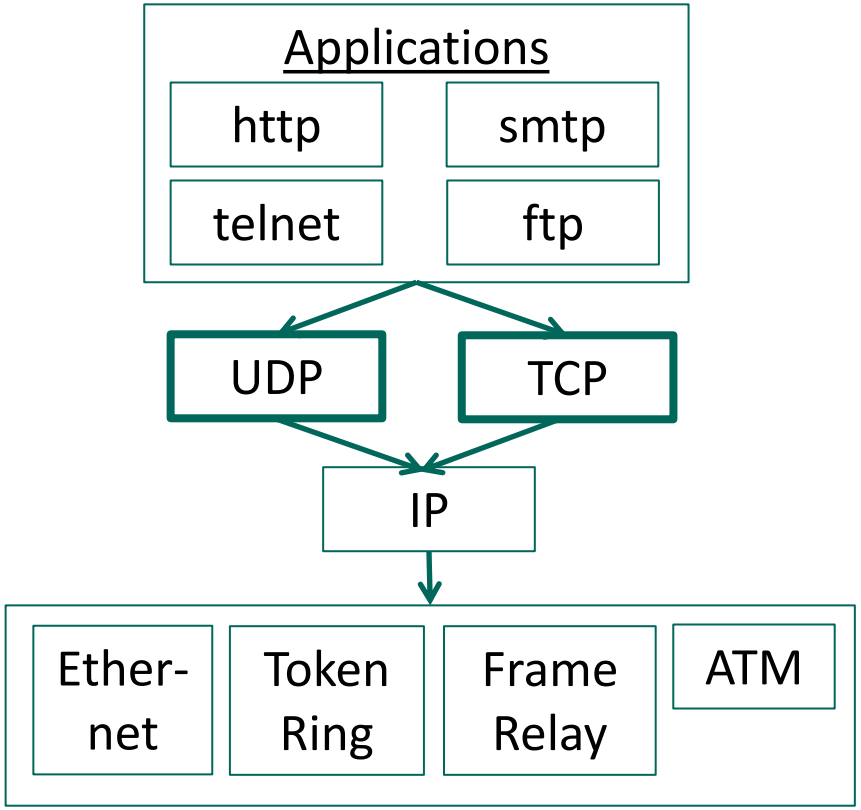
- The structure of the internet is based on a model commonly referred to as 'The ISO Model'. The earliest version of the model consists of a 7-layer structure, which is still used (although a 5-layer model is more appropriate for describing TCP/IP).



8.2 TCP/IP v. UDP

- Historically, two transport mechanisms have dominated the flow of traffic over the internet: UDP (the older of the two) and (more recently) TCP.





8.2 TCP/IP v. UDP

- The C language routines for internet communications are *extensive*; they allow you to build non-ISO protocols that use the physical hardware of the internet but have nothing to do with ISO standards

- While our C-based internet applications sit—appropriately—in the top layer(s) of the ISO stack, our code will need to specify certain features related to *how* the information is transmitted over the internet. Therefore an understanding of the differences between UDP and TCP/IP and client/server architecture is essential, since it effects how our programs are actually written. (Note however, that we don't need to worry about the details of packet handling; that *is* dealt with automatically by C's networking software)

You cannot write internet programs in C 'by rote';
you need to understand how all the pieces fit together
and what they do before you begin coding.

8.2 TCP v. UDP

TCP

- a *connection-oriented* protocol based; it is *peer-to-peer* or *end-to-end* communication.
- messages are guaranteed to arrive at their destination. The protocol itself deals handles all 'hand-shaking' and acknowledgements
- Information is read as streams. Therefore, there are no packet boundaries to worry about; information is contiguous, not fragmented
- Messages are ordered when they arrive
- Computes a checksum on data received to ensure it has not been corrupted
- 'Heavy-weight' and slow—but carries the bulk of internet traffic
- Examples: WWW, SMTP, FTP

UDP

- a *connection-less* protocol based on "*best-effort delivery*"; there is no guarantee that messages will arrive at their destination. It is the responsibility of the client to retransmit a message if it does not receive an acknowledgement from the target computer.
- Datagrams sent out as packets; size limited compared to TCP/IP
- Messages can arrive out of order
- Fast, efficient, and lightweight—still very useful in a variety of situations
- Examples: DNS, VoIP

8.3 Clients and Servers

- Technically, a 'client' or a 'server' is a piece of software, not a piece of hardware.
- In general, the application that initiates contact is considered the client; the application that awaits a request is the server. However, this distinction is not always clear-cut. Code that acts as a server in one case may need to act as a client in another, depending on the application.

8.3 Clients and Servers – The Client Model

- In a client/server architecture, clients are easier to implement that servers because:
  - The server needs to deal with concurrency issues. By comparison, the client can function as a 'stand-alone' piece of software. Also, concurrency means dealing with global variables, which are shared between different threads of the application.
  - The server needs to deal with issues such as security, privacy, and authentication
  - The server needs to access information protected by the OS, such as ports, files, and devices. Therefore, the server needs to have special privileges.
  - Servers may be *stateful*; that is, they 'remember' information about clients to speed communications. This, however, adds overhead to the code.

"The combination of special privileges and concurrent operations usually make servers more difficult to design and implement than clients"

- D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, Prentice-Hall, 2001, pg. 11

8.3 Clients and Servers – The Client Model

- Of the two transport protocols, TCP/IP is the easier one to implement in code because:

1. With UDP, the programmer needs to handle out-of-order arrival of data packets

2. Because of TCP's built-in error correction, testing connections is easier; there's less to worry about and less to deal with.

3. UDP works well provided the network is reliable, but it may give the programmer a false sense of security if code is tested on a LAN

"Beginners, as well as most experienced professionals, prefer TCP over UDP. TCP's connection-oriented style simplifies programming and TCP's reliability relieves the programmer of the responsibility for detecting and correcting errors"

- D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, Prentice-Hall, 2001, pg. 14

8.4 Network Programming Overview

- TCP/IP does not specify an API for network communications, however, there are minimum expected 'standards' for a networking interface, which must include such capabilities as:
  - Allocating resources for communications
  - Specifying local and remote endpoints
  - Initiating a connection
  - Sending a datagram (on the client side)
  - Waiting for incoming connections (on the server side)
  - Sending or receiving data
  - Handle error conditions
  - Terminate a connections gracefully

See Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 40, for a complete list

8.4 Network Programming Overview

- A ***socket*** is a software abstraction designed to handle network communications. Functionally, it is analogous to a file. Because of this, the set of functions that handle communications is known as the *socket API*.

- In gcc, the `socket.h` library is found in your file system under `usr/include/sys`. Thus, to include sockets in your program, you need to include the line:

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

  Additionally, certain predefined constants are found in the library

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

  which should also be included in any network program written in C.

- In the Windows OS, equivalent libraries are stored in `<winsock.h>`

8.4 Network Programming Overview—Sockets

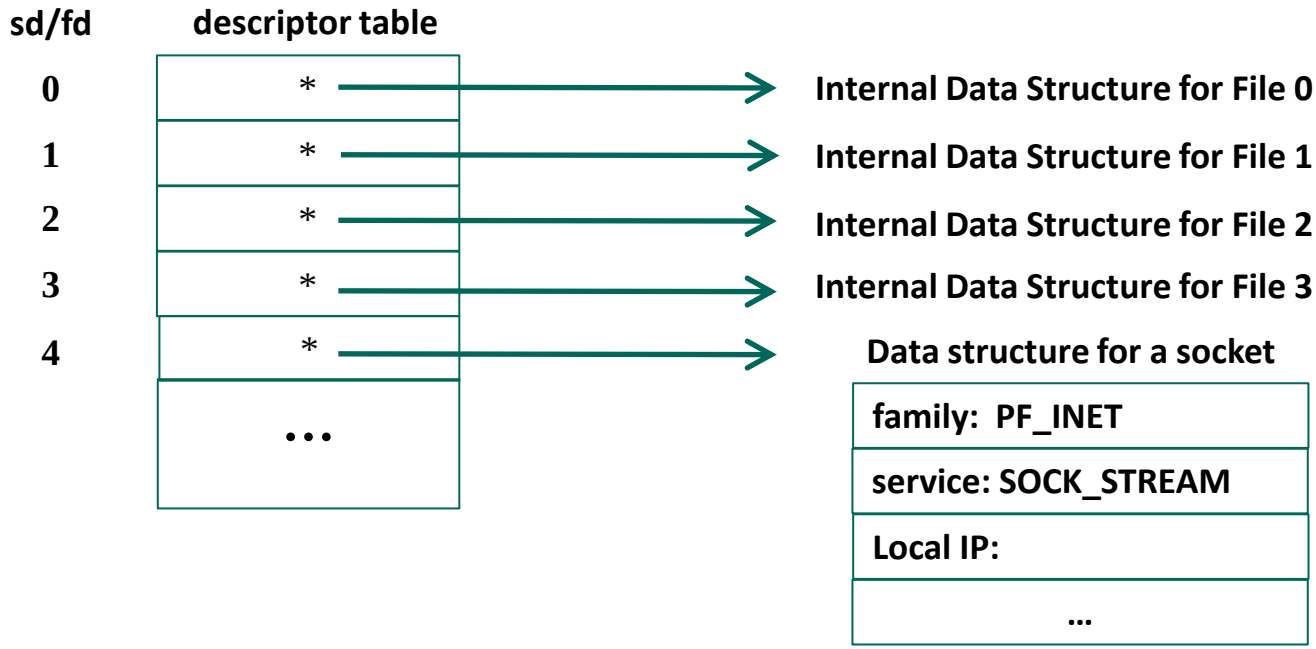
- *To implement the minimum TCP/IP client configuration we require no fewer than six libraries, five major functions to handle data transmission/reception, four lesser functions required to initialize these services, and approximately a half dozen structures to store information essential to the entire process.*

- The socket API was written in a way that does not favor internet standards in particular. In practice, this means that to set a socket for TCP/IP use, there are extra values that need to be established to ensure that our I/O follows TCP/IP standards.

- All functions involving sockets rely on the programmer first obtaining a ***socket descriptor*** which, like a file descriptor, is a numerical handle, this time to a socket rather than a file.

8.5 The Socket Interface

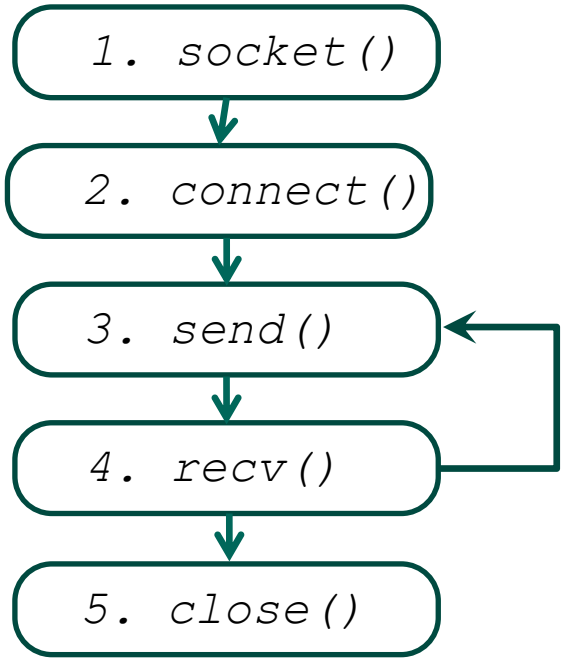
- The mechanics of the socket descriptor follow that of the file descriptor in section 7.9. Essentially, sockets are treated in exactly the same way as files. In fact, they use the same descriptor table as do files. But now the descriptor table pointer points to a data structure for a *socket*, not a data structure for a *file* (both being declared internally by the OS.)



From: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001

8.4 Network Programming Overview

- A simple diagram for client-side operations shows the sequence of events required to initiate contact, transmit a request to the server, and receive information back:



Create a socket in the current device. The socket descriptor will handle all network communications

Connect to a server

Send packets to the server

Receive information, and continue transferring data until finished

Close the connection when finished

From Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 59

8.5 The Socket Interface – 1. `socket()`

`socket()`

- When a socket is created, it does not contain any information about how it will be used. In fact, the software for network programming is so general-purpose that you can essentially create non-ISO compliant protocols for your own use, and fill in the socket details that correspond to your non-ISO standard.

- Because different protocols will generally implement communications in different ways, `socket()` allows the programmer to use different *protocol families*. For our purposes, we will always be using TCP/IP, which uses a standard predefined constant value, **`PF_INET`**.

- Also because of this flexibility, it is necessary to specify the type of transport to be used. For TCP, we use the symbolic constant **`SOCK_STREAM`** (for UDP, we'd use `SOCK_DGRAM`).

8.5 The Socket Interface – 1. `socket()`

To set up a `socket()` for a TCP/IP client, use the following code:

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

int sd;  // holds the socket descriptor

sd = socket(PF_INET, SOCK_STREAM, 0);
    …
```

The protocol value.
Use 0 for TCP

This code sets up the socket for future use. At this point, we haven't done anything with it yet, merely declared our intentions to the OS: this is the equivalent of plugging in a telephone prior to making a call.

- Note the parallels between `socket()` and `open()`:

```
int fd = open(char *fname, int access, int permis);
```

8.5 The Socket Interface – 2. `connect()`

`connect()`

- `connect()` does four things:

1. It checks the socket descriptor (`sd`) to ensure that the socket is valid and is not already in use;
2. It uses information about the remote address (the server, in our case) stored in the `sockaddr_in` structure and puts it into the socket structure using the value of **`sd`**;
3. It determines the local address information if the socket does not come with one automatically;
4. It initiates the TCP connection and tells the programmer when that connection is valid, returning -1 otherwise.

8.5 The Socket Interface – 2. `connect()`

- *Before you can use* `connect()`, you must define a ***communication endpoint*** first. A communications endpoint consists of:
  - A ***protocol port number***—which hardware port will be used for sending/receiving information?
  - An ***IP address***—where will you be sending the information to?

  This is really the first thing your program should do, but since it is associated with establishing a connection, we've deferred it until now.

- In general, endpoints may be *local* (with the client) or *remote* (with the server).

- The communication endpoint information is stored in a structure called **`socketaddr_in`**. A pointer to this structure must be passed to `connect()` before connect can do anything.

8.5 The Socket Interface – 2. `connect()`

- The format of `connect()` is:

```
retcode = connect(sd, remaddr, remaddrlen);
```

where

`retcode` returns -1 if the attempt to connect fails, 0 otherwise;

`sd` is the socket descriptor obtained from `socket()`;

`remaddr` is the address of a structure of type `sockaddr_in`; this stores the communications endpoint information;

`remaddrlen` is the length, in bytes, of `sockaddr_in`.

Why do you need `remaddrlen`? The socket API allows for a wide variety of possible implementations: `remaddr` can point to differently-sized structures. `remaddrlen` tells `connect()` how much space it will need in order to hold the `sockaddr_in` structure.

8.5 The Socket Interface – 2. `connect()`

- The structure of `sockaddr_in` is important, since certain of its values must be determined before its address can be passed to `connect()`. (Note: **`sin`** = **s**ocket **in**terface.)

```
struct sockaddr_in {
    u_char sin_len;          //total length
    u_short sin_family;      //type of address
    u_short sin_port;        //target port number
    struct in_addr sin_addr; //IP address
    char sin_zero[8];        // unused, set to 0
};
```

where `u_char` and `u_short` are assumed typedef'd to an `unsigned char` and an `unsigned short` respectively.

8.5 The Socket Interface – 2. `connect()`

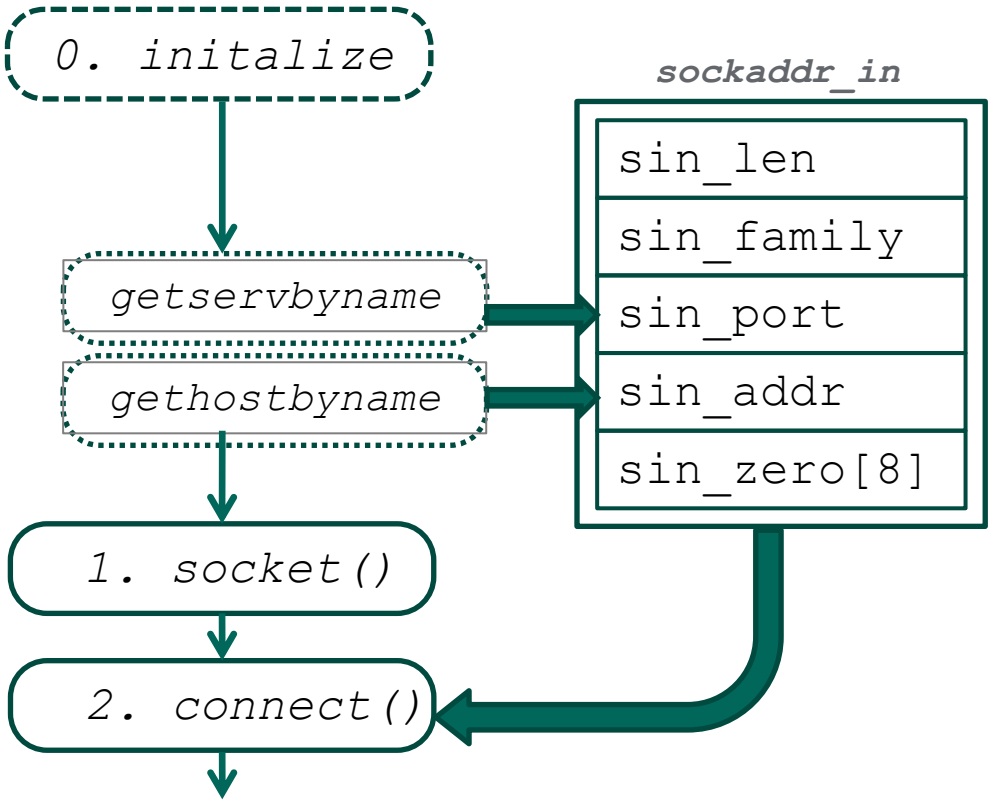
- Of the five members in the `sockaddr_in` structure, the first two members are easily set, while the final member is unused. The members `sin_port` and `sin_addr`—itself a structure—must each be determined by special functions, called `getservbyname()` and `gethostbyname()`.

```
struct sockaddr_in {
    u_char sin_len;           //total length
    u_short sin_family;       //type of address
    u_short sin_port;         //use getservbyname
    struct in_addr sin_addr; //use gethostbyname
    char sin_zero[8];         //unused, set to 0
};
```

From Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 59

8.5 Network Programming Overview

- So our initial schematic diagram of the flow of execution needs to be modified somewhat:



Get communication endpoint information

Create a socket in the current device. The socket descriptor will handle all network communications

Connect to a server

8.5 The Socket Interface – 2. `connect()`

Question: Why do we need these two functions?

1. `connect()` needs a communications endpoint to tell it *what to connect to*.

2. A communications endpoint requires both a service port (like 80 for the web) and the host IP (like 173.206.169.70). In general, you won't know these in advance. So these two functions are responsible for returning this crucial information.

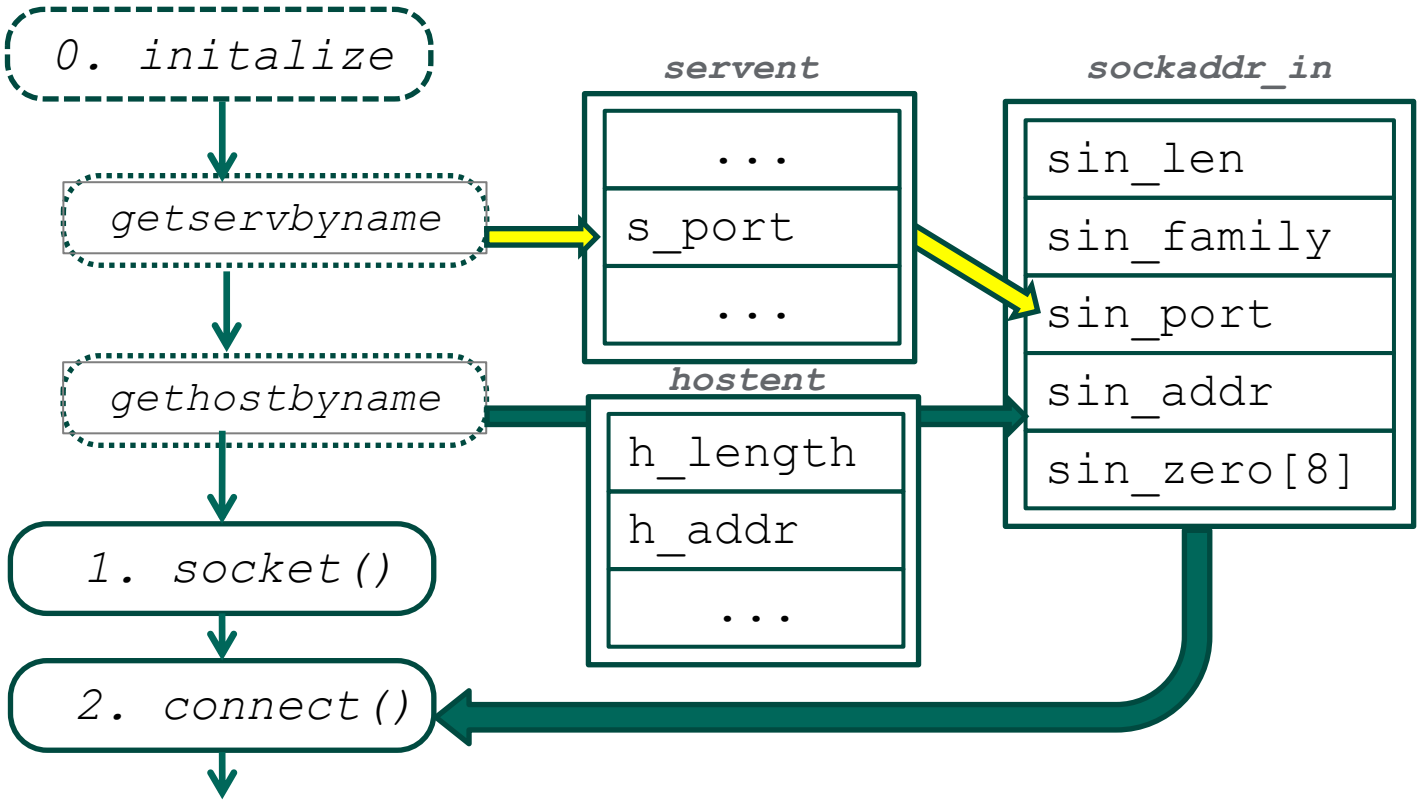
8.5 The Socket Interface – 2. `connect()`

`getservbyname()` and ***`gethostbyname()`***

- These two functions are members of a family of utility functions that collectively return specific information related to a protocol, transport, or service. Such utility functions typically return either a value, structure or a pointer to a particular structure, and these structures will contain the information we are interested in. (The details need not concern us.) For example, `getservbyname()` takes as arguments a specific service ("smtp") and transport ("tcp"), and returns the official protocol port number inside a pointer to a structure of type `servent`

8.5 Network Programming Overview

- So our initial schematic diagram needs to be modified further:



8.5 The Socket Interface – 2. `connect()`

`getservbyname()`

- The format of `getservbyname()` is

```
struct servent *pse =
        getservbyname(char* service, char* transport);
```

where:

`pse` is a pointer to a **serv**ice information **ent**ry; it returns `NULL` if the function fails;

`servent` is a structure containing the member **`s_port`**, which we need for **`sockaddr_in`**;

`service` is a pointer to a string associated with the desired port, e.g. "http", "ftp", "smtp"…

`transport` is a pointer to a string, either "tcp" or "udp"

8.5 The Socket Interface – 2. `connect()`

`getservbyname()`

- To get the protocol port number used by "http" over "tcp", use

```
struct servent *sptr;      // pointer to servent

if (sptr = getservbyname("http", "tcp")
   // sptr->s_port now contains port number
```

Most operations involving network programming deal with structures. However, it is rarely necessary to initialize *every* member of a structure in order to implement the code properly.

(This is not unlike what happens with classes in Java that need to be passed to other methods.)

8.5 The Socket Interface – 2. `connect()`

- We wish to load the value of the member `sin_port` in the structure `sockaddr_in` with the value of the member `s_port` in the structure `servent`, which is in turn returned by `getservbyname()`.

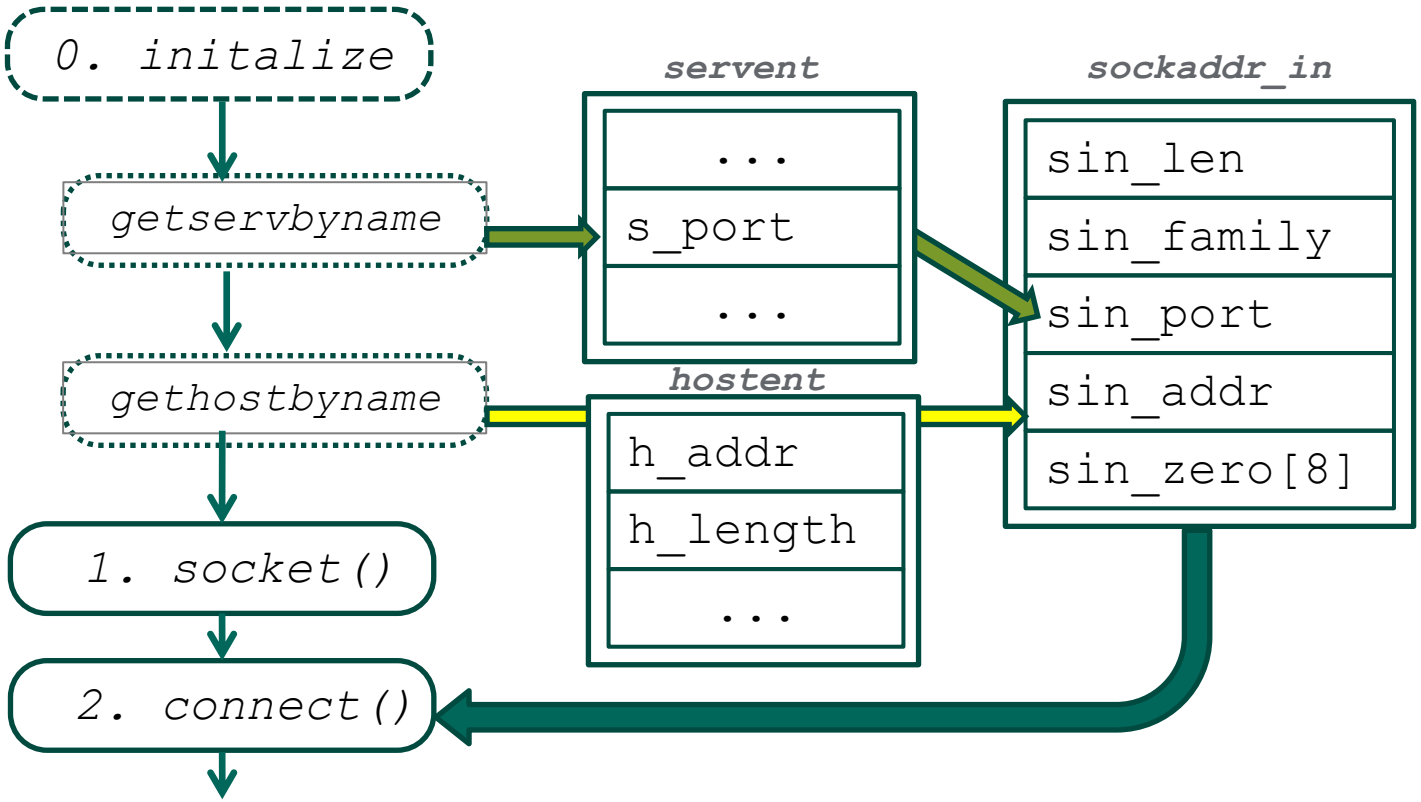
```
struct servent *pse;
struct sockaddr_in sin;

if (pse = getservbyname(service, transport))
   sin.sin_port = pse->s_port;
```

Note that, whenever a function such as `fopen()` or `getservbyname()` returns a pointer to a structure, it will *almost* always have malloc'ed the space being pointed to, *and* filled it with the requested information. (`malloc` itself is an exception; it returns a pointer to reserved space, but doesn't load anything into it.)

8.5 Network Programming Overview

- The second utility function needed to initialize a member of `sockaddr_in` is `gethostbyname()`.



8.5 The Socket Interface – 2. `connect()`

Reminder: Why are we doing this?

1. We need to initialize the structure `sockaddr_in` so that the function `connect()` can establish a connection with a host (i.e. server)
2. We've already set one of these values, the service port, using the function `getservbyname()`
3. We now wish to set another value in `sockaddr_in`, corresponding to the IP address. But this value is, itself, a structure:

```
struct sockaddr_in {
    u_char sin_len;          //total length
    u_short sin_family;      //type of address
    u_short sin_port;        //use getservbyname
    struct in_addr sin_addr; //use gethostbyname
    char sin_zero[8];        //unused, set to zero
};
```

8.5 The Socket Interface – 2. `connect()`

- In fact, `sin_addr` has one member only: it stores the IP address as a 32-bit unsigned long. If `sin` is declared as a `sockaddr_in` structure as before, we can access this value using:

```
sin.sin_addr
```

- Since the actual IP address of a site is not usually known to us in advance, we need yet another utility function that will return a structure containing the 32-bit IP address. This function is `gethostbyname()`, and it returns a pointer to a `hostent` structure, which contains the member `h_addr`. Thus, assuming `phe` is a pointer to a `hostent` structure, to access this member you'd use:

```
phe->h_addr
```

8.5 The Socket Interface – 2. `connect()`

`gethostbyname()`

- The format of `gethostbyname()` is

```
struct hostent *phe = gethostbyname(char* host);
```

where:

`phe` is a pointer to a **host** information **ent**ry; it returns `NULL` if the function fails;

`hostent` is a structure containing the member **`h_addr`**, which we need for **`sockaddr_in`**;

`host` is a pointer to a string associated with the target server, e.g. "www.google.com"

8.5 The Socket Interface – 2. `connect()`

- An example of `gethostbyname()` is

```
struct hostent *hptr;
char *targetServer = "www.google.ca";

if (hptr = gethostbyname(targetServer)){
   printf("IP address of %s is %s", targetServer,
                  inet_ntoa(hptr->h_addr));
}
```

where `inet_ntoa()` is a function that converts a unsigned long to a string in the form 173.206.169.70

8.5 The Socket Interface – 2. `connect()`

- We wish to load the value of the member `sin.s_addr` in the structure `sockaddr_in` with the value of the member `h_addr` in the structure `hostent`, which is in turn returned by `gethostbyname()`. If this was as straightforward as before we'd write something like:

```
sin.sin_addr = phe->h_addr;
```

8.5 The Socket Interface – 2. `connect()`

- There is one catch, however. `hostent` is a defined as

```
struct hostent{
  char *h_name;       /* official host name  */
  char **h_aliases;   /* other aliases       */
  int h_addrtype;     /* address type        */
  int h_length;       /* address length      */
  char **h_addr_list; /* list of address     */
}
```

with **`h_addr`** is defined as:

```
#define h_addr h_addr_list[0]
```

8.5 The Socket Interface – 2. `connect()`

- In other words, **`h_addr_list`** can be thought of as a pointer to strings, and **`h_addr`** is defined as the address of the first string. Therefore, you can't set

```
sin.sin_addr = phe->h_addr;    //WRONG!
```

…you have to copy the string pointed to by address **`h_addr`** to the address corresponding to **`sin.s_addr`**. This can be done as follows:

```
memcpy(&sin.sin_addr, phe->h_addr,
       phe->h_length);
```

8.5 The Socket Interface – 2. `connect()`

- So the code to load the IP address into the `sin.s_addr` of `sockaddr_in`;

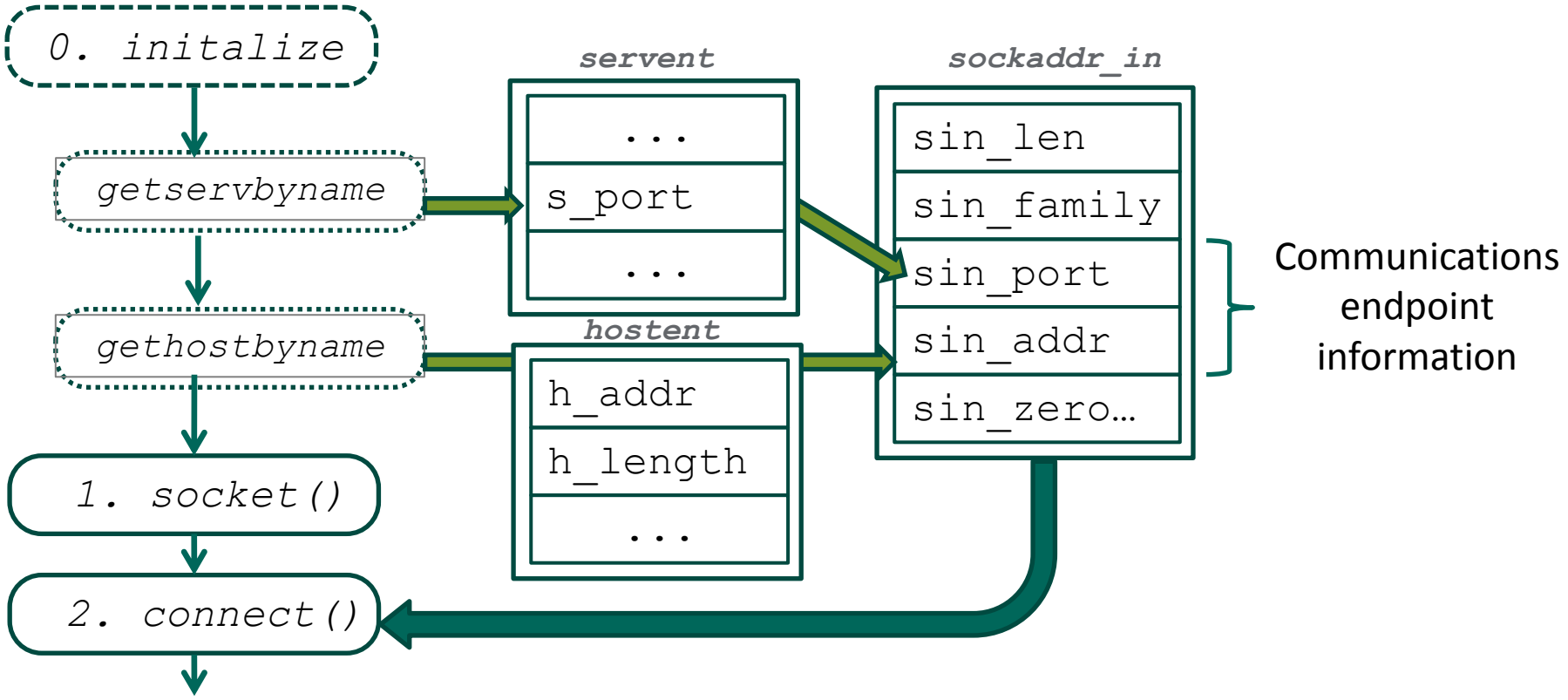
```
struct hostent *phe;
struct sockaddr_in sin;

if (pse = gethostbyname(host))
        memcpy(&sin.sin_addr, phe->h_addr,
                phe->h_length);
```

where `memcpy()` copies the number of bytes in the third argument from the second argument address to the first.

8.5 Network Programming Overview

- To summarize: a ***communications endpoint*** consists of a ***protocol port number*** and ***IP address***. Once we have a socket descriptor *and* the `sockaddr_in` structure has been initialized with the endpoint information, we are ready to use `connect()` to talk to the server.



8.5 The Socket Interface – 2. `connect()`

- Recall that the format of `connect()` is:

```
retcode = connect(sd, retddr, retaddrlen);
```

where

`retcode` returns -1 if the attempt to connect fails, 0 otherwise;

`sd` is the socket descriptor obtained from `socket()`;

`retddr` is the address of a structure of type `sockaddr_in`;

`retaddrlen` is the length, in bytes, of `retaddr`.

8.5 The Socket Interface – 2. `connect()`

- Of the three arguments required by `connect()`:

> **`sd`** is the socket descriptor obtained from `socket()`;
> **`retddr`** is the address of a structure of type `sockaddr_in`;
> **`retaddrlen`** is the length, in bytes, of `sockaddr`.

➡ **`sd`** is our socket descriptor; it was returned by `socket()`;

➡ **`&sin`** is the address of our `sockaddr_in` structure

➡ **`sizeof(sin)`** is the size of the `sockaddr_in` structure

8.5 The Socket Interface – 2. `connect()`

- Hence the call to `connect()` will be

```
if (connect(sd, &sin, sizeof(sin)) < 0)
    // output some error,
    // else, you're connected
```

8.5 The Socket Interface – 3. `send()`

- Compared with `connect()`, the call to `send()` is simplicity itself:

```
void send(int sd, char* req, int reqleng,0)
```

where:

`sd` is the socket descriptor;

`req` is a pointer to a string containing the information to be sent to the server;

`reqleng` is the size of the string to the server;

The final '0' is the default used for TCP/IP communications

Assuming **`req`** is a string set to e.g. "GET" or "POST", then

```
send(sd, req, strlen(req), 0);
```

Is a valid `send` command to the server.

8.5 The Socket Interface – 4. `recv()`

- The **`receive()`** command is trickier, since TCP/IP returns a stream of data which may appear in the buffer a few characters at a time. The format is:

```
int n = recv(int sd, char* bptr, int bufleng,0)
```

where:

`n` is the number of bytes received on each call to **`recv()`**;

`sd` is the socket descriptor;

`bptr` is a pointer to an array of chars that stores the information received from the server;

`bufleng` is the size of the array that stores the information;

The final '0' is the default used for TCP/IP communications

8.5 The Socket Interface – 4. `recv()`

- An example of a simple **`send()`**/**`receive()`** combination (assuming connectivity has already been established) is:

```
#define BLEN 1200          //buffer size to use
char buf[BLEN];            //buffer to hold response
int n, buflen = BLEN       //holds sizes
char *bptr = buf, *req = "POST something";
...

send(sd, req, strlen(req), 0);

while ((n=recv(sd, bptr, buflen, 0)) > 0){
      bptr += n;   // ptr to next available space
      buflen -=n;  // decrease buf size available
}
```

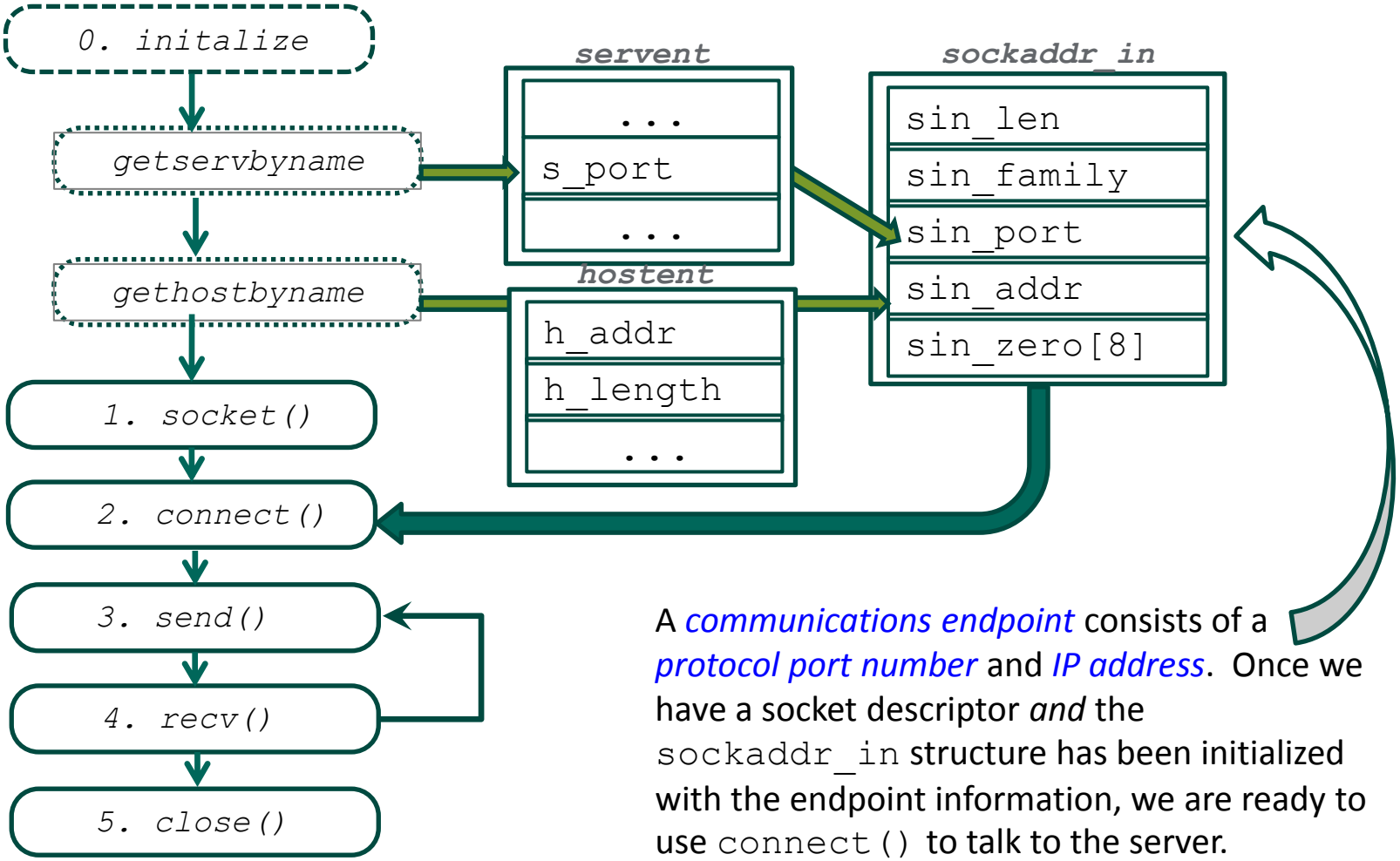
8.5 The Socket Interface – 5. `close()`

- When the message from the server is fully received and the connections is no longer necessary, `close()` terminates the connection and deallocates the socket descriptor.

```
close(sd);
```

The Socket Interface – Summary

- The complete structure of a socket-based TCP/IP client looks like this:



A *communications endpoint* consists of a *protocol port number* and *IP address*. Once we have a socket descriptor *and* the `sockaddr_in` structure has been initialized with the endpoint information, we are ready to use `connect()` to talk to the server.

The Socket Interface – Summary

This can be summarized in the following algorithm:

1. Find the IP address and protocol port number of the server at the remote communications endpoint;
2. Allocate a socket;
3. Specify that the connections needs an arbitrary, unused protocol port on the local machine, and allow TCP to choose one;
4. Connect the socket to the server;
5. Transmit a request to the server;
6. Fill a buffer with the data received until bytes are no longer returned; and
7. Close the connection.

Taken, with modifications from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 70.

8.6 Network Programming Summary - Notes

- Despite its name, `servent` has nothing to do with the server. Remember, the 'serv' stands for services.
- And curiously, the 'host' in `hostent` is not the current client, but the server it is talking to. For most purposes, a 'host' and a 'server' are the same thing.
- Only the members of structures actually used directly or indirectly by `connect()` need to be initialized
- The code shown above is generally considered the minimum required to establish a connection. Examples with less error checking are more vulnerable to failure.

The Socket Interface – Example

Setting up a socket and connecting to the internet using TCP/IP typically requires functions found in no fewer than six (fairly standard) libraries:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
```

Additionally, our code will need three other familiar libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

The Socket Interface – Example

- We need to create a communications endpoint for use in the `connect()` function. We therefore will require a `sockaddr_in` structure, along with pointers to the two other 'support' structures' that supply endpoint information:

```
struct sockaddr_in sin;//comm endpoint info
struct servent *pse;   //servent holds s_port
struct hostent *phe;   //hostent holds h_addr
```

Additionally, our code will need variables to hold the socket descriptor and the socket type:

```
int sd, type;
```

The Socket Interface – Example

- Furthermore, we'll need to set up some variables to hold the information received from the host/server, along with the actual message to send:

```
#define BLEN 120      //buffer size to use
char buf[BLEN];       //buffer to hold response
int n, buflen = BLEN //holds sizes
char *bptr, *req = "GET / HTTP/1.1\r\n\r\n";
 bptr = buf;
```

The Socket Interface – Example

- Since we have a `sin` to hold communications endpoint information, we can initialize members of this structure directly, prior to connect. In the `sockaddr_in` structure, the first two members, `sin_length` and `sin_family` can be set directly:

```
sin.sin_length = sizeof(sin);
sin.sin_family = AF_INET;
```

where **`AF_INET`** is a symbolic constant that specifies that the addresses used will be following the TCP/IP convention. This value is similar to **`PF_INET`**, used as an argument to the `socket()` call, and is often confused with it. (They are defined as being equal to the same value integer value, so confusing the two doesn't affect the code).

The Socket Interface – Example

- To set the next two members of the `sockaddr_in` structure, we need to call our two utility functions: `getservbyname()` and `gethostbyname()`. The former returns a pointer to a `servent` structure, from which we get the **`s_port`** value:

```
if (pse = getservbyname(service, transport))
        sin.sin_port = pse->s_port;
else
        // error message: can't get service,
        // and exit
```

The Socket Interface – Example

- Similarly, we need to get the `h_addr` out of the `hostent` structure; this is returned by `gethostbyname()`:

```
if (phe = gethostbyname(host))
  memcpy(&sin.sin_addr, phe->h_addr, phe-> h_length);
else
  // error message: can't get host,
  // and exit
```

`memcpy()` is needed because `h_addr` is really the address of the 0th element of an array. Thus rather than copy the value stored *in* `h_addr` to `sin.sin_addr`, we need `memcpy()` to copy the *address* stored in `h_addr` to the address of the `sin_addr` member.

The Socket Interface – Example

- This completes the 'regular' business of initializing the `sockaddr_in` structure prior to passing it to `connect()`. We can now execute the five 'standard' functions needed to implement a (still-simplified) TCP/IP client:

```
sd = socket(PF_INET, SOCK_STREAM, 0);
```

where the final argument is the protocol value. This can reliably be set to 0 for TCP/IP purposes. But here too there is a function to determine this value, if we require it. Assuming

```
struct protent *ppe   //like servent and hostent
```

we can set `ppe` using

```
ppe = getprotobyname(transport);
```

and then the call to socket becomes:

```
sd = socket(PF_INET, SOCK_STREAM, ppe->p_proto);
```

The Socket Interface – Example

- If **sd** is not equal to -1, we can proceed

```
if (sd < 0)
  // signal socket failure and exit
if (connect(sd, &sin, sizeof(sin)) <0)
  // output some error and exit
else {
  send(sd, req, strlen(req), 0);
  while ((n=recv(sd, bptr, buflen, 0)) > 0){
    bptr += n;// ptr to next available space
    buflen -=n; // decrease buf size availbl.
}
close(sd);
```

The Socket Interface – Variations

- Because the socket interface allows for so many possible ways to connect a client to a server, the following variations may be encountered.
- First, while `sockaddr_in` is the most appropriate structure to use for holding the communications endpoint using TCP/IP, a second, smaller structure is sometimes used. It has the format:

```
struct sockaddr {    //alternative to sockaddr_in
    u_char sa_len;          // total length
    u_short sa_family;      // type of address
    char sa_data[14];       // value of address
};
```

The Socket Interface – Variations

- Second, it is not uncommon to see short-cuts in code, like this:

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

  struct sockaddr_in host;
  char *message, host_reply[2000];
  int sd = socket(AF_INET,  SOCK_STREAM , 0);
  if (sd == -1)
        printf("Could not create socket"); // and exit

  host.sin_addr.s_addr = inet_addr("74.125.235.20");
  host.sin_family = AF_INET;
  host.sin_port = htons( 80 );
  if (connect(sd , (struct sockaddr_in*)&host,
                                    sizeof(host)) < 0){
          puts("connect error");
          return 1;
  } ...
```

The Socket Interface – Variations

```
    ...
    puts("Connected\n");
    message = "GET / HTTP/1.1\r\n\r\n";
    if( send(sd , message , strlen(message), 0) < 0){
            puts("Send failed");
             return 1;
    }
    puts("Data Send\n");
    if( recv(sd, server_reply, 2000 , 0) < 0) {
            puts("recv failed");
    }
    puts("Reply received\n");
    puts(server_reply);
    return 0;
}
```

Taken with modifications from: http://www.binarytides.com/socket-programming-c-linux-tutorial/

The Socket Interface – Variations

The output is:

```
Connected

Data Send

Reply received

HTTP/1.1 302 Found
Location: http://www.google.co.in/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=0edd21a16f0db219:FF=0:TM=1324644706:LM=1324644706:S=z6hDC9cZfGEowv_
Date: Fri, 23 Dec 2011 12:51:46 GMT
Server: gws
Content-Length: 221
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.in/">here</A>.
</BODY></HTML>
```

From: http://www.binarytides.com/socket-programming-c-linux-tutorial/

The Socket Interface – Example – Time of Day Services

"daytime" is a service, like "http" but much simpler. It returns the time of day in standardized format. The code that accesses this service starts with the libraries needed along, a few extra 'buffer' variables:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
```

The Socket Interface – Example – Time of Day Services

```
#define BLEN 120      // buffer size to use
#define LINELEN 128   // holds daytime info
char buf[BLEN];       // buffer to hold response
int n, buflen = BLEN // holds sizes
char *bptr, *req = "GET / HTTP/1.1\r\n\r\n";
 bptr = buf;

int sd;

struct sockaddr_in sin;// comm port info
struct servent *pse;   // servent holds s_port
struct hostent *phe;   // hostent holds h_addr
void connectTCP();     // forward reference to
                       // socket software
```

The daytime service itself is forward referenced as

```
void TCPdaytime(char* host, char* service);
```

The Socket Interface – Example – Time of Day Services

In `main`() itself, we only need to connect to the socket and call the service:

```
int main(void){
  connectTCP();
  TCPdaytime("localhost", "daytime");
}
```

The Socket Interface – Example – Time of Day Services

where connectTCP() is just:

```
void connectTCP()
        sin.sin_length = sizeof(sin);
        sin.sin_family = AF_INET;

        if (pse = getservbyname(service, transport))
                sin.sin_port = pse->s_port;
        else
                errorexit("Couldn't get service");

        if (phe = gethostbyname(host))
                memcpy(&sin.sin_addr, phe->h_addr,
                        phe-> h_length);
        else
                errorexit("Couldn't get host");
```

The Socket Interface – Example – Time of Day Services

```
sd = socket(PF_INET, SOCK_STREAM, 0);

if (sd < 0)
  errorexit("Couldn't get socket");
else {
  send(sd, req, strlen(req), 0);
  while ((n=recv(sd, bptr, buflen, 0)) > 0){
    bptr += n;// ptr to next available space
    buflen -=n; // decrease buf size availbl.
}
close(sd);
```

The Socket Interface – Example – Time of Day Services

In `TCPdaytime`, we handle the file I/O business to the screen

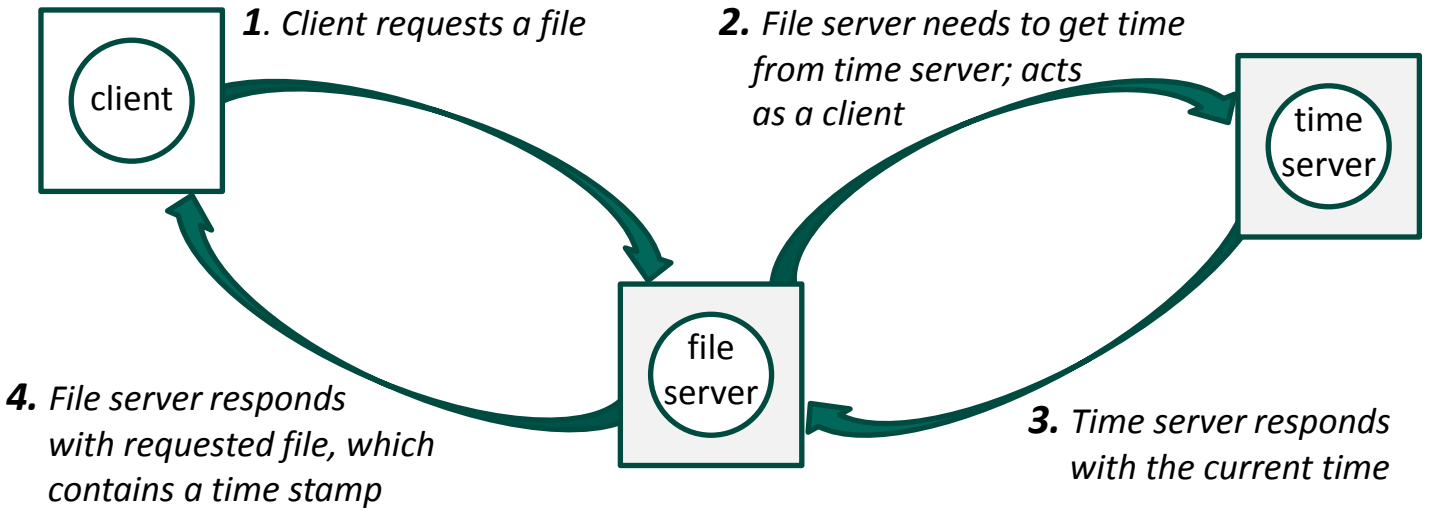
```
void TCPdaytime(char* host, char* service);
      while ((n = read(sd, buf, LINELEN) > 0){
            buf[n] = '\0';
            (void)fputs(buf, stdout);
      }
}
```

8.7 Server Issues – Statefullness

- A server may choose to maintain information about the clients to which it is connected. This information is called ***state information***, and it allows servers to be more efficient, leading to faster communications.
- A server that maintains this information is considered to be ***stateful***, while servers that do not maintain this information are considered to be ***stateless***.

8.7 Server Issues – Servers as Clients

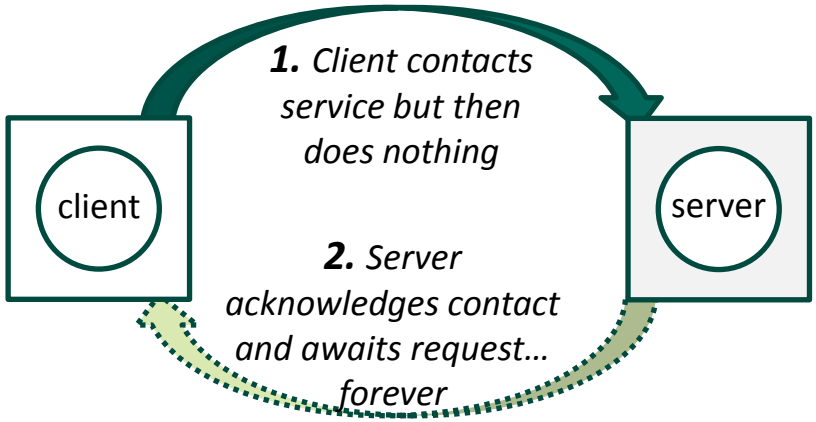
- The distinction between clients and servers is somewhat artificial, since most servers, at some time, need to behave as clients. In the following example, a server that requires time-of-day services to complete a request from a client must itself become a client before it can complete its task. Thus, *the server software will generally need to have client software built into it*.



Adapted from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 19

8.7 Server Issues – Deadlock and Denial-of-Service

- Finally, note that in contrast to a client, which (presumably) gets turned off at the end of the day, a server must be designed to run forever. It must handle resources and deallocate them when it is reasonable to do so. In other words, clients can be 'dumb', but servers need to be very, very smart—and this is reflected in their code
- *Deadlock* refers to a condition which results when a service cannot proceed because it is waiting for a condition which will never be fulfilled. For example, consider the situation in which a client continues to request information from a server, but never acknowledges a response.



Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 104

8.7 Server Issues – Deadlock and Denial-of-Service

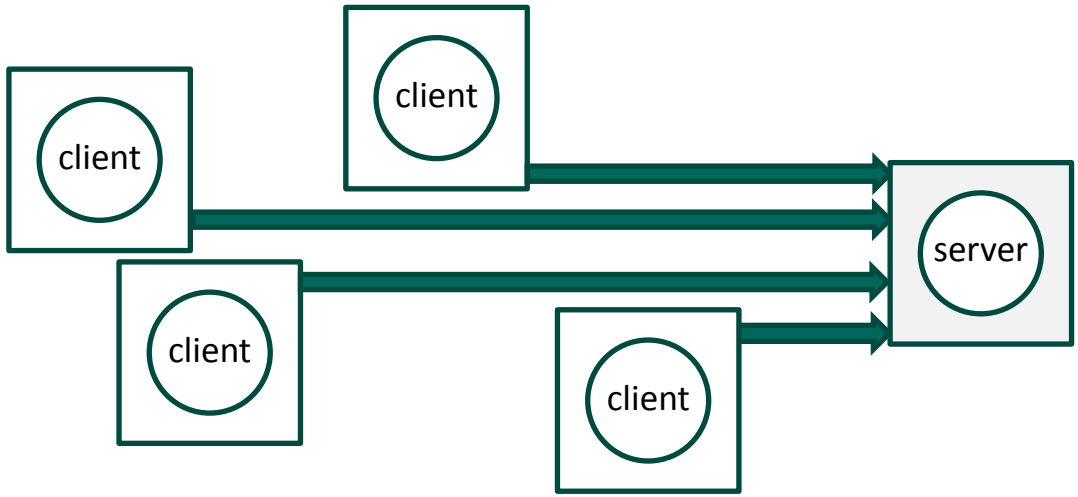
- A related, more modern example, is the denial-of-service attack. Imagine a TCP client which establishes communication with a server and then crashes, or hangs. The server has acknowledged the connection, allocated resources, and then waits for the client to ship information. Now imagine that the same thing happens with 10,000 clients all connecting and hanging at the same time. How does the server 'know' when, and what, resources to free up? This is essentially what happens during a denial-of service attack. The problem is not so much with the number of concurrent requests from clients, as the response of the server to all the requests.

Clients can be dumb; Servers need to be smart

Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 104

8.7 Server Issues – Concurrency

- "*Concurrency refers to real or apparent simultaneous computing*."* Parallel processing involves multiple processors/cores that operate independently, and thus constitutes real concurrency; a multi-tasking OS involves switching between threads, and so gives the impression of concurrency through time-slicing or time-sharing.
- By the very nature of their operation, servers must be written with concurrency in mind.



Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 23

8.7 Server Issues – Concurrency

- A server which is not operating concurrently is called an *iterative server*, i.e. it processes incoming requests sequentially, completing one before handling the next. This is generally very wasteful of time.
- *Concurrent servers* have traditionally not been built around multicore architectures; apparent concurrency arises out of time-sharing a single processor.
- Even if a processor successfully multi-tasks incoming requests, there is one additional problem: most or all of these requests will arrive at the same port for any one server. So the OS software must be able to juggle not just multiple events happening almost-simultaneously, but also deal with multiple requests for the same service.

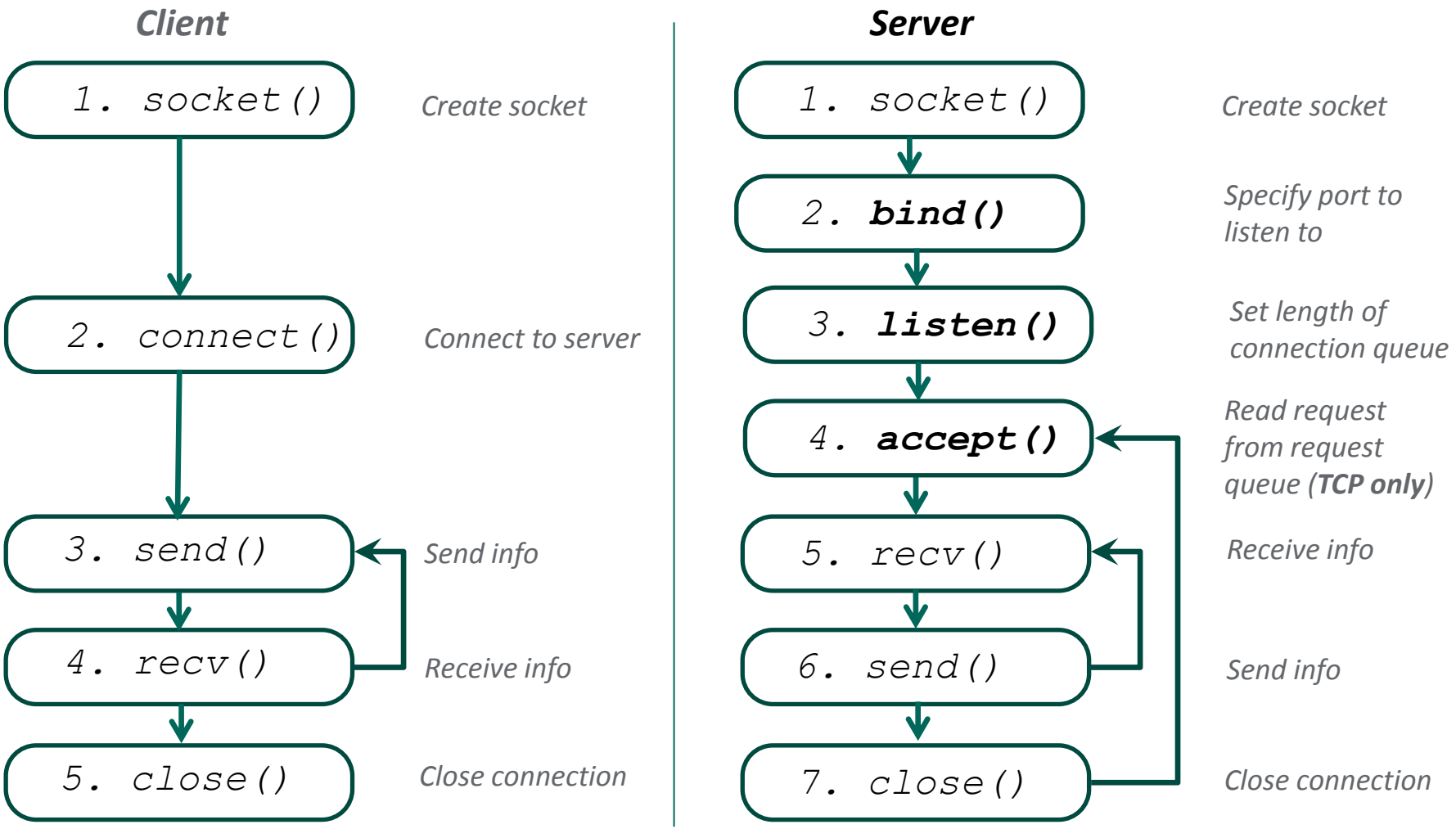
Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 102

8.7 Server Issues – Concurrency

- Clients are focused on performing one task at a time; recall that one of the functions of `connect()` is to ensure that the socket is allocated for a single purpose. With servers, the requirements are exactly the opposite: modern servers must be able to handle multiple incoming request.
- Servers which handle TCP are *connection-oriented*, that is, the server maintains an active connection with the client. Servers which handle UDP are *connectionless*, since UDP is a 'best-effort' service, in which accurate reception of a transmitted message is not guaranteed. Consequently, servers must be written differently depending on the transport protocol used.
- An application protocol designed for one form of transport may perform incorrectly or inconsistently if used over the other form of transport: messages intended for TCP delivery should not be used over UDP

Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 104

8.8 Server Architectures and Algorithms



From Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 59

8.8 Server Architectures and Algorithms

Differences between client and server architecture:

- Client needs communications endpoint to connect with, hence uses *connect()*; server sits passively awaiting call;
- Server needs to *bind* to a particular port so that incoming messages can be dealt with, i.e. make a software connection between activity on a specific port and the software designed to handle it;
- Client is autonomous and can transmit 'whenever it feels like it'. Server must listen for incoming information and can then choose to accept or reject them.
- *recv()* and *send()* calls are reversed between client and server.
- Closing a client means the operation is ended; closing the server means that a particular connection is closed. However, the service itself never truly closes.

From Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 59

8.8 Server Architectures and Algorithms

Servers require the use of three additional functions not found in the client:

bind()

Binds the local endpoint address—the port and IP address—to the socket. For TCP/IP, this means using the **sockaddr_in** structure and socket descriptor. The format of ***bind()*** is:

```
void bind(int sd, struct sockaddr_in* sin, int sz_sin)
```

where:

sd is a socket descriptor, returned from socket()

sin is a pointer to a `sockaddr_in` structure

sz_sin is the size of the `sockaddr_in` structure

Note that servers do not use `connect()`; `bind()` is the server-side version of this function.

8.8 Server Architectures and Algorithms

listen()

A socket may be *active* (if it is a client) or *passive* (if it is a server). A newly created socket is neither. *listen()* places the socket in passive mode, allowing it to accept incoming messages. Like the `recv()` function in a client, servers function as loops that continuously accept incoming information. ***listen()*** tells the operating system to buffer any incoming information until the program has time to read it. The format of ***listen()*** is:

```
void listen(int sd, int qlen)
```

where:

sd is a socket descriptor; and

qlen is the size of the buffer space allocated to the OS

8.8 Server Architectures and Algorithms

`accept()`

Despite the name, this function has nothing to do with deciding whether to accept the incoming message or not. In fact, `accept()` is much more complicated than that. It reads from the incoming request queue and creates a *new* socket for each new connection request. `accept()` uses the original socket as the incoming request 'master', and handles each request with a 'slave' process which requires its own socket. Thus the *same* socket descriptor (sd) always handles the initial request from each *new* client. However, `connect()` creates a *new* socket descriptor to handle *repeated* requests from already-known clients.

Because it is designed to handle connection-oriented requests, `accept()` is not needed for UDP transport protocols. An iterative connectionless server only needs to handle the requests as they come in; it is not required to maintain ongoing communication with multiple clients on an ongoing basis.

8.8 Server Architectures and Algorithms

- The combination of iterative v. concurrent and connectionless v. connection-oriented means that there are four possible models for servers:

Iterative Connectionless	**Concurrent Connectionless**
Iterative Connection-oriented	**Concurrent Connection-oriented**

where: connection-oriented = TCP and connectionless = UDP

Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, Prentice-Hall, 2001, D. E. Comer and D. L. Stevens, pg. 104

8.8 Server Architectures and Algorithms

A simple, iterative*, connection-oriented server*, will need to perform the following steps:

1. Create a socket and bind it to a well-known address for the service being offered
2. Place the socket in passive mode, making it ready for any incoming transmission;
3. Accept the next connection request from the socket, and obtain a new socket for the connection;
4. For each connected socket, repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol; and
5. When finished with a particular client, close the connection and return to step 3 to accept a new connection.

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 111.

8.8 Server Architectures and Algorithms

By contrast, a simple, iterative*, connectionless server* only needs to:

1. Create a socket and bind it to a well-known address for the service being offered
2. Repeatedly read a request from the client, formulate a response, and send a reply back to the client according to the application protocol.

In a limited sense, this server looks much like a client operating in reverse. However, as with the client software, there are a few additional 'gottchas' that need to be dealt with.

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 113.

8.8 Server Architectures and Algorithms

- The functions used for server software are mostly the same as are used for client software. Therefore your code should start with:

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

#include <netdb.h>
#include <netinit/in.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 113.

8.8 Server Architectures and Algorithms

- Like client software, server software needs a communications endpoint. As before, this means initializing **`sockaddr_in`** structure.

```
struct servent *pse;  // ptr to service information entry
struct protoent *ppe; // ptr to protocol information ent

struct sockaddr_in sin;  //internet endpoint

// Note: no struct hostent *phe.  Why?  The server is
// the host; no need to call gethostbyname().  However,
// this does not mean that finding the host's IP
// address is easy
```

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 128.

8.8 Server Architectures and Algorithms

- In this example, our server will act as a simple time server, returning the time at the server to any client that requests it.

```
int sd;
char *service = "time";     // use time service in this ex
char *transport = "UDP"     // i.e. connectionless
```

Taken, with modifications, from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 128.

8.8 Server Architectures and Algorithms

- In this case, the first members of `sockaddr_in` are set the same as before:

```
sin.sin_length = sizeof(sin);
sin.sin_family = AF_INET;
```

At this point in the client software, we called **`gethostbyname()`** to get the IP of the server the client connects to. This function returned the **`hostent`** structure, and from this we assigned a value to **`sin.sin_addr`** like this:

```
// used for client, but not for server
memcpy(&sin.sin_addr, phe->h_addr, phe-> h_length);
```

8.8 Server Architectures and Algorithms

- In principle, it should be easy for the server to 'know' its own address. Because of the architecture of the internet, this turns out not to be the case.
- Because many servers are connected to the internet via routers and other devices, they do not have one specific IP address, but several. If the server specifies its address as being at one particular IP, it will not be able to accept communications from clients that use its other IPs.
- The solution is to set the host address to **`INADDR_ANY`**. This is a 'generic' IP address that allows the server to accept everything comes its way; the routers do the job of ensuring that the IPs correspond to the actual server being used.
- The definition of **`INADDRY_ANY`** is:

```
#define INADDR_ANY ((unsigned long int) 0x00000000)
```

i.e. this is the 'NULL pointer' of IP addresses.

8.8 Server Architectures and Algorithms

- Setting the host address IP is therefore much simpler for the server than the client:

```
sin.sin_addr.s_addr = INADDR_ANY;
```

- Finally, we can set the port address as before using:

```
if (pse = getservbyname(service, transport)
   sin.sin_port = pse->s-port;
```

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 128.

Note: Network Byte Order

- One issue not dealt with earlier is that of *network byte order*. This means that integer values must be represented with the most significant byte first—i.e. big endian, regardless of the platform on which the information is stored.
- Socket routines include several functions designed to insure network byte order. These should be used to insure portability between different architectures.
- These functions are divided into two sets depending on the size of integer they operate on: short (16-bit integral values) and long (32-bit integral values). For example:

```
htons() // convert int from host to network short
htonl() // convert int from host to network long
ntohs() // convert int from network to host short
ntohl() // convert int from network to host long
```

Note: Network Byte Order

- Software often hides this information from the programmer. However some socket routines require that network byte order must be used. Such a conversion is required before the server's port address can be assigned. To ensure that the value is properly assigned, it is not unusual to see the port value coerced, somewhat awkwardly, like this

```
if (pse = getservbyname(service, transport)
    sin.sin_port =
        htons(ntohs((unsigned short)pse->s-port));
```

8.8 Server Architectures and Algorithms

- Finally, we can use the sockaddr_in structure to obtain a socket descriptor

```
sd = socket(PF_INET, SOCK_DGRAM, 0);
```

(In other words, nothing really new up to this point, except for `INADDR_ANY`)

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 128.

8.8 Server Architectures and Algorithms

- The next step is **`bind()`**, which connects the socket to the host endpoint

```
if (bind(sd, (struct sockaddr_in *)&sin, sizeof(sin)) <0)
        // signal error and exit
else
        // socket is bound to port
        // okay to continue with connection
```

Taken, with modifications, from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 129.

8.8 Server Architectures and Algorithms

- If this were a TCP (i.e. a connection-oriented) server, we would at this point use **`listen()`** to place the socket into passive mode. Instead, we'll loop through infinitely, checking the message queue for incoming data in much the same way as we did with the client. For this, we require the services of two special functions, the server-side UDP analogs of **`recv()`** and **`send()`**.

Taken, with modifications, from*: Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 129.

8.8 Server Architectures and Algorithms

- These new functions differ in the following way: they store the address of the incoming transmission—something the client doesn't need to do.

```
recvfrom(sd, messbuf, mlen, flags, from_addr, flen);
sendto(sd, messbuf, mlen, flags, to_addr, tlen);
```

where:

sd is the socket descriptor;
messbuf is a pointer to a buffer to the next datagram;
mlen is the length of the message;
flags controls special cases, and is not of interest to us;
from_addr and **to_addr** point to buffers containing the addresses used for receiving from and sending to;
flen and **tlen** are pointers to the lengths allocated to these buffers.

Both functions return -1 if upon failure.

From: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 113-114.

8.8 Server Architectures and Algorithms

- So the final piece of the program is a loop that calls these two functions (forever) and returns the time at the server. We'll assume that the following have already been defined/declared:

```
#include  <time.h>

#define UNIXEPOCH 2208988800UL
//number of seconds from 1/1/1900 to 1/1/1970

char buf[1];      // input buffer
time_t now;       // current time, sent as string
unsigned int alen; // from address length

struct sockaddr_in fsin; // client's address
```

Taken, with modifications, from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 129.

8.8 Server Architectures and Algorithms

- And here's the code:

```
while(1){
  if (recvfrom(sock, buf, sizeof(buf), 0,
     (struct sockaddr_in *)&fsin, sizeof(fsin))<0)
      // -1 returned; print error message and exit

  // get the time and set to network byte order
  time(&now);   // get current time
  now = htonl((unsigned long)(now + UNIXEPOCH));

  //send the time to the client
  sendto(sd, (char*)&now, sizeof(now), 0,
      (struct sockaddr_in *)&fsin, sizeof(fsin));
}
```

8.8 Server Architectures and Algorithms

- The above example shows how we would deal with a connectionless server. For a connection-oriented server, the code would differ in the following ways.
- First, we need to have our server listen to the port by putting it into passive mode:

```
if (listen(msock, qlen) < 0)
    // signal error and exit
```

where **`qlen`** is the size of the buffer space allocated to the OS, as mentioned earlier, and **`msock`** is the socket descriptor returned from the call to **`socket()`**. Its been given a new name here, because we will be using the initial socket descriptor as the master socket, to distinguish it from the slave sockets that will actually respond to repeated requests from the client.

Taken, with modifications, from: *Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version*, D. L. Comer and D.L. Stevens, Prentice-Hall, 2001, pg. 129.

8.8 Server Architectures and Algorithms

- Now our code can use **`accept()`** to read the information at the incoming socket and pass that information to a new socket.

```
while(1){
    ssock = accept(msock,
       (struct sockaddr_in *)&fsin, sizeof(fsin));
    if (ssock < 0)
      // print error and exit

  // get the time and set to network byte order
  char *pts = time(&now);  // get current time
  send(ssock, pts, strlen(pts)); //send to client

  close(ssock);
}
```

8.8 Server Architectures and Algorithms

- We've assumed the declaration of two integer values:

```
int msock, ssock;     //master and slave sockets
```

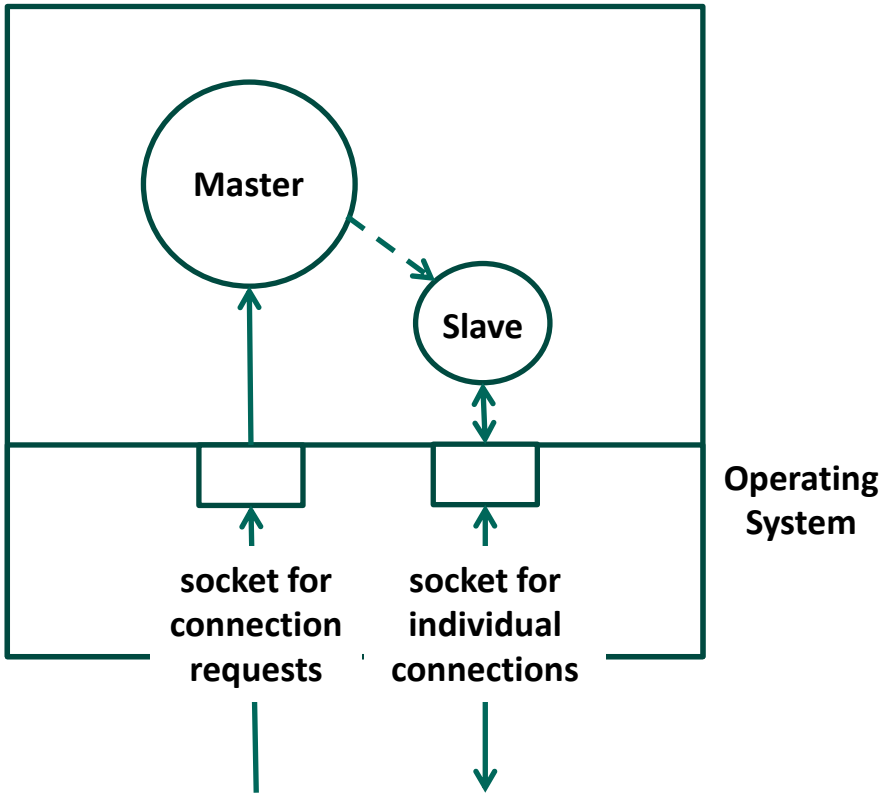
Notice what happens inside **accept()**:

```
ssock = accept(msock,
    (struct sockaddr_in *)&fsin, sizeof(fsin));
```

accept() takes the master socket and returns a slave, which is then used by **send()** to transmit the requested information to the client. **accept()** performs a three-way handshake with the client, blocking any other requests in the process. When the handshake is completed, the function allocates a new socket for the incoming connection, returning this value to the calling program. If the connection is not completed, **accept()** waits forever and the server becomes deadlocked.

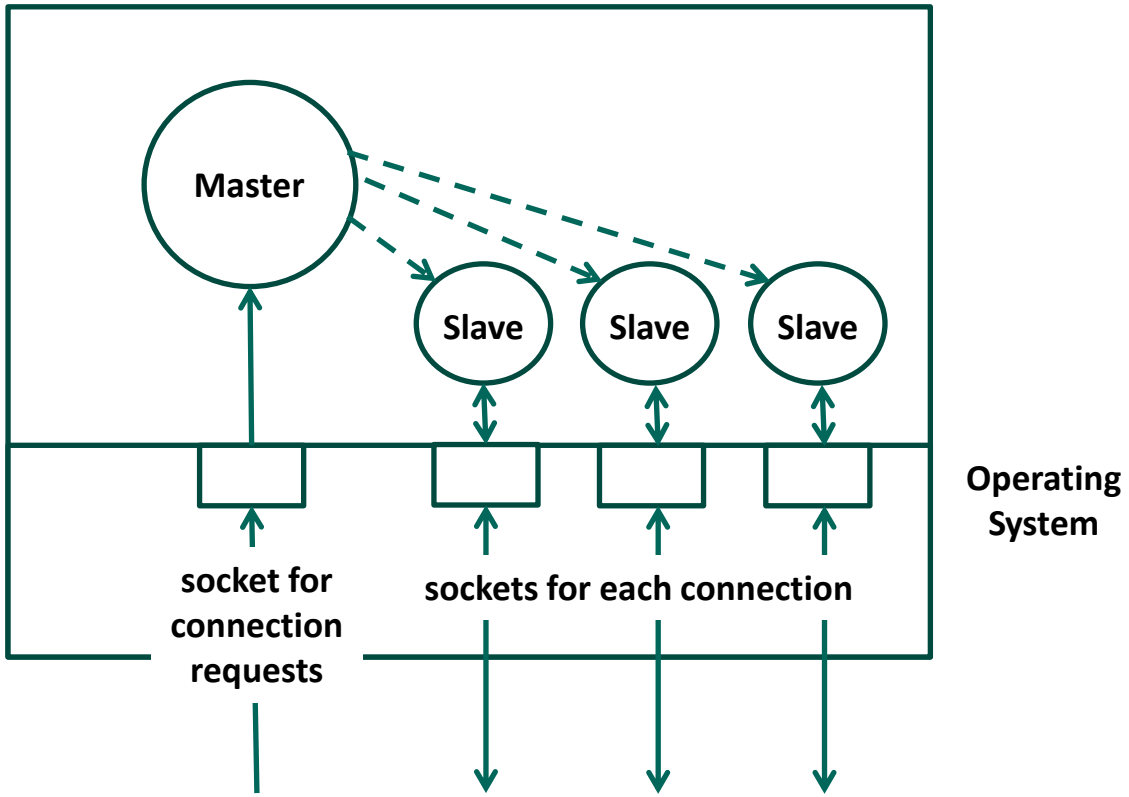
8.8 Server Architectures and Algorithms

- All that's really happened here is that we've accepted a request on one socket and responded to it on another. This corresponds to:



8.8 Server Architectures and Algorithms

- In a concurrent, connection-oriented server, we can create multiple slaves to deal with each incoming request. The master socket handles the incoming requests only, and passes each request to a slave socket.



8.8 Server Architectures and Algorithms

- In the connection-oriented server, **`fork()`** is used to create a separate thread for each slave socket.

```
while(1){
    ssock = accept(msock,
       (struct sockaddr_in *)&fsin, sizeof(fsin));
    if (ssock < 0)
          if (errno = EINTR) continue;
          // else print error and exit
          ...
```

Where **`EINTR`** is an error generated whenever a request is received while **`accept()`** is blocking inputs. In other words, this is the one type of error that should not cause our server to halt.

8.8 Server Architectures and Algorithms

- In the connection-oriented server, **`fork()`** is used to create a separate thread for each slave socket.

```
...
switch(fork()){
   case 0:  //PID = 0 for child process
      close(msock);
      char *pts = time(&now);
      send(ssock, pts, strlen(pts));
      exit(0);  // exit this thread
   default:
      close(ssock);
      break;
   case -1:
      // fork error; exit
}
}
```

Note: Resource Material

- For anyone doing detailed work on network programming in C, the following two books are considered 'classics'

  Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications Linux/POSIX Sockets Version, D. E. Comer and D.L. Stevens, Prentice-Hall, 2001.

  TCP/IP Illustrated, Volume 2: The Implementation, G. R. Wright and W. R. Stevens, Addison-Wesley, 1995.