CS 43: Computer Networks

02: Protocols & Layering

September 5, 2024

SWARTHMORE COLLEGE

Slides adapted from Kurose & Ross, Kevin Webb

Reading Quiz

An example of an application layer protocol is..

A. HTTP: Hyper Text Transfer Protocol

B. Abstraction Protocol

C. Layering Protocol

D. All of the above

What is a protocol?

Goal: get message from sender to receiver

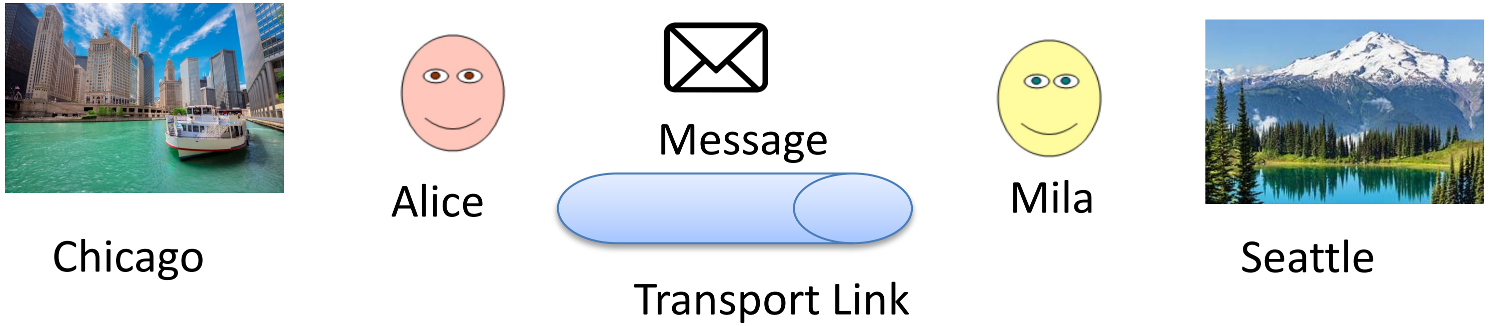
Protocol: message format + transfer procedure

- Expectations of operation
  - first you do x, then I do y, then you do z, ...

- Multiparty! so no central control
  - sender and receiver are separate processes

A “Simple” analogous task: Postal Mail

Alice moves to Chicago and Mila to Seattle for summer internships. Alice would like to send Mila a birthday card. Think of this as filling two different pieces of information (1. the birthday card, 2. the mailing envelope).



Chicago

Alice

Message

Transport Link

Mila

Seattle

A “Simple” analogous task: Postal Mail

Alice would like to send Mila a birthday card.

1. **Construct the message and header. Have the header and message portions changed from the previous scenario?**
2. **List the message format and transfer procedure of the “mail sending protocol” that Alice uses.**
   - Who chooses the drop-off point?
   - Is this the only protocol in use?
3. **Message transportation and delivery**
   - Whose job is it to:
     - choose the carrier?
     - plan the route?
     - deliver the message?
     - ensure the message is not lost?

A “Simple” analogous task: Postal Mail

- **<u>Message transportation and delivery</u>**
- Who’s job is it to:

1. provide the sender and receiver addresses?
2. choose the carrier?

(1, 2): Alice decides as the “end host”

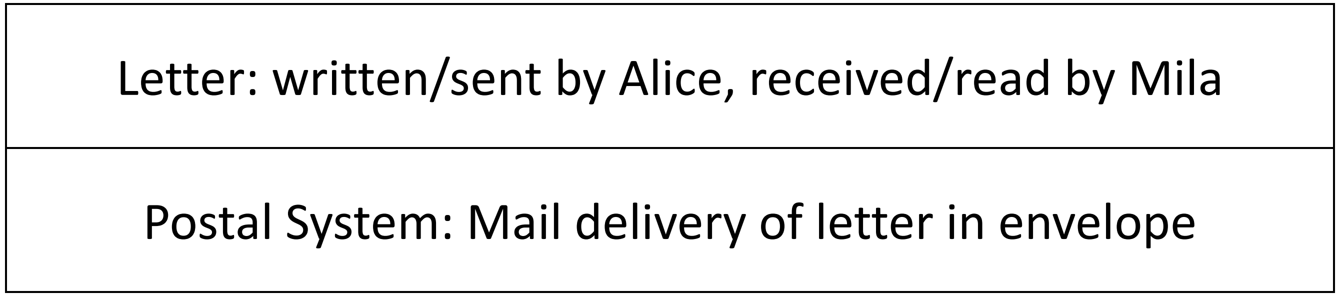
3. plan the route?
4. transport vehicles?

(3, 4): Postal Department decides as the service that provides message transfer

5. ensure the message is not lost? (reliability)

Reliability? Open question – stay tuned!

Layering: Separation of Functions



Letter: written/sent by Alice, received/read by Mila

Postal System: Mail delivery of letter in envelope

- Alice and Mila
  - Don’t have to know about delivery
  - However, aid postal system by providing addresses
- Postal System
  - Only has to know addresses and how to deliver
  - Doesn’t care about “data”: Alice, Mila, letter

Abstraction!

- Hides the complex details of a process
- Use abstract representation of relevant properties make reasoning simpler
- Ex: Alice and Mila knowledge of postal system:
  - Letters with addresses go in, come out other side

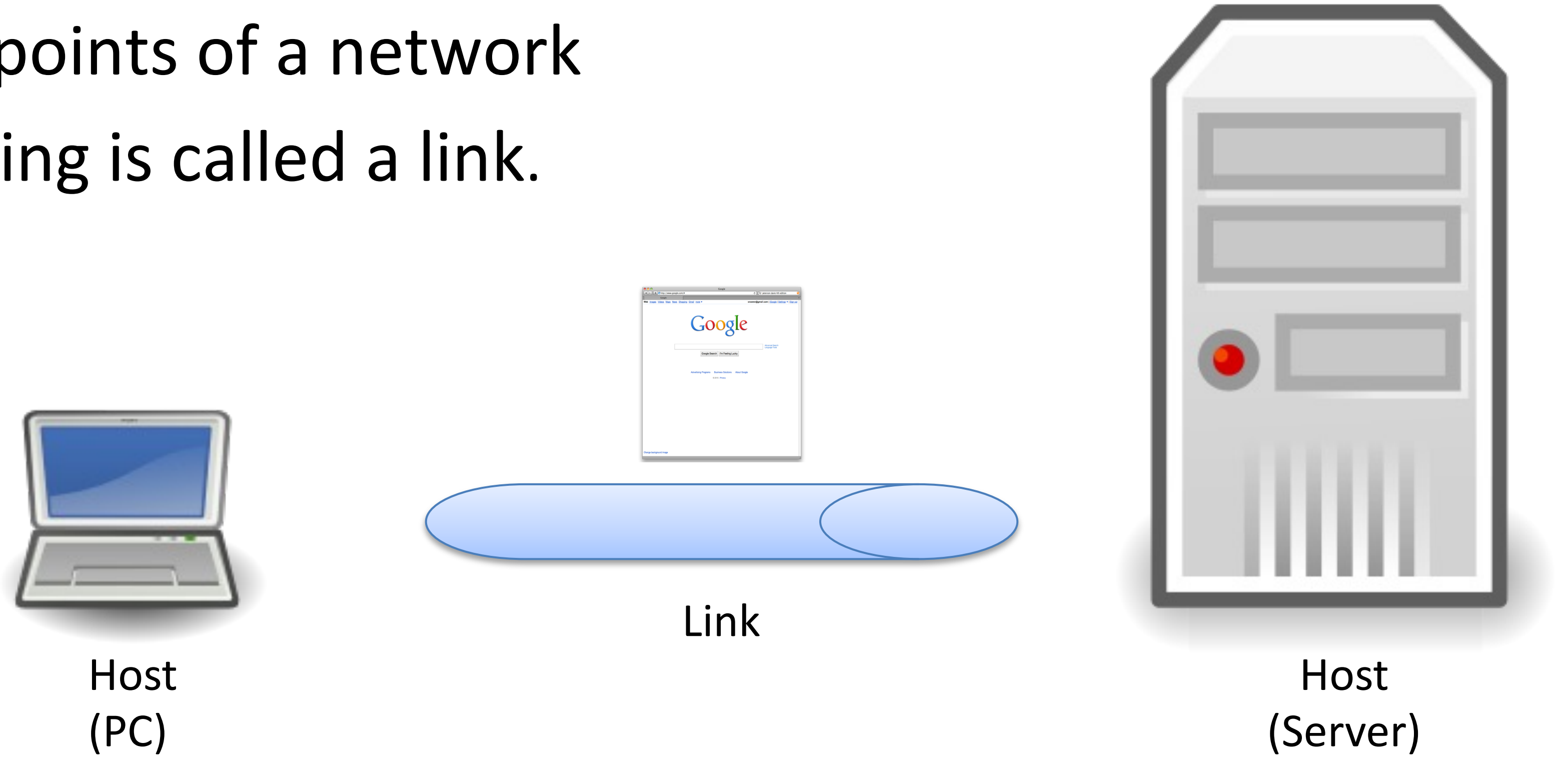
A “Simple” analogous task: Postal Mail

- Many more considerations..
  - Who decides the the sender and receiver addresses? Does someone maintain a mapping peoples’ names to addresses?
  - Can Mila always be guaranteed of this delivery date? What factors influence delivery ?
  - What if the mail gets lost – who’s responsibility is it? Alice, Mila or someone else?
  - What about security? privacy?

A “Simple” Task

Send information from one computer to another

- hosts: endpoints of a network
- The plumbing is called a link.

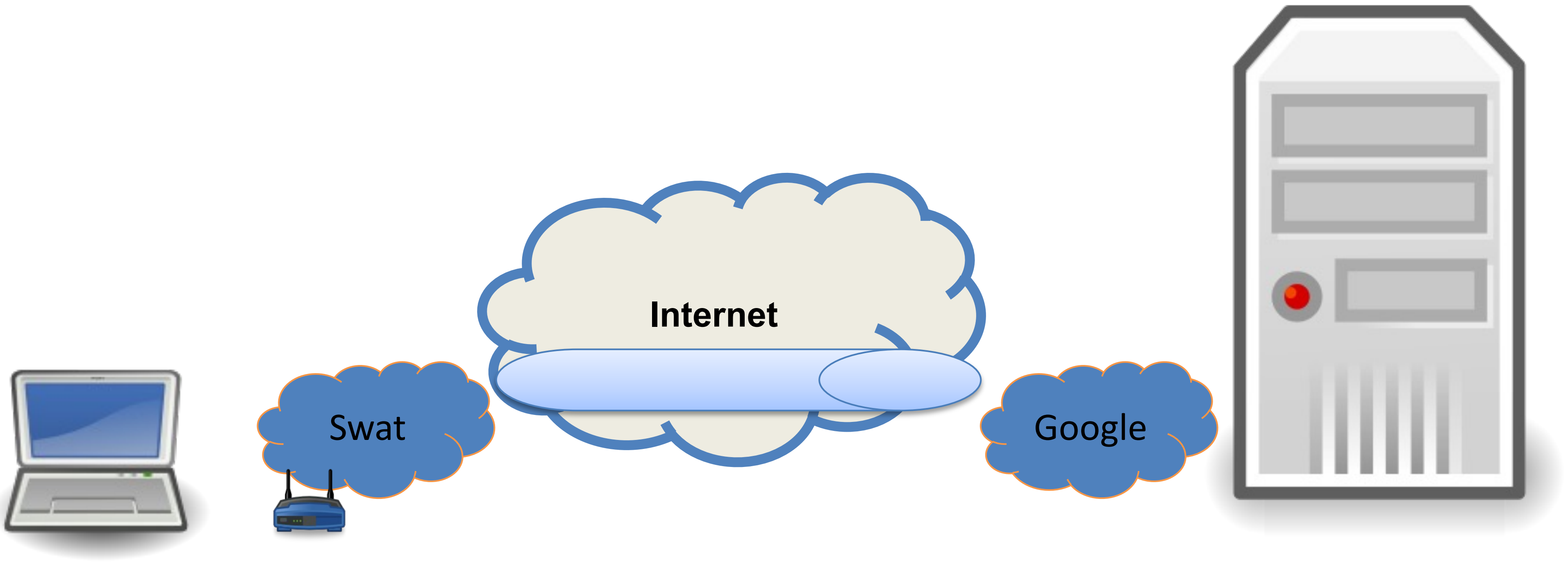


Host
(PC)

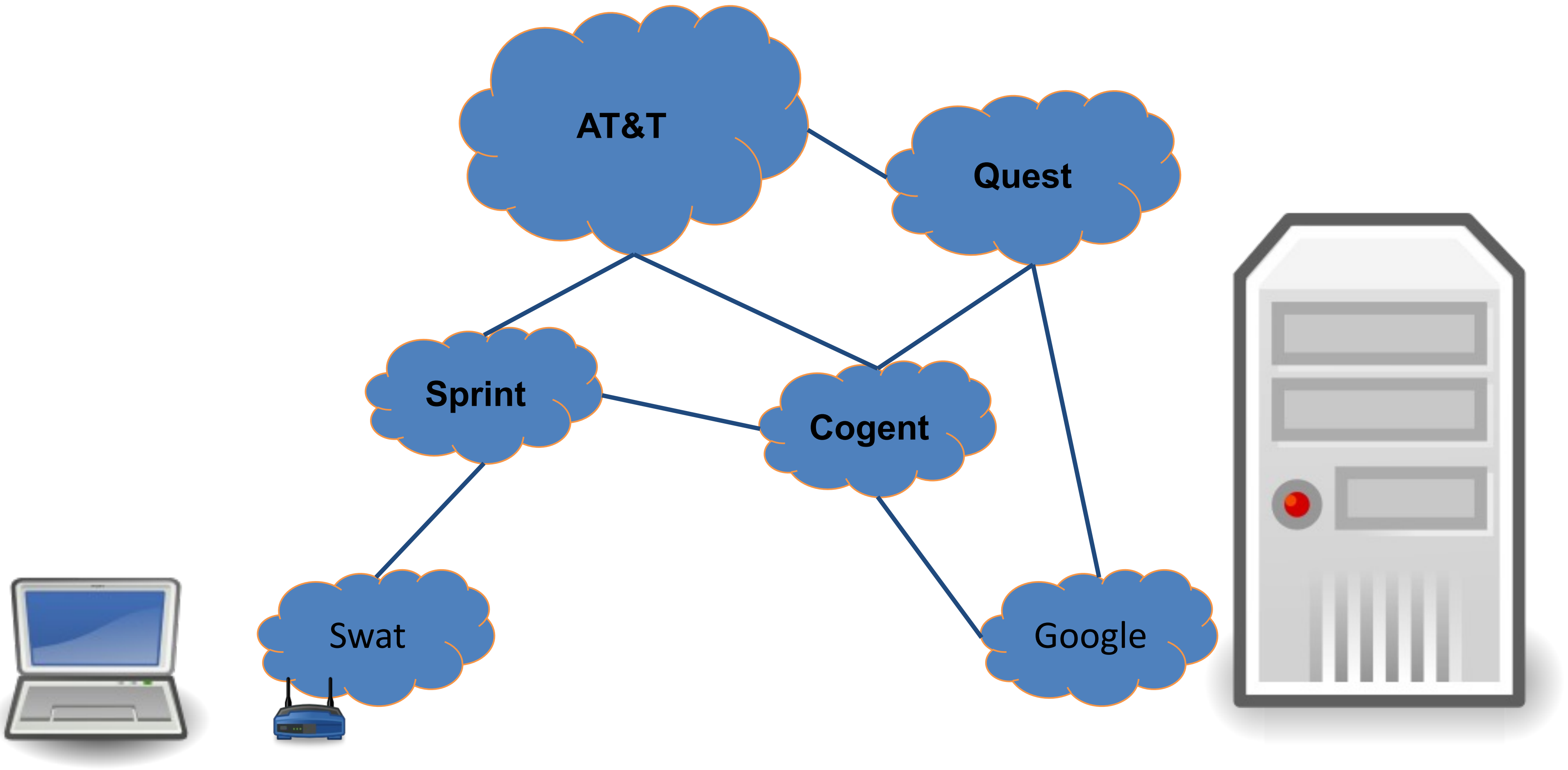
Link

Host
(Server)

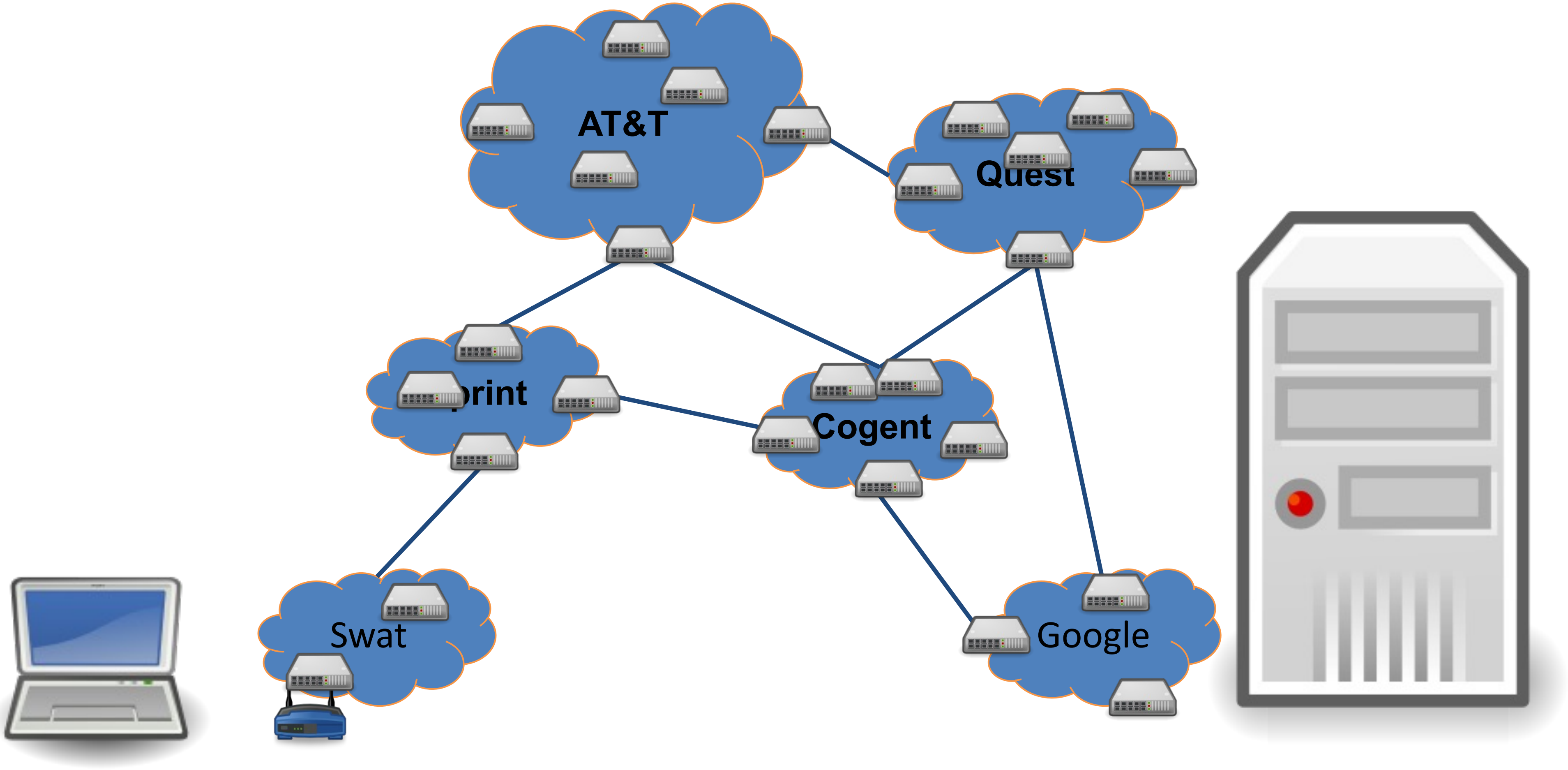
Not Really So Simple…



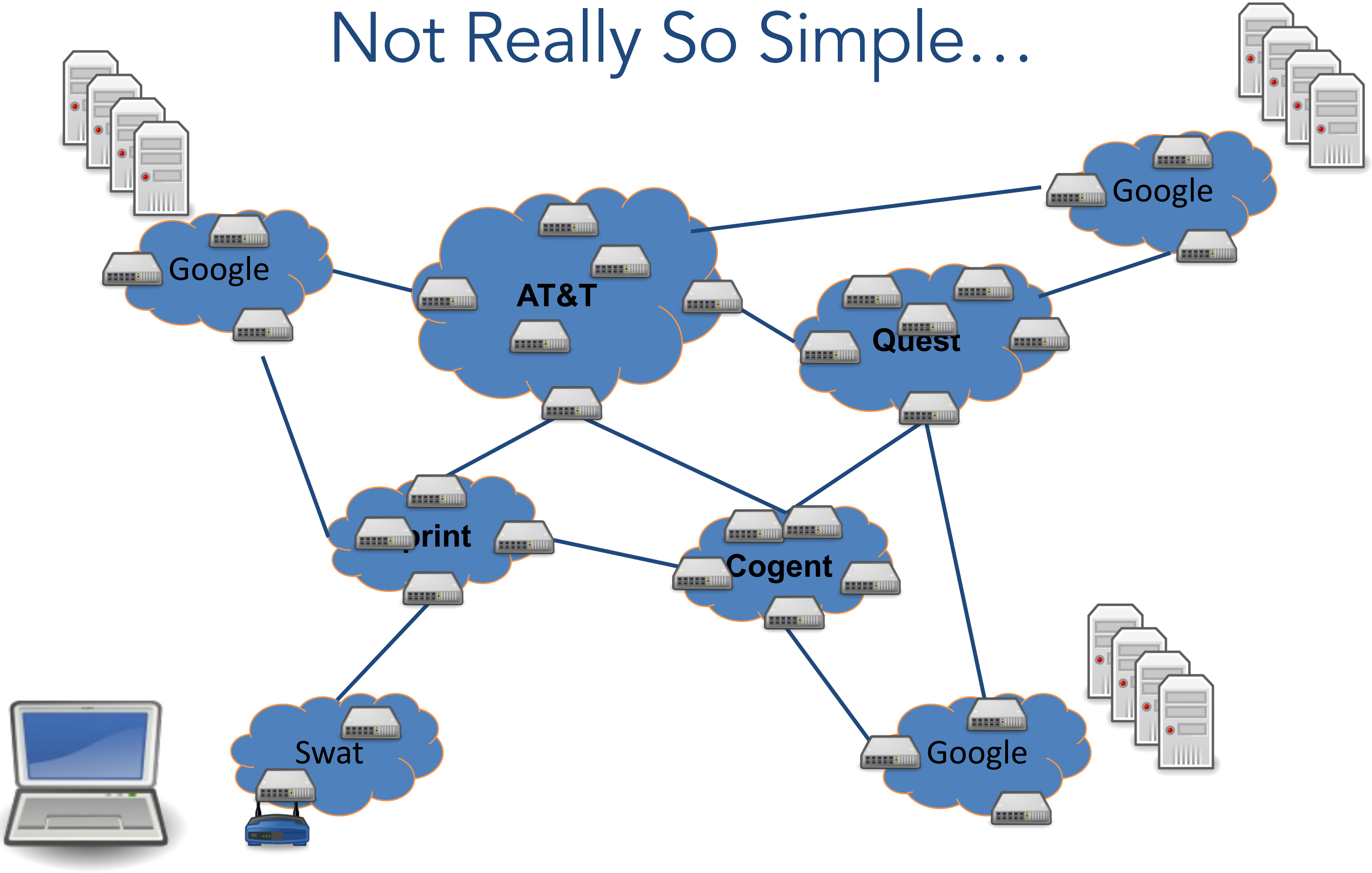
Not Really So Simple…



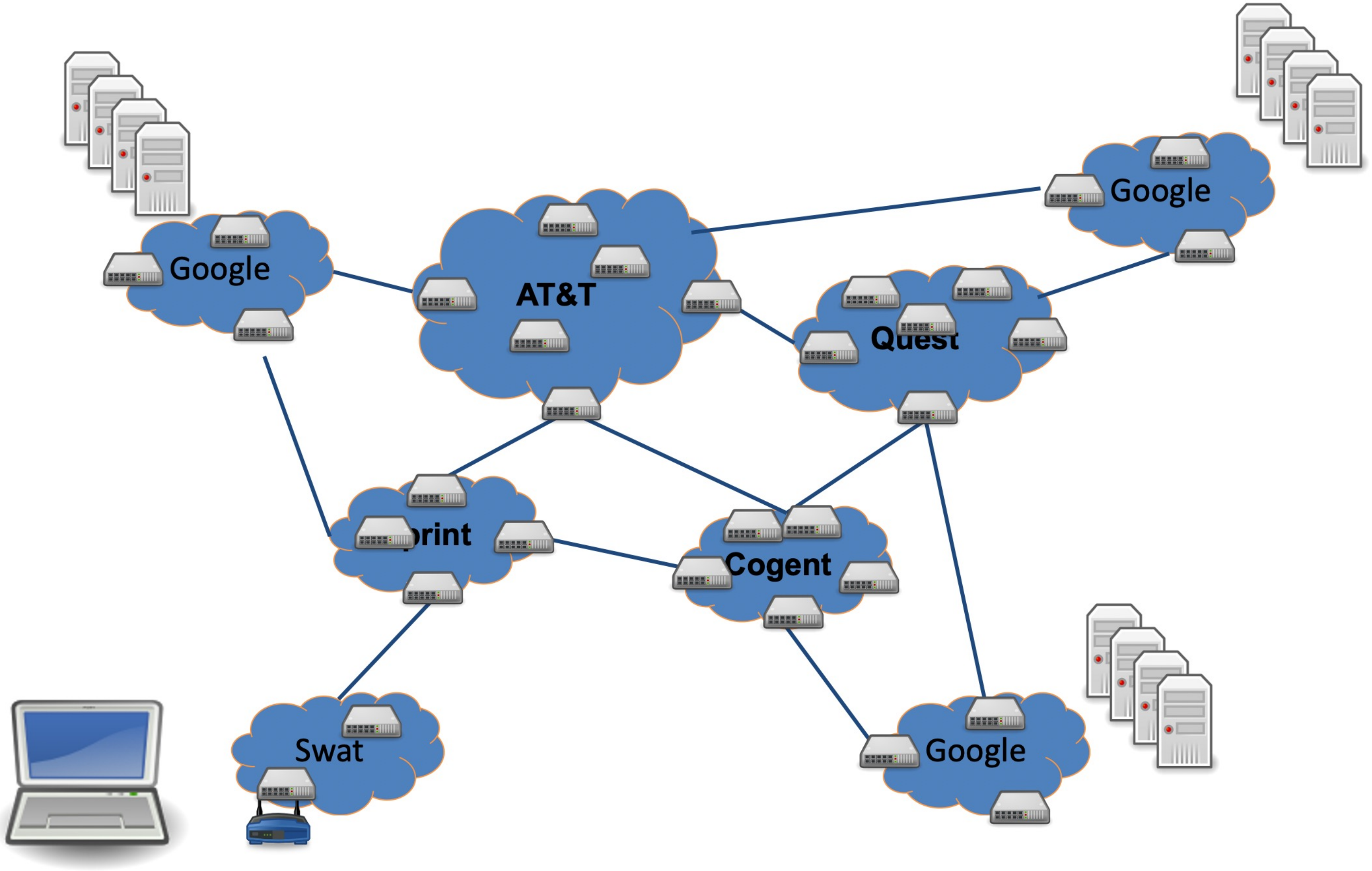
Not Really So Simple…



Not Really So Simple…

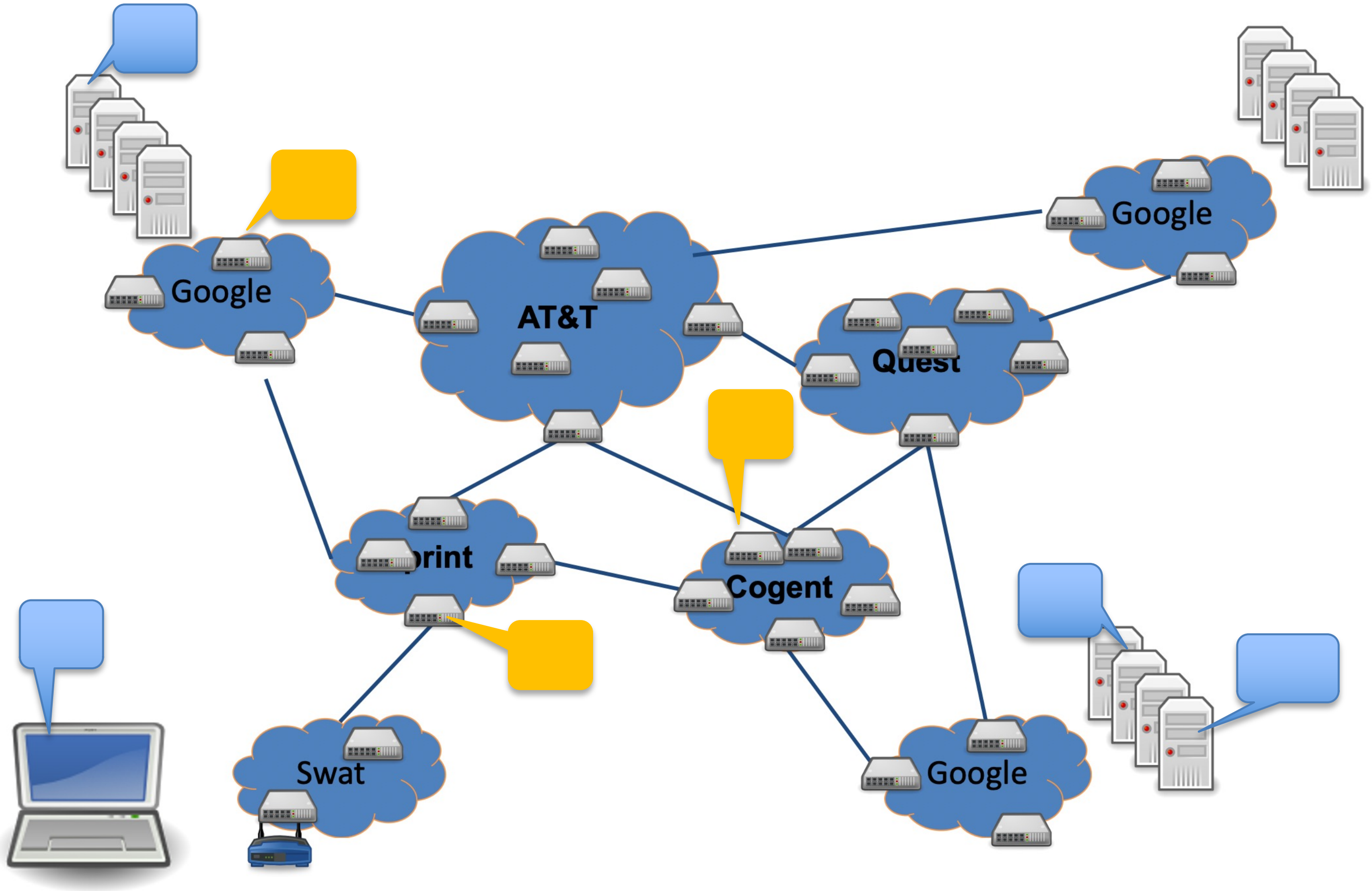


Are we done?



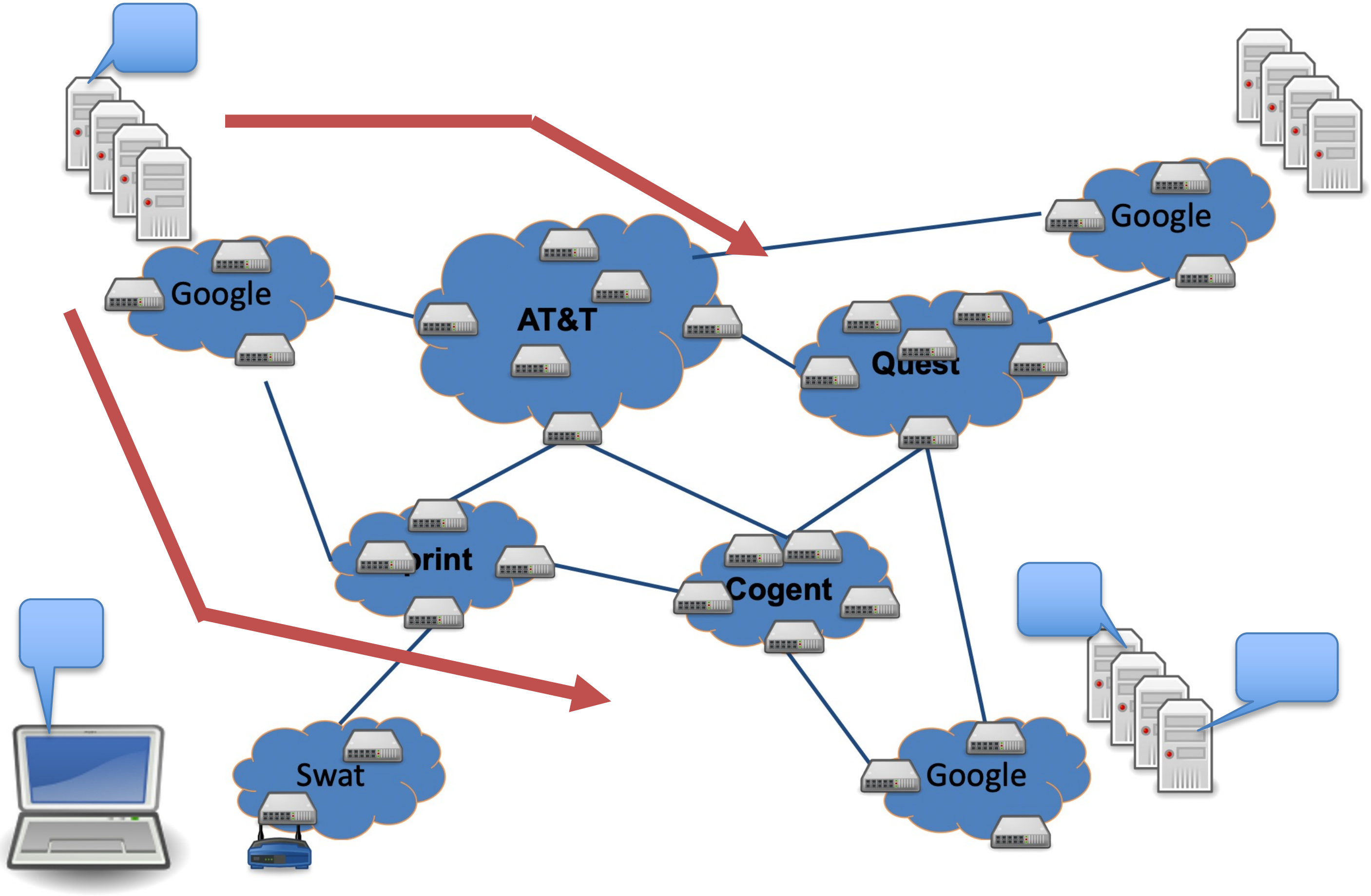
We only need… naming and addressing

Agreeing on how to describe/express a host, application, network, etc.



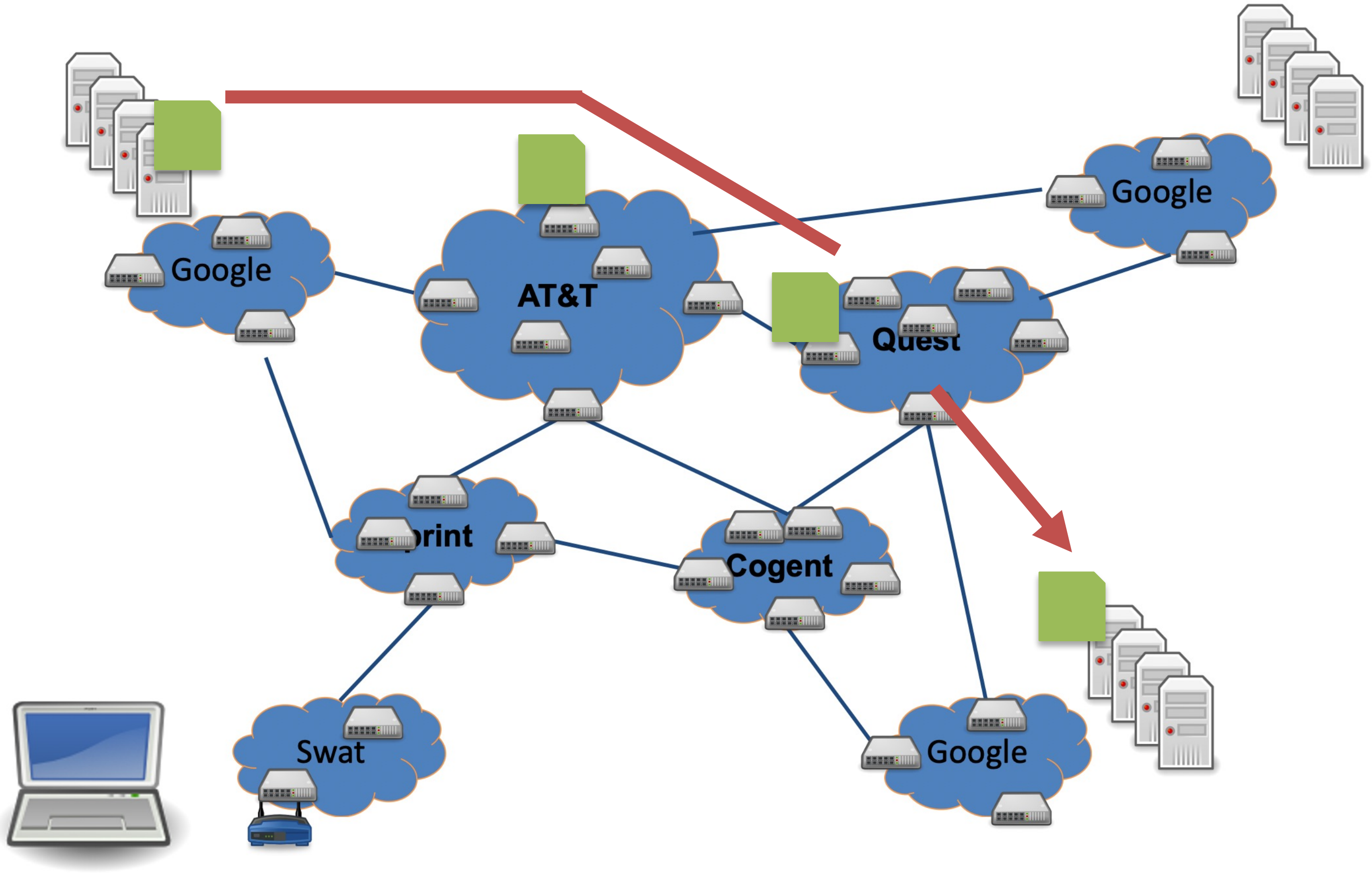
We only need… moving data to the destination

Routing: deciding how to get it there



We only need… moving data to the destination

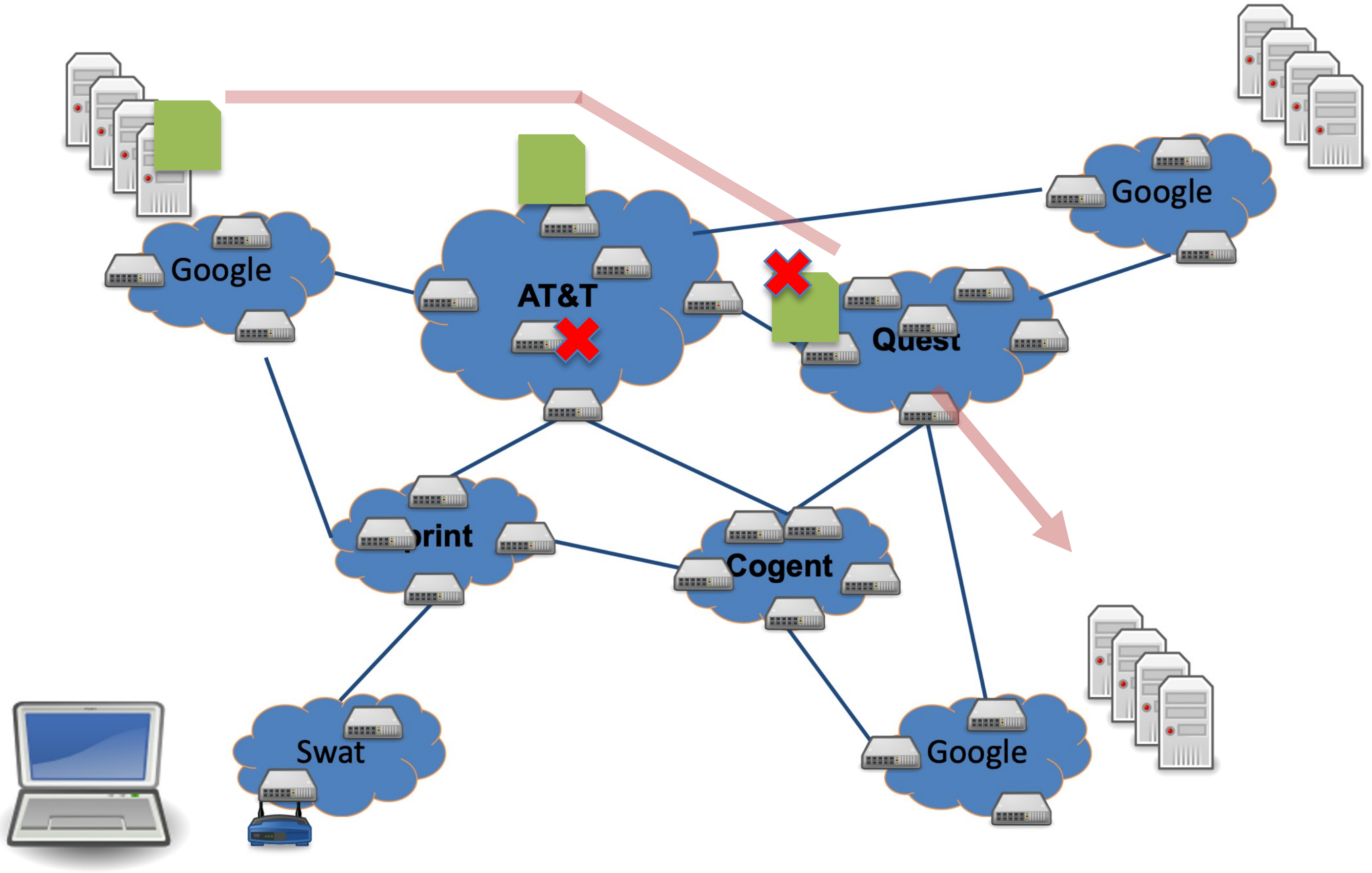
Forwarding: copying data across devices/links



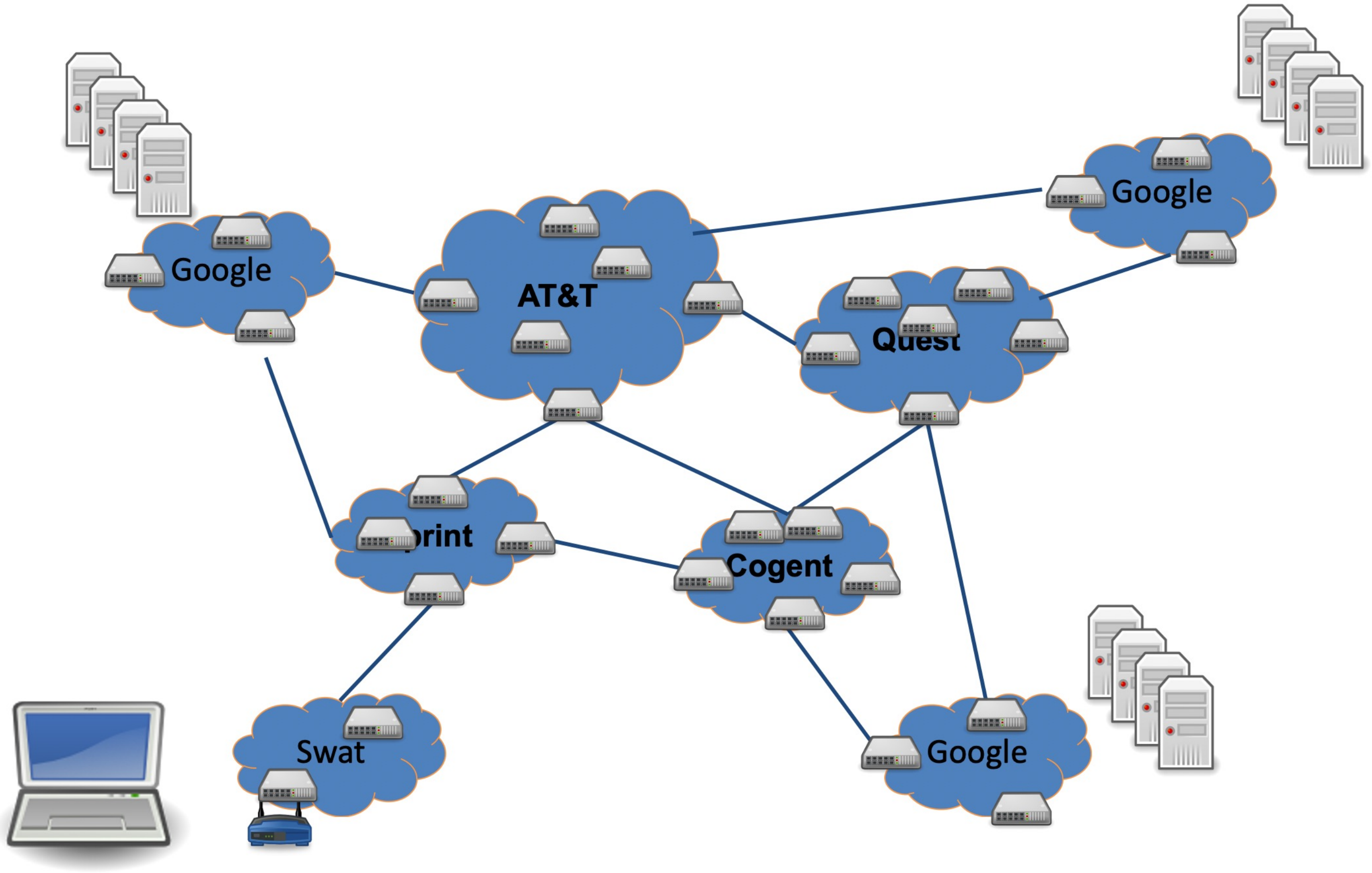
We only need… reliability and fault tolerance

how can we …guarantee that the data arrives?

…handle link or device failures?



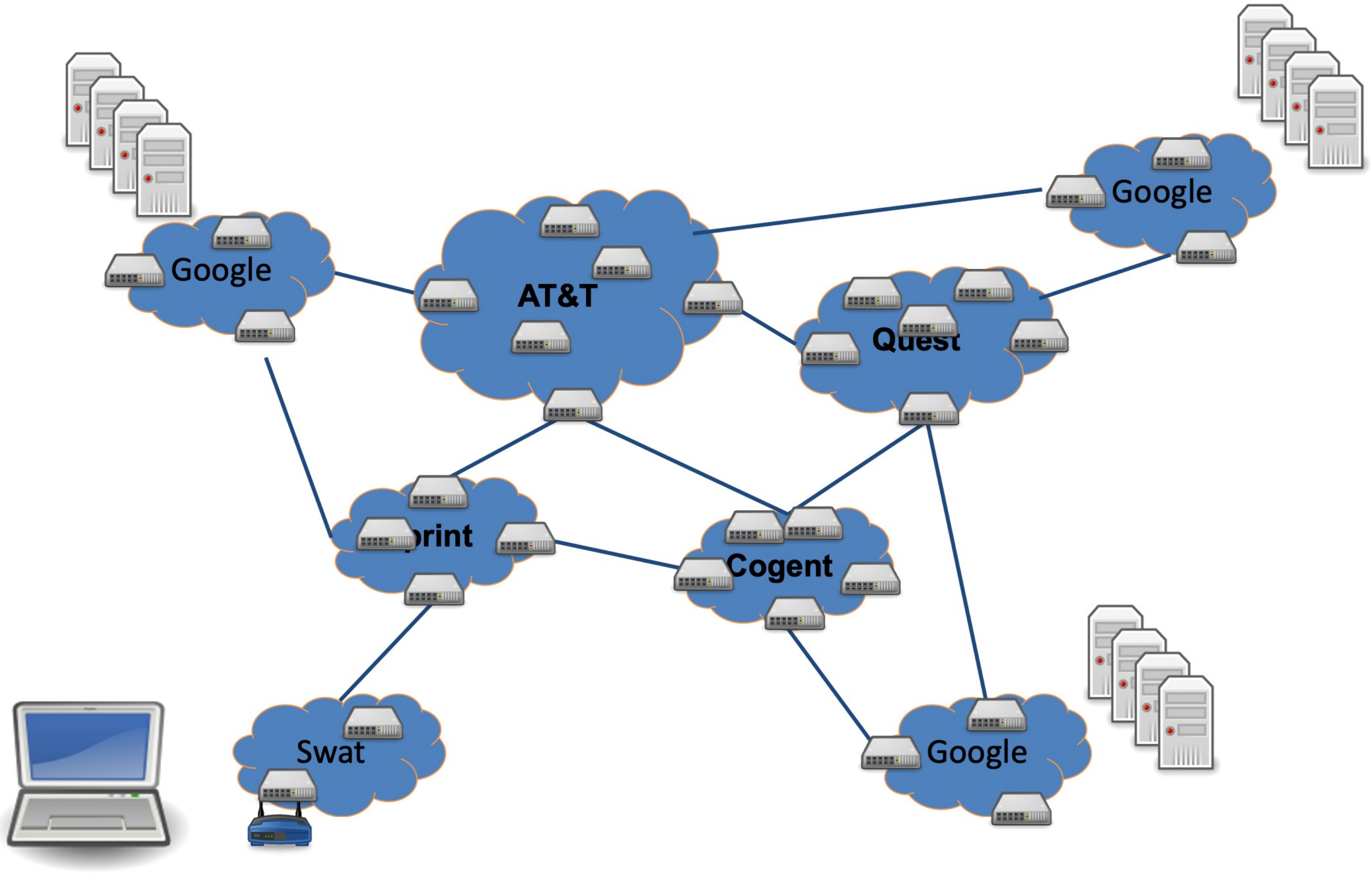
We only need… security and privacy



We only need… to manage complexity and scale up

Layering abstraction: divide responsibility

Protocols: standardize behavior for interoperability



We only need…

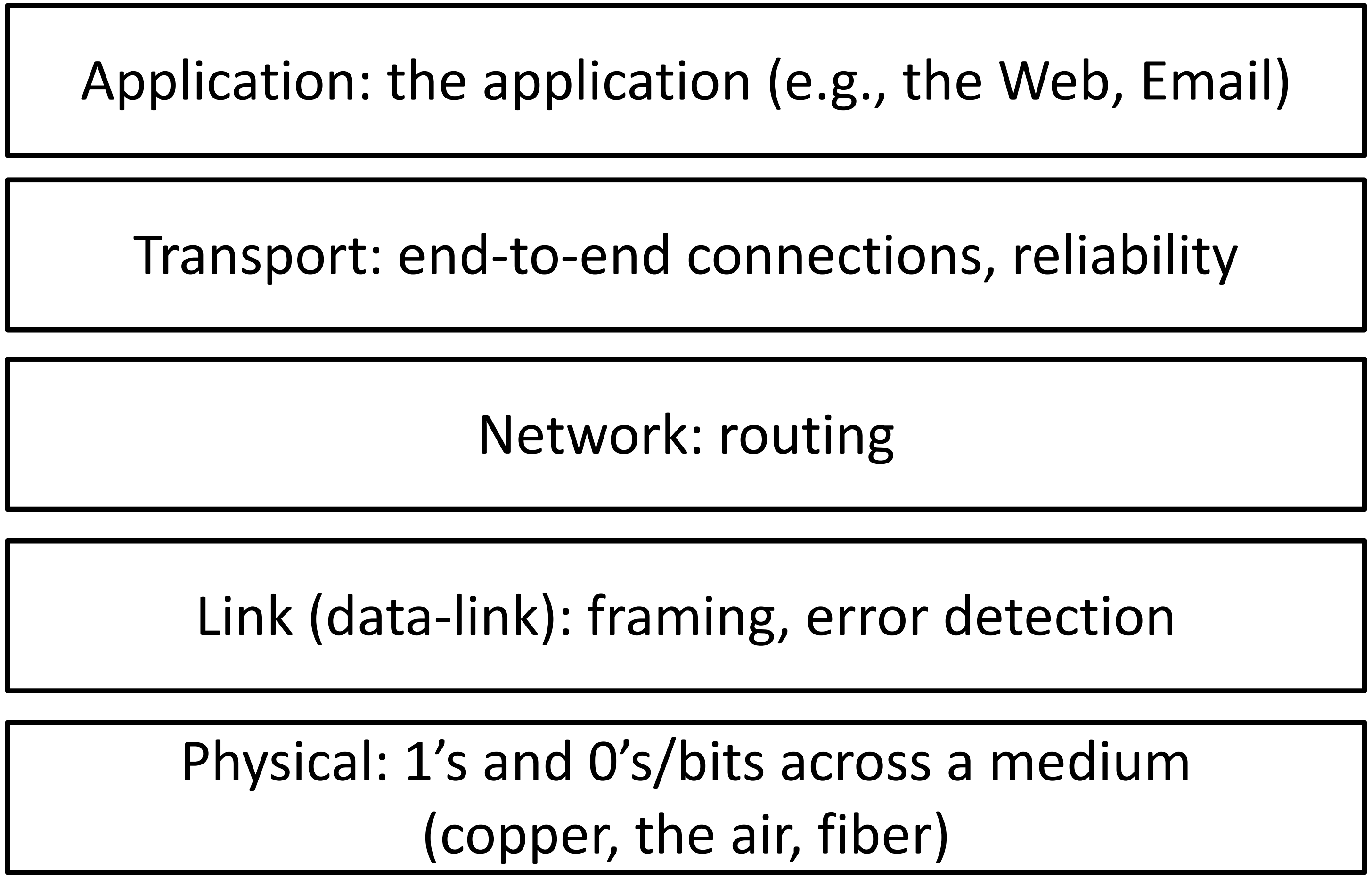
- Manage complexity and scale up
- Naming and addressing
- Moving data to the destination
- Reliability and fault tolerance
- Resource allocation, Security, Privacy..

We only need…

- Manage complexity and scale up
- Naming and addressing
- Moving data to the destination
- Reliability and fault tolerance
- Resource allocation, Security, Privacy..

(Lots of others too.)

Five-Layer Internet Model



Application: the application (e.g., the Web, Email)

Transport: end-to-end connections, reliability

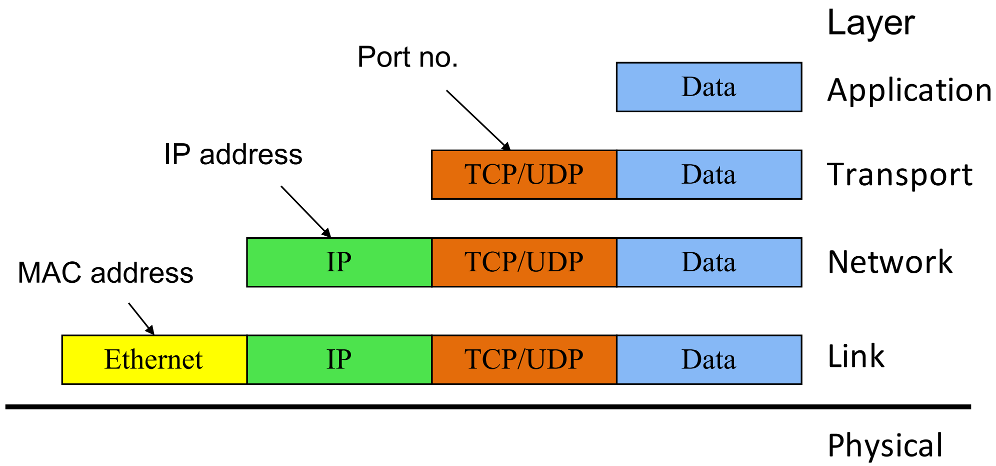
Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium
(copper, the air, fiber)

Application Layer
(HTTP, FTP, SMTP, Tiktok)

- Does whatever an application does!



Layer

Port no.

Data — Application

IP address

TCP/UDP | Data — Transport

MAC address

IP | TCP/UDP | Data — Network

Ethernet | IP | TCP/UDP | Data — Link

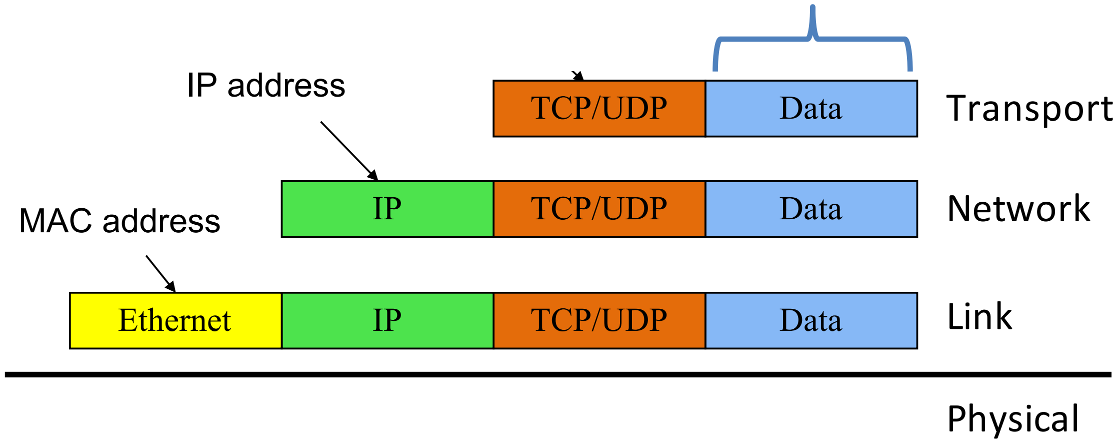
Physical

Transport Layer (TCP, UDP)

- Provides
  - Ordering
  - Error checking
  - Delivery guarantee
  - Congestion control
  - Flow control

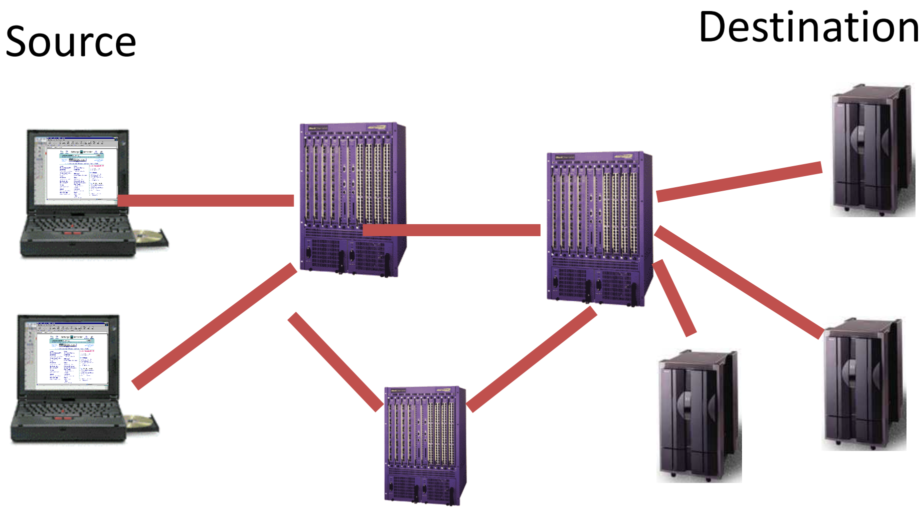
- Or doesn't!

Application Layer Data becomes the payload for the transport layer

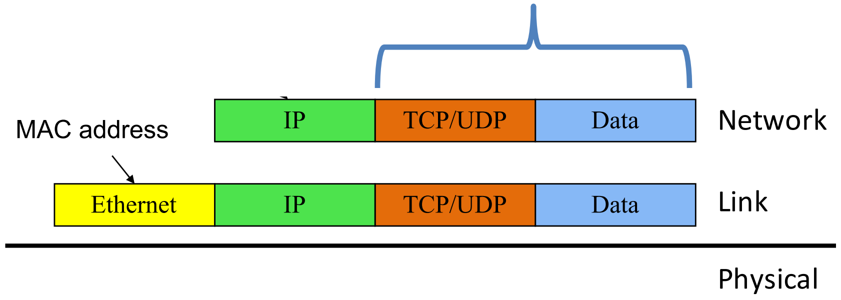


Network Layer (IP)

- **Routers**: choose paths through network



Transport layer data + header becomes payload for the network layer



MAC address

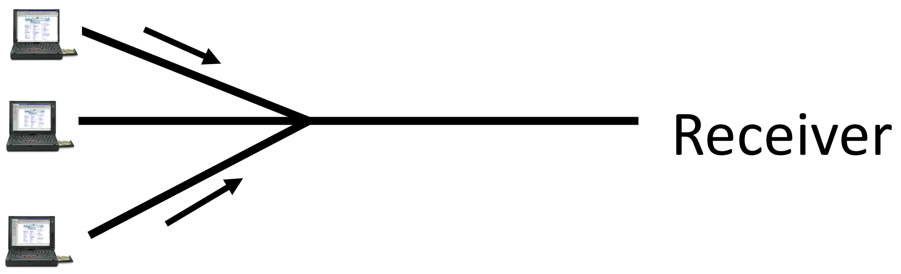
Network: IP | TCP/UDP | Data

Link: Ethernet | IP | TCP/UDP | Data

Physical

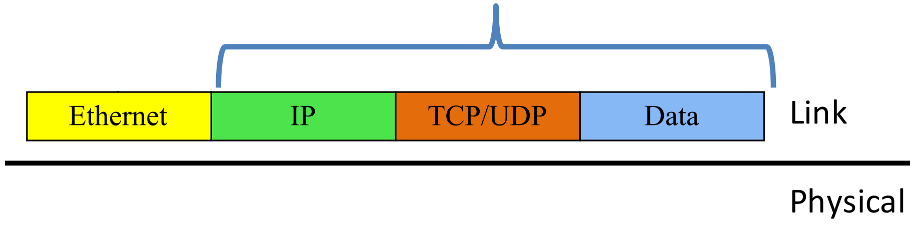
Link Layer (Ethernet, WiFi, Cable)

- Who’s turn is it to send right now?
- Break message into frames
- Media access: can it send the frame now?

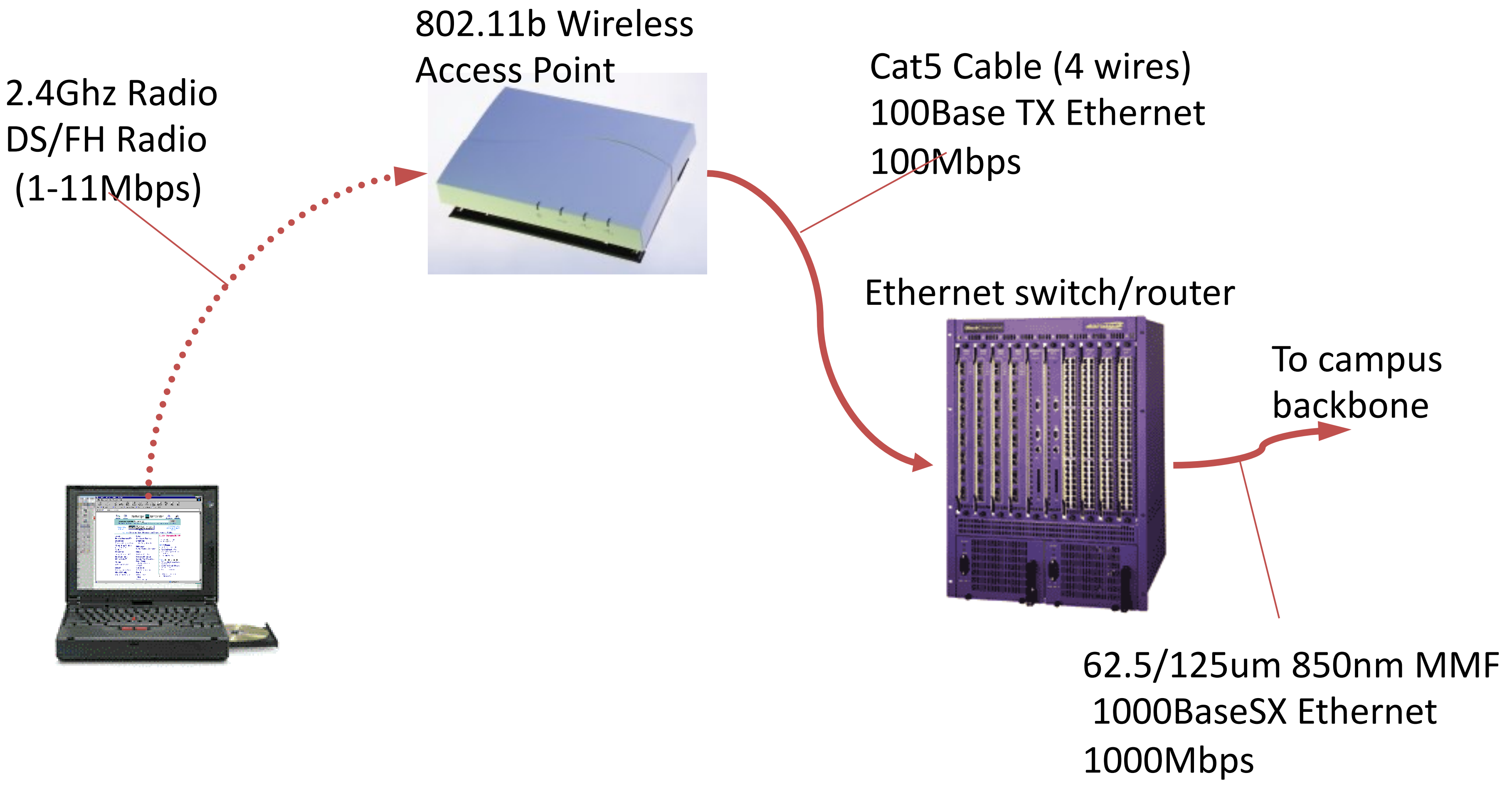


- Send frame, handle “collisions”

Network layer data + header becomes payload for the link layer



Physical layer – move actual bits!
(Cat 5, Coax, Air, Fiber Optics)



802.11b Wireless Access Point

2.4Ghz Radio
DS/FH Radio
(1-11Mbps)

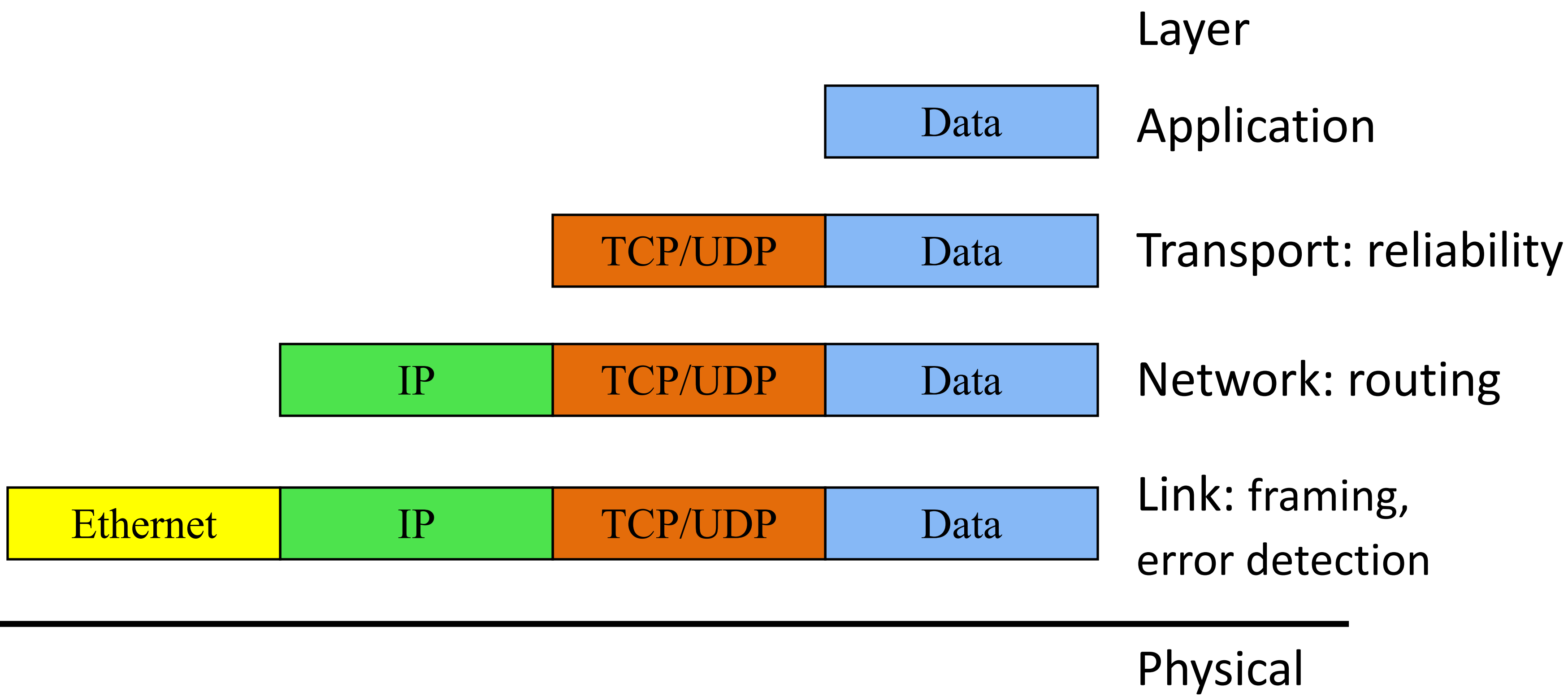
Cat5 Cable (4 wires)
100Base TX Ethernet
100Mbps

Ethernet switch/router

To campus backbone

62.5/125um 850nm MMF
1000BaseSX Ethernet
1000Mbps

Layering and encapsulation



Layer

Data — Application

TCP/UDP | Data — Transport: reliability

IP | TCP/UDP | Data — Network: routing

Ethernet | IP | TCP/UDP | Data — Link: framing, error detection

Physical

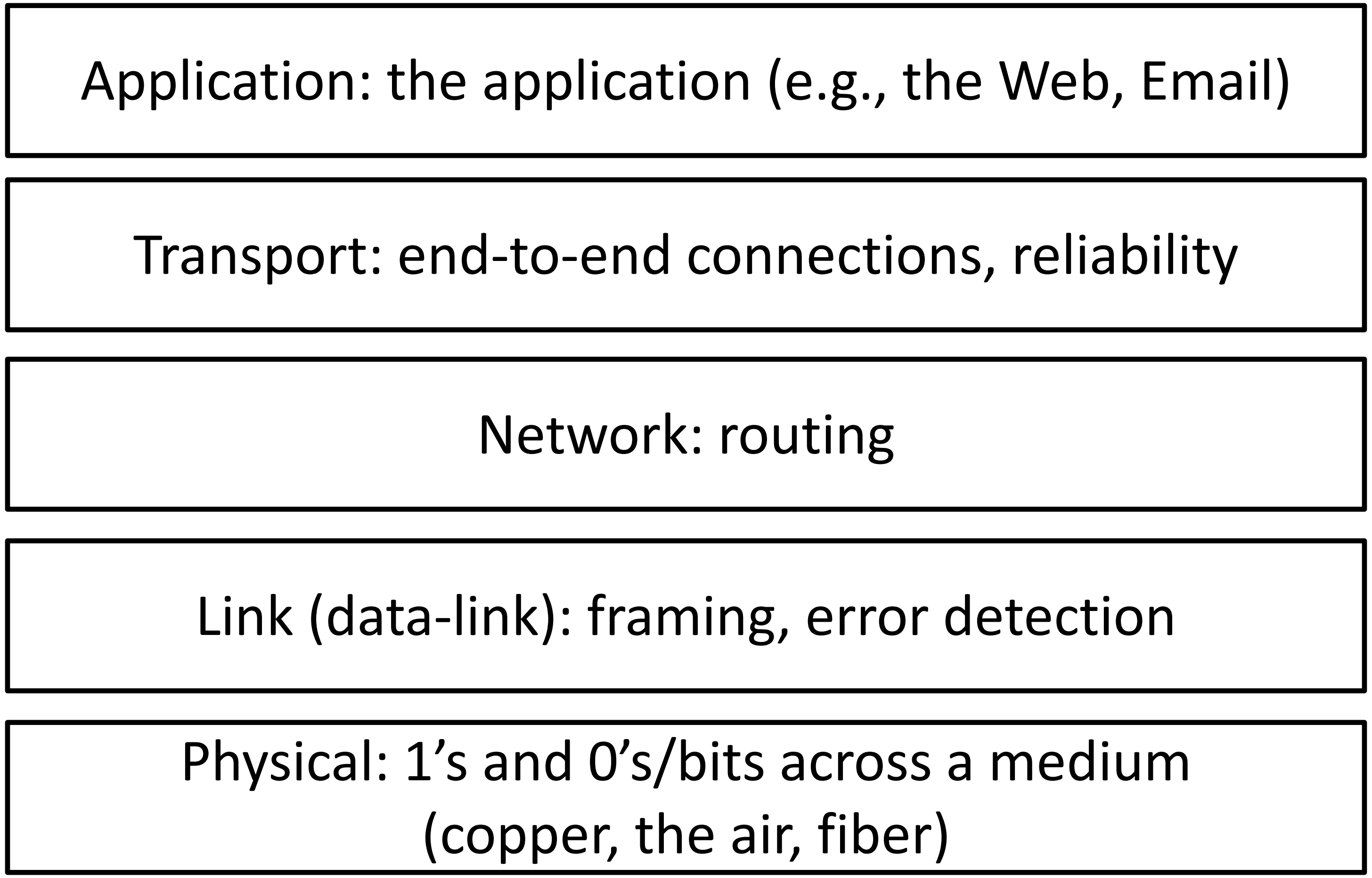
Layering: Separation of Functions

- explicit structure allows identification, relationship of complex system's pieces
  - layered reference model for discussion
  - reusable component design
- modularization eases maintenance
  - change of implementation of layer's service transparent to rest of system,
  - e.g., change in postal route doesn't effect delivery of lette

Abstraction!

- Hides the complex details of a process
- Use abstract representation of relevant properties make reasoning simpler
- Ex: Your knowledge of postal system:
  - Letters with addresses go in, come out other side

Five-Layer Internet Model



Application: the application (e.g., the Web, Email)

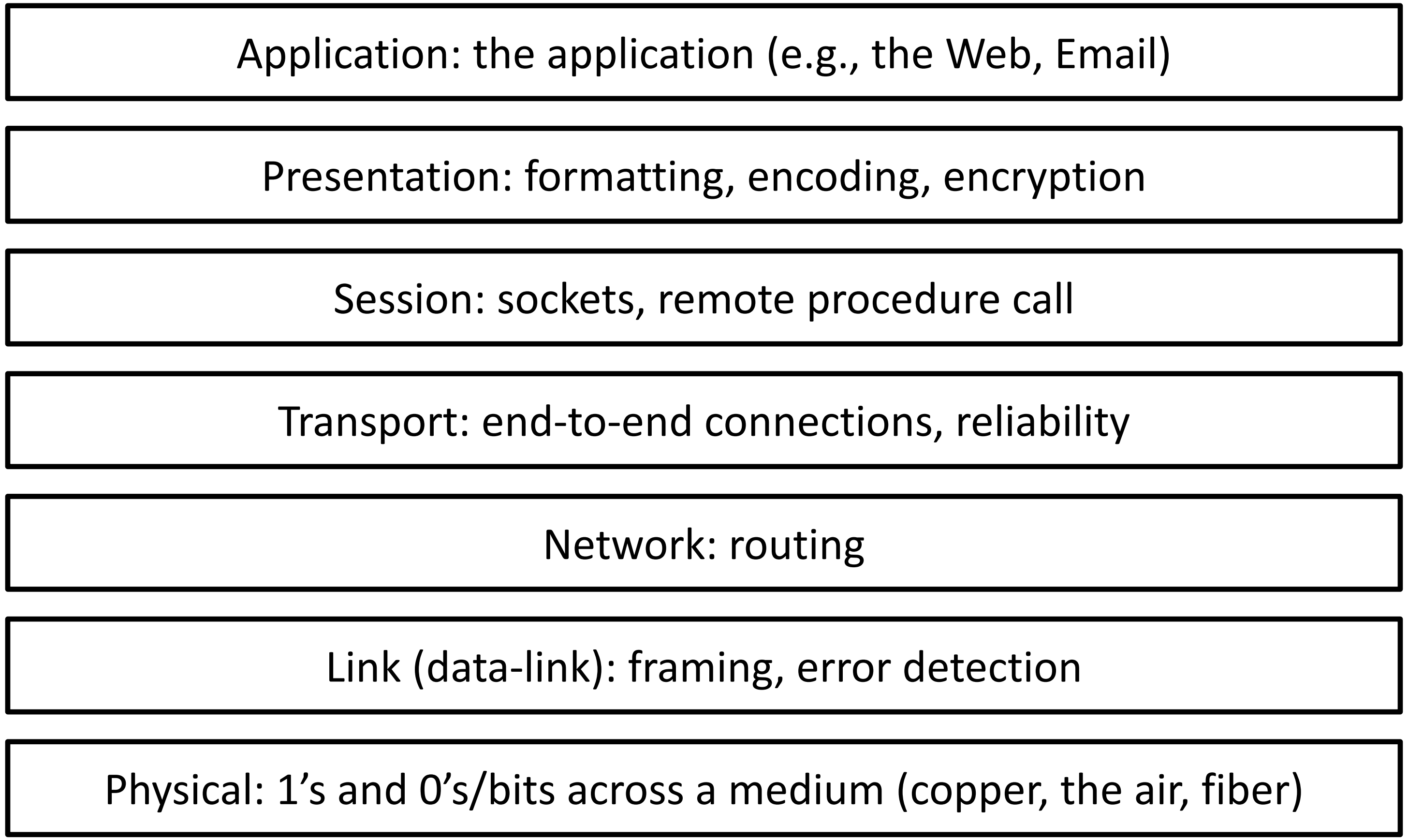
Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium (copper, the air, fiber)

OSI Seven-Layer Model



Application: the application (e.g., the Web, Email)

Presentation: formatting, encoding, encryption

Session: sockets, remote procedure call

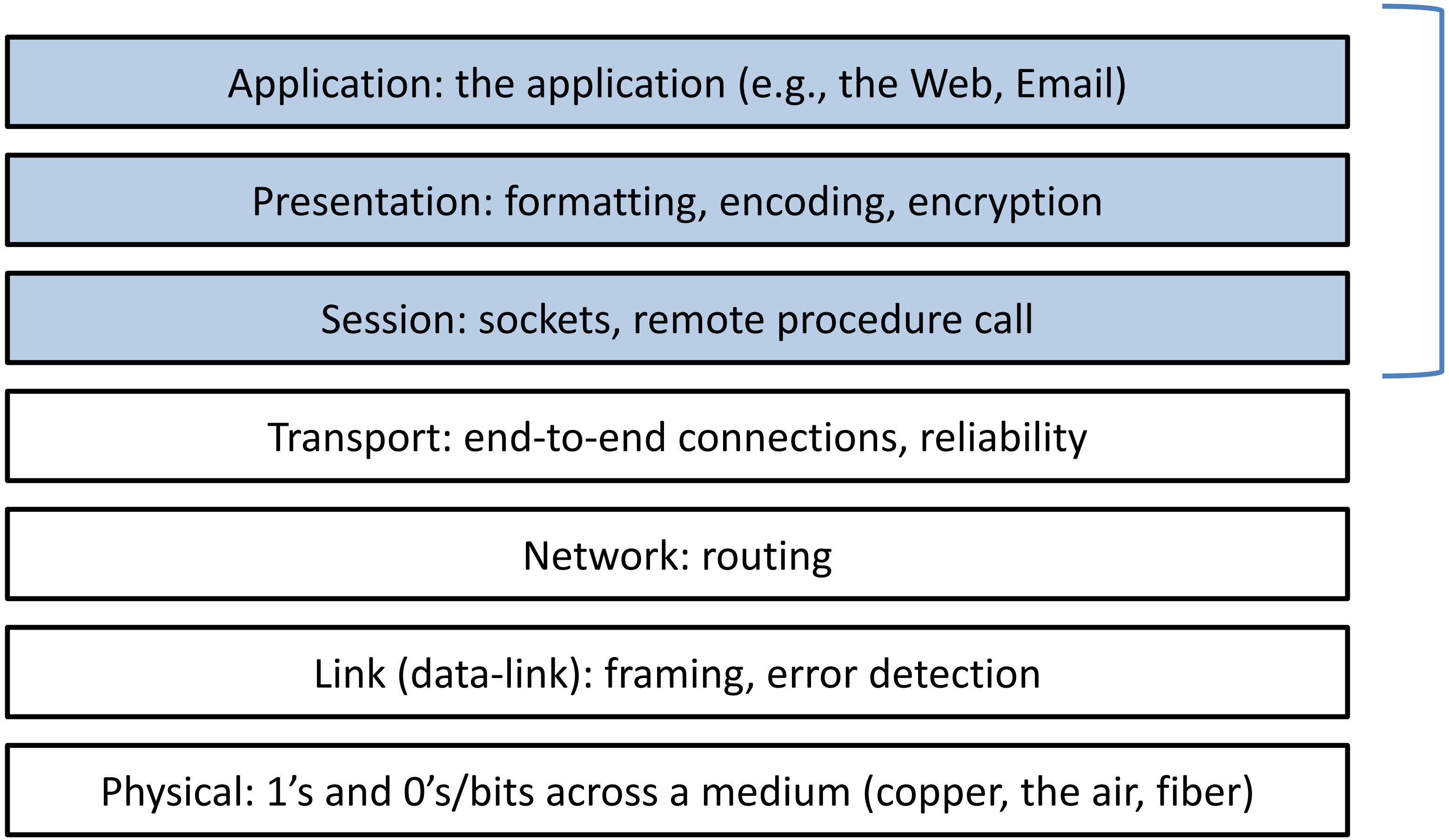
Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium (copper, the air, fiber)

OSI Seven-Layer Model



Application: the application (e.g., the Web, Email)

Presentation: formatting, encoding, encryption

Session: sockets, remote procedure call

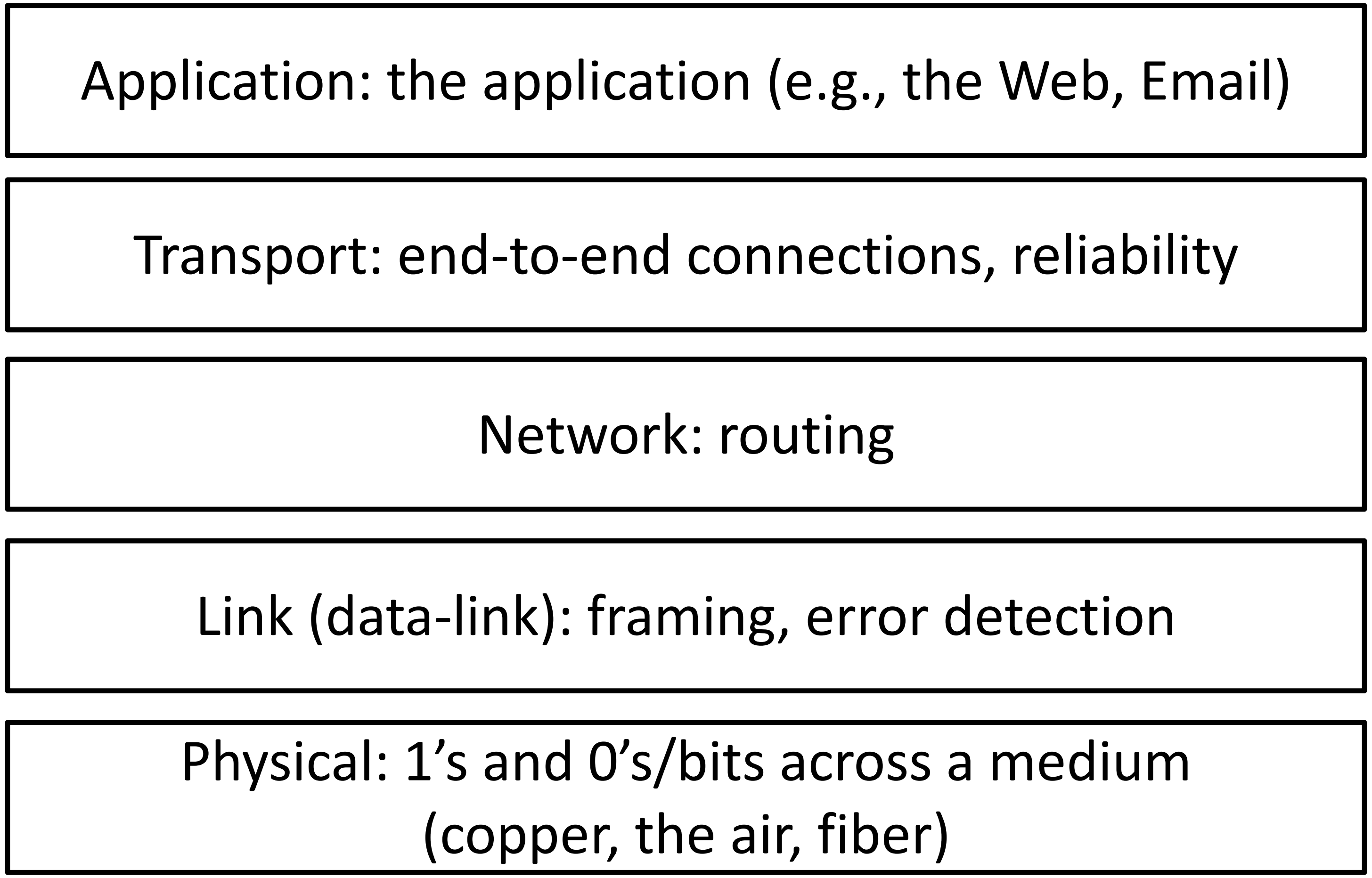
Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium (copper, the air, fiber)

Five-Layer Internet Model



Application: the application (e.g., the Web, Email)

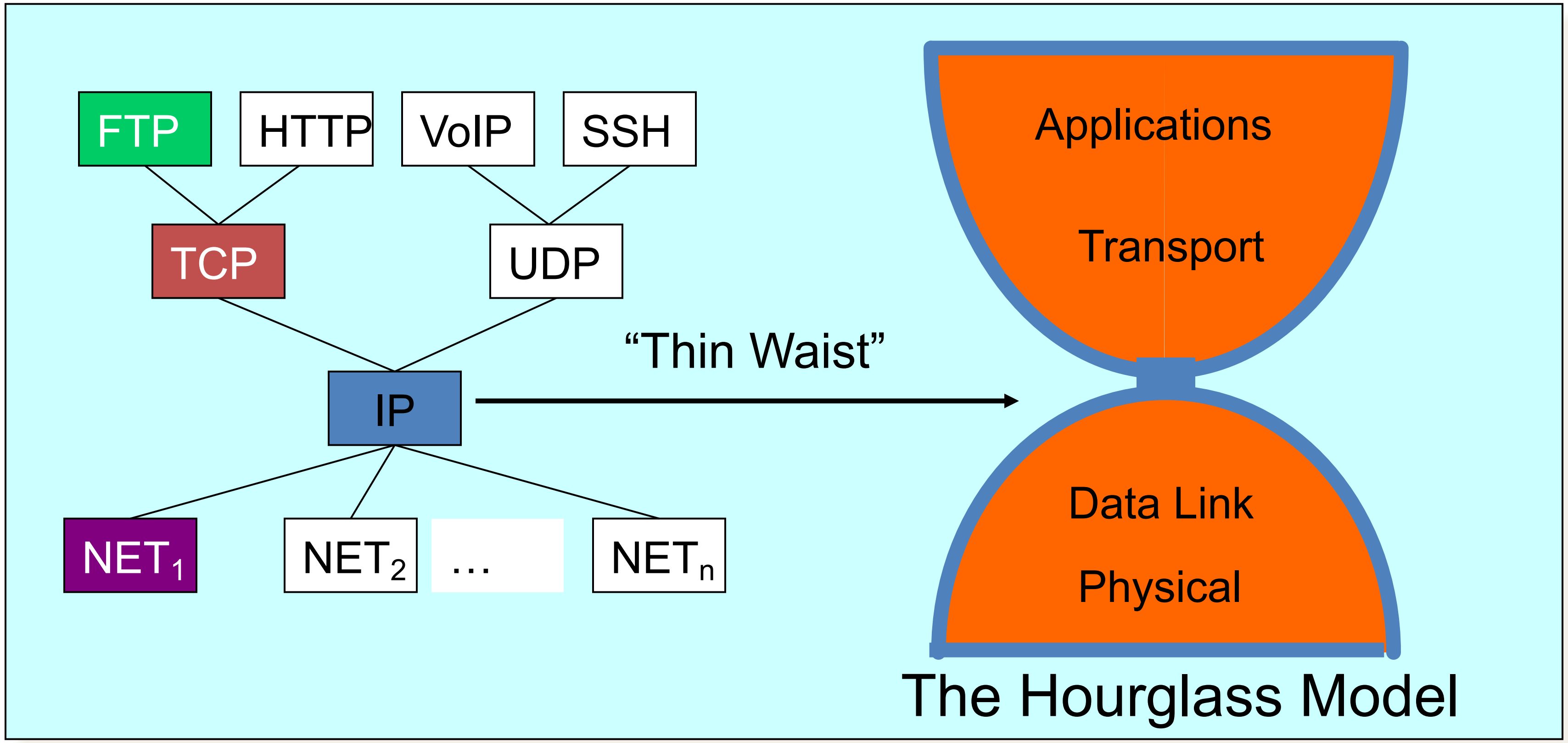
Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

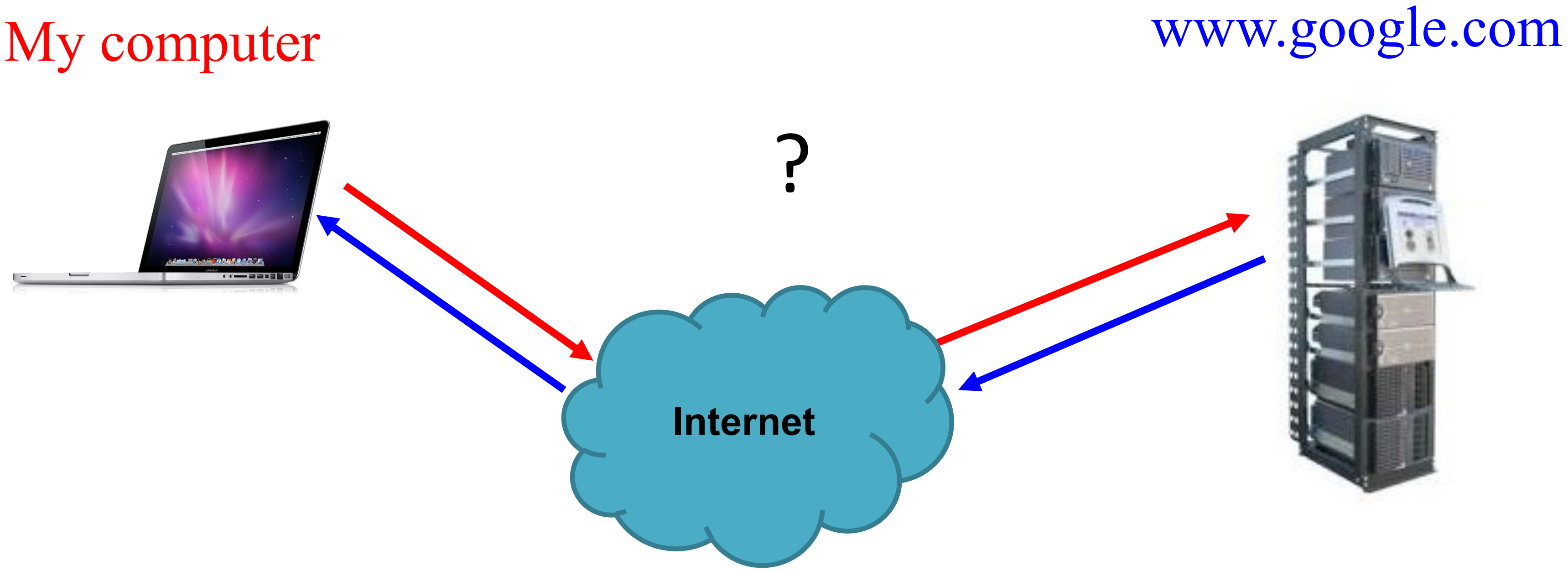
Physical: 1’s and 0’s/bits across a medium
(copper, the air, fiber)

Internet Protocol Suite



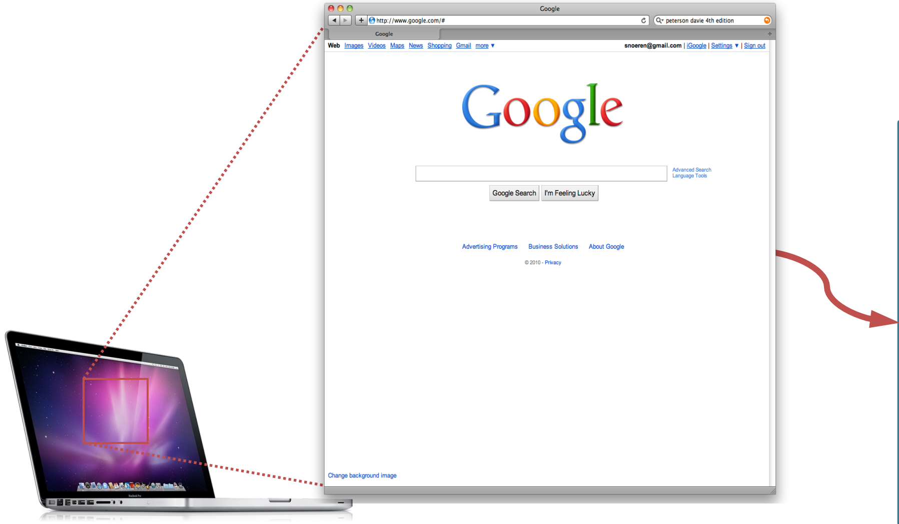
Putting this all together

- **ROUGHLY**, what happens when I click on a Web page from Swarthmore?



Application Layer: Web request (HTTP)

- Turn click into HTTP request

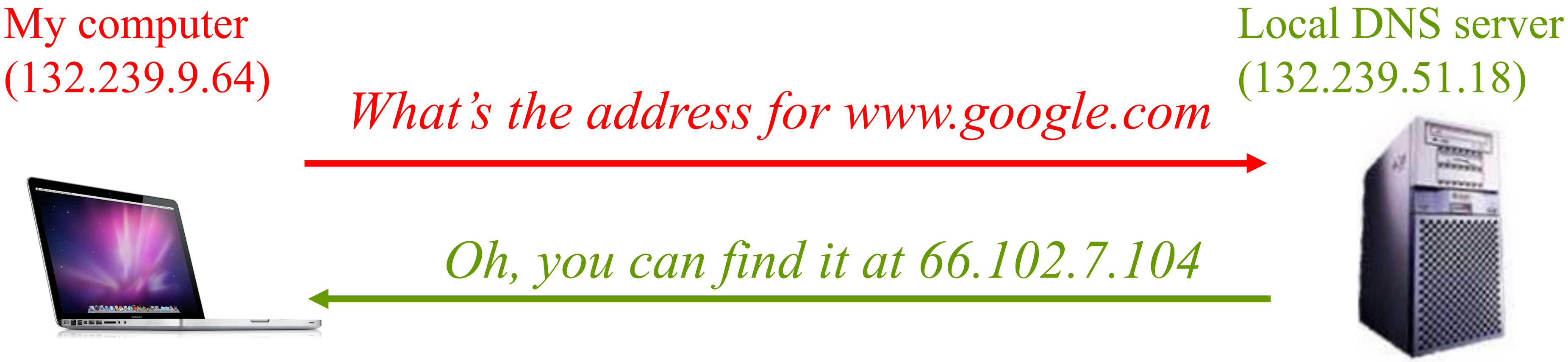


Google
http://www.google.com/#
peterson davie 4th edition
Google
Web Images Videos Maps News Shopping Gmail more ▾
snoeren@gmail.com | iGoogle | Settings ▾ | Sign out
Google
Advanced Search
Language Tools
Google Search
I'm Feeling Lucky
Advertising Programs Business Solutions About Google
© 2010 - Privacy
Change background image

GET http://www.google.com/ HTTP/1.1
Host: www.google.com
…

Application Layer: Name resolution (DNS)

- Where is www.google.com?



My computer
(132.239.9.64)

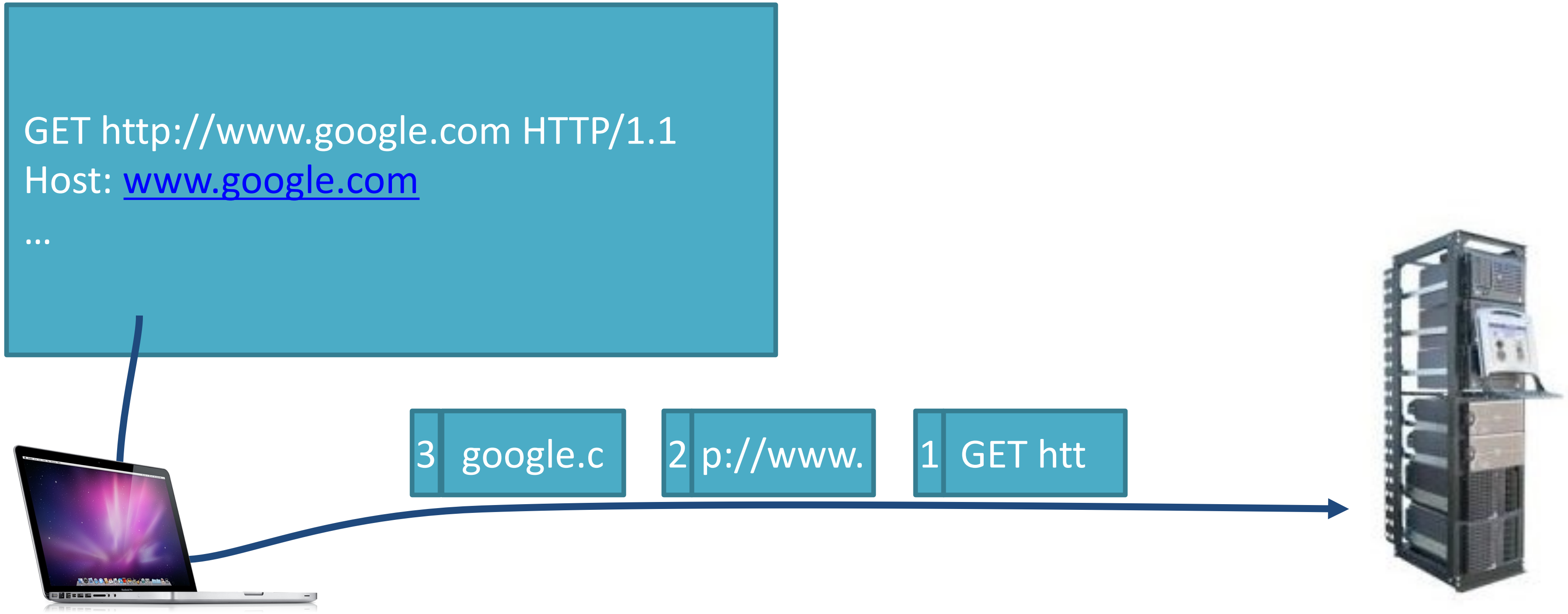
Local DNS server
(132.239.51.18)

What's the address for www.google.com

Oh, you can find it at 66.102.7.104

Transport Layer: TCP

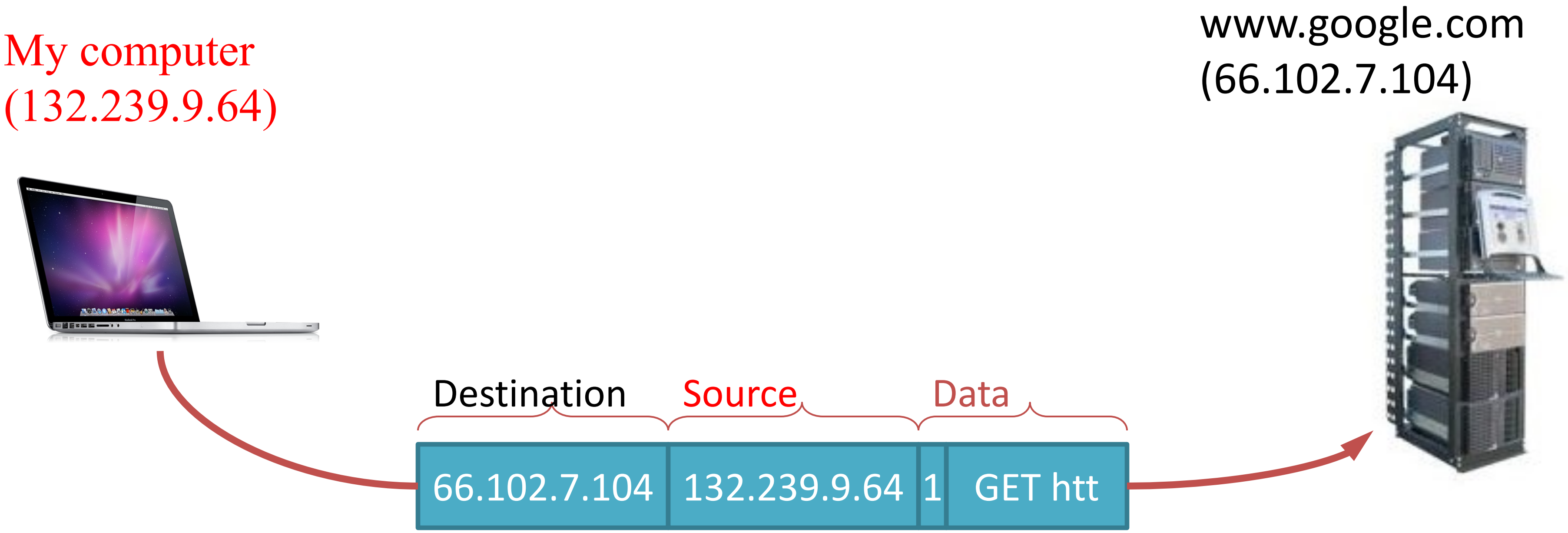
- Break message into packets (TCP segments)
- Should be delivered reliably & in-order



GET http://www.google.com HTTP/1.1
Host: www.google.com
…

Network Layer: Global Network Addressing

- Address each packet so it can traverse network and arrive at host



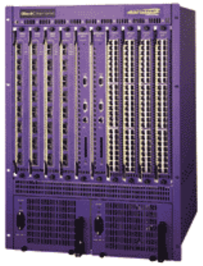
My computer
(132.239.9.64)

www.google.com
(66.102.7.104)

Network Layer: (IP) At Each Router

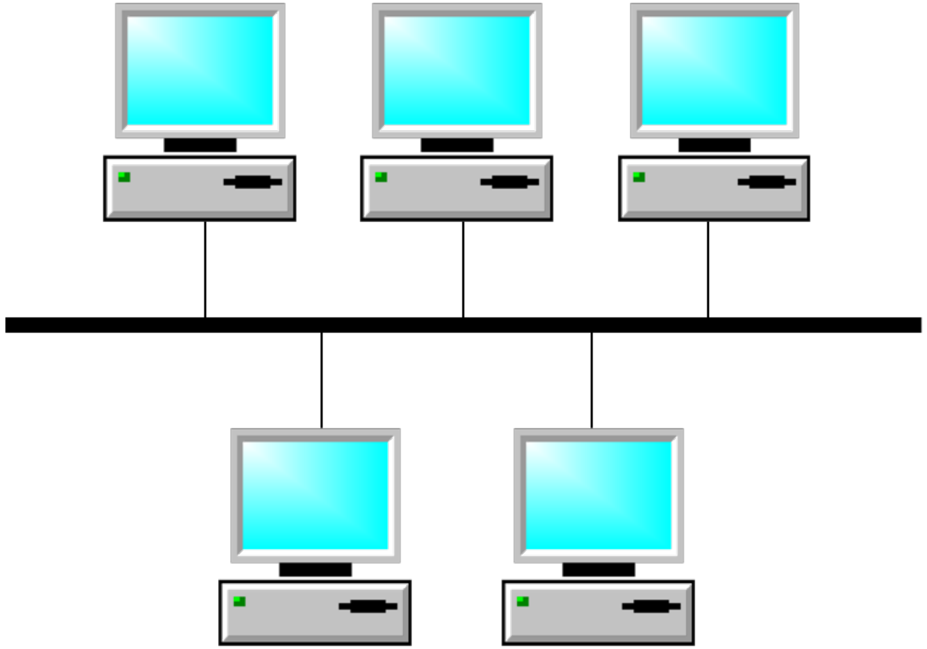
- Where do I send this to get it closer to Google?

- Which is the best route to take?

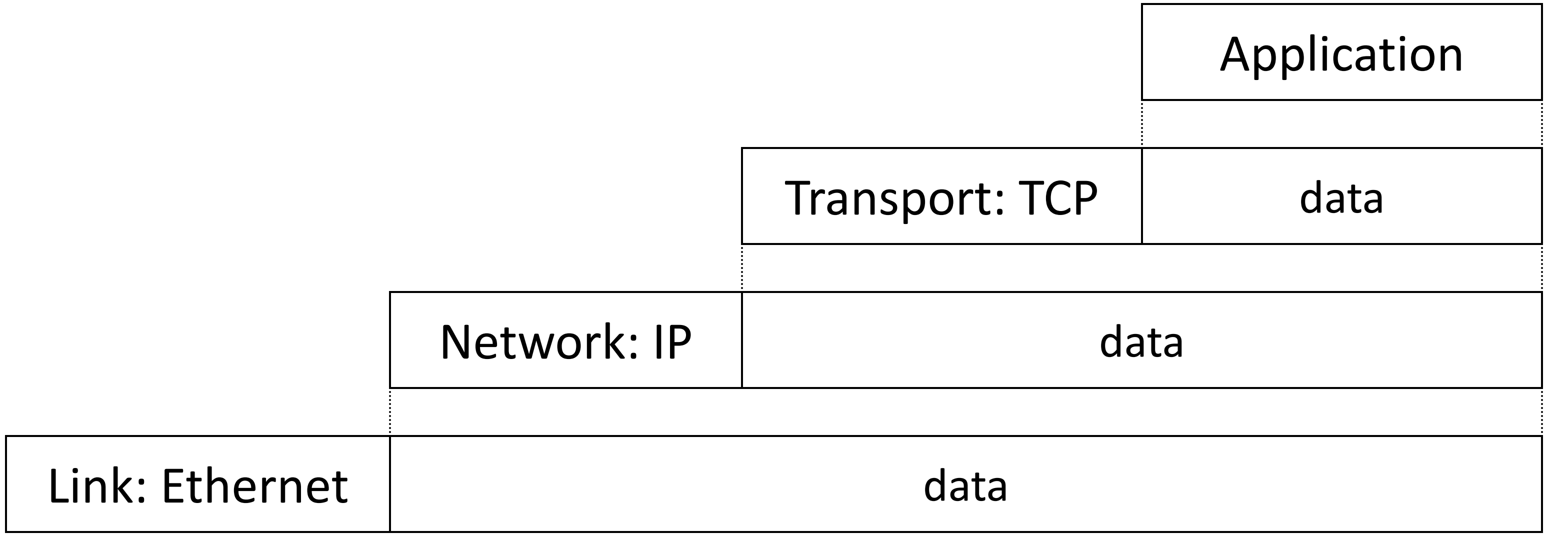


Link & Physical Layers (Ethernet)

- Forward to the next node!
- Share the physical medium.
- Detect errors.

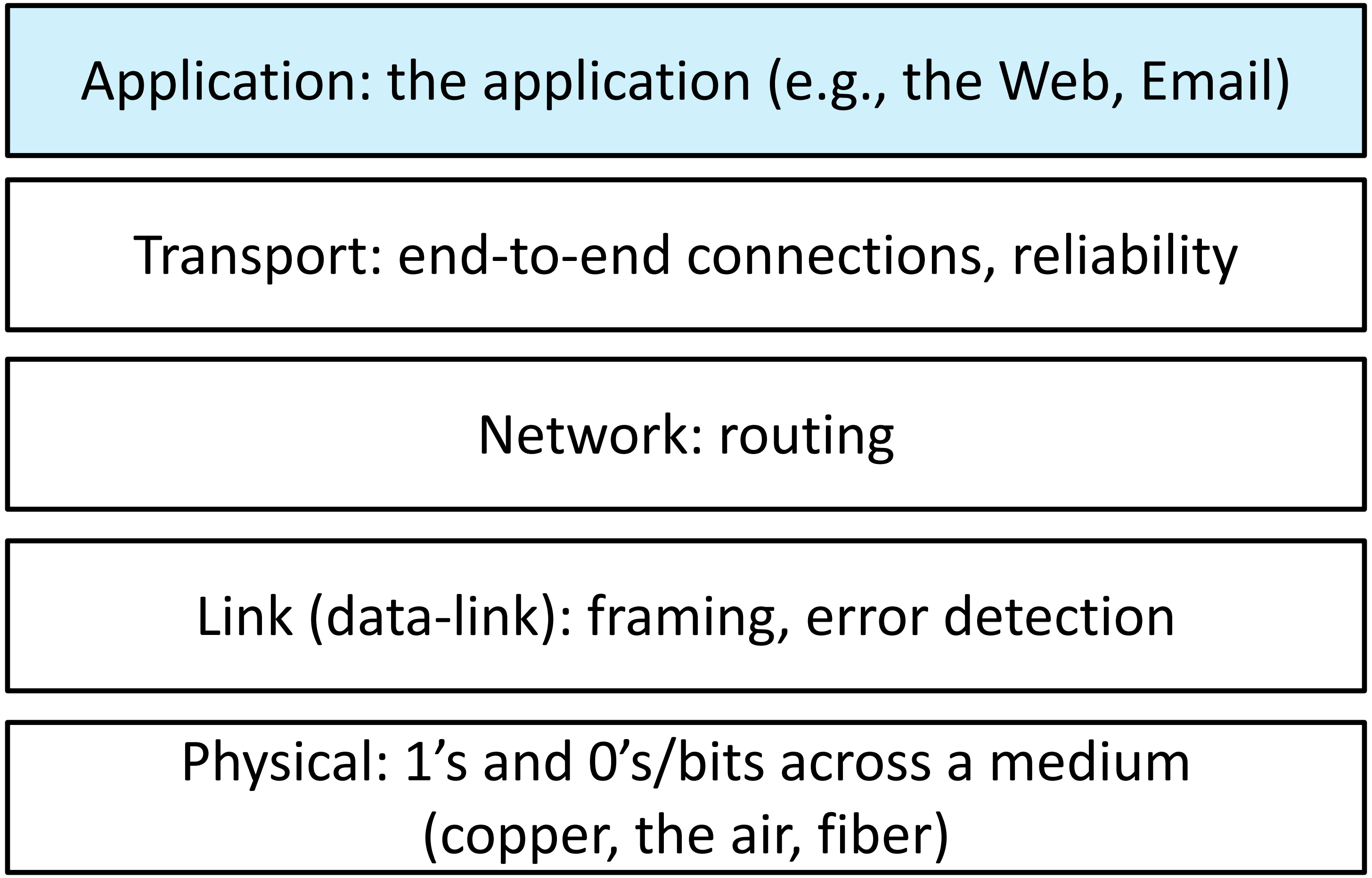


Message Encapsulation



- Higher layer within lower layer
- Each layer has different concerns, provides abstract services to those above

Five-Layer Internet Model



Application: the application (e.g., the Web, Email)

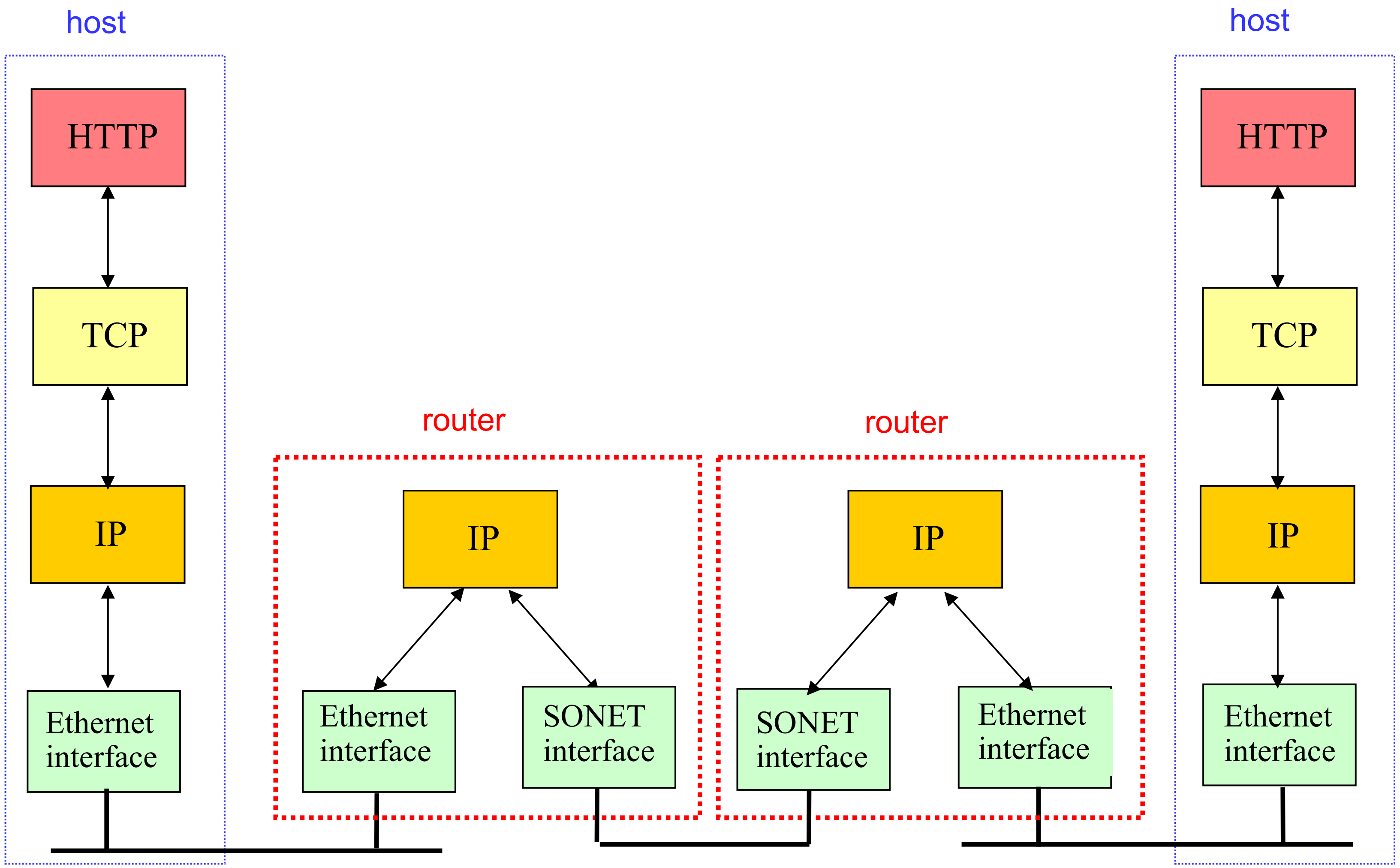
Transport: end-to-end connections, reliability

Network: routing

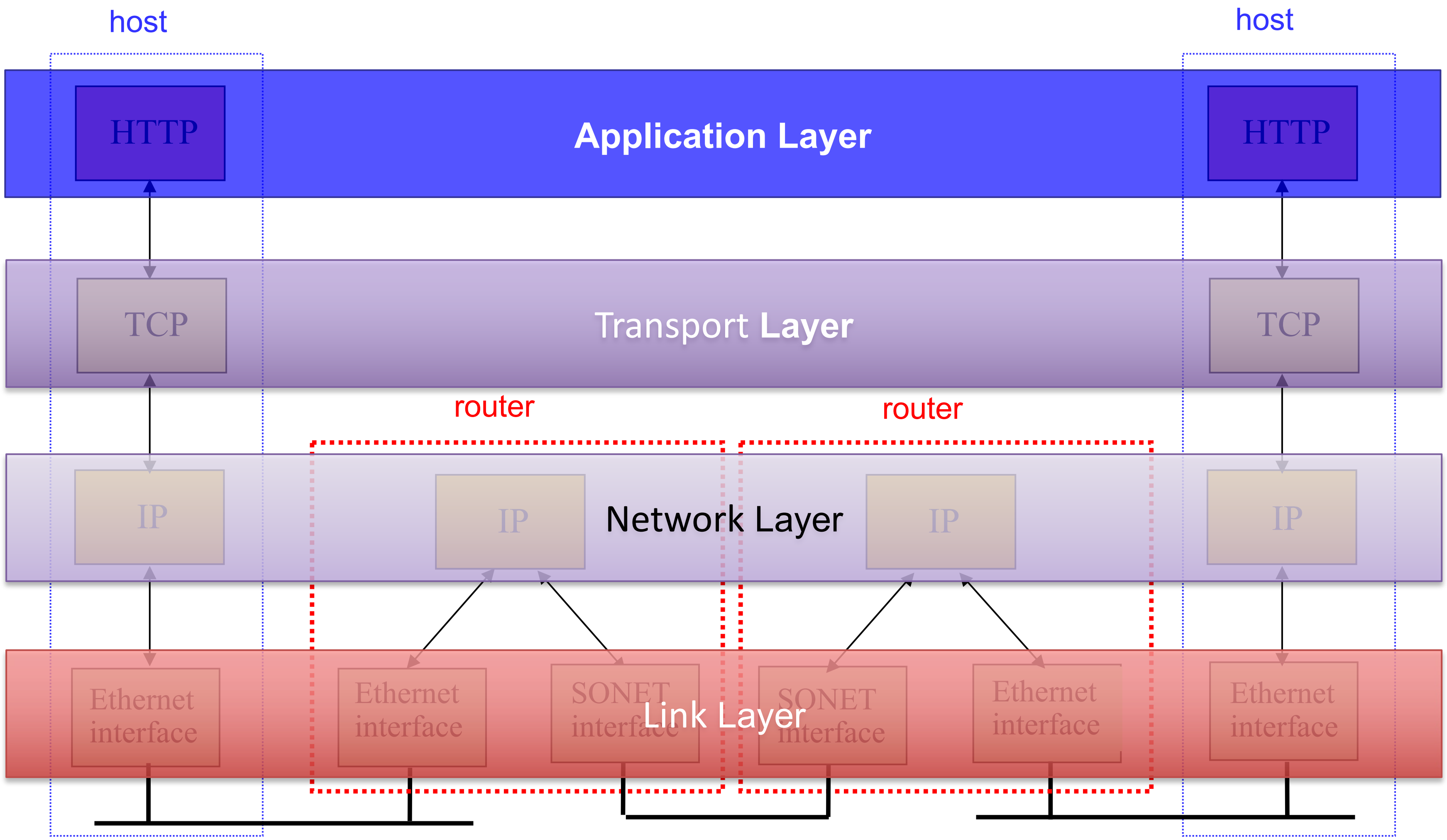
Link (data-link): framing, error detection

Physical: 1's and 0's/bits across a medium
(copper, the air, fiber)

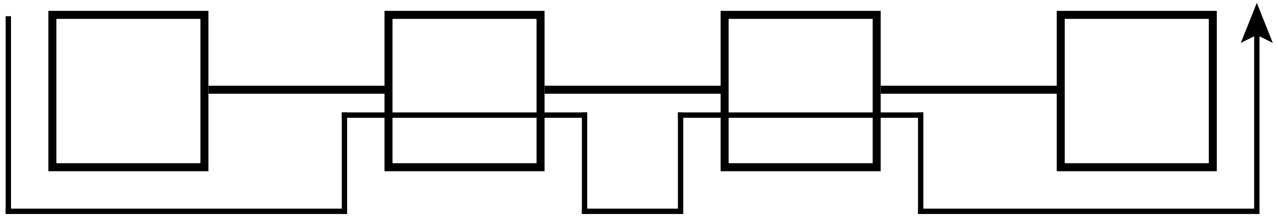
TCP/IP Protocol Stack



TCP/IP Protocol Stack



The "End-to-End" Argument



Don't provide a function at lower layer if you have to do it at higher layer anyway ...

... unless there is a very good performance reason to do so.

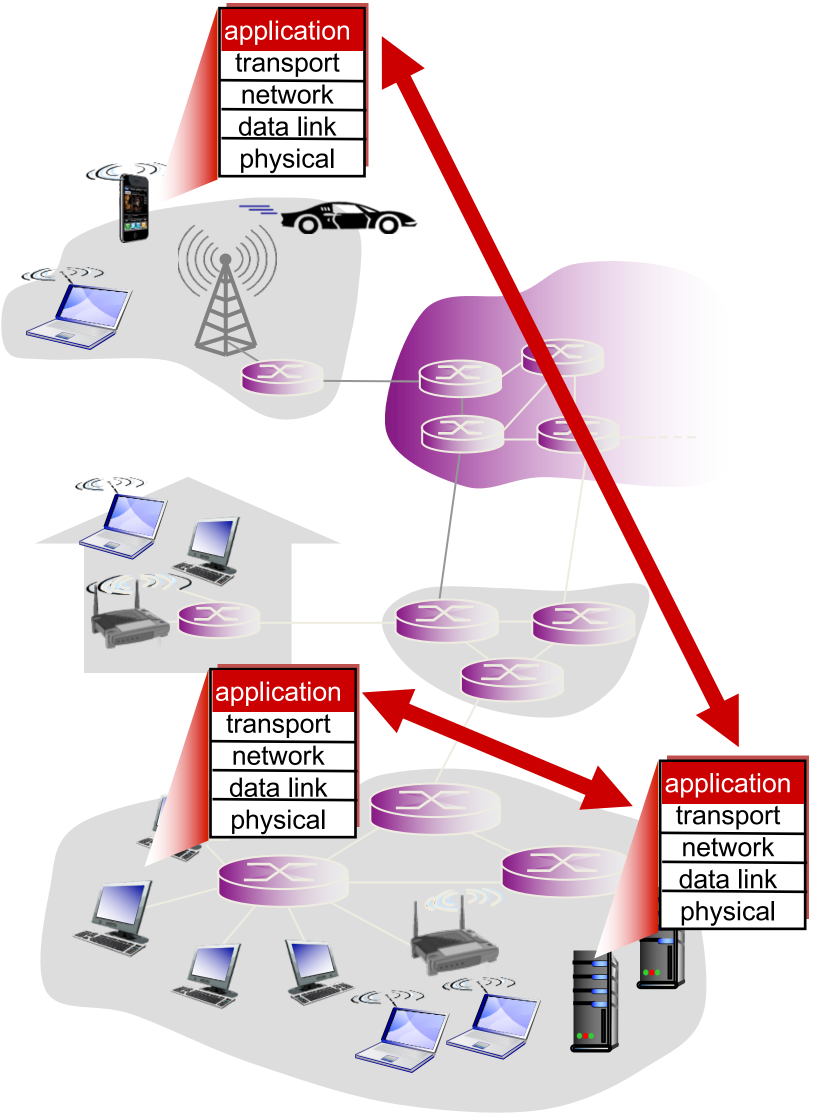
Examples: error control, quality of service

Reference: Saltzer, Reed, Clark, "End-To-End Arguments in System Design," ACM Transactions on Computer Systems, Vol. 2 (4), pp. 277-288, 1984.

Creating a network app

write programs that:

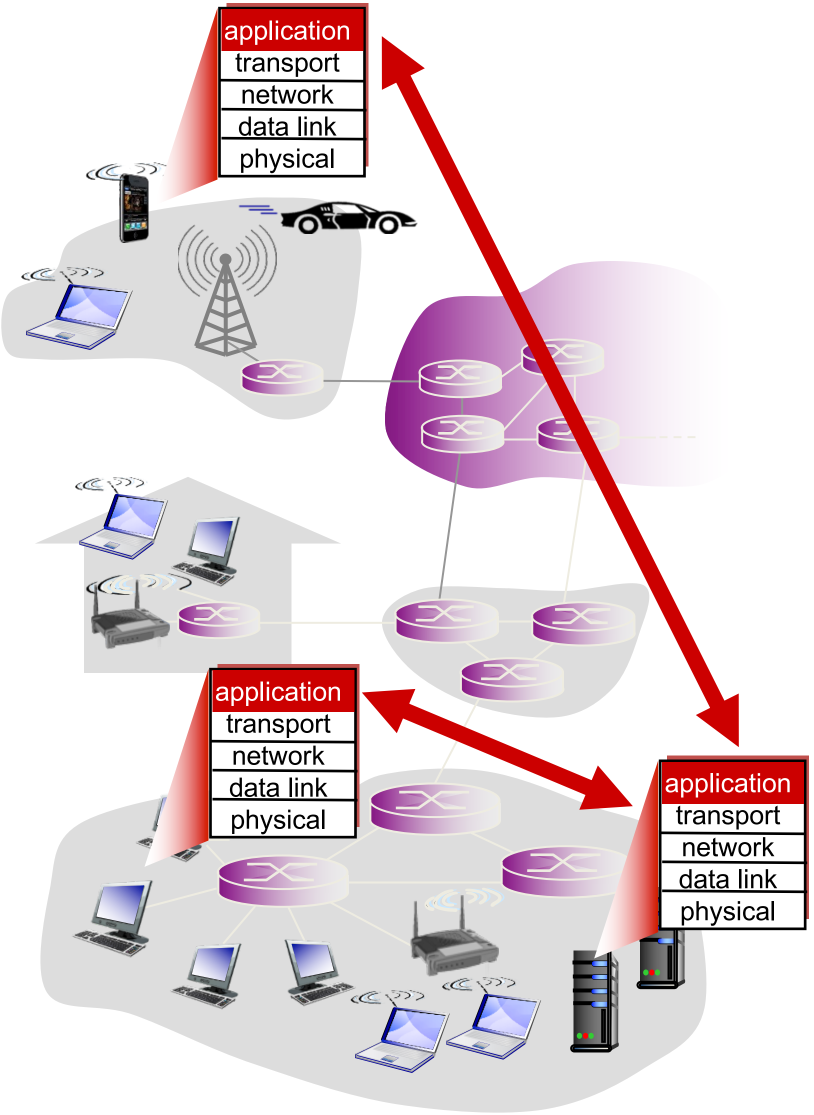
- run on (different) *end systems*
- communicate over network
- e.g., web server s/w communicates with browser software



Creating a network app

no need to write software for network-core devices!

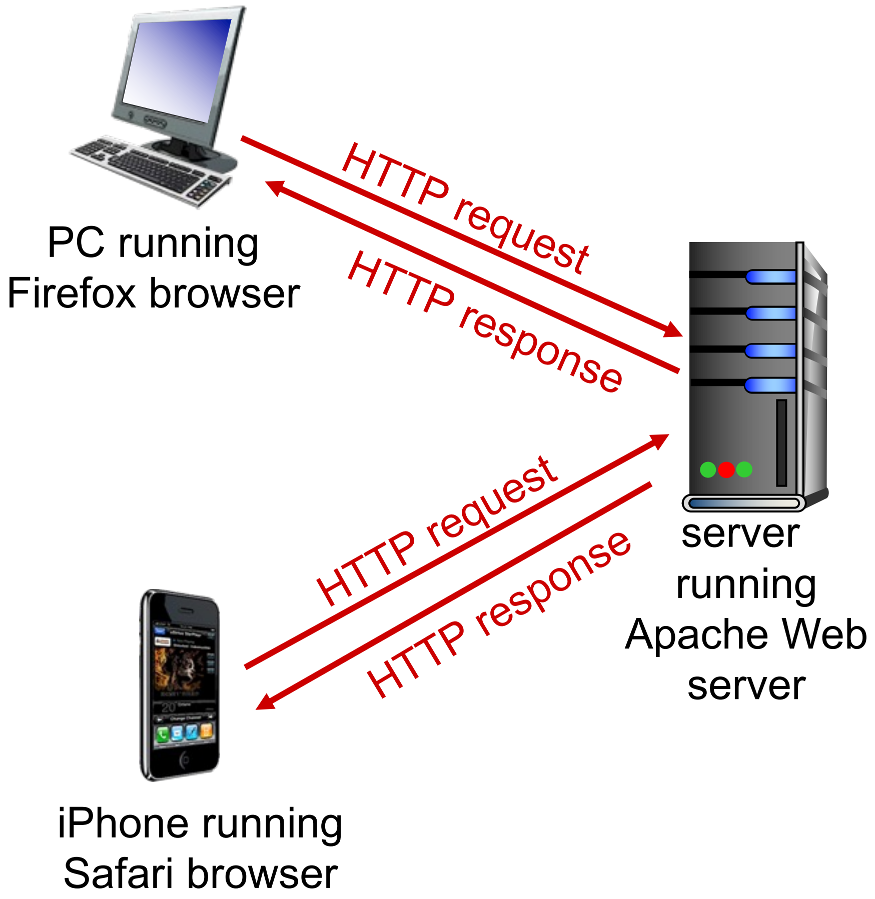
- network-core devices do not run user applications
- applications on end systems
  - – rapid app development, propagation



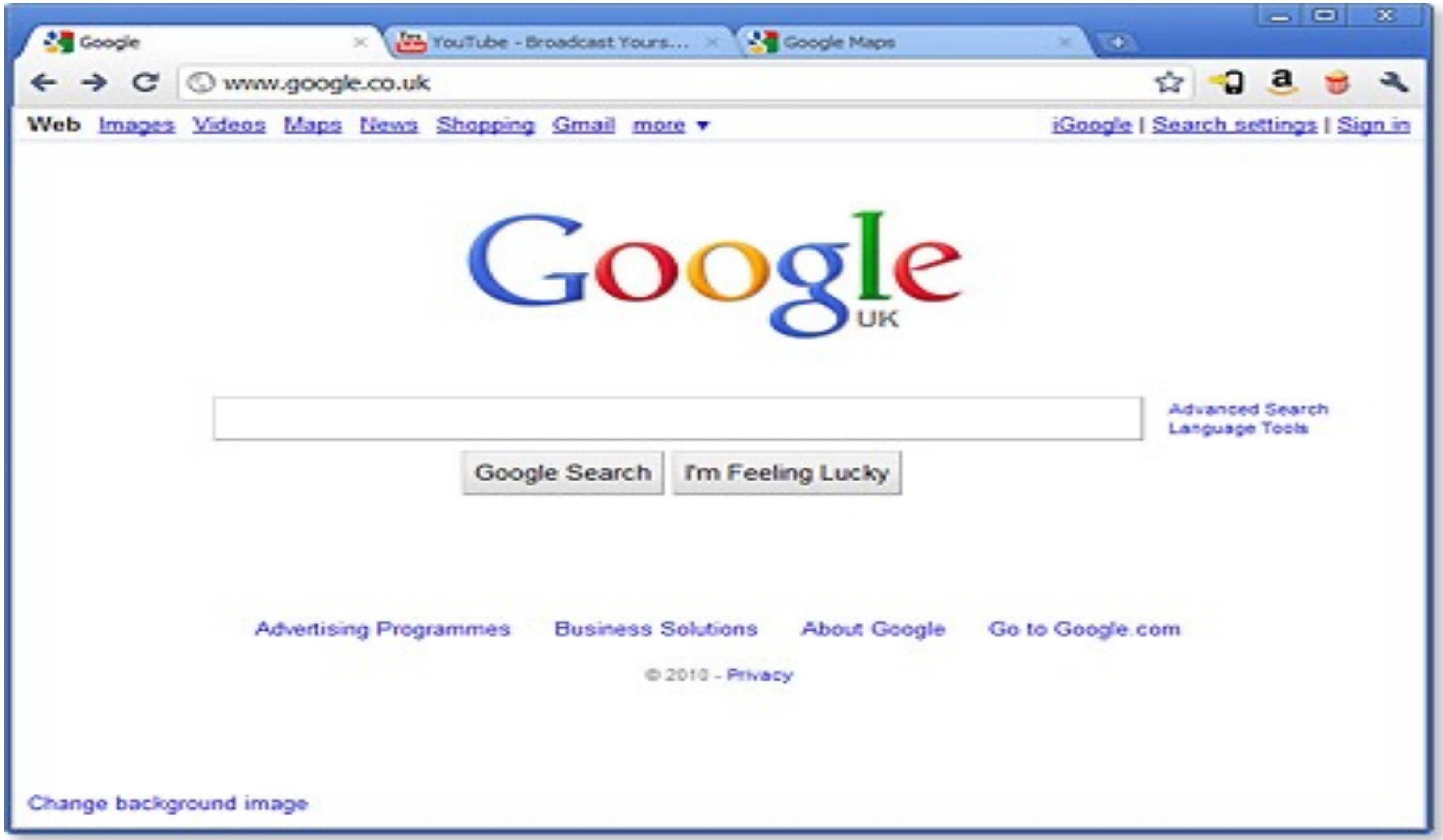
HTTP: HyperText Transfer Protocol

Client/Server model

- client: browser that uses HTTP to request, and receive Web objects.
- server: Web server that uses HTTP to respond with requested object.


PC running Firefox browser
HTTP request
HTTP response
server running Apache Web server
HTTP request
HTTP response
iPhone running Safari browser

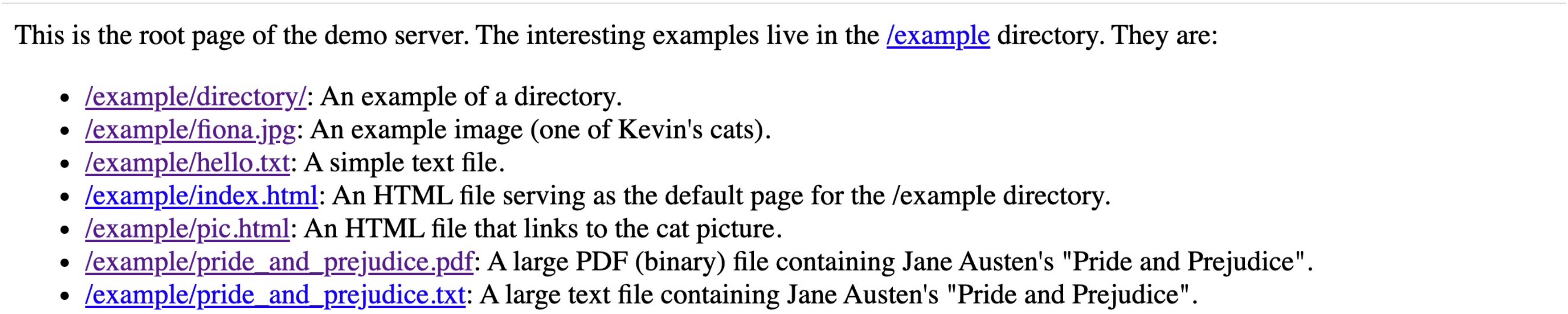
What IS A Web Browser?



HTTP and the Web

- web page consists of objects
- object can be: an HTML file (index.html)

demo.cs.swarthmore.edu/index.html

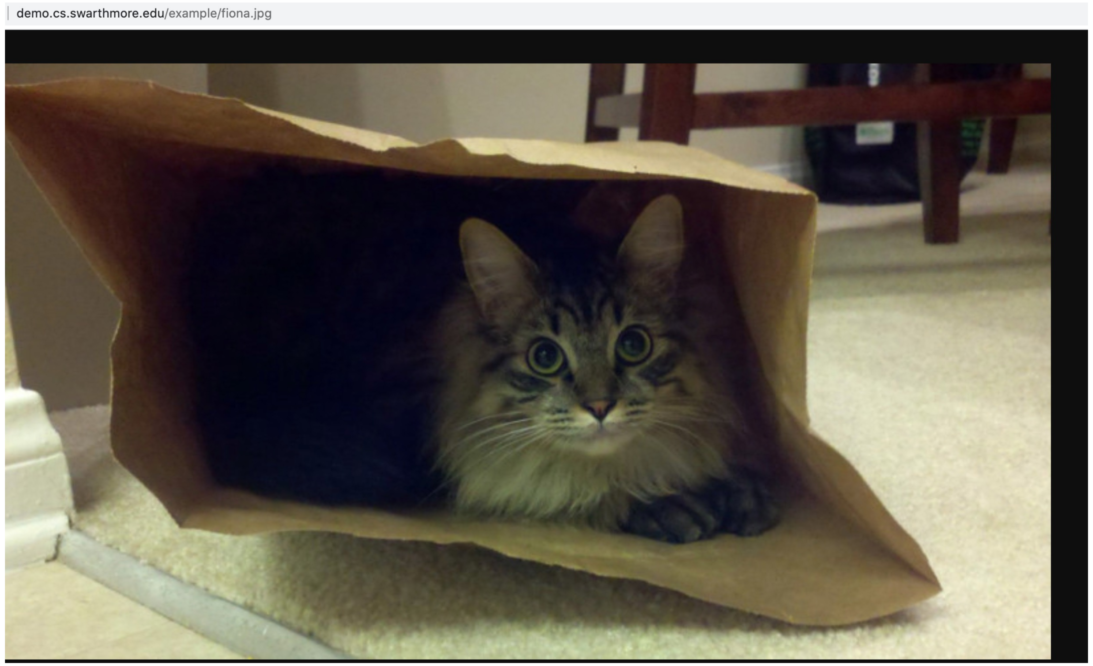


This is the root page of the demo server. The interesting examples live in the /example directory. They are:

- /example/directory/: An example of a directory.
- /example/fiona.jpg: An example image (one of Kevin's cats).
- /example/hello.txt: A simple text file.
- /example/index.html: An HTML file serving as the default page for the /example directory.
- /example/pic.html: An HTML file that links to the cat picture.
- /example/pride_and_prejudice.pdf: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- /example/pride_and_prejudice.txt: A large text file containing Jane Austen's "Pride and Prejudice".

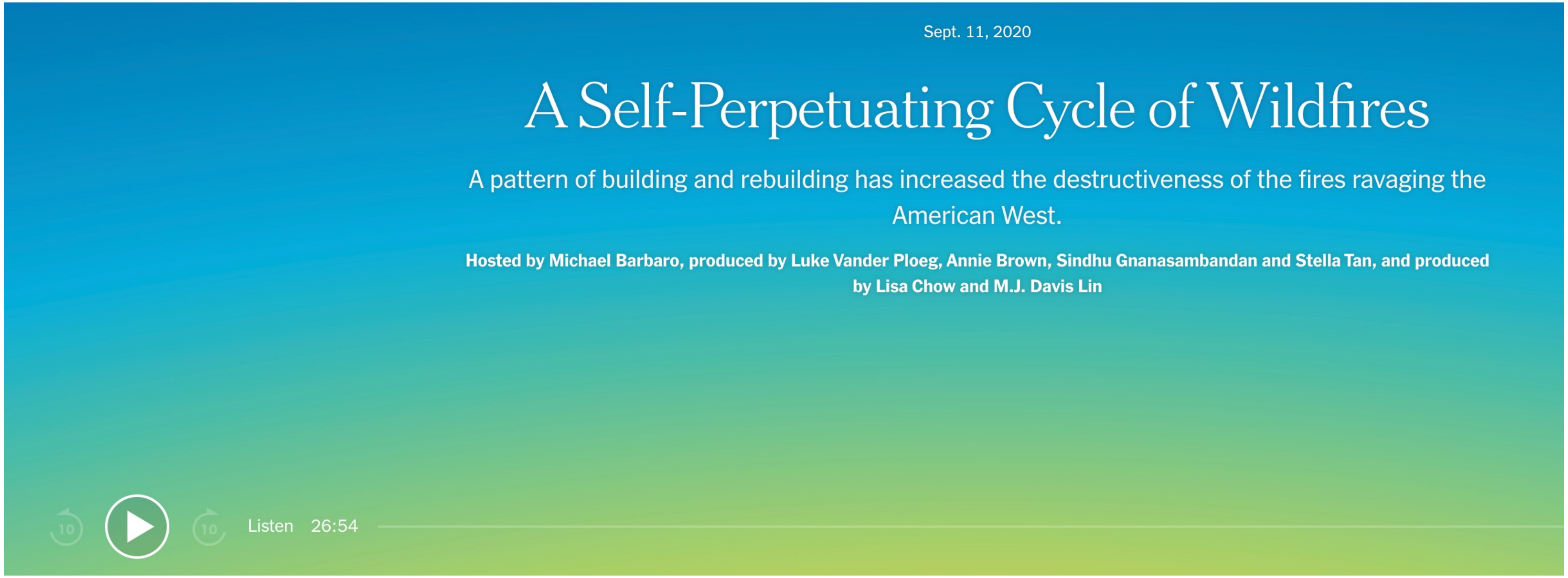
Web objects

- web page consists of objects
- object can be: JPEG image



Web objects

- web page consists of objects
- object can be: audio file



Sept. 11, 2020

A Self-Perpetuating Cycle of Wildfires

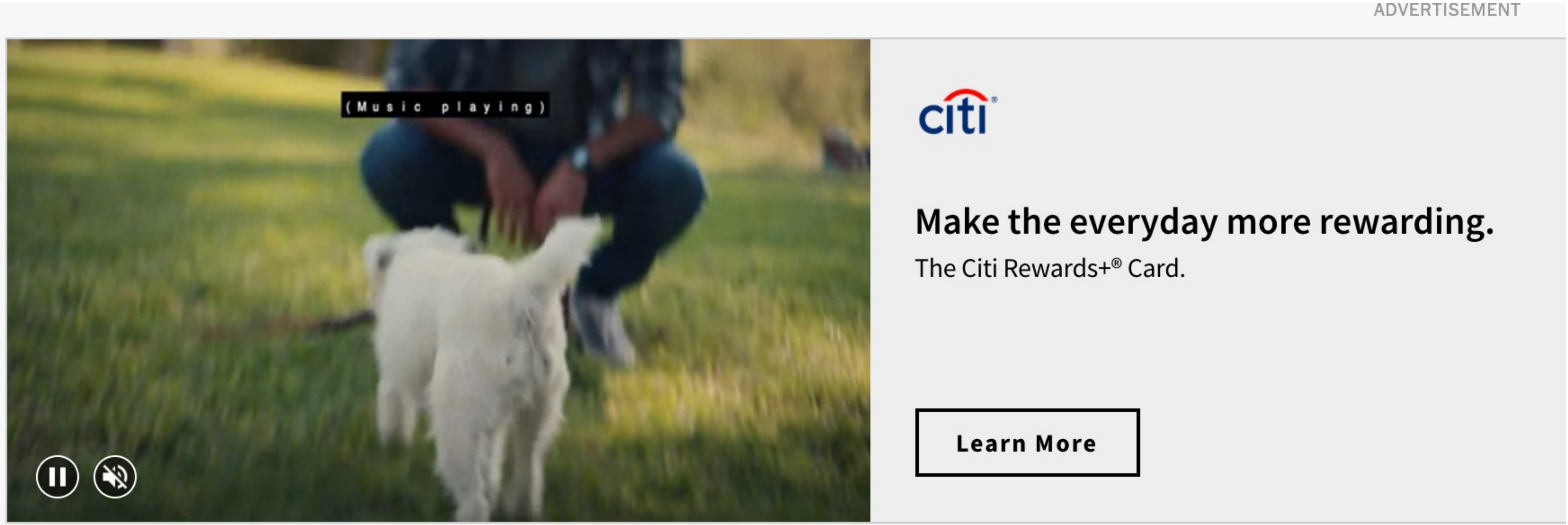
A pattern of building and rebuilding has increased the destructiveness of the fires ravaging the American West.

Hosted by Michael Barbaro, produced by Luke Vander Ploeg, Annie Brown, Sindhu Gnanasambandan and Stella Tan, and produced by Lisa Chow and M.J. Davis Lin

Courtesy: New York Times

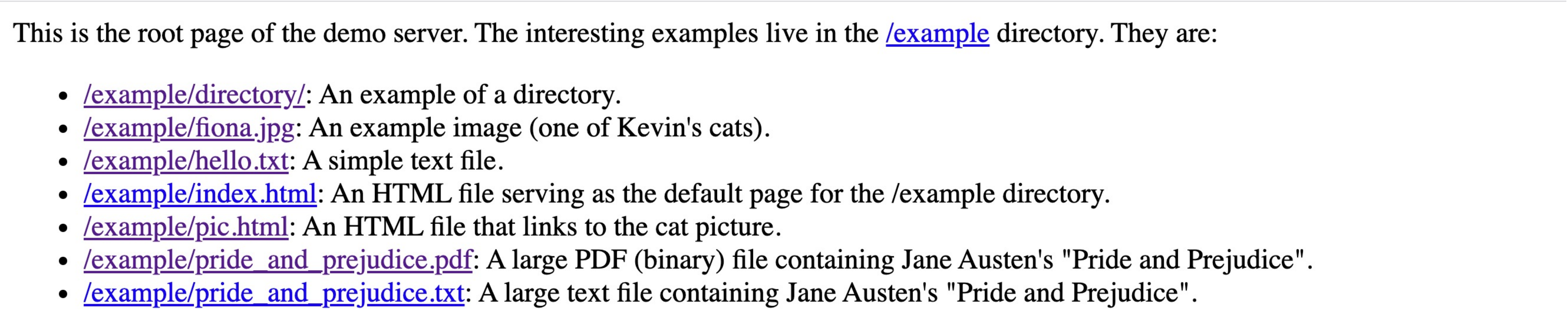
Web objects

- web page consists of objects
- object can be: video, java applets, etc.



HTTP and the Web

- a web page consists of base HTML-file which includes several referenced objects
- each object is addressable by a URL, e.g.,

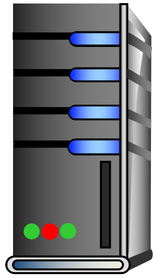


This is the root page of the demo server. The interesting examples live in the /example directory. They are:

- /example/directory/: An example of a directory.
- /example/fiona.jpg: An example image (one of Kevin's cats).
- /example/hello.txt: A simple text file.
- /example/index.html: An HTML file serving as the default page for the /example directory.
- /example/pic.html: An HTML file that links to the cat picture.
- /example/pride_and_prejudice.pdf: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- /example/pride_and_prejudice.txt: A large text file containing Jane Austen's "Pride and Prejudice".

$$\underbrace{\texttt{demo.cs.swarthmore.edu}}_{\text{host name}}\texttt{/}\underbrace{\texttt{example/pic.html}}_{\text{path name}}$$

HTTP Overview



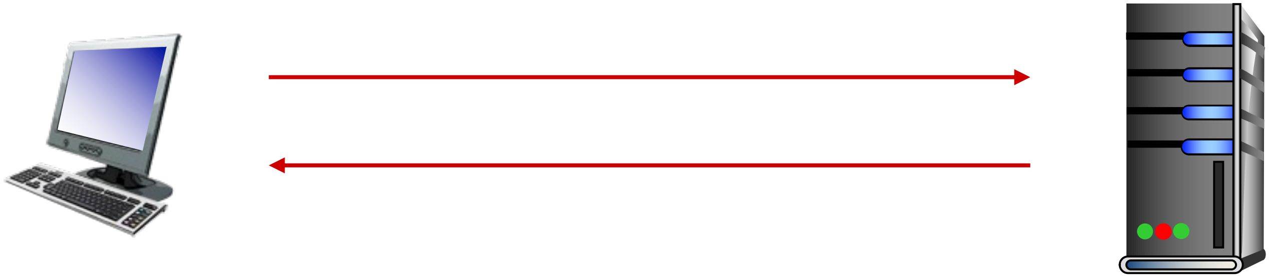
1. User types in a URL.

http://some.host.name.tld/directory/name/file.ext

host name

path name

HTTP Overview



2. Browser establishes connection with server using the Sockets API.

Calls socket()   // create a socket

Looks up “some.host.name.tld” (DNS: getaddrinfo)

Calls connect() // connect to remote server

Ready to call send() // Can now send HTTP requests

HTTP Overview

3. Browser requests data the user asked for

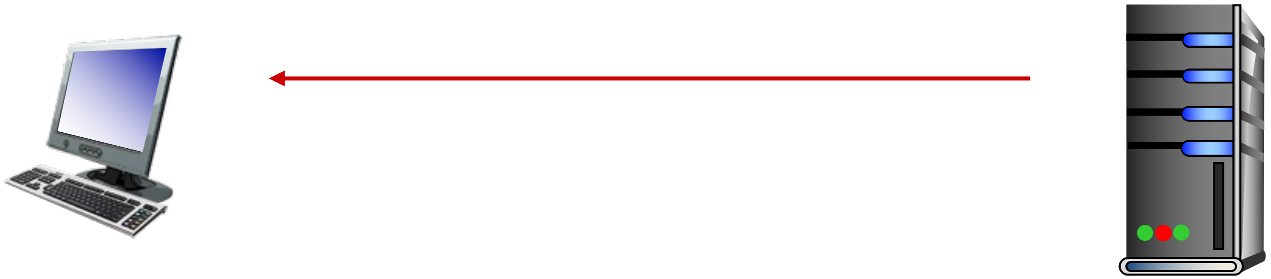
```
GET /directory/name/file.ext HTTP/1.0
Host: some.host.name.tld
```

Required fields

[other optional fields, for example:]

```
User-agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Accept-language: en
```

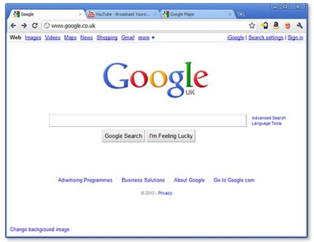
HTTP Overview



4. Server responds with the requested data.

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1299
Date: Sun, 01 Sep 2013 21:26:38 GMT
[Blank line]
(Data data data data…)
```

HTTP Overview



5. Browser renders the response, fetches any additional objects, and closes the connection.

HTTP Overview

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser renders the response, fetches other objects, and closes the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.

HTTP Overview (Lab 1)

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. ~~Browser renders the response, fetches other objects,~~ Save the file and close the connection.

It's a document retrieval system, where documents point to (link to) each other, forming a "web".

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet demo.cs.swarthmore.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at example server.

Anything typed is sent to server on port 80 at demo.cs.swarthmore.edu

Trying out HTTP (client side) for yourself

2. Type in a GET HTTP request:

(Hit carriage return twice) This is a minimal, but complete, GET request to the HTTP server.

```
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu
(blank line)
```

3. Look at response message sent by HTTP server!

Example

```
$ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu

HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: text/html
Accept-Ranges: bytes
ETag: "316912886"
Last-Modified: Wed, 04 Jan 2017 17:47:31 GMT
Content-Length: 1062
Date: Wed, 05 Sep 2018 17:27:34 GMT
Server: lighttpd/1.4.35
```

Response headers

Example

```
$ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu
```

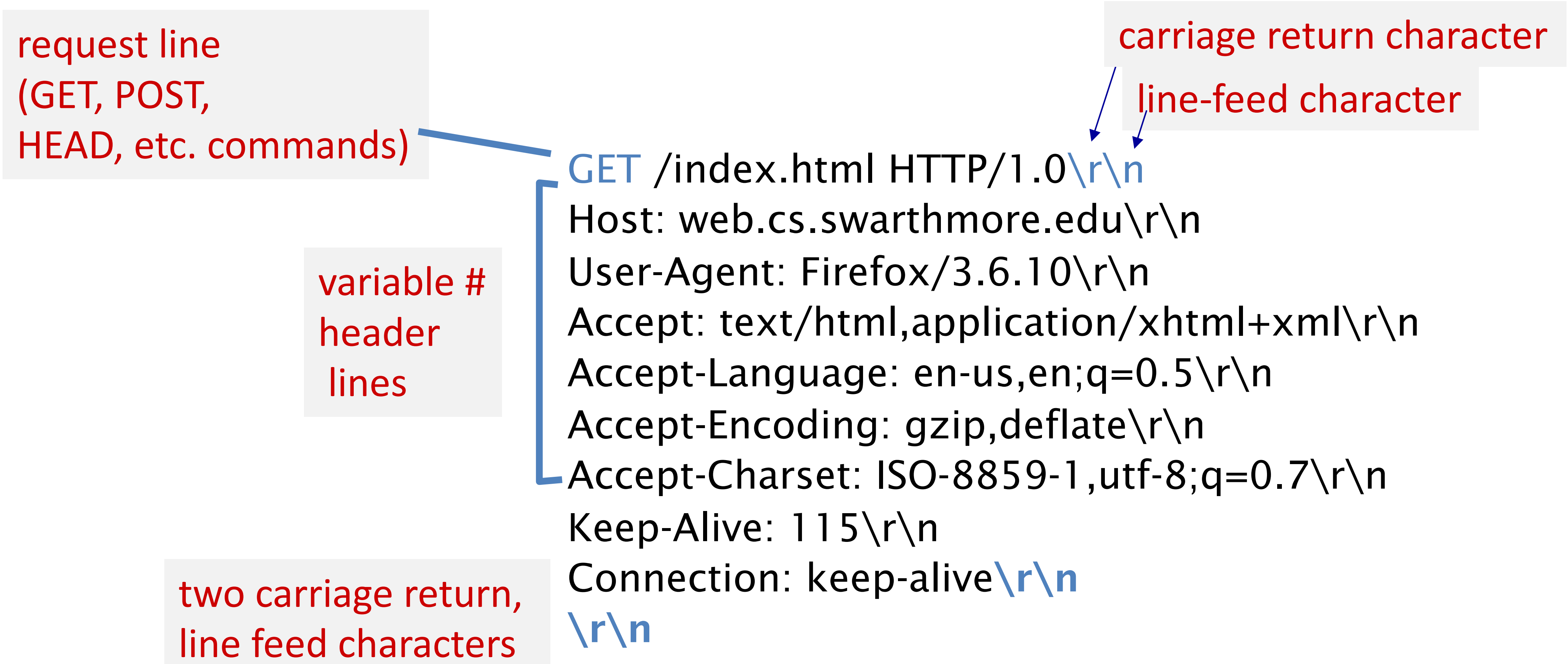
Response
headers

```
<html><head><title>Demo Server</title></head>
<body>
.....
</body>
</html>
```

Response
body
(This is what you
should be saving in
lab 1.)

HTTP request message

- two types of HTTP messages: **request, response**
- **HTTP request message:** ASCII (human-readable format)



request line (GET, POST, HEAD, etc. commands)

carriage return character

line-feed character

variable # header lines

two carriage return, line feed characters

```
GET /index.html HTTP/1.0\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP response message

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

status line (protocol status code status phrase)

variable # header lines

two carriage return, line feed characters

data, e.g., requested HTML file: may not be text!

HTTP response status codes

Status code appears in first line of server-to-client response message.

200 OK

- Request succeeded, requested object later in this msg

301 Moved Permanently

- Requested object moved, new location specified later in this msg (Location:)

400 Bad Request

  - Request msg not understood by server

403 Forbidden

  - You don’t have permission to read the object

404 Not Found

  - Requested document not found on this server

505 HTTP Version Not Supported

HTTP response status codes

Status code appears in first line of server-to-client response message.

Many others! Search “list of HTTP status codes”

420 Enhance Your Calm (twitter)

- Slow down, you’re being rate limited

451 Unavailable for Legal Reasons

- Censorship?

418 I’m a Teapot

- Response from a teapot requested to brew a beverage (announced Apr 1)

Client-Server communication

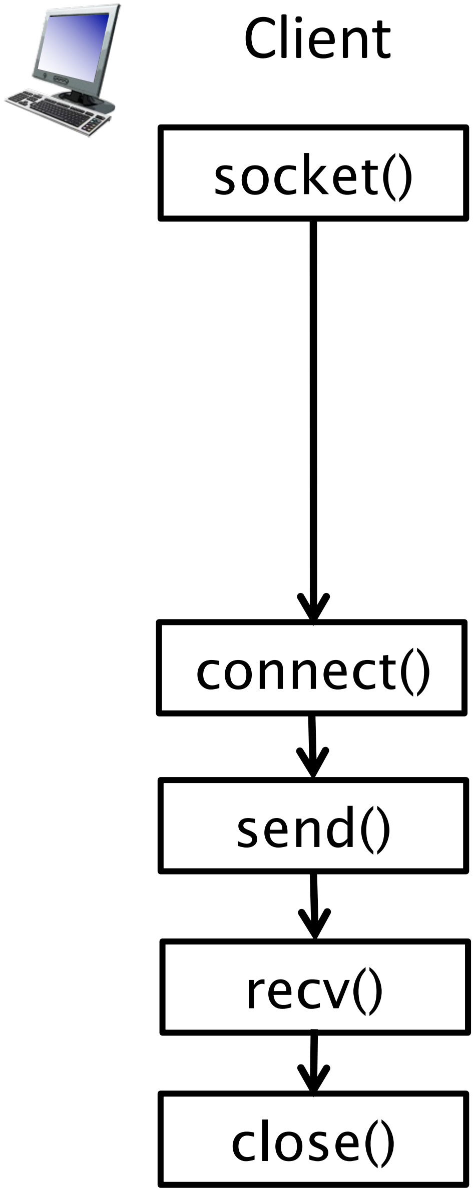
- Client:
  - initiates communication
  - must know the address and port of the server
  - active socket
- Server:
  - passively waits for and responds to clients
  - passive socket

What is a socket?

An abstraction through which an application may send and receive data,

in the same way as a open-file handle allows an application to read and write data to storage.

TCP Socket Procedures: Client



create a new communication endpoint

actively attempt to establish a connection

receive some data over a connection

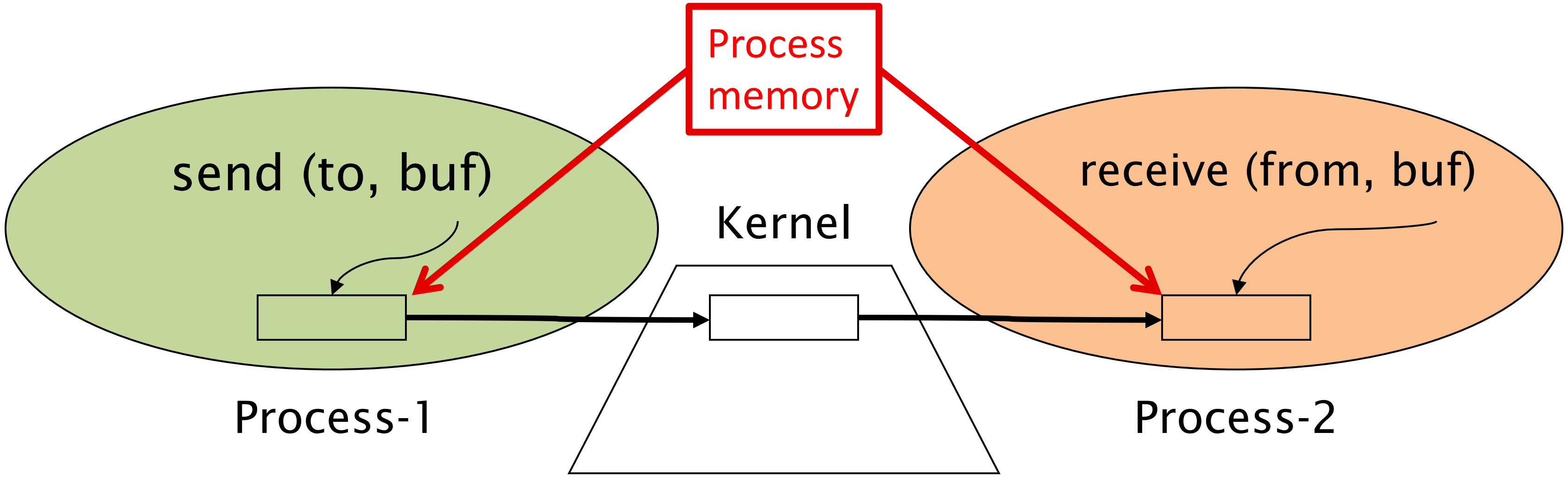
send some data over a connection

release the connection

Recall Inter-process Communication (IPC)

- Processes must communicate to cooperate
- Must have two mechanisms:
  - Data transfer
  - Synchronization
- On a single machine:
  - Threads (shared memory)
  - Message passing

Message Passing (local)

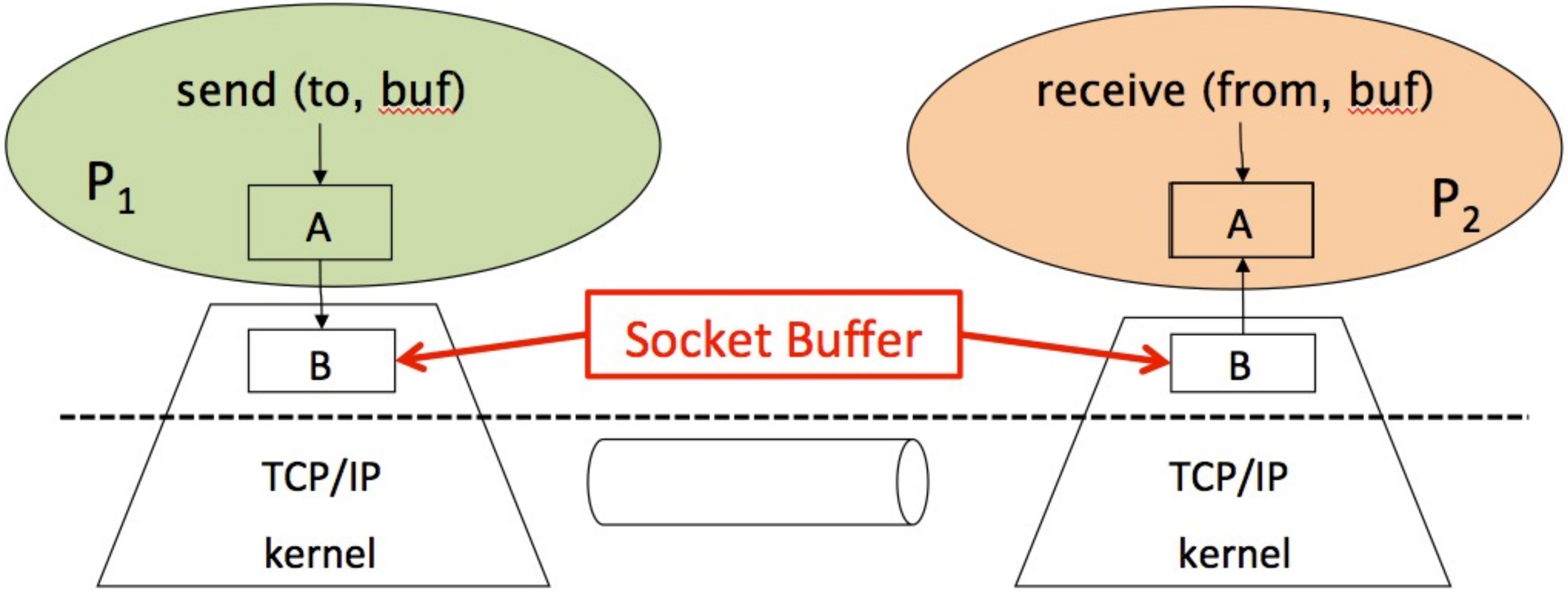


- Operating system mechanism for IPC
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization

Interprocess Communication (non-local)

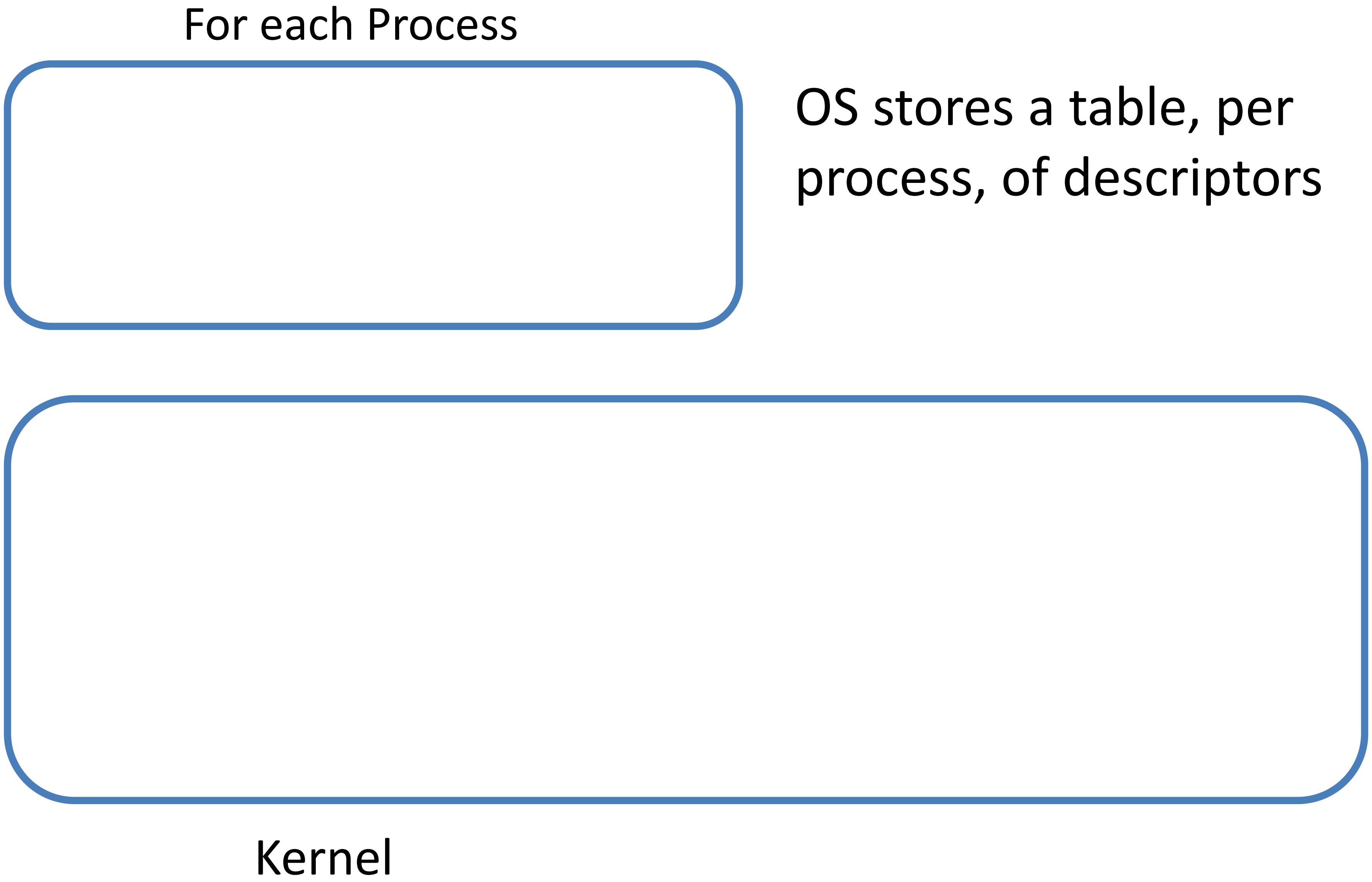
- Processes must communicate to cooperate
- Must have two mechanisms:
  - Data transfer
  - Synchronization
- Across a network:
  - Threads (shared memory) NOT AN OPTION!
  - Message passing

Message Passing (network)



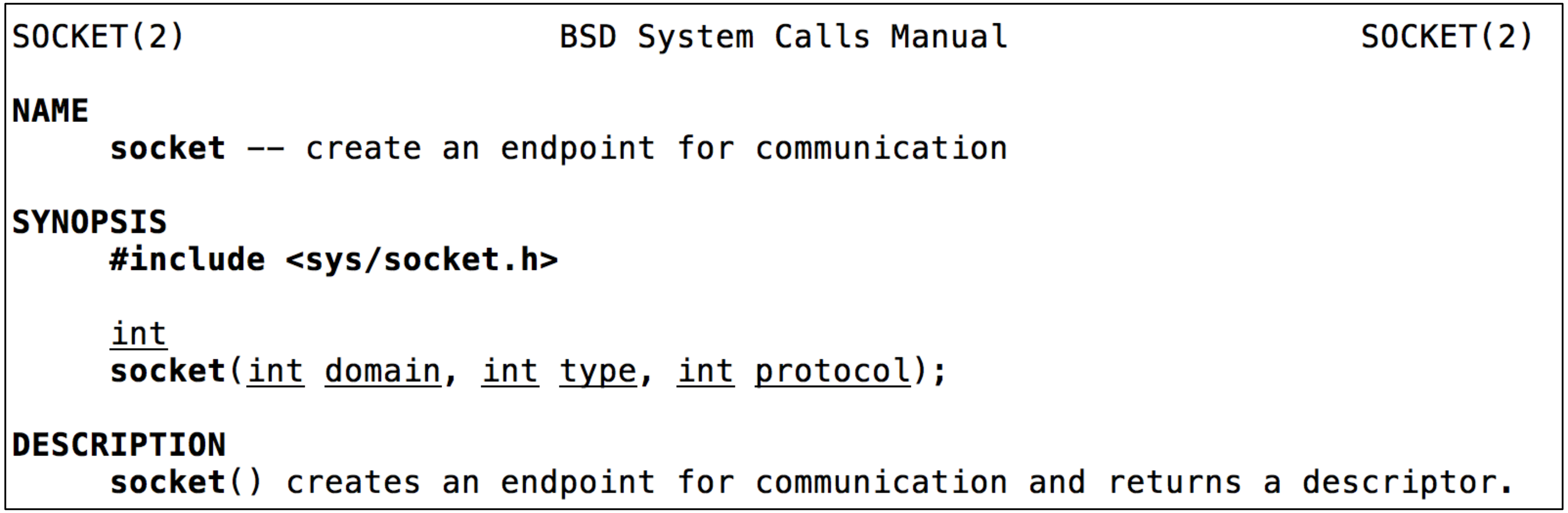
- Same synchronization
- Data transfer
  - Copy to/from OS socket buffer
  - Extra step across network: hidden from applications

Descriptor Table



OS stores a table, per process, of descriptors

Descriptors



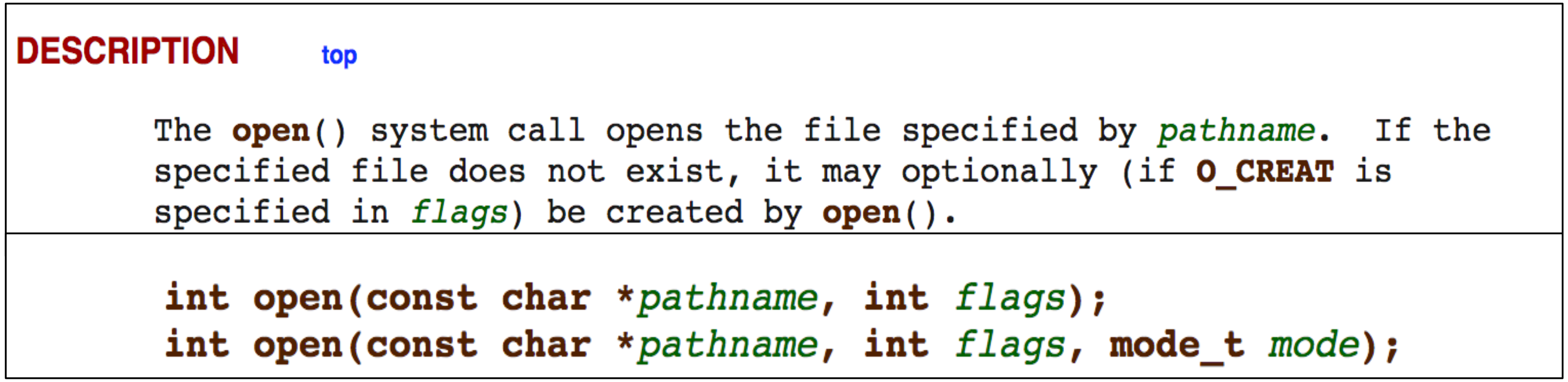
```
SOCKET(2)                BSD System Calls Manual                SOCKET(2)

NAME
     socket -- create an endpoint for communication

SYNOPSIS
     #include <sys/socket.h>

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

DESCRIPTION
     socket() creates an endpoint for communication and returns a descriptor.
```

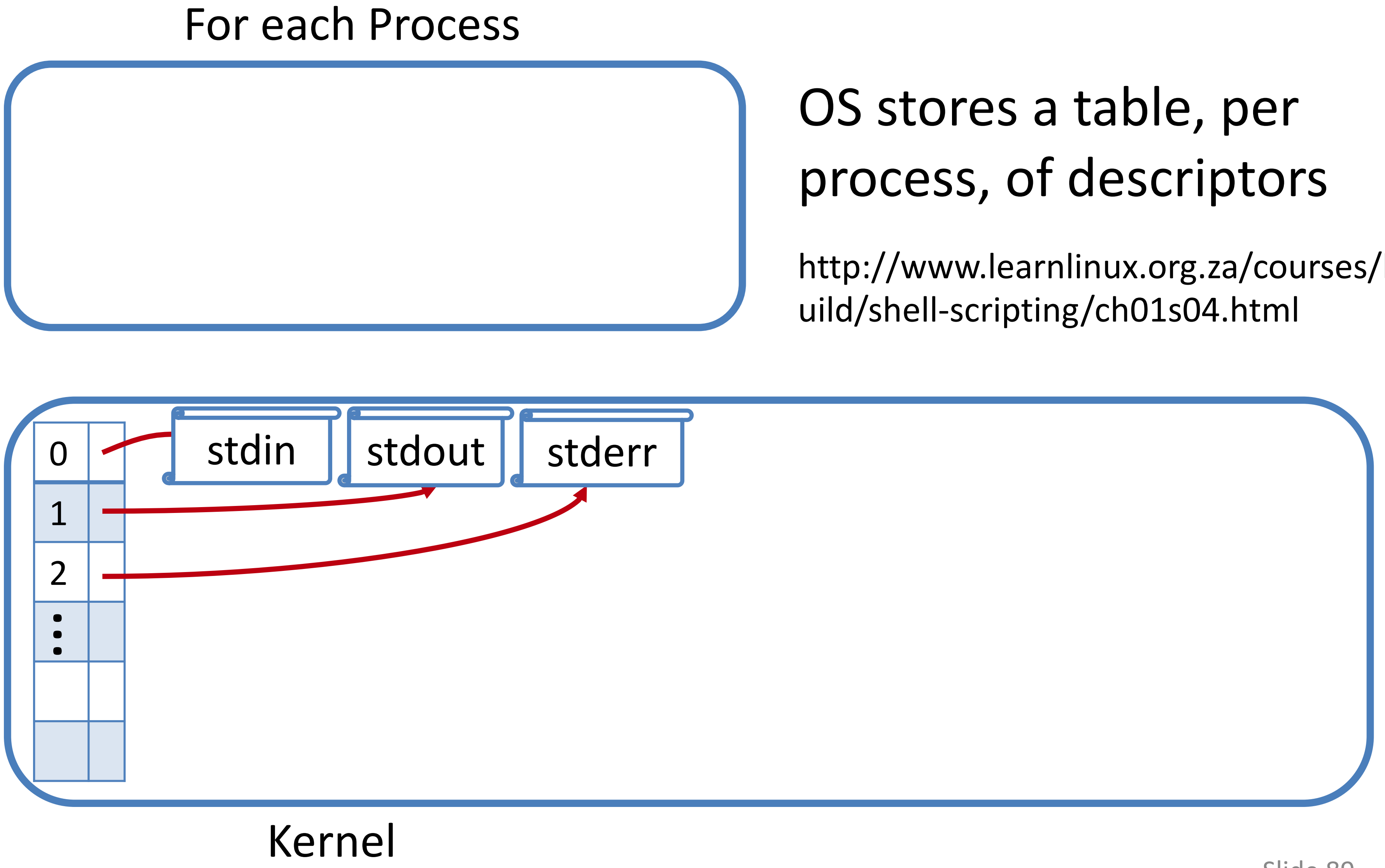


DESCRIPTION top

```
       The open() system call opens the file specified by pathname.  If the
       specified file does not exist, it may optionally (if O_CREAT is
       specified in flags) be created by open().
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

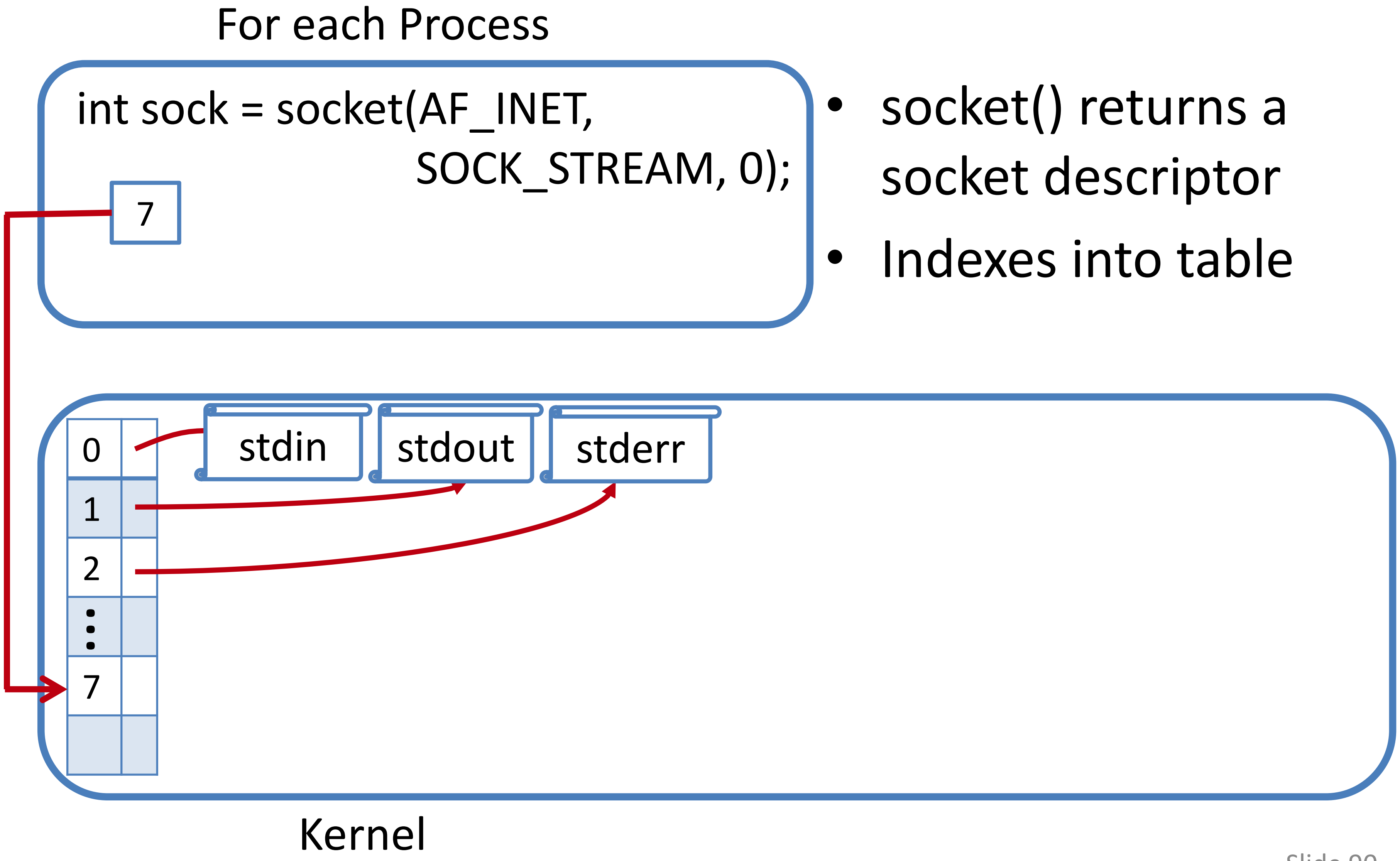
Descriptor Table



OS stores a table, per process, of descriptors

http://www.learnlinux.org.za/courses/build/shell-scripting/ch01s04.html

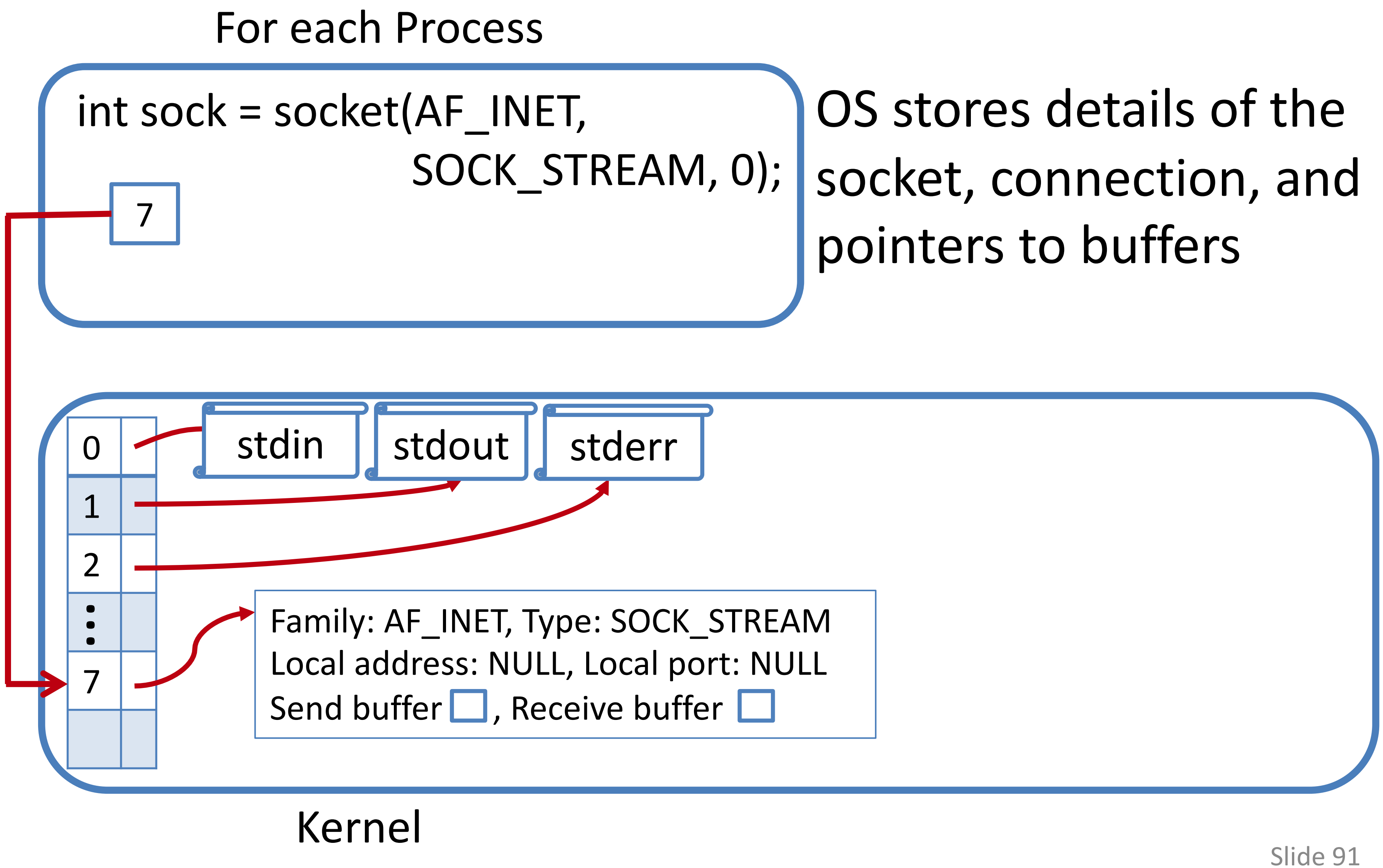
socket()



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

- socket() returns a socket descriptor
- Indexes into table

socket()



For each Process

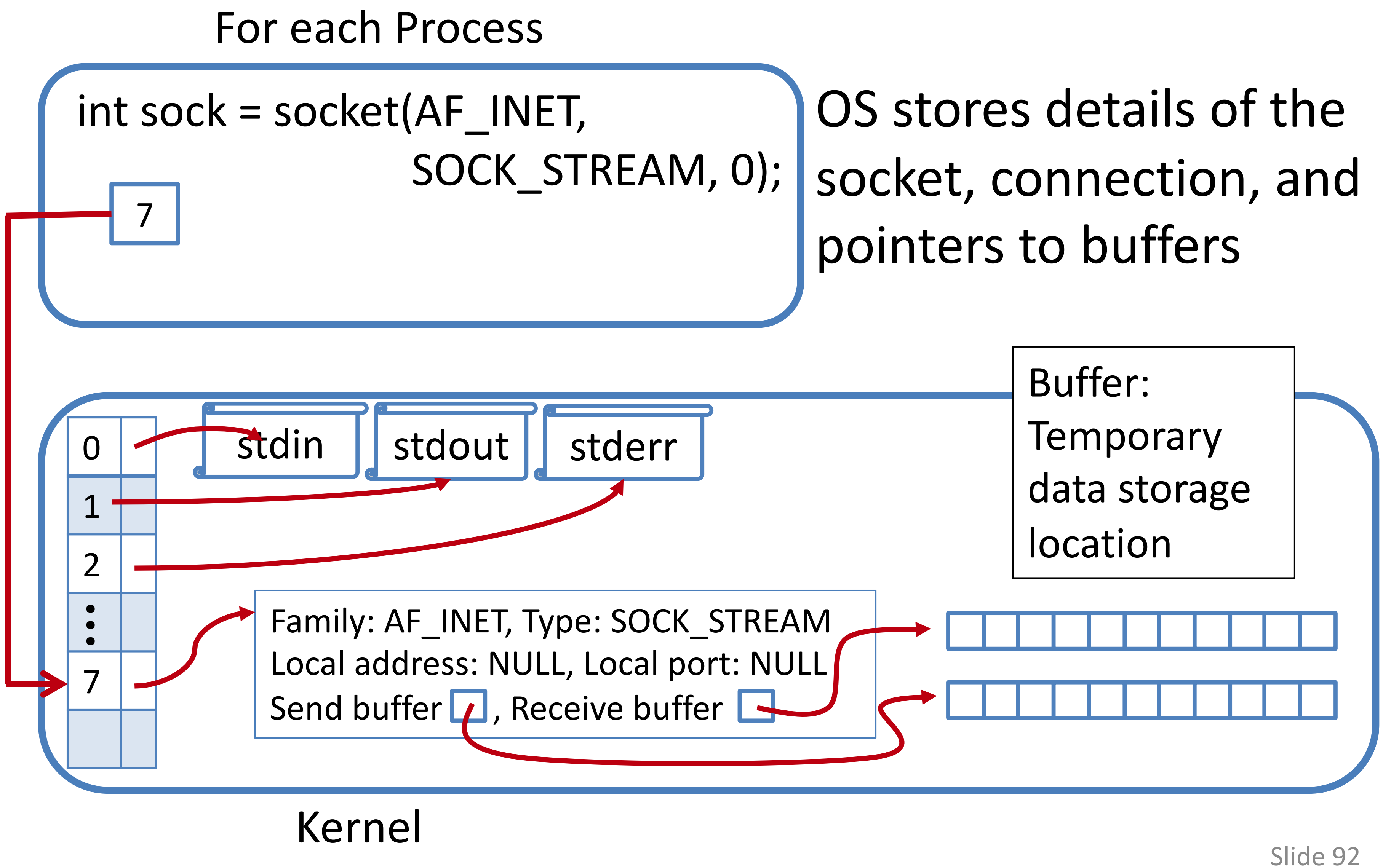
```
int sock = socket(AF_INET,
                  SOCK_STREAM, 0);
```

OS stores details of the socket, connection, and pointers to buffers

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

Kernel

socket()



For each Process

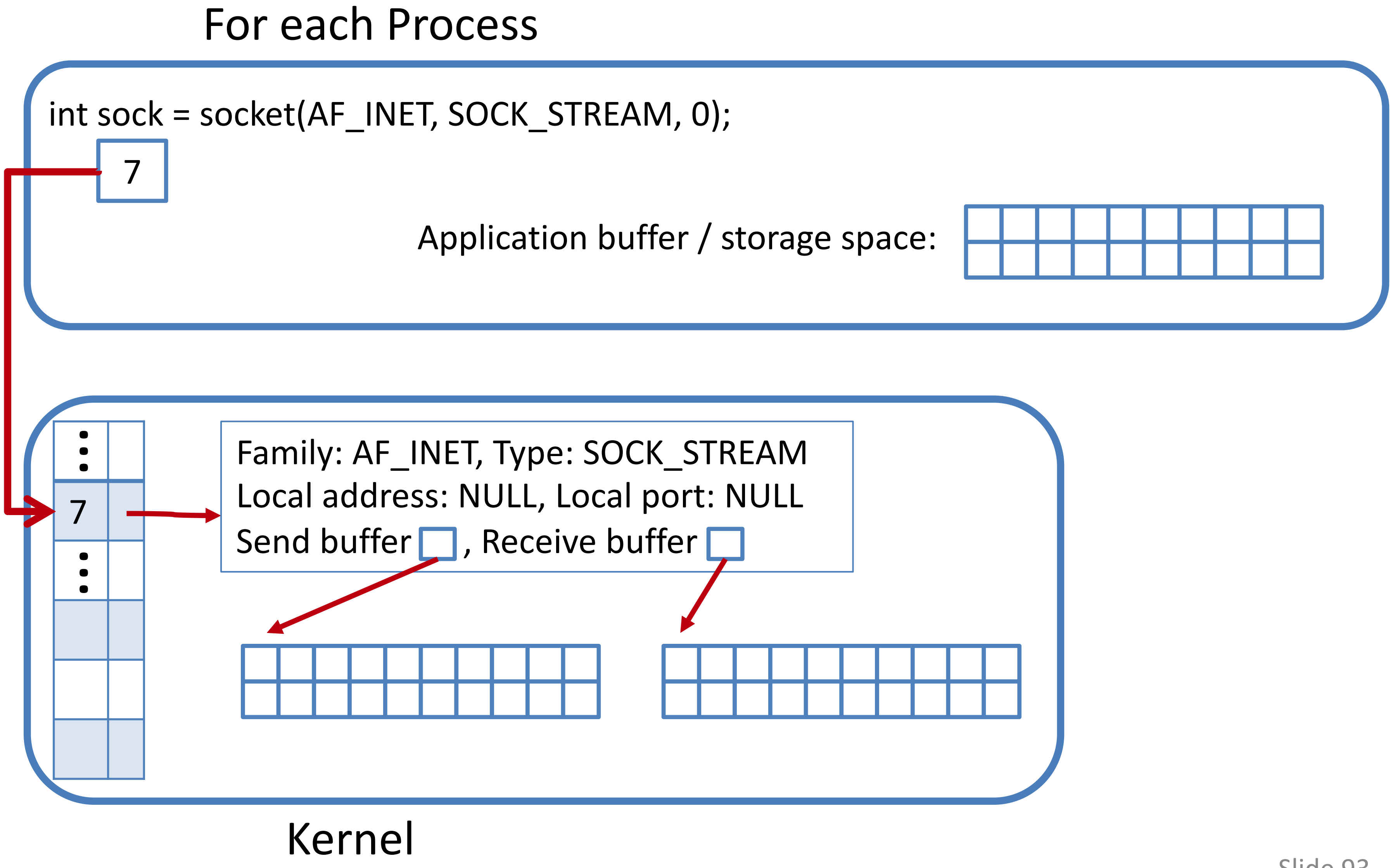
```
int sock = socket(AF_INET,
                  SOCK_STREAM, 0);
```

OS stores details of the socket, connection, and pointers to buffers

Buffer: Temporary data storage location

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

Socket Buffers



For each Process

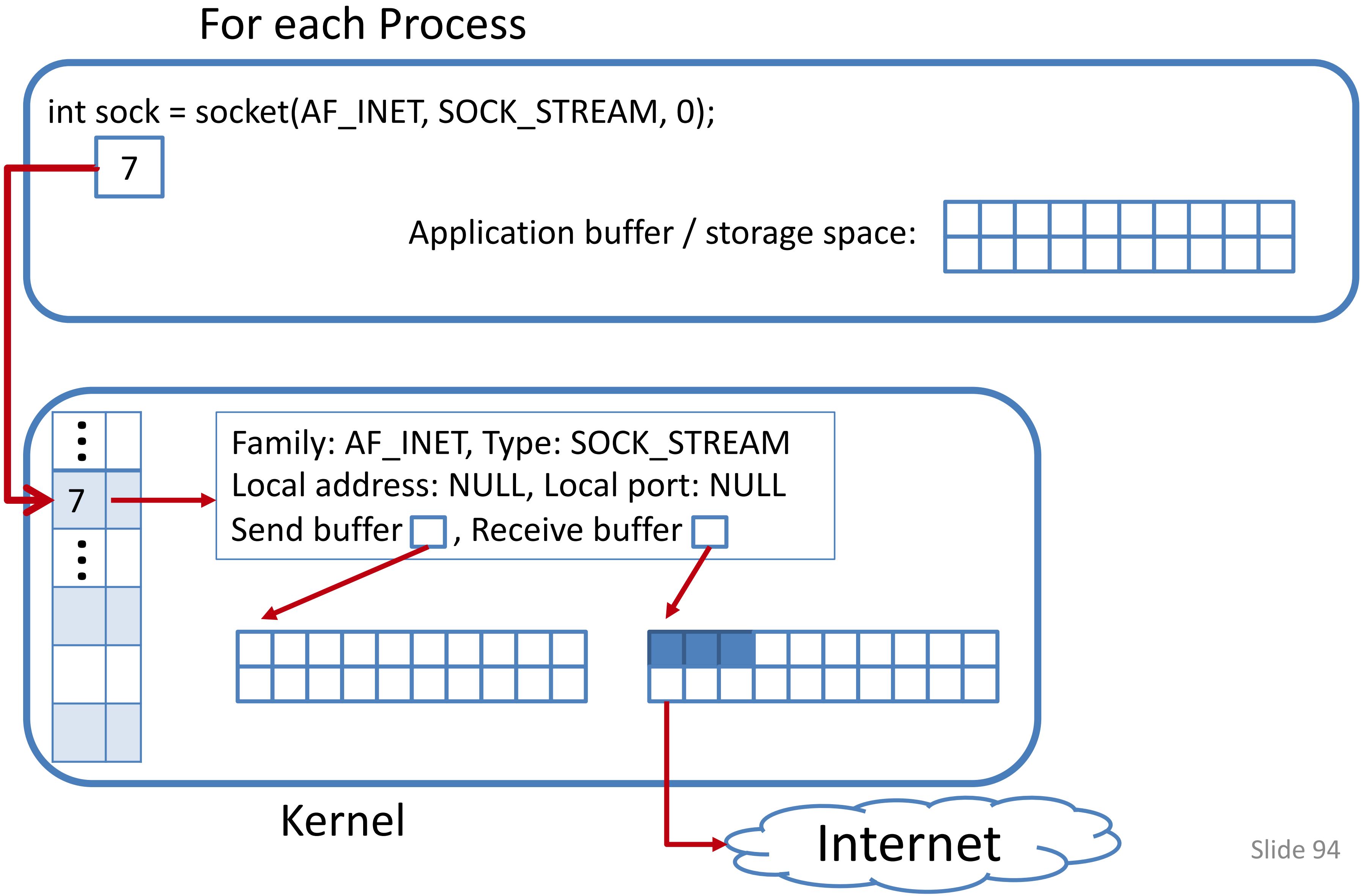
int sock = socket(AF_INET, SOCK_STREAM, 0);

Application buffer / storage space:

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

Kernel

Socket Buffers



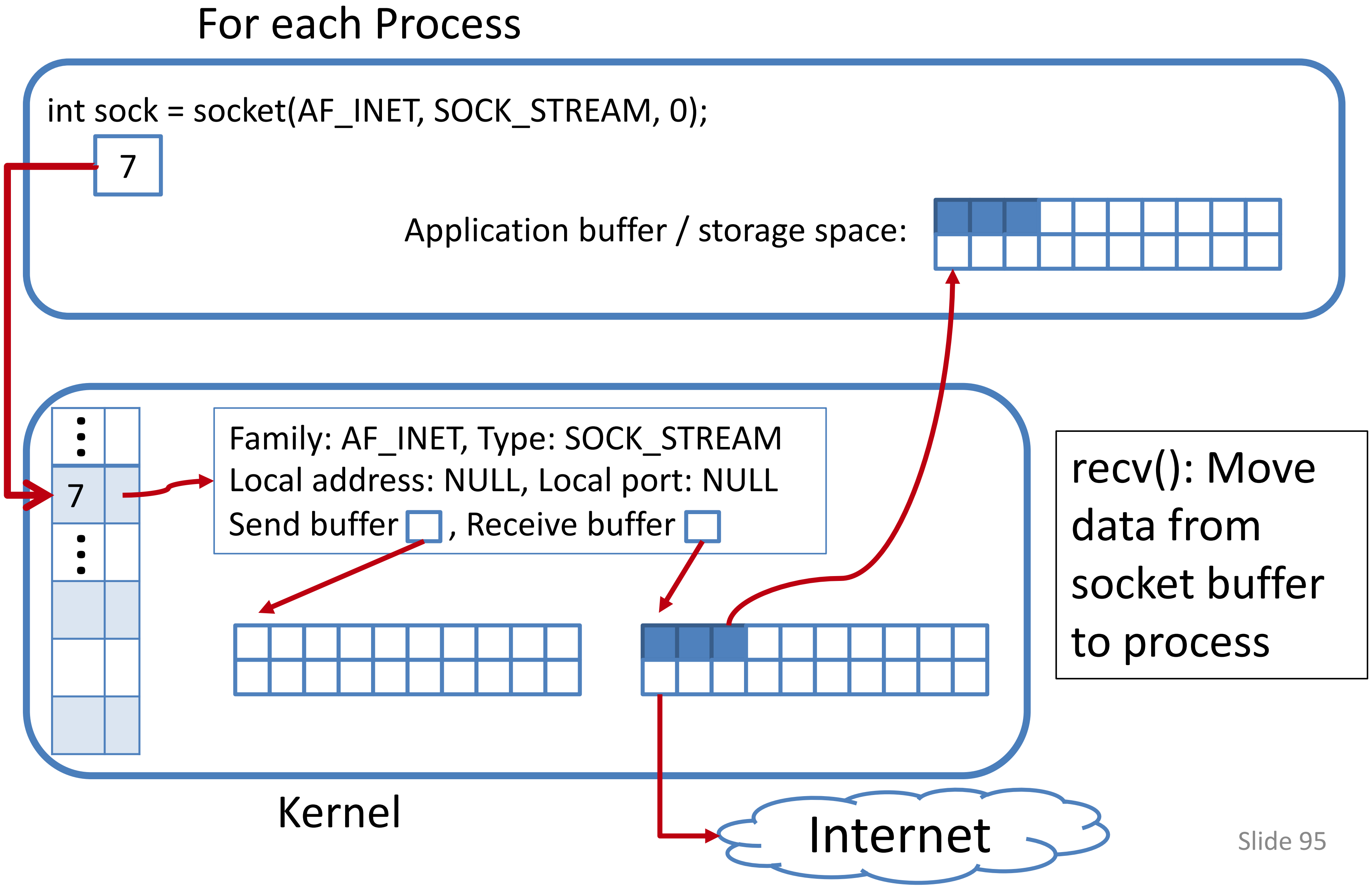
For each Process

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

Application buffer / storage space:

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

Socket Buffers



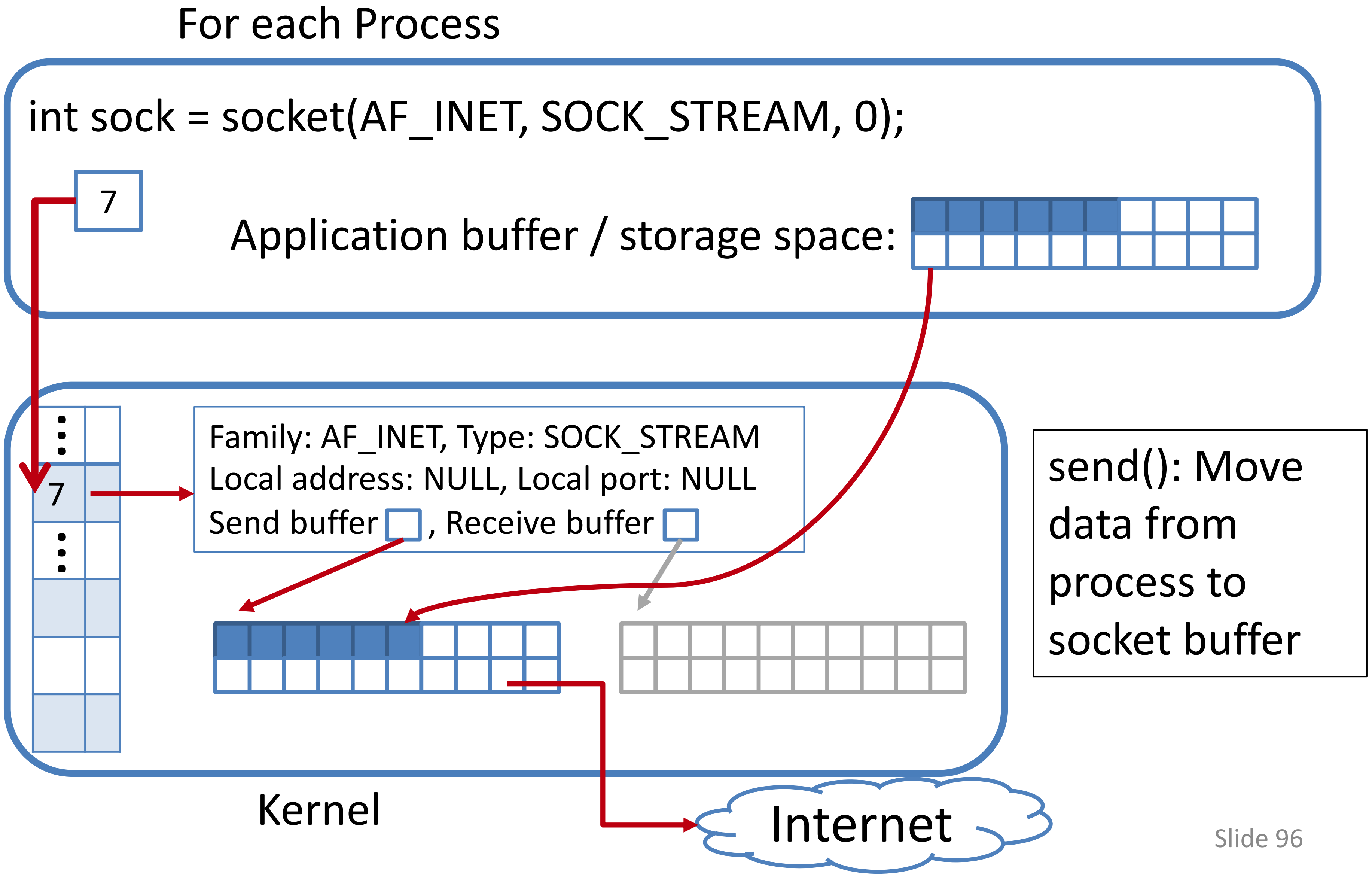
For each Process

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

7
Application buffer / storage space:
7
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □
Kernel
Internet

recv(): Move data from socket buffer to process

Socket Buffers



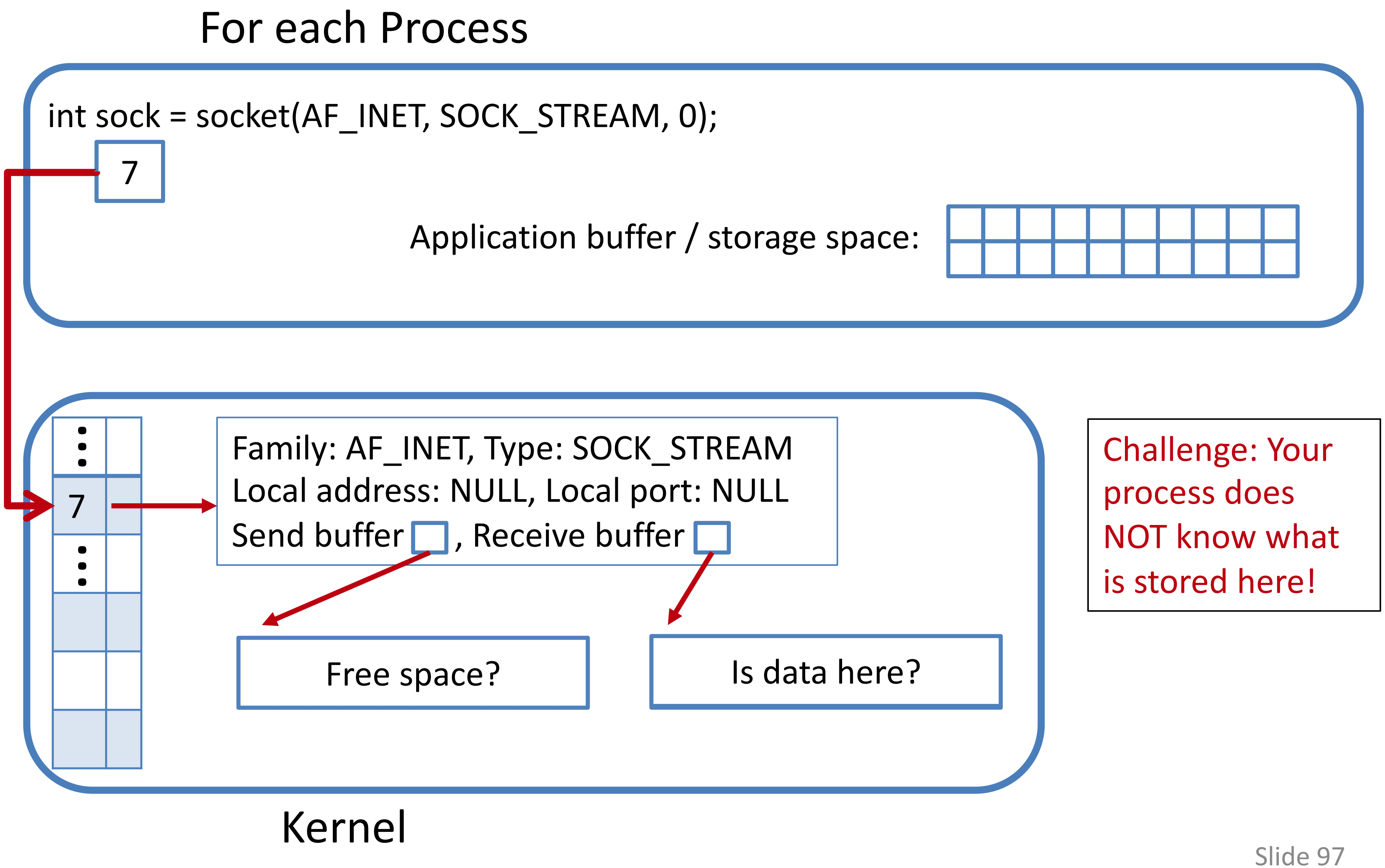
For each Process

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

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

send(): Move data from process to socket buffer

Socket Buffers



For each Process

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

Application buffer / storage space:

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer □, Receive buffer □

Free space?

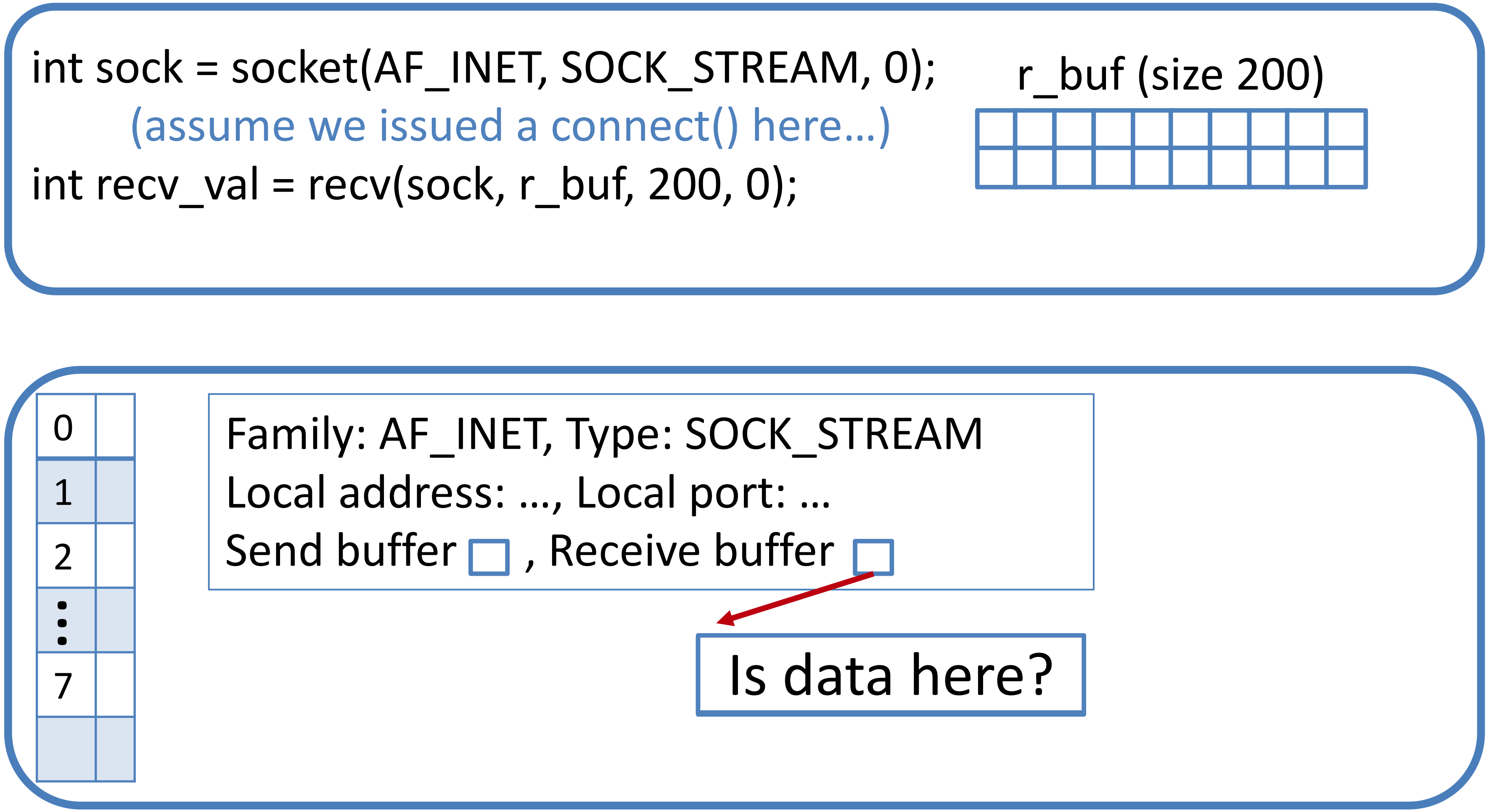
Is data here?

Kernel

Challenge: Your process does NOT know what is stored here!

recv()

For each Process



```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we issued a connect() here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

Family: AF_INET, Type: SOCK_STREAM

Local address: …, Local port: …

Send buffer □ , Receive buffer □

Is data here?

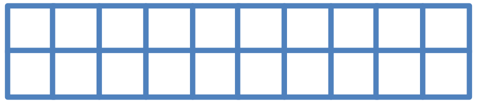
Kernel

What should we do if the receive socket buffer is empty? If it has 100 bytes?

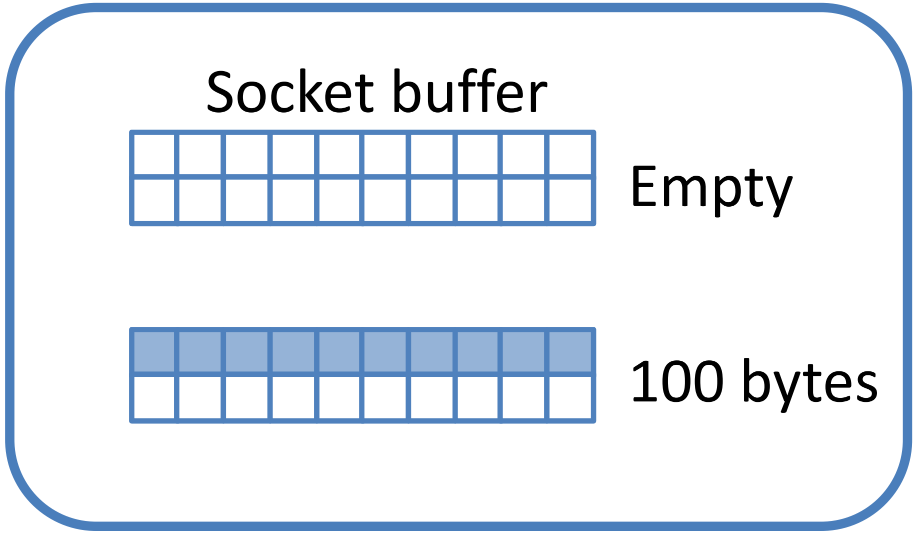
For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:



Socket buffer

Empty

100 bytes

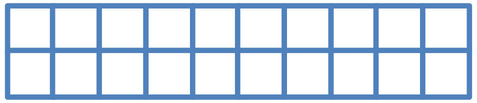
Kernel

What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

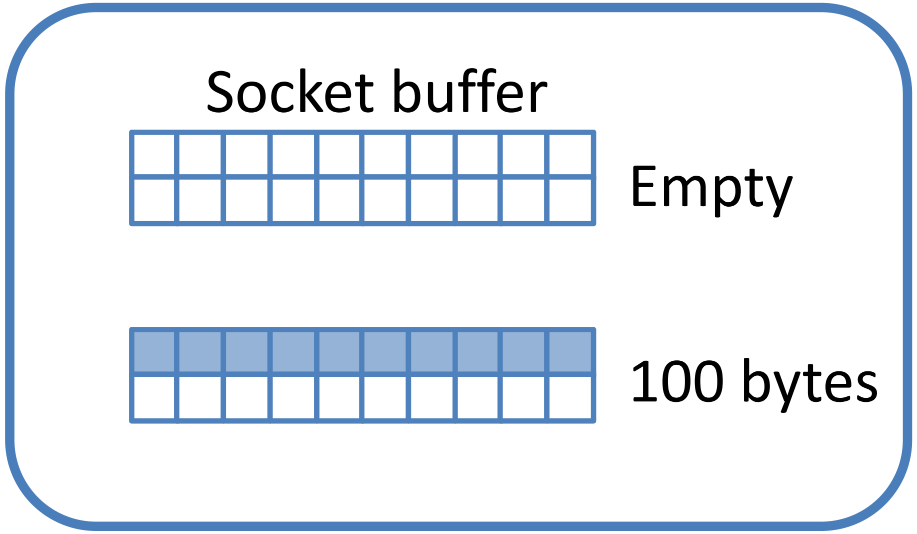
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

	Empty	100 Bytes
A	**Block**	**Block**
B	**Block**	**Copy 100 bytes**
C	**Copy 0 bytes**	**Block**
D	**Copy 0 bytes**	**Copy 100 bytes**
E	**Something else**	



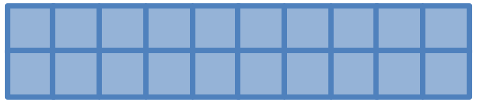
Kernel

What should we do if the send socket buffer is full? If it has 100 bytes?

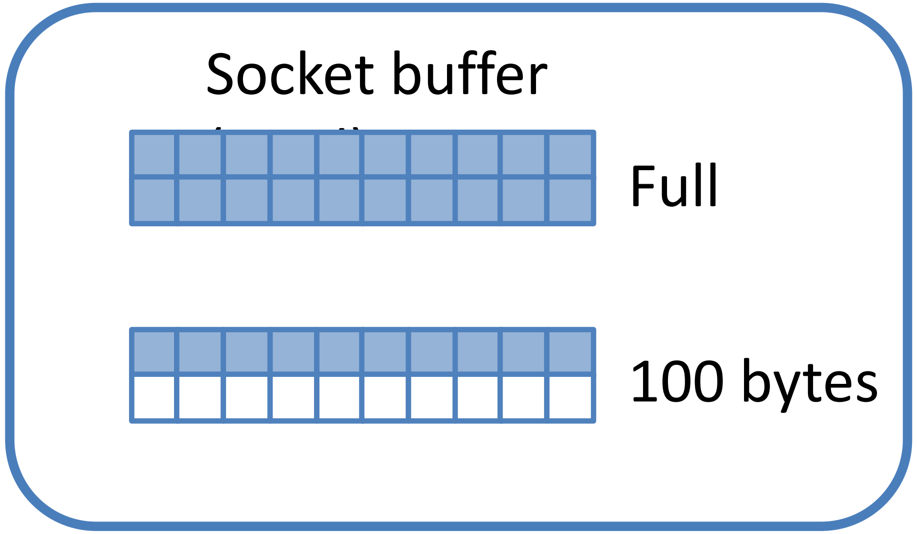
For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

s_buf (size 200)



Two Scenarios:



Socket buffer

Full

100 bytes

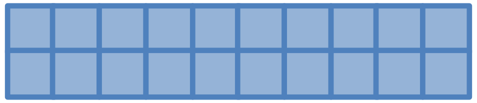
Kernel

What should we do if the send socket buffer is full?
If it has 100 bytes?

For each Process

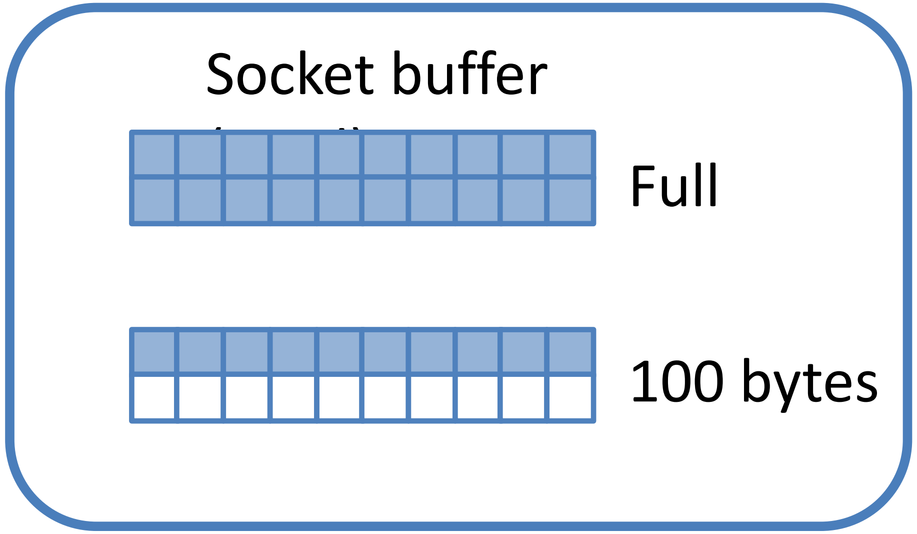
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
          (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

s_buf (size 200)



Two Scenarios:

	Full	100 Bytes
A	Return 0	Copy 100 bytes
B	Block	Copy 100 bytes
C	Return 0	Block
D	Block	Block
E	Something else	



Kernel

Blocking Implications

recv()

- Do not assume that you will recv() all of the bytes that you ask for.
- Do not assume that you are done receiving.
- Always receive in a loop!*

send()

- Do not assume that you will send() all of the data you ask the kernel to copy.
- Keep track of where you are in the data you want to send.
- Always send in a loop!*

* Unless you're dealing with a single byte, which is rare.

ALWAYS check send()/recv() return values!

When recv() returns a non-zero number of bytes always call recv() again until:

- the server closes the socket,
- or you've received all the bytes you expect.

ALWAYS check send()/recv() return values!

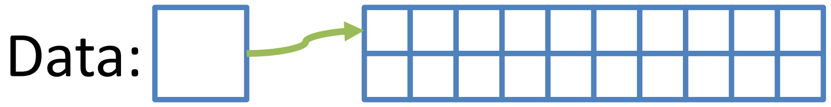
When recv() returns a non-zero number of bytes always call recv() again until:

- In the case of your web client: keep receiving until the server closes the socket.

ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);



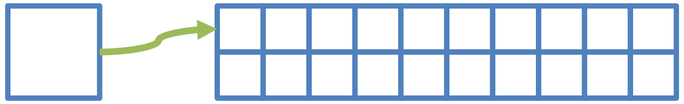
Data:

ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

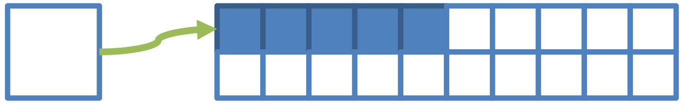
Data size to receive = unknown
recv(sock, data, 200, 0);

Data:



Data received = 50
Remaining buffer size = 150

Data:



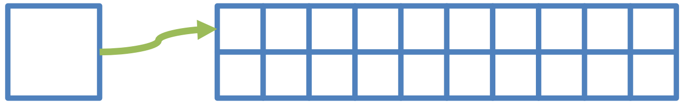
ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown

recv(sock, data, 200, 0);

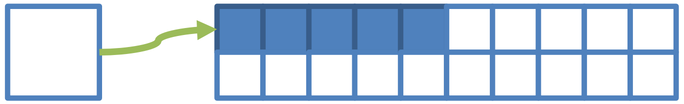
Data:



Data received = 50

Remaining buffer size = 150

Data:



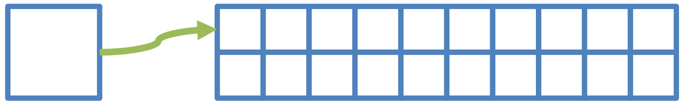
// Receive remaining bytes from offset of 50

ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

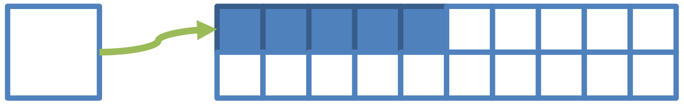
Data size to receive = unknown
recv(sock, data, 200, 0);

Data:



Data received = 50
Remaining buffer size = 150

Data:



// Receive remaining bytes from offset of 50
recv(sock, data + 50, 200 – 50, 0)
Data received = ?

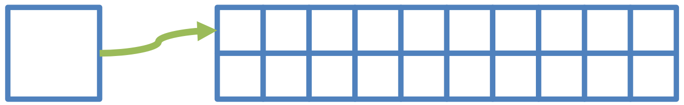
ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown

recv(sock, data, 200, 0);

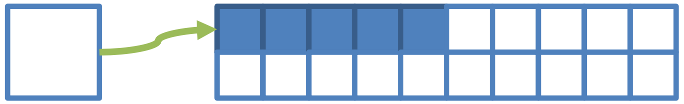
Data:



Data received = 50

Remaining buffer size = 150

Data:



// Receive remaining bytes from offset of 50

recv(sock, data + 50, 200 – 50, 0)

Data received = ?

Repeat until server closes the socket. (return value = 0)

Blocking Summary

send()

- Blocks when socket buffer for sending is full

- Returns less than requested size when buffer cannot hold full size

recv()

- Blocks when socket buffer for receiving is empty

- Returns less than requested size when buffer has less than full size

Always check the return value!