Operating Systems and C
13. Network Programming

Course Evaluations

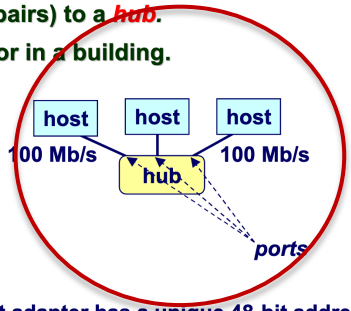
- Course evaluation is now open
- A note on the importance of answering the course evaluation:
  - I've been in the committee that receives the results.
  - Often, course results are discarded due to low response rate.
  - If you like this course, or have ideas on how to improve it, please answer the survey.

A note about this lecture

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.
- 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.

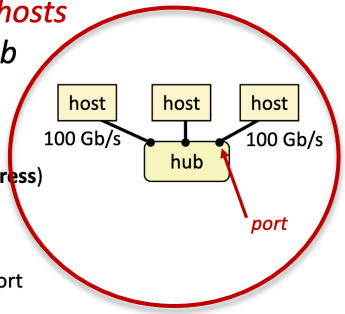
Note: Hubs are on their way out. Bridges (switches, routers) became cheap enough to replace them (means no more broadcasting)

2003

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

2020

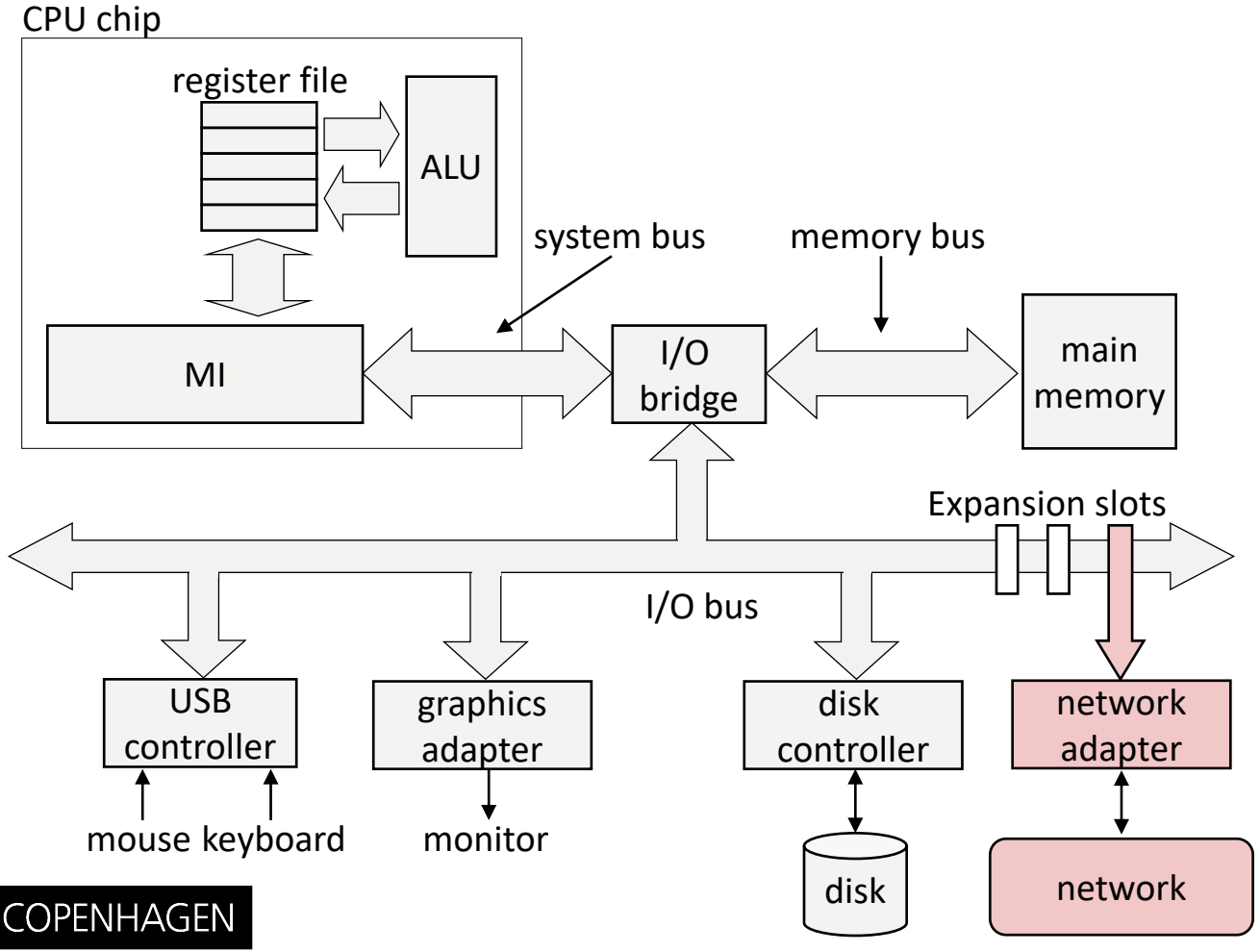
Outline

- Computer Networks
- Network Mapping (DNS, ports, IPs)
- Sockets

Communication Abstraction

Source: Saltzer and Kaashoek

Hardware Organization of a Network Host



Computer Networks

A *network* is a hierarchical system of boxes and wires organized by geographical proximity

SAN (System Area Network) spans cluster or machine room

LAN (Local Area Network) spans a building or campus

WAN (Wide Area Network) spans country or world

- Historically WAN is connected through high-speed point-to-point phone lines (2003)
  - Today more often high-speed point-to-point fiber-optic cables (2022)

The *Internet* is an interconnected set of networks

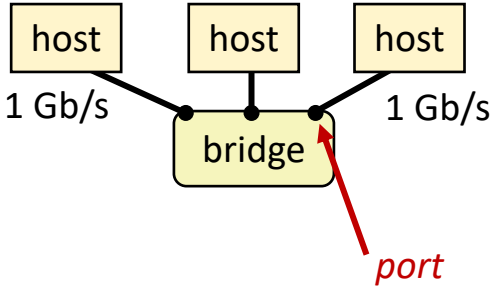
The Global IP Internet (uppercase "I") is the most famous example of an internet (lowercase "i")

Let's see how it is built from the ground up

Lowest Level: Ethernet Segment

Ethernet segment consists of a collection of *hosts* connected by wires (twisted pairs) or WiFi to a *hub*

Spans room or floor in a building



Operation

Each Ethernet adapter has a unique 48-bit address (**MAC address**)

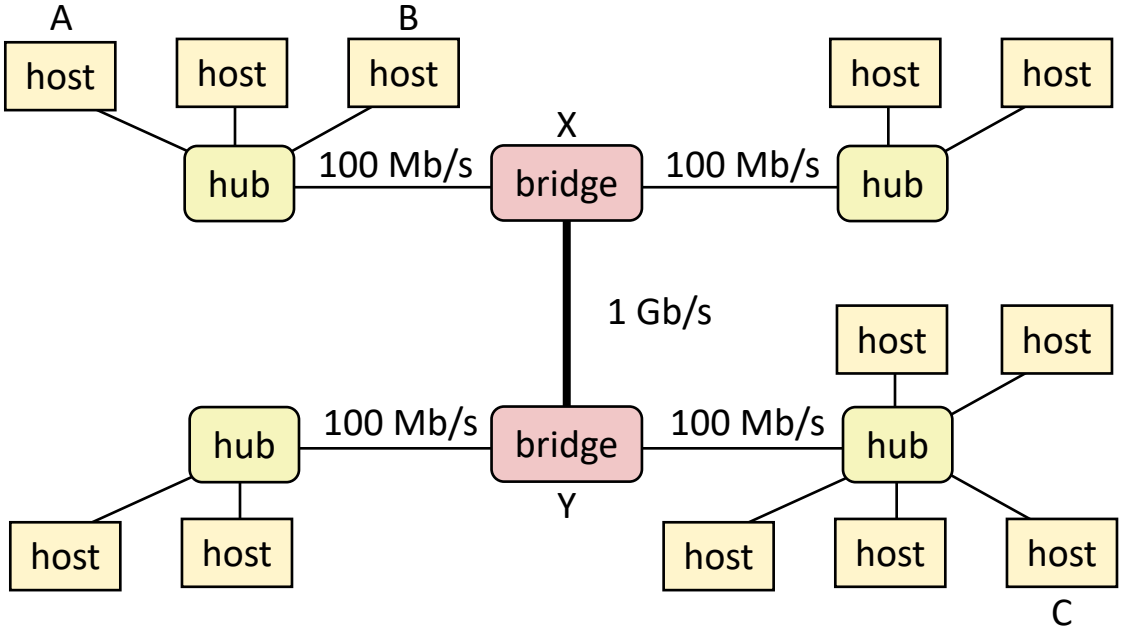
- E.g., 00:16:ea:e3:54:e6

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 (2003)
- Note: Hubs are out. Bridges has taken over (2022)
- Bridges are intelligent enough to send frames only to the intended recipient.

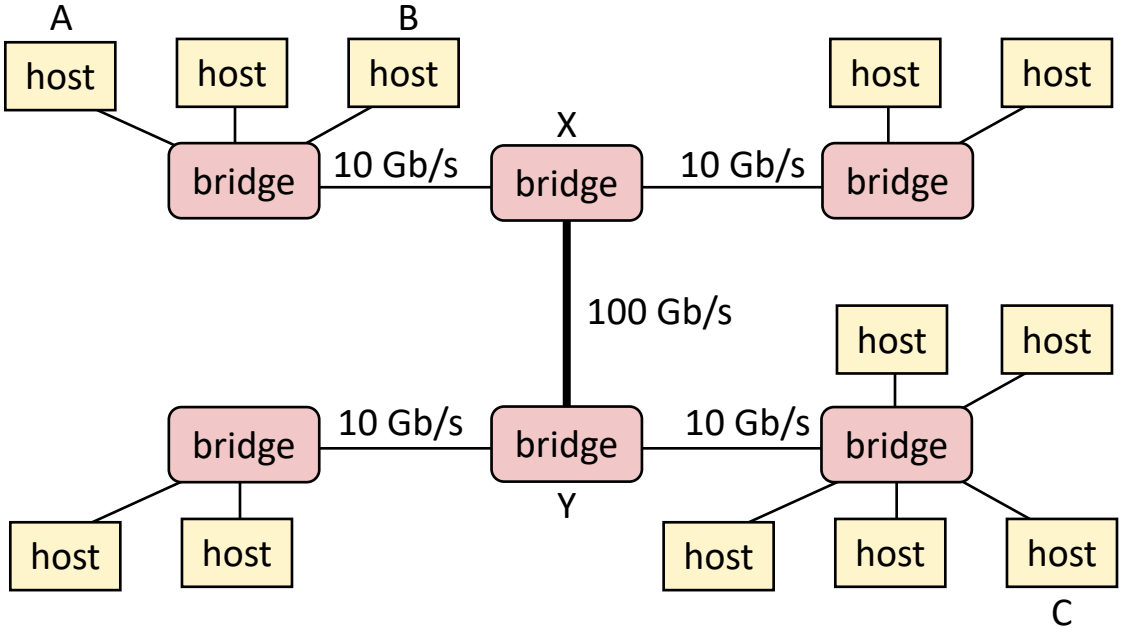
Next Level: Bridged Ethernet Segment (2003)



Spans building or campus

Bridges cleverly learn which hosts are reachable from which ports and then selectively copy frames from port to port

Next Level: Bridged Ethernet Segment (2022)

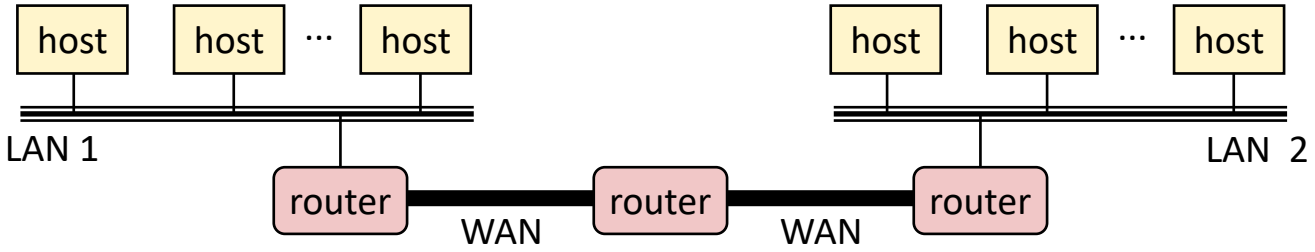


Spans building or campus

Bridges cleverly learn which hosts are reachable from which ports and then selectively copy frames from port to port

Next Level: internets

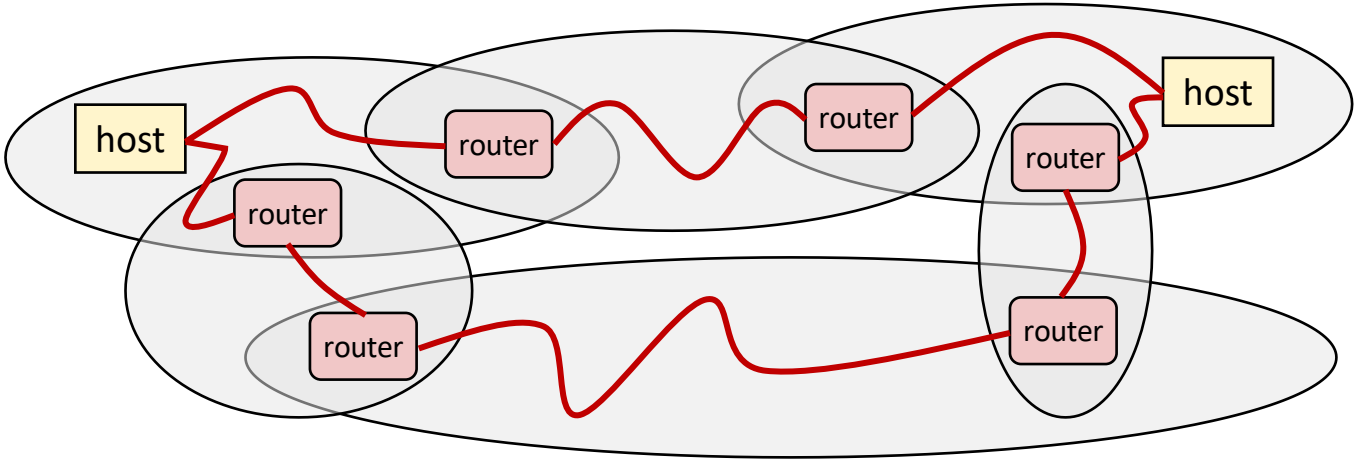
Multiple incompatible LANs can be physically connected by specialized computers called *routers*
The connected networks are called an *internet* (lower case)



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 router & link capacities

Send packets from source to destination by hopping through networks

- Router forms bridge from one network to another
- Different packets may take different routes

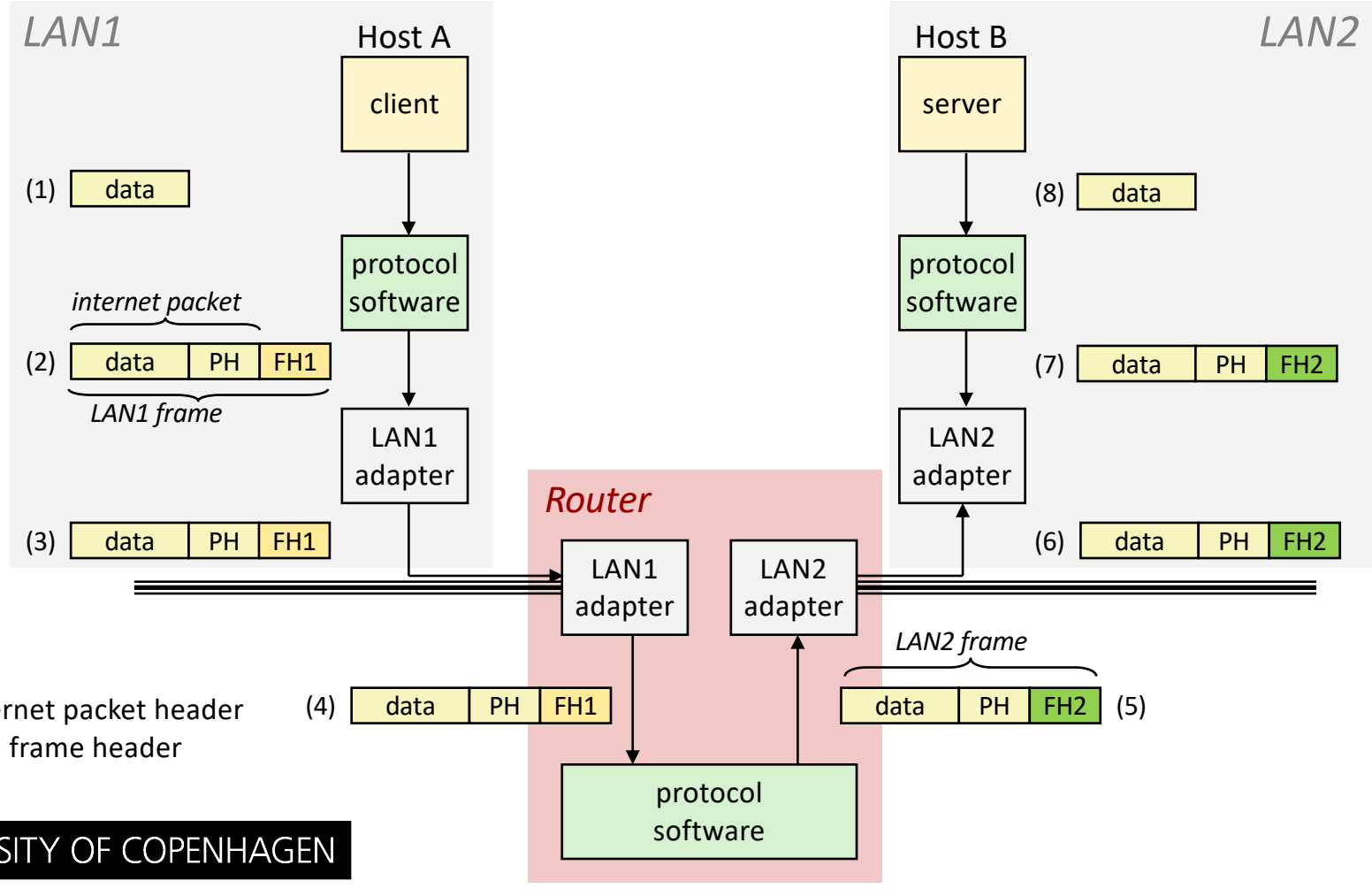
The Notion of an internet Protocol

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

Transferring internet Data Via Encapsulation



PH: Internet packet header
FH: LAN frame header

What does an 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

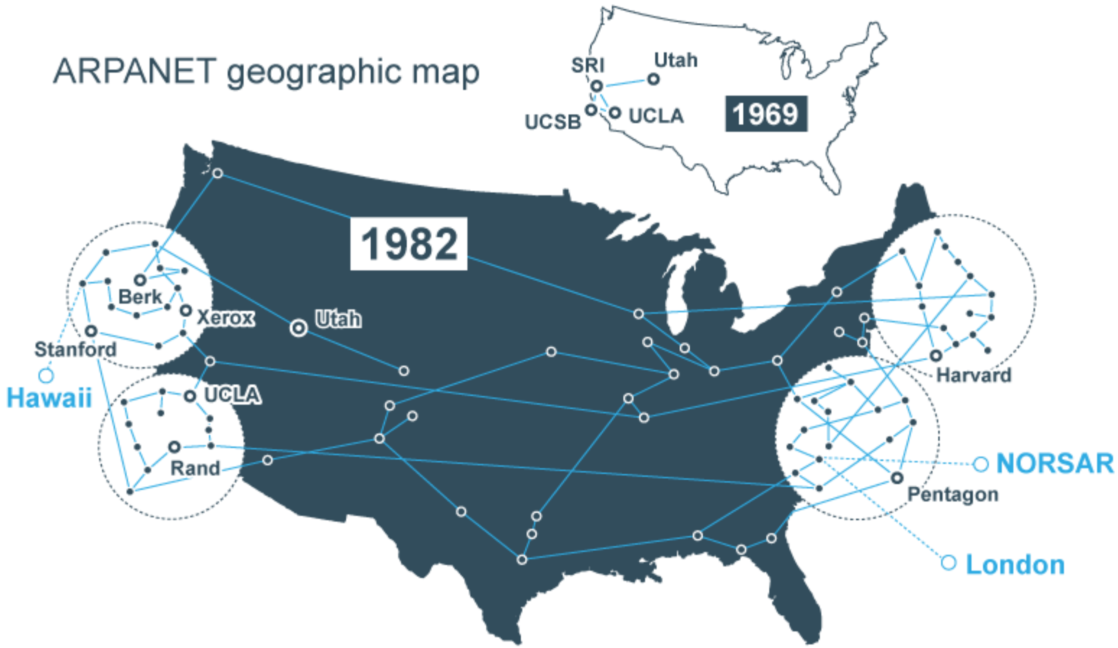
Other Issues

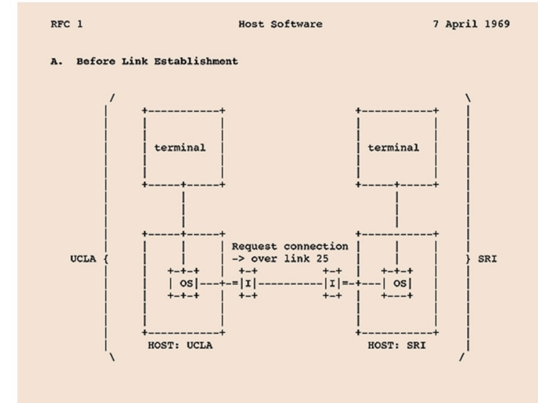
We are glossing over a number of important questions:

- What if different networks have different maximum frame sizes? (segmentation)
- How do routers know where to forward frames?
- How are routers informed when the network topology changes?
- What if packets get lost?

These (and other) questions are addressed by the area of systems known as *computer networking*

ARPANET

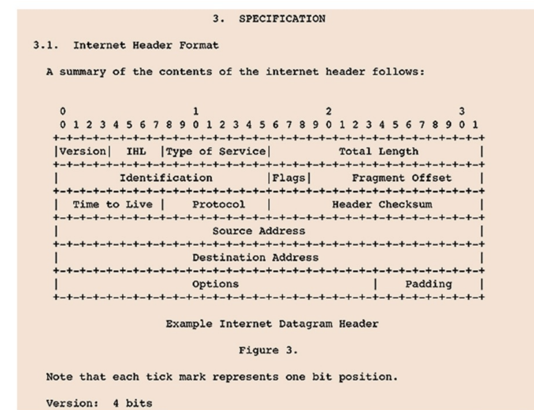




```
RFC 1                      Host Software                  7 April 1969

A.  Before Link Establishment

                         terminal                     terminal

                       Request connection
UCLA                   -> over link 25                              SRI
              OS                                       OS

           HOST: UCLA                              HOST: SRI
```

Images: Internet Engineering Task Force

Issued in April 1969 and written by Steve Crocker, then a grad student at UCLA, RFC 1 and its companion, RFC 2, by Bill Duvall of SRI, describe various aspects of the software used to connect the ARPANET's host computers and IMPs. The fluidity of the RFC process is apparent at the start. "Very little of what is here is firm, and reactions are expected," Crocker wrote.



```
                    3.  SPECIFICATION

3.1.  Internet Header Format

  A summary of the contents of the internet header follows:

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |Version|  IHL  |Type of Service|          Total Length         |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |         Identification        |Flags|      Fragment Offset    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Time to Live |    Protocol   |         Header Checksum       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Source Address                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Destination Address                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                    Example Internet Datagram Header

                               Figure 3.

  Note that each tick mark represents one bit position.

  Version:  4 bits
```

Issued in January 1980, this RFC outlines one of the foundational elements of Internet architecture: Internet Protocol. To this day, most Internet data is routed from one network to another in IPv4 packets, as laid out in this RFC. Because IPv4 addresses are limited to 4 billion hosts, a new IPv6 protocol was introduced as an Internet standard in 1995.

https://spectrum.ieee.org/tech-history/cyberspace/todays-internet-still-relies-on-an-arpanetera-protocol-the-request-for-comments

Global IP Internet (upper case)

Most famous example of an internet

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 (Unreliable Datagram Protocol)

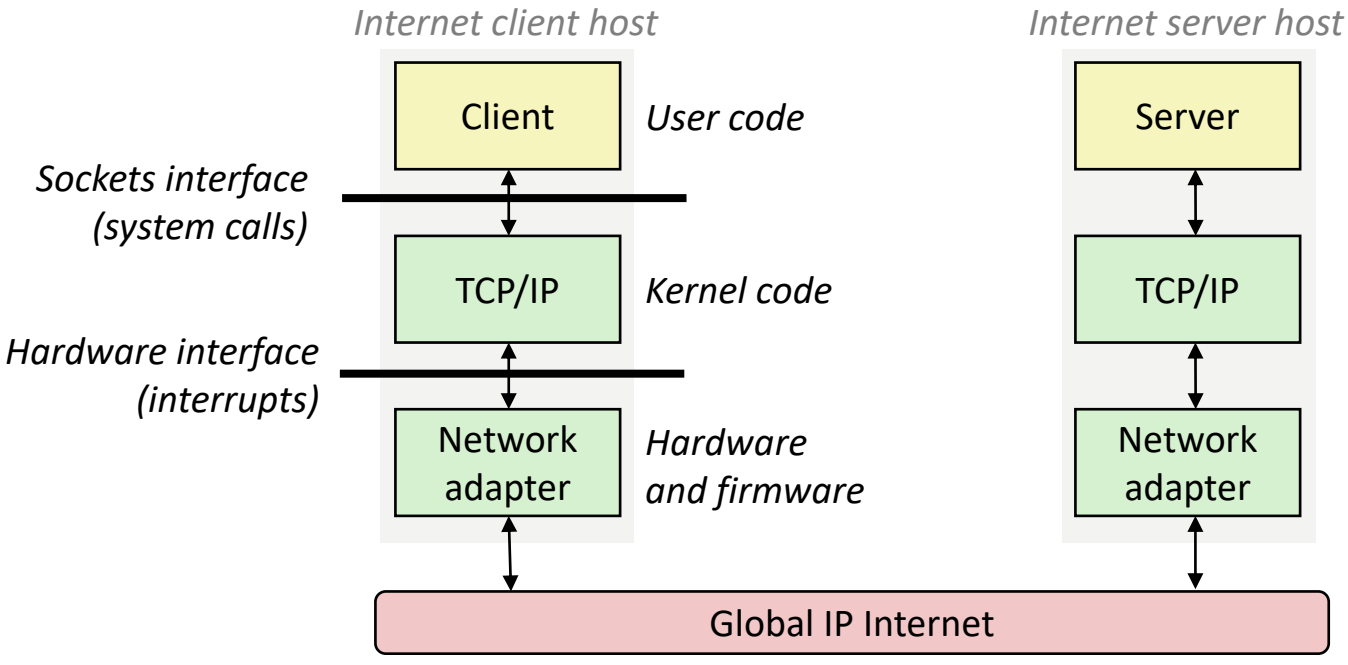
Uses IP to provide *unreliable* datagram delivery from *process-to-process*

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*

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*

   130.226.140.95 (2003)

   - We are trying to fully replace 32-bit addresses with 64-bits addresses. It has not happened yet (2022)

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*

   130.226.140.95 is mapped to cos.itu.dk

3. A process on one Internet host can communicate with a process on another Internet host over a *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) */
};
```

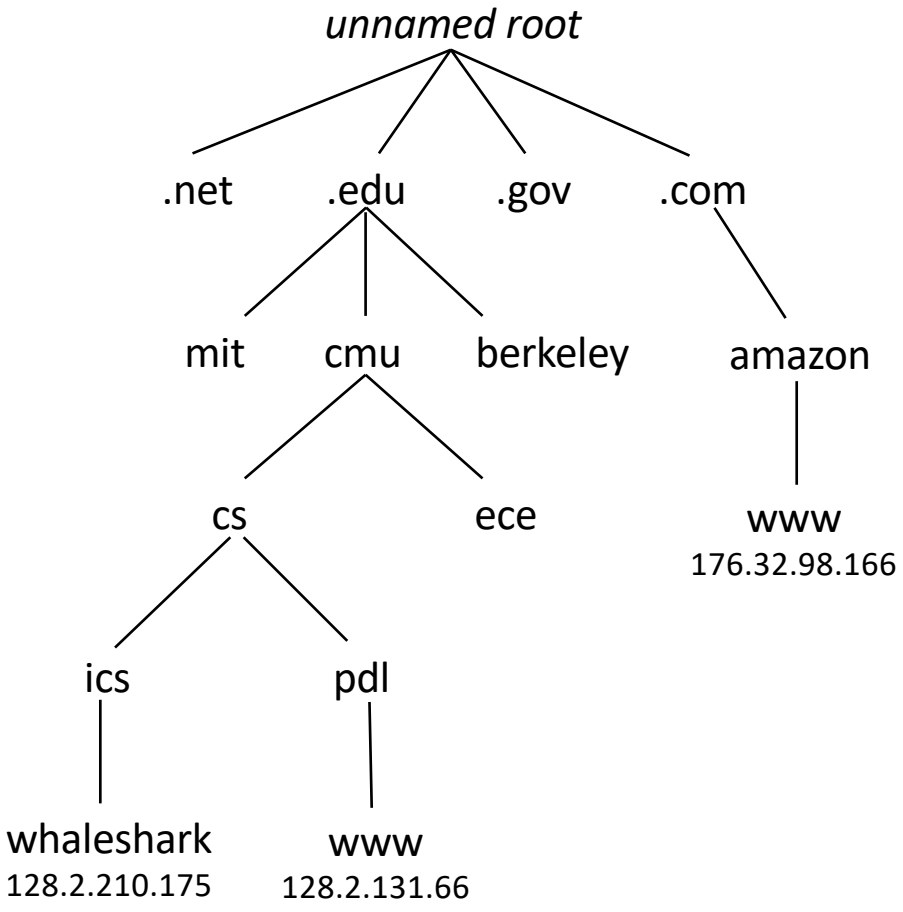
Dotted Decimal Notation

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`

Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.

(2) Internet Domain Names



First-level domain names

Second-level domain names

Third-level domain names

Domain Naming System (DNS)

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.

(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

Popular services have permanently assigned *well-known ports* *and* corresponding *well-known service names*:

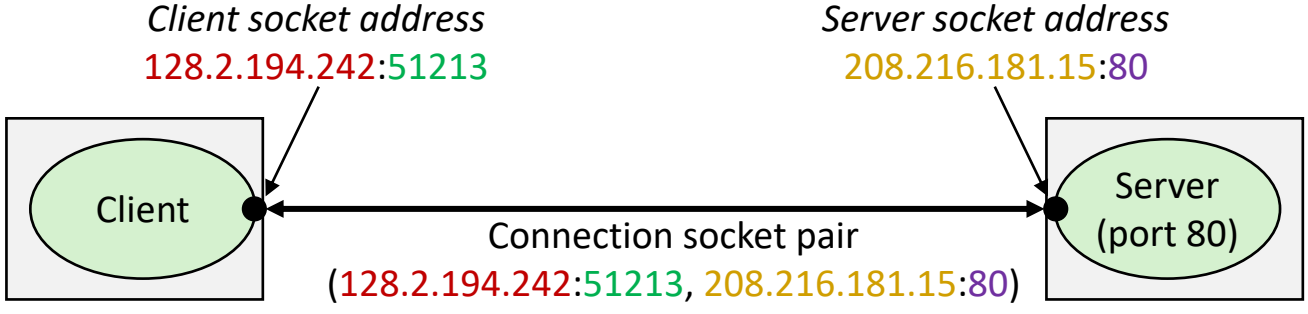
- echo server: 7/echo
- ssh servers: 22/ssh
- email server: 25/smtp
- Web servers: 80/http

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

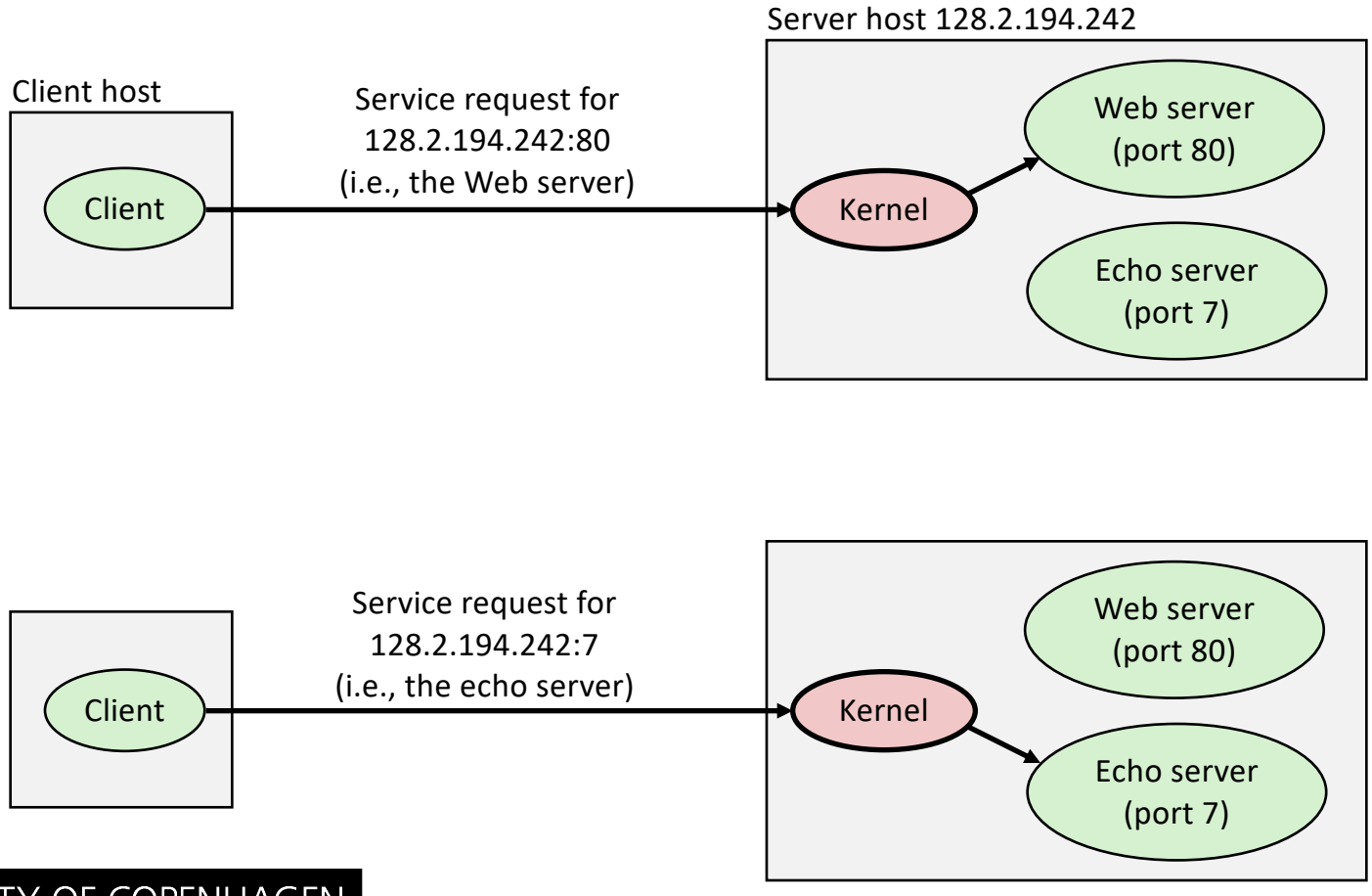
Anatomy of a Connection

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)

```
(cliaddr:cliport, servaddr:servport)
```



Using Ports to Identify Services



Break!

ANALOG

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

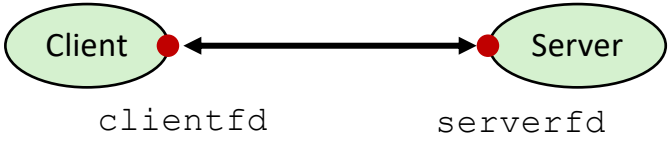
What is a socket?

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:

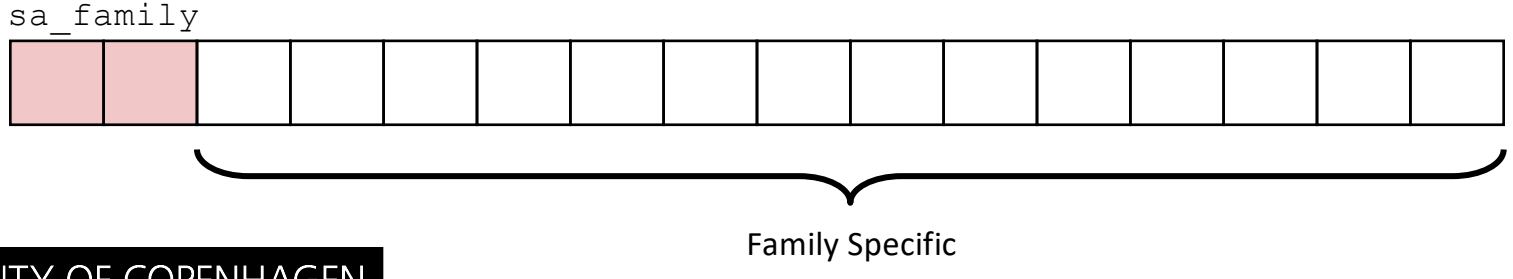
For address arguments to **`connect`**, **`bind`**, and **`accept`**

Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed

For casting convenience, we adopt the Stevens convention:

`typedef struct sockaddr SA;`

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

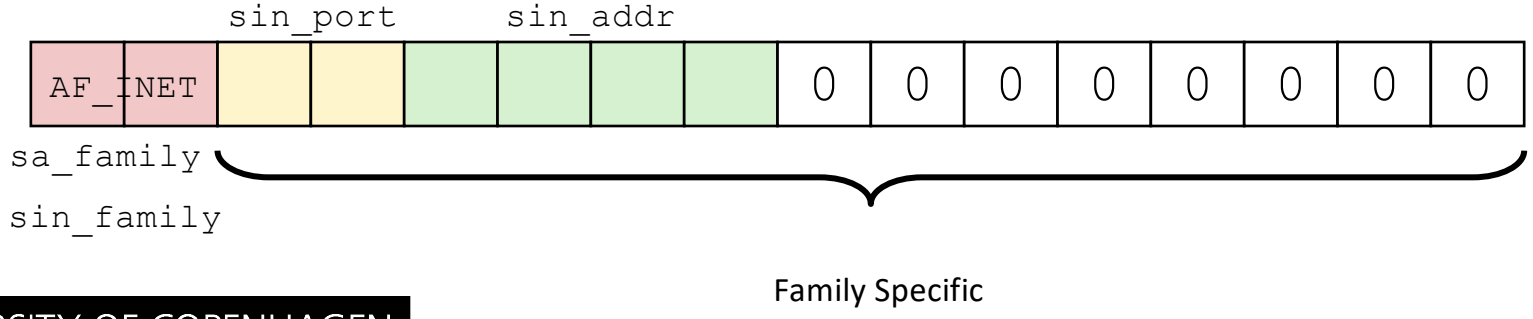


Socket Address Structures

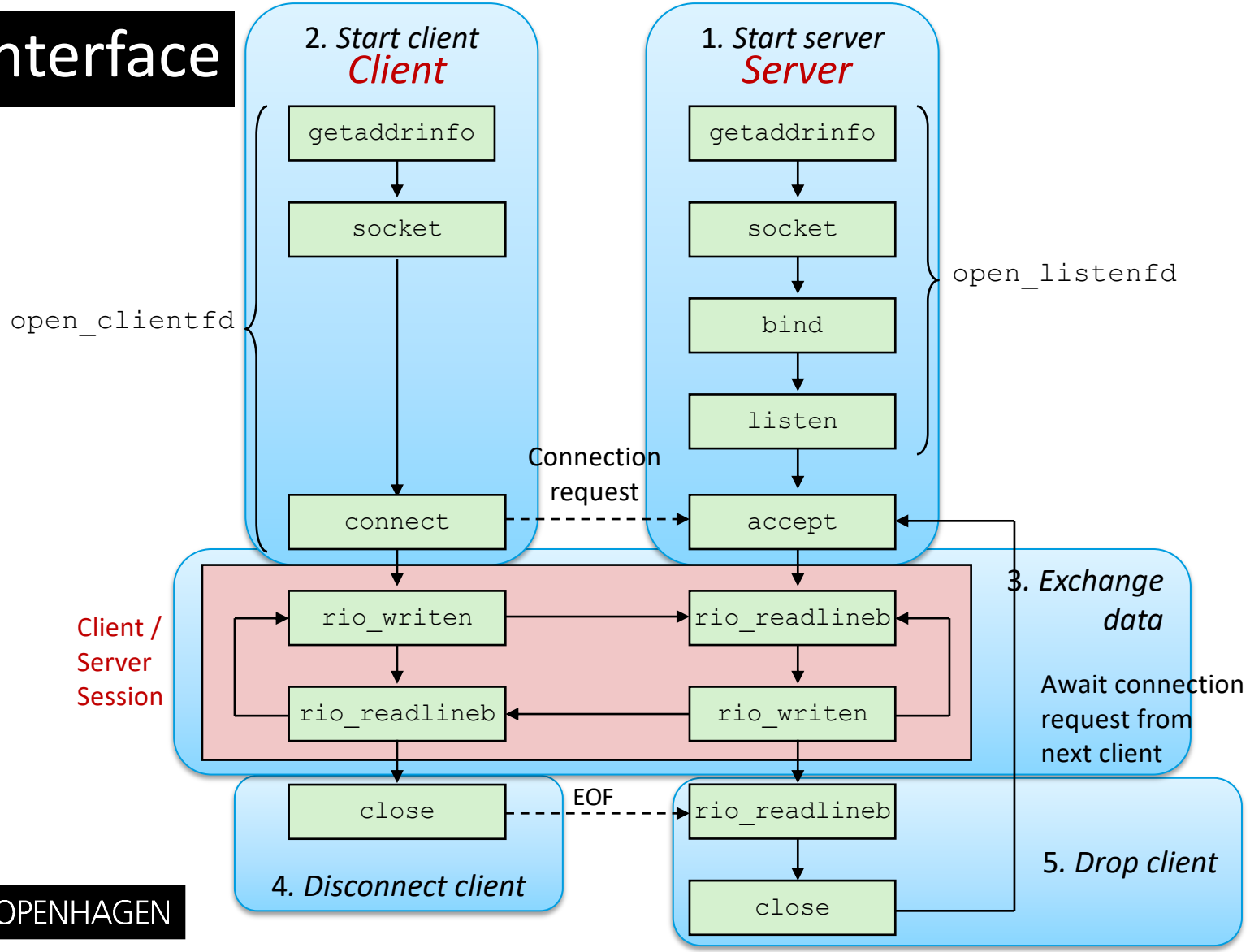
Internet-specific socket address:

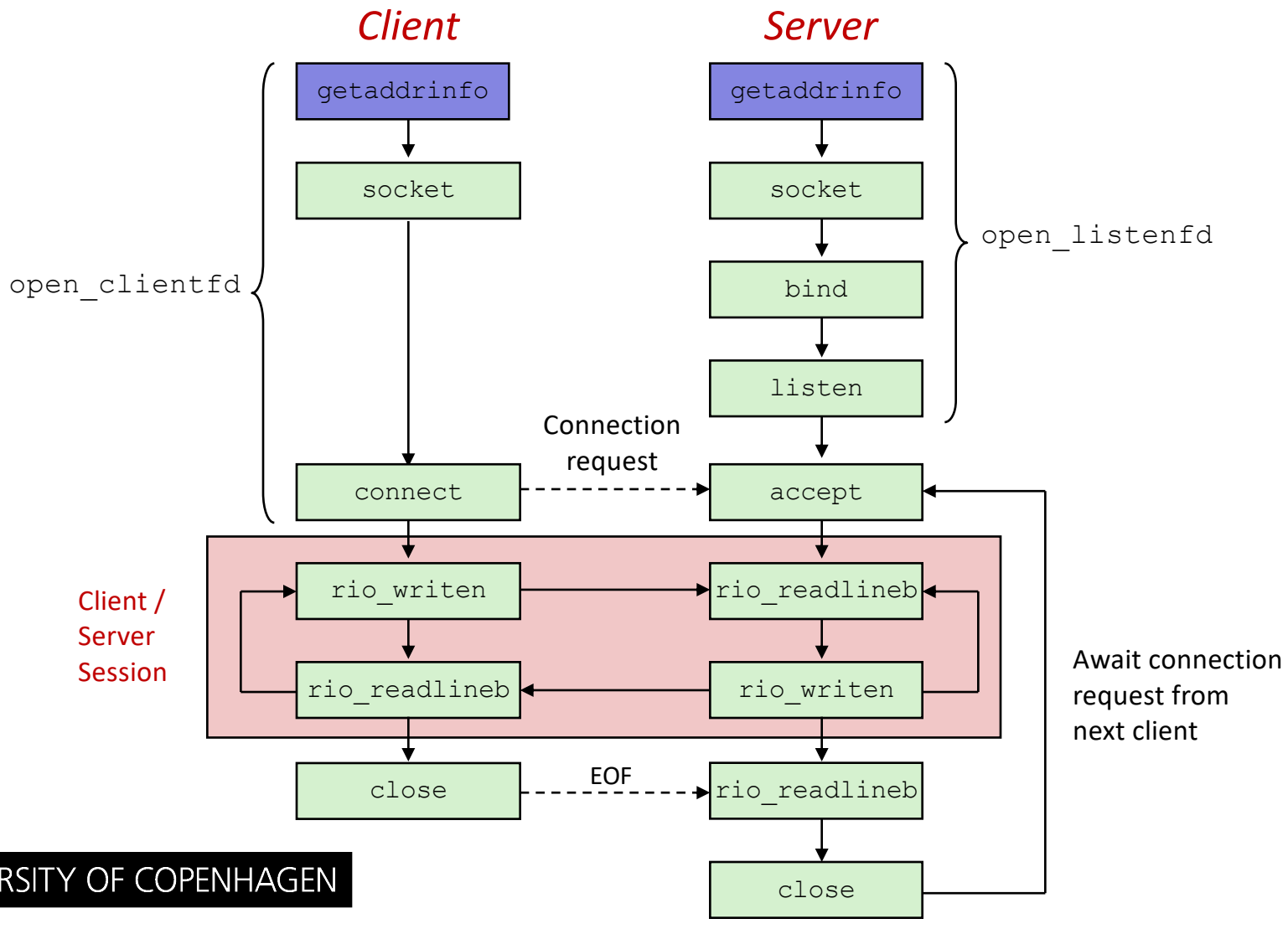
Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
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




Client
Server
getaddrinfo
getaddrinfo
socket
socket
bind
listen
open_clientfd
open_listenfd
Connection request
connect
accept
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client

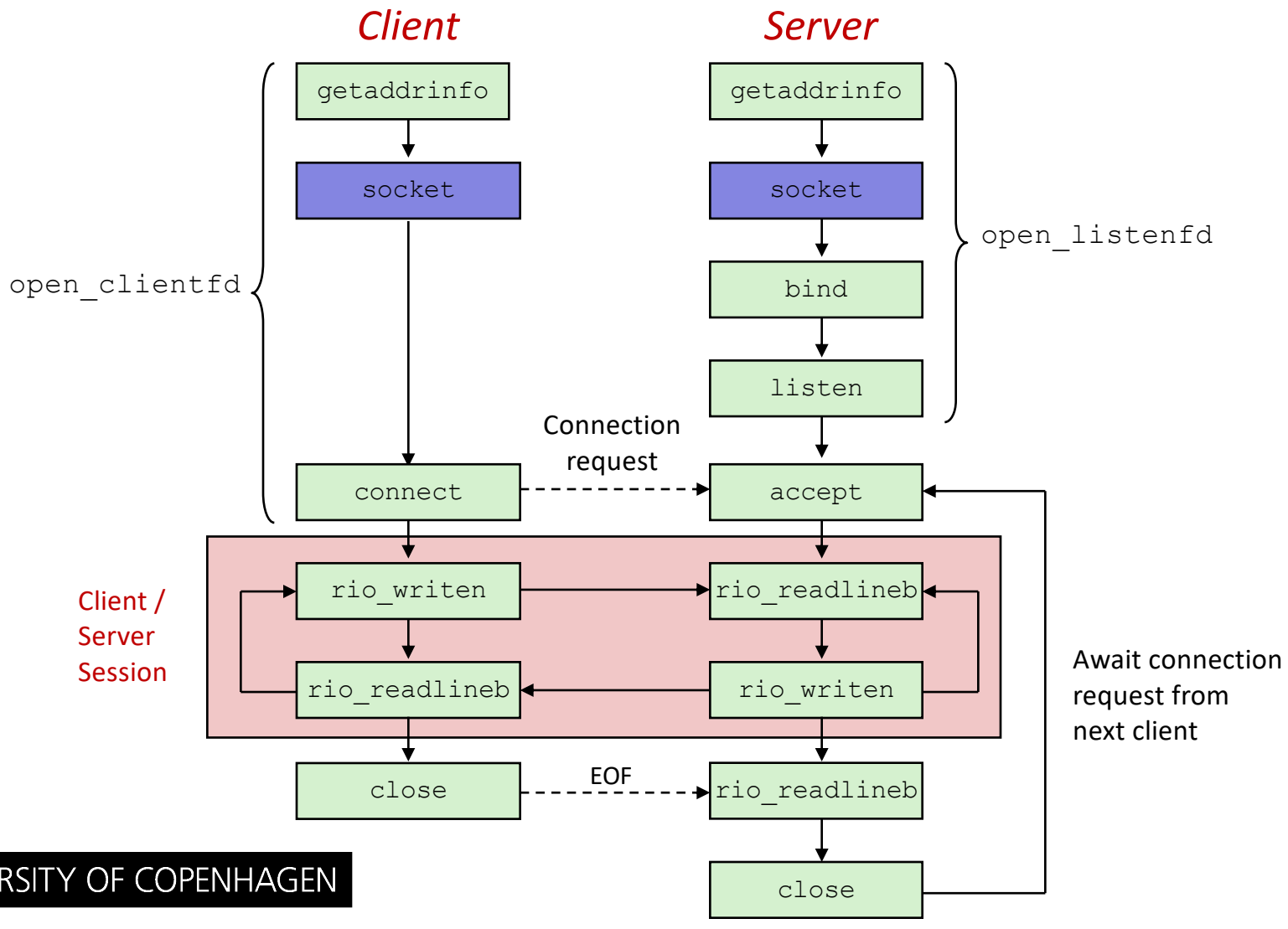
Host and Service Conversion: `getaddrinfo`

```
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.


Client
Server
getaddrinfo
getaddrinfo
socket
socket
bind
listen
open_clientfd
open_listenfd
Connection request
connect
accept
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client

Sockets Interface: `socket`

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

Example:

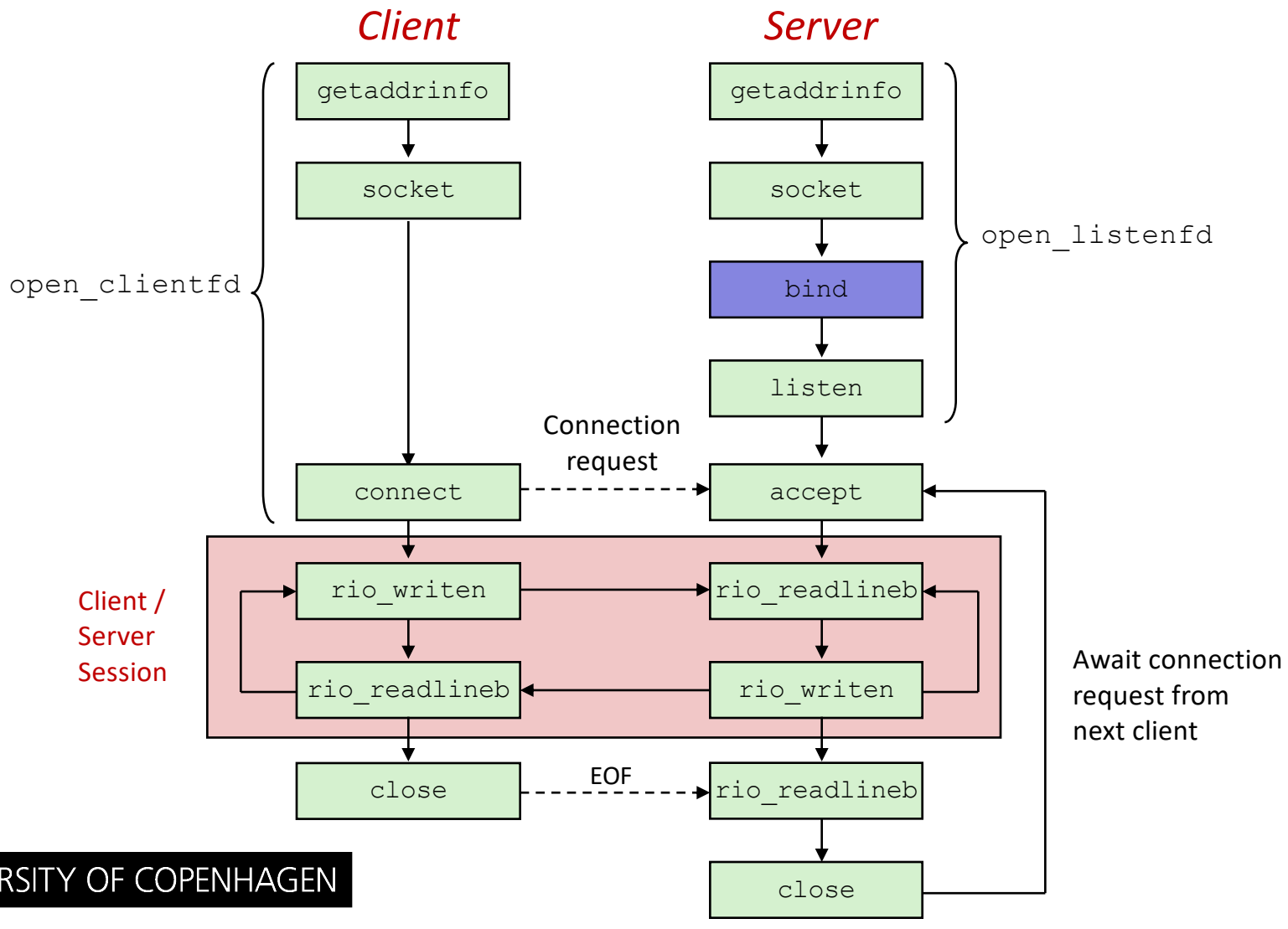
```
int socket(int domain, int type, int protocol)
```

```
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.


Client
Server
getaddrinfo
socket
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
Connection request
connect
accept
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client

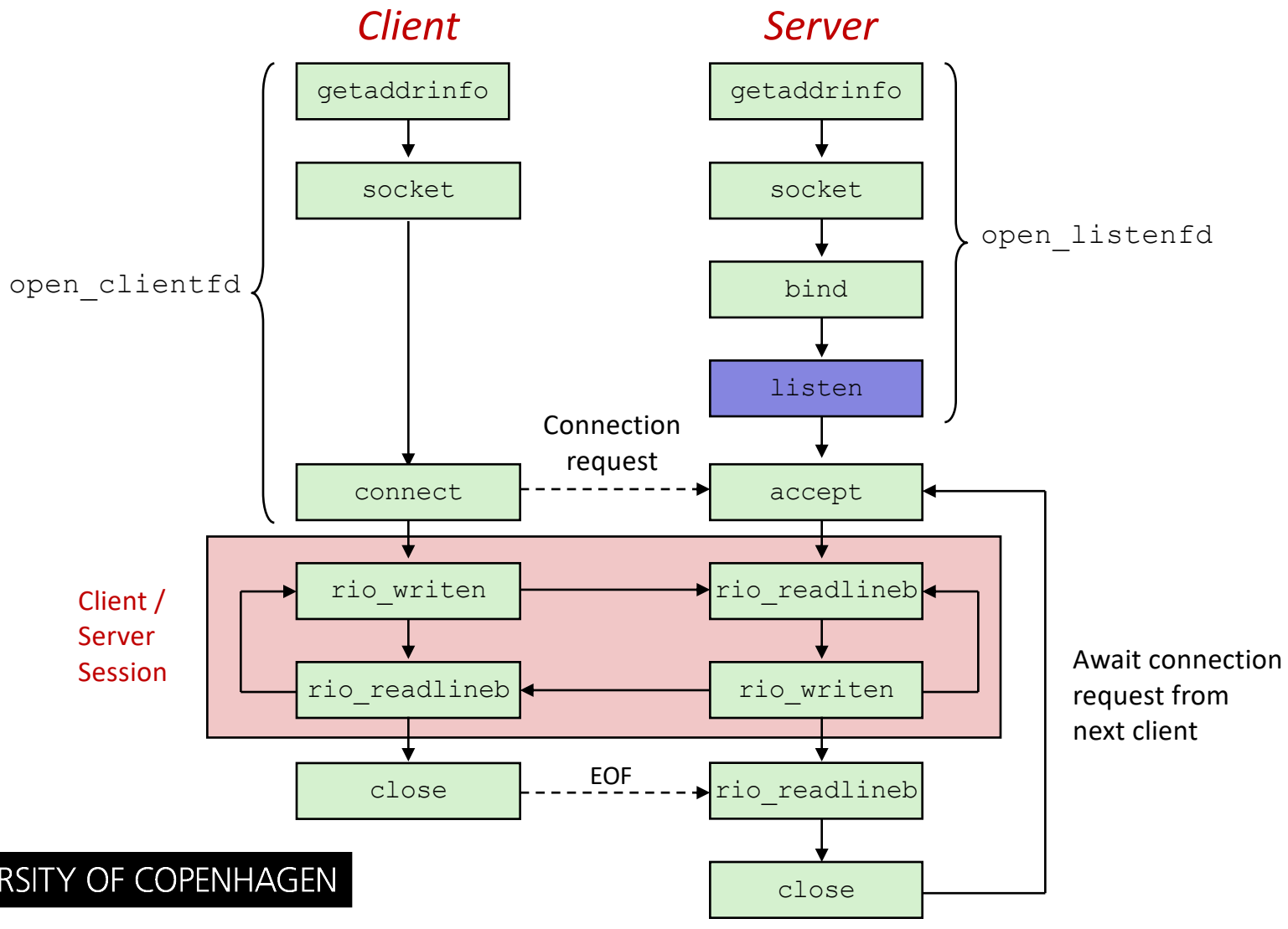
Sockets Interface: `bind`

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`.


Client
Server
getaddrinfo
socket
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
Connection
request
connect
accept
Client /
Server
Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection
request from
next client

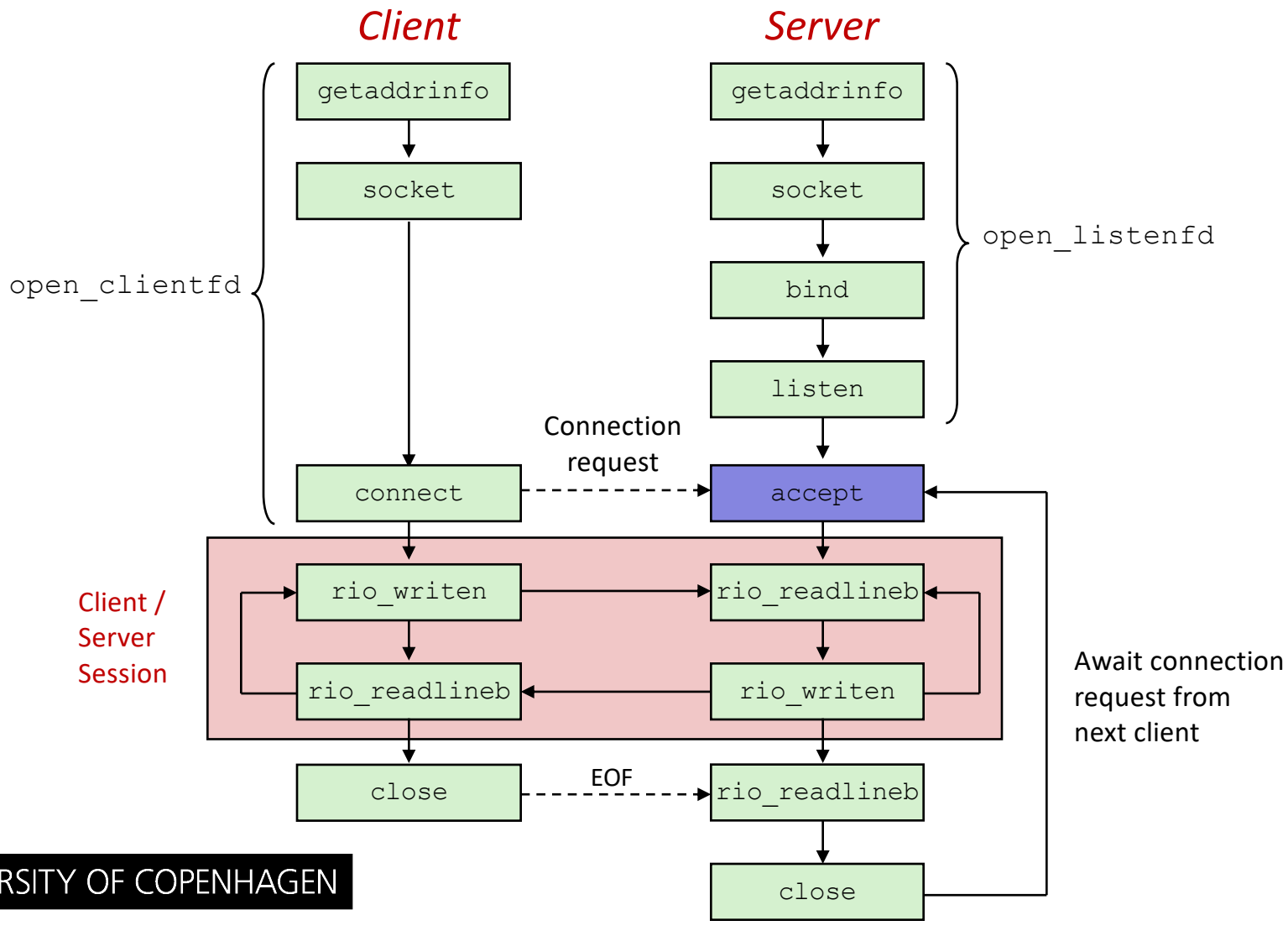
Sockets Interface: `listen`

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.


Client
Server
getaddrinfo
socket
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
Connection
request
connect
accept
Client /
Server
Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection
request from
next client

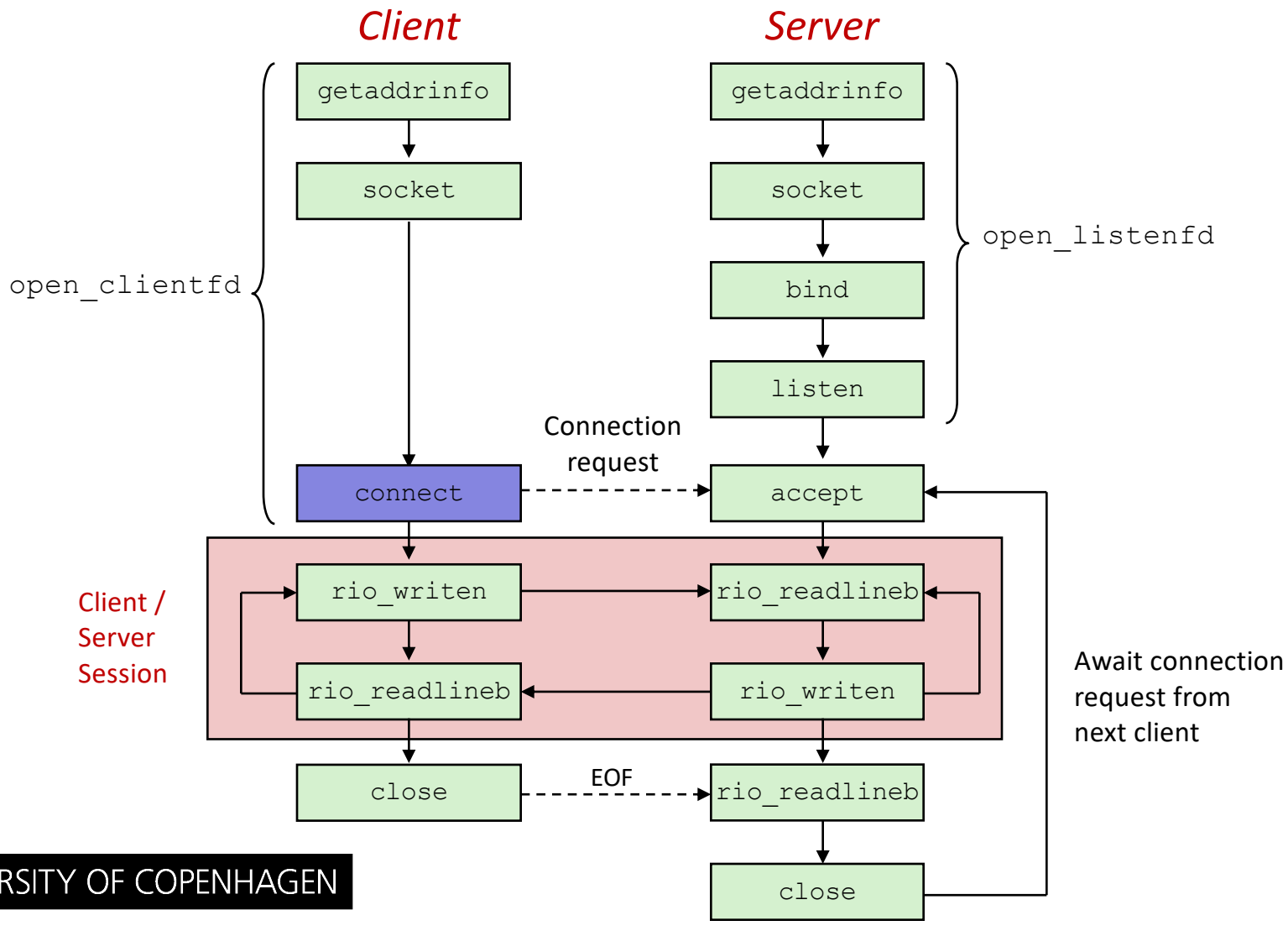
Sockets Interface: `accept`

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.


Client
Server
getaddrinfo
socket
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
Connection request
connect
accept
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client

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

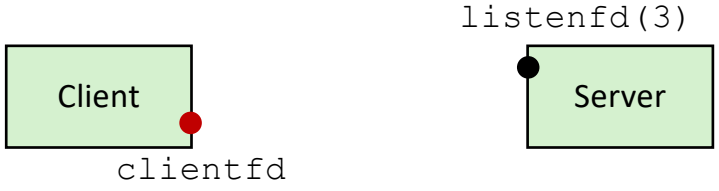
`(x:y, addr.sin_addr:addr.sin_port)`

`x` is client address

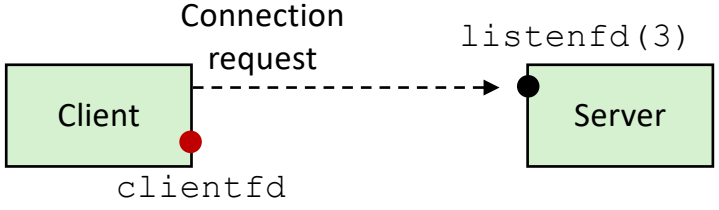
`y` is ephemeral port that uniquely identifies client process on client host

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

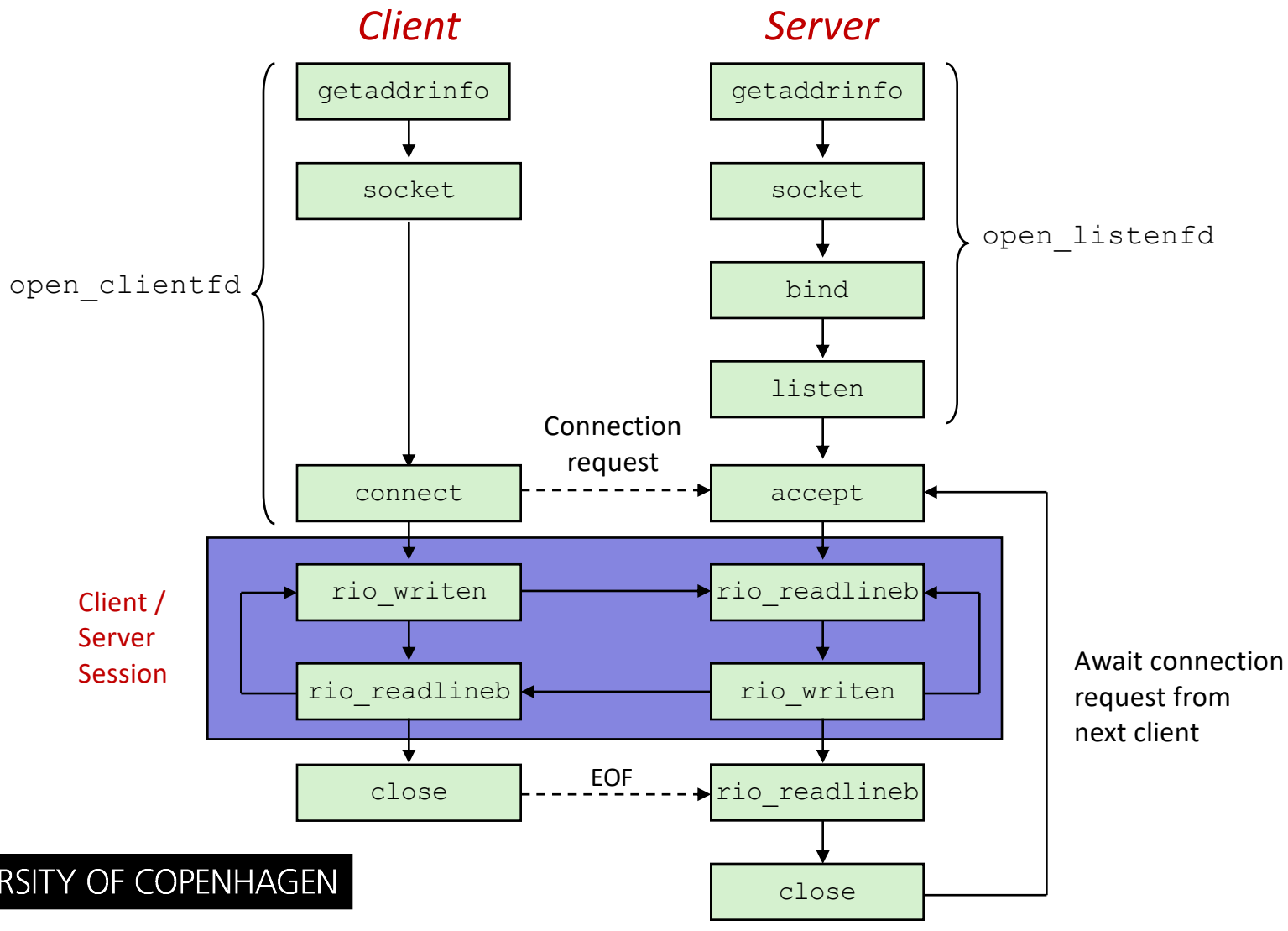
- End point for client connection requests
- Created once and exists for lifetime of the server

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


Client
Server
getaddrinfo
socket
open_clientfd
getaddrinfo
socket
bind
listen
open_listenfd
Connection request
connect
accept
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client

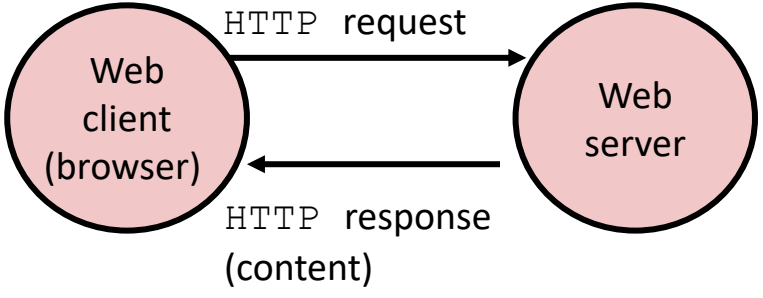
Web Server Basics

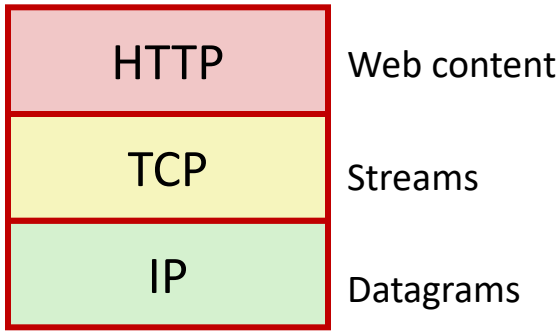
Clients and servers communicate using the HyperText Transfer Protocol (HTTP)

- 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

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

```c
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

Take-Aways

Network as a strictly layered system: physical (ethernet), kernel (IP/TCP), applications

Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network via naming scheme and delivery mechanism.

Socket as communication abstraction. 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. Client connects to a server via a socket. Servers bind sockets to address:port, listen and accept incoming connections from clients. Thereafter, clients and servers can read and write.