UNIVERSITY OF COPENHAGEN

Computer Networking I

David Marchant

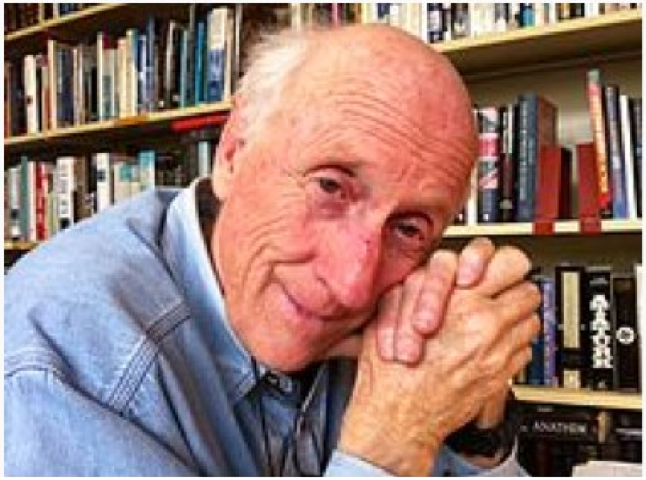
Based on slides compiled by Marcos Vaz Salles, with adaptions by Vivek Shah and Michael Kirkedel Thomsen

Networking is Relevant

Source: Freedman

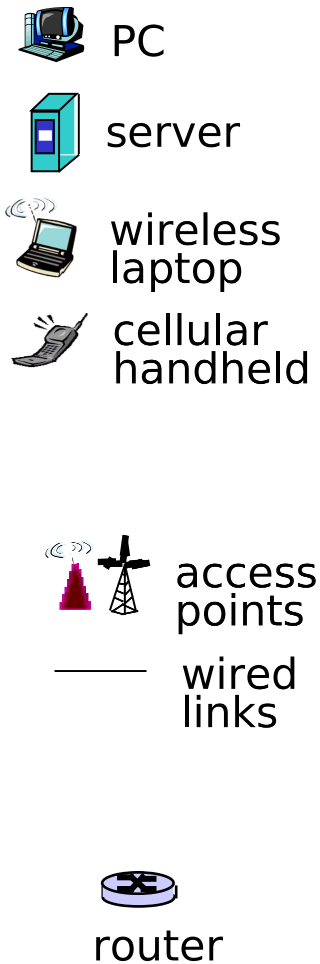
Information wants to be free because it has become so cheap to distribute, copy, and recombine... It wants to be expensive because it can be immeasurably valuable to the recipient. (1985)

Stewart Brand

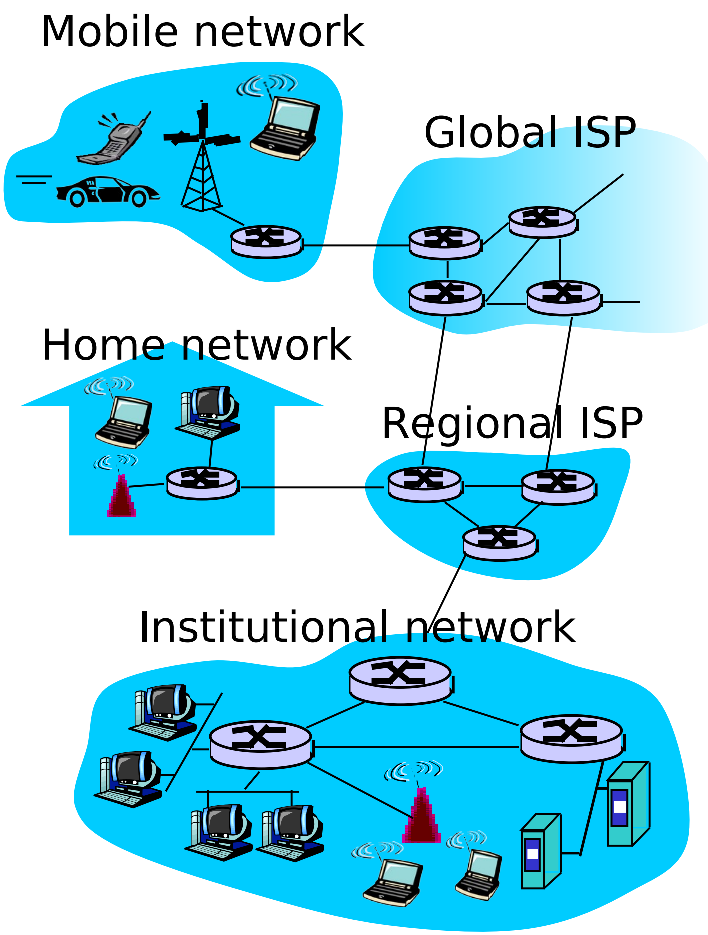


Source: Freedman

What’s the Internet: “nuts and bolts” view



- A network of networks
- Millions of connected computing devices: *hosts = end systems*
  - running *network apps*
- *communication links*
  - fiber, copper, radio, satellite
  - transmission rate = *bandwidth*
- *routers:* forward packets (chunks of data)



Source: Kurose & Ross (partial)

Key Concepts in Networking

- Protocols
  - Speaking the same language
  - Syntax and semantics
- Layering
  - Standing on the shoulders of giants
  - A key to managing complexity
- Resource allocation
  - Dividing scarce resources among competing parties
  - Memory, link bandwidth, wireless spectrum, paths
- Naming
  - What to call computers, services, protocols, …

Source: Freedman

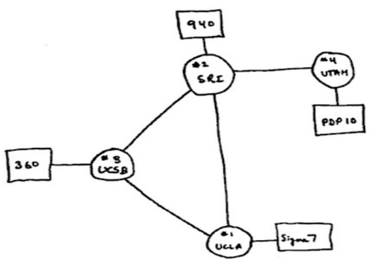
Key Concepts in Networking

- Protocols
  - Speaking the same language
  - Syntax and semantics
- Layering
  - Standing on the shoulders of giants
  - A key to managing complexity
- Resource allocation
  - Dividing scarce resources among competing parties
  - Memory, link bandwidth, wireless spectrum, paths
- Naming
  - What to call computers, services, protocols, …

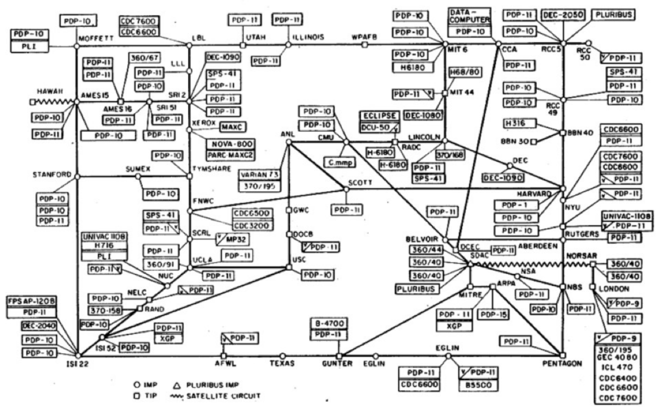
Source: Freedman

Key Concepts in Networking

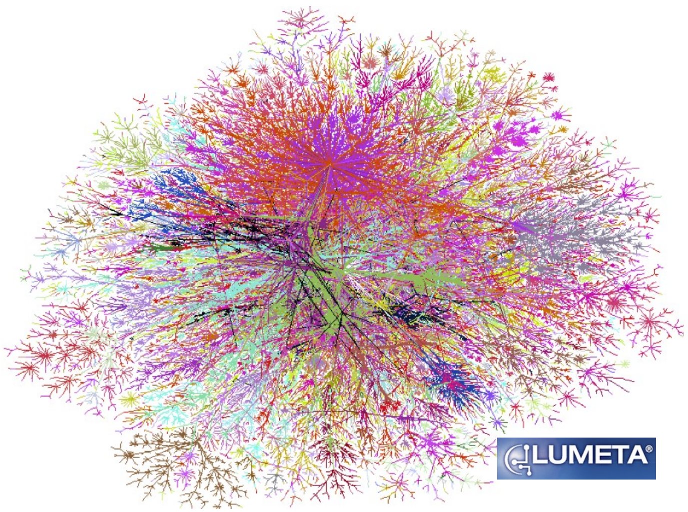
- Protocols
  - Speaking the same language
  - Syntax and semantics



1969



1977



1998

All speak IPv4
“Internet Protocol version 4”

Source: Freedman

Protocol design is about tradeoffs

- **How should hosts and routers communicate?**
  - Standard protocol
  - Fast: Machine readable in hardware at line rates

- **Browsers, web servers, and proxies?**
  - Can be slower: software readable
  - Human readable
  - Extensible and forward-compatible
  - Not everybody might be familiar with extensions

Source: Freedman

IPv4 Packet

4-bit Version	4-bit Header Length	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)
16-bit Identification		3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)	8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address			
32-bit Destination IP Address			
Options (if any)			
Payload			

20-byte header

Source: Freedman

Example: HyperText Transfer Protocol

Request

```
GET /courses/archive/spr09/cos461/ HTTP/1.1
Host: www.cs.princeton.edu
User-Agent: Mozilla/4.03
CRLF
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 2 Feb 2009 13:09:03 GMT
Server: Netscape-Enterprise/3.5.1
Last-Modified: Mon, 21 Feb  2009 11:12:23 GMT
Content-Length: 42
CRLF
Site under construction
```

Source: Freedman (partial)

Key Concepts in Networking

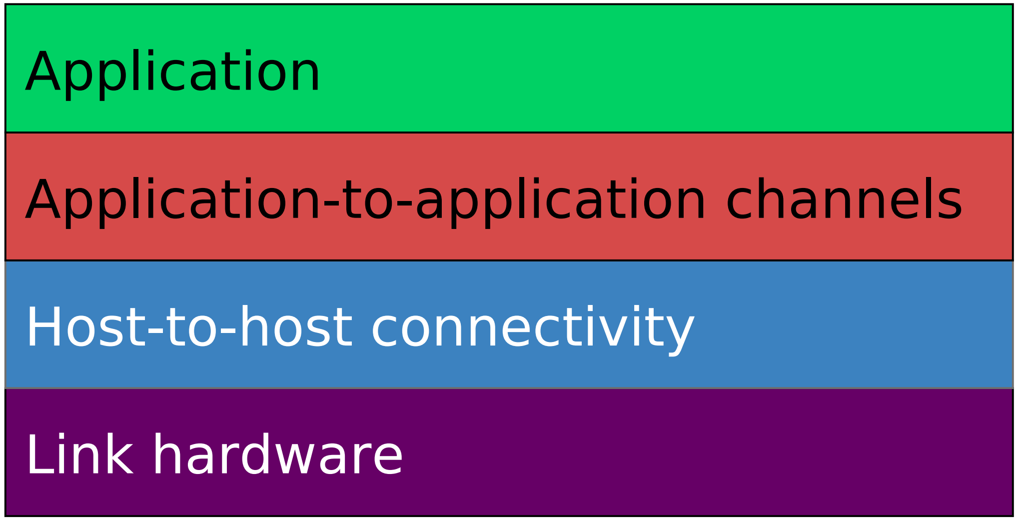
- Protocols
  - Speaking the same language
  - Syntax and semantics
- Layering
  - Standing on the shoulders of giants
  - A key to managing complexity
- Resource allocation
  - Dividing scarce resources among competing parties
  - Memory, link bandwidth, wireless spectrum, paths
- Naming
  - What to call computers, services, protocols, …

Source: Freedman

Layering = Functional Abstraction

- Sub-divide the problem
  - Each layer relies on services from layer below
  - Each layer exports services to layer above

- Interface between layers defines interaction
  - Hides implementation details
  - Layers can change without disturbing other layers



Source: Freedman

Key Concepts in Networking

Application Layer

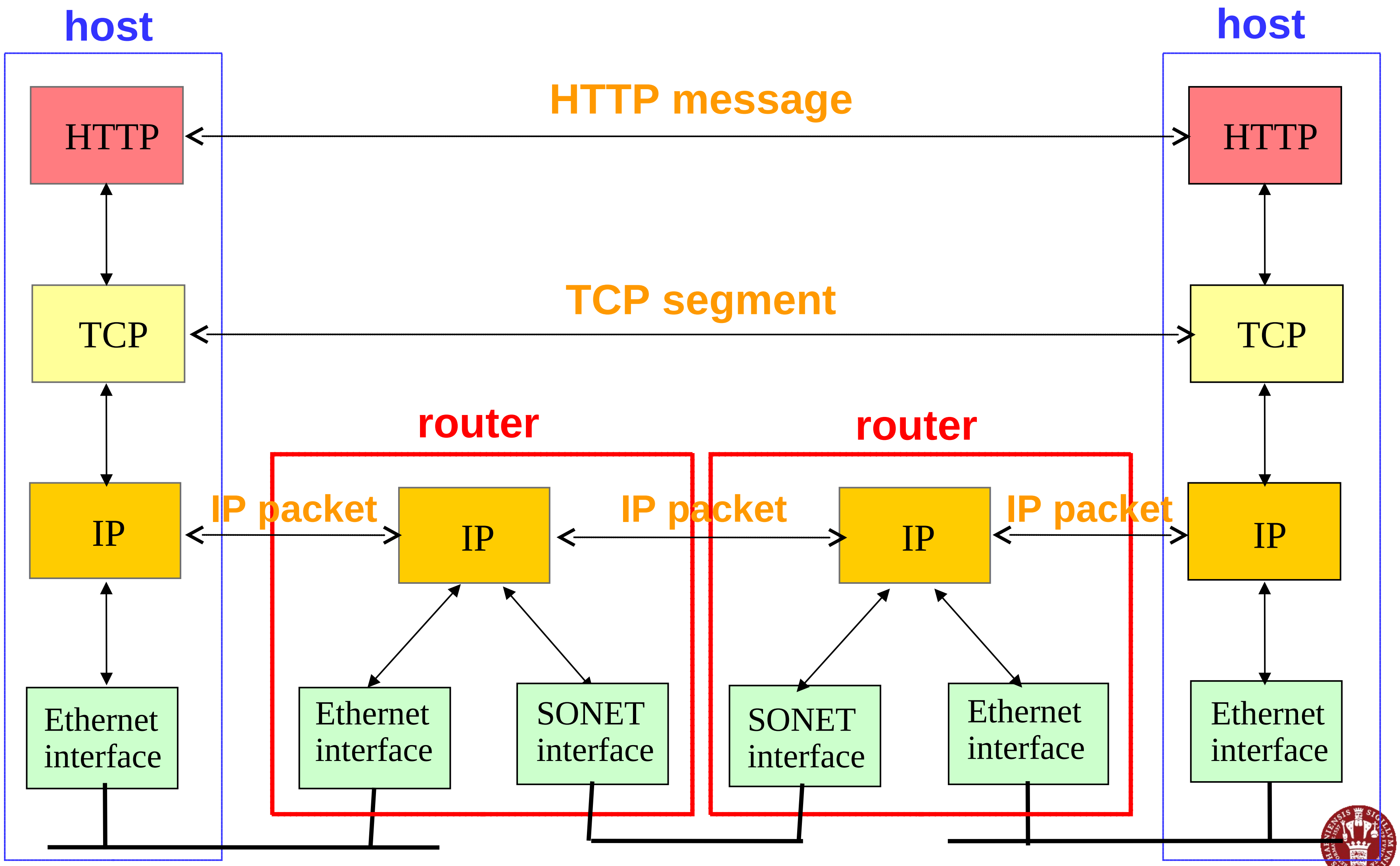
Transport Layer

Network Layer

Link Layer

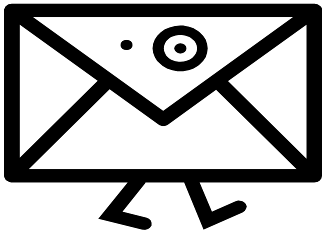
Physical Layer

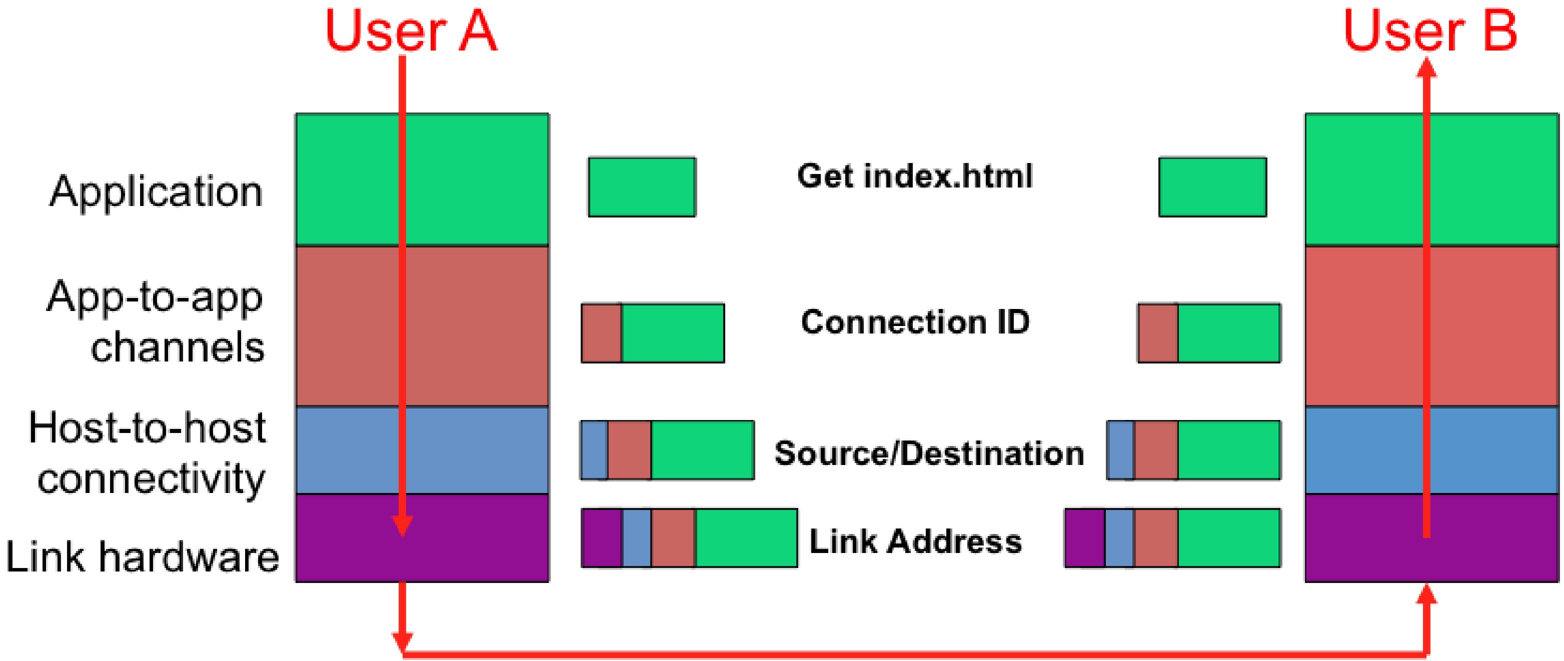
IP Suite: End Hosts vs. Routers



Source: Freedman

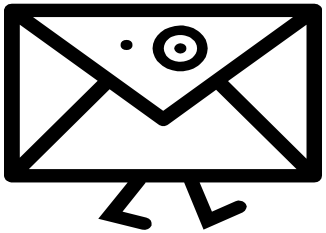
Layer Encapsulation in HTTP

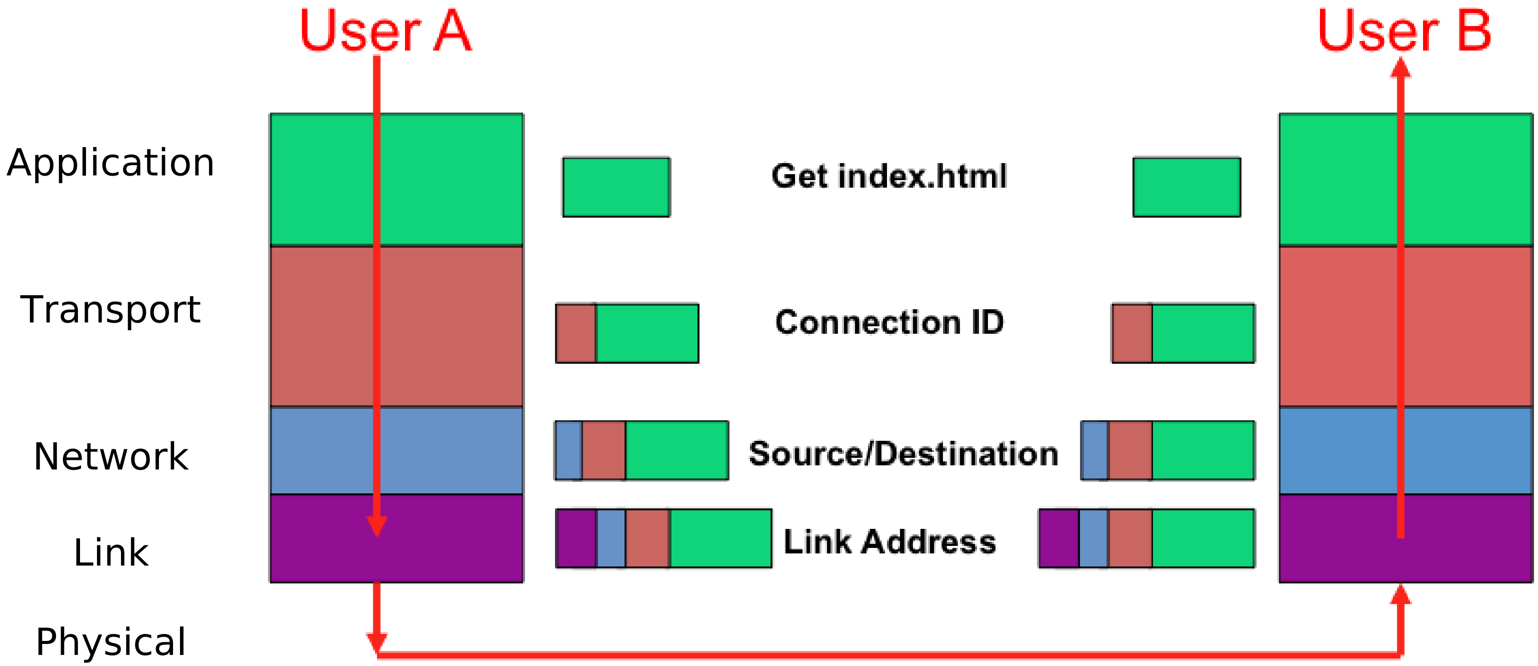




Source: Freedman

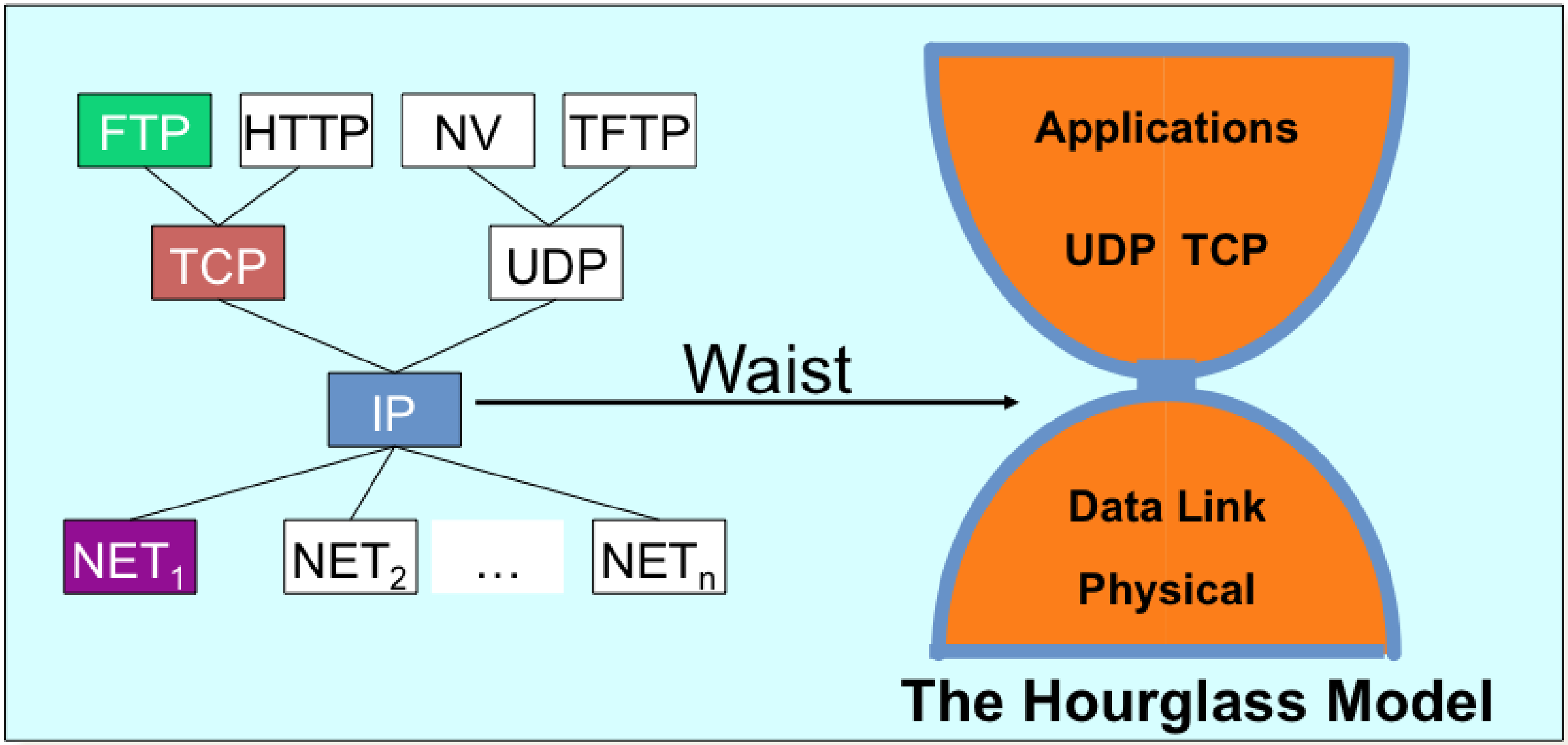
Layer Encapsulation in HTTP



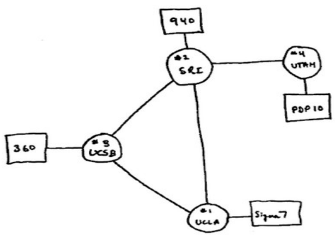


Source: Freedman

The Internet Protocol Suite



The waist facilitates interoperability



Source: Freedman

Key Concepts in Networking

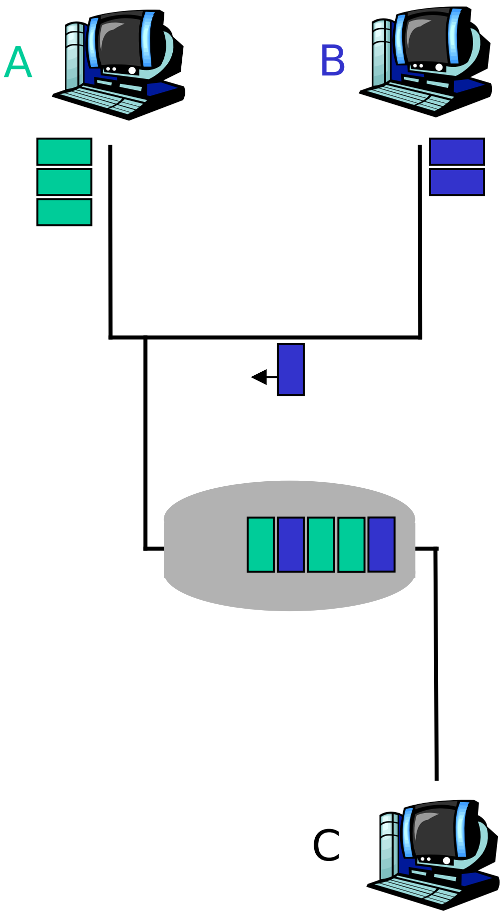
- Protocols
  - Speaking the same language
  - Syntax and semantics
- Layering
  - Standing on the shoulders of giants
  - A key to managing complexity
- Resource allocation
  - Dividing scarce resources among competing parties
  - Memory, link bandwidth, wireless spectrum, paths
- Naming
  - What to call computers, services, protocols, …

Source: Freedman

Network Core: Packet Switching

Data broken up into smaller 'packets'

- Can be sent seperately, each using full bandwidth
- Full bandwidth used on a single packet
- Packets from different sources are interspersed
- Resources only used when needed
- Unreliable transfer



Key Concepts in Networking

- Protocols
  - Speaking the same language
  - Syntax and semantics
- Layering
  - Standing on the shoulders of giants
  - A key to managing complexity
- Resource allocation
  - Dividing scarce resources among competing parties
  - Memory, link bandwidth, wireless spectrum, paths
- Naming
  - What to call computers, services, protocols, …

Source: Freedman

A Programmers View of the Internet

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

- 128.2.203.179

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

- 128.2.217.3 is mapped to www.cs.cmu.edu

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

Source: Freedman

IPv4 and IPv6

- **The original Internet Protocol, with its 32-bit addresses, is known as *Internet Protocol Version 4* (IPv4)**

- **1996: Internet Engineering Task Force (IETF) introduced *Internet Protocol Version 6* (IPv6) with 128-bit addresses**
  - Intended as the successor to IPv4

- **As of 2021, majority of Internet traffic still carried by IPv4**
  - Only 30-35% of users access Google services using IPv6.

- **We will focus on IPv4, but will show you how to write networking code that is protocol-independent.**

Source: Freedman

IPv4, IPv6 and beyond

IPv0	Reserved
IPv1-3	Initial attempts, became IPv4 and TCP(‘73-’79)
IPv4	First actually deployed protocol (‘81)
5	Experiment (‘90 -’95) into video streaming protocol
IPv6	Improvement on IPv4 (‘95)
IPv7	Incorrectly assigned when IPv6 was assumed to already be in use(‘88)
8	PIP, obsolete replacement for IPv4 (‘94)
9 IPv9 IPv9	TUBA, obsolete replacement for IPv4 (’92), April Fools Joke by IETF (‘94), Chinese research project, seemingly abandoned (‘04)
IPv11-14	Unassigned
IPv15	Reserved

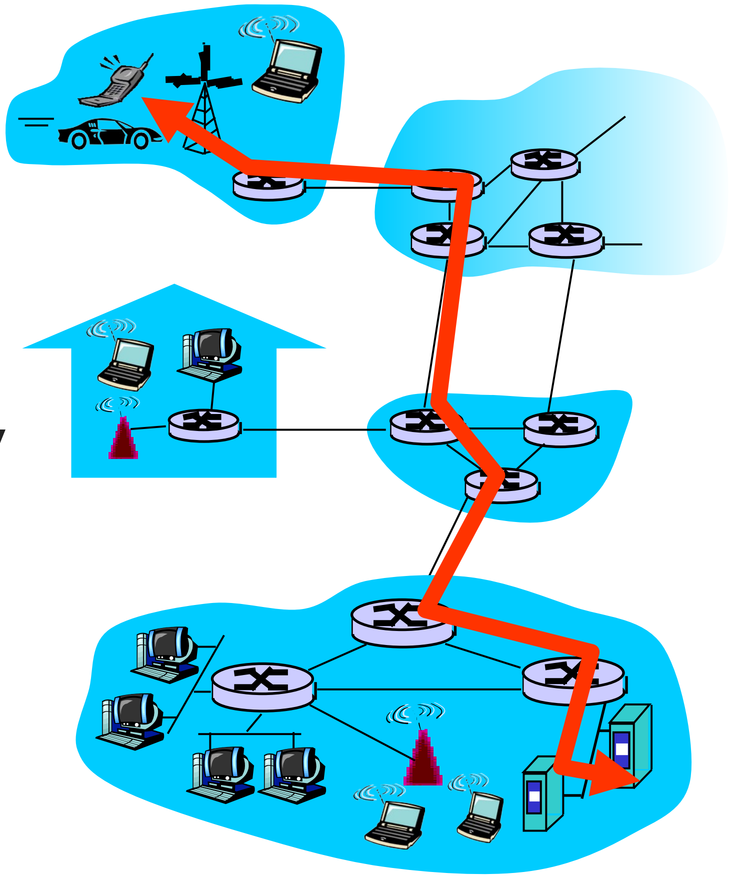
Source: Freedman

Using the network

Source: Freedman

Network Core: Circuit Switching

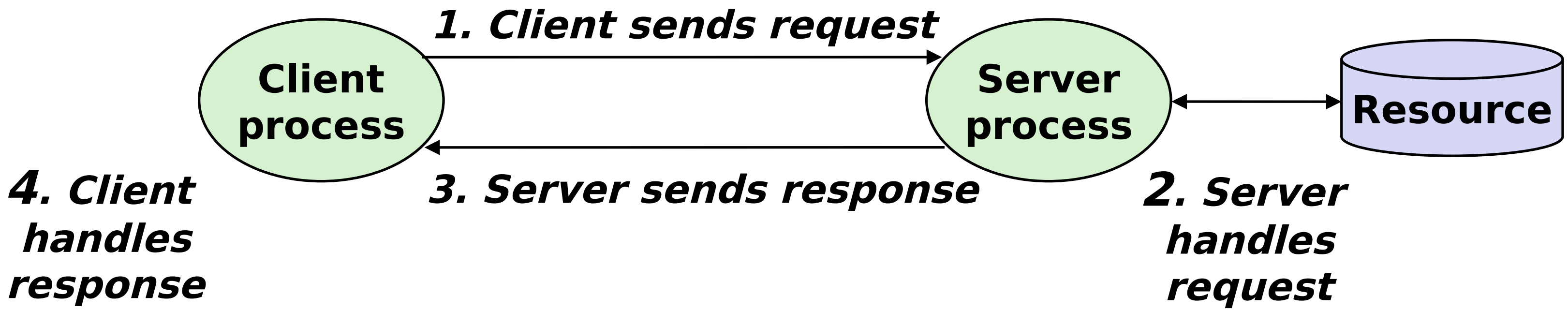
- To send a message we can use libaries at the application layer
- Underlying layers will take care of most of the work
- Can be done in (almost) any lanaguage. C is difficult though, so we'll use Python



Source: Kurose & Ross

A Client-Server Transaction

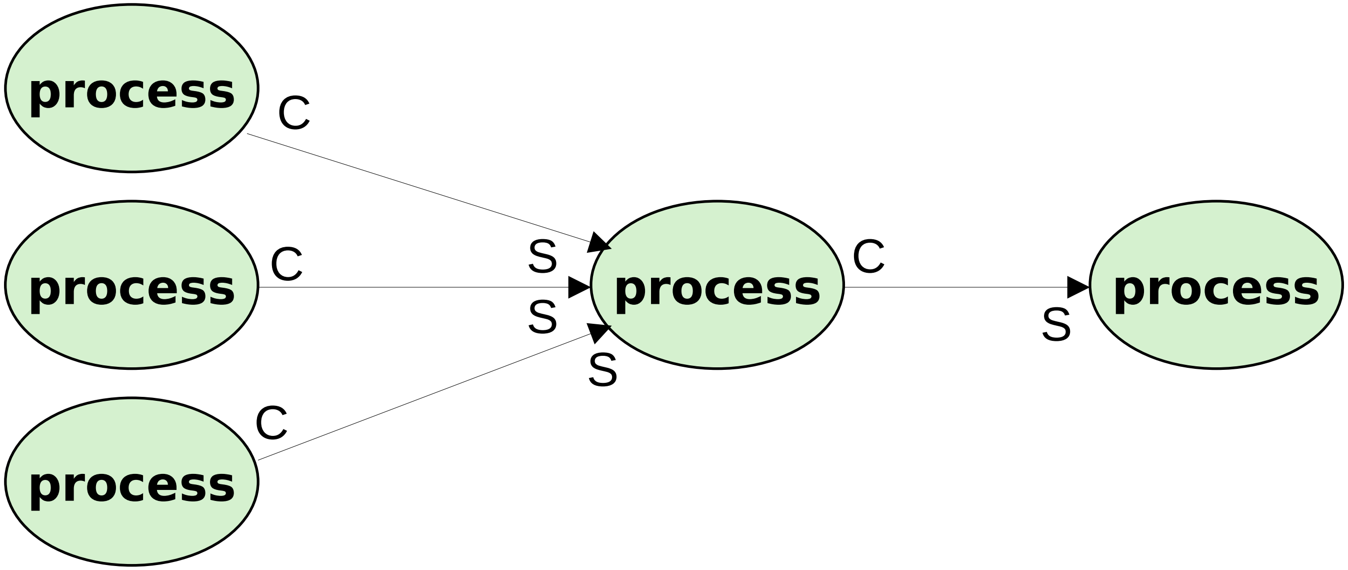
- **Most network applications are based on the client-server model:**
  - A ***server*** process and one or more ***client*** processes
  - Server manages some ***resource***
  - Server provides ***service*** by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)



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

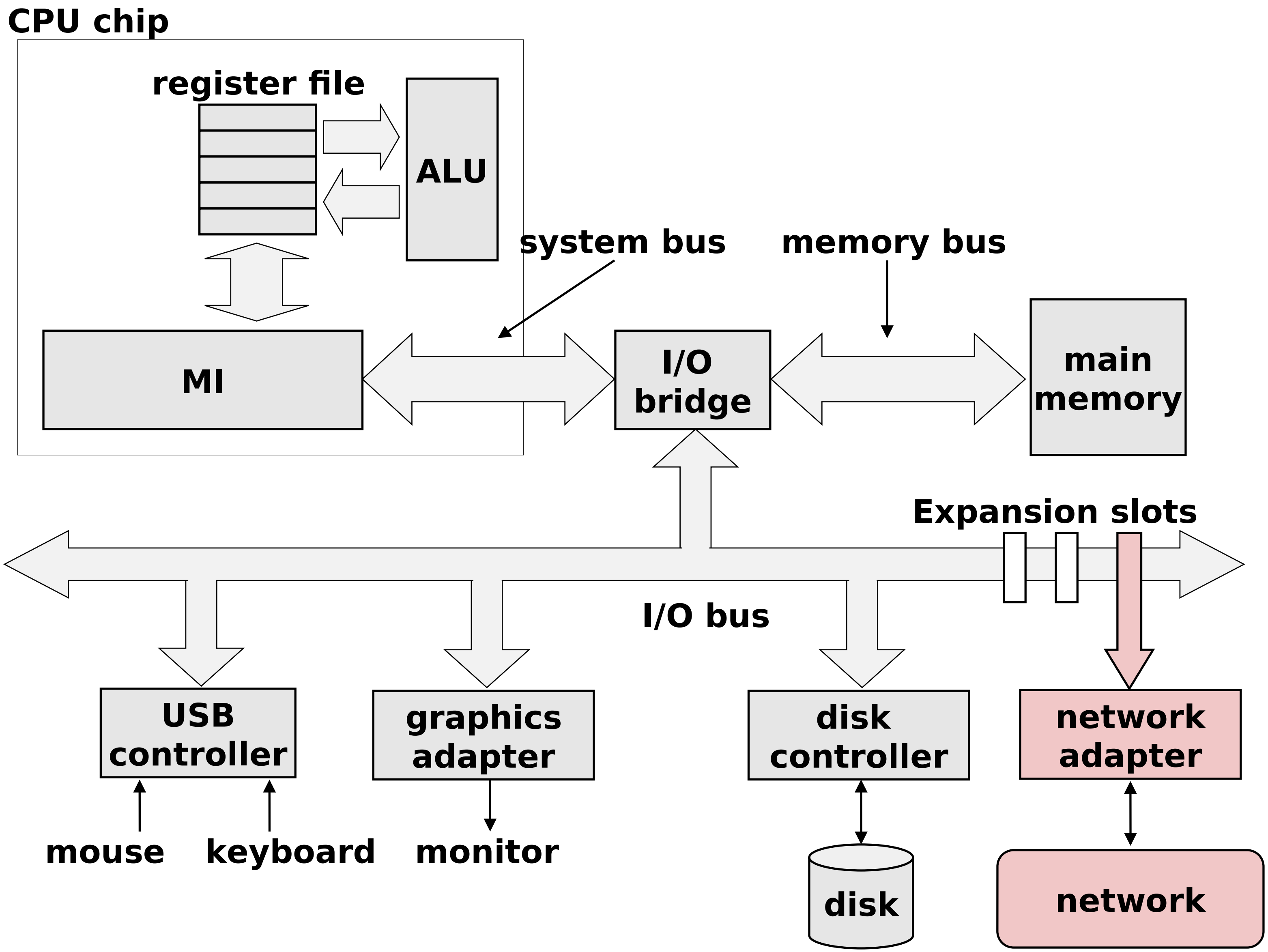
A Client-Server Transaction

- **Most network applications are based on the client-server model:**
  - There can be multiple ***clients*** connected to a single ***server***
  - Individual hosts can act as both ***clients*** and ***servers*** at the same time



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

Hardware Organization of a Network Host



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

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

Sockets Interface

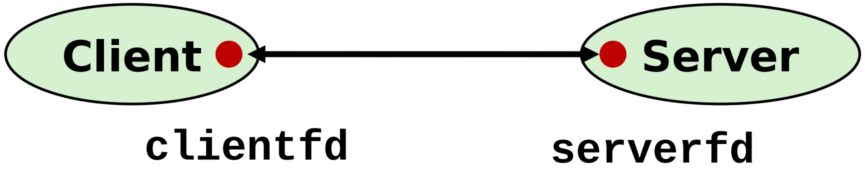
- **Set of system-level functions used in conjunction with Unix I/O to build network applications.**

- **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

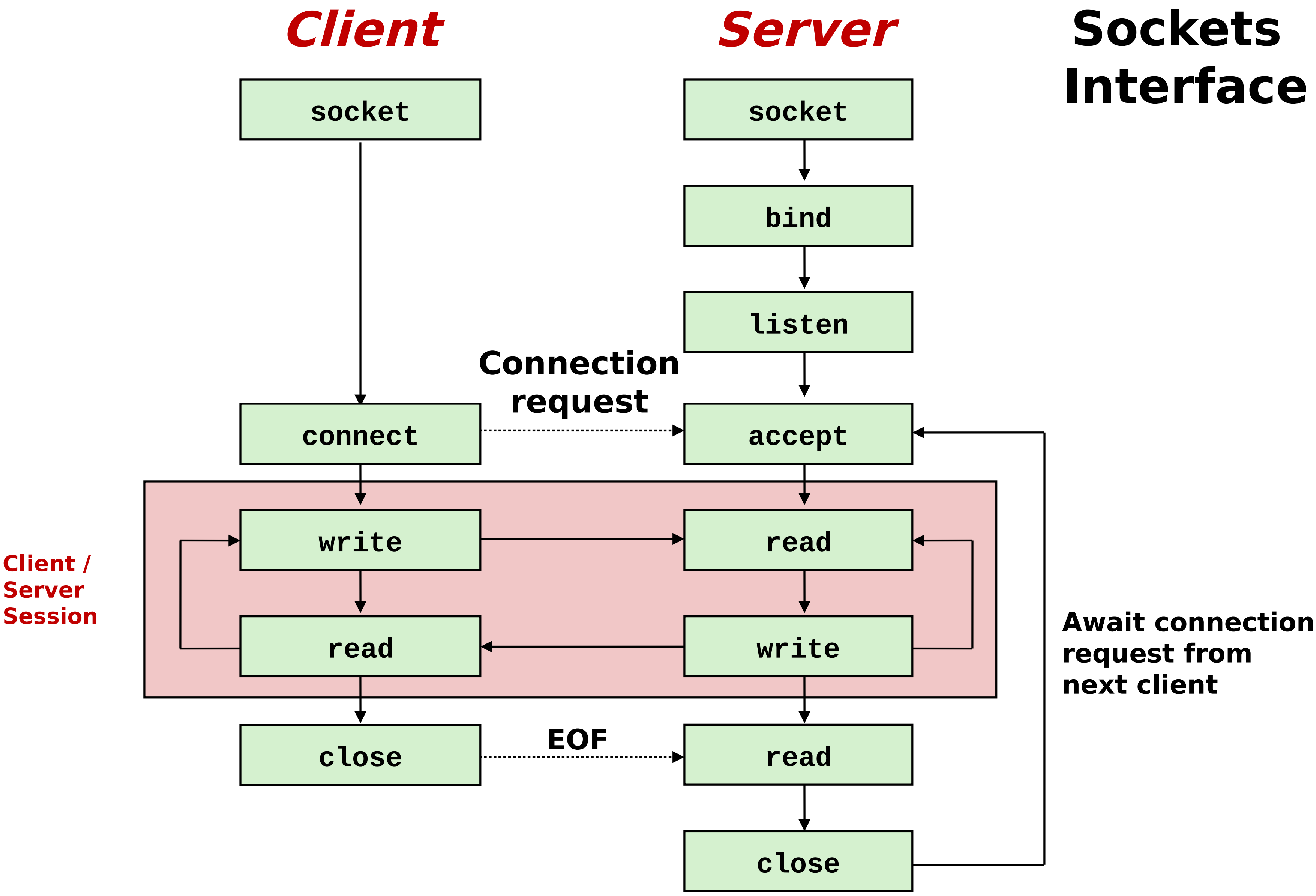
Sockets

- **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**

Sockets Interface



Sockets Interface: `socket`

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

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

- **Example:**

C:

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

int socket_fd = 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

Python:

```
from socket import *

with socket(AF_INET, SOCK_STREAM) as sock:
    ...
```

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.

C:
```
listen(socket_fd, 10);
```

Python:
```
sock.listen(10)
```

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.**

C:

```
socklen_t clientlen;
struct sockaddr_storage clientaddr;
conn_fd = accept(socket_fd, (SA *) &clientaddr, &clientlen);
```

Python:

```
Conn, conn_addr = sock.accept()
```

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

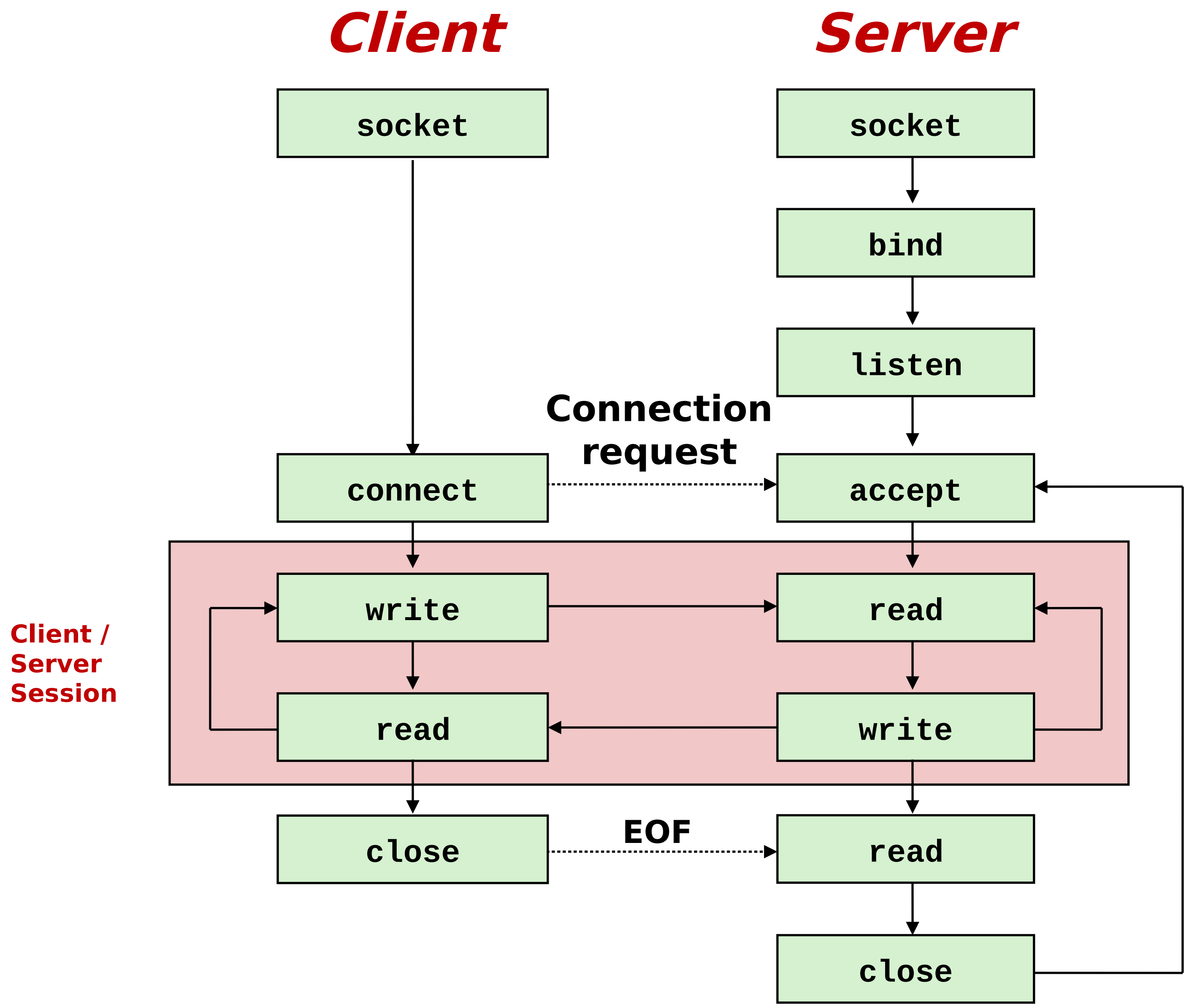
C:

```
struct sockaddr s_addr;
connect(socket_fd, (struct sockaddr *)&s_addr, sizeof(s_addr));
```

Python:

```
client_sock.connect("130.226.237.173", 56)
```

Sockets Interface



Python Example

Client:

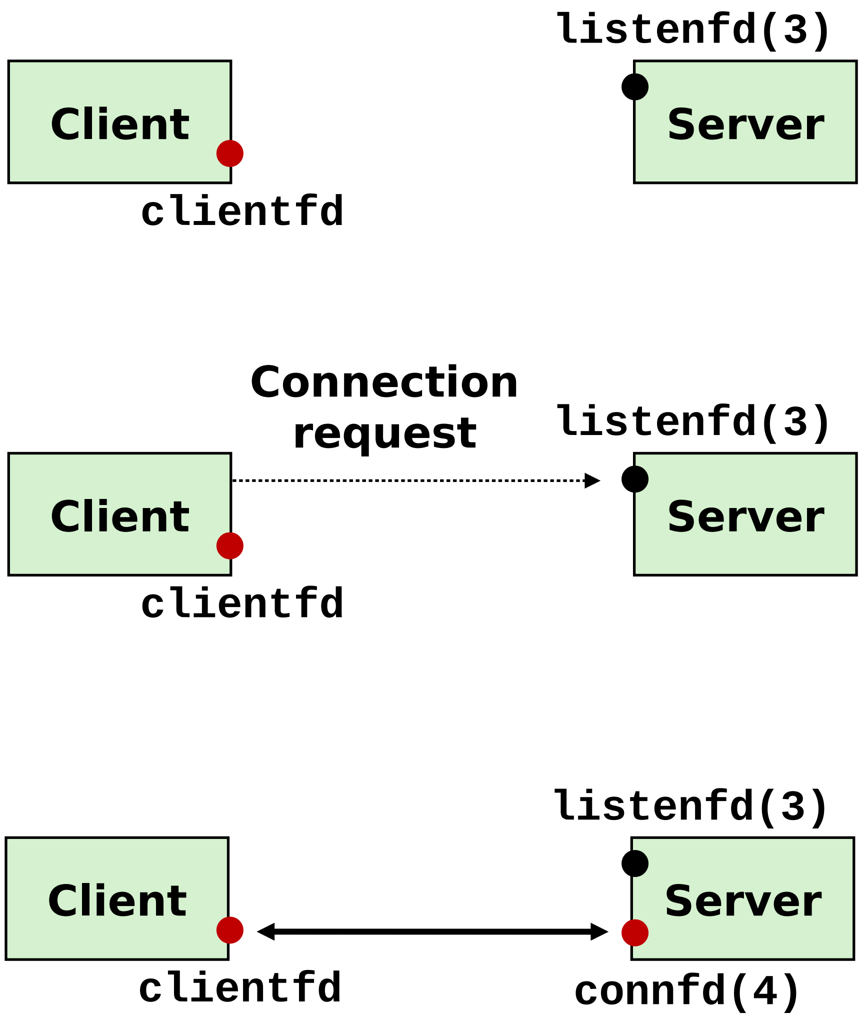
```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
    client_socket.connect((“127.0.0.1”, 5678))
    request = bytearray(“This is a message”.encode())
    client_socket.sendall(request)
    response = client_socket.recv(1024)
    print(response)
```

Note that these lines are where our processes synchronise

Server:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_ST                    socket:
    server_socket.bind((“127.0.0.1”, 5678))
    server_socket.listen()
    while True:
        connection, connection_address = server_socket.accept()
        with connection:
            message = connection.recv(1024)
            connection.sendall(response)
```

`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

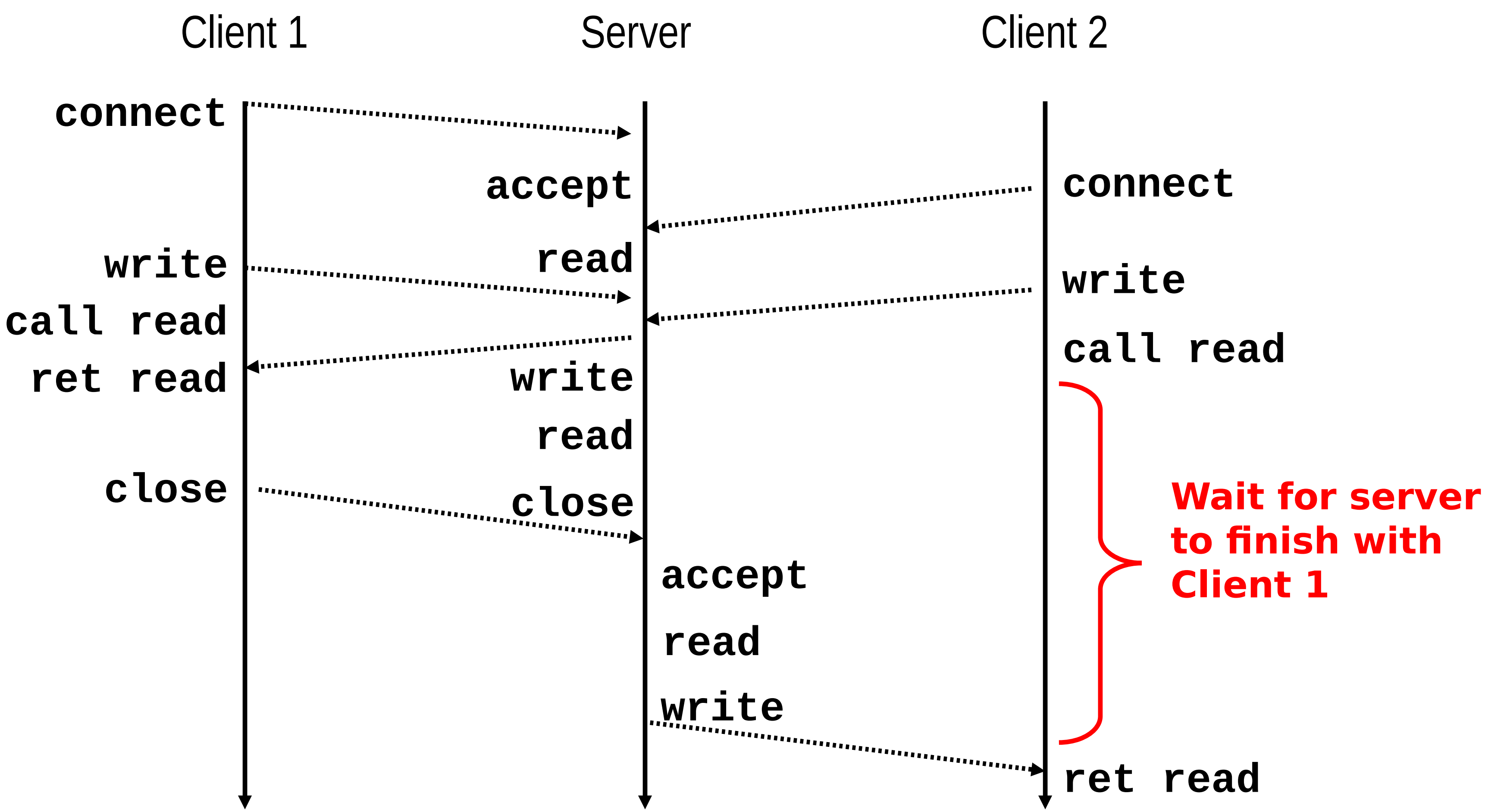
- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **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

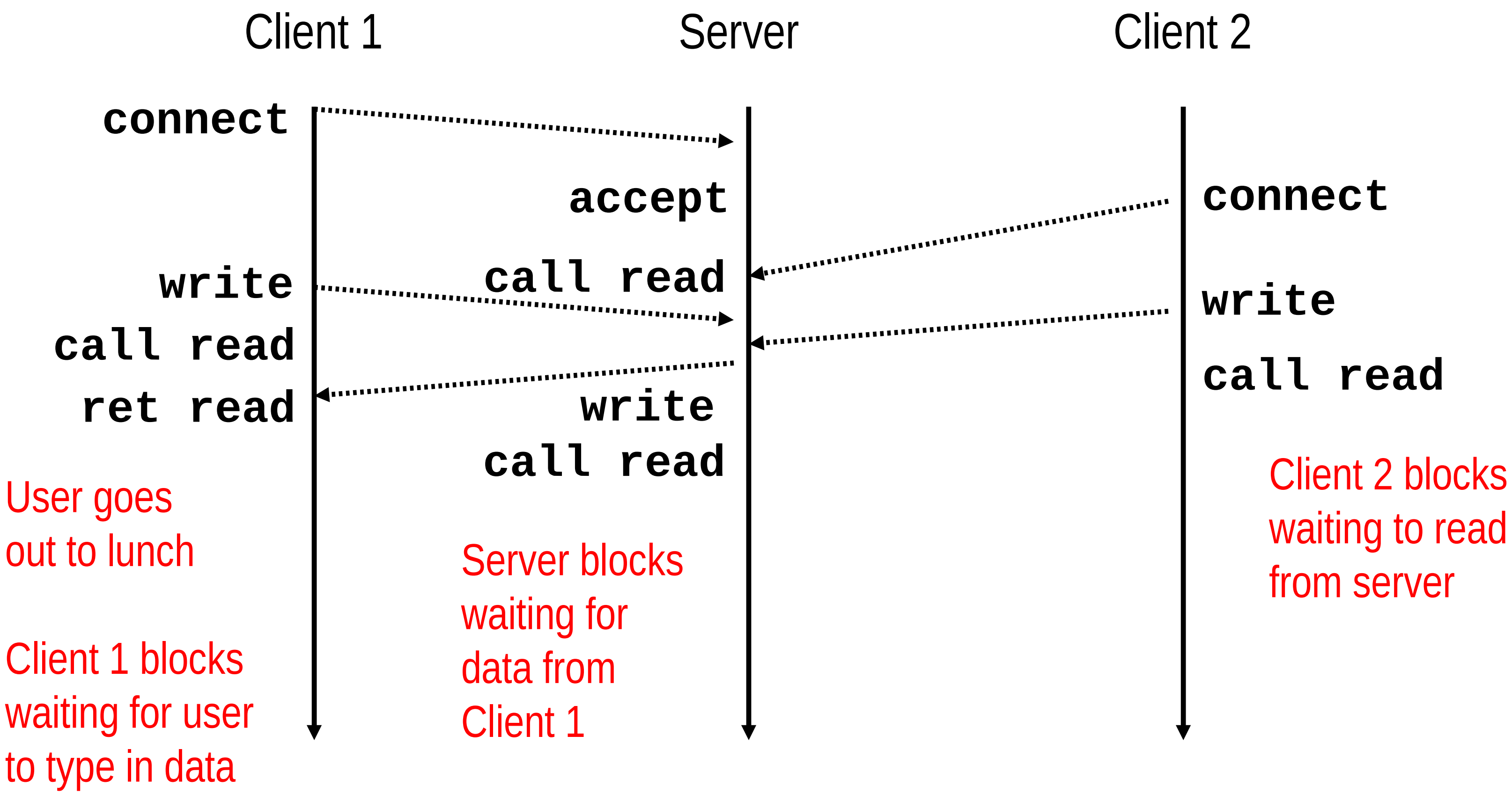
- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Iterative Servers

- Iterative servers process one request at a time

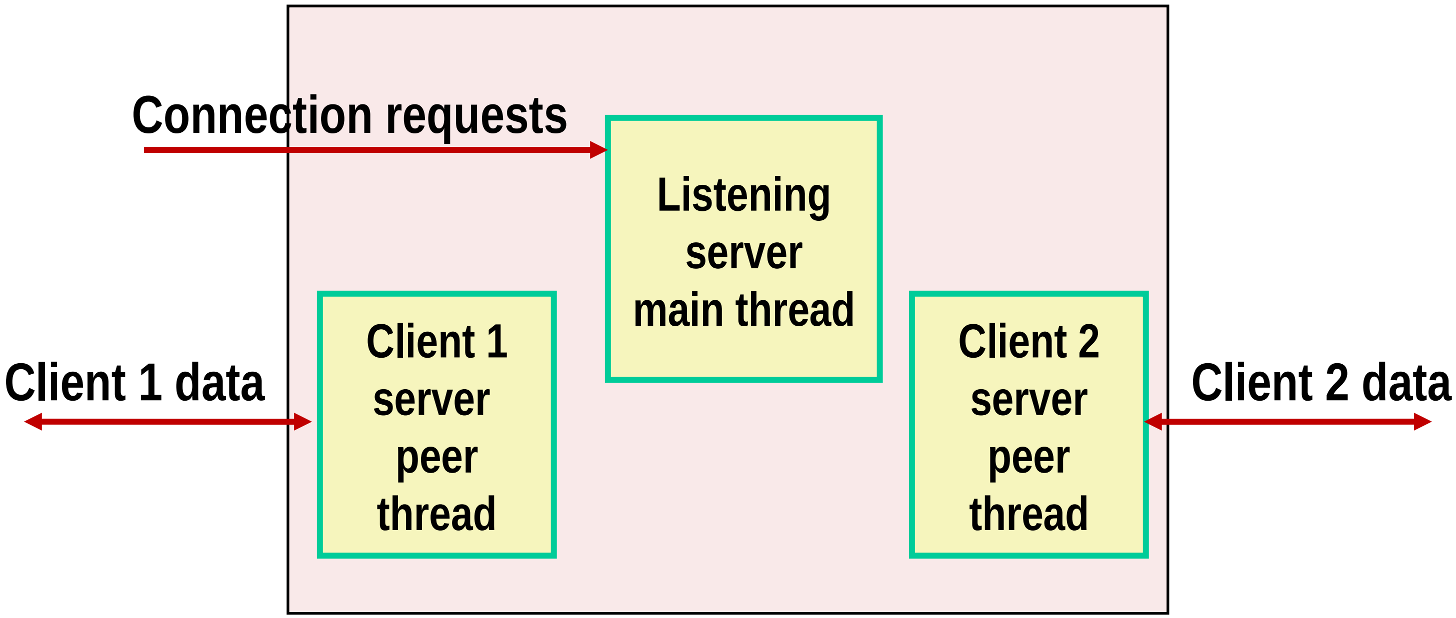


Fundamental Flaw of Iterative Servers



- **Solution: use *concurrent servers* instead**
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Thread-based Server Execution Model



- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variable
- We'll get into what this actually means after Christmas . . .

Concurrent servers

- Use socketserve library
- Two parts: Server class, and Handler class
- Server is a class to run continuously, handle errors, and spawn new threads to respond to each new request
- Handler is what actually reads in new messages and processes them

Concurrent servers

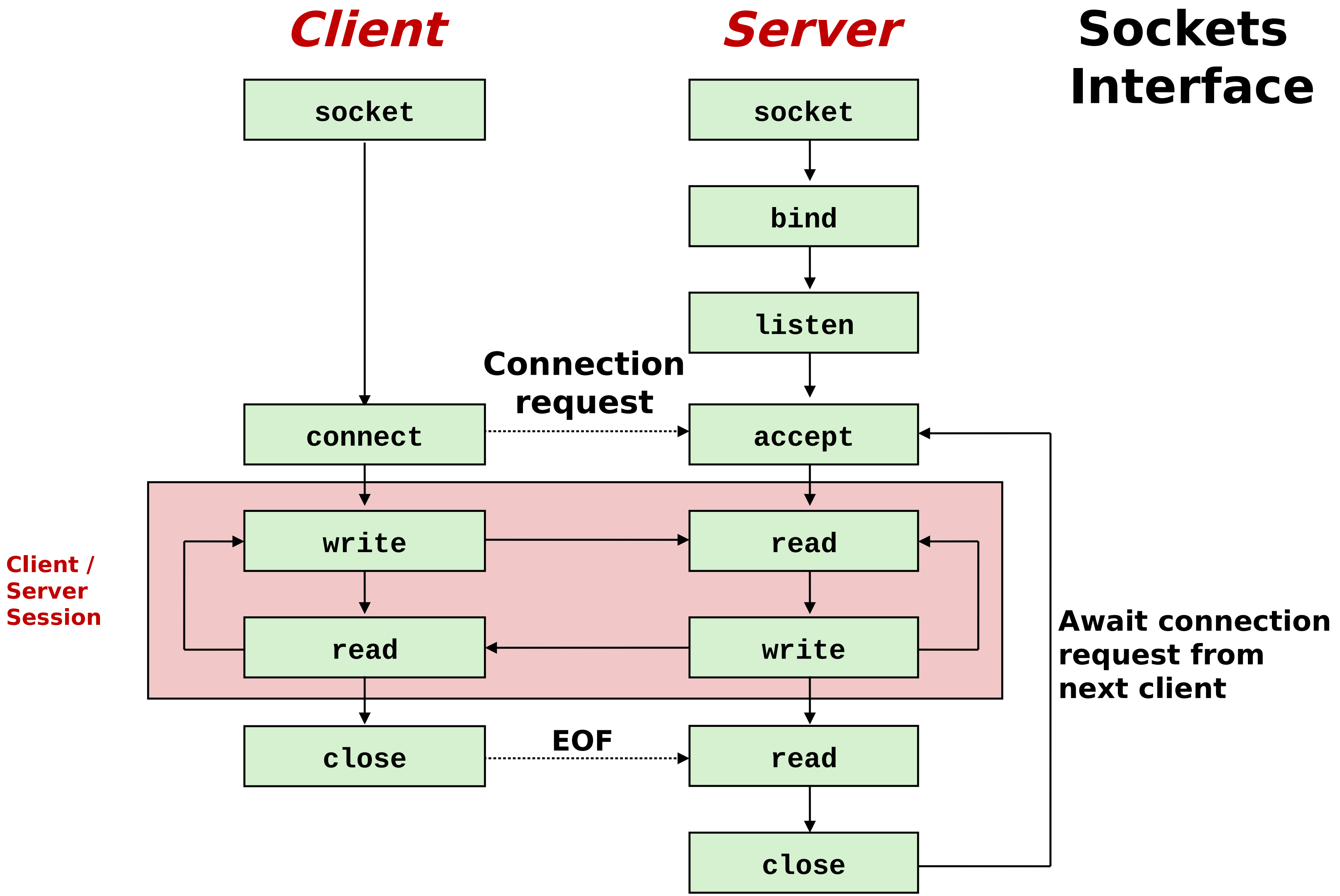
Iterative Server:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind((“127.0.0.1”, 5678))
    server_socket.listen()
    while True:
        connection, connection_address = server_socket.accept()
        with connection:
            message = connection.recv(1024)
            connection.sendall(response)
```

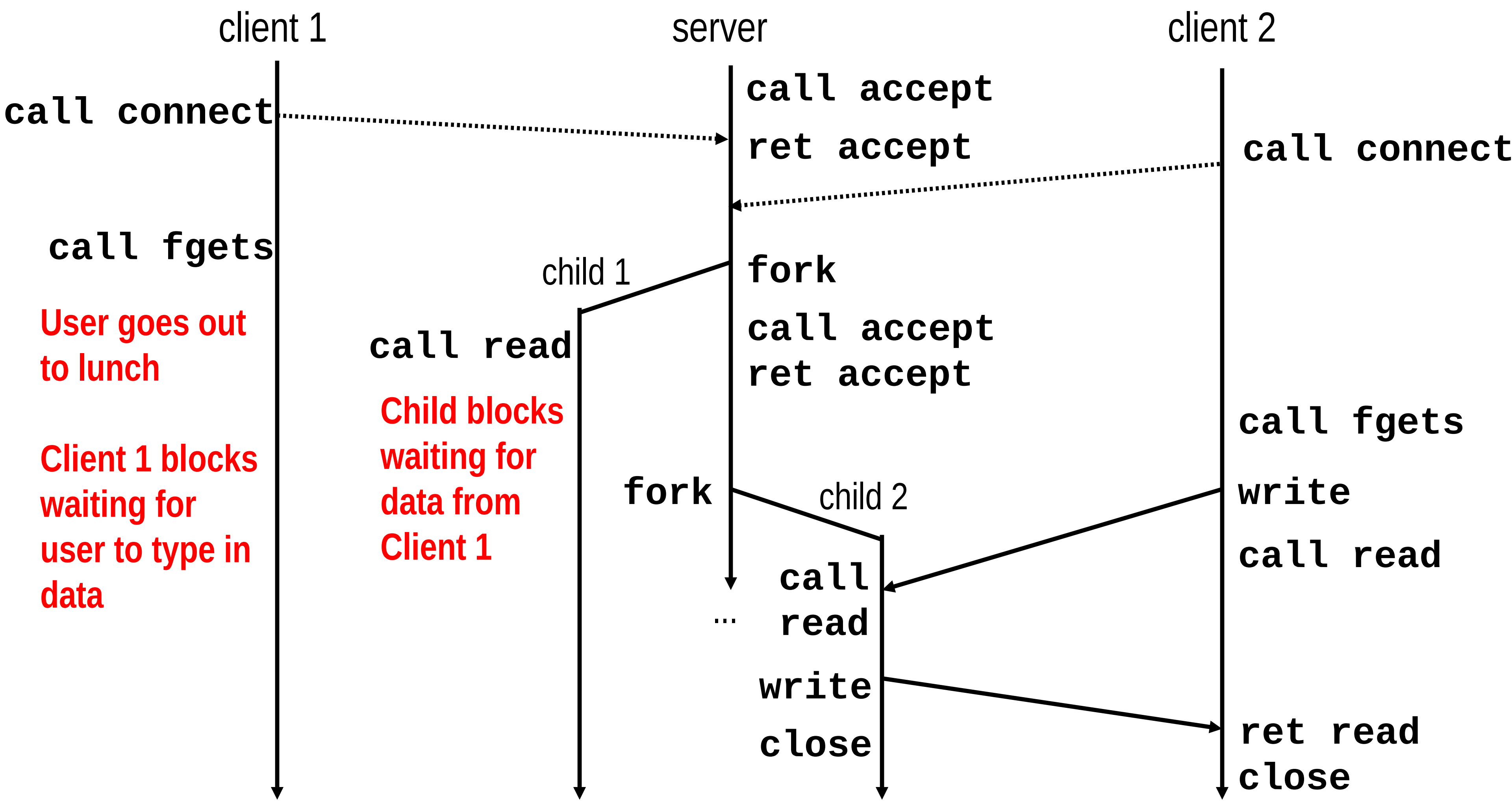
Concurrent Server:

```
from socketserver import ThreadingTCPServer, StreamRequestHandler

class MyHandler(StreamRequestHandler):
    def handle(self) -> None:
        message = self.request.recv(1024)
        self.request.sendall(message)

with ThreadingTCPServer((“127.0.0.1”, 5678), MyHandler) as my_server:
    my_server.serve_forever()
```

Sockets Interface



Concurrent servers



Concurrent servers

- Demonstrated in exercises for this week
- This allows for Non-blocking servers. Very good for performance as all clients can get respodned too without a huge queue.
- We have not really talked about threading yet (and might not till after you’ve finished the networking assignment), so don’t get too concerned with the details just yet.