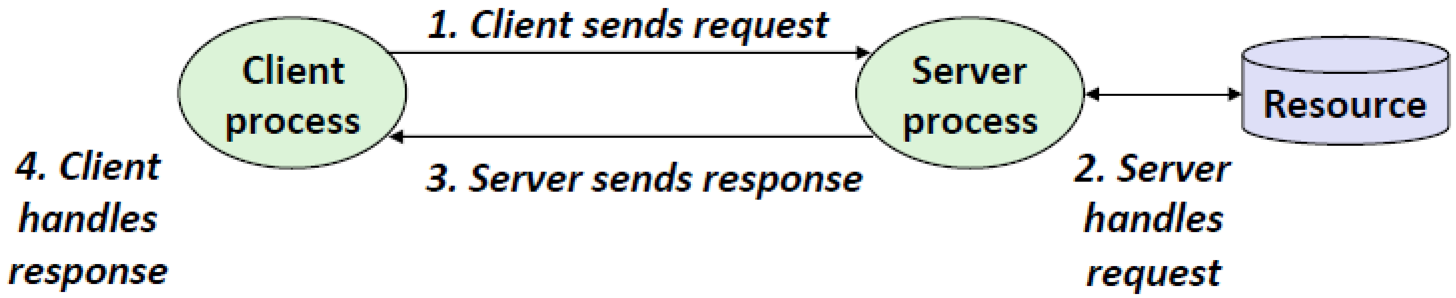
Systems Software/Programming

Network Programming

A Client-Server Transaction

- Most network application are based on client-server model
  - A server process and one or more client processes
  - Server manages some resource
  - Server provides services to client by managing client resources
  - Server activated by request from client

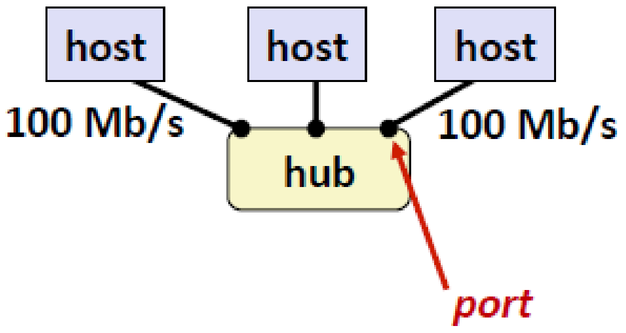


Note: clients and servers are processes running on hosts (can be the same or different hosts)

Computer Network

- A network is a hierarchical system of boxes and wires organized by geographical proximity
  - SAN (System Area Network) spans cluster or machine room Switched Ethernet, Quadrics QSW, …
  - LAN (Local Area Network) spans a building or campus Ethernet is most prominent example
  - WAN (Wide Area Network) spans country or world Typically high--speed point--to--point phone lines
- An internetwork (internet) is an interconnected set of networks The Global IP Internet (uppercase "I") is the most famous example of an internet (lowercase "i")
- Let's see how an internet is built from the ground up

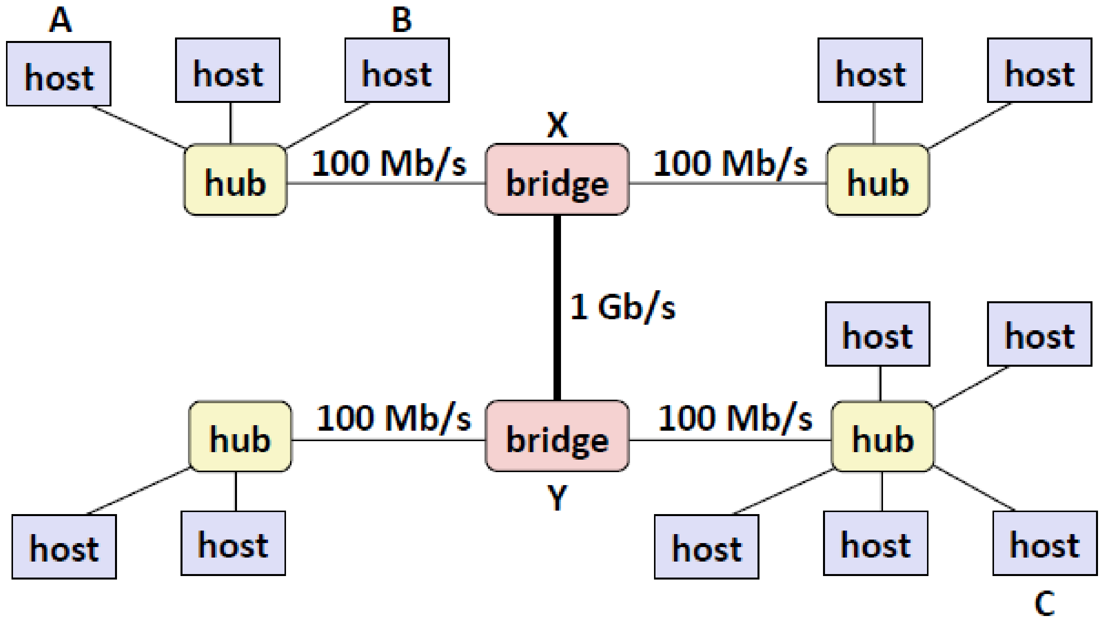
Lowest Level : Ethernet Segment



- Ethernet segment consists of a collection of hosts connected by wires (twisted pairs) to a hub
- Spans room or floor in a building
- Operation
  - Each Ethernet adapter has a unique 48--bit address (MAC address)
  - E.g., 00:16:ea:e3:54:e6
  - Hosts send bits to any other host in chunks called frames
  - Hub slavishly copies each bit from each port to every other port
    - Every host sees every bit
    - Note: Hubs are on their way out. Bridges (switches, routers) became cheap enough to replace them

Next Level : Bridged Ethernet Segment

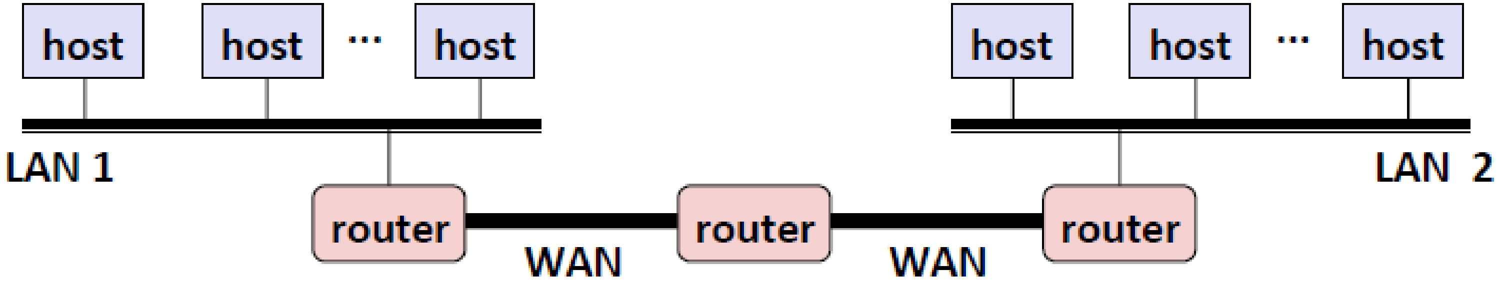
- Spans building or campus
- Bridges cleverly learn which hosts are reachable from which ports and then selectively copies frames from port to port



For simplicity hubs and bridges including wires are shown as a line called Local Area Network (LAN)

Next Level : internets (internetworks)

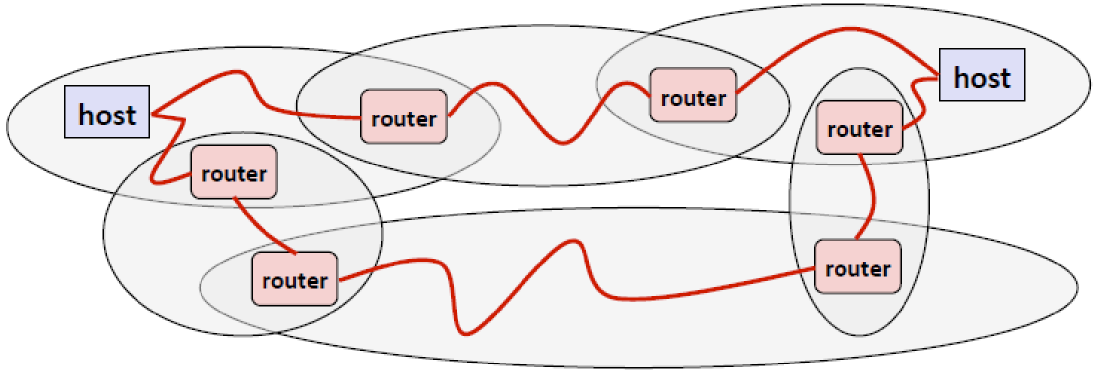
- Multiple incompatible LANs can be physically connected by special computers called routers
- The connected networks are called internet



LAN 1 and LAN 2 might be completely different, totally incompatible (e.g., Ethernet, Fibre Channel, 802.11, T1-links, DSL, ...)*

Logical Structure of an internet

- Ad hoc interconnection of networks
  - No particular topology
  - Vastly different routers and link capacities
- Send packets from source to destination by hoping through networks
  - Router forms bridge from one network to another
  - Different packets may take different routes



Notion of internet Protocol

- How is it possible to send bits across incompatible LANs and WANs?
- Solution: protocol software running on each host and router
  - Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network.
  - Smooths out the differences between the different networks

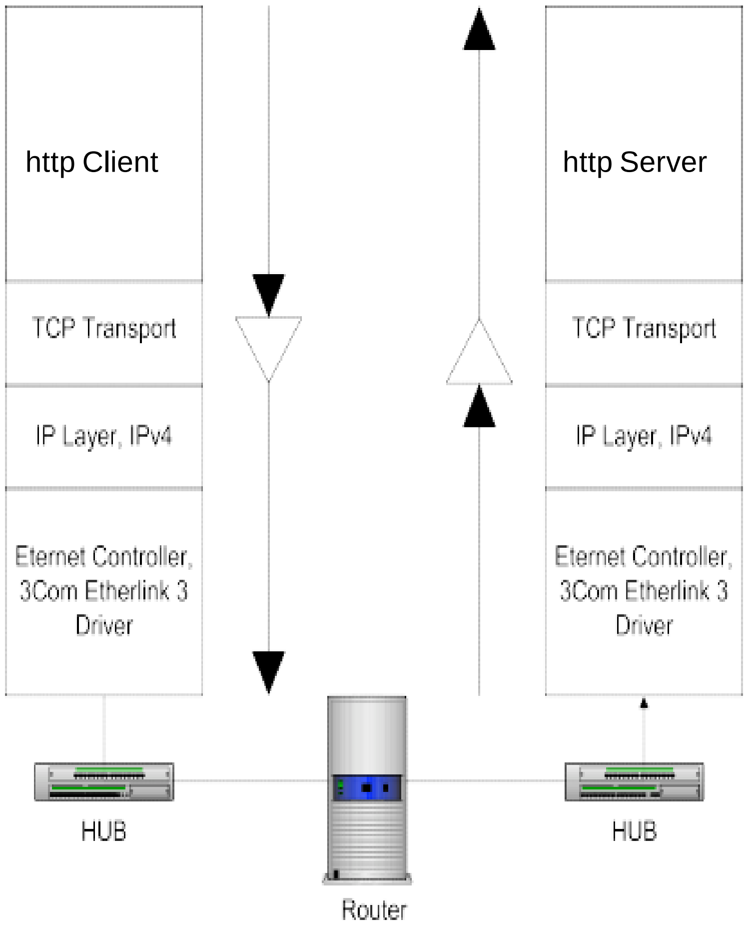
What Does internet Protocol Do?

- Provides a naming scheme
  - An internet protocol defines a uniform format for host addresses
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it
- Provides a delivery mechanism
  - An internet protocol defines a standard transfer unit (packet)
  - Packet consists of header and payload
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

Global IP (Internet Protocol)

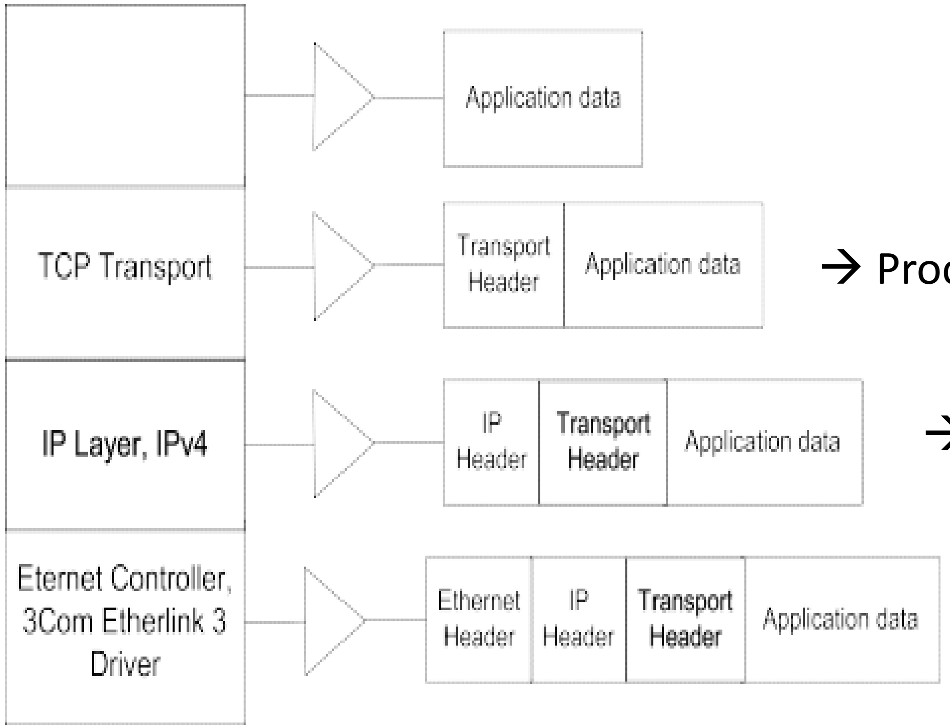
- Based on the TCP/IP protocol family
  - IP (Internet Protocol) :
    - Provides basic naming scheme and unreliable delivery capability of packets (datagrams) from host--to--host
  - UDP (User Datagram Protocol)
    - Uses IP to provide unreliable datagram delivery from process--to--process
  - TCP (Transmission Control Protocol)
    - Uses IP to provide reliable byte streams from process--to--process over connections
- Accessed via a mix of Unix file I/O and functions from the sockets interface

How Client Communicate to Server ?



How Client Communicate to Server ?

- Client Application puts data through sockets
- Each successive layer wraps the received data with its own header:



→ Process to Process Communication

→ Host to Host Communication

TCP/IP Header Formats

IP Header Format

- Packets may be broken up called fragments if data is too large to fit in a single packet
- Packets if not delivered will live in the network till Time-To-Live

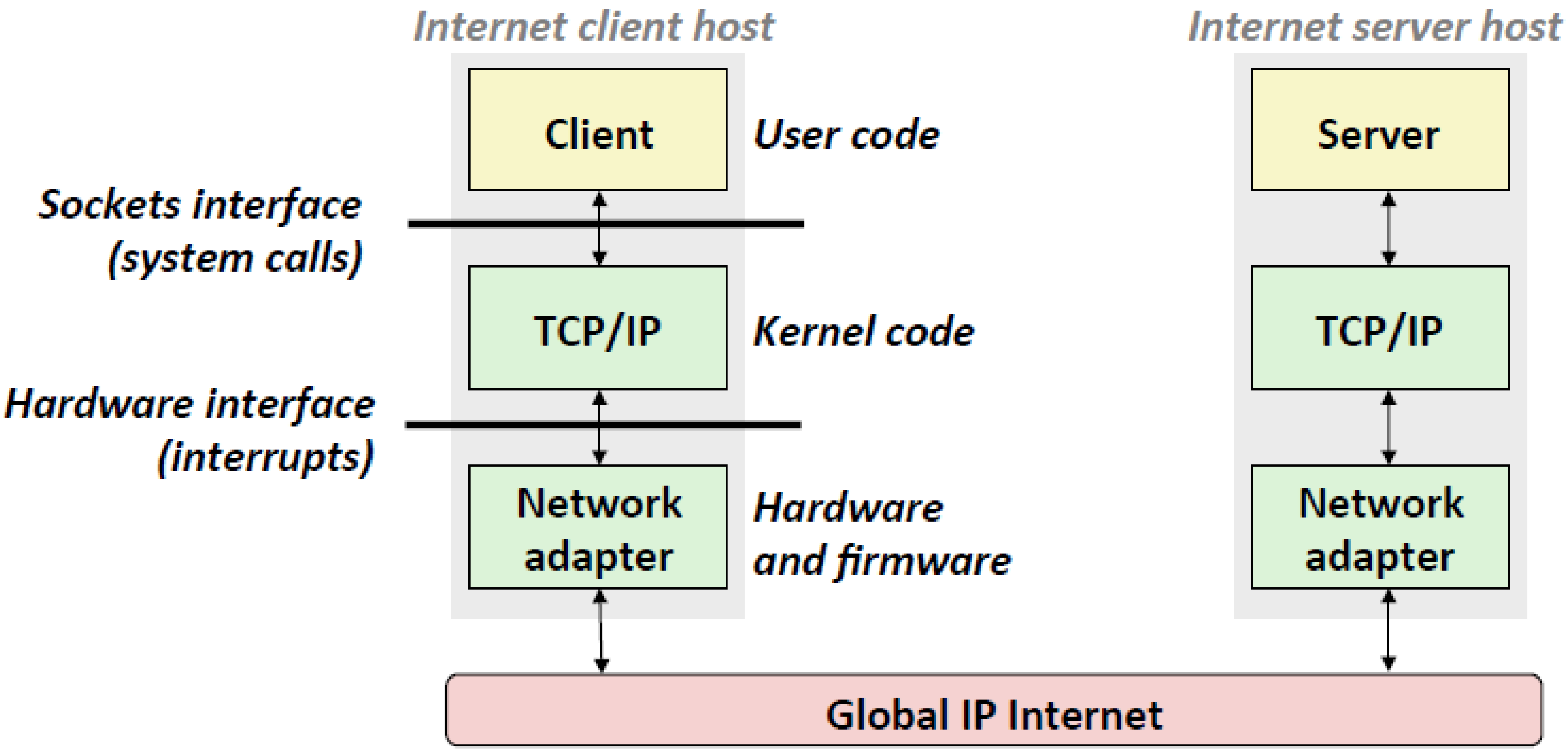
Preamble	Length of data	Fragmentation Information (if it's too big for an ethernet frame buffer)	Time To Live	Protocol (TCP, UDP)	Checksum	Source Address (192.32.63.5)	Destination Address (192.32.65.1)	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	1 byte	1 byte	2 bytes	4 bytes	4 bytes	variable	variable

TCP Header Format

- Source and Destination ports
- Sequence number indicates which byte in overall data this segment starts with
- Acknowledgement number indicates all bytes up to which recipient has received successfully

Source Port	Destination Port	Sequence Number	Acknowledgement Number	Flags	Window Size	Checksum	Urgent Pointer	Options	Datagram (THE DATA) (up to 12k bits)
2 bytes	2 bytes	4 bytes	4 bytes	2 bytes	2 bytes	2 bytes	2 bytes	variable	variable

Hardware and Software Organization of an Internet Application



Internet From a Programmers View

1. Hosts are mapped to a set of 32--bit IPv4 addresses e.g. 128.2.203.179
2. The set of IP addresses is mapped to a set of identifiers called Internet domain names.
   - 104.238.110.159 is mapped to www.daiict.ac.in

   $ ping www.daiict.ac.in

   PING www.daiict.ac.in (104.238.110.159) 56(84) bytes of data.

   64 bytes from ip-104-238-110-159.ip.secureserver.net (104.238.110.159): icmp_seq=1 ttl=57 time=349 ms

   ^C

   --- www.daiict.ac.in ping statistics ---

   1 packets transmitted, 1 received, 0% packet loss, time 0ms

   rtt min/avg/max/mdev = 349.388/349.388/349.388/0.000 ms

   You can also use https://www.whatismyip.com/dns-lookup/ to get IP address mapped to domain name.
3. A process on one Internet host can communicate with a process on another Internet host over a Internet connection

(1) 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) */
};
```

- By convention, each byte in a 32--bit IP address is represented by its decimal value and separated by a period
  - IP address: 0x8002C2F2 = 128.2.194.242 → Dotted Decimal Format
  - Big-endian

	1000	1001	1002	1003	
LSB	0xF2	0xC2	0x02	0x80	MSB

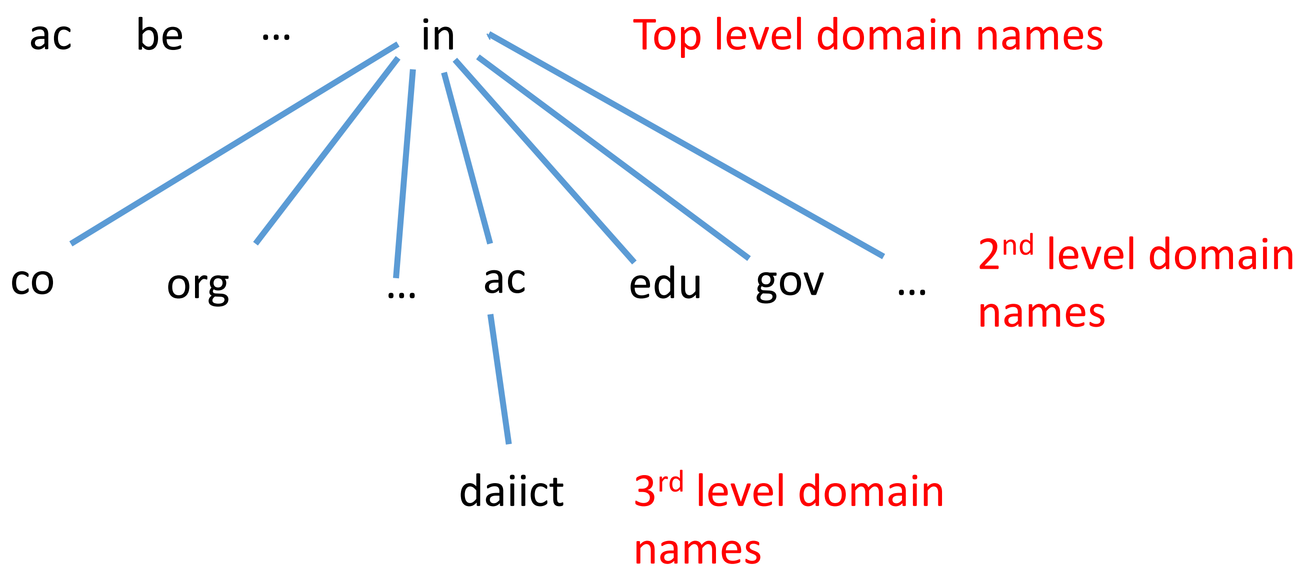
Little-endian

	1000	1001	1002	1003	
MSB	0x80	0x02	0xC2	0xF2	LSB

  - Use getaddrinfo and getnameinfo functions to convert between IP addresses and doted decimal format.

(2) Internet Domain Names

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS
- Conceptually, programmers can view the DNS database as a collection of millions of host entries.
  - Each host entry defines the mapping between a set of domain names and IP addresses.
  - In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.



Properties of DNS Mappings

- Can explore properties of DNS mappings using nslookup
  - Output edited for brevity
- Each host has a locally defined domain name localhost which always maps to the loopback address 127.0.0.1

$ nslookup localhost

Address: 127.0.0.1

- Use hostname to determine real domain name of local host:

$ hostname

faculty-OptiPlex-3040

- Simple case: one--to--one mapping between domain name and IP address:

$ nslookup abel.daiict.ac.in

Address: 10.100.71.142

Properties of DNS Mappings

- Multiple domain names mapped to the same IP address:

$ nslookup cs.mit.edu
Address: 18.25.0.23
$ nslookup eecs.mit.edu
Address: 18.25.0.23

- Same domain names mapped to multiple IP addresses

$ nslookup www.google.com
Address: 74.125.200.103
Address: 74.125.200.105
Address: 74.125.200.104
Address: 74.125.200.99
Address: 74.125.200.106

(3) 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)

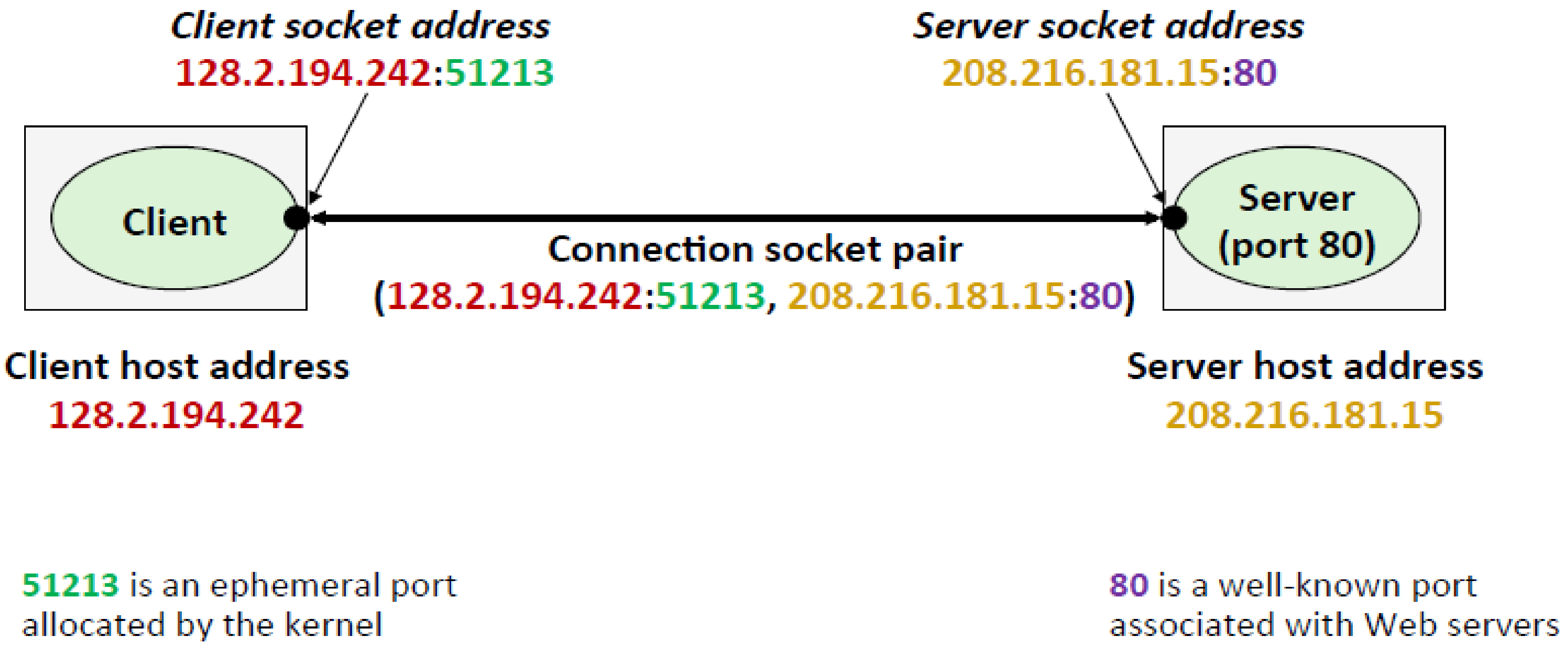
Well-known Ports and Service Names

- 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
  - File Transfer Protocol server : 21/ftp

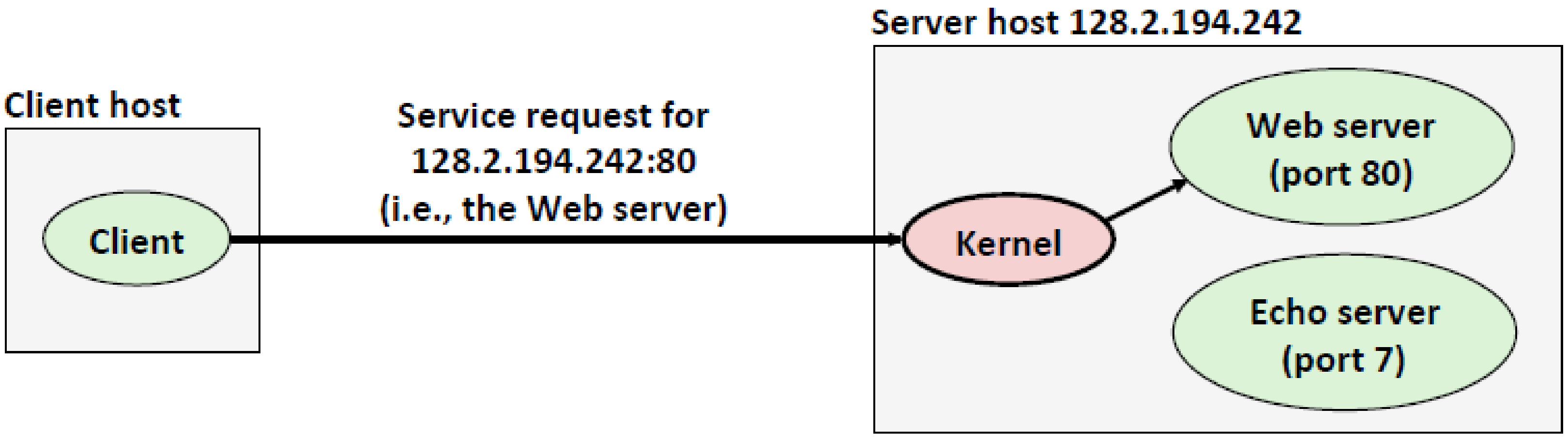
- Mappings between well--known ports and service names is contained in the file /etc/services on each Linux machine.

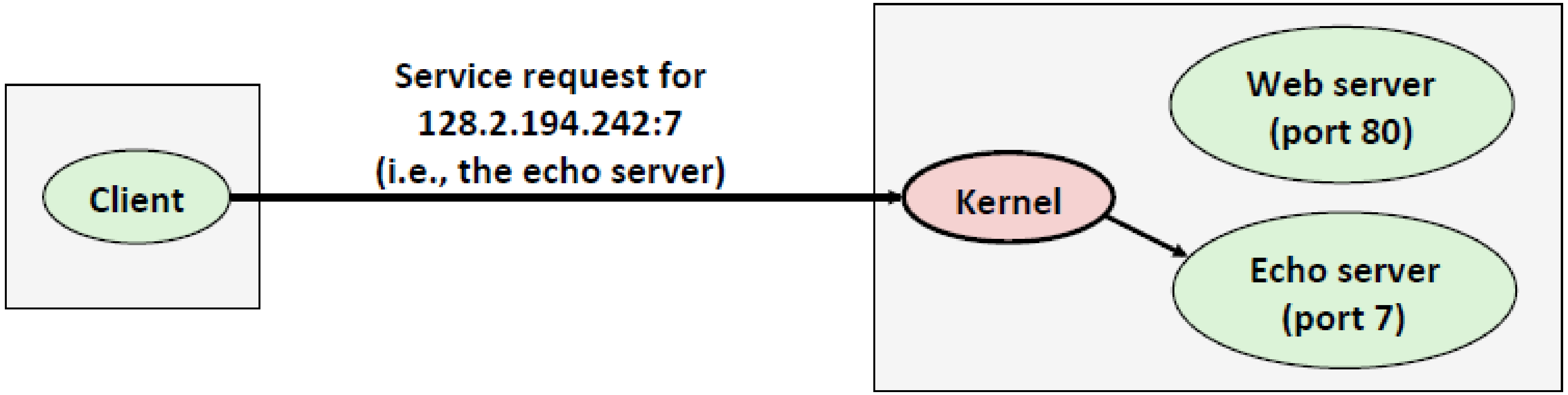
Anatomy of Connection

- connection is uniquely identified by the socket addresses of its endpoints (socket pair) :
  - (clientIPaddr:clientport, serverIPaddr:serverport)



Using Ports to Identify Services



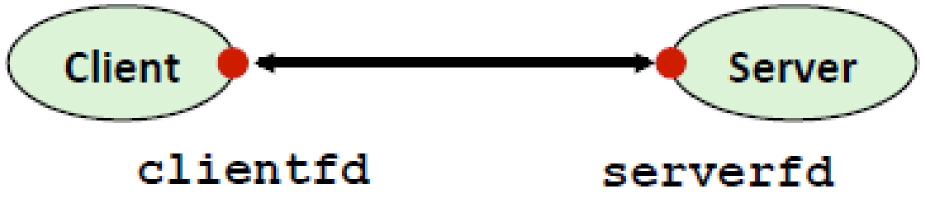


Socket Interface

- 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

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

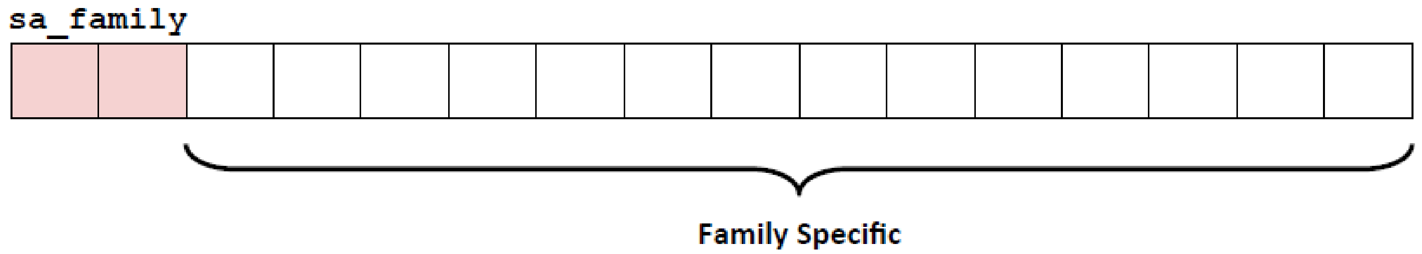


- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors

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 Stevens convention:

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



Socket Address Structures

- Internet-specific socket address IPv4: Must cast (struct sockaddr_in *) to (struct sockaddr *) for functions that take socket address arguments.

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

Internet-specific socket address IPv6

```
struct in6_addr {
        unsigned char s6_addr[16]; /* IPv6 address */
};

struct sockaddr_in6 {
 sa_family_t sin6_family; /* AF_INET6 */
in_port_t sin6_port; /* port number */
uint32_t sin6_flowinfo; /* IPv6 flow information */
struct in6_addr sin6_addr; /* IPv6 address */
uint32_t sin6_scope_id; /* Scope ID (new in 2.4) */
 };
```

Get IP address for a given hostname using hostent structure

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
struct hostent {
        char  *h_name;          /* official name of host */
        char **h_aliases;       /* alias list */
        int    h_addrtype;      /* host address type */
        int    h_length;        /* length of address */
        char **h_addr_list;     /* list of addresses */
    }
    #define h_addr h_addr_list[0] /* for backward compatibility */
```

gethostbyname() and inet_ntoa()

$ nslookup www.google.com → returns two addresses first IPv4 and second IPv6

Name: www.google.com

Address: 172.217.160.132

Name: www.google.com

Address: 2404:6800:4007:80a::2004

These functions works only for IPv4

struct hostent *gethostbyname(const char *name); → returns ptr to hostent structure given hostname

char *inet_ntoa(struct in_addr in); → convert in_addr network byte order into string

NetworkProgramming\gethostbyname_example.c

$ ./gethostbyname_example.out www.google.com

Hostname: www.google.com

IP Address 1: 172.217.160.132

gethostbyname2() and inet_ntop()

These functions works for IPv4 or IPv6 but not both

struct hostent *gethostbyname2(const char *name, int af); → returns ptr to hostent structure given hostname and protocol family (i.e. af = either AF_INET or AF_INET6)

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size); → convert **src** network byte order into string pointed by **dst**

NetworkProgramming\gethostbyname2_example.c

$ ./gethostbyname2_example.out www.google.com

Hostname: www.google.com

IP Address 1: 2404:6800:4009:80a::2004

Host and Service Conversion: getaddrinfo()
Works for both IPv4 and IPv6 simultaneously

- 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.

int getaddrinfo(const char *host, /* Hostname or address */

const char *service, /* Port or service name */

const struct addrinfo *hints,/* Input parameters (Filtering) */

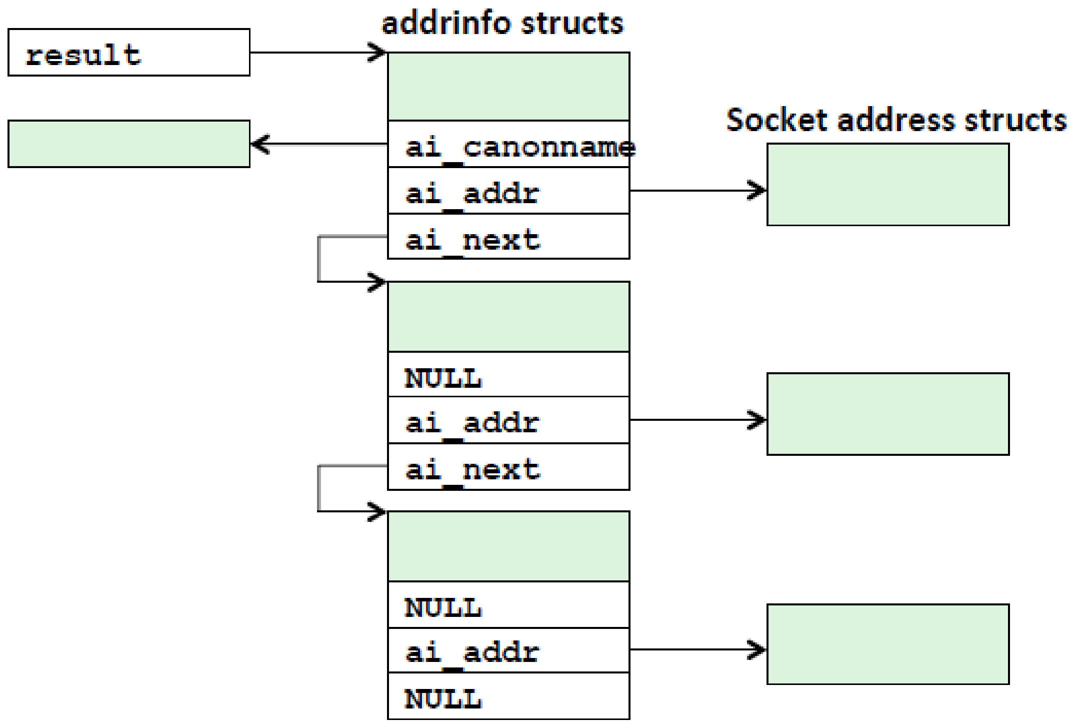
struct addrinfo **result); /* Output linked list */

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

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

Linked List returned by 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
- Advantages:
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code (works with IPv4 and IPv6 addresses)
- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.



addrinfo structure

struct addrinfo {

int ai_flags; /* Hints argument flags (AI_PASSIVE – used in server for passive TCP connection, AI_ADDRCONFIG – used so that IPv4 or IPv6 any type of addresses can be used, AI_NUMERICSERV – used when providing numeric value of port number*/

int ai_family; /* First arg to socket function (AF_INET or AF_INET6 or AF_UNSPEC) */

int ai_socktype; /* Second arg to socket function (SOCK_STREAM or SOCK_DGRAM or 0 means ANY)*/

int ai_protocol; /* Third arg to socket function (0 means ANY – generally only 1 protocol per family) */

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()

int getnameinfo(

const struct sockaddr *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 */

flags = NI_NUMERICHOST | NI_NUMERICSERV; /* Display address string instead of domain name and port number instead of service name */

- getnameinfo displays a socket address to the corresponding host (name or IP) and service (service or port).
  - Replaces obsolete gethostbyaddr and getservbyport funcs.
  - Reentrant and protocol independent.

getaddrinfo() example

NetworkProgramming\hostinfo.c

$ ./hostinfo.out www.daiict.ac.in

220.226.182.128 → Returned IPv4 IP address

$ ./hostinfo.out localhost

127.0.0.1

$ ./hostinfo.out www.twitter.com

104.244.42.65

104.244.42.1

$ ./hostinfo.out www.google.com

172.217.163.100

2404:6800:4007:809::2004

$ ./hostinfo.out www.facebook.com

157.240.13.35 → Returned IPv4 IP address

2a03:2880:f10c:83:face:b00c:0:25de → Returned IPv6 IP address

If we disable line

#define IPv4 1, it will provide IPv4 as well as IPv6 addresses

$ ./hostinfo.out www.google.com

172.217.163.100

2404:6800:4007:811::2004

If we enable line

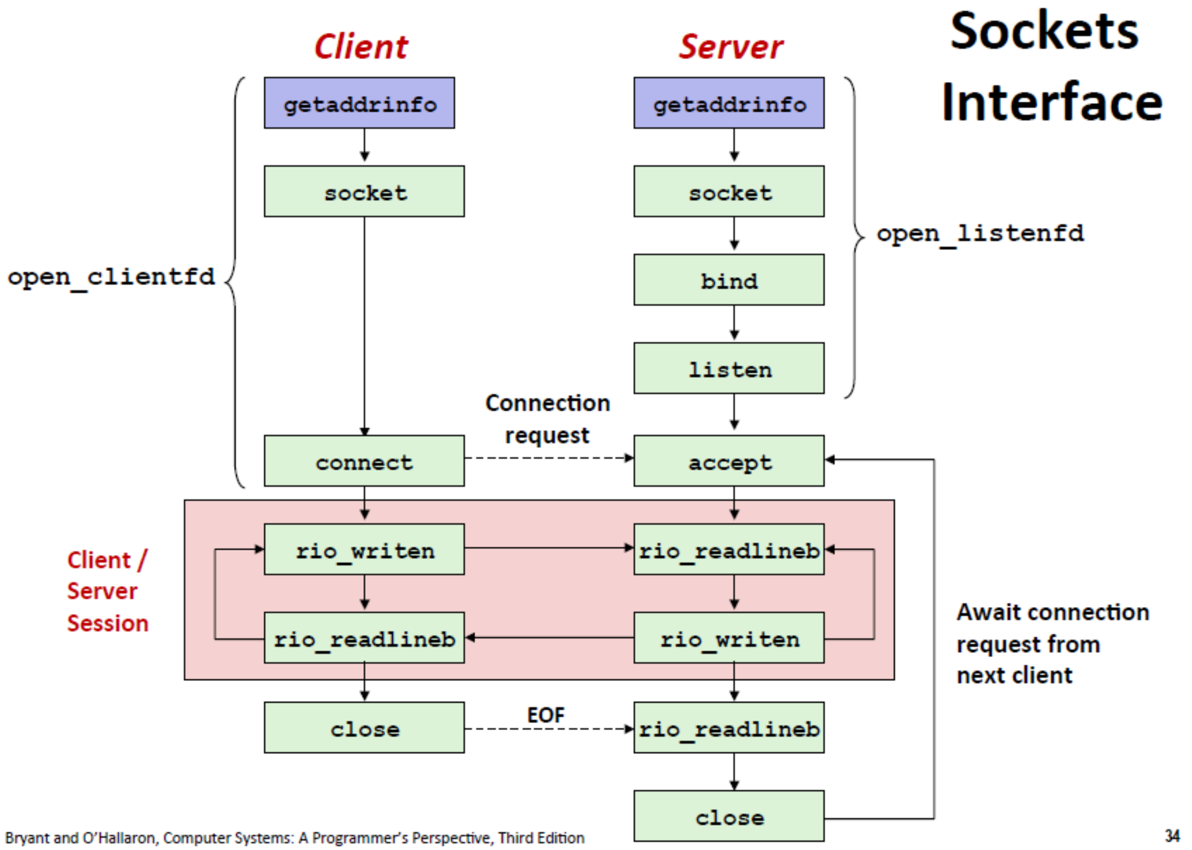
#define IPv4 1, it will provide IPv4 addresses only

$ ./hostinfo.out www.google.com

172.217.163.100

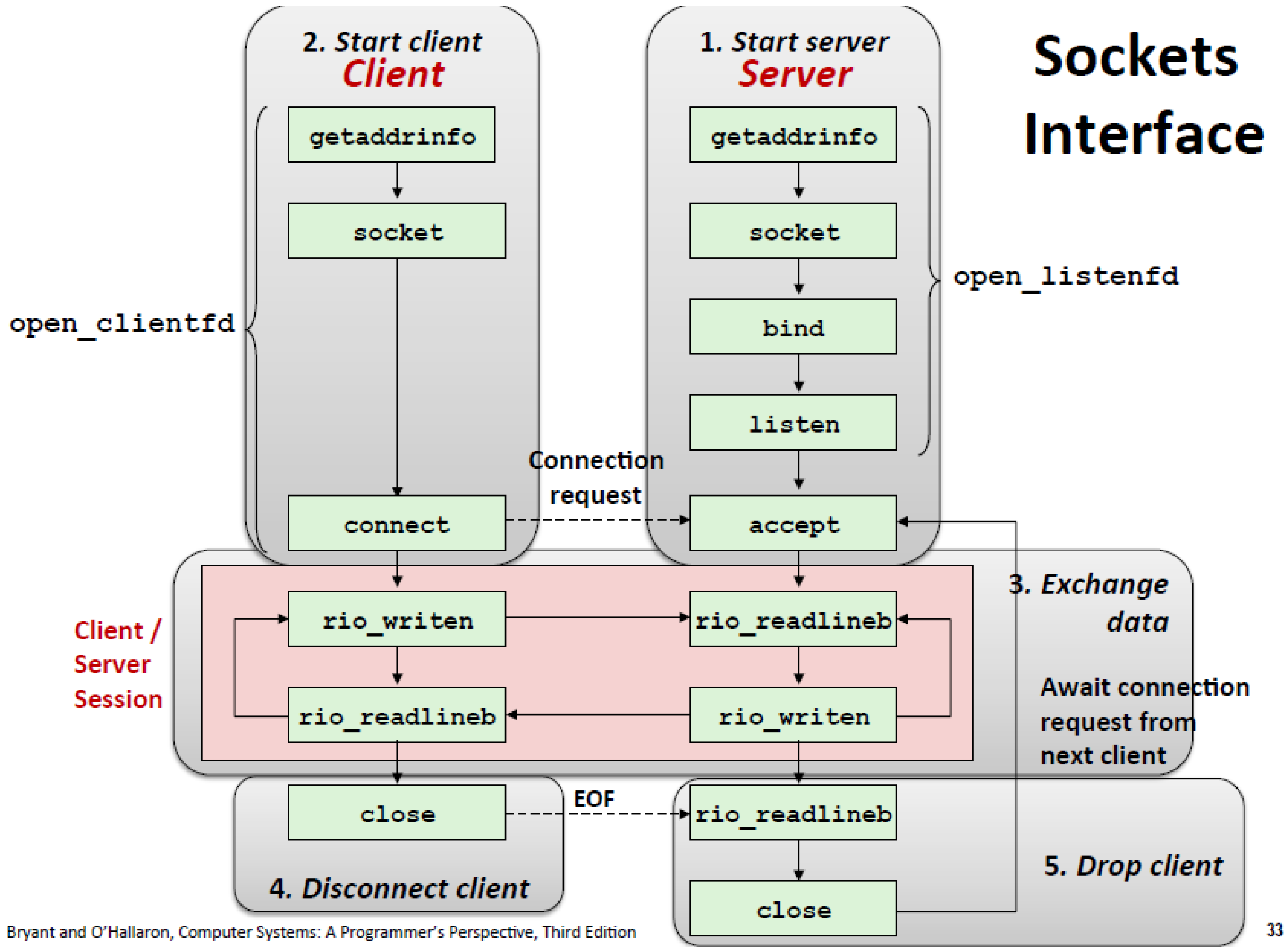
Client Server Communication using Socket Interface

Host and Service Conversion: getaddrinfo()



- Clients: Using server IP/hostname and service/port calls getaddrinfo. It walks through the returned list of server socket addresses, trying each socket address in turn, until the calls to socket and connect succeed.
- Server: Using service/port calls getaddrinfo. It walks through the returned list of socket addresses (possible to have different IP) until calls to socket and bind succeed.

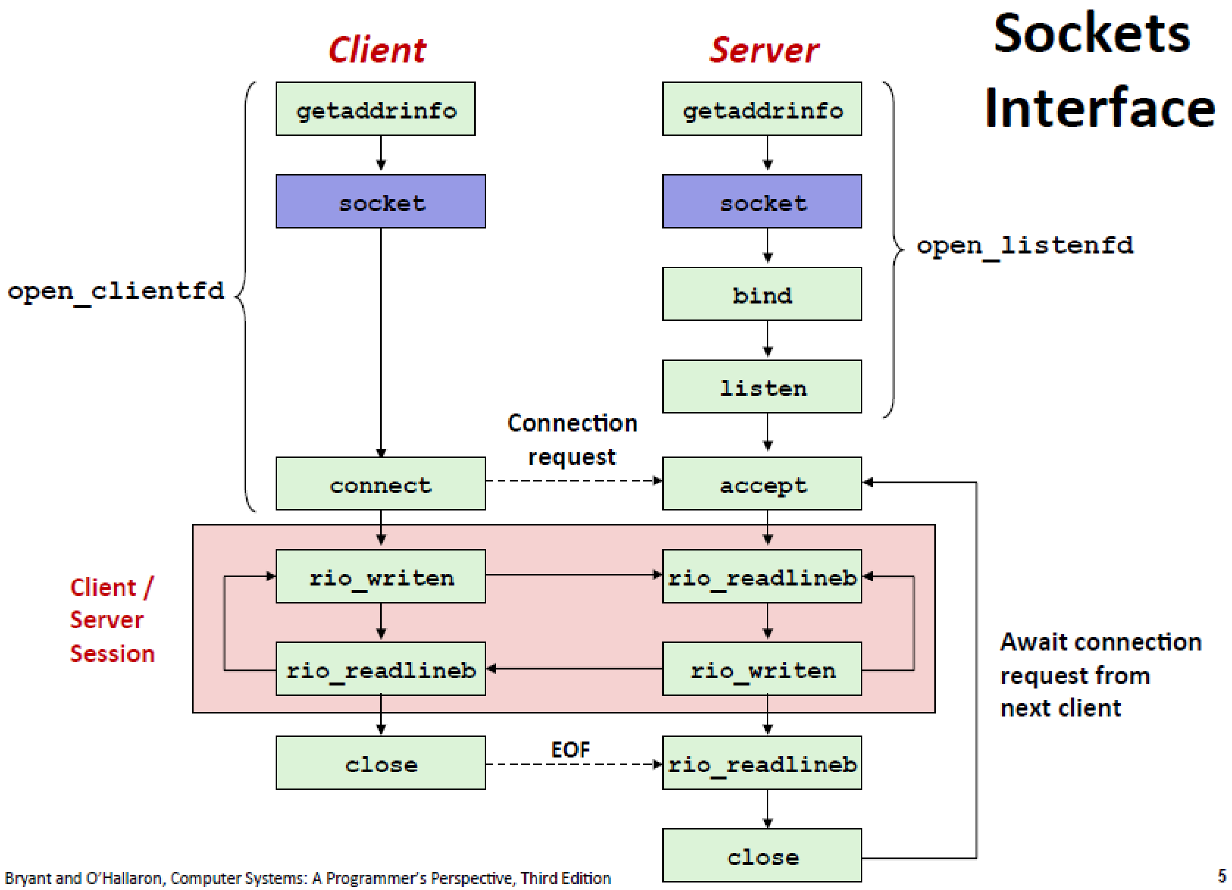
Socket Interface



open_clientfd → user function to start client
open_listenfd → user function to start server

rio_written = write(fd, buf, numofbytes)
rio_readlineb = read(fd, buf, numofbytes)

Socket Interface: socket()



Socket Interface: socket()

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

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

domain : indicates protocol family (same as ai_family from struct addrinfo)

type : socket type to use with protocol family (same as ai_socktype from struct addrinfo)

Protocol : protocol to be used from protocol family with a specific socket type. Generally there is only one protocol so we pass 0 (same as ai_protocol from struct addrinfo)

- Example:

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

sockfd will be same as clientfd for Client and listenfd for Server

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

Socket Domains and Types

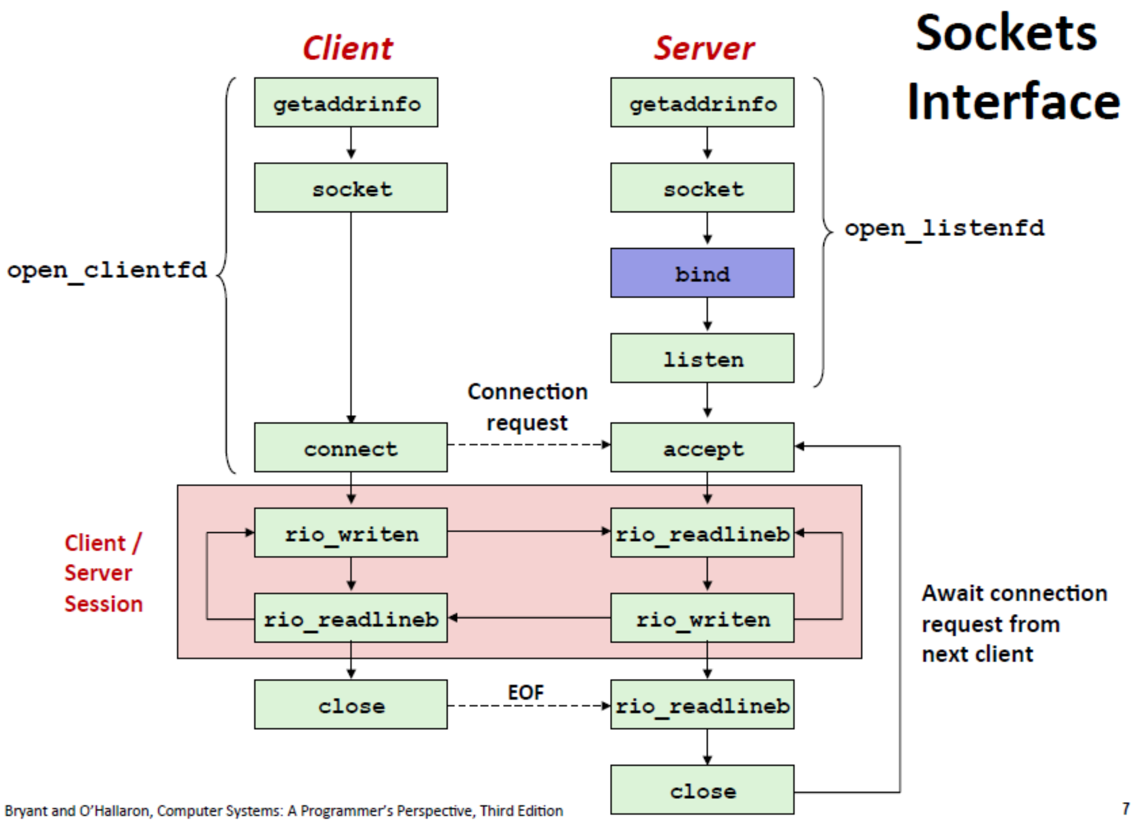
Domains

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols
AF_IPX	IPX - Novell protocols
AF_NETLINK	Kernel user interface device
AF_X25	ITU-T X.25 / ISO-8208 protocol
AF_AX25	Amateur radio AX.25 protocol
AF_APPLETALK	AppleTalk
AF_PACKET	Low level packet interface

Types

Name	Purpose
SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported
SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
SOCK_SEQPACKET	Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each input system call.
SOCK_RAW	Provides raw network protocol access
SOCK_RDM	Provides a reliable datagram layer that does not guarantee ordering
SOCK_PACKET	Obsolete and should not be used in new programs

Socket Interface: bind()



Socket Interface: bind()

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

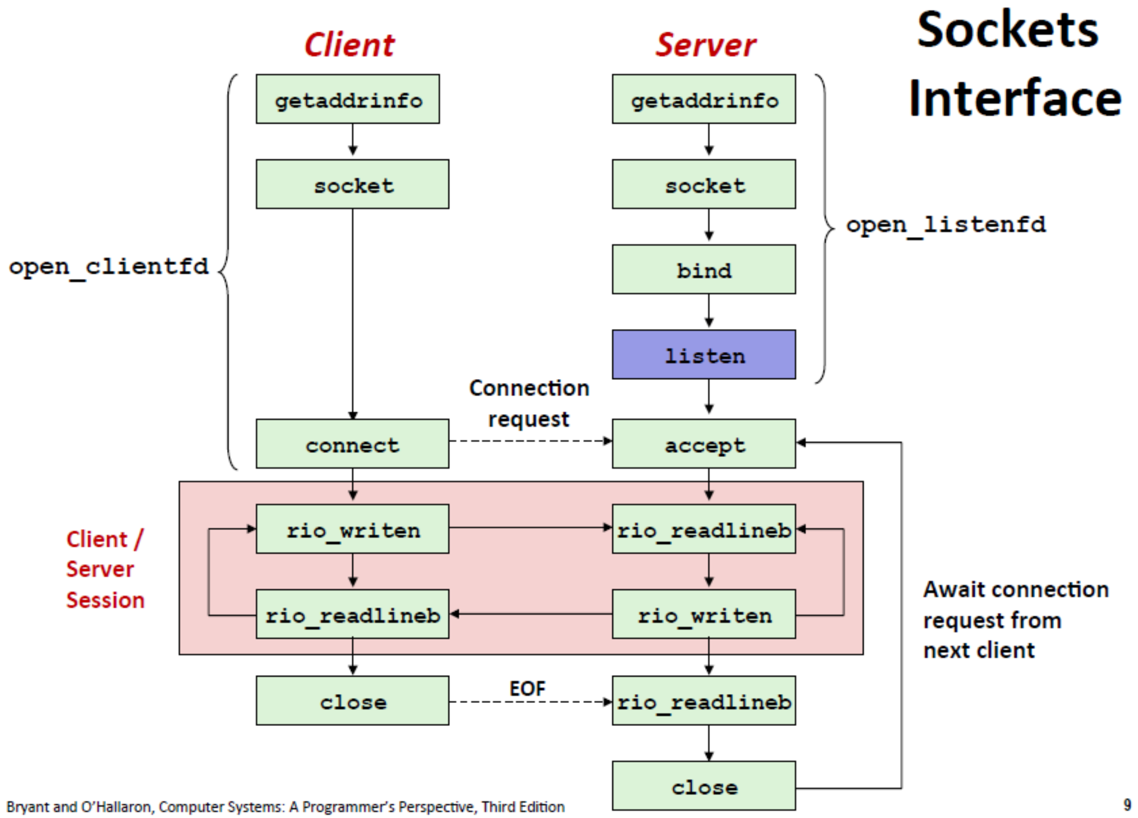
int bind(int listenfd, struct sockaddr *srv_addr, socklen_t srv_addrlen);

- The process can read bytes that arrive on the connection whose endpoint is srv_addr by reading from descriptor listenfd.
- Similarly, writes to listenfd are transferred along connection whose endpoint is srv_addr.
- Best practice is to use getaddrinfo to supply the arguments srv_addr and addrlen

struct sockaddr: Casted Address for any of the following types

Name	Purpose	sockaddr variants
AF_UNIX, AF_LOCAL	Local communication	sockaddr_un
AF_INET	IPv4 Internet protocols	sockaddr_in
AF_INET6	IPv6 Internet protocols	sockaddr_in6
AF_IPX	IPX - Novell protocols	sockaddr_ipx
AF_NETLINK	Kernel user interface device	sockaddr_nl
AF_X25	ITU-T X.25 / ISO-8208 protocol	sockaddr_x25
AF_AX25	Amateur radio AX.25 protocol	sockaddr_ax25
AF_APPLETALK	AppleTalk	sockaddr_atalk
AF_PACKET	Low level packet interface	sockaddr_ll

Socket Interface: listen()



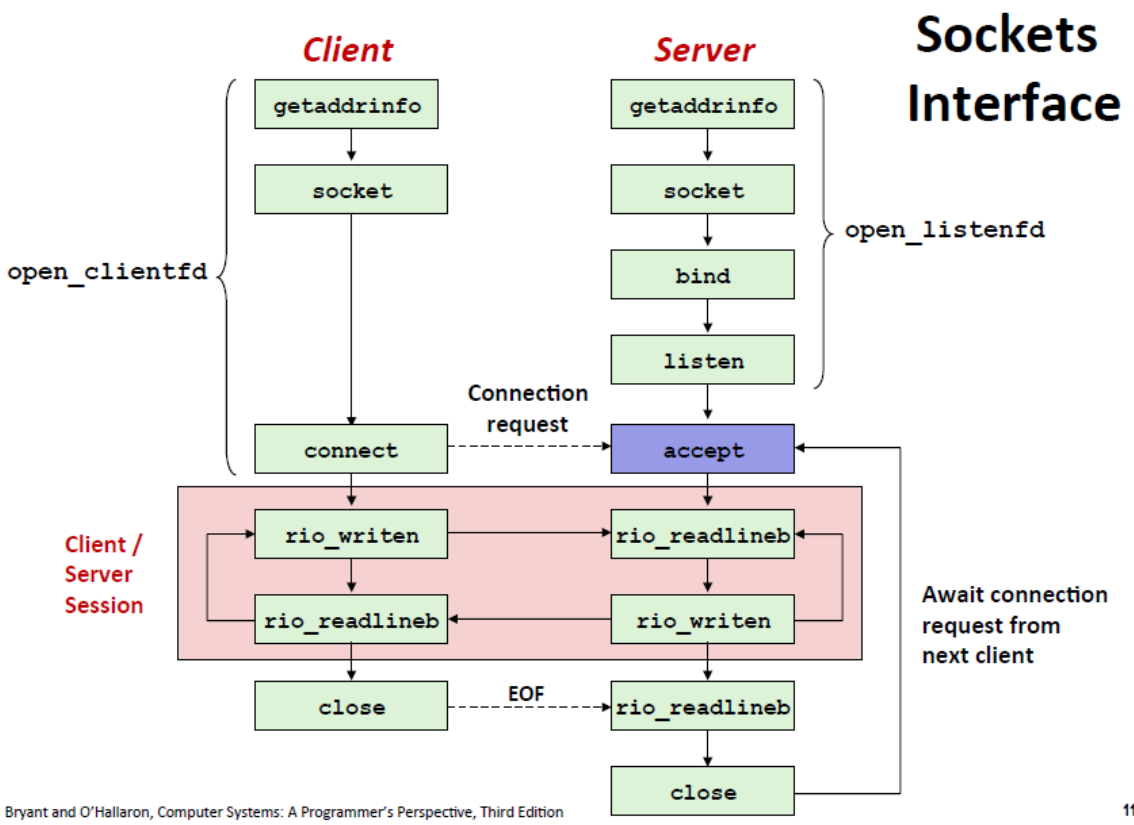
Socket Interface: listen()

- By default, kernel assumes that descriptor from socket function is an active socket that will be on the client end of the 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 listenfd, int backlog);

- Converts listenfd from an active socket to a listening socket that can accept connection requests from clients.
- backlog is the pending number of connection requests that the kernel should queue up before starting to refuse requests.

Socket Interface: accept()



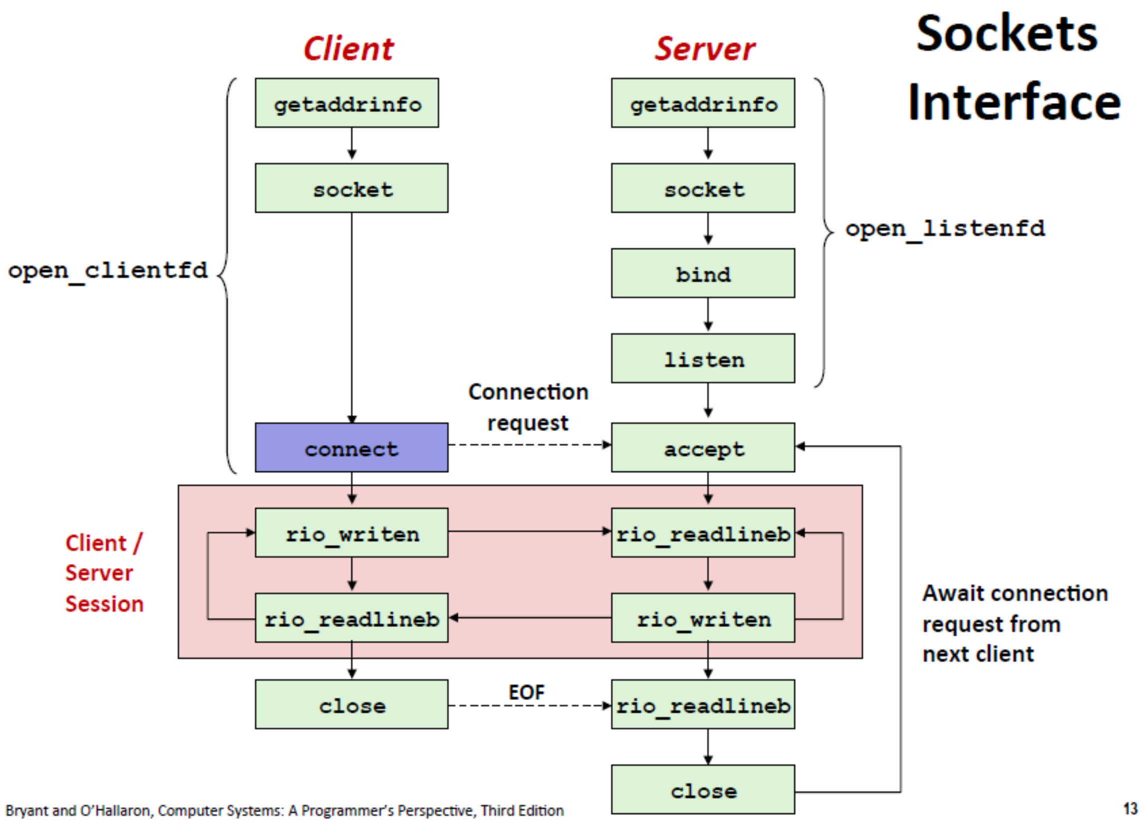
Socket Interface: accept()

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

int accept(int listenfd, struct sockaddr *clnt_addr, int *clnt_addrlen);

- Waits for connection request to arrive on the connection bound to listenfd, then fills in client's socket address in clnt_addr and size of the socket address in clnt_addrlen.
- Returns a new connection descriptor connfd that is the bound to clnt_addr which is used to communicate with the client via Unix I/O routines. So for every new accepted connection, a new connfd is created for an accepted client connection to be used for communication with that client only. (so that listenfd can be freed to be used for accepting new connections)

Socket Interface: connect()



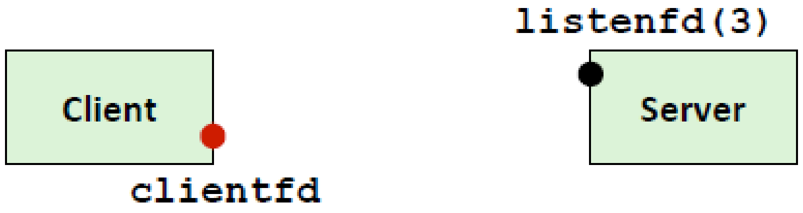
Socket Interface: connect()

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

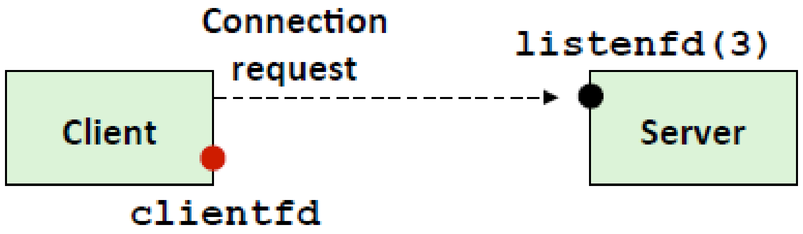
int connect(int clientfd, struct sockaddr *srv_addr, socklen_t srv_addrlen);

- Attempts to establish a connection with server at socket address srv_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 srv_addr and srv_addrlen.

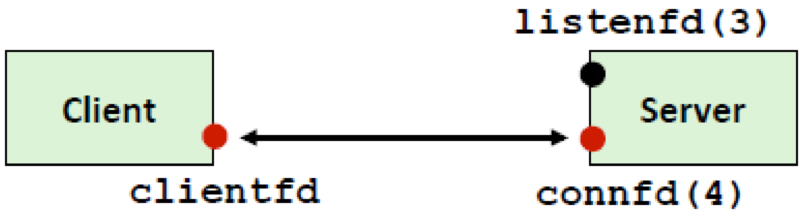
accept Illustrated



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



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

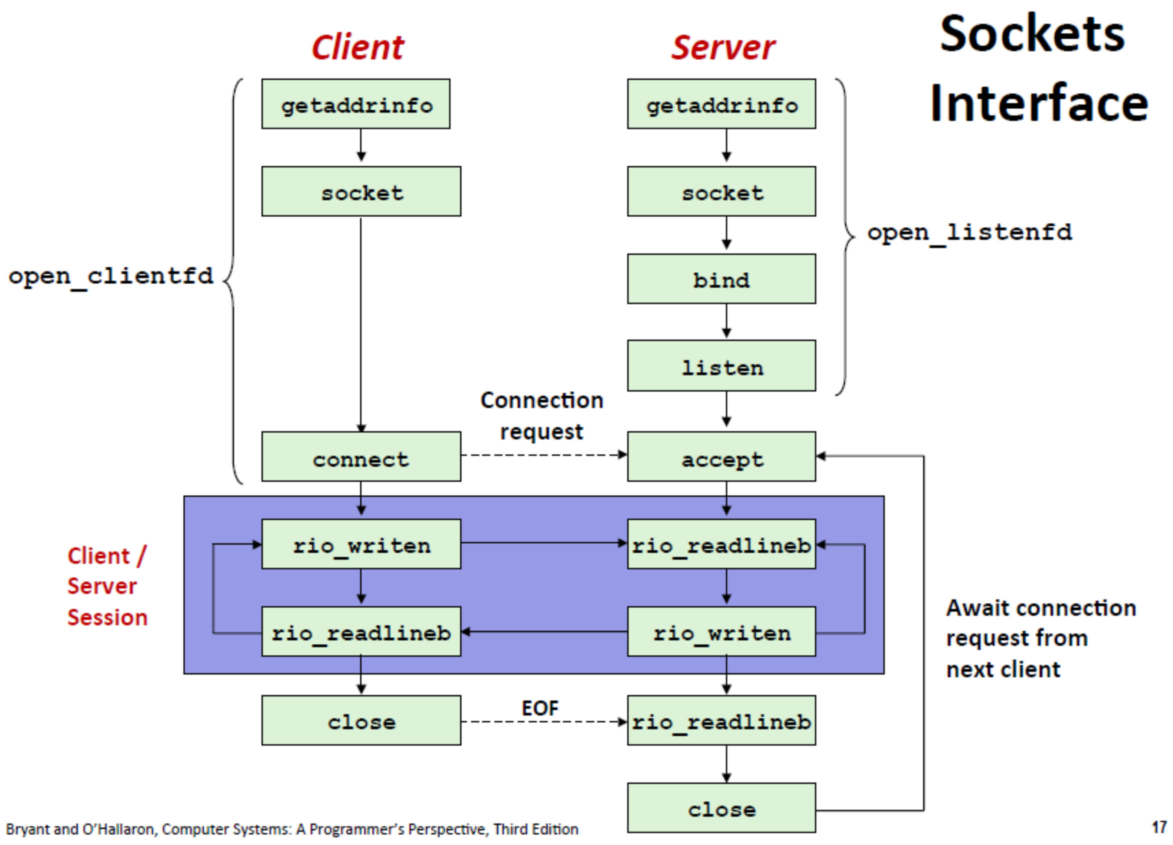


3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs Listening Descriptors

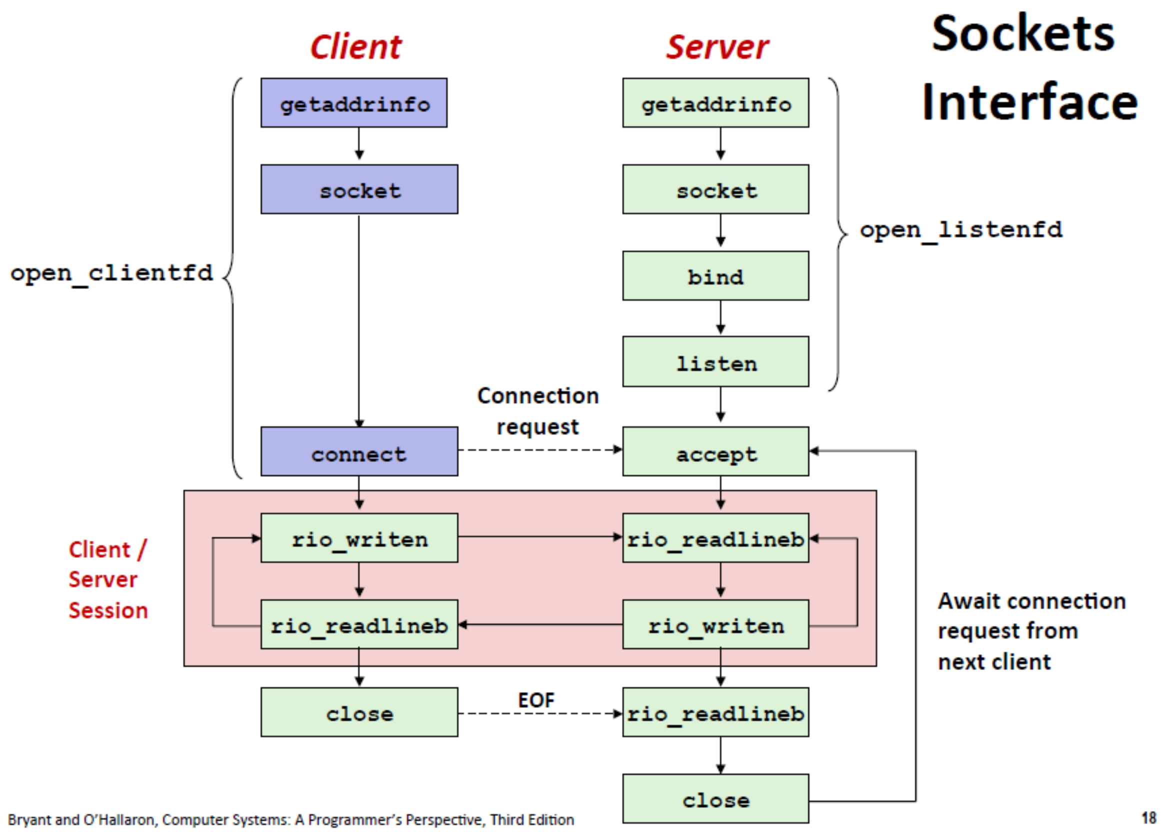
- Listening descriptor (listenfd)
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- Connected descriptor (connfd)
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request 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

Client-Server Session



- Server side **listenfd** is bound to server socket address for listening to clients connection requests.
- Client side **clientfd** is bound to server socket address to send and receive data to server.
- Server side **connfd** is bound to client socket address to send and receive data to client.

Socket Helper Function: open_clientfd



Socket Helper Function: open_clientfd

```
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 a numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections where we get IPv4 or IPv6 addresses */
    getaddrinfo(hostname, port, &hints, &listp);
```

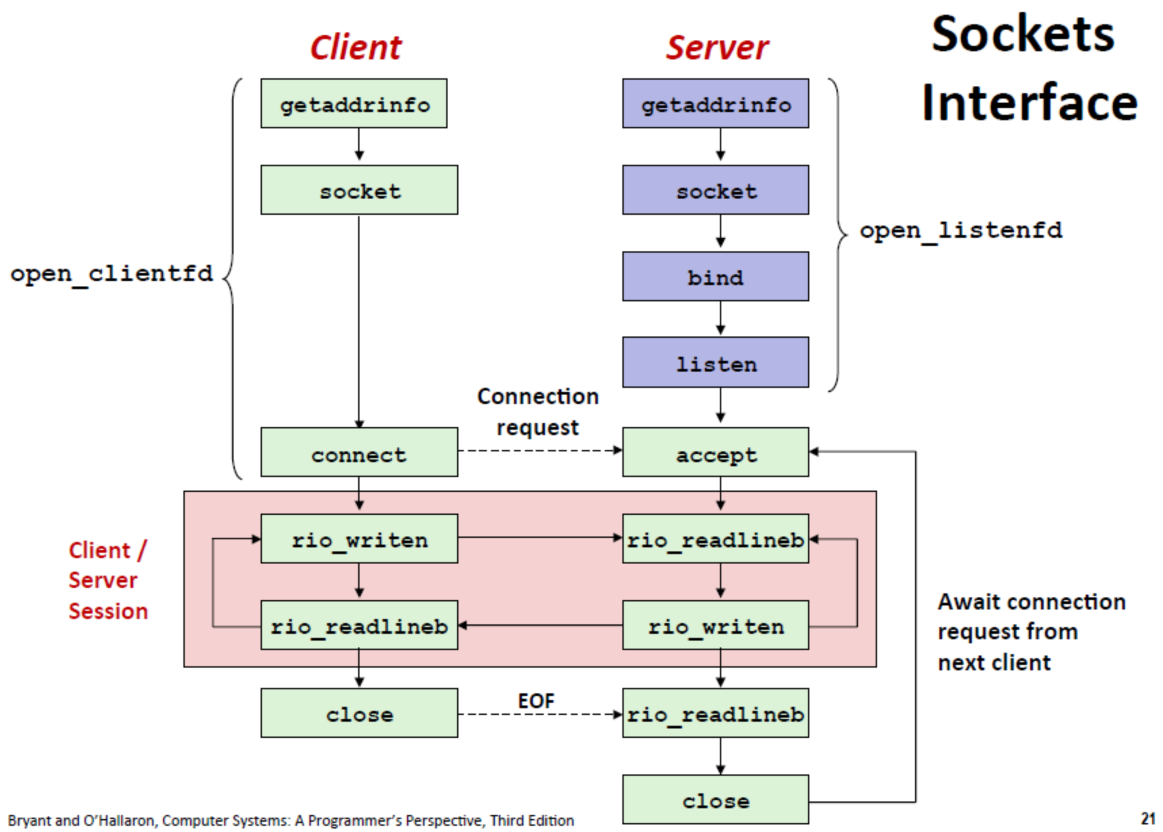
Socket Helper Function: open_clientfd

```
/* 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 */
  }  /* end for */
```

Socket Helper Function: open_clientfd

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

Socket Helper Function: open_listenfd



Socket Helper Function: open_listenfd

```
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 connections */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... on any IP address
AI_PASSIVE - used for server for TCP passive connection, AI_ADDRCONFIG -
to use both IPv4 and IPv6 addresses */
    hints.ai_flags |= AI_NUMERICSERV;            /* ... using port number */
    getaddrinfo(NULL, port, &hints, &listp);
```

Socket Helper Function: open_listenfd

```
/* 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 in case if process was killed during previous
execution and port was not freed. SOL_SOCKET =Socket API, set SO_REUSEADDR =optval(1)*/
    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 */
} /* end for */
```

Socket Helper Function: open_listenfd

```
 /* Clean up */
   freeaddrinfo(listp);
   if (!p) /* No address worked */
       return -1;
   /* Make it a listening socket ready to accept connection requests */
   if (listen(listenfd, LISTENQ) < 0) {
       close(listenfd);
           return -1;
   } /* end if */
   return listenfd;
} /* end open_listenfd */
```

Test Your Own Echo Client Using Default Echo Server on Port 7 (without using your own server)

- First, make sure echo service is running on port 7:

$ cat /etc/services | grep 7/tcp

echo 7/tcp

If not you will have to install inetd service to have echo service on port 7

Our Echo Client Implementation: [NetworkProgramming\echoclient.c](NetworkProgramming\echoclient.c)

$ ./echoclient.out localhost 7

host:127.0.0.1, service:7

message from our client to echo server on port 7

message from our client to echo server on port 7 → echoed

another test

another test → echoed

^C

Test Your Own Echo Server Using Telnet (without using your own client)

```
$ telnet 10.0.0.6 15020
Trying 10.0.0.6...
Connected to 10.0.0.6.
Escape character is '^]'.
this is a test from telnet client
this is a test from telnet client
howdy from telnet client
howdy from telnet client
^]
telnet> Connection closed.
$
```

Our Echo Server Implementation:

NetworkProgramming\echoserver.c

```
$ ./server.out 15020
Waiting for a new Client to connect
Connected to (10.0.0.6, 56166)
Start Communication with Client
server received 35 bytes
server received message : this is a test from telnet client

server received 26 bytes
server received message : howdy from telnet client

End Communication with Client
Waiting for a new Client to connect
```

Test Your Own Echo Client and Echo Server

```
$ ./client.out faculty-Optiplex-3040 15010
this is a test from client 1
this is a test from client 1
howdy from client 1
howdy from client 1

$ ./client.out faculty-Optiplex-3040 15010
this is a test from client 2
this is a test from client 2
hello from client 2
hello from client 2

$
```

```
$ ./server.out 15010
Waiting for a new Client to connect
Connected to (localhost, 45752)
Start Communication with Client
server received 29 bytes
server received message : this is a test from client 1

server received 20 bytes
server received message : howdy from client 1

server received 1 bytes
server received message :

End Communication with Client
→ Continue
```

```
Continue →
Waiting for a new Client to connect
Connected to (localhost, 45754)
Start Communication with Client
server received 29 bytes
server received message : this is a test from client 2

server received 21 bytes
server received message : hellow from client 2

server received 1 bytes
server received message :

End Communication with Client
Waiting for a new Client to connect
```

Test 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:
  - $ telnet <host> <portnumber>
  - Creates a connection with a server running on <host> and listening on port <portnumber>

Echo Server Problem

- Echo Server is able to handle only 1 client connection at a time because the main thread goes in loop unless client end the connection
- How to fix it?

Solution of Echo Server Problem

```
while (1) {
  printf("Waiting for a new Client to connect\n");
  clientlen = sizeof(struct sockaddr_storage); /* Important! */
  connfd = accept(listenfd, (struct sockaddr *)&clientaddr, &clientlen);
  if (fork() == 0)          /* child will handle a new client everytime server accepts a connection from client */
  {
     getnameinfo((struct sockaddr *) &clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
     printf("Connected to (%s, %s)\n", client_hostname, client_port);
     printf("Start Communication with Client\n");
     echo(connfd);
     printf("End Communication with Client\n");
     close(connfd);
  }
}
```

Solution of Echo Server Problem

- Would it be better to use multiple threads or multiple processes to handle clients?
- What are the challenges with each scheme?