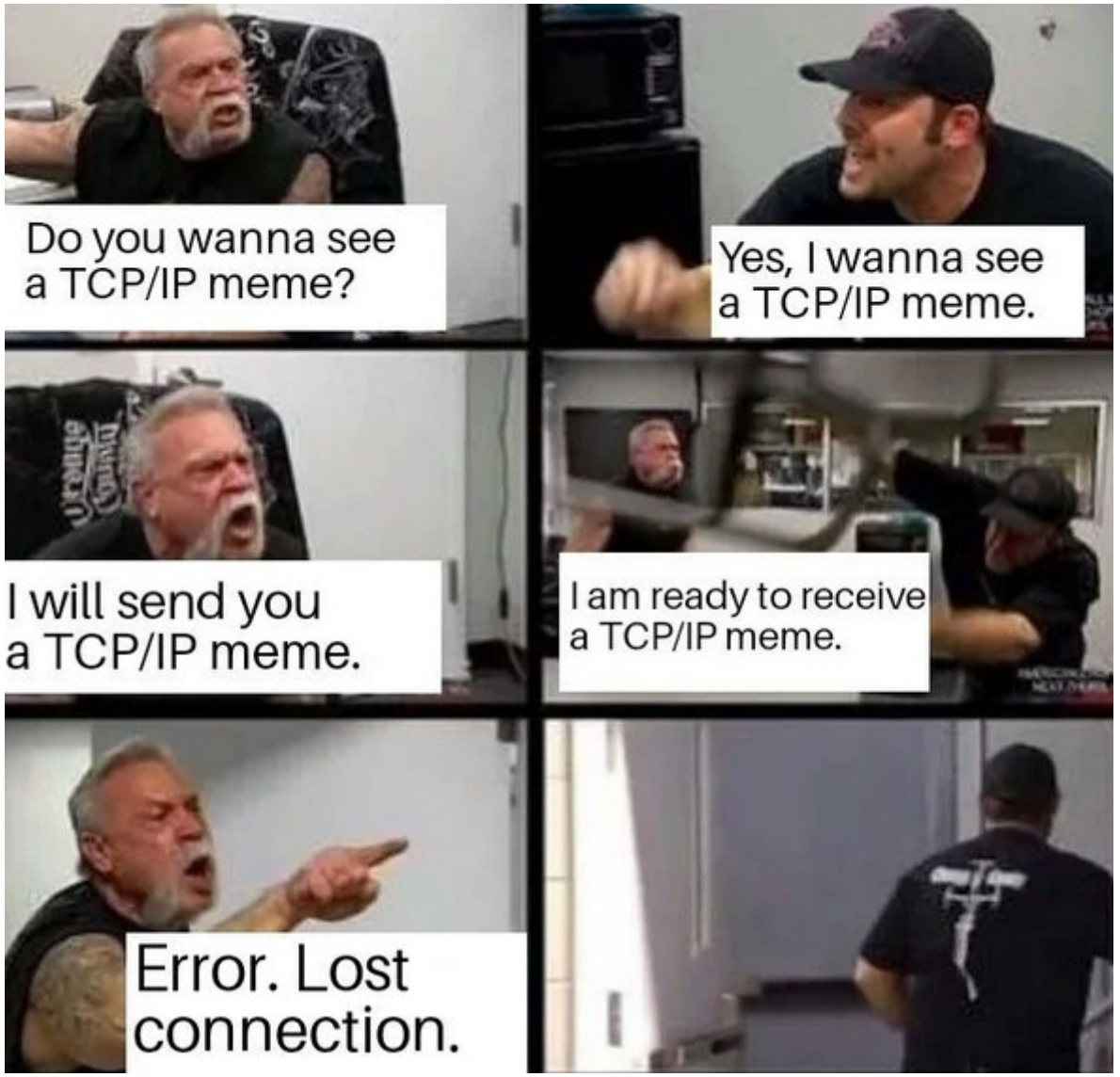
Networking: TCP/IP



Review: React, Promises, Closures

- **React** is a JavaScript framework that lets you build reusable *components* that are *mounted* into the DOM tree
  - React components can be composed together
  - Components can contain state that gets transferred to and from a remote REST API (as in Project 3)
- **Promises** are objects created immediately to handle asynchronous events that will occur later
  - Fetch() -> “JavaScript promises to send the request and call something later”
- **Closures** are objects that contain the *lexical environment* in which a function was defined so that it can access everything that was in scope when it was created

Promises

- a Promise is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected
  - *fulfilled*: meaning that the operation completed successfully
  - *rejected*: meaning that the operation failed

- If the executor function succeeds, then the method provided by `.then()` runs
- If the executor function fails, then the method provided by `.catch()` runs

Creating Promises

```
let p = new Promise((resolve, reject) => {

   // do some asynchronous work
   // in "pending" state

   // call reject if there's an error
   if (error happens) {
      // enter "rejected" state
      reject("Error");
   }

   // call resolve when promise complete
   // enter "fulfilled" state
   resolve("All finished");

});
```

Then you can call

```
p.then( otherStuff);
```

Promises

- Control the flow of deferred and asynchronous operations
- First class representation of a value that may be made asynchronously and be available in the future
- Added to JavaScript in ES6

- Examples of values that will be available in the future
  - The response to a server request: `fetch()`
  - The data from parsing a JSON string: `json()`

Using a `Promise`

- `fetch()` returns a `Promise`
- `response.json()` returns a `Promise`

```
function showUser() {
  function handleResponse(response) {
    return response.json();
  }

  function handleData(data) {
    console.log(data);
  }

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
```

Using a `Promise`

- After the value is available, the `Promise` calls a function provided by `.then()`

```
function showUser() {
  function handleResponse(response) {
    return response.json();
  }

  function handleData(data) {
    console.log(data);
  }

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
```

Promises explained again

- Functions performing asynchronous tasks return a `Promise`
- A `Promise` is an object to which you can attach a callback
  - Using `.then()`

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```

Promise states

- A `Promise` is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected
  - *fulfilled*: meaning that the operation completed successfully
  - *rejected*: meaning that the operation failed

- On success, the method provided by `.then()` runs

Chaining promises

- A common need is to execute two or more asynchronous operations back-to-back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.
- Example:
  1. Request
  2. Parse JSON
- We accomplish this by creating a *promise chain*

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```

Error handling

- We can also provide a callback for handling a errors
- A Promise will call one of the two callbacks provided by
  - `.then()`
  - `.catch()`

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

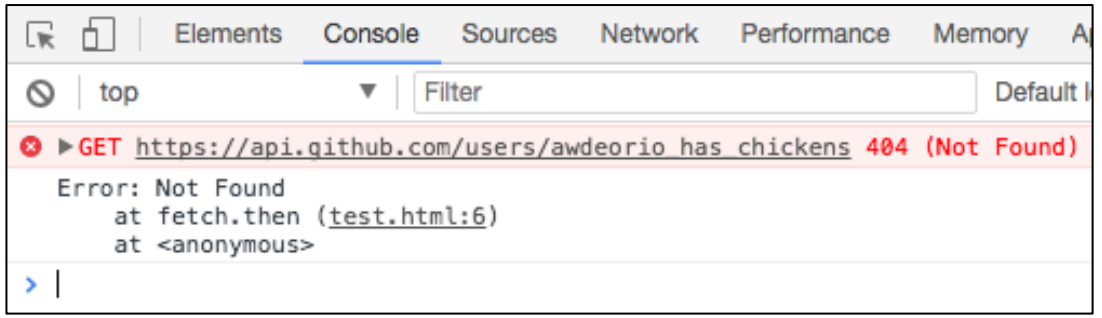
Error example

- REST APIs typically return errors in JSON format instead of HTML

```
$ http https://api.github.com/users/awdeorio_has_chickens
HTTP/1.1 404 Not Found

{
  "documentation_url": "https://developer.github.com/v3/users/#get-a-single-user",
  "message": "Not Found"
}
```

Error propagation



- A promise chain stops if there's an exception, looking down the chain for catch handlers instead
- REST API returned 4xx will trigger error
- Similar to `try/catch` in synchronous code

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio_has_chickens')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

Exercise

- What is the output? How long does this program take?

```
function main() {
  wait(1000).then(() => console.log('1 s passed'));
  wait(0).then(() => console.log('0 s passed'));
  wait(500).then(() => console.log('0.5 s passed'));
}

main();
```

Solution

- What is the output? How long does this program take?

```
function main() {
  wait(1000).then(() => console.log('1 s passed'));
  wait(0).then(() => console.log('0 s passed'));
  wait(500).then(() => console.log('0.5 s passed'));
}

main();
```

Output
0 s passed
0.5 s passed
1 s passed

Runtime
1.0s

Exercise

- What is the output? How long does this program take?

```
function main() {
  wait(1000)
  .then(() => {
    console.log('1 s passed');
    return wait(0);
  })
  .then(() => {
    console.log('0 s passed');
    return wait(500);
  })
  .then(() => console.log('0.5 s passed'));
}
main();
```

Solution

- What is the output? How long does this program take?

```
function main() {
  wait(1000)
  .then(() => {
    console.log('1 s passed');
    return wait(0);
  })
  .then(() => {
    console.log('0 s passed');
    return wait(500);
  })
  .then(() => console.log('0.5 s passed'));
}
main();
```

Output

```
1 s passed
0 s passed
0.5 s passed
```

Runtime

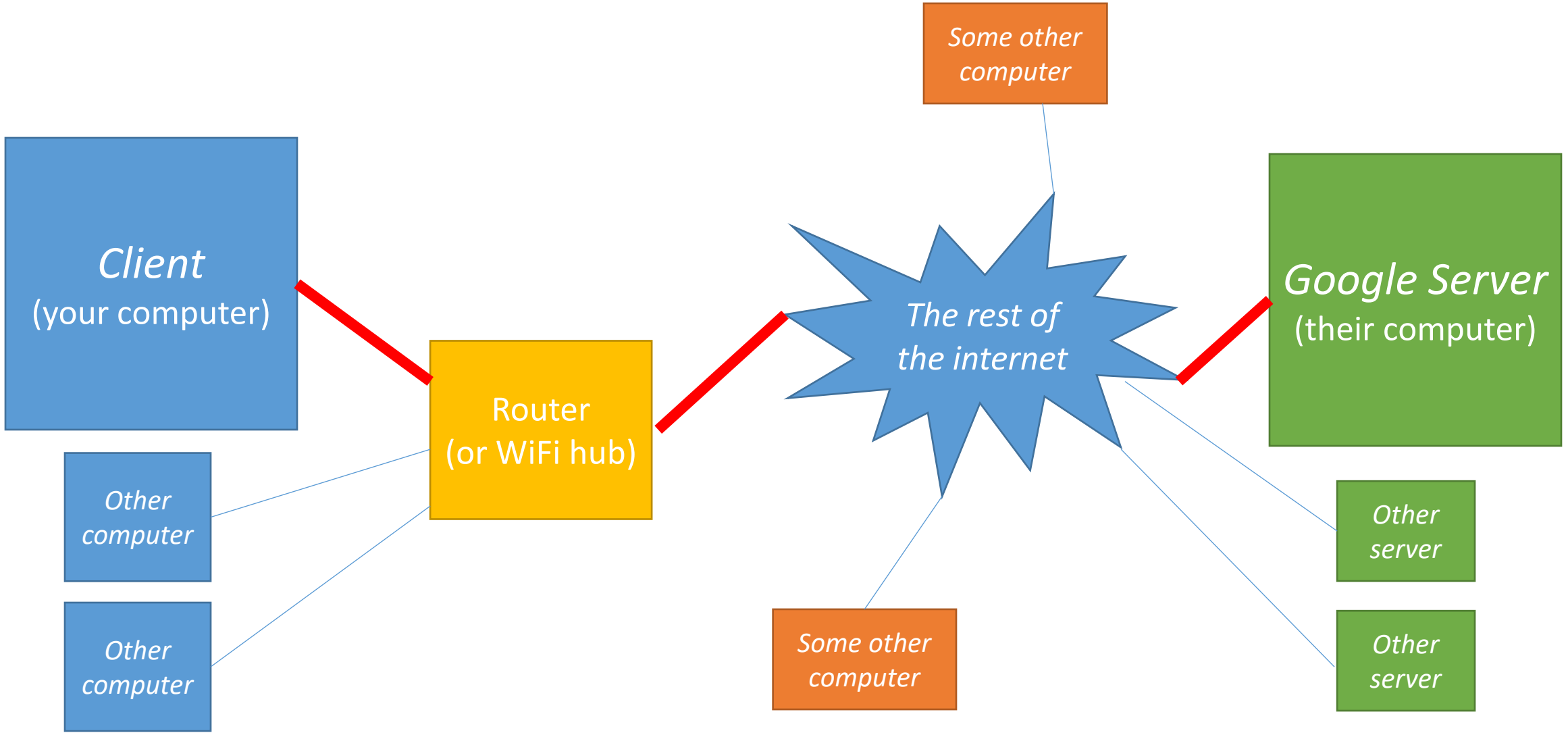
```
1.5s
```

One-Slide Summary: Networking

- A **network** is a communication medium for exchanging information between computers
- Your computer has a **network interface** that is uses to exchange data with other computers on a network
- The **Internet** is a worldwide network supported by various companies and governments

- We use the **Internet Protocol** to assign each computer an **IP address** that can be used to send information to that computer
  - IP addresses are used to route **packets** of information from one computer to another
- We use the **Transmission Control Protocol** and **User Datagram Protocol** to exchange packets over a network
  - We refer to TCP/IP as an implementation of TCP on top of IP
- Then, other **application-level protocols** (like HTTP) can then be transmitted using TCP
- We use the **Domain Name Service** to map *human-readable domain names* to *IP addresses*

Network: How do computers talk to each other?



Networked communication

- Your computer has a **network interface**
  - A **network interface card (NIC)** – this can be wireless!
- The operating system lets programs use the NIC to communicate
- See: socket.h <- header for C functions that use the network

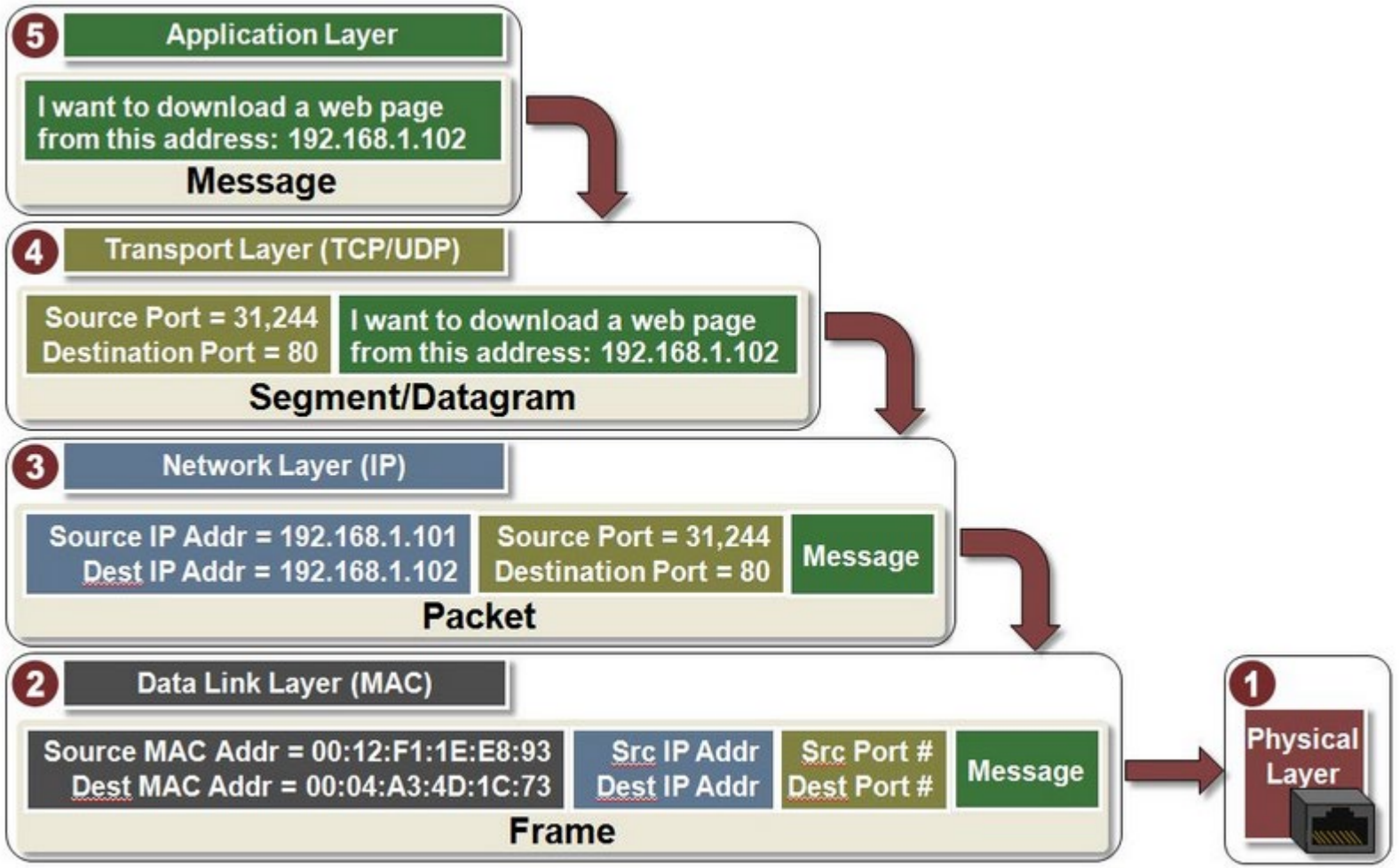
```
kleach@DESKTOP-3QOH0CH:~/eecs485/l9$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.50.[redacted]  netmask 255.255.255.0  broadcast 192.168.50.255
        inet6 fe80::c4a2:45f4:[redacted]  prefixlen 64  scopeid 0xfd<compat,link,site,
        ether 38:d5:47:[redacted]  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Network Layers

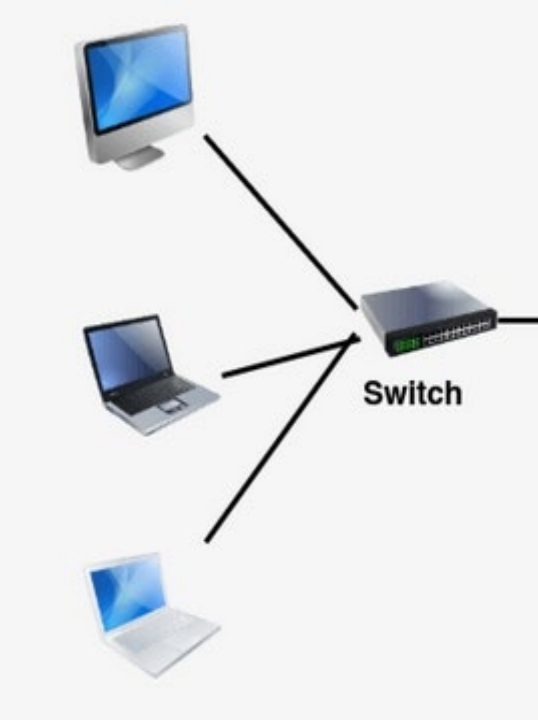
- Networking is built in layers
  - Separation of concerns
  - Build one layer to provide services/guarantees to the next
- Physical layer lets you send bits from one interface to another
  - Once you have that, you can build up frames on top of individual bits
    - Then, build packets on top of frames to hop over multiple interfaces
      - Etc…

Layer #	Layer Name	Protocol	Protocol Data Unit	Addressing
5	Application	HTTP, SMTP, etc…	Messages	n/a
4	Transport	TCP/UDP	Segments/ Datagrams	Port #s
3	Network or Internet	IP	Packets	IP Address
2	Data Link	Ethernet, Wi-Fi	Frames	MAC Address
1	Physical	10 Base T, 802.11	Bits	n/a

Network Layers



Physical Layer, Media Access Control



- (out of scope for this class)
- **Physical layer** – basically use physics to put bits on wires between computers

- **Media Access Control Layer** – each interface has a physical address called a MAC address used to signal between interfaces
  - e.g., if you use WiFi, how does your computer know if a message sent over the radio is intended for it?

- Physical + MAC: you can now send messages between interfaces (*not necessarily between computers*)

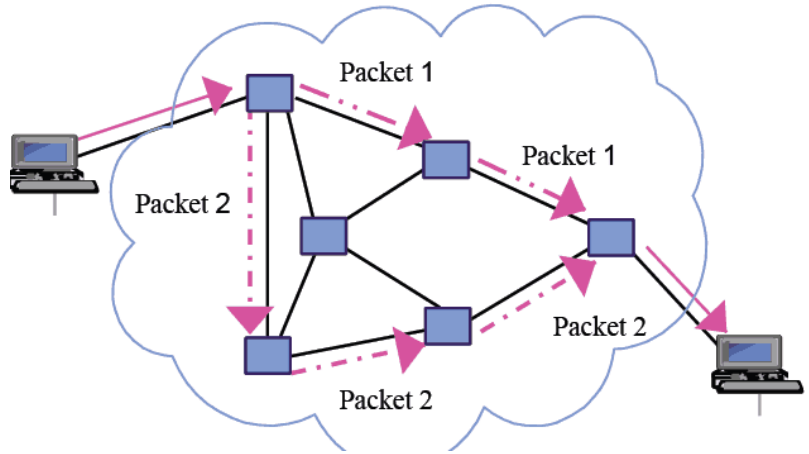
Internet Protocol (IP)

- How does information travel to the right place?

- Every computer on the Internet has an address
  - Google 172.217.5.14
  - Amazon 205.251.242.103
  - My laptop 141.212.109.1

- Newer version: longer addresses
  - IPv4: 32 bits is 4 billion computers
  - IPv6: 64 bits is way more

IP, Packets, and Switching



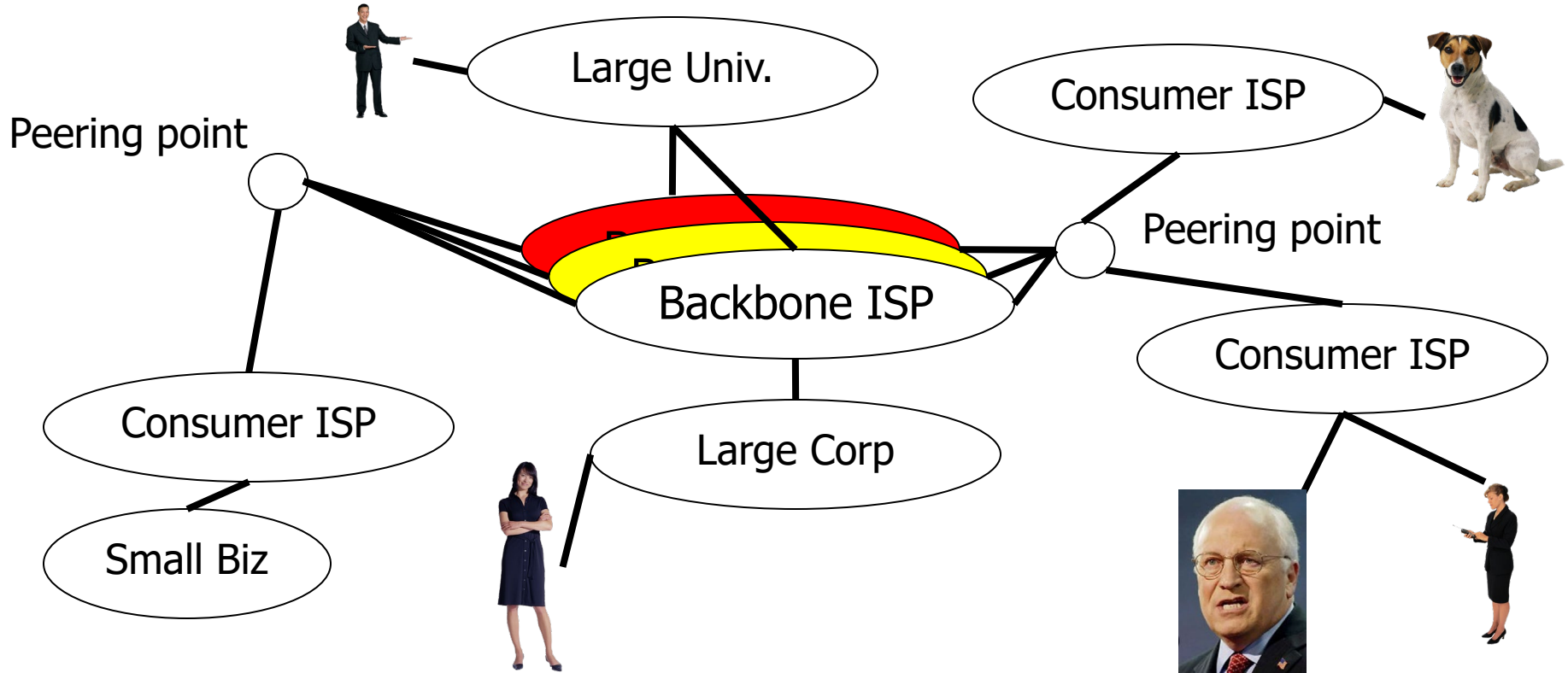
- Each message is broken into a sequence of **packets**
  - **Packets** are sent to a destination **IP address**
  - A **packet** is a ~1500 byte (max) message that contains
    - Source IP (for return communication)
    - Destination IP
    - Some other metadata
    - The **payload**

- Individual packets can take *any route* to the destination
  - Even if multiple packets are part of the same message

- Thought question
  - Why limit the packet size?

The internet is packet switched

- Internet is a best-effort, packet-switched network
  - Basic unit is packet, sent by hosts
  - Packets may arrive late, or not at all
  - IP routers form the core of the Internet



Transmission Control Protocol (TCP)

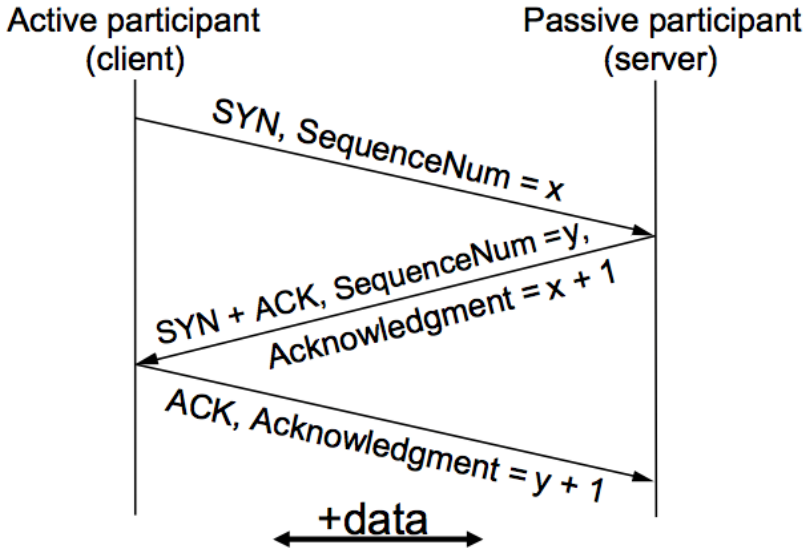
- Problem
  - Packets can arrive out of order
    (e.g., if one packet takes a longer route than another)
  - Packets can disappear
    (e.g., bad connection, lost en-route)
  - Packets can be repeated
    (e.g., faulty network hardware, or resending before a timeout)
  - Multiple networked applications may exist on one host
    (how do we share one IP amongst multiple applications?)

- TCP: abstraction to make it look like these problems don't exist
  - We *control* the *transmission* of long messages with this *protocol*
  - TCP creates a **virtual circuit** between two applications

- TCP/IP – TCP along with IP
  - This is pretty much the universal standard for all things Internet

TCP Big Idea

- Use special messages to indicate the state of communication
  - **SYN**: initial message client sends to open a connection
  - **SYN-ACK:** server responds to indicate it's ready to open a connection
  - **ACK:** client sends to confirm opening a connection
- After this *three-way handshake*, the connection begins



TCP segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

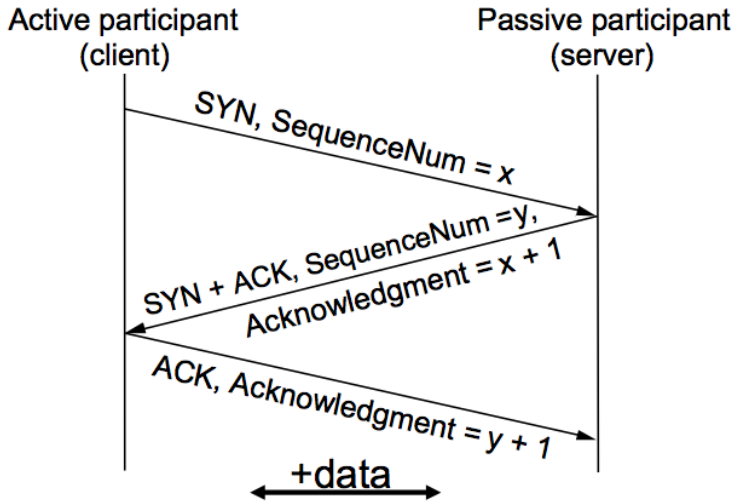
TCP: Ports

- In addition to an IP address, TCP adds the notion of **ports**
- A **port** is just a 16-bit number unique to an application
  - Port 80 for HTTP, 443 for HTTPS, 25 for SMTP, 22 for SSH, etc.
- When you make a networked application over TCP, you specify a port
  - Usually, any port over 1000 can be used as you like
    - e.g., Counter Strike uses 27015, Flask uses 5000 by default, Minecraft uses 25565
- When you open a connection, you specify a port to connect to
  - "I want to ssh to server.blah on port 22!"
- You also open a port on the client for return communication
  - SSH server: "OK, I'll send from *my* port 22 to *your* port 54932"

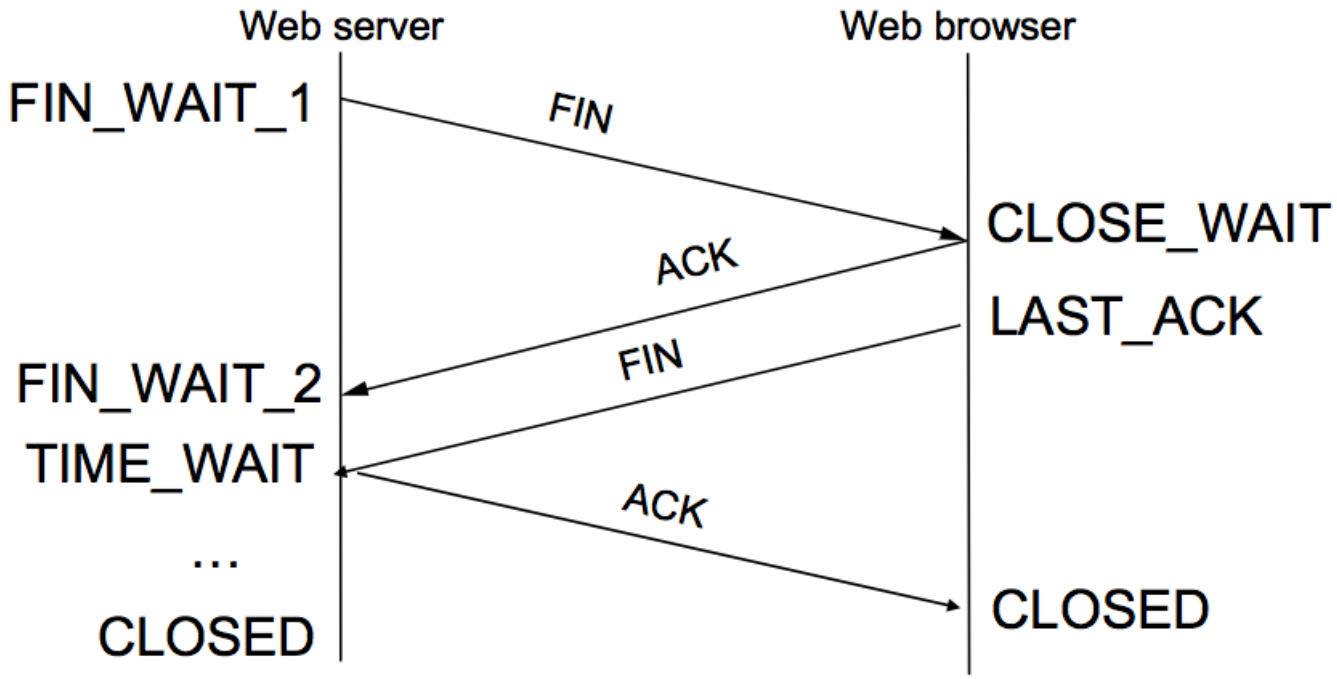
TCP: Sequencing

- Client and server exchange ACK messages (with payloads)
  - Maintain **sequence numbers** with each message
  - That way, both the client and server know how much message has been sent!
    - Consider: “I received an ACK for messages 1 and 3, but not 2 – I better resend 2!”

- Sequence numbers start as a random number (x)
  - Increment by number of bytes sent in packet



TCP connection teardown



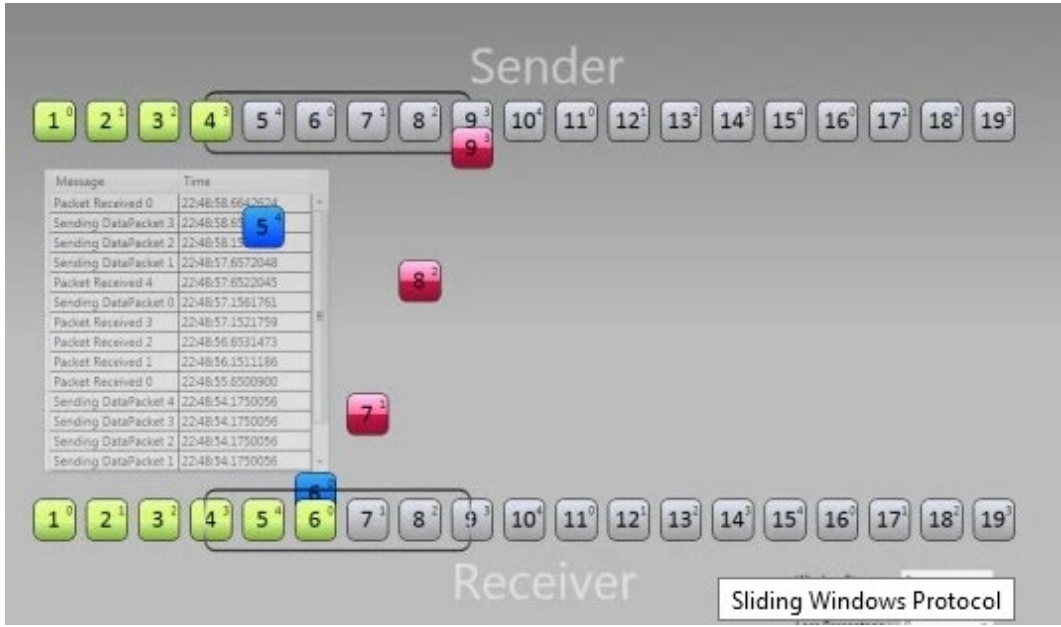
TCP and reliability

- Basic principle of reliable TCP connection is retransmission
  - Each packet has 32-bit Sequence Number
  - Every SeqNo is ACKed by receiver
  - When timeout expires, sender retransmits

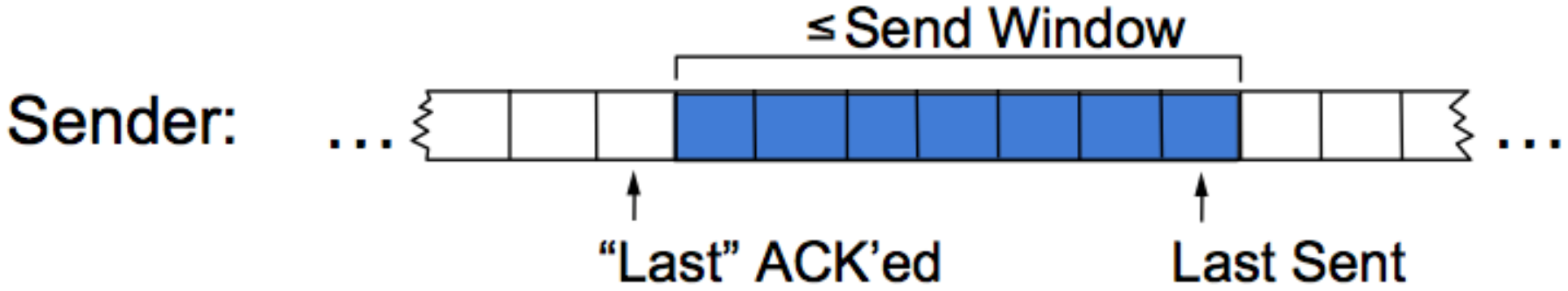
- Sometimes, extra retransmissions occur
  - e.g., an ACK can come *after* the original timeout
    - Two ACKs for the same packet… (sender/receiver both keep track of sequence numbers, remember?)
- Sometimes, retransmissions fail
  - Usually, just try resending X times, then assume connection was lost

Flow control

- Problem: Do we really have to wait for *every packet* to get ACK'd?
  - You pay the cost of network latency for every 1500 bytes
  - Instead: buffer, send a bunch of packets and check ACKs later
- Next problem: How fast can we send data before receiver gets overwhelmed?
- Flow control is about the **receiver**
  - Limited storage, slow network
  - Maybe you send a bunch of packets, but the receiver can't keep up!



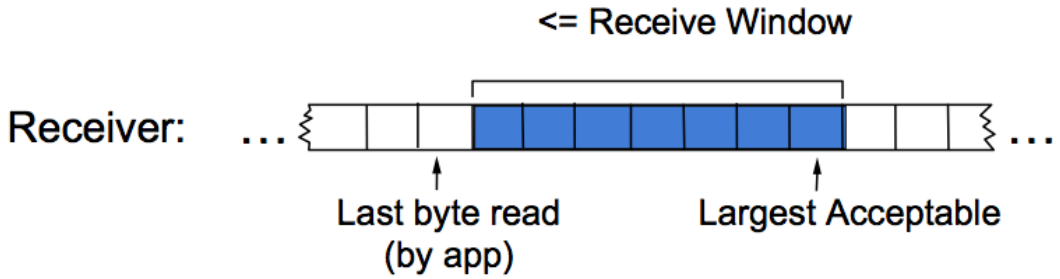
Sliding window: Sender



- Sender buffers unACKed data
- Only removes data from send buffer after it's been ACKed
- Send window determined by receiver's advertisements

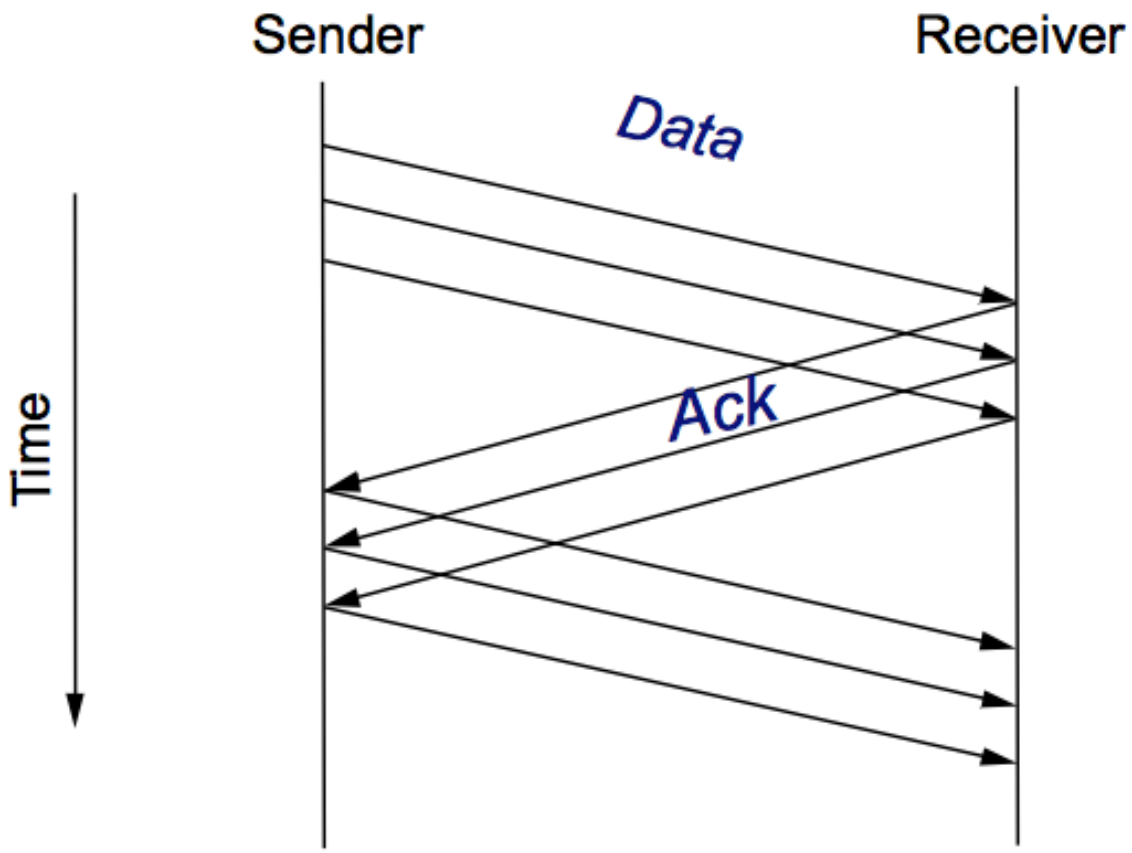
Sliding window: Receiver

- Receiver buffers data that is
  - Out-of-order
  - Not yet read off by application



- ACKs data as it arrives
- Removes data from buffer as app reads
- Shrinks/expands advertised window in response to application behavior

Sliding window



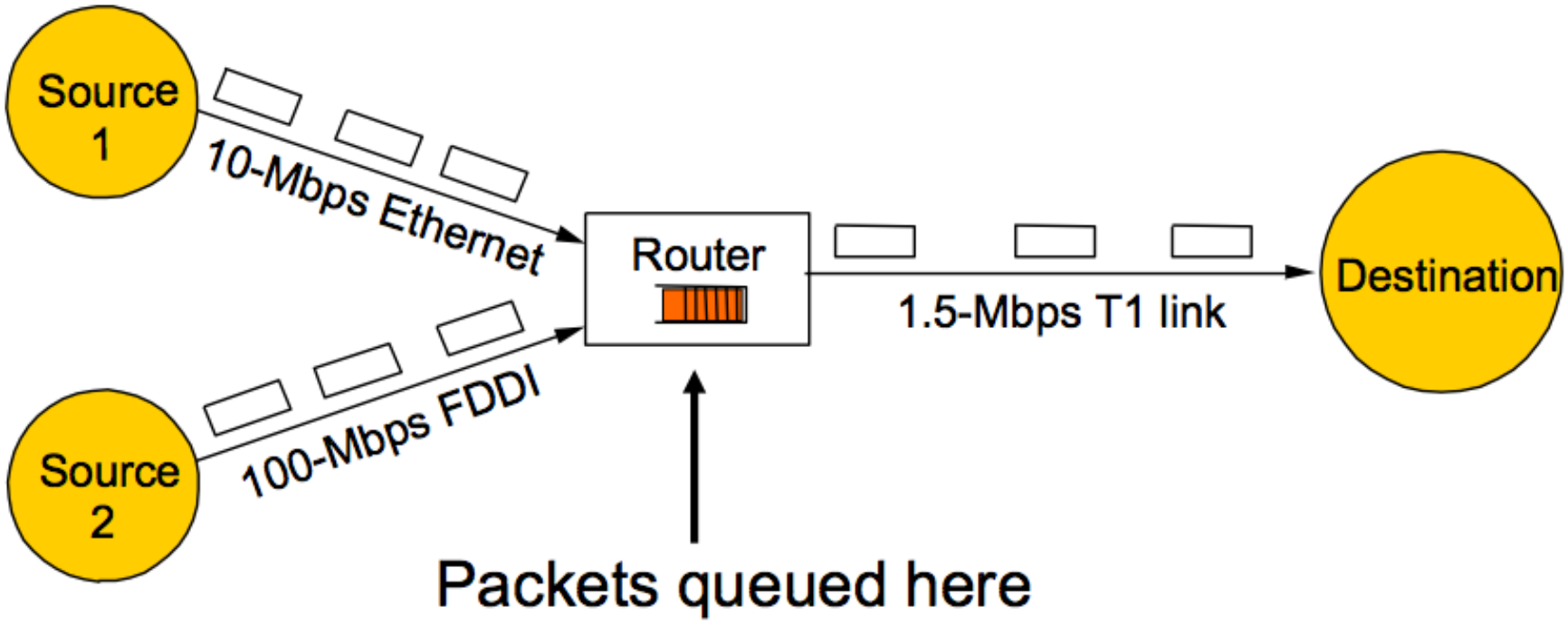
Thought question

If the sender has lots of data to send, which of these situations will occur when the sliding window is working properly?

A. The receiver's queue will usually be almost full
B. The receiver's queue will usually be about half full
C. The receiver's queue will usually be almost empty

Congestion Control

- Problem: routers get overloaded when output speed not high enough for inputs

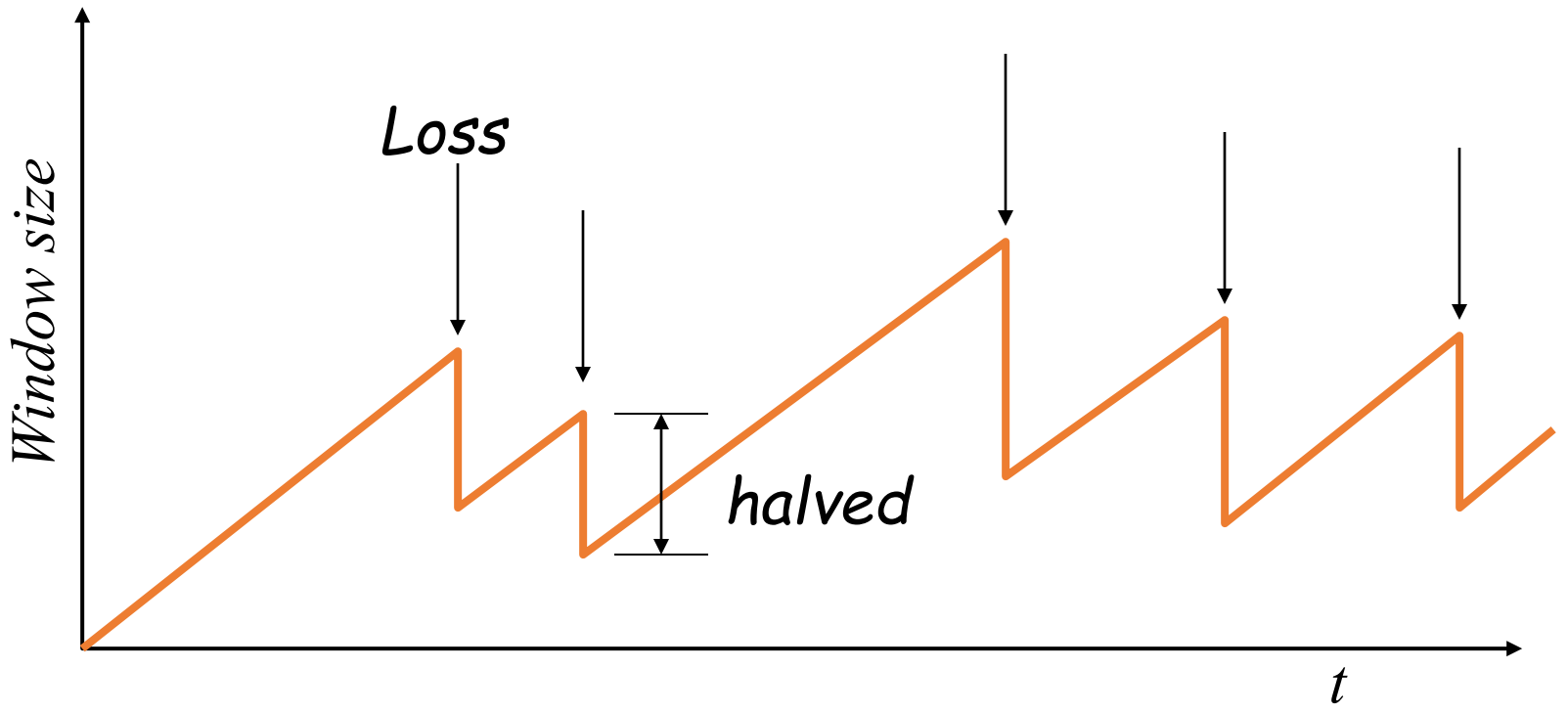


TCP congestion window

- Each TCP sender maintains a **congestion window** to manage network capacity (which changes based on congestion)
  - Maximum number of bytes to have in transit
  - I.e., number of bytes still awaiting acknowledgments

- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring

AIMD sawtooth



AIMD

- AIMD: Additive Increase Multiplicative Decrease

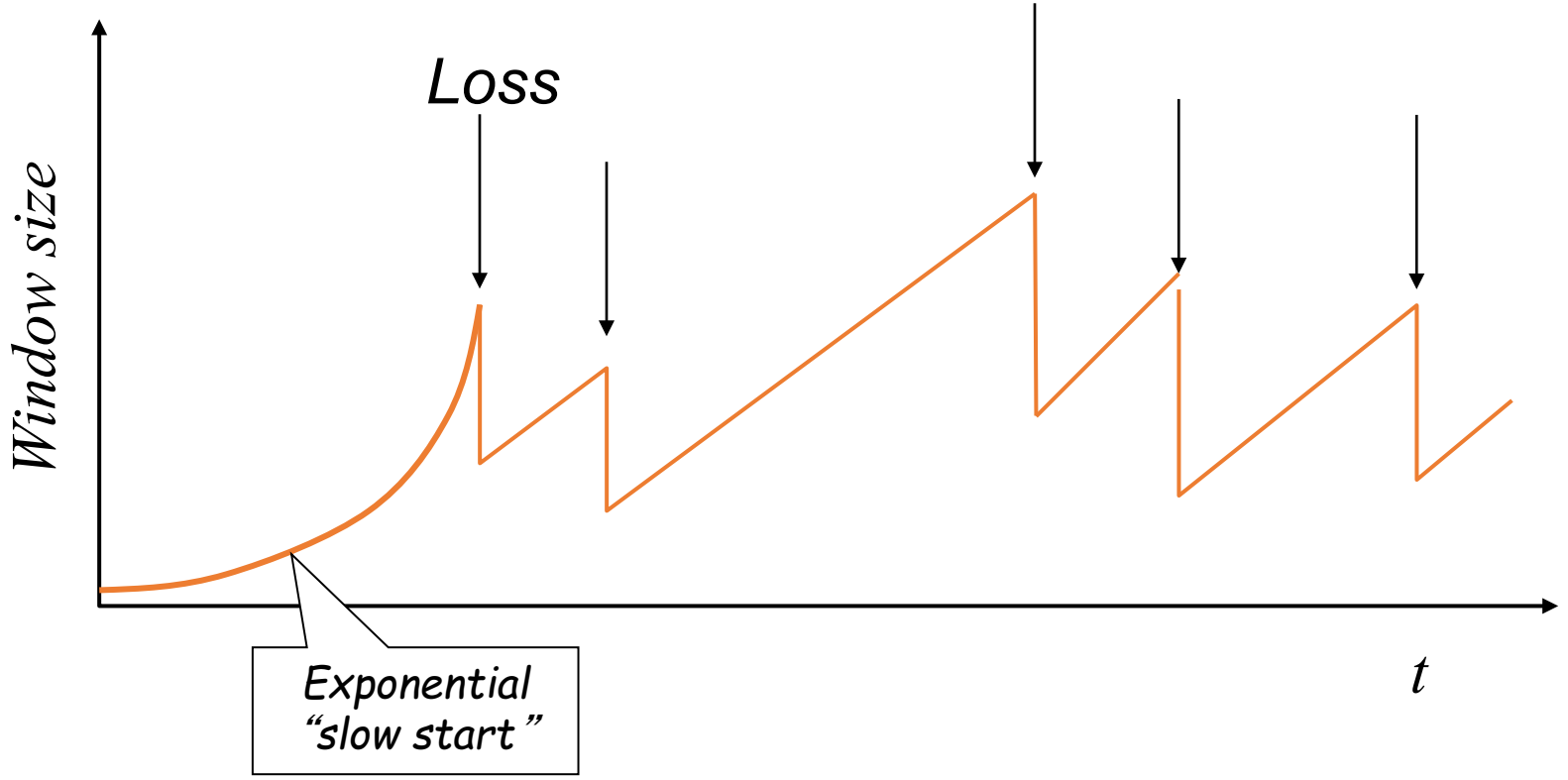
- After packet timeout `cwnd = cwnd/2`
- After packet ACK `cwnd += 1`

Flow control and congestion control

- Flow control (aka sliding window)
  - Keep a *fast sender* from overwhelming a *slow* ***receiver***
- Congestion control
  - Keep a *set of senders* from overloading the ***network***

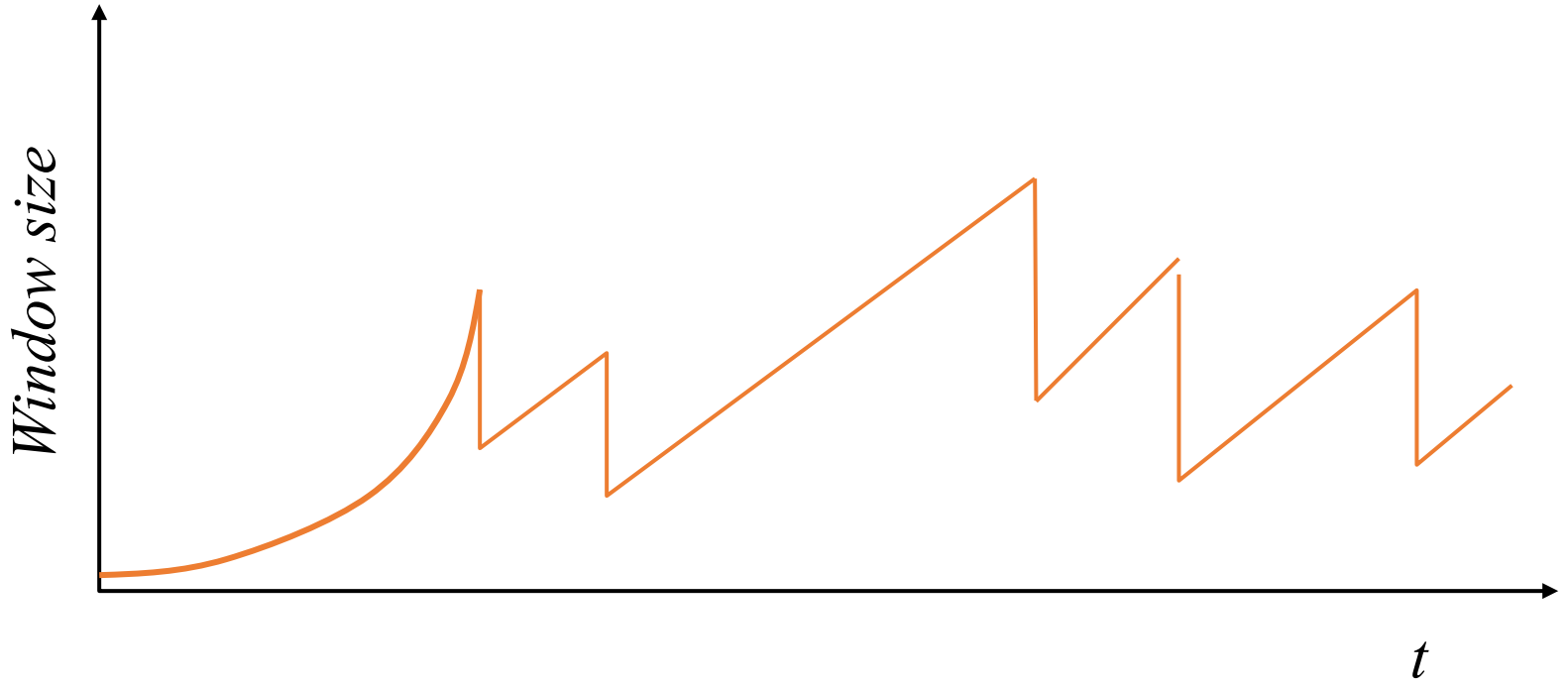
- **We need both!**
- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - **TCP window:** `min(congestion_window, receiver_window)`

Slow start and TCP sawtooth



Speed up start using exponential, not linear increase

Thought question



- Using this chart, explain why HTTP 1.1 will be many times faster than HTTP 1.0.

Aside: UDP, the User Datagram Protocol

- The **user datagram protocol** is a simple way to exchange packets between applications
  - Like TCP, you specify **ports**
  - But UDP does **not** provide any reliability
    - No flow control, no congestion control
  - It's up to the "user" (the application) to know how to manage its own traffic
- UDP common where resending, reliability not necessary
  - Streaming video (if you drop a few frames, no big deal)
- No notion of "connection"
  - You just send a packet (a datagram) to a server… you don't know if it received it

DNS, the Domain Name Service

- Memorizing IP addresses is inconvenient
  - Also, IP address does not always reflect host's purpose
    - Is it a webserver, a fileserver, a client computer?
- We use the **domain name service** to apply *human-readable names* to *IP addresses*
- www.umich.edu -> 141.211.243.251
- When you connect to a host, your computer *first*
  - Connects to a DNS server
    - Looks up the *top level domain* (e.g., *.edu), finds IP address of DNS server for that TLD
    - Looks up the *domain name* next in the chain (e.g., *.umich.edu), finds DNS server for that domain
    - Looks up the next *subdomain* (e.g., www.umich.edu), finds IP address of host

DNS, the Domain Name Service

- When you want a website, you pay for it to be listed in public DNS servers
  - Godaddy.com, other registrars will let you pay for DNS listings for a website
- Alternatively, you can get *subdomains* publicly listed
  - e.g., EECS DCO registered my host: kjleach.eecs.umich.edu
  - When someone requests that host:
    - DNS lookup for .edu, then for .umich.edu, then for .eecs.umich.edu
    - Eecs.umich.edu is a DNS server that DCO maintains, they have a listing for kjleach.eecs.umich.edu
      - Eecs.umich.edu is the *authority* for that domain
- Thought question: can you always trust what a DNS server tells you?