LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 634: Networks Programming

Lecture 05: BSD Sockets Programming

Instructor: Haidar M. Harmanani

Network Programming

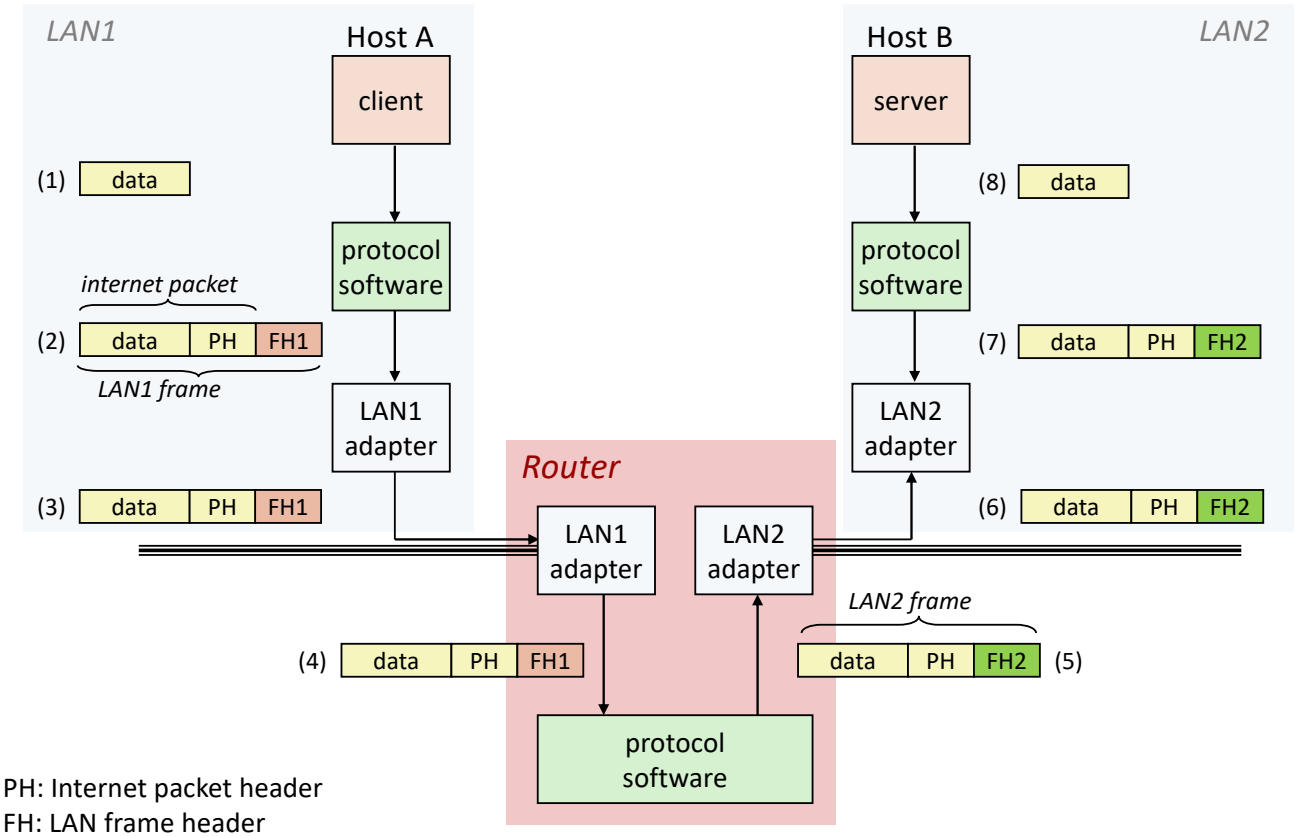
- Network allows arbitrary applications to communicate
- Programmer does not need to understand network technologies
- Network facilities are accessed through an Application Program Interface
- Demonstrate that a programmer can create Internet application software without understanding the underlying network technology
  - Introduce a set of API's
  - Discuss a couple of examples
- We will introduce the concept of client server computing as well as network communication using sockets

Basic Terms

- Host
  - Single computer, end-system. Could be personal computer, dedicated system (print or file server) or time-sharing system.
- Process
  - Any program which is executed by computer's operation system.
- Thread
  - Separate part of process, providing it's specific working flow, and sharing the process data and resources with other threads.
- Inter-Process Communication
  - Sharing of information and resources by two or more different processes.
- Inter-Host Communication
  - Communication between two or more processes, running on different hosts in the network.

Client-Server Model

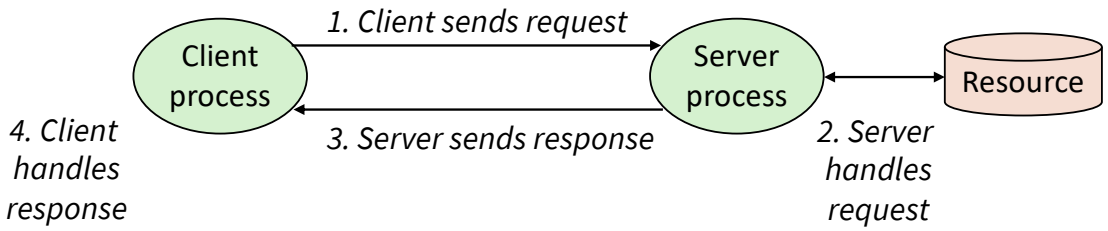
Recall: Transferring internet Data Via Encapsulation



PH: Internet packet header
FH: LAN frame header

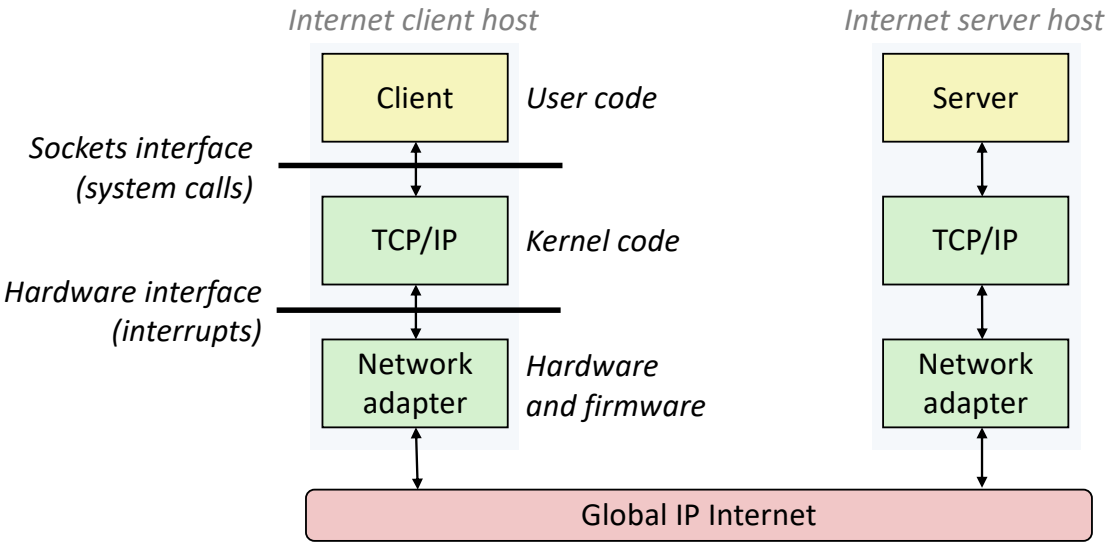
A Client-Server Transaction

- *Client Server* Model
  - A pair of communicating applications, one side starts execution and waits indefinitely for the other to contact it
  - Need to address this issue since TCP/IP does not respond to incoming communication requests on its own
- Most network applications are based on the client-server model:
  - A ***server*** process and one or more ***client*** processes
  - Server manages some ***resource***
  - Server provides ***service*** by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)



Note: clients and servers are processes running on hosts (can be the same or different hosts)

Hardware and Software Organization of an Internet Application



A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*
   – 151.101.1.67

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*
   – 151.101.1.67 is mapped to www.cnn.com

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

IP Addresses

- 32-bit IP addresses are stored in an *IP address struct*
  - IP addresses are always stored in memory in *network byte order* (big-endian byte order)
  - True in general for any integer transferred in a packet header from one machine to another.
    - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t  s_addr; /* network byte order (big-endian) */
};
```


Internet Connections

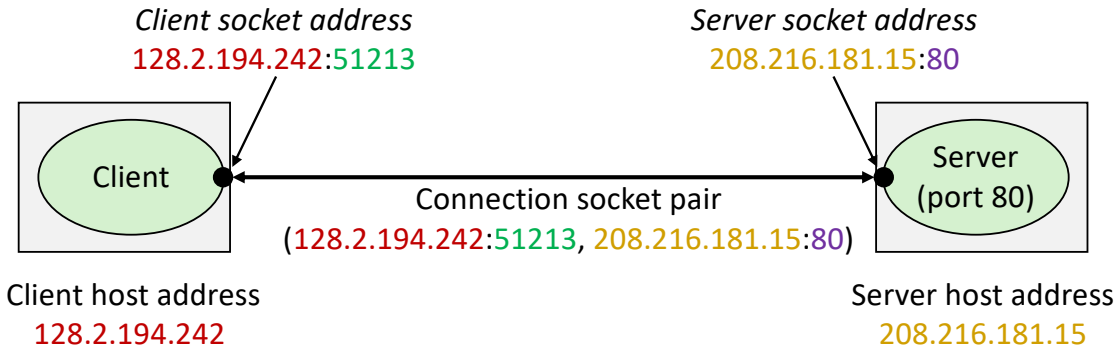
- Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:
  - *Point-to-point*: connects a pair of processes.
  - *Full-duplex*: data can flow in both directions at the same time,
  - *Reliable*: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

- *A socket* is an endpoint of a connection
  - *Socket address* is an `IPaddress:port` pair

- A *port* is a 16-bit integer that identifies a process:
  - ***Ephemeral port*:** Assigned automatically by client kernel when client makes a connection request.
  - ***Well-known port:*** Associated with some *service* provided by a server (e.g., port 80 is associated with Web servers)

Anatomy of a Connection

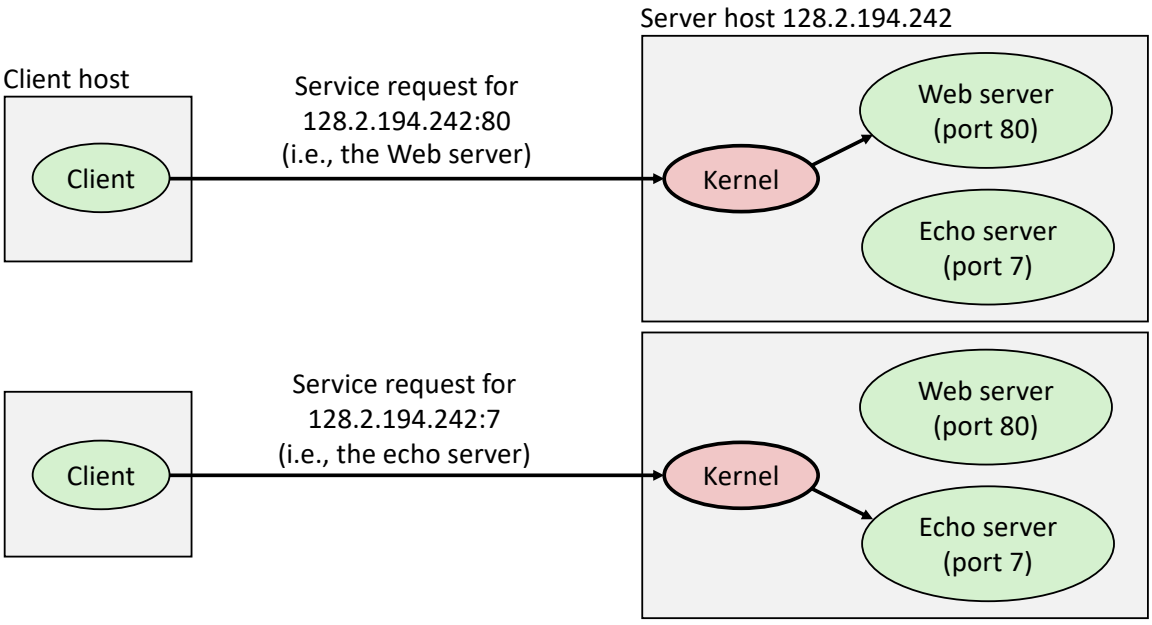
- A connection is uniquely identified by the socket addresses of its endpoints (socket pair)
  - (cliaddr:cliport, servaddr:servport)



51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers

Using Ports to Identify Services



Client-Server Model

- Client
  - An application that initiates a peer-to-peer communication and invoked by end users
    - Contacts a server
    - Sends a request
    - Awaits a response
- Server
  - A program that awaits for incoming communication requests from clients
    - Receives the client's request
    - Performs necessary computation
    - Returns result to the client
- Why use servers?
  - Perform computations which are too complex for client
  - Centralize data (ease of updating)
  - Security

Client-Server Model

- Not necessarily over network
- Single machine uses interprocess communication
- Single computer may run multiple servers
- Dedicated computer is often referred to as a server

Client-Server Model

- Servers can be simple
  - Time-of-day server
- Servers can be complex
  - Web server
  - Database server

Client-Server Model

- Servers usually implemented as applications
- Run on any platform which supports TCP/IP
- Multiple servers can offer same service
  - Improve reliability and performance

Standard Vs. Non-Standard

- Standard application services
  - Defined by TCP/IP and assigned universally recognized protocol port identifiers
- Non-Standard application services
  - Application services which are locally defined
- Difference is important when communicating outside the local environment
- TCP/IP standard application protocols
  - remote terminal client (Telnet)
  - electronic mail client that uses the standard SMTP protocol
  - file transfer client that uses the standard ftp protocol

Well-known Ports and Service Names

- Popular services have permanently assigned *well-known ports* *and* corresponding *well-known service names*:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - Web servers: 80/http
- Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

Parameterization

- Clients contact servers
- It is nice that the client can
  - Specify the remote machine when servers operate
  - Protocol port number at which the server is listening
- A software that allows a user to specify protocol port number is called fully parameterized client

Connection Vs. Connectionless

- Recall that there are two styles of interactions
  - A connection oriented style that corresponds to TCP transport protocol
  - A connectionless style that corresponds to UDP transport protocol
- Critical to differentiate among both styles

TCP

- Provides needed reliability to communicate across the Internet
  - Verifies date of arrival
  - Retransmits segments that do not arrive
  - Compute checksum over the data to verify no corruption occurred during transmission
  - Eliminates duplicate packets
  - Provides flow control
    - Sender does not transmit data faster than the receiver can consume it
  - Informs client & server if underlying network is inoperable

UDP

- Provides no guarantees for reliability delivery
  - Provides a best effort delivery
    - Relies on the underlying IP Internet to deliver packets
    - IP depends on underlying hardware and intermediate routers
  - Clients requests maybe lost, duplicated, delayed or delivered out of order
  - Server responses may be lost
  - Its is the client/server application program responsibility to detect and correct such errors
- Works well
  - If underlying Internet Works well!
  - In a local environment

Common Mistakes

- Choose a connectionless transport (UDP) to implement the software
  - Test the software on a LAN
  - Software appears to work well since LAN's seldom delay, drop, or deliver packets out of order
- Now, use software over the Internet
  - May fail and produce incorrect results

When to Use What?

- Use TCP because it is safer and more reliable
- Use UDP if
  - Application protocol handles reliability
  - Application requires hardware broadcast or multicast
  - Application Cannot tolerate virtual circuit overhead

Stateless Vs. Stateful Servers

- State Info.
  - Info. that servers maintain about the status of ongoing interactions with clients
    - Servers that do not keep state info are called stateless
    - Those that keep state info are called stateful servers
- Example
  - A server that allows clients to remotely access info kept in files on a local disk
  - Server awaits for a client contact over the Net
    - Receives one or two request types
      - Extract data from a file or store in a file
    - Performs specified requested operation and replies to the client

Stateless Vs. Stateful

- Example (Cont.)
  - A stateless server maintains no transactions info
    - A message from a client must specify file name, position from which data be extracted, and number of bytes to extract

Handle	File Name	Current Posn.
1	test.c	0
2	tcp.doc	456
3	dept.text	38
4	tetris.exe	128

  - A stateful server maintains state table info for clients
  - Stateful are good
    - In an ideal world where there is reliable delivery and no crash
    - Make messages smaller & Processing simpler
    - Lead to a complex applications protocols

Servers as Clients

- A server program may need to access network services that require it to act as a client
- Must be careful to avoid circular dependencies among servers

Concurrent Processing

- Concurrency
  - Real or apparent simultaneous computing
- Two approaches
  - Time sharing
    - switch a single processor among multiple computations quickly enough to give the appearance of simultaneous access
  - Multiprocessing
    - Multiple processors perform computations simultaneously

Concurrency in Networks

- Machines on a single network
  - Pairs of application programs that communicate concurrently
- Within a computer system
  - Multiple users on a time sharing system invoke a client application that communicates with an application on another machine
  - Concurrency among multiple client programs automatic
    - The OS allows multiple users to each invoke a client concurrently
  - Unlike concurrency in clients, it requires considerable effort in servers

Concurrent Processing

- On a uniprocessor architecture, CPU can execute one process at any instant in time
  - OS switches the CPU among all executing processes
- Concurrent Execution
  - Apparently simultaneously execution
  - When Multiple processes execute a piece of code concurrently, each process has its own independent copy of the variables associated with the code as well as its activation records.

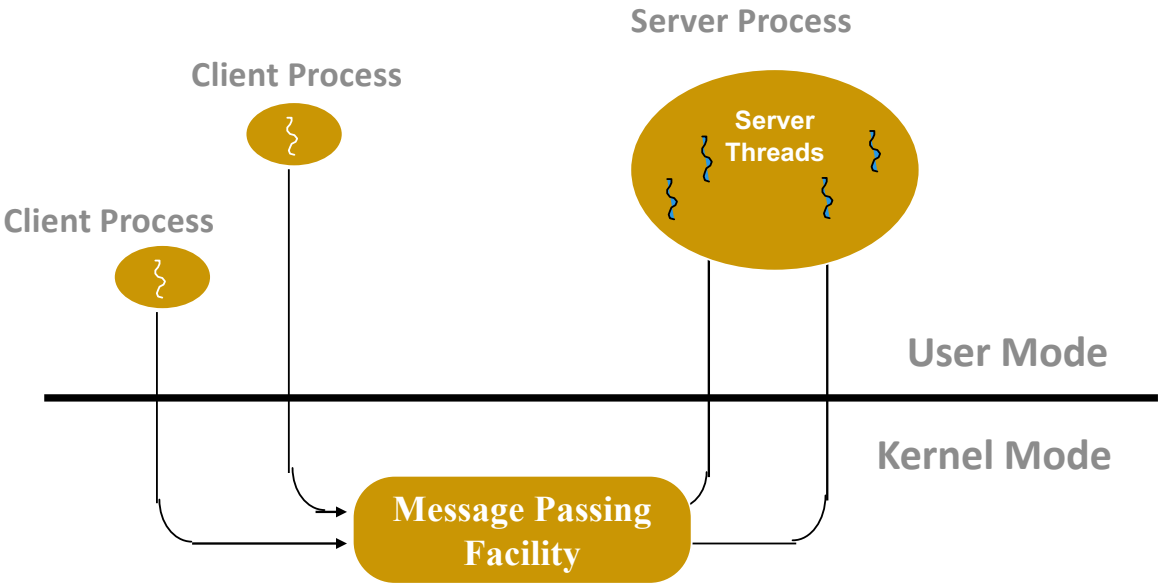
Recall from Last Week …

```
int sum;
main()
{
  int i, pid;
  sum = 0;

  pid = fork();
  for (i = 0; i < 6; i++) {
     printf("i is %d\n"; i);
     fflush(stdout);
     sum += i;
  }
  printf("sum is %d\n", sum);
  exit(0);
}
```


Timeslicing

- In a concurrent processing, the OS allocates available power to each one for a short time before moving on to the next
- Timeslicing
  - A system that shares available CPU among several processes concurrently
- How? On what rate?
  - Approximately 50% of CPU
  - For n processes, each receive 1/n

Threads in Action: Multithreaded Server



Diverge Processes

- fork returns an integer process identifier (pid)
  - Process Id differs in the original and newly created processes
  - Concurrent programs use the value returned by fork to decide how to proceed

```
main()
{
  int pid;

  pid = fork();
  if (pid != 0)
     printf("Original process prints this\n");
  else
     printf("New process prints this\n");
  exit(0);
}
```

Context Switching & Protocol Software

- Context Switching
  - When an OS stops temporarily executing a process and switch to another one
  - Switching process context requires the use of the CPU
  - While CPU is busy switching, none of the application processes receive any services
    - Overhead needed to support concurrent processing
- Protocol Software minimize context switching

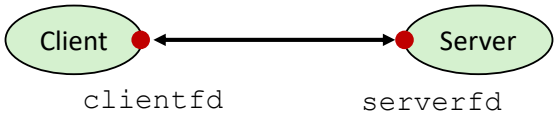
Sockets Programming

Sockets Interface

- Set of system-level functions used in conjunction with Unix I/O to build network applications.
- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.
- Available on all modern systems
  - Unix variants, Windows, OS X, IOS, Android, ARM

Sockets

- What is a socket?
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - ***Remember:*** All Unix I/O devices, including networks, are modeled as files
- Clients and servers communicate with each other by reading from and writing to socket descriptors



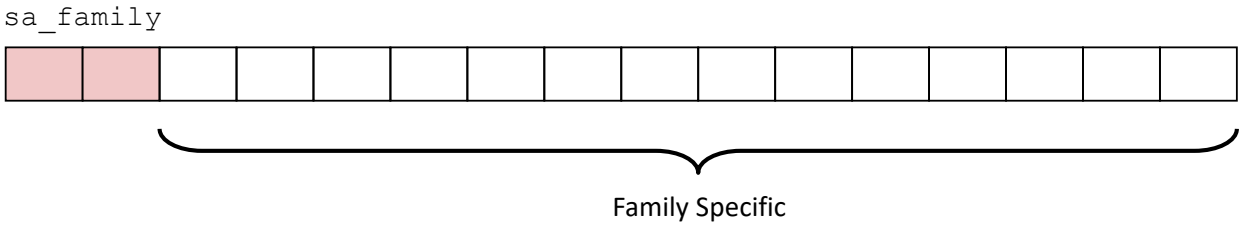
- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors

Socket Address Structures

- Generic socket address:
  - For address arguments to **`connect`**, **`bind`**, and **`accept`**
  - Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed
  - For casting convenience, we adopt the following convention:

```
typedef struct sockaddr SA;
```

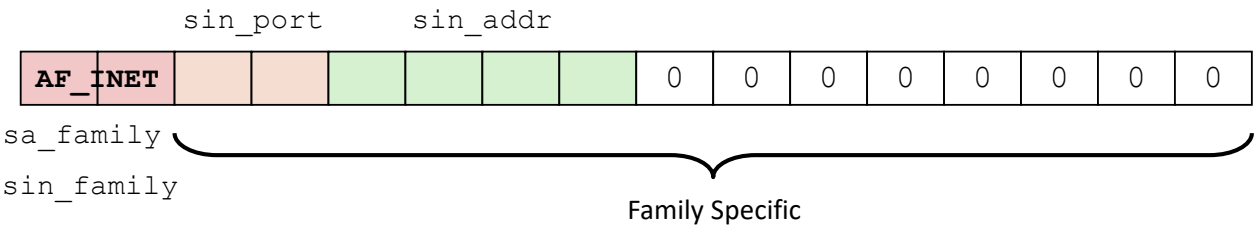
```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```



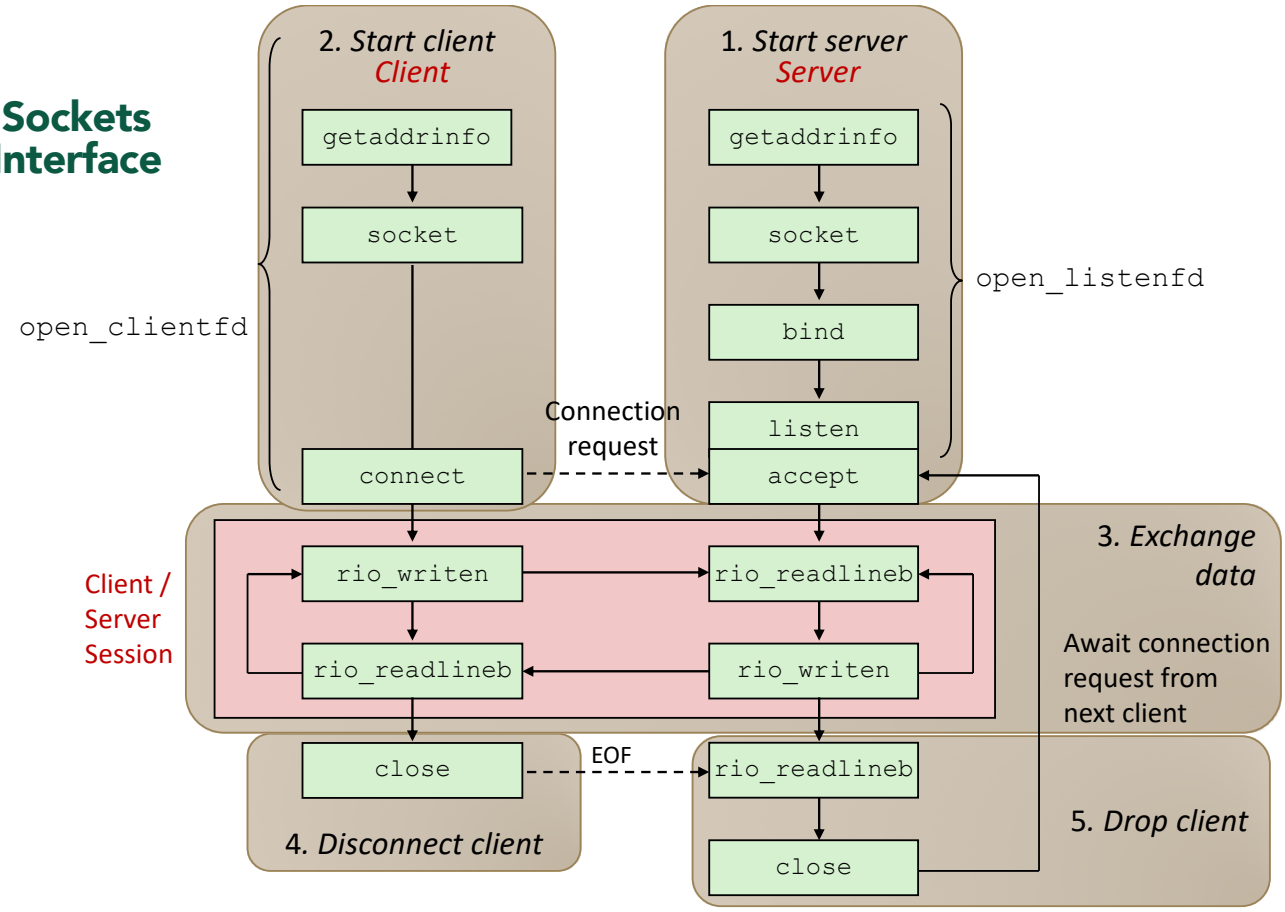
Socket Address Structures

- Internet-specific socket address:
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

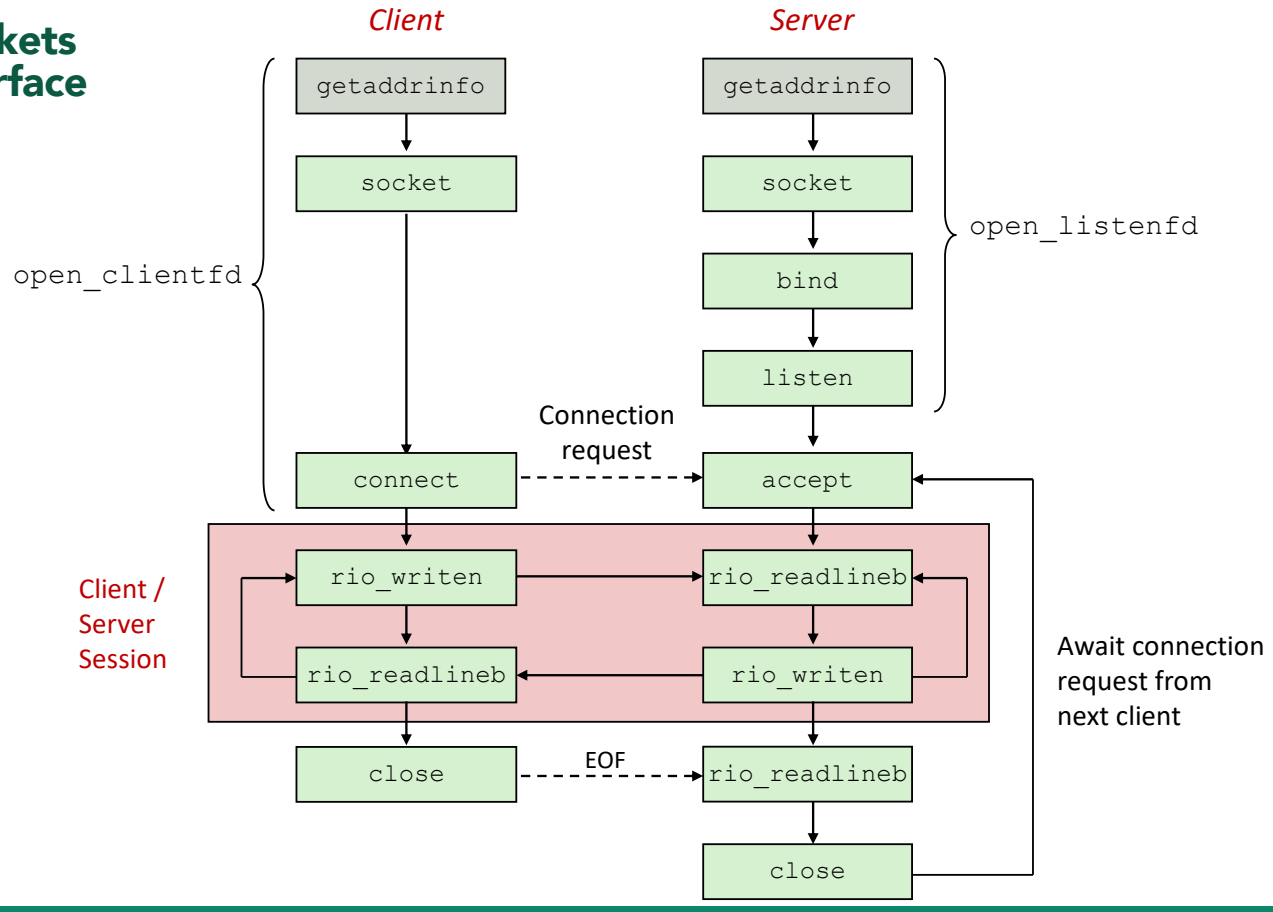
```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



Sockets Interface



Sockets Interface



Sockets Programming Using C

Host and Service Conversion: getaddrinfo

- `getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
  - Replaces obsolete `gethostbyname` and `getservbyname` functions

- Advantages:
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

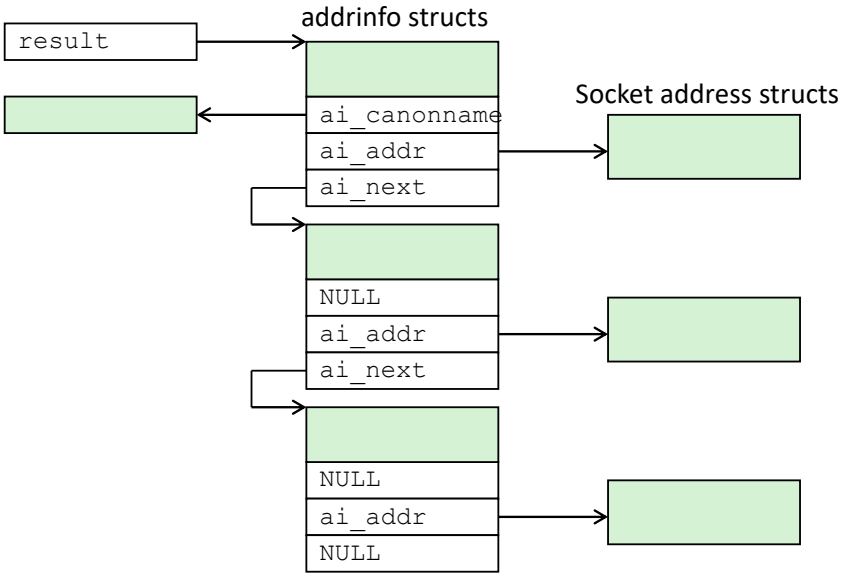
- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

Host and Service Conversion: getaddrinfo

```
int getaddrinfo(const char *host,              /* Hostname or address */
                const char *service,           /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);    /* Output linked list */

void freeaddrinfo(struct addrinfo *result);   /* Free linked list */

const char *gai_strerror(int errcode);        /* Return error msg */
```

- Given `host` and `service`, `getaddrinfo` returns `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.
- Helper functions:
  - `freeadderinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.

Linked List Returned by `getaddrinfo`



- Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- Servers: walk the list until calls to `socket` and `bind` succeed.

```
struct addrinfo {
    int              ai_flags;     /* Hints argument flags */
    int              ai_family;    /* First arg to socket function */
    int              ai_socktype;  /* Second arg to socket function */
    int              ai_protocol;  /* Third arg to socket function  */
    char            *ai_canonname; /* Canonical host name */
    size_t           ai_addrlen;   /* Size of ai_addr struct */
    struct sockaddr *ai_addr;      /* Ptr to socket address structure */
    struct addrinfo *ai_next;      /* Ptr to next item in linked list */
};
```

- Each addrinfo struct returned by `getaddrinfo` contains arguments that can be passed directly to `socket` function.
- Also points to a socket address struct that can be passed directly to `connect` and `bind` functions.

Host and Service Conversion: getnameinfo

- `getnameinfo` is the inverse of `getaddrinfo`, converting a socket address to the corresponding host and service.
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
                char *host, size_t hostlen,    /* Out: host */
                char *serv, size_t servlen,    /* Out: service */
                int flags);                    /* optional flags */
```

Conversion Example

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
```

hostinfo.c

Conversion Example (cont)

```c
    /* Walk the list and display each IP address */
    flags = NI_NUMERICHOST; /* Display address instead of name */
    for (p = listp; p; p = p->ai_next) {
        Getnameinfo(p->ai_addr, p->ai_addrlen,
                    buf, MAXLINE, NULL, 0, flags);
        printf("%s\n", buf);
    }

    /* Clean up */
    Freeaddrinfo(listp);

    exit(0);
}
```

hostinfo.c

Running hostinfo

```
> ./hostinfo localhost
127.0.0.1

> ./hostinfo twitter.com
199.16.156.230
199.16.156.38
199.16.156.102
199.16.156.198
```


Sockets Interface: socket

- Clients and servers use the `socket` function to create a *socket descriptor*:

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

- Example:

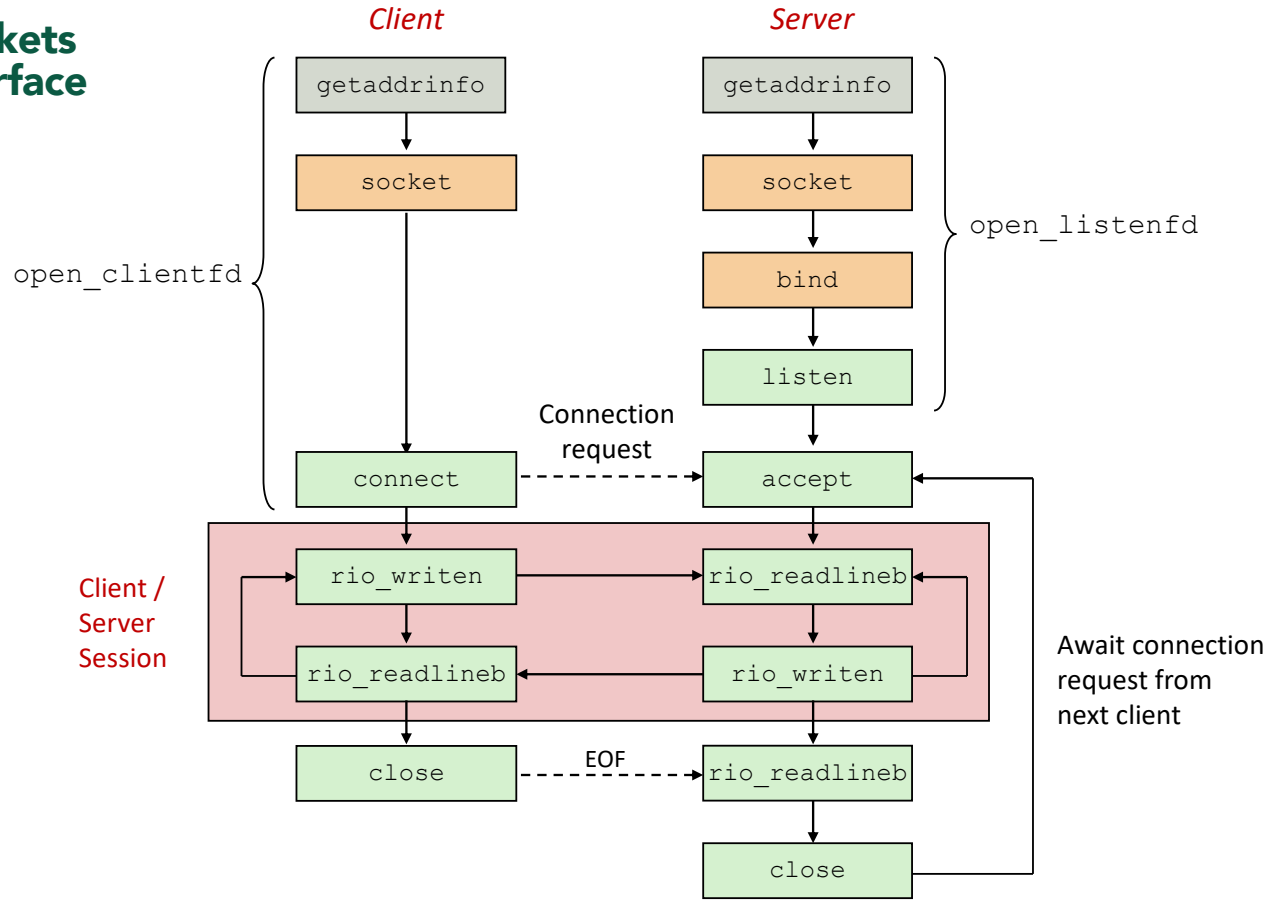
```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

Indicates that the socket will be the end point of a connection

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface



Sockets Interface: bind

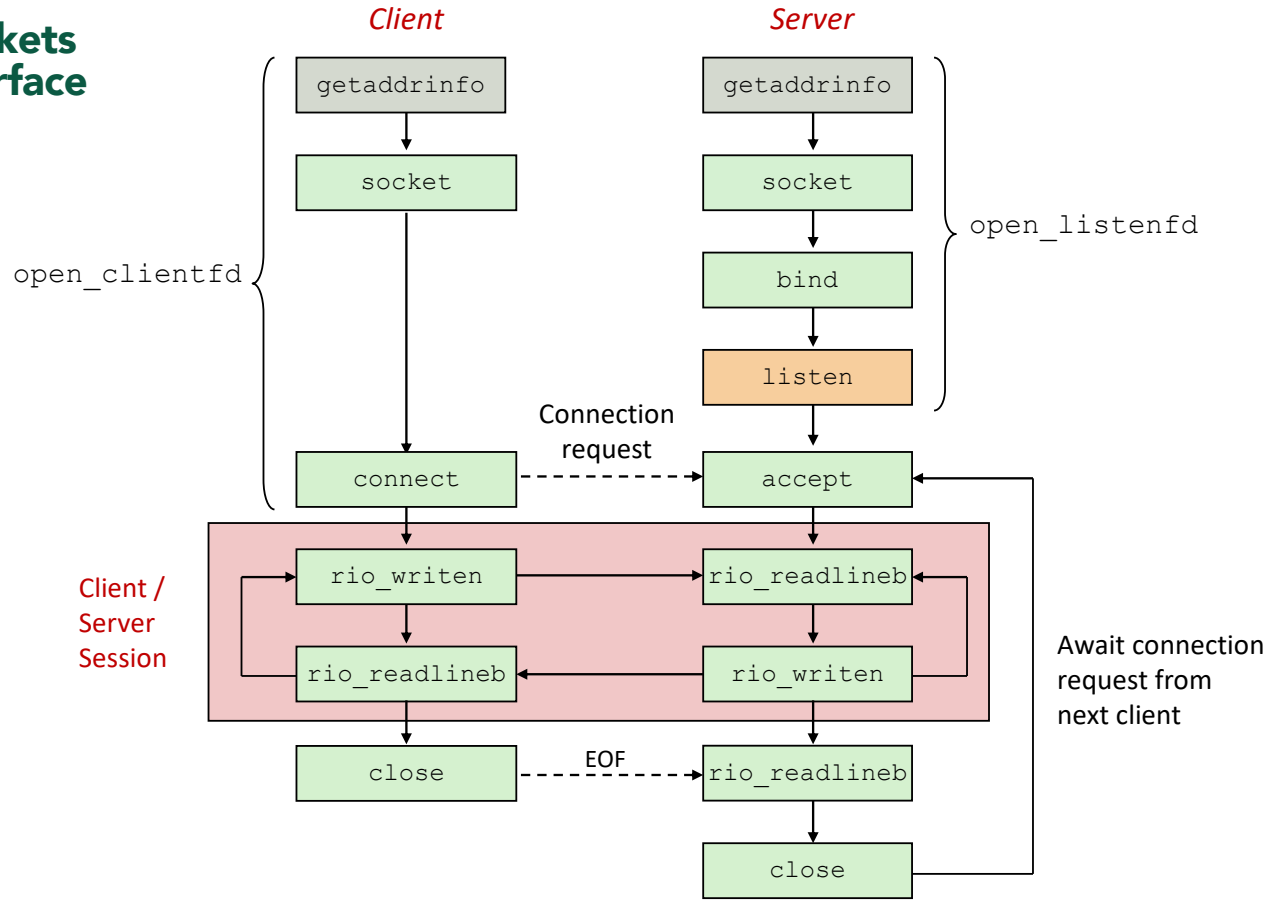
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

Sockets Interface



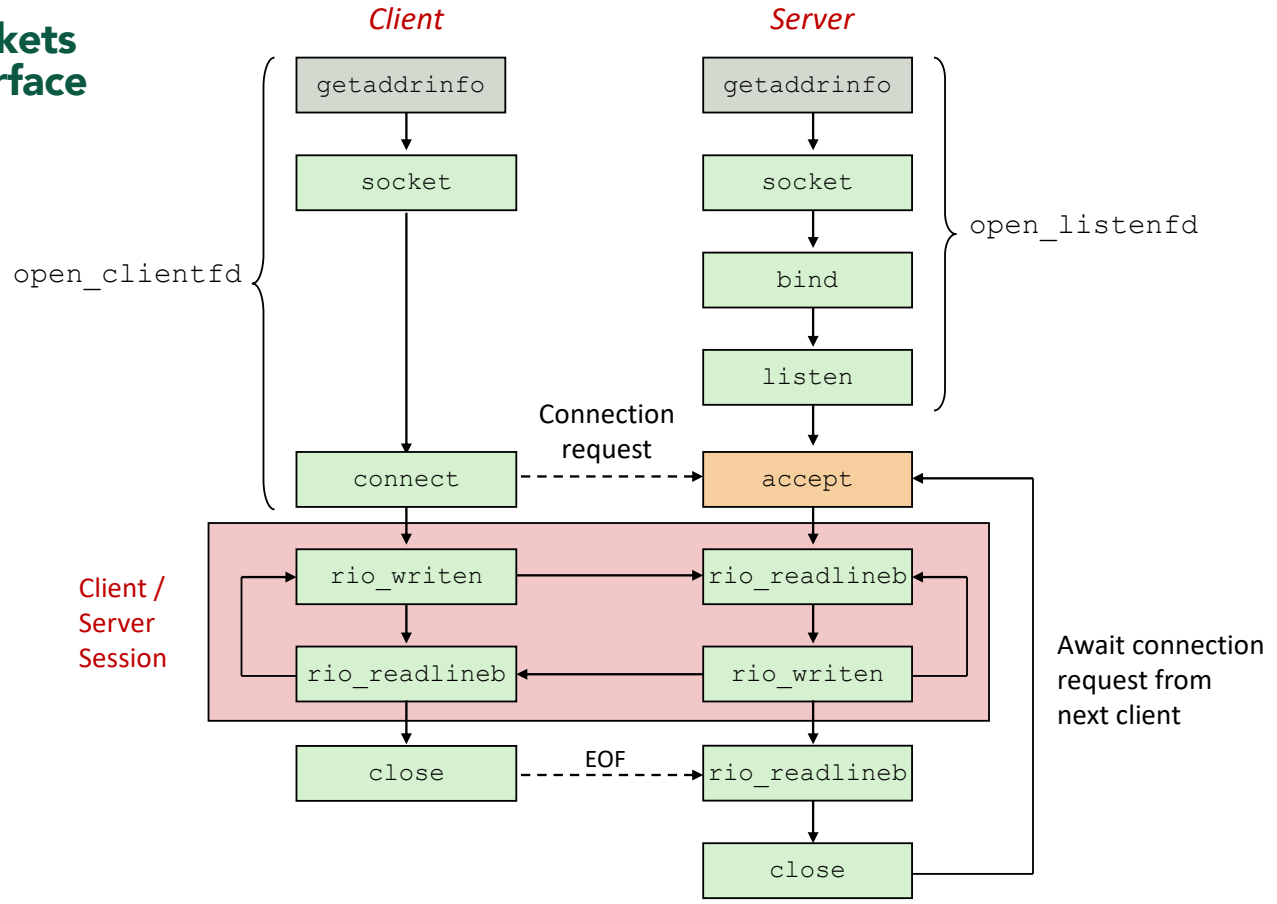
Sockets Interface: listen

- By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

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

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.
- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

Sockets Interface



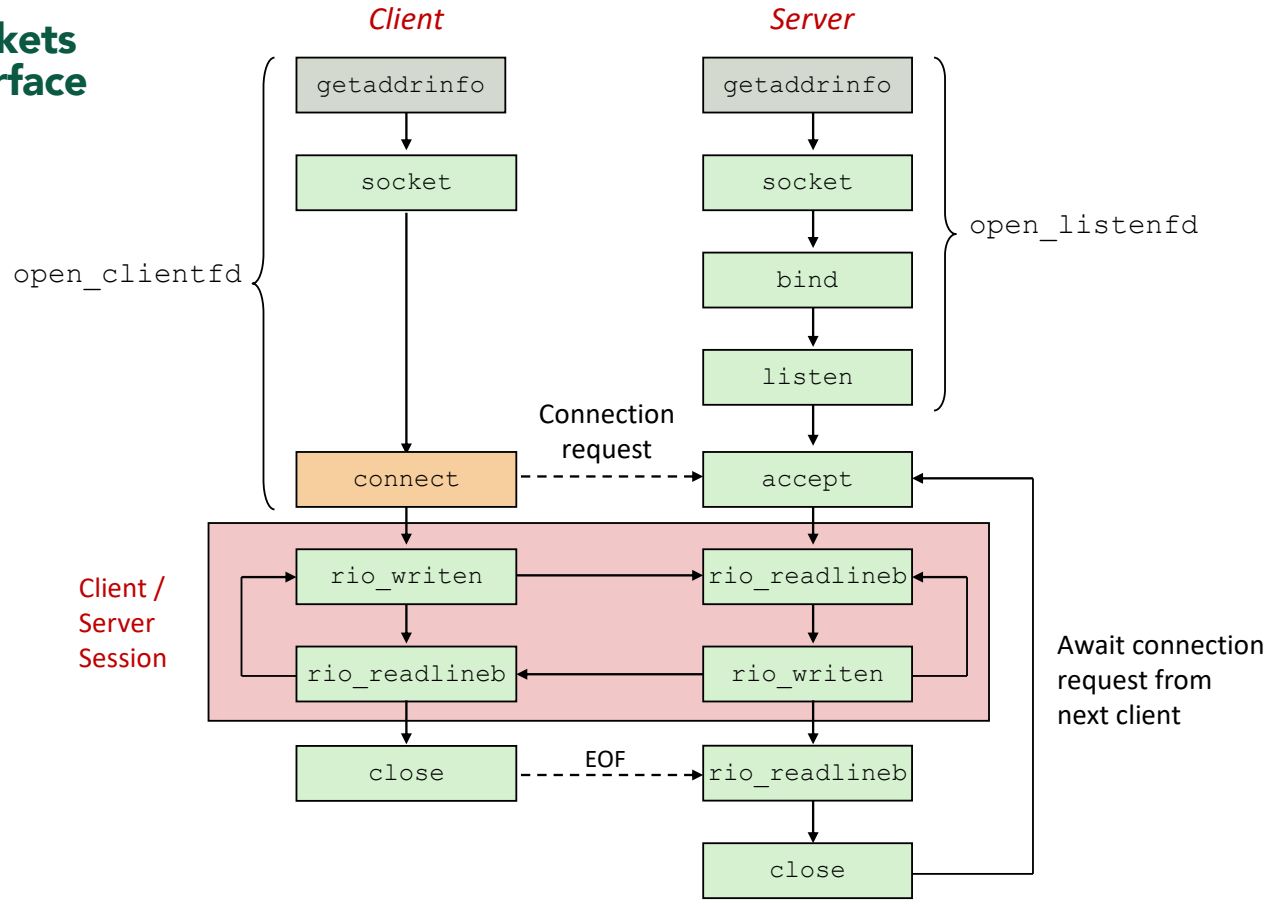
Sockets Interface: accept

- Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.
- Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.

Sockets Interface



Sockets Interface: connect

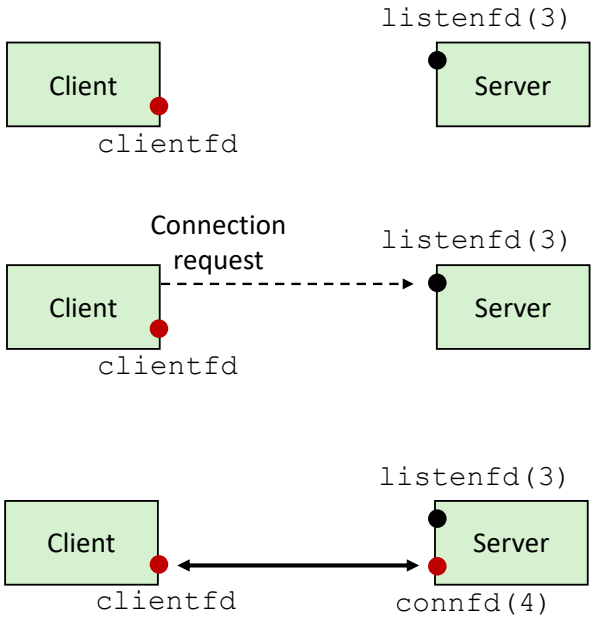
- A client establishes a connection with a server by calling connect:

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair
    `(x:y, addr.sin_addr:addr.sin_port)`
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

accept Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

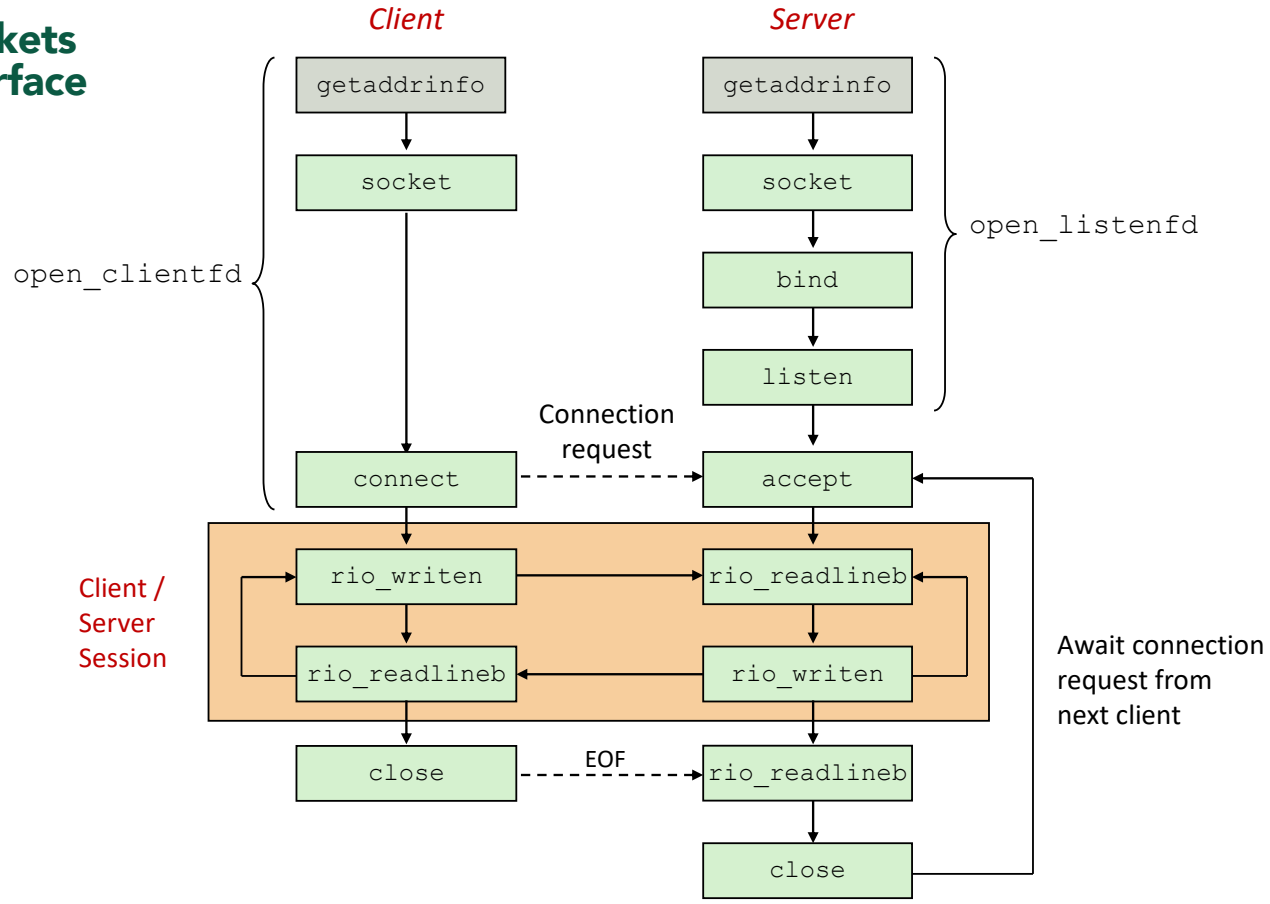
2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

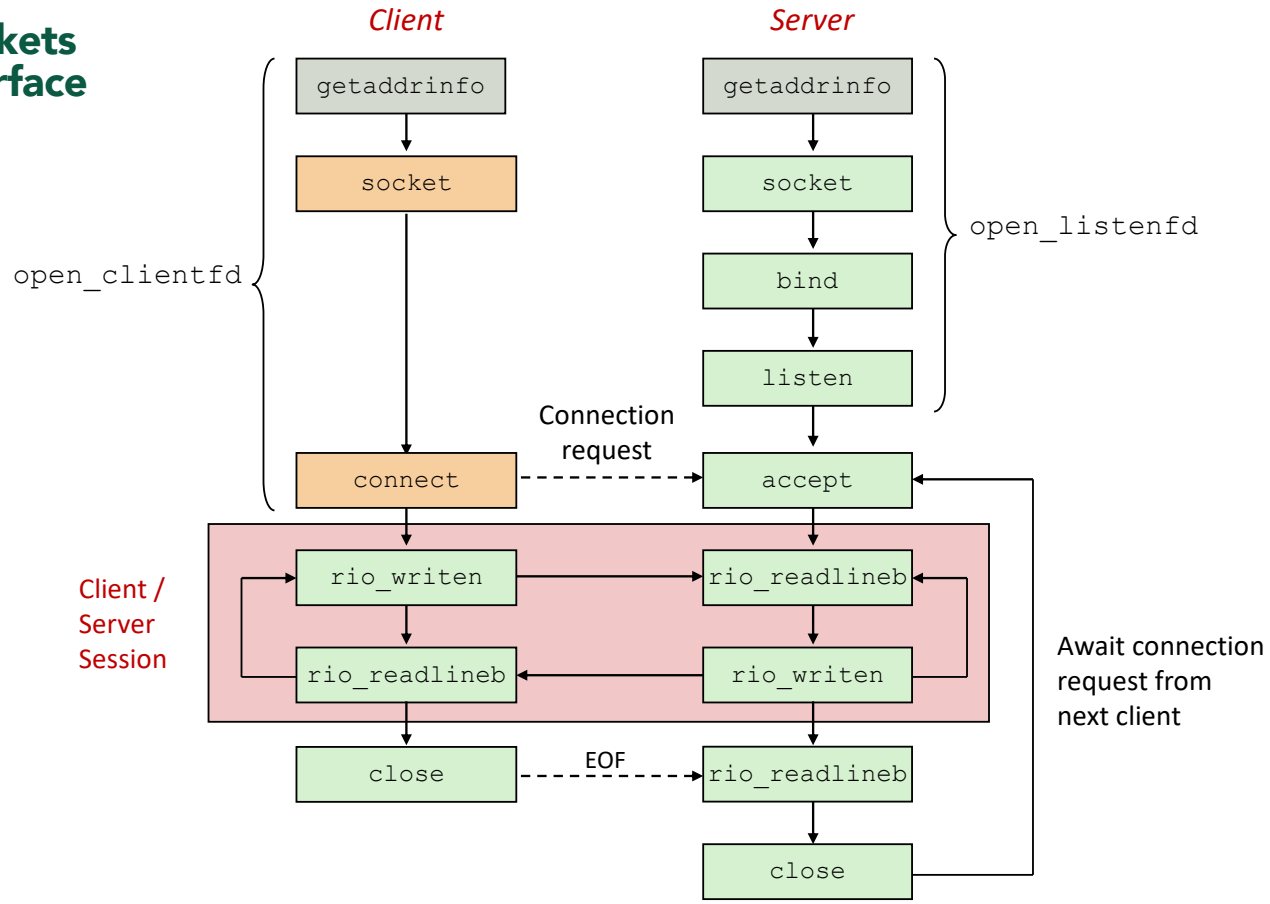
Connected vs. Listening Descriptors

- Listening descriptor
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- Connected descriptor
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client
- Why the distinction?
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Sockets Interface



Sockets Interface



Sockets Helper: `open_clientfd`

- Establish a connection with a server

```
int open_clientfd(char *hostname, char *port) {
  int clientfd;
  struct addrinfo hints, *listp, *p;

  /* Get a list of potential server addresses */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
  hints.ai_flags = AI_NUMERICSERV;  /* …using numeric port arg. */
  hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
  getaddrinfo(hostname, port, &hints, &listp);
```

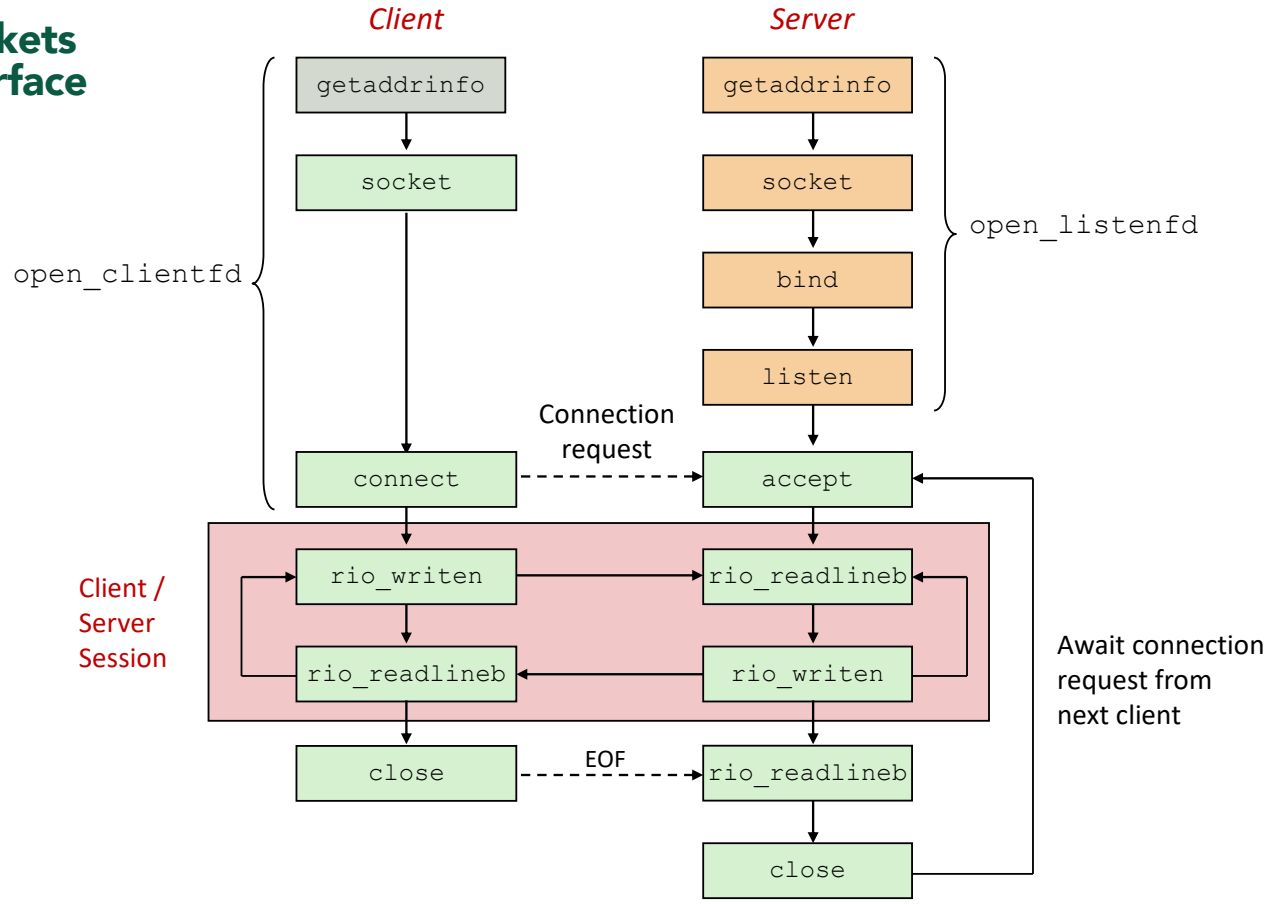
csapp.c

Sockets Helper: `open_clientfd` (cont)

```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        Close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```

csapp.c

Sockets Interface



Sockets Helper: `open_listenfd`

- Create a listening descriptor that can be used to accept connection requests from clients.

```
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;             /* Accept connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* …on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;            /* …using port no. */
    getaddrinfo(NULL, port, &hints, &listp);
```

csapp.c

Sockets Helper: `open_listenfd` (cont)

```
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        close(listenfd); /* Bind failed, try the next */
    }
```

csapp.c

Sockets Helper: open_listenfd (cont)

- Key point: `open_clientfd` and `open_listenfd` are both independent of any particular version of IP.

```
    /* Clean up */
    freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept conn. requests */
    if (listen(listenfd, LISTENQ) < 0) {
        close(listenfd);
        return -1;
    }
    return listenfd;
}
```

csapp.c

Echo Client: Main Routine

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    rio_readinitb(&rio, clientfd);

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        rio_writen(clientfd, buf, strlen(buf));
        rio_readlineb(&rio, buf, MAXLINE);
        fputs(buf, stdout);
    }
    close(clientfd);
    exit(0);
}
```

echoclient.c

Iterative Echo Server: Main Routine

```c
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        close(connfd);
    }
    exit(0);
}
```

echoserveri.c

Echo Server: `echo` function

- The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.
  - EOF condition caused by client calling **`close(clientfd)`**

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```

echo.c

Testing Servers Using `telnet`

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers

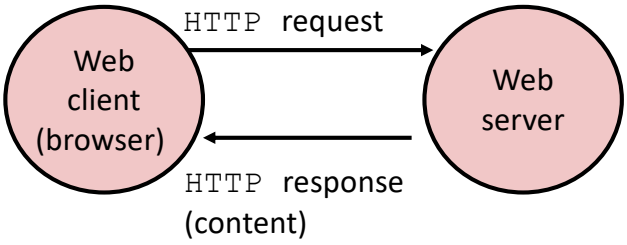
- Usage:
  - **`linux> telnet <host> <portnumber>`**
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

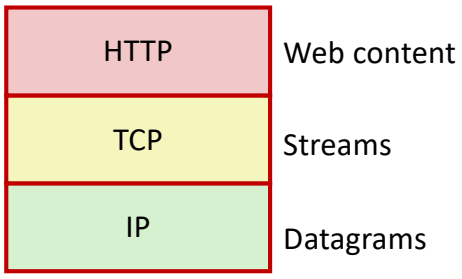
Testing the Echo Server With `telnet`

```
> ./echoserveri 15213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes



> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
Hi there!
Hi there!
Howdy!
Howdy!
^]
telnet> quit
Connection closed.
makoshark>
```


Web Server Basics

- Clients and servers communicate using the HyperText Transfer Protocol (HTTP)
  - Client and server establish TCP connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (eventually)
- Current version is HTTP/1.1
  - RFC 2616, June, 1999.





`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Web Content

- Web servers return *content* to clients
  - *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- Example MIME types

– `text/html`	HTML document
– `text/plain`	Unformatted text
– `image/gif`	Binary image encoded in GIF format
– `image/png`	Binar image encoded in PNG format
– `image/jpeg`	Binary image encoded in JPEG format

You can find the complete list of MIME types at:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

Static and Dynamic Content

- The content returned in HTTP responses can be either *static* or *dynamic*
  - *Static content*: content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips
    - Request identifies which content file
  - *Dynamic content*: content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client
    - Request identifies file containing executable code
- Bottom line: *Web content is associated with a file that is managed by the server*

URLs and how clients and servers use them

- Unique name for a file: URL (Universal Resource Locator)
- Example URL: `http://www.cmu.edu:80/index.html`
- Clients use *prefix* (`http://www.cmu.edu:80`) to infer:
  - What kind (protocol) of server to contact (HTTP)
  - Where the server is (`www.cmu.edu`)
  - What port it is listening on (80)
- Servers use *suffix* (`/index.html`) to:
  - Determine if request is for static or dynamic content.
    - No hard and fast rules for this
    - One convention: executables reside in `cgi-bin` directory
  - Find file on file system
    - Initial "`/`" in suffix denotes home directory for requested content.
    - Minimal suffix is "`/`", which server expands to configured default filename (usually, `index.html`)

HTTP Requests

- HTTP request is a *request line*, followed by zero or more *request headers*
- Request line: `<method> <uri> <version>`
  - `<method>` is one of `GET`, `POST`, `OPTIONS`, `HEAD`, `PUT`, `DELETE`, or `TRACE`
  - `<uri>` is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
    - See http://www.ietf.org/rfc/rfc2396.txt
  - `<version>` is HTTP version of request (`HTTP/1.0` or `HTTP/1.1`)
- Request headers: `<header name>: <header data>`
  - Provide additional information to the server

HTTP Responses

- HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line ("\r\n") separating headers from content.

- Response line:

  `<version> <status code> <status msg>`

  - <version> is HTTP version of the response
  - <status code> is numeric status
  - <status msg> is corresponding English text
    - 200 OK Request was handled without error
    - 301 Moved Provide alternate URL
    - 404 Not found Server couldn't find the file
- Response headers: `<header name>: <header data>`
  - Provide additional information about response
  - `Content-Type:` MIME type of content in response body
  - `Content-Length:` Length of content in response body

Example HTTP Transaction

```
whaleshark> telnet www.cmu.edu 80        Client: open connection to server
Trying 128.2.42.52...                    Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET / HTTP/1.1                           Client: request line
Host: www.cmu.edu                        Client: required HTTP/1.1 header
                                         Client: empty line terminates headers
HTTP/1.1 301 Moved Permanently           Server: response line
Date: Wed, 05 Nov 2014 17:05:11 GMT      Server: followed by 5 response headers
Server: Apache/1.3.42 (Unix)             Server: this is an Apache server
Location: http://www.cmu.edu/index.shtml Server: page has moved here
Transfer-Encoding: chunked               Server: response body will be chunked
Content-Type: text/html; charset=...     Server: expect HTML in response body
                                         Server: empty line terminates headers
15c                                      Server: first line in response body
<HTML><HEAD>                             Server: start of HTML content
...
</BODY></HTML>                           Server: end of HTML content
0                                        Server: last line in response body
Connection closed by foreign host.       Server: closes connection
```

- **HTTP standard requires that each text line end with "\r\n"**
- **Blank line ("\r\n") terminates request and response headers**

Example HTTP Transaction, Take 2

```
whaleshark> telnet www.cmu.edu 80        Client: open connection to server
Trying 128.2.42.52...                    Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1                Client: request line
Host: www.cmu.edu                        Client: required HTTP/1.1 header
                                         Client: empty line terminates headers
HTTP/1.1 200 OK                          Server: response line
Date: Wed, 05 Nov 2014 17:37:26 GMT      Server: followed by 4 response headers
Server: Apache/1.3.42 (Unix)
Transfer-Encoding: chunked
Content-Type: text/html; charset=...
                                         Server: empty line terminates headers
1000                                     Server: begin response body
<html ..>                                Server: first line of HTML content
…
</html>
0                                        Server: end response body
Connection closed by foreign host.       Server: close connection
```


Tiny Web Server

- Tiny Web server described in text
  - Tiny is a sequential Web server
  - Serves static and dynamic content to real browsers
    - text files, HTML files, GIF, PNG, and JPEG images
  - 239 lines of commented C code
  - Not as complete or robust as a real Web server
    - You can break it with poorly-formed HTTP requests (e.g., terminate lines with "\n" instead of "\r\n")

Tiny Operation

- Accept connection from client
- Read request from client (via connected socket)
- Split into <method> <uri> <version>
  - If method not GET, then return error
- If URI contains "`cgi-bin`" then serve dynamic content
  - (Would do wrong thing if had file "`abcgi-bingo.html`")
  - Fork process to execute program
- Otherwise serve static content
  - Copy file to output

Tiny Serving Static Content

```
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    Rio_writen(fd, buf, strlen(buf));

    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

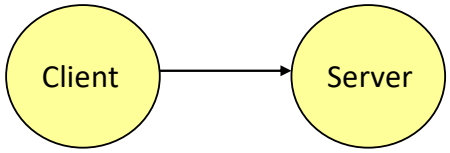
tiny.c

Serving Dynamic Content

- Client sends request to server

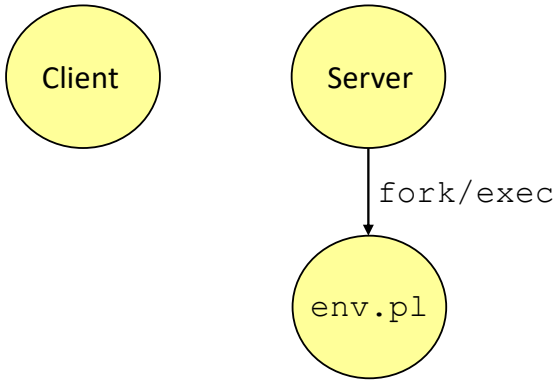
```
GET /cgi-bin/env.pl HTTP/1.1
```

- If request URI contains the string "`/cgi-bin`", the Tiny server assumes that the request is for dynamic content



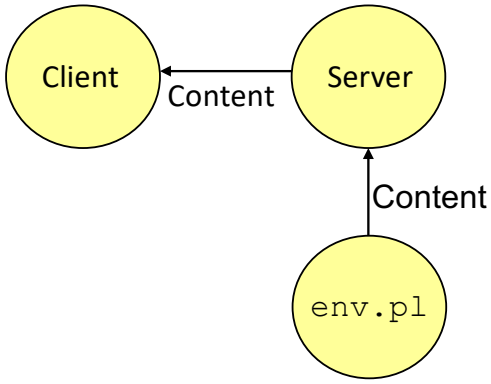
Serving Dynamic Content (cont)

- The server creates a child process and runs the program identified by the URI in that process



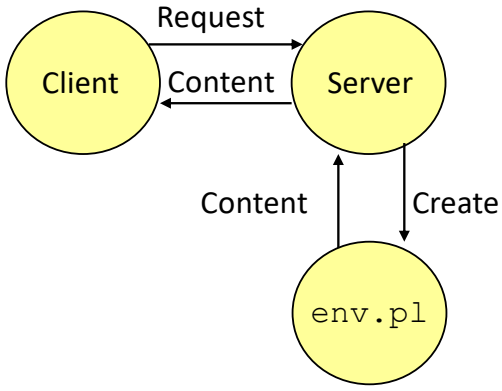
Serving Dynamic Content (cont)

- The child runs and generates the dynamic content
- The server captures the content of the child and forwards it without modification to the client



Issues in Serving Dynamic Content

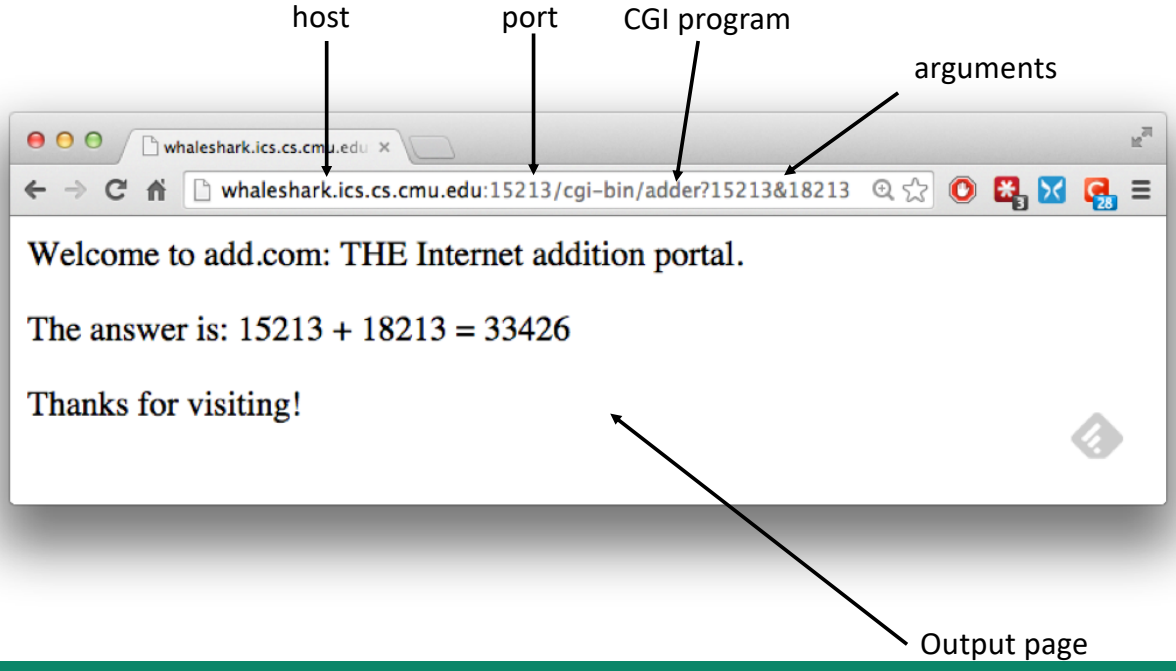
- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common Gateway Interface (CGI) specification.



CGI

- Because the children are written according to the CGI spec, they are often called *CGI programs.*
- However, CGI really defines a simple standard for transferring information between the client (browser), the server, and the child process.
- CGI is the original standard for generating dynamic content. Has been largely replaced by other, faster techniques:
  - E.g., fastCGI, Apache modules, Java servlets, Rails controllers
  - Avoid having to create process on the fly (expensive and slow).

The add.com Experience



Serving Dynamic Content With GET

- **Question:** How does the client pass arguments to the server?
- **Answer:** The arguments are appended to the URI

- Can be encoded directly in a URL typed to a browser or a URL in an HTML link
  - `http://add.com/cgi-bin/adder?15213&18213`
  - `adder` is the CGI program on the server that will do the addition.
  - argument list starts with "`?`"
  - arguments separated by "`&`"
  - spaces represented by "`+`" or "`%20`"

Serving Dynamic Content With GET

- URL suffix:
  - `cgi-bin/adder?15213&18213`

- Result displayed on browser:

```
Welcome to add.com: THE Internet addition portal.

The answer is: 15213 + 18213 = 33426

Thanks for visiting!
```

Serving Dynamic Content With GET

- Question: How does the server pass these arguments to the child?
- Answer: In environment variable QUERY_STRING
  - A single string containing everything after the "?"
  - For add: `QUERY_STRING` = "`15213&18213`"

```
    /* Extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }
```

adder.c

Serving Dynamic Content with GET

- Question: How does the server capture the content produced by the child?
- Answer: The child generates its output on `stdout`. Server uses `dup2` to redirect `stdout` to its connected socket.

Serving Dynamic Content with GET

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
{
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) { /* Child */
        /* Real server would set all CGI vars here */
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
        Execve(filename, emptylist, environ); /* Run CGI program */
    }
    Wait(NULL); /* Parent waits for and reaps child */
}
```


Serving Dynamic Content with GET

- Notice that only the CGI child process knows the content type and length, so it must generate those headers.

```
/* Make the response body */
sprintf(content, "Welcome to add.com: ");
sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
        content, n1, n2, n1 + n2);
sprintf(content, "%sThanks for visiting!\r\n", content);

/* Generate the HTTP response */
printf("Content-length: %d\r\n", (int)strlen(content));
printf("Content-type: text/html\r\n\r\n");
printf("%s", content);
fflush(stdout);

exit(0);
```

adder.c

Serving Dynamic Content With GET

```
bash:makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
GET /cgi-bin/adder?15213&18213 HTTP/1.0

HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 117
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 15213 + 18213 = 33426
<p>Thanks for visiting!
Connection closed by foreign host.
bash:makoshark>
```

HTTP request sent by client

HTTP response generated by the server

HTTP response generated by the CGI program

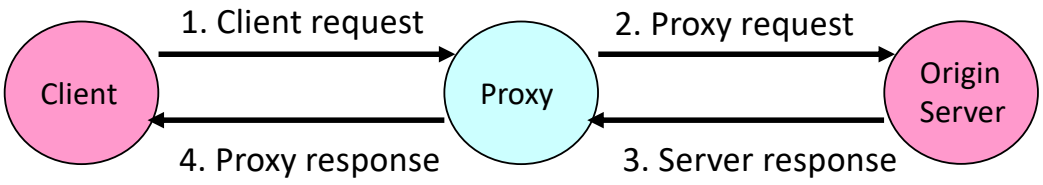
For More Information

- W. Richard Stevens et. al. “Unix Network Programming: The Sockets Networking API”, Volume 1, Third Edition, Prentice Hall, 2003
  - THE network programming bible.
- Michael Kerrisk, “The Linux Programming Interface”, No Starch Press, 2010
  - THE Linux programming bible.
- Complete versions of all code in this lecture is available from the 213 schedule page.
  - `http://www.cs.cmu.edu/~213/schedule.html`
  - csapp.{.c,h}, hostinfo.c, echoclient.c, echoserveri.c, tiny.c, adder.c
  - You can use any of this code in your assignments.

Proxy

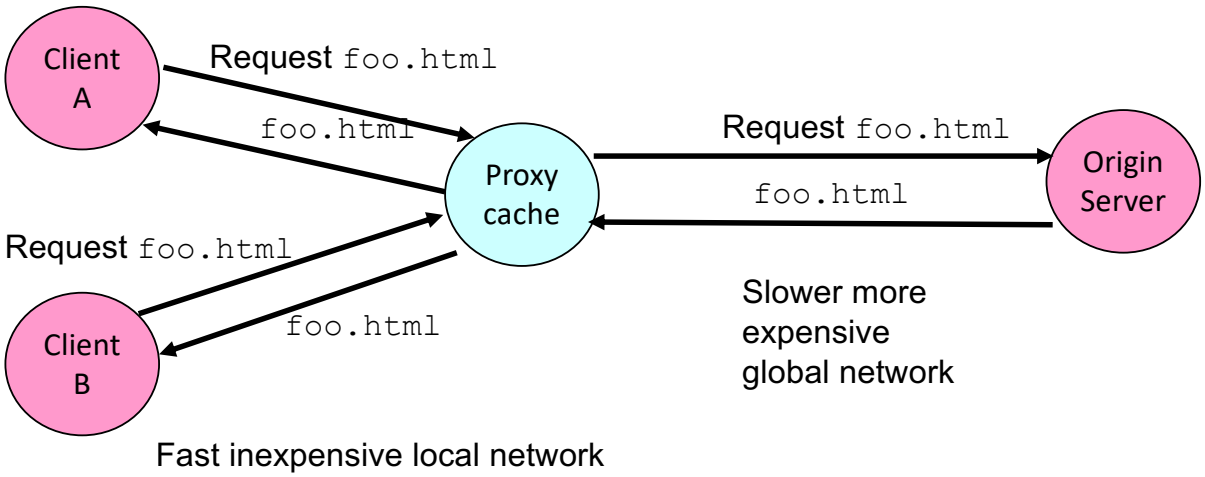
Proxies

- A *proxy* is an intermediary between a client and an *origin server*
  - To the client, the proxy acts like a server
  - To the server, the proxy acts like a client



Why Proxies?

- Can perform useful functions as requests and responses pass by
  - Examples: Caching, logging, anonymization, filtering, transcoding



LAU
الجامعة اللبنانية الأميركية
Lebanese American University

C Example

```c
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<netdb.h>
#include<arpa/inet.h>

int main(int argc , char *argv[])
{
  char *hostname = "www.google.com";
  char ip[100];
  struct hostent *he;
  struct in_addr **addr_list;
  int i;

  if ( (he = gethostbyname( hostname ) ) == NULL)
    {
      //gethostbyname failed
      herror("gethostbyname");
      return 1;
    }
```



```c
//Cast the h_addr_list to in_addr , since h_addr_list also
// has the ip address in long format only

  addr_list = (struct in_addr **) he->h_addr_list;

  for(i = 0; addr_list[i] != NULL; i++)
    {
      //Return the first one;
      strcpy(ip , inet_ntoa(*addr_list[i]) );
    }

  printf("%s resolved to : %s" , hostname , ip);
  return 0;
}
```

Network/Socket Programming in Python

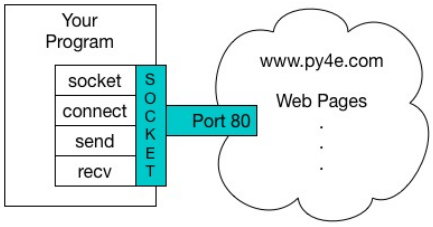
Sockets in Python

- Python has built-in support for TCP Sockets

```
import socket
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect( ('data.pr4e.org', 80) )
```

An HTTP Request in Python

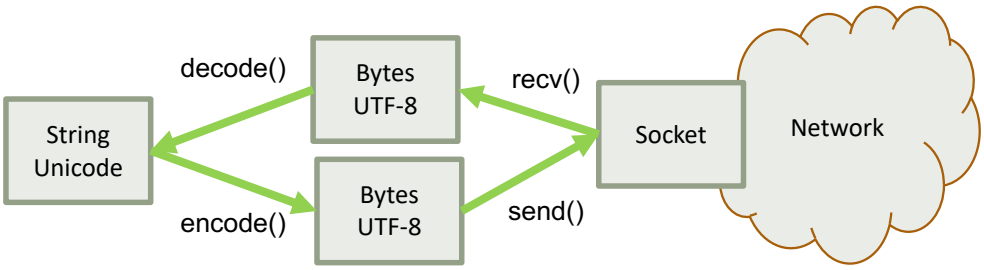
```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\n\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if (len(data) < 1):
        break
    print(data.decode())
mysock.close()
```




```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
Content-Length: 167
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

HTTP Header

```
while True:
    data = mysock.recv(512)
    if ( len(data) < 1 ) :
        break
    print(data.decode())
```




```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\n\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if (len(data) < 1):
        break
    print(data.decode())
mysock.close()
```


Using urllib in Python

- Since HTTP is so common, we have a library that does all the socket work for us and makes web pages look like a file

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())
```

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())
```

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief

Like a File...

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)
```

Reading Web Pages

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://www.dr-chuck.com/page1.htm')
for line in fhand:
    print(line.decode().strip())
```

```
<h1>The First Page</h1>
<p>If you like, you can switch to the <a
href="http://www.dr-chuck.com/page2.htm">Second
Page</a>.
</p>
```


Following Links

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://www.dr-chuck.com/page1.htm')
for line in fhand:
    print(line.decode().strip())
```

```
<h1>The First Page</h1>
<p>If you like, you can switch to the <a
href="http://www.dr-chuck.com/page2.htm">Second
Page</a>.
</p>
```

The First Lines of Code Google?

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://www.dr-chuck.com/page1.htm')
for line in fhand:
    print(line.decode().strip())
```


Parsing HTML (a.k.a. Web Scraping)

What is Web Scraping?

- When a program or script pretends to be a browser and retrieves web pages, looks at those web pages, extracts information, and then looks at more web pages
- Search engines scrape web pages - we call this “spidering the web” or “web crawling”

Why Scrape?

- Pull data - particularly social data - who links to who?
- Get your own data back out of some system that has no “export capability”
- Monitor a site for new information
- Spider the web to make a database for a search engine

Scraping Web Pages

- There is some controversy about web page scraping and some sites are a bit snippy about it.
- Republishing copyrighted information is not allowed
- Violating terms of service is not allowed

The Easy Way - Beautiful Soup

- You could do string searches the hard way
- Or use the free software library called BeautifulSoup from www.crummy.com

BeautifulSoup Installation

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup

...
```

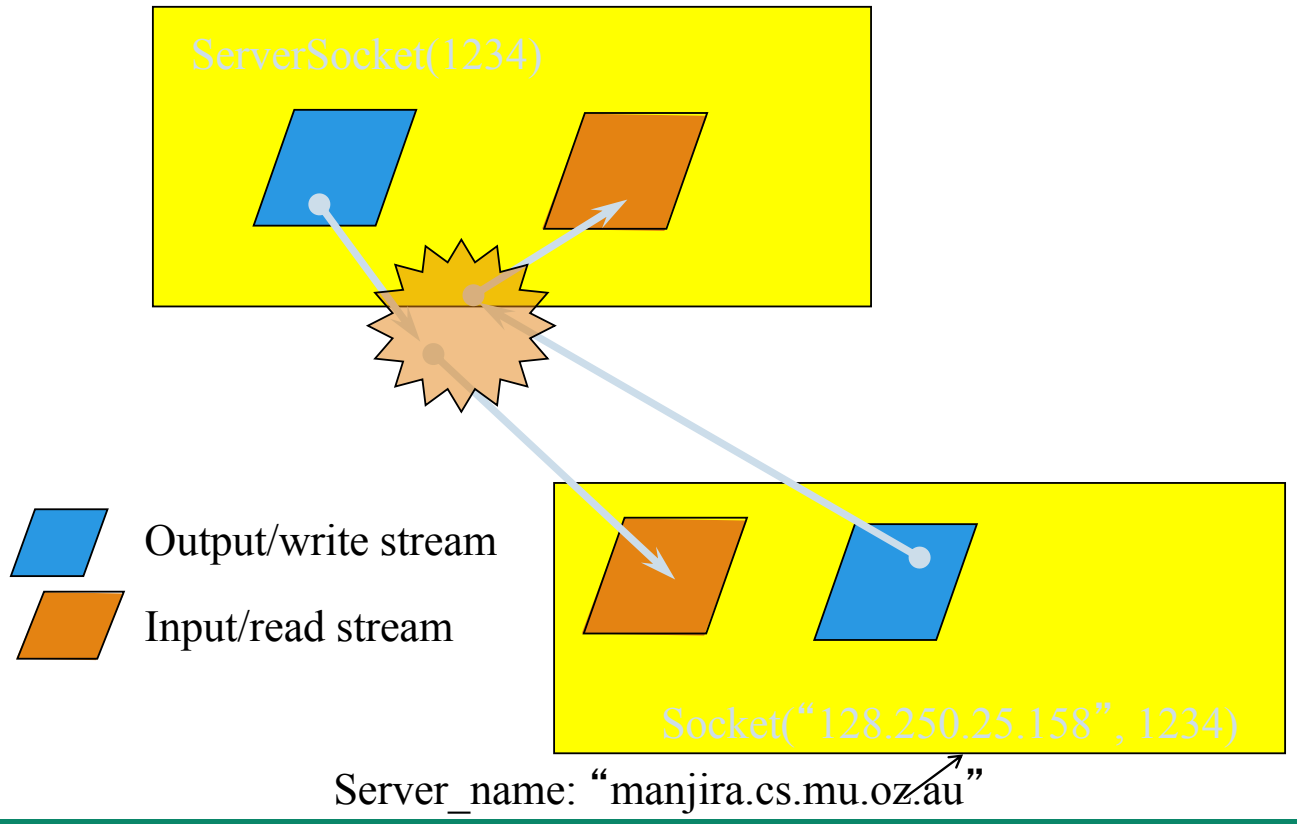
urllinks.p


```
import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup

url = input('Enter - ')
html = urllib.request.urlopen(url).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))
```

Network/Socket Programming in Java

java.net

- Used to manage:
  - URL streams
  - Client/server sockets
  - Datagrams



Server side Socket Operations

1. Open Server Socket:
   ServerSocket server;
   DataOutputStream os;
   DataInputStream is;
   server = new ServerSocket(PORT);
2. Wait for Client Request:
   Socket client = server.accept();
3. Create I/O streams for communicating to clients
   is = new DataInputStream(client.getInputStream());
   os = new DataOutputStream(client.getOutputStream());
4. Perform communication with client
   Receive from client: String line = is.readLine();
   Send to client: os.writeBytes("Hello\n");
5. Close sockets: client.close();

For multithreaded server:

while(true) {
   i. wait for client requests (step 2 above)
   ii. create a thread with "client" socket as parameter (the thread creates streams (as in step (3) and does communication as stated in (4). Remove thread once service is provided.
}

Client side Socket Operations

1. Get connection to server:
   client = new Socket(server, port_id);
2. Create I/O streams for communicating to clients
   is = new DataInputStream(client.getInputStream());
   os = new DataOutputStream(client.getOutputStream());
3. Perform communication with client
   Receiive from client: String line = is.readLine();
   Send to client: os.writeBytes("Hello\n");
4. Close sockets: client.close();

A simple client (simplified code)

```
import java.net.*;
import java.io.*;
public class SimpleClient {
  public static void main(String args[]) throws IOException {
        // Open your connection to a server, at port 1234
        Socket s1 = new Socket("130.63.122.1",1234);
        // Get an input file handle from the socket and read the input
        InputStream s1In = s1.getInputStream();
        DataInputStream dis = new DataInputStream(s1In);
        String st = new String (dis.readUTF());
        System.out.println(st);
        // When done, just close the connection and exit
        dis.close();
        s1In.close();
        s1.close();
  }
}
```

A simple server (simplified code)

```
import java.net.*;
import java.io.*;
public class ASimpleServer {
  public static void main(String args[]) {
        // Register service on port 1234
        ServerSocket s = new ServerSocket(1234);
        Socket s1=s.accept(); // Wait and accept a connection
        // Get a communication stream associated with the socket
        OutputStream s1out = s1.getOutputStream();
        DataOutputStream dos = new DataOutputStream (s1out);
        // Send a string!
        dos.writeUTF("Hi there");
        // Close the connection, but not the server socket
        dos.close();
        s1out.close();
        s1.close();
  }
}
```


Final Thoughts – Distributed Applications

- Master computer distributes work through network
- Each computer completes its task and returns result
- Internet allows large distributed computing projects

Final Thoughts – Distributed Applications

- Also called “grid applications”
- A large number exists today
  - www.aspenleaf.com/distributed/distrib-projects.html
- Most are voluntary
  - Intel philanthropic peer-to-peer program
  - http://www.intel.com/cure/
- Some are hidden
  - KaZaA