

CS 168

Transport and TCP

Fall 2022

Sylvia Ratnasamy

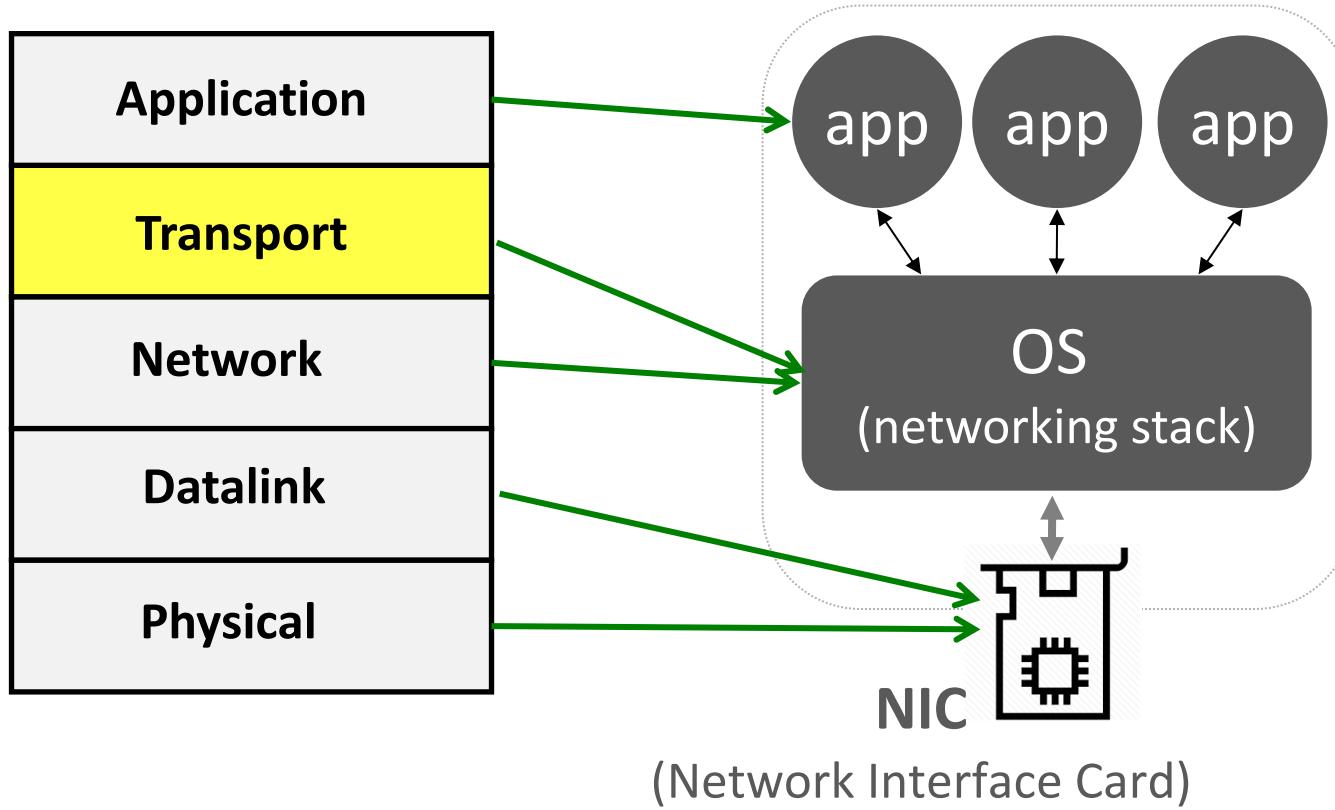
CS168.io

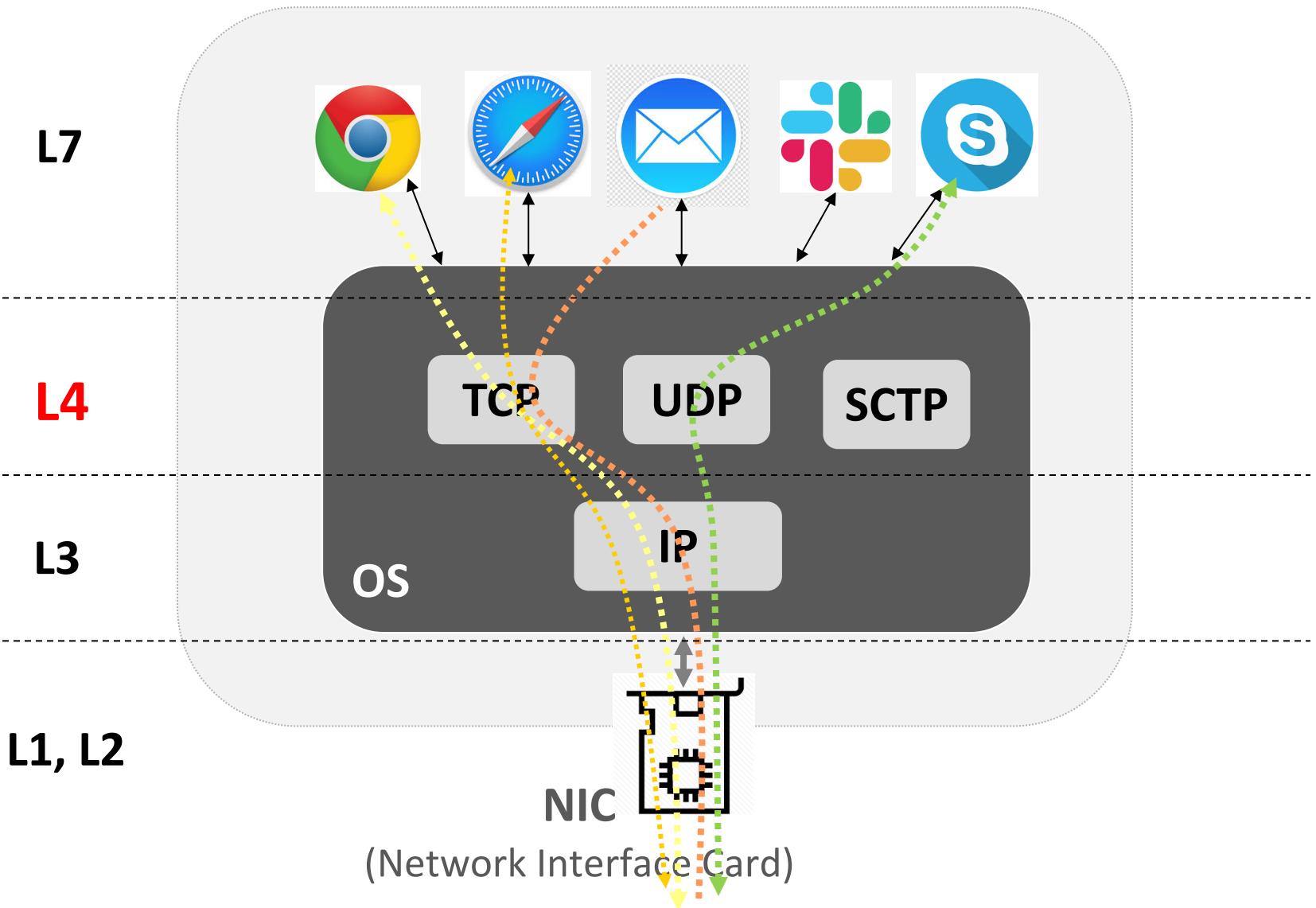
Agenda

- What does the Transport layer do?
- The design of TCP

Transport Layer

Transport in our layered architecture





Role of Transport Layer

- Bridging the gap between
 - **The abstractions application designers want**
 - **The abstractions networks can easily support**
- Could have been built into apps, but a common implementation makes app development easier

Role of Transport Layer?

- **Application layer**
 - Communication for specific applications
 - E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)
- Transport layer
 - *What do we need here?*
- Network layer
 - Logical communication between nodes
 - E.g., IP

Role of Transport Layer?

- Application layer
 - Communication for specific applications
 - E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)
- Transport layer
 - *What do we need here?*
- **Network layer**
 - Best-effort global packet delivery
 - IP

Role of Transport Layer?

- Application layer
 - Communication for specific applications
 - E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)
- **Transport layer**
 - *What do we need here?*
- Network layer
 - Best-effort global packet delivery
 - IP

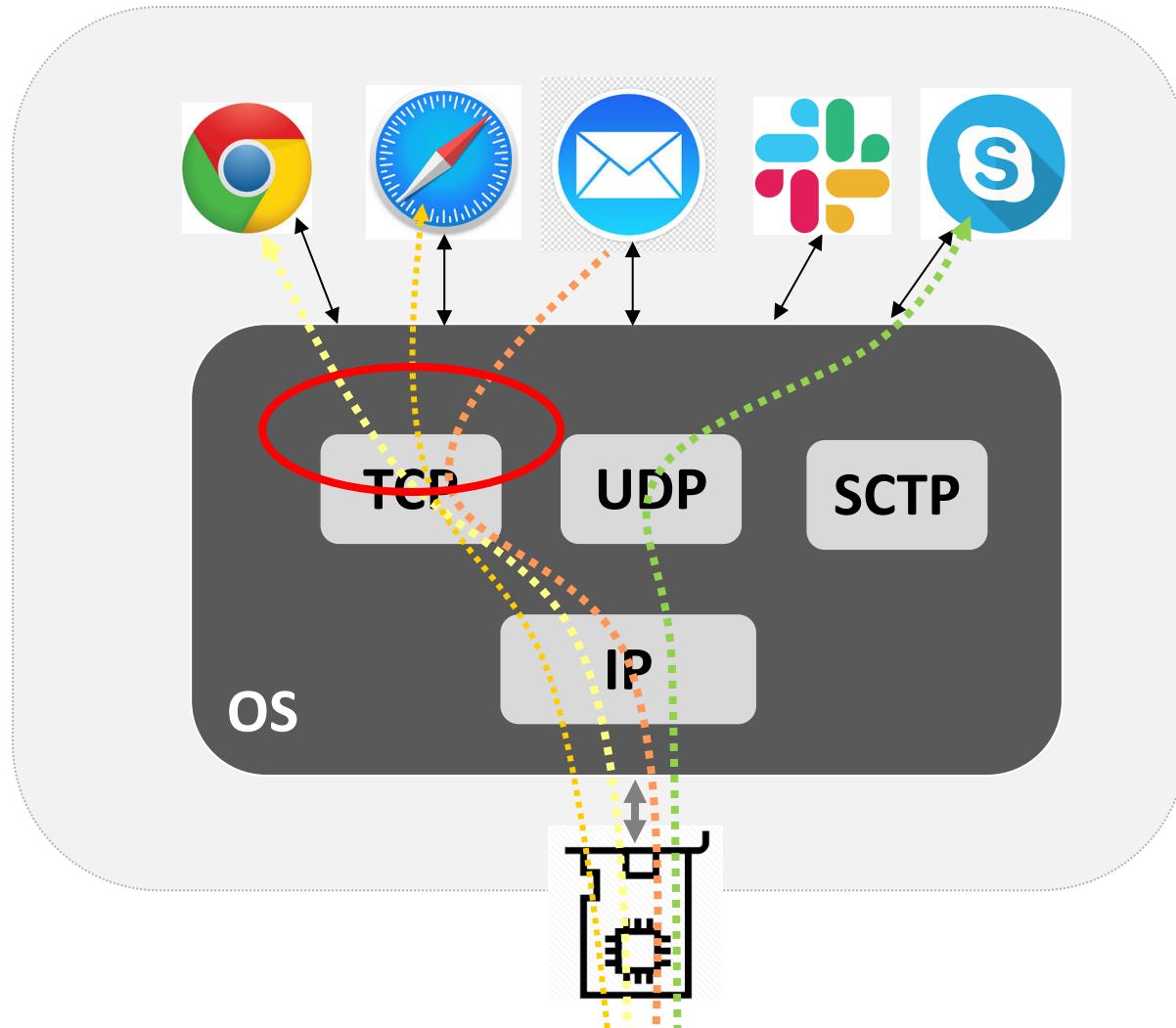
What does the transport layer address?

- Demultiplexing
 - Identify app this data belongs to (lecture#3)
- Reliability
 - Last lecture
- Translate from packets to app-level abstractions
 - E.g., between bytestreams and packets (this lecture)
- Avoid overloading the receiver (this lecture)
- Avoid overloading the network (future lectures)

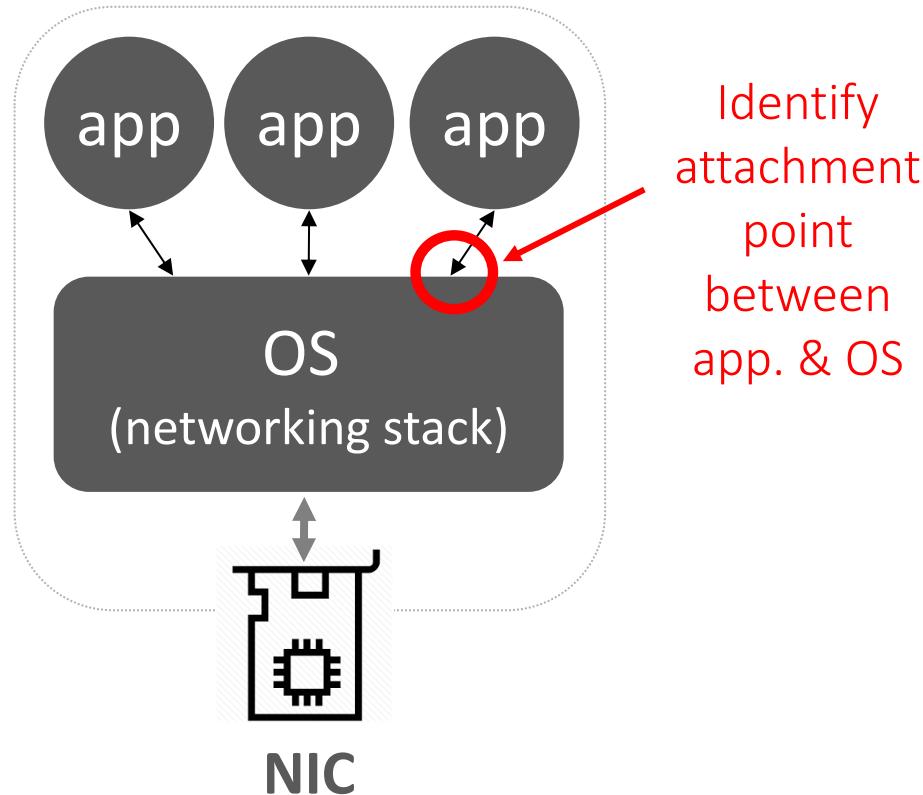
Let's first talk about these issues in general

...and then how TCP addresses them

Demultiplexing?



Recall: logical ports



Place where app connects to the OS network stack

Hence, demultiplexing

- Achieved by defining a field (“port”) that identifies the application
- Field is carried in a packet’s **L4 protocol header**

Reliable Delivery

- Last lecture
 - We've identified our design building blocks
 - Checksums
 - ACK/NACKs
 - Timeouts
 - Retransmissions
 - Sequence numbers
 - Windows
 - And discussed tradeoffs in how to apply them
 - Individual vs. Full vs. Cumulative ACKs
 - Timeout vs. ACK-driven loss detection

Application-layer abstractions

- Ideally, app doesn't see the gory details of the network
 - packets, ACKs, duplicates, reordering, corruption, ...
- Want a higher-level abstraction that meets app needs

Application Abstractions

- **Reliable in-order bytestream delivery (TCP)**
 - Logical “pipe” between sender and receiver
 - Bytes inserted into pipe by sender-side app
 - They emerge, in order, at the receiving app
- **Individual message delivery (UDP)**
 - Unreliable (application responsible for resending)
 - Messages limited to single packet

What does the transport layer address?

- Demultiplexing
 - Identify app this data belongs to (logical ports, lecture#3)
- Reliability
 - Last lecture (though TCP does things a little differently)
- Translate from packets to app-level abstractions
 - E.g., between bytestreams and packets
- Avoid overloading the receiver
- Avoid overloading the network

How big should the window be?

- **Lecture#12:** Pick window size W to balance three goals
 - Take advantage of network capacity (“fill the pipe”)
 - But don’t overload the receiver (flow control)
 - And don’t overload links (congestion control)
- **Lecture#12:** For the first goal: $W \times \text{pkt_size} \sim \text{RTT} \times B$
 - RTT is round-trip time and B is the bottleneck BW
 - This is an upper bound on the desired size of W
- Now consider the other two goals...

Don't overload the receiver

- Consider the transport layer at the receiver side
- May receive packets out-of-order but can only deliver them to the application in order
- Hence, the receiver must buffer incoming packets that are out of order
 - Must continue to do so until all “missing” packets arrive!
- Must ensure the receiver doesn’t run out of buffers

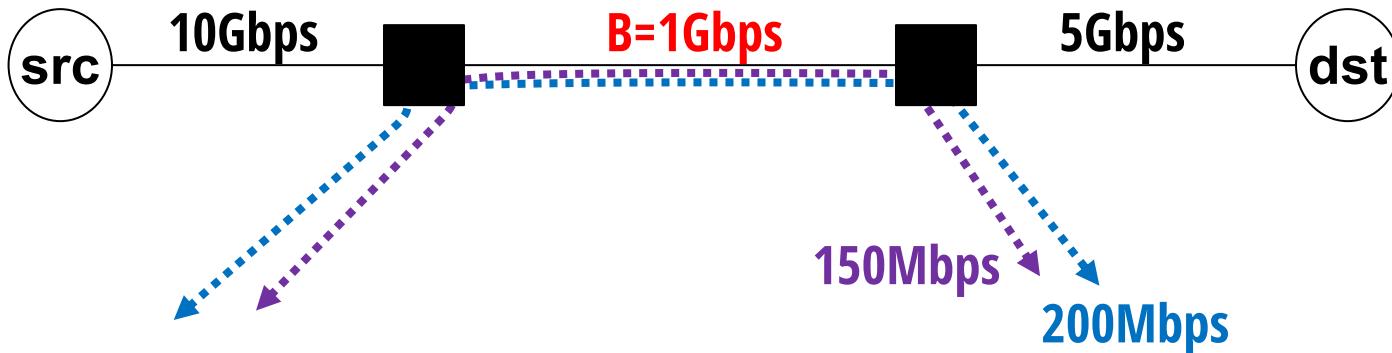
Hence: Flow Control

The basic idea is very simple...

- Receiver tells sender how much space it has left
 - TCP calls this the “**advertised**” window
- Advertisement is carried in ACKs
- Sender adjusts its window accordingly
 - Packets in flight cannot exceed the receiver’s advertised window

Don't overload the network

- Previously: sender sets W to fully consume the bottleneck link bandwidth
 - I.e., sender is sending data at the rate of B
- In practice, bottleneck is shared with other flows
- Hence, sender should only consume *its share* of B
- But what is this share?



Congestion Control

- The transport layer at the sender implements a congestion control algorithm that dynamically computes the sender's share of the bottleneck link BW
- TCP calls this the sender's **congestion window (cwnd)**
- Computed to balance multiple goals
 - Maximize my performance
 - Without overloading any link (avoid dropped packets)
 - While sharing bandwidth "fairly" with other senders
- Topic for (multiple) future lectures

How big should the window be?

- Pick window size W to balance three goals
 - Take advantage of network capacity (“fill the pipe”)
 - But don’t overload the receiver (flow control)
 - And don’t overload links (congestion control)
- First goal: $W \sim RTT \times B$
- Second: $W \sim$ receiver’s advertised window
- Third: $W \sim$ sender’s congestion window (cwnd)
- Window size is set to the **minimum** of the above

In practice

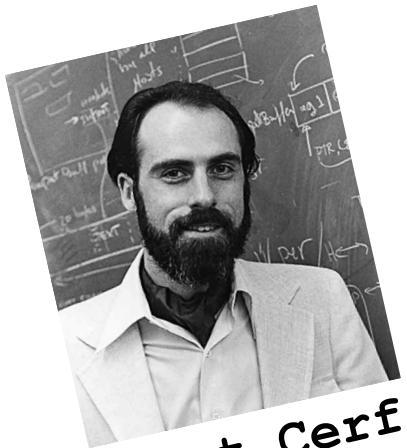
- A sender's cwnd should be $\leq \text{RTT} \times B$
- And it's difficult for the sender to discover B
- Hence, window size is the minimum of:
 - The congestion window computed at the sender
 - The receiver's advertised window

Recap: what the transport layer tackles

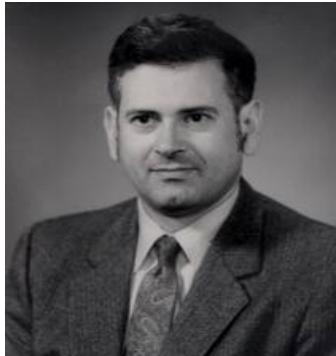
- Demultiplexing
 - logical ports
- Reliability
 - acks, timeouts, windows, etc.
- Translation between abstractions
 - between packets and bytestreams (coming up)
- Avoid overloading the receiver
 - receiver's advertised window
- Avoid overloading the network
 - sender computes a congestion window

What if your app doesn't want all these features?

- E.g., an application that doesn't need reliability
- E.g., an app that exchanges very short messages
- UDP: User Datagram Protocol
 - A no-frills, minimalist protocol
 - Only implements mux/demux



Vint Cerf



Bob Kahn

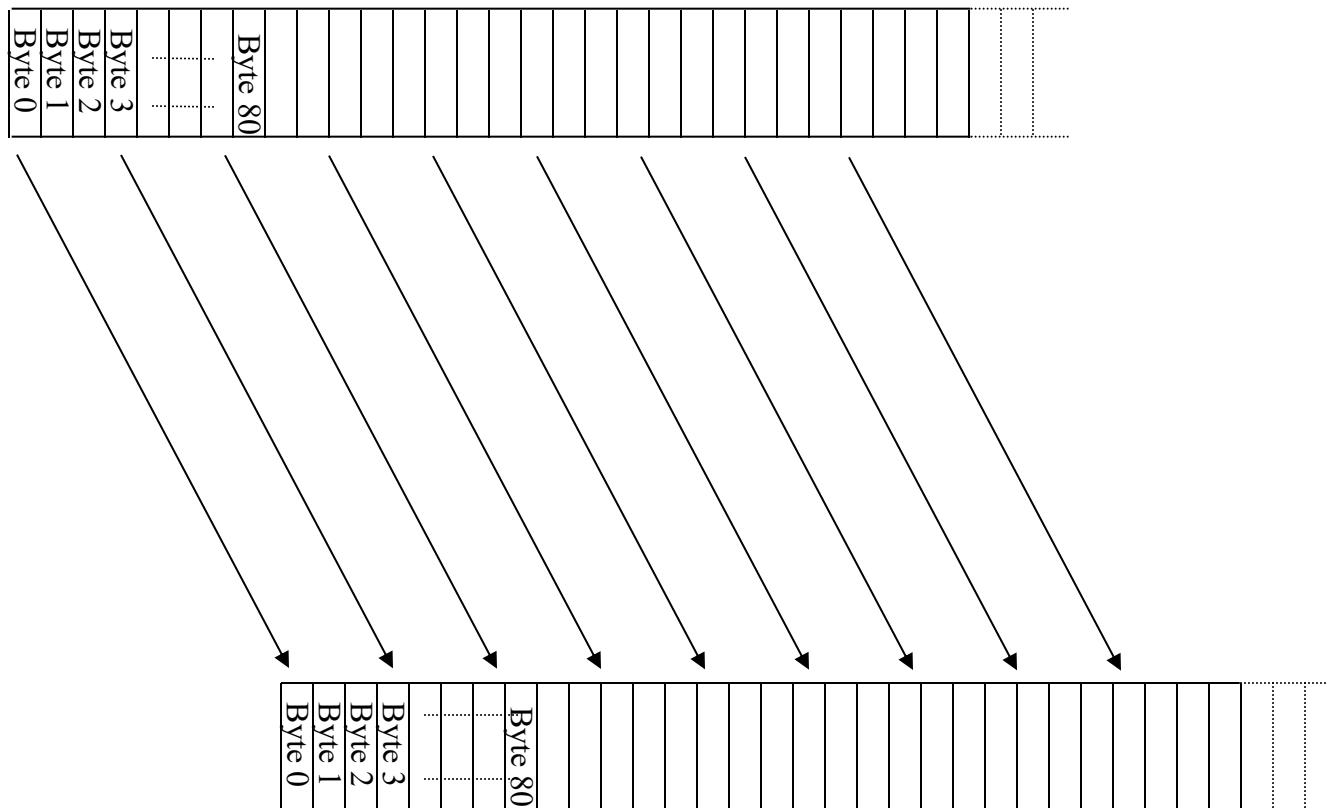
TCP

The TCP Abstraction

- TCP delivers a reliable, in-order, bytestream

TCP “Stream of Bytes” Service...

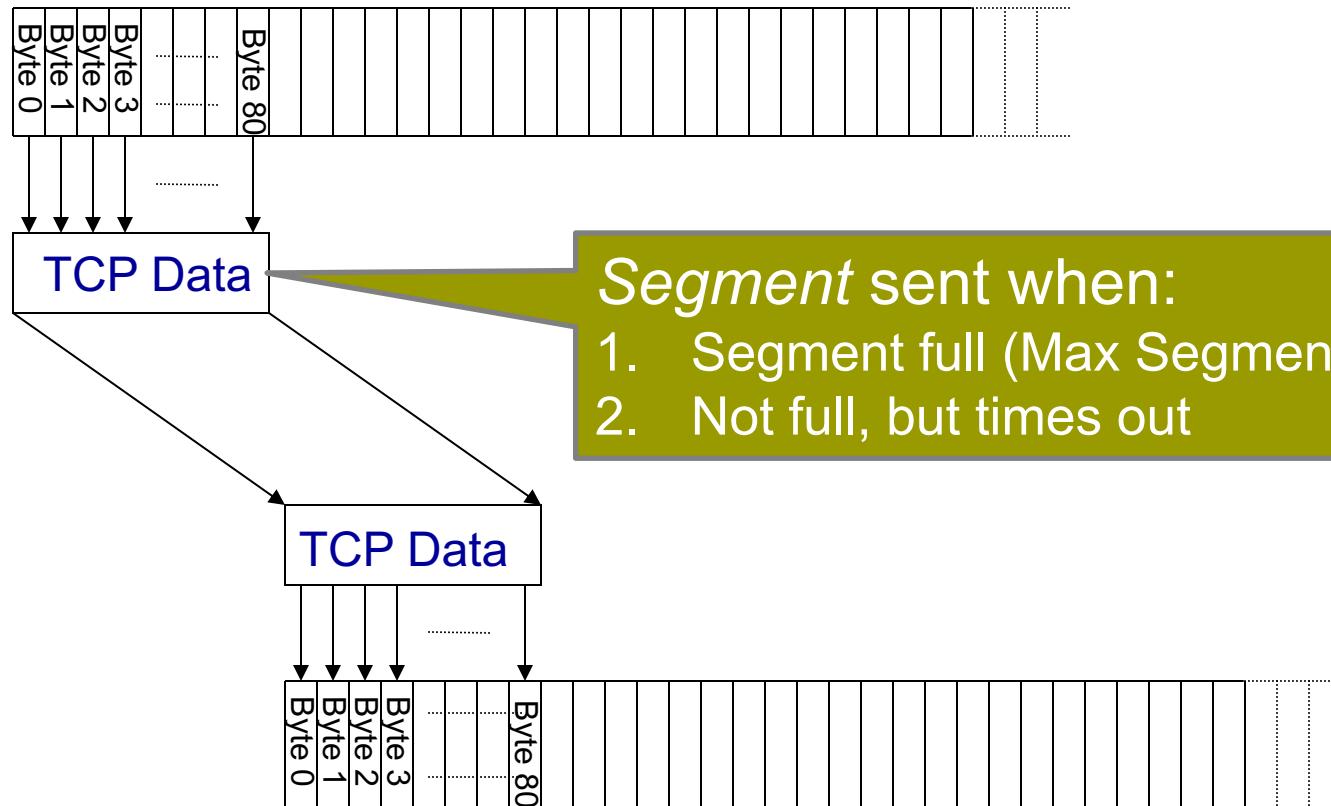
Application @ Host A



Application @ Host B

... Implemented Using TCP “Segments”

Application @ Host A



Application @ Host B

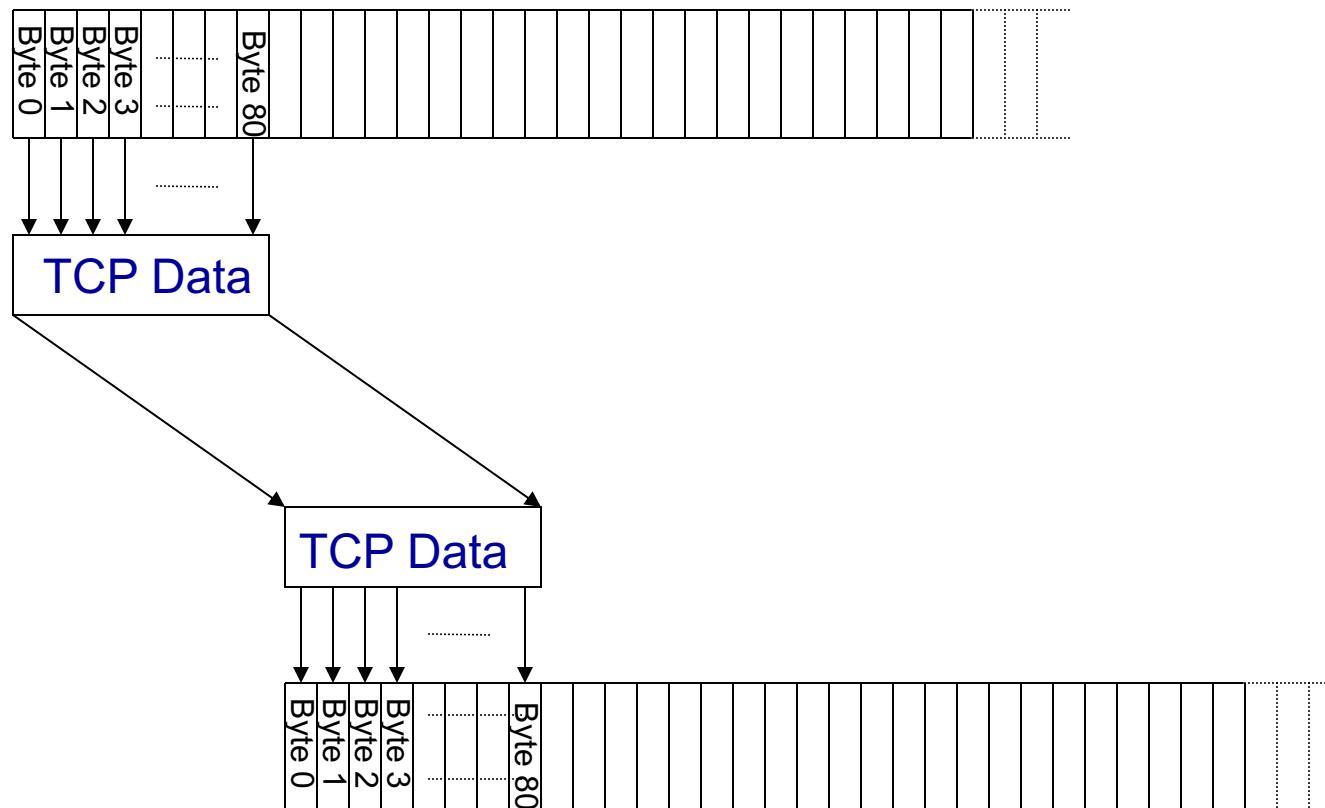
TCP Segment



- TCP packet
 - IP packet with a TCP header and data inside
- IP packet
 - No bigger than Maximum Transmission Unit (**MTU**)
- TCP **segment**
 - No more than **Maximum Segment Size (MSS)** bytes
 - $MSS = MTU - (\text{IP header}) - (\text{TCP header})$

... Implemented Using TCP “Segments”

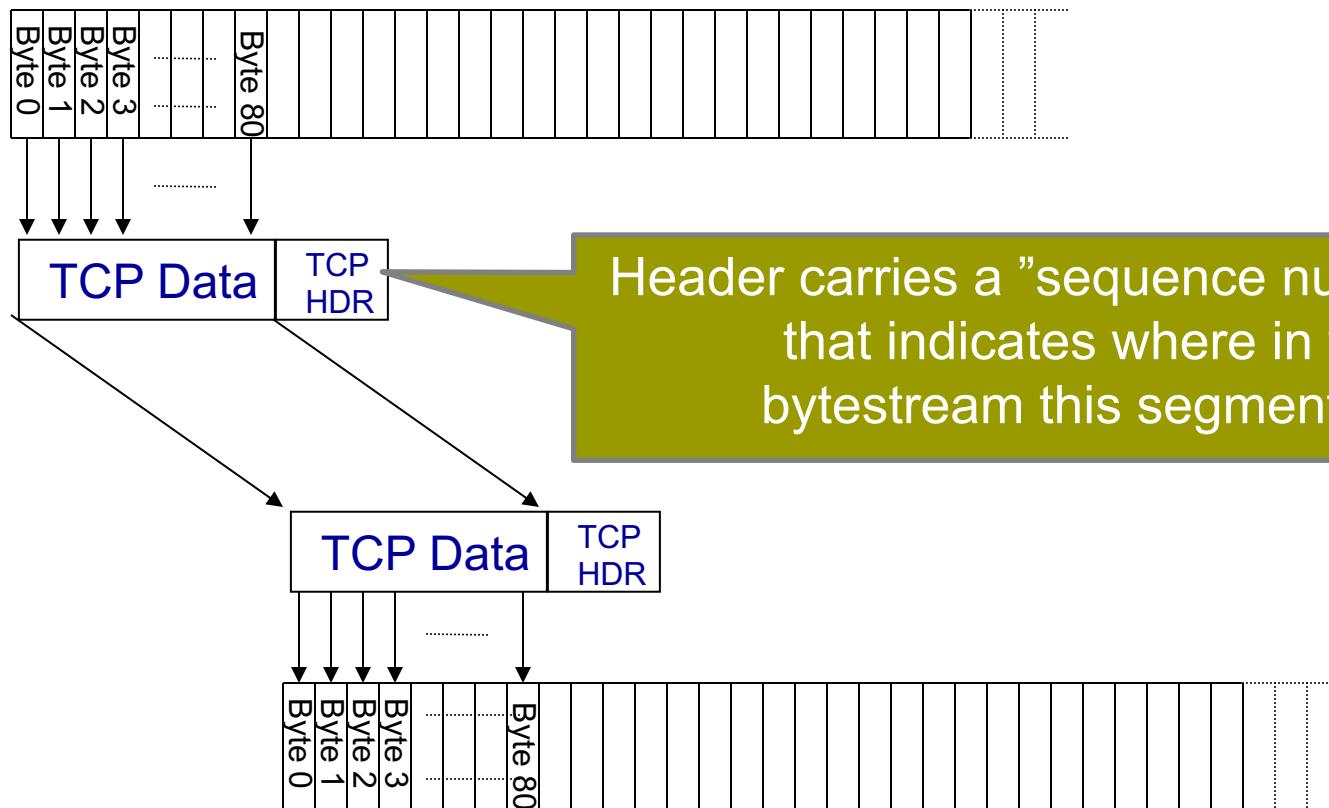
Application @ Host A



Application @ Host B

... Described by TCP headers

Application @ Host A



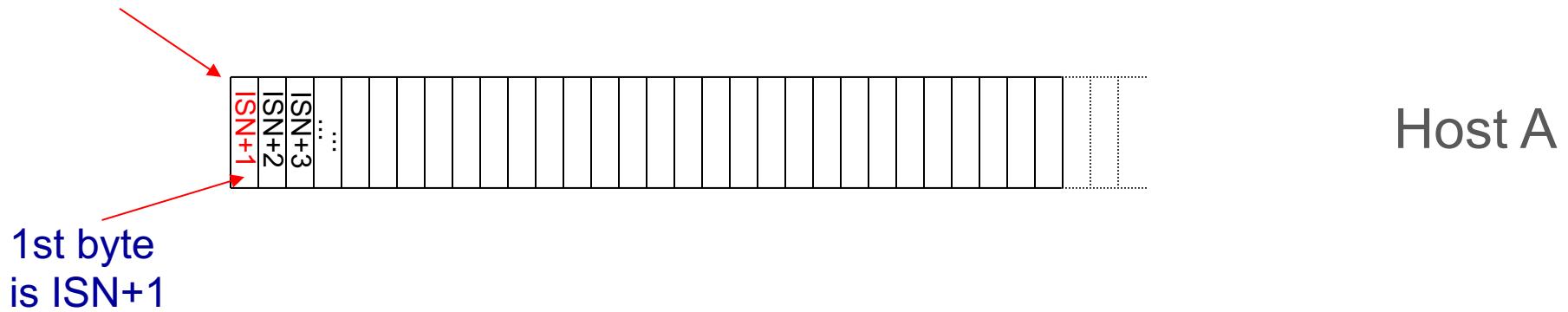
Application @ Host B

Major Notation Change

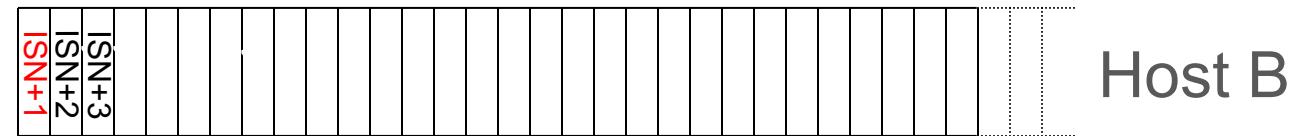
- Previously we focused on packets:
 - Packets had numbers
 - ACKs referred to those numbers
 - Window sizes expressed in terms of # of packets
- TCP focuses on bytes. Thus,
 - Packets identified by the bytes they carry
 - ACKs refer to the bytes received
 - Window size expressed in terms of # of bytes
- You should be prepared to reason in terms of either

TCP Sequence Numbers

Numbering starts with
an **ISN (Initial Sequence Number)**

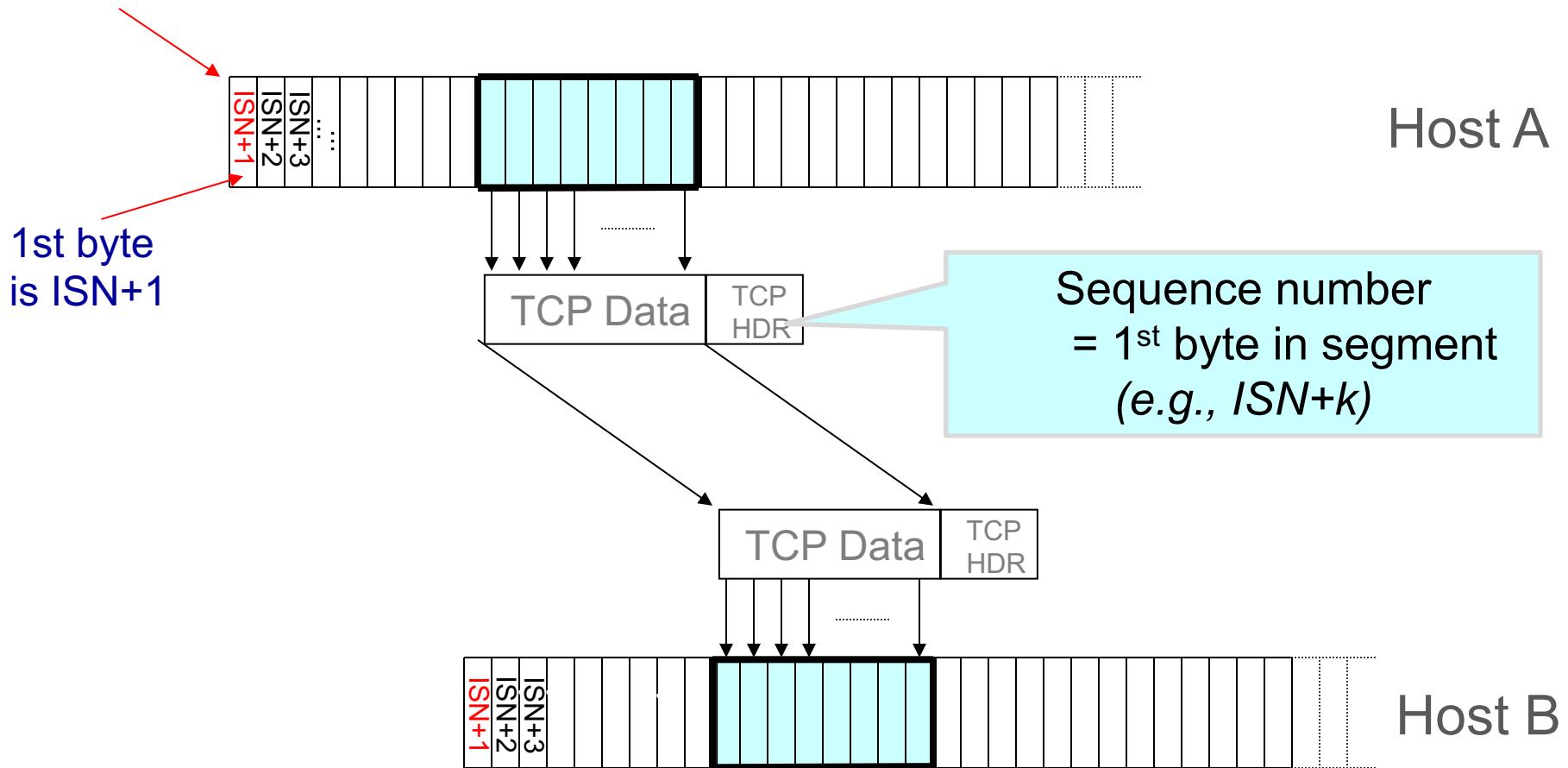


1st byte
is ISN+1



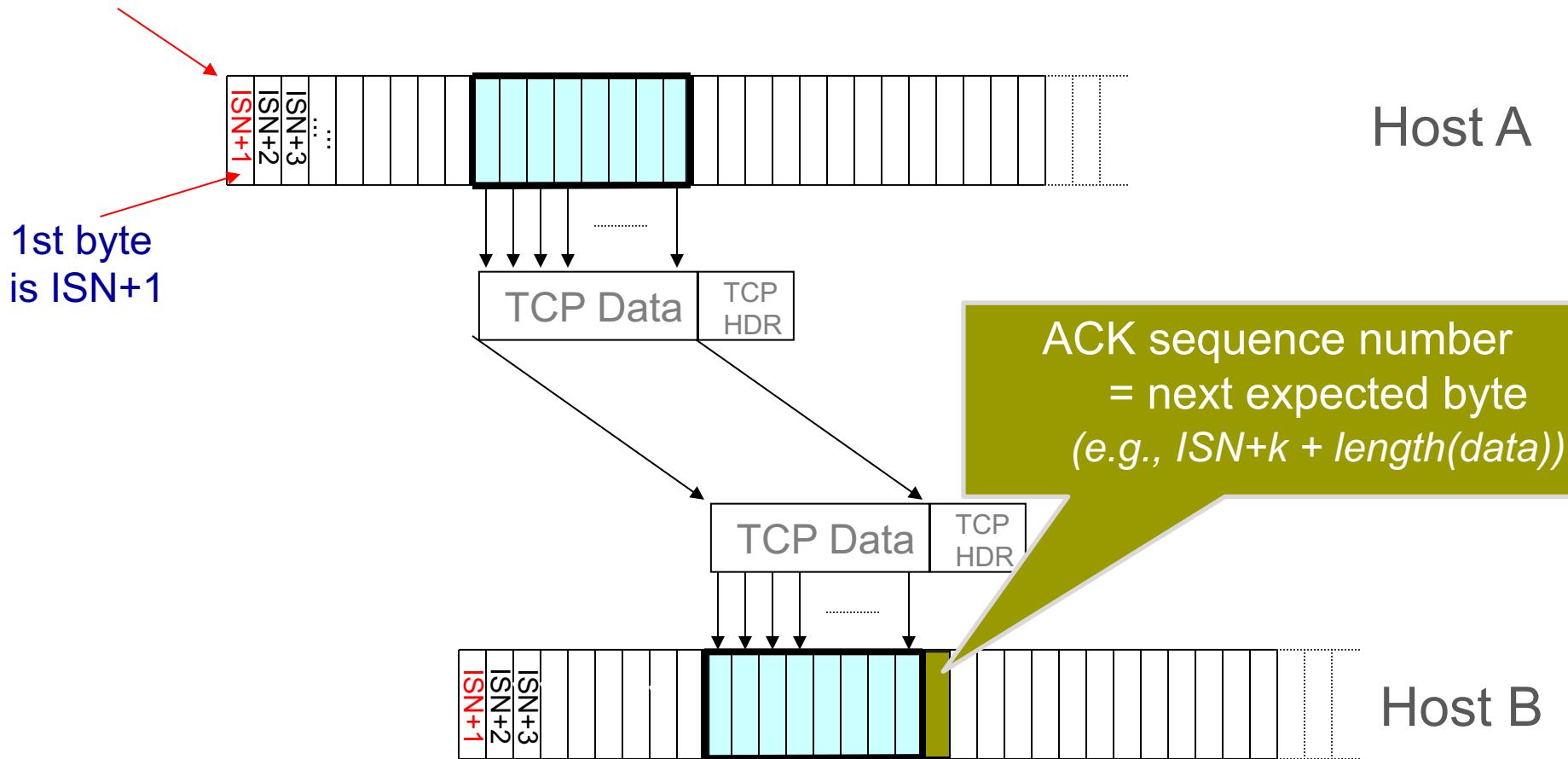
TCP Sequence Numbers

Numbering starts with
an **ISN (Initial Sequence Number)**



TCP Sequence Numbers

Numbering starts with
an **ISN (Initial Sequence Number)**



The TCP Abstraction

- TCP delivers a reliable, in-order, bytestream
- Reliability requires keeping state
 - Sender: packets sent but not ACKed, related timers
 - Receiver: out-of-order packets
- Each bytestream is called a **connection** or session
 - Each with their own connection state
 - State is in hosts, not network!

Note#1: TCP is “connection oriented”

- TCP includes a connection setup and tear-down step
 - Used to initialize connection state at both endpoints
 - Details coming up ...

#2: TCP connections are full-duplex

- So far, we've talked about a connection as having a sender side and a receiver side
- But connections in TCP are **full-duplex**
 - Each side of the connection can be sender *and* receiver
 - I.e., A can send data to B, while B sends data to A
 - Simultaneously, over the same connection
 - Packets carry both data and ACK info
- We can usually ignore this point (for this class)
- Except when it comes to connection establishment
- Will return to this later ...

The TCP Abstraction

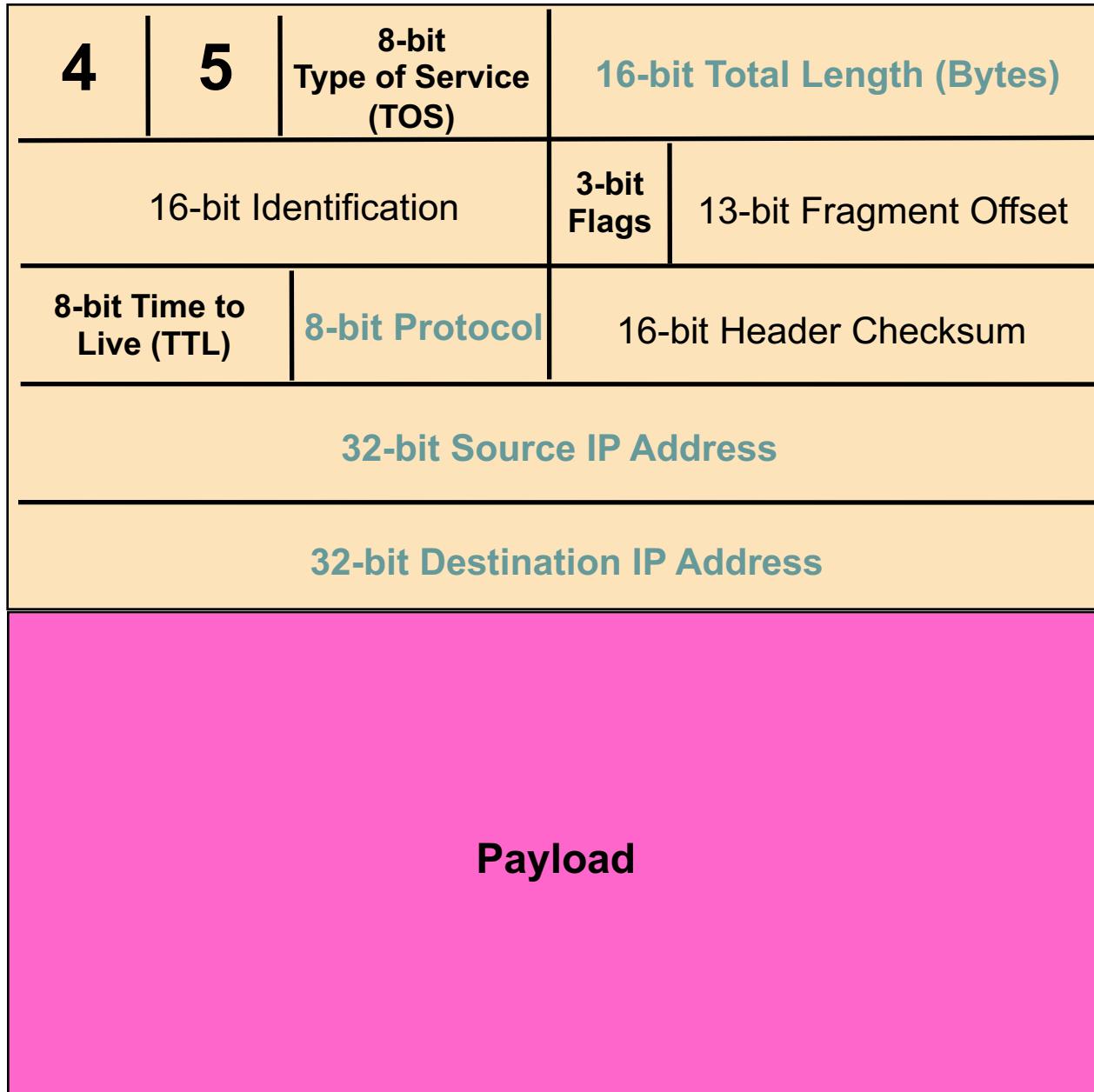
- TCP delivers a reliable, in-order, bytestream
- TCP is connection-oriented
 - Per-connection state is maintained at sender & receiver

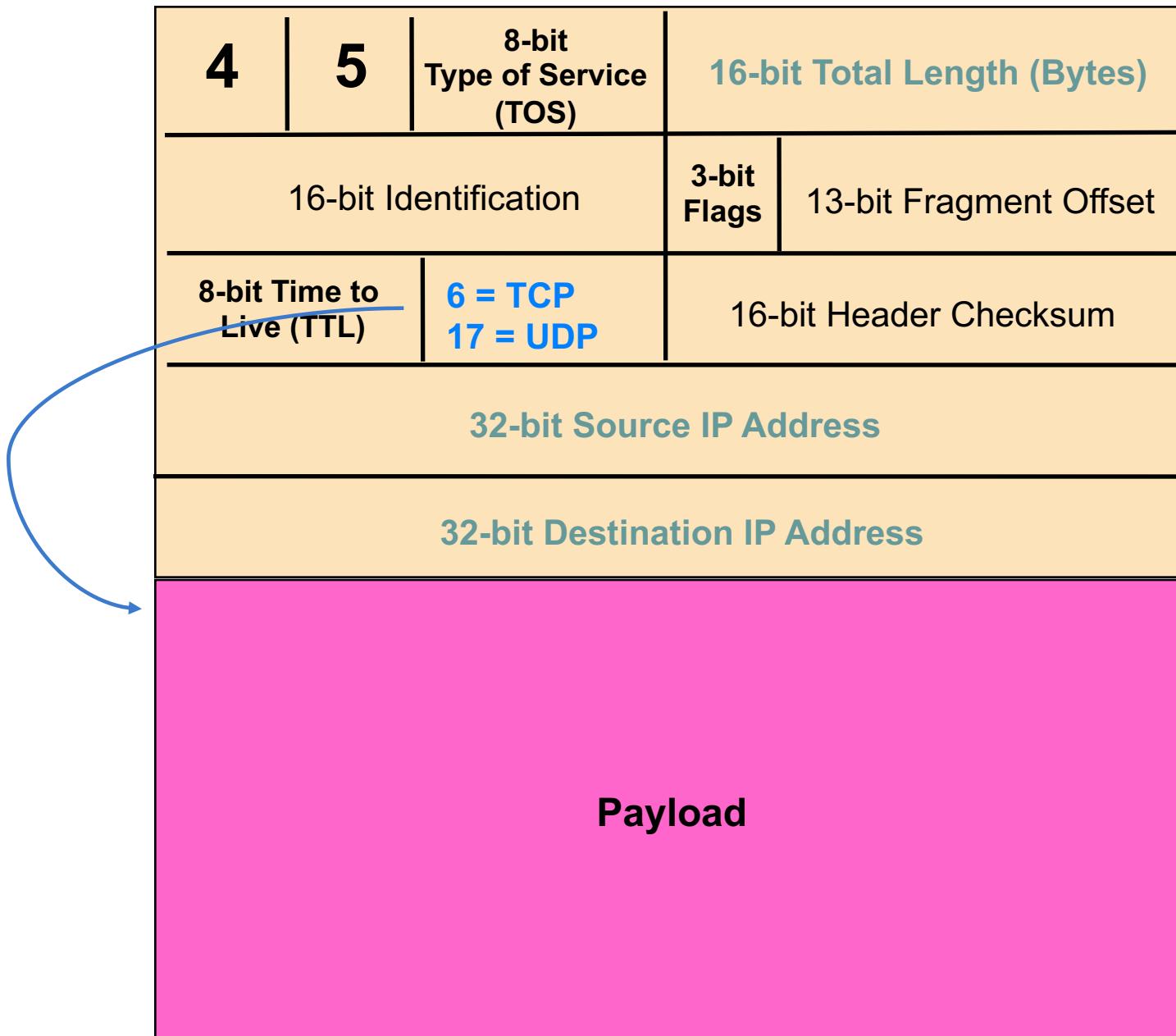
Functionality

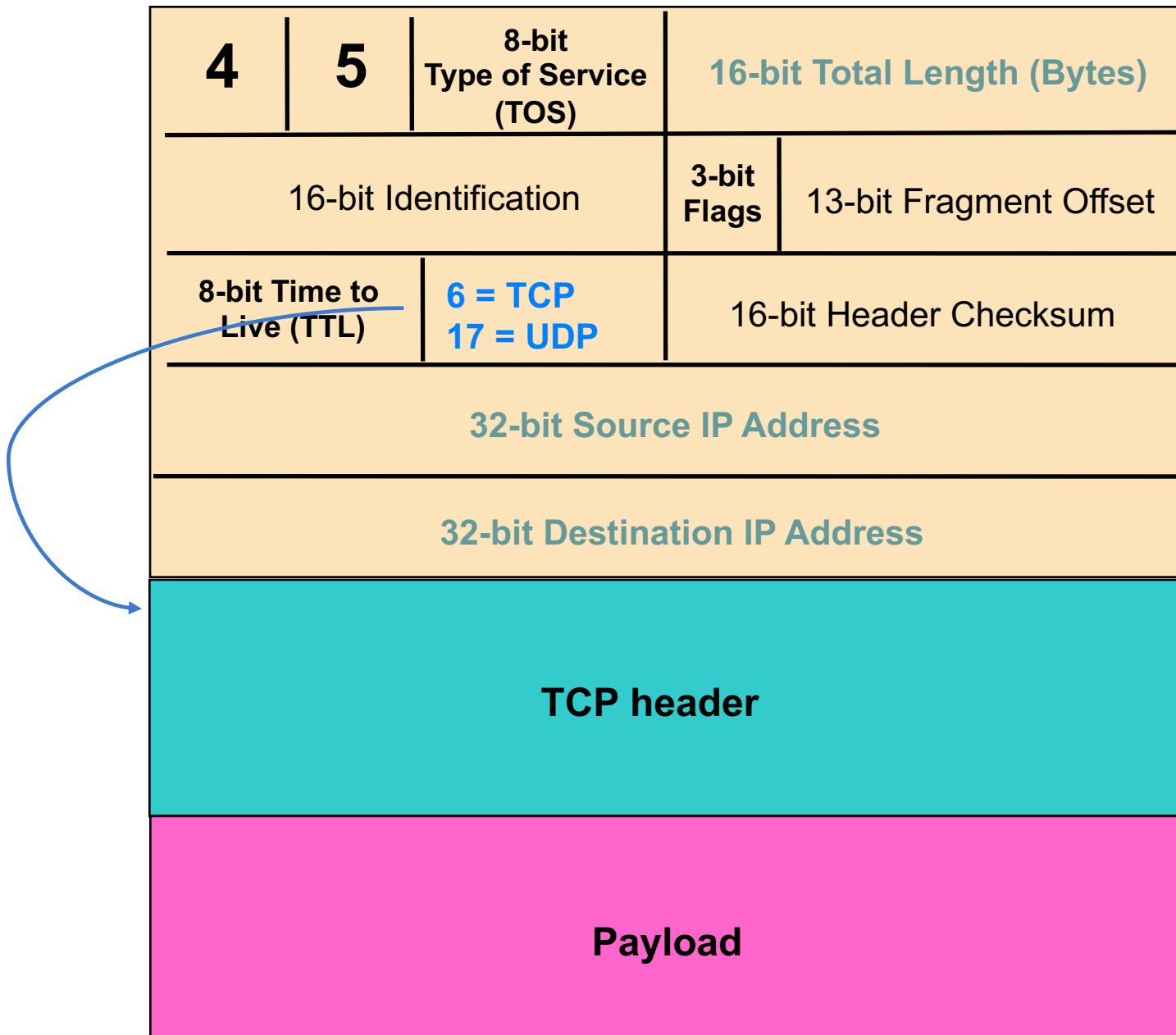
- Mux/demux among processes
- Reliability
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

Ports

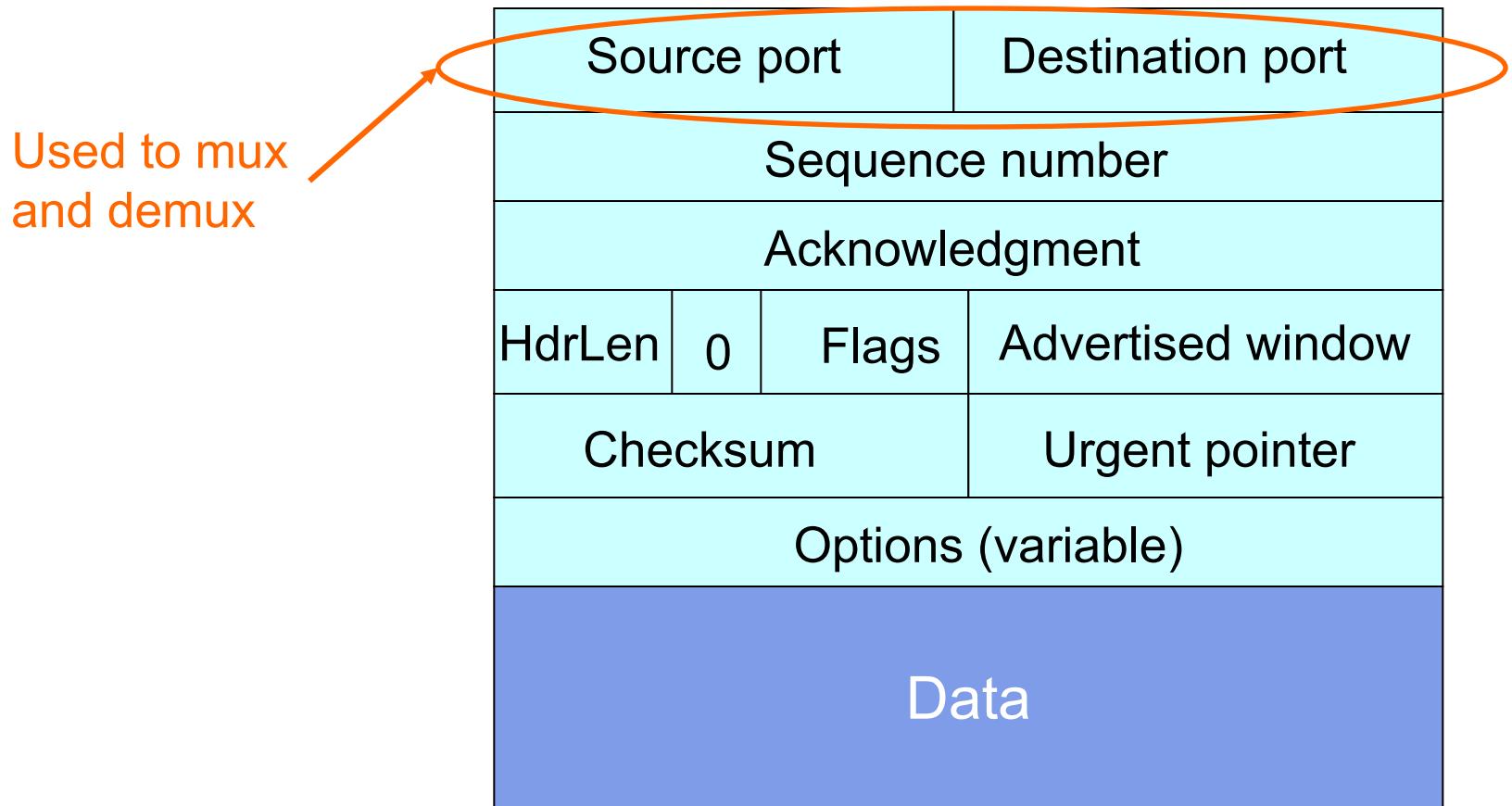
- 16-bit port address space for TCP and UDP
- Some ports are “well known” (0-1023)
 - e.g., ssh:22, http:80
 - Services can listen on well-known port
 - Client (app) knows appropriate port on server
- Other ports are “ephemeral” (most 1024-65535):
 - Given to clients (at random)







TCP Header



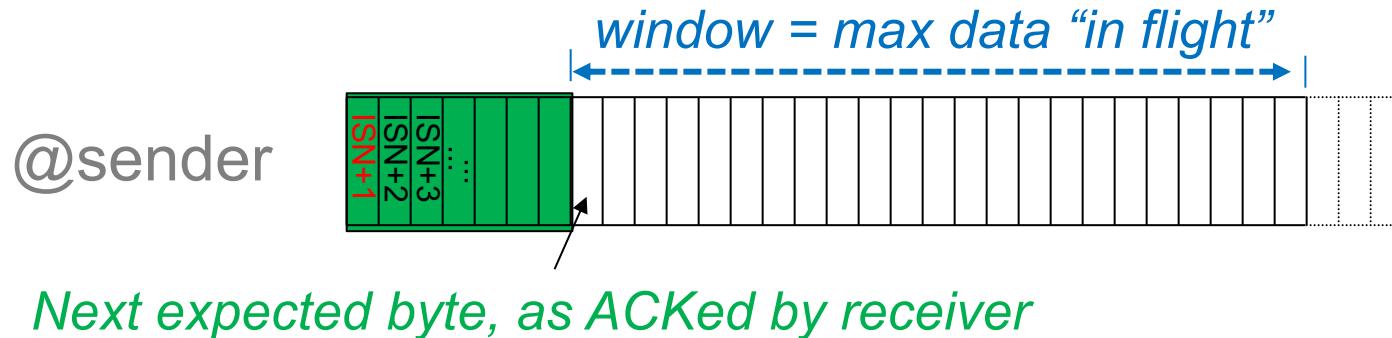
Functionality

- Mux/demux among processes
- Reliability
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

How does TCP handle reliability?

Many of our previous ideas, with some key differences

- Sequence numbers are **byte offsets**
- Uses **cumulative ACKs**; with “next expected byte” semantics
- Uses **sliding window**: up to W contiguous bytes in flight



How does TCP handle reliability?

Many of our previous ideas, with some key differences

- Sequence numbers are **byte offsets**
- Uses **cumulative ACKs**; with “next expected byte” semantics
- Uses **sliding window**: up to W contiguous bytes in flight
- Retransmissions triggered by **timeouts** and **duplicate ACKs**
- **Single timer**, for left hand side (1^{st} byte) of the window
- Window size is a function of cwnd and advertised window
 - With special accounting for duplicate ACKs (future lecture)
- **Timeouts are computed from RTT measurements**
 - Covered in section

ACKing and Sequence Numbers

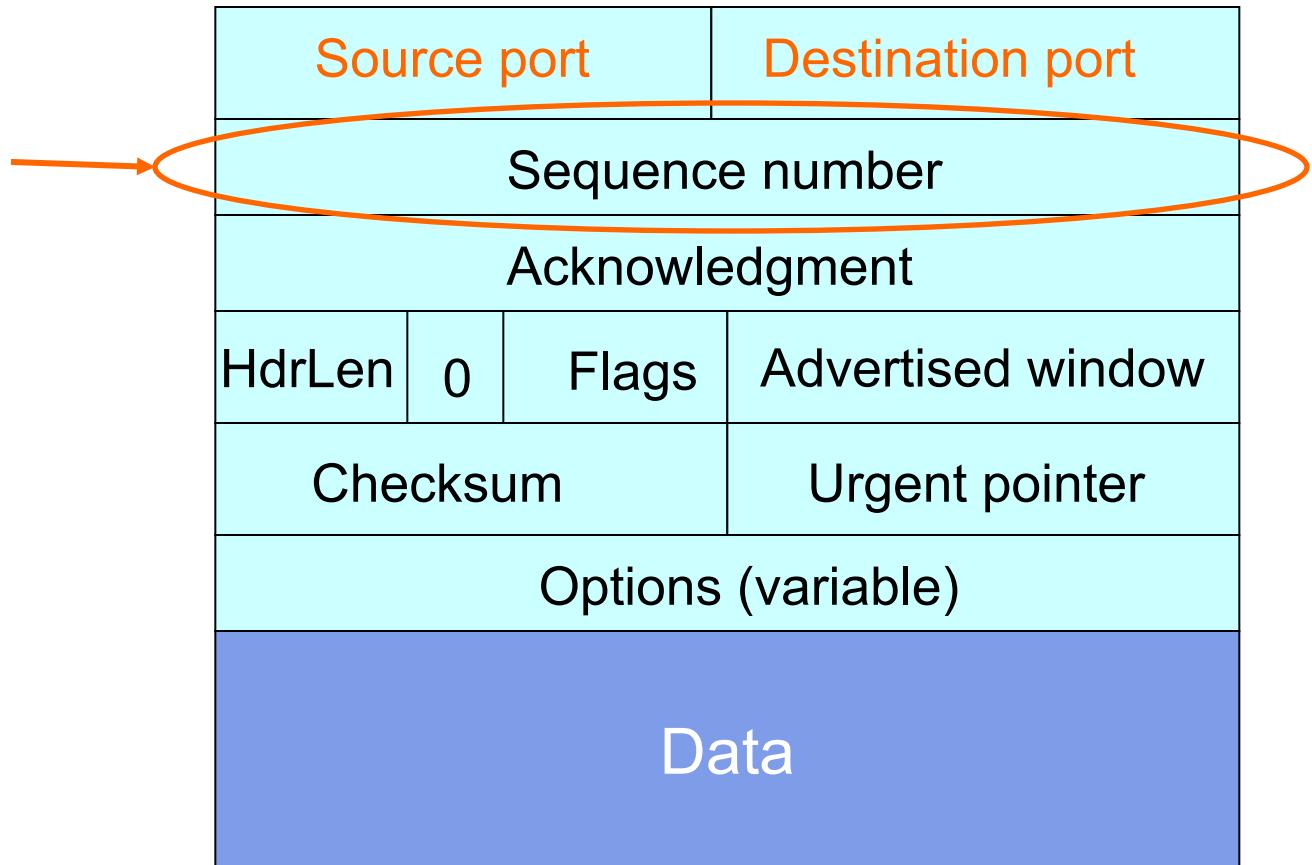
- Sender sends packet
 - Data starts with sequence number X
 - Packet contains B bytes
 - X, X+1, X+2,X+B-1
- Upon receipt of packet, receiver sends an ACK
 - If all data prior to X already received:
 - ACK acknowledges X+B (**because that is next expected byte**)
 - If highest contiguous byte received is a smaller value Y
 - ACK acknowledges Y+1 (**because TCP uses cumulative ACKs**)

Pattern (w/ only one packet in flight)

- Sender: seq number = X , length= B
- Receiver: ACK= $X+B$
- Sender: seq number = $X+B$, length= B
- Receiver: ACK= $X+2B$
- Sender: seq number = $X+2B$, length= B
- Seq number of next packet is same as last ACK

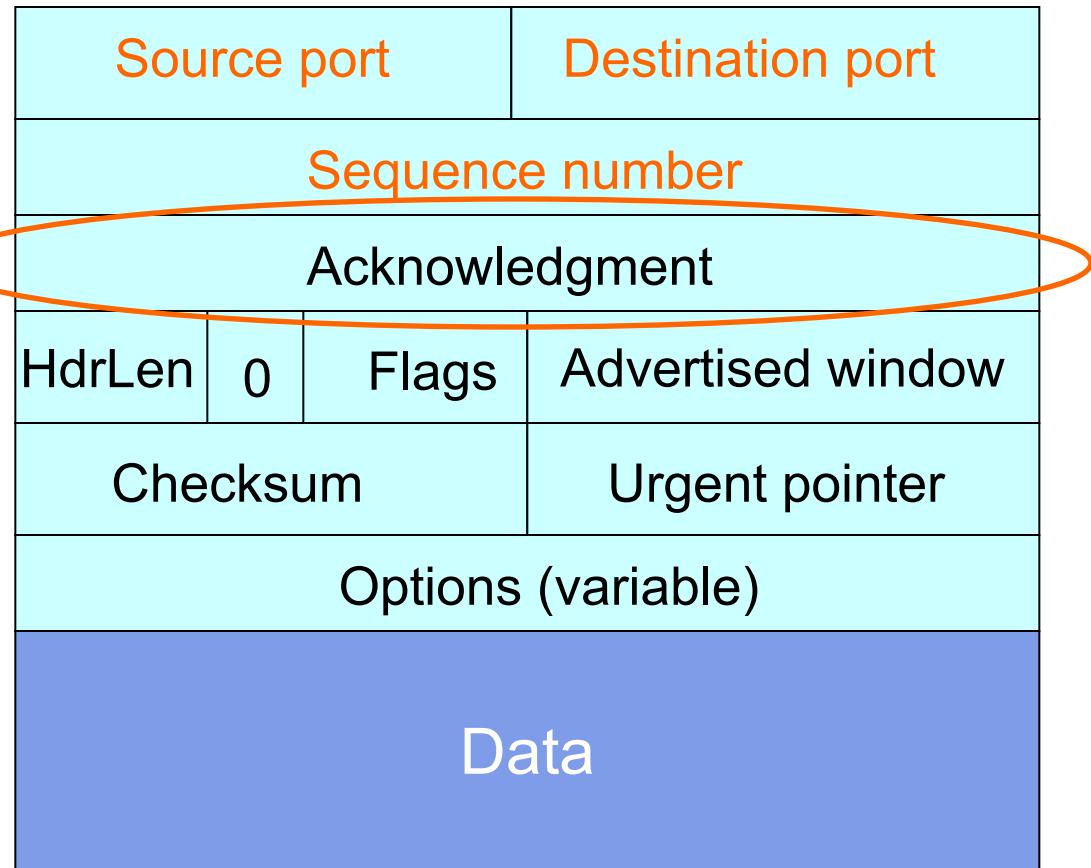
TCP Header

Starting byte offset of data carried in this segment

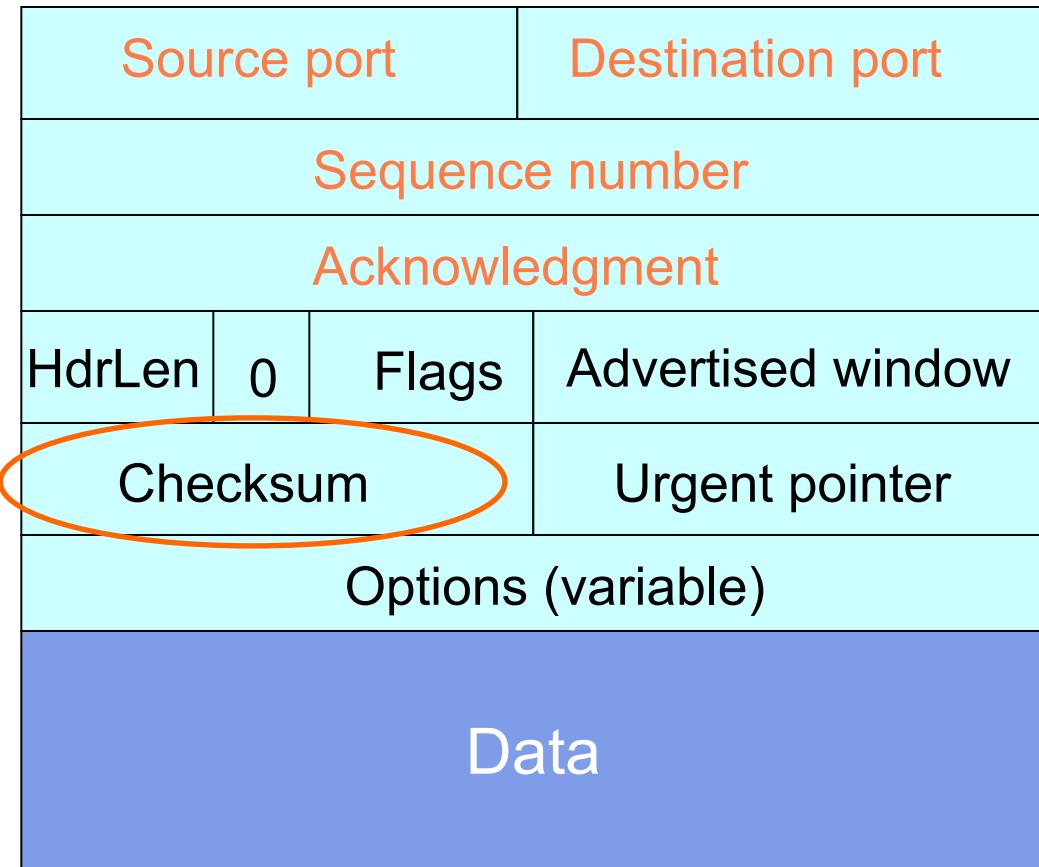


TCP Header

Acknowledgment gives sequence number just beyond the highest sequence number received **in order** (i.e., next expected byte)



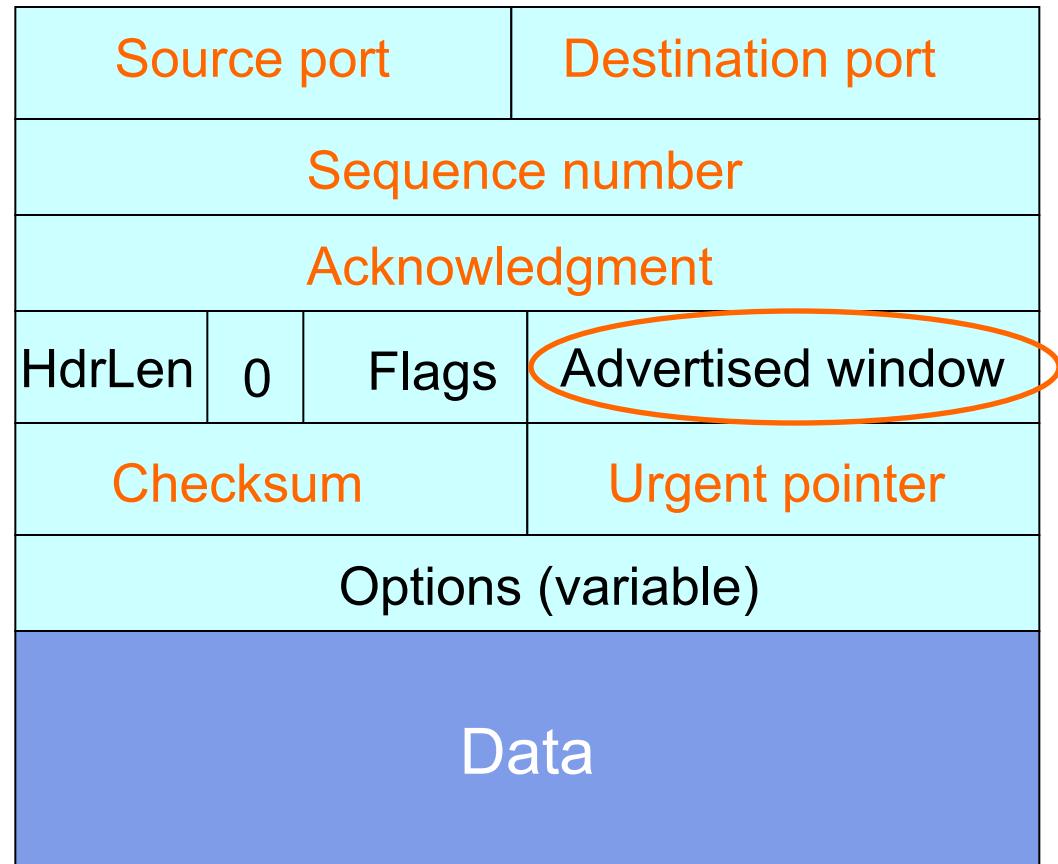
TCP Header



Functionality

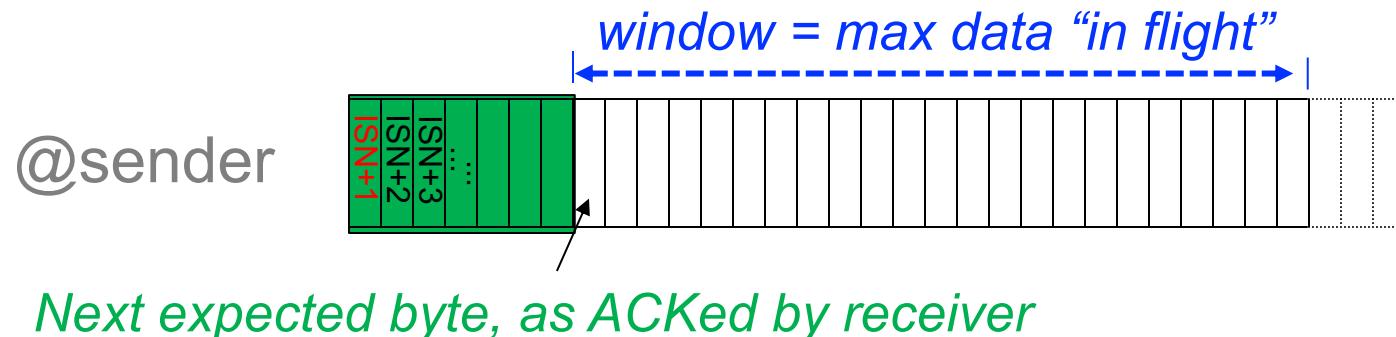
- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

TCP Header



Implementing Sliding Window

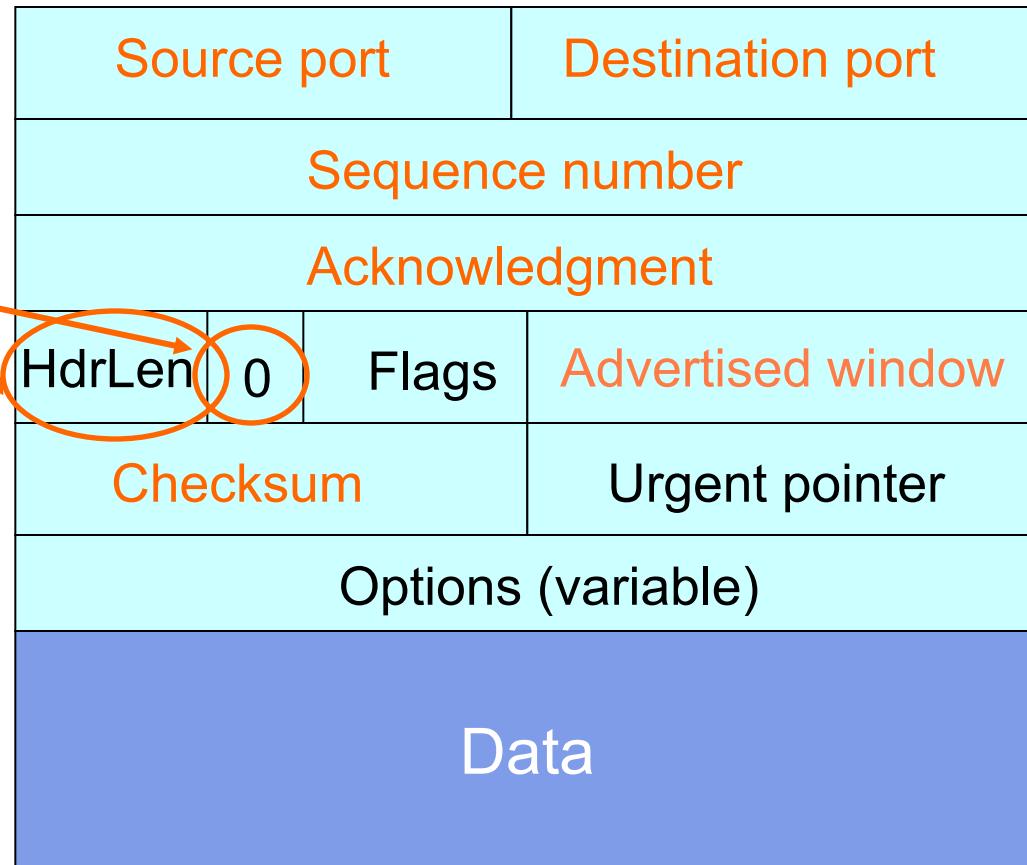
- Sender maintains a window
 - Data that has been sent but not yet ACK'ed
 - Window size = maximum amount of data in flight
- **Left edge** of window:
 - Beginning of **unacknowledged** data
- **Right edge** of window (ignoring congestion control)
 - Depends on the window advertised by receiver
 - Which depends on receiver's buffer space



TCP Header: What's left?

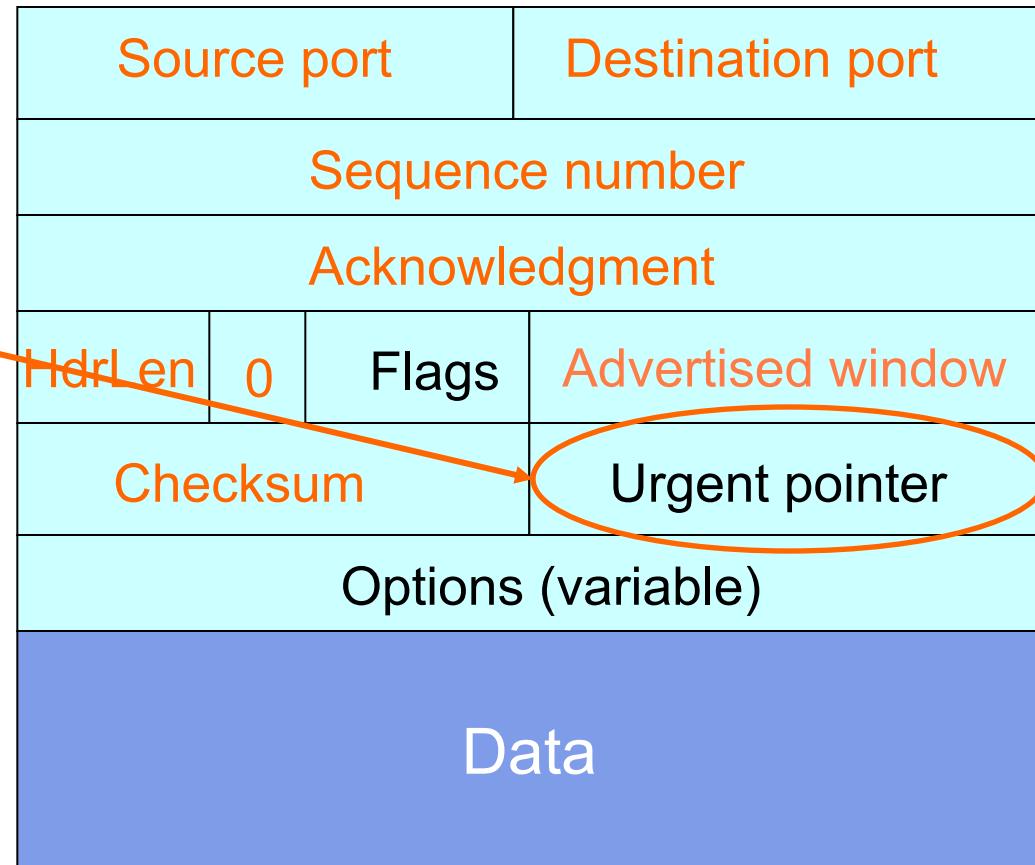
“Must Be Zero”
6 bits reserved

Number of 4-byte
words in TCP
header;
5 = no options

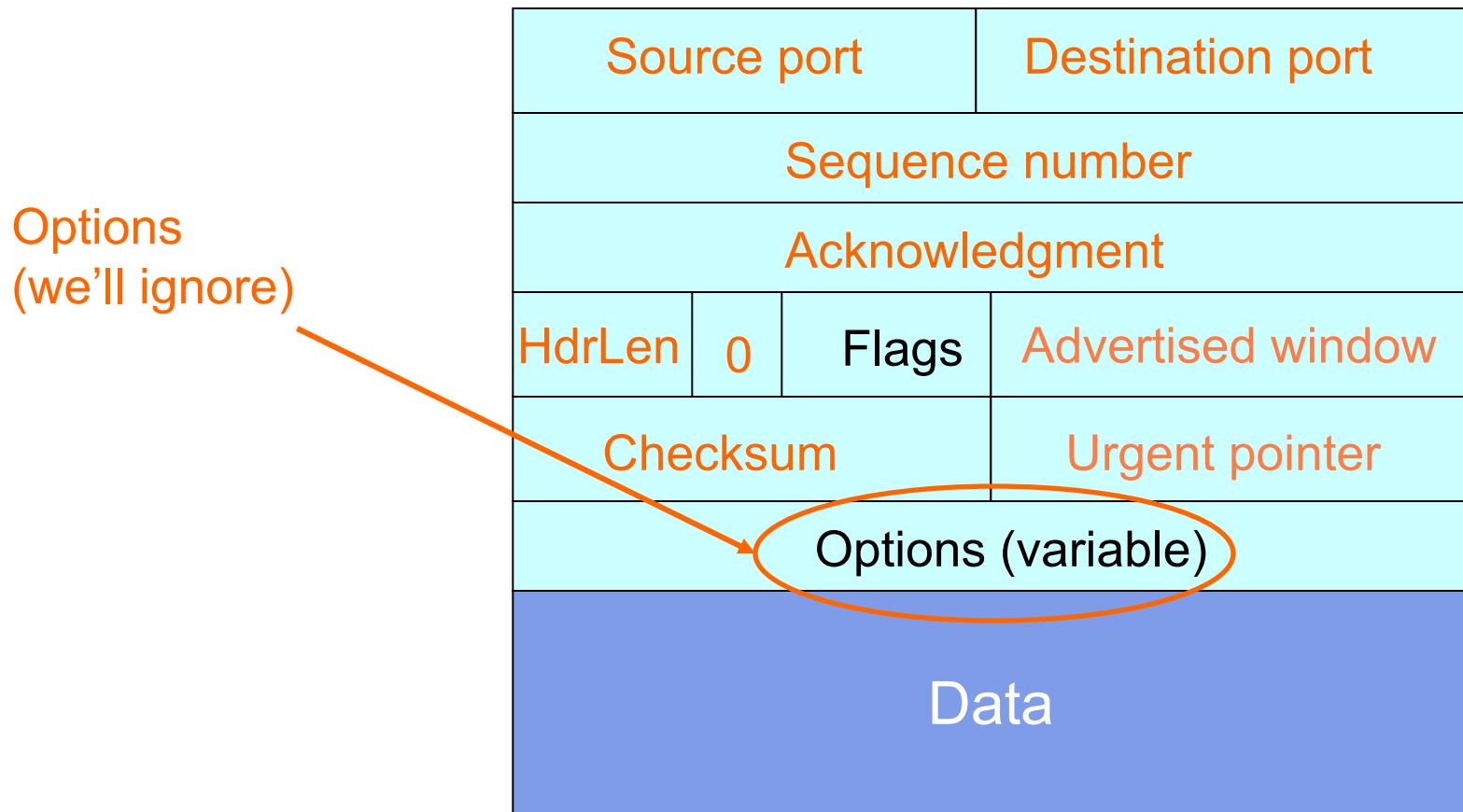


TCP Header: What's left?

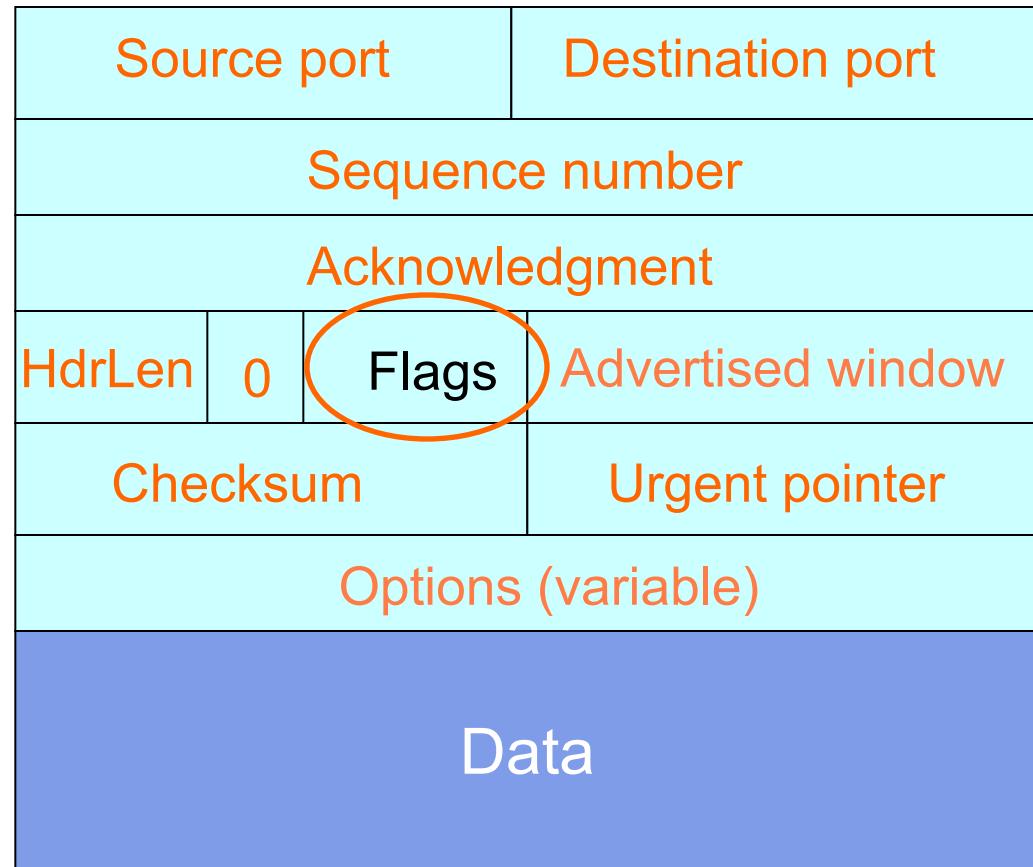
Used with **URG** flag to indicate urgent data (not discussed further)



TCP Header: What's left?



TCP Header: What's left?



Functionality

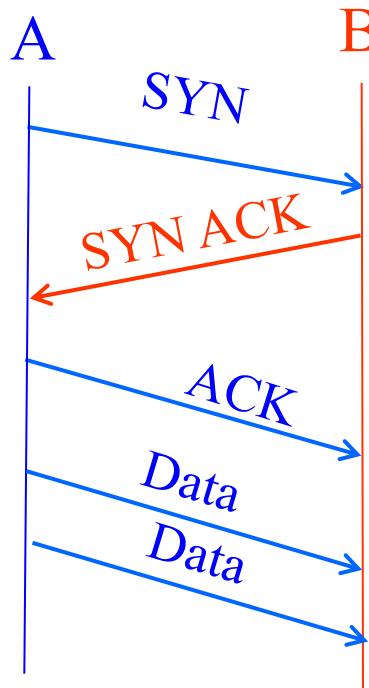
- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (future lecture)
- “Connection” set-up & tear-down

Functionality

- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (future lecture)
- “Connection” set-up & tear-down

TCP Connection Establishment and Initial Sequence Numbers

Establishing a TCP Connection

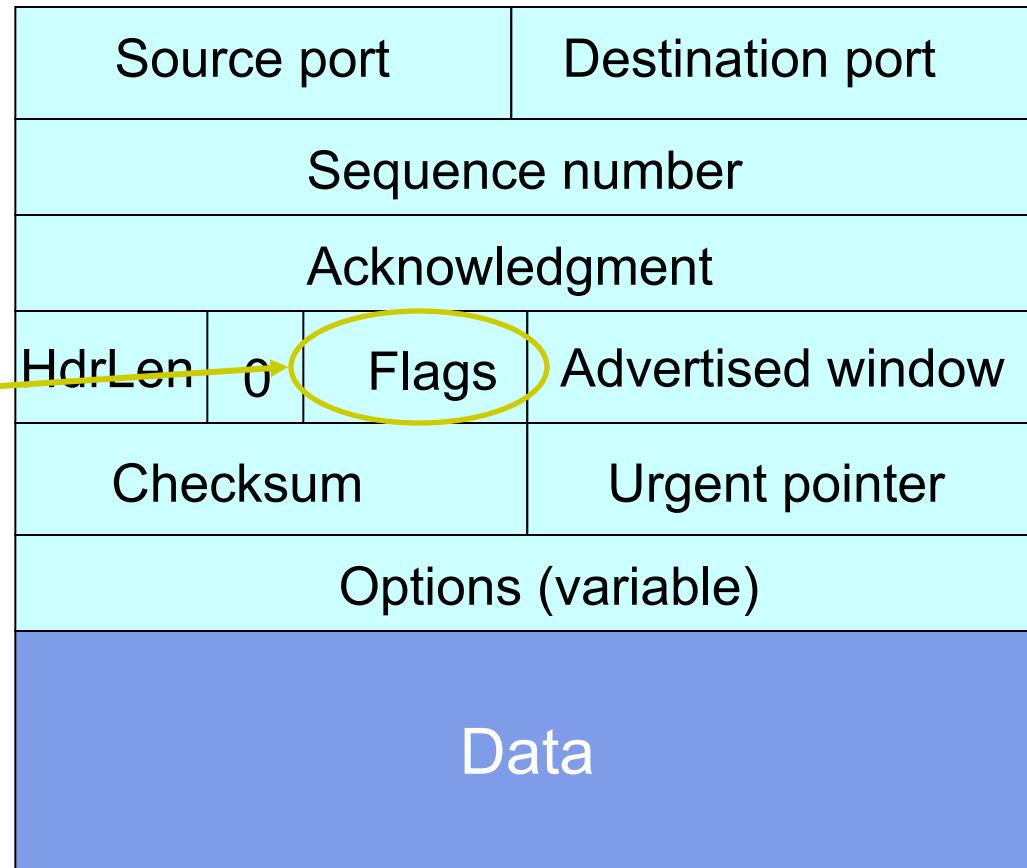


Each host tells its ISN to the other host.

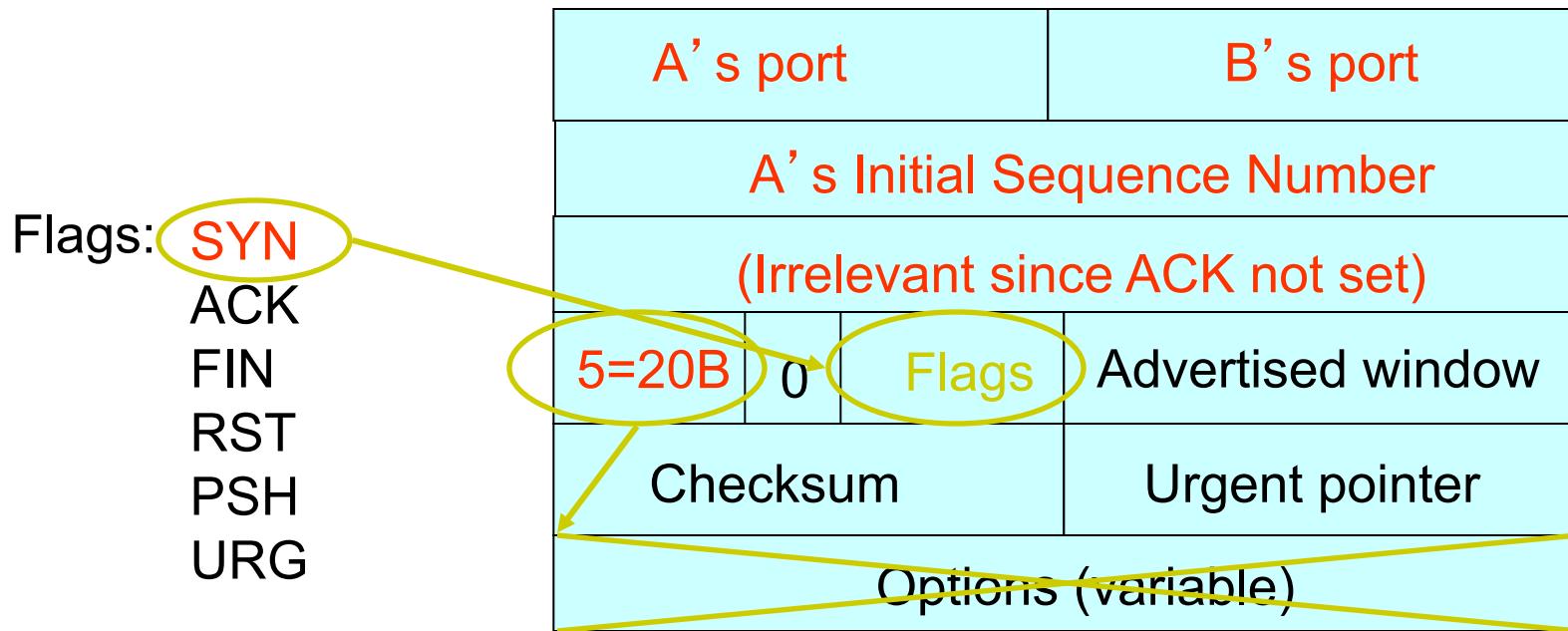
- Three-way handshake to establish connection
 - Host A sends a **SYN** to host B
 - Host B returns a SYN acknowledgment (**SYN ACK**)
 - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header

Flags: SYN
ACK
FIN
RST
PSH
URG

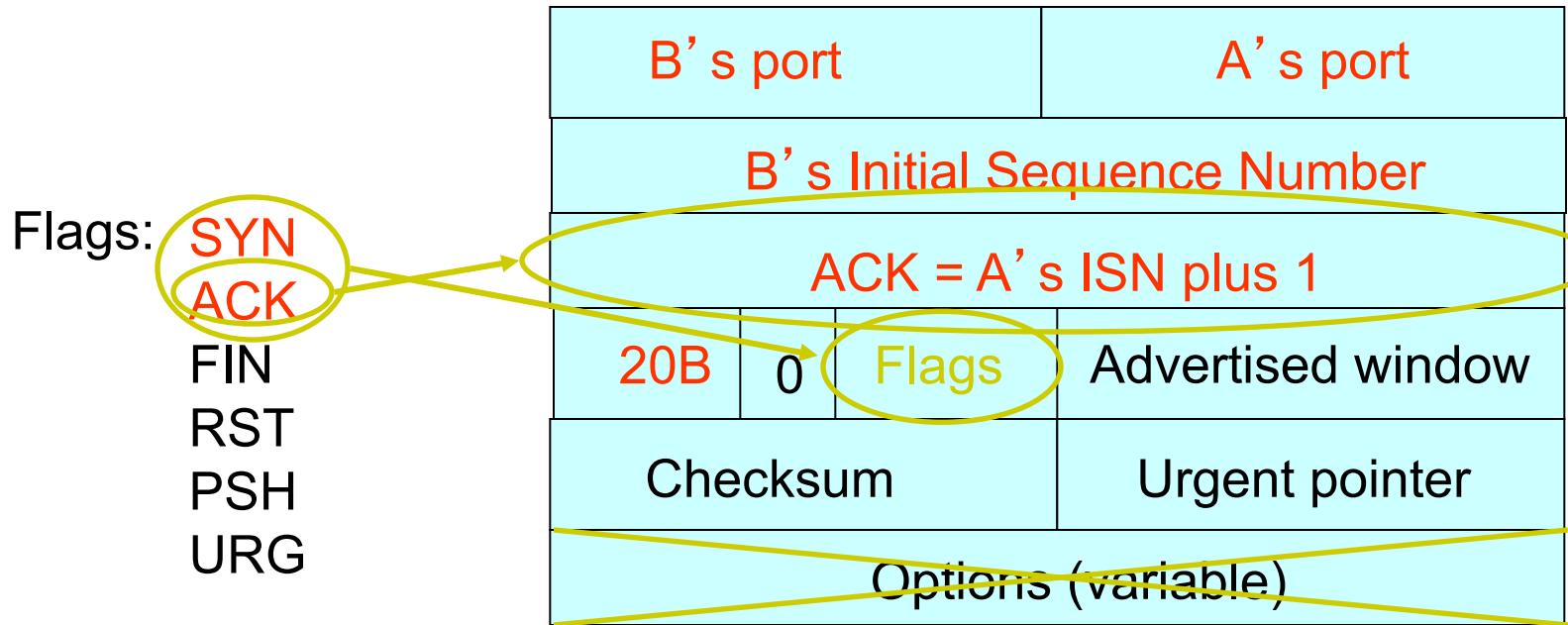


Step 1: A's Initial SYN Packet



A tells B it wants to open a connection...

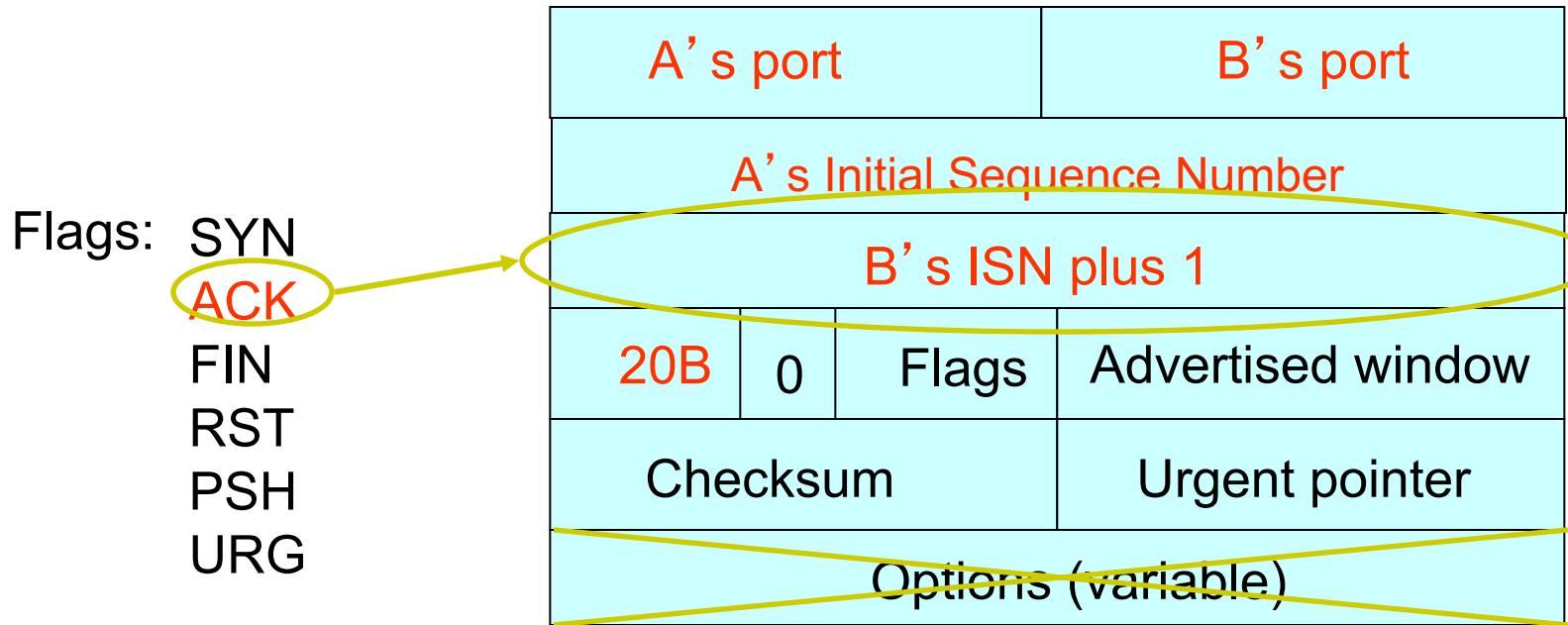
Step 2: B's SYN-ACK Packet



B tells A it accepts, and is ready to hear the next byte...

... upon receiving this packet, A can start sending data

Step 3: A's ACK of the SYN-ACK



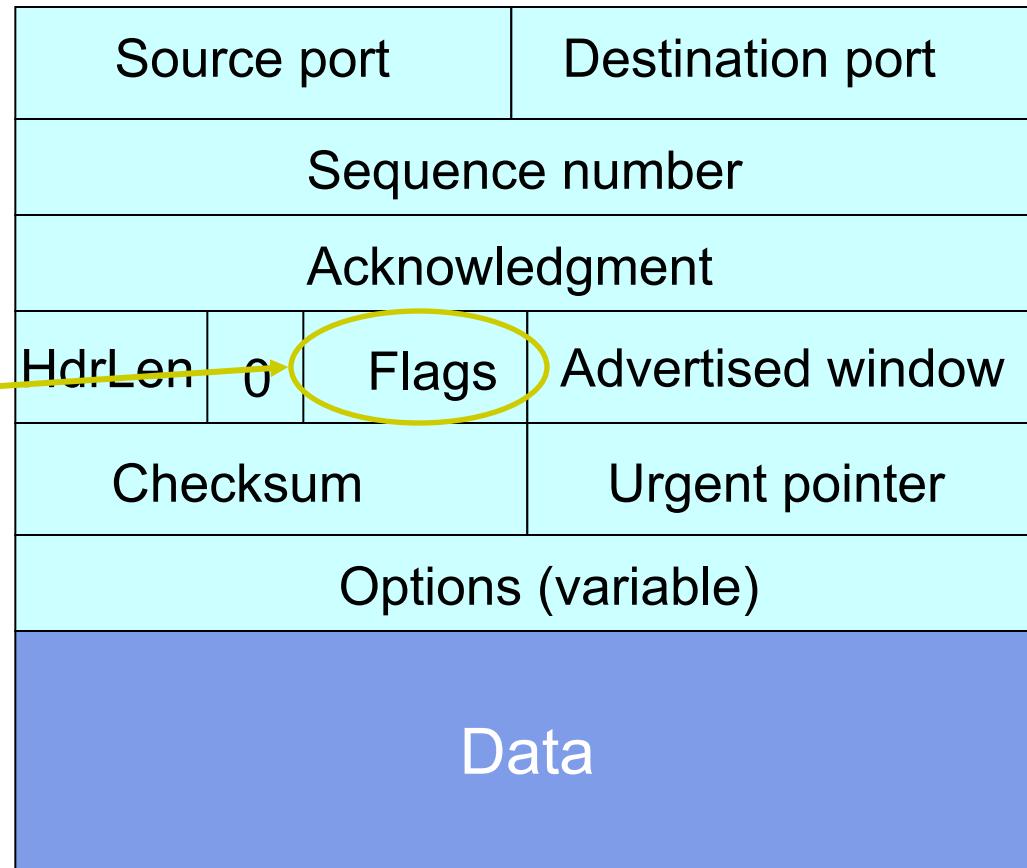
A tells B it's likewise okay to start sending

... upon receiving this packet, B can start sending data

Tearing Down the Connection

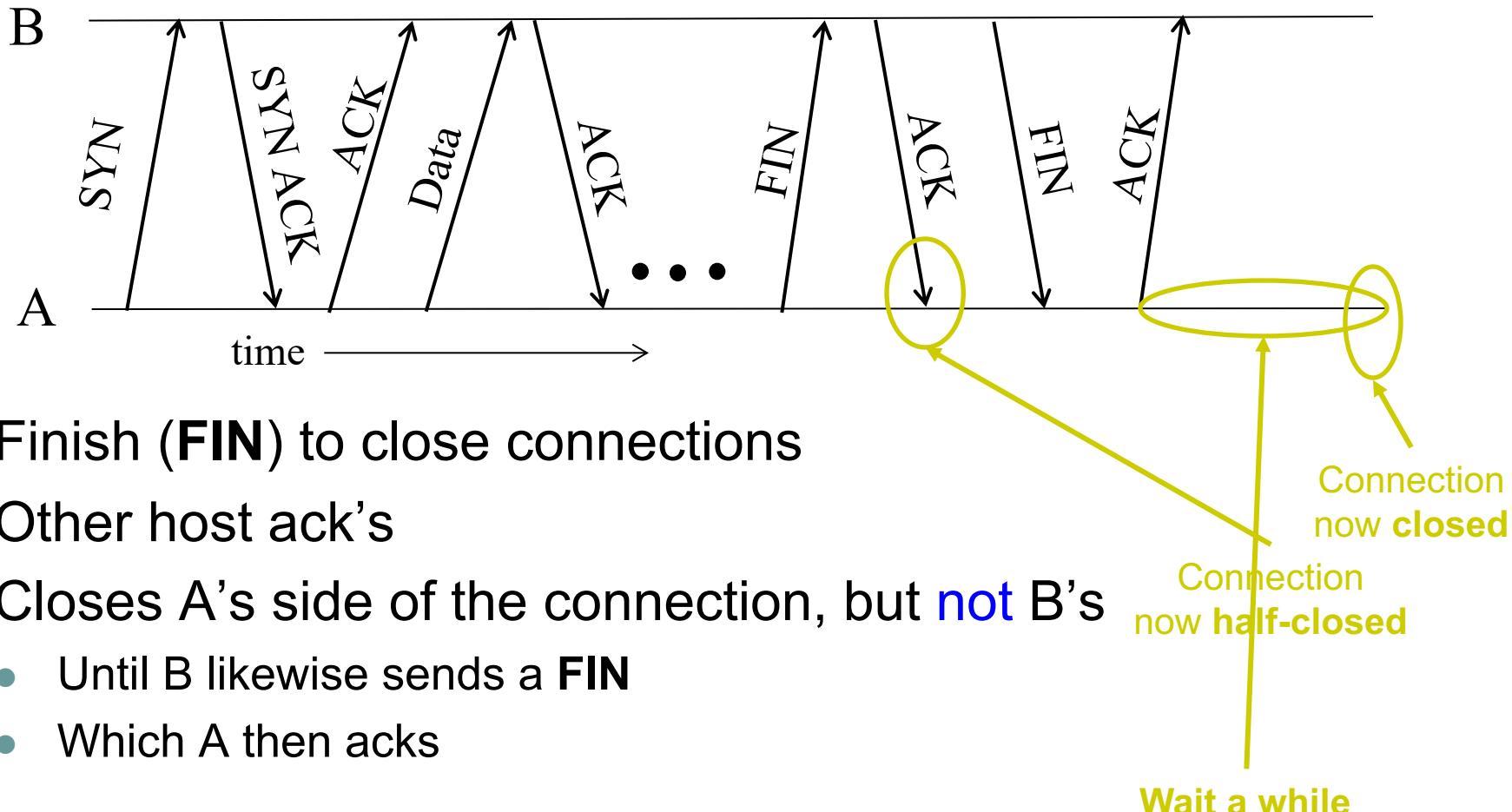
TCP Header

Flags: **SYN**
ACK
FIN
RST
PSH
URG

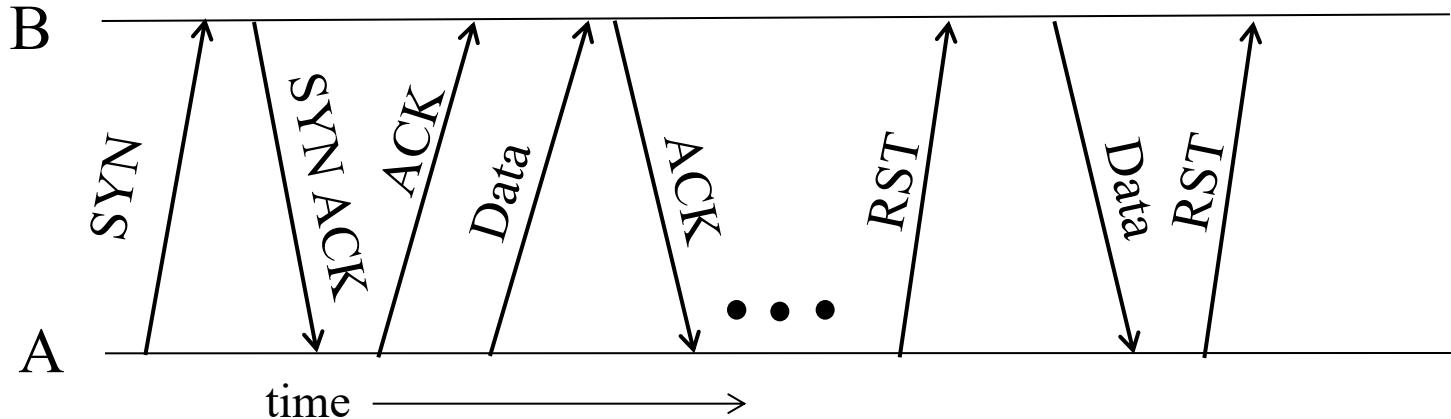


See /usr/include/netinet/tcp.h on Unix Systems

Normal Termination, One Side At A Time

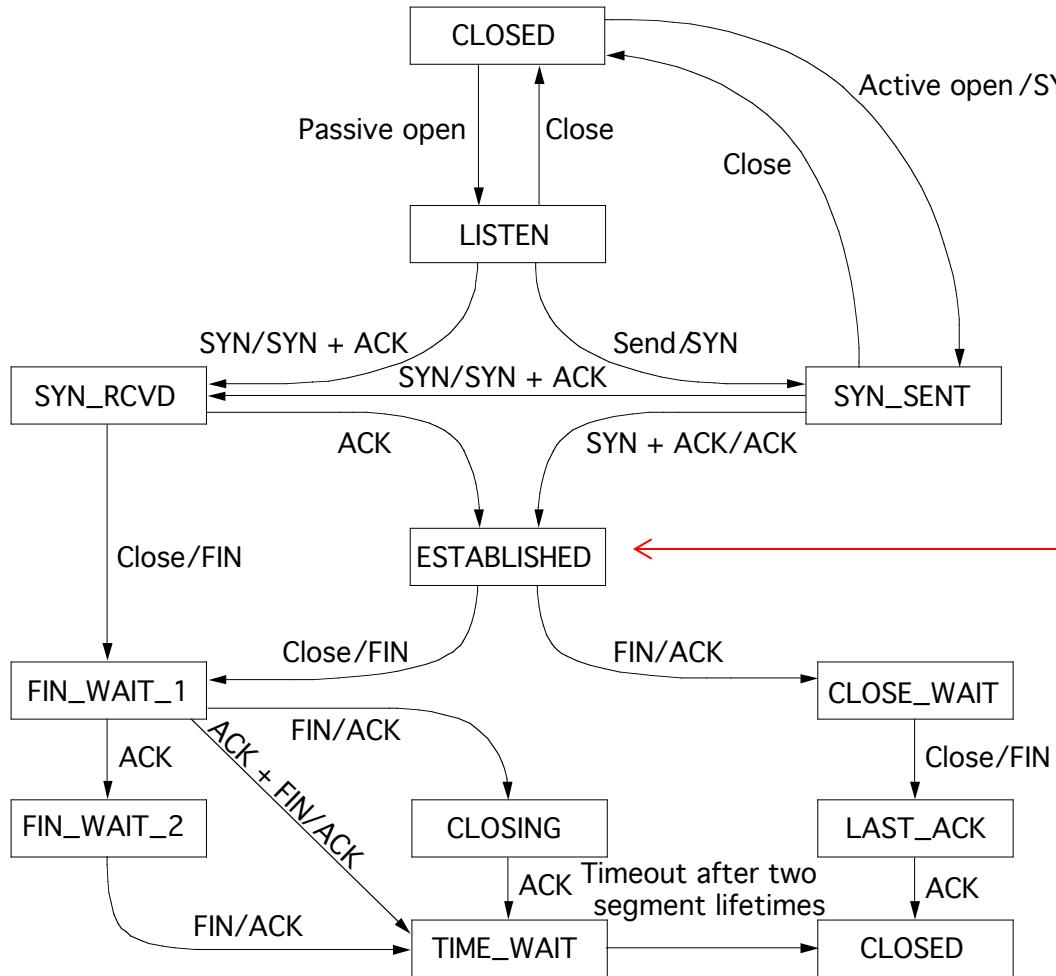


Abrupt Termination



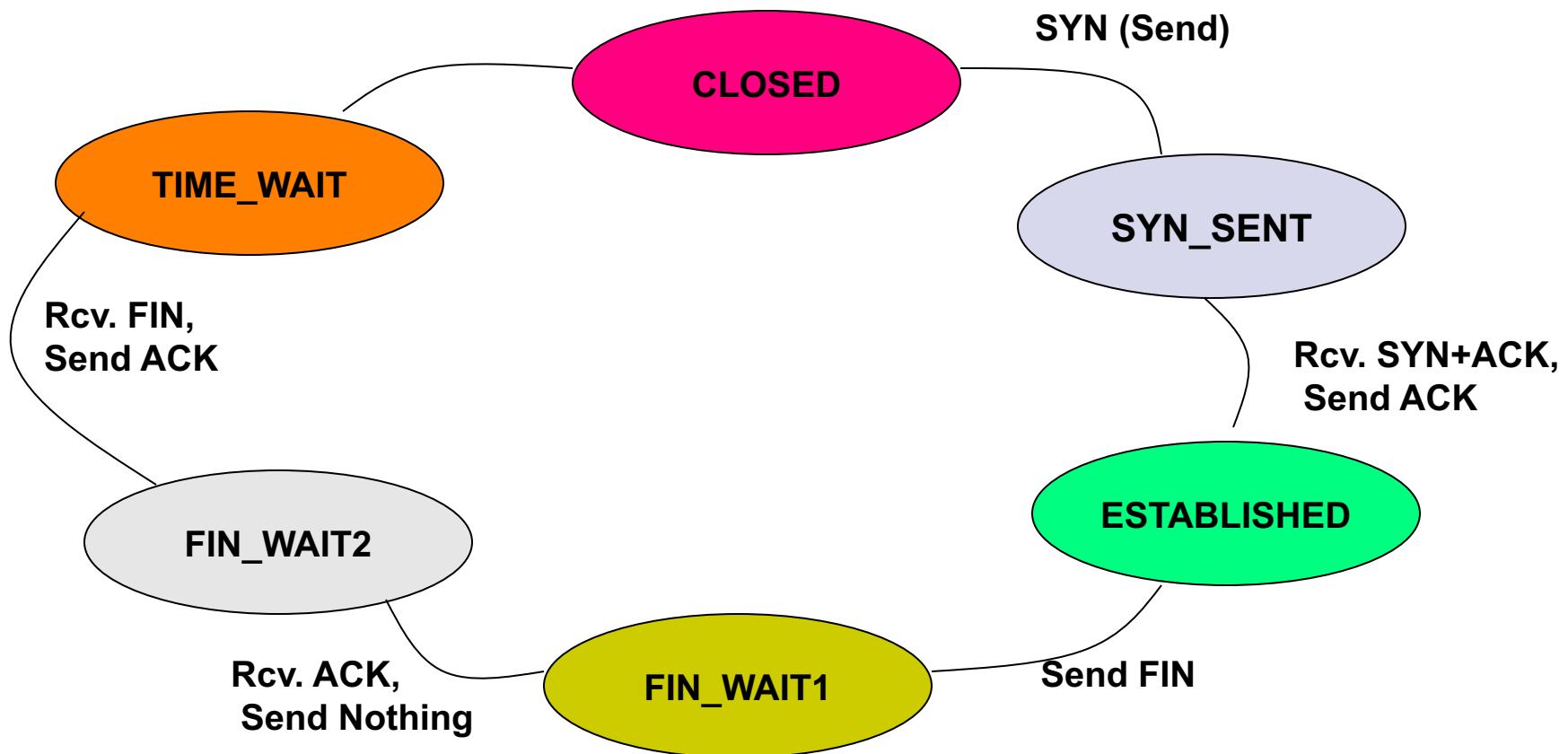
- A sends a RESET (**RST**) to B
 - E.g., because A restarted
- **That's it**
 - B does not ack the **RST**
 - Thus, **RST** is not delivered reliably
 - And: any data in flight is lost
 - If B sends anything more, will elicit another **RST**

TCP State Transitions



Data, ACK exchanges are in here

An Simpler View of the Client Side



In Summary

- TCP
 - An elegant (though not perfect) piece of engineering that has stood the test of time
 - Thought experiment: will TCP continue to be a good solution?
 - Plenty of evolution in individual pieces
 - **Congestion control**
 - Better acknowledgements, ISN selection, timer estimation, etc.
 - But core architectural decisions/abstractions remain
 - Bytestreams, connection oriented, windows etc.
- Next time: start on congestion control!

Any Questions?

Congestion Control

CS 168

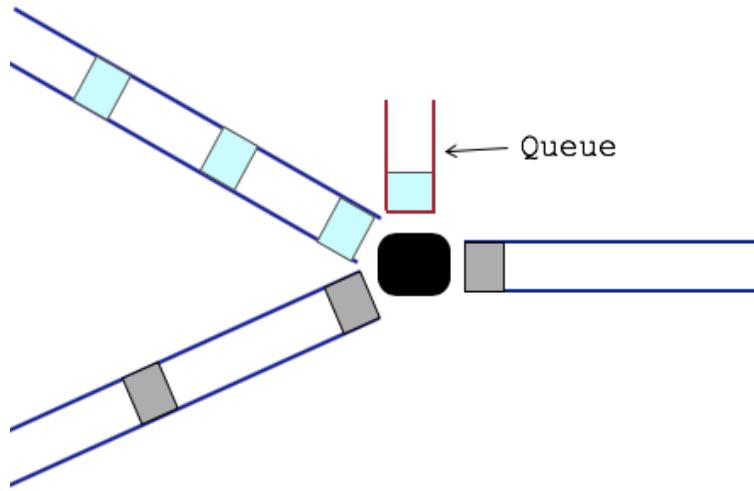
<http://cs168.io>

Sylvia Ratnasamy

Today: Congestion Control

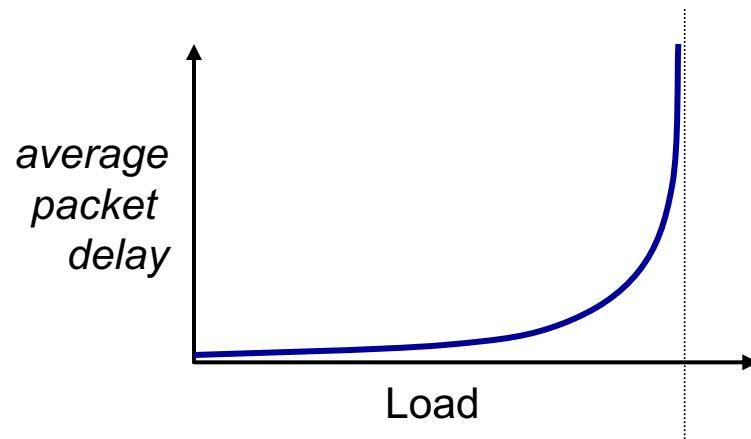
- One of the "core" topics in networking
 - Will occupy us for ~3 lectures
- Today: concepts and design space
 - Thu: CC in TCP
 - Next week: advanced CC

Recall: Lecture 3



- If two packets arrive at a router at the same time, the router will transmit one and buffer the other
- If many packets arrive close in time
 - the router cannot keep up → gets **congested**
 - causes packet **delays** and **drops**

Congestion is harmful



Typical queuing system with bursty arrivals

Some History: TCP in the 1980s

- Sending rate only limited by flow control
 - Dropped packets → senders retransmit, repeatedly!
- Led to “congestion collapse” in Oct. 1986

In October of '86, the Internet had the first of what became a series of ‘congestion collapses’. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. In particular, we wondered if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was “yes”.

Van Jacobson



- Researcher in the networking group at LBL
- Many contributions to the early TCP/IP stack
- Creator of many widely used network tools
 - [traceroute](#), [tcpdump](#), [Berkeley Packet Filter](#), ...
- Later Chief Scientist at Cisco, now at Google
 - Recently: [BBR](#), a new TCP CC protocol used by Google

Their Approach

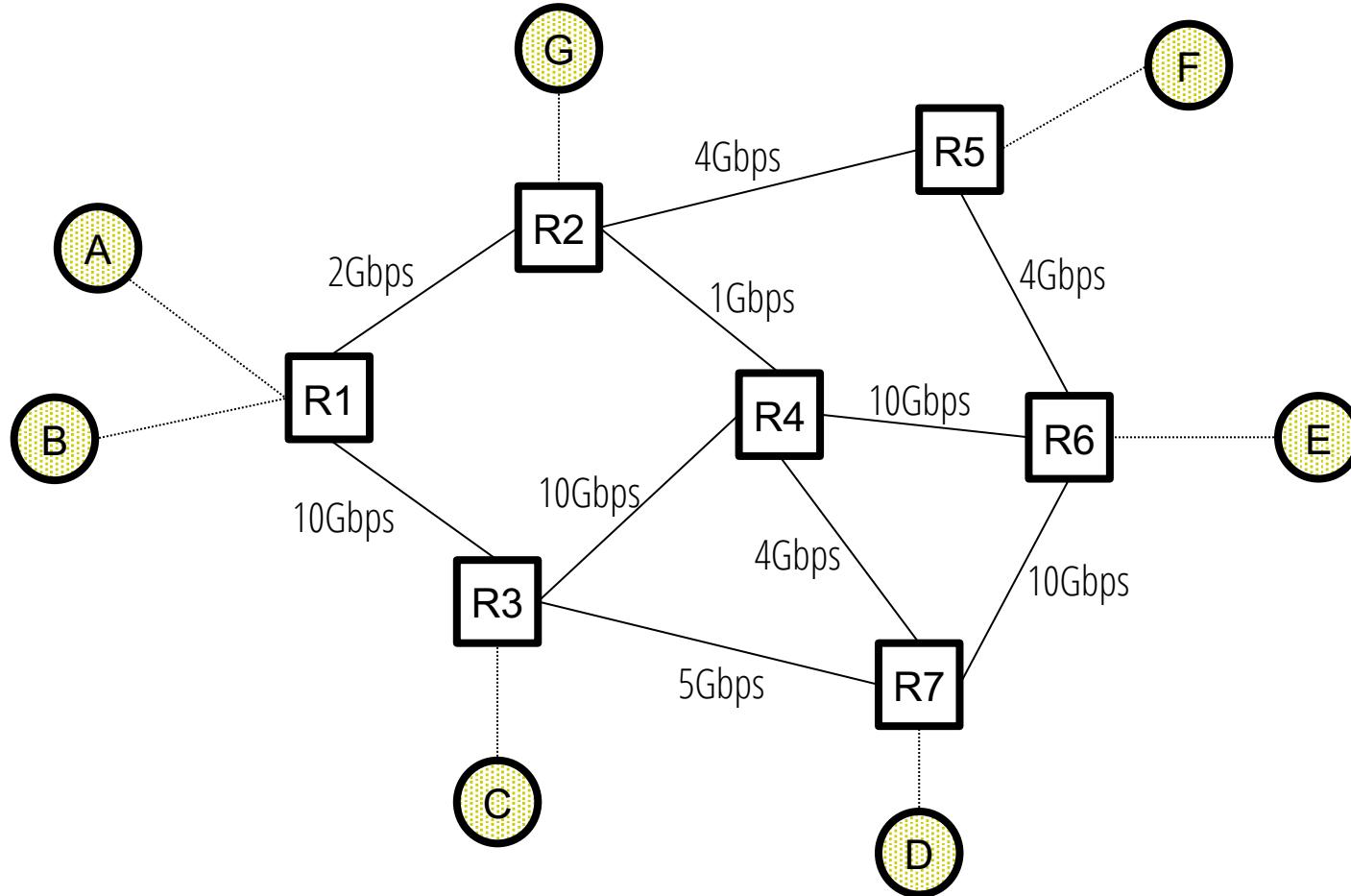
- Incremental extension to TCP's existing protocol
 - Source adjusts its window size based on observed packet loss
- A pragmatic and effective solution
 - Required no upgrades to routers or applications!
 - Patch of a few lines of code to BSD's TCP implementation
 - Quickly adopted and has been the de-facto approach since
 - A lesson on wisdom in system design
- Extensively researched and improved upon
 - Countless variants (we will not discuss)
 - As well as radically different approaches (we'll see a few next week)

CC more generally...

- Huge literature on the problem
 - In systems, control theory, game theory, stats, econ
- Recent resurgence of interest in industry
 - New pressure for high-performance (cloud services)
 - New context (datacenters, new app workloads)
 - New methods (ML)

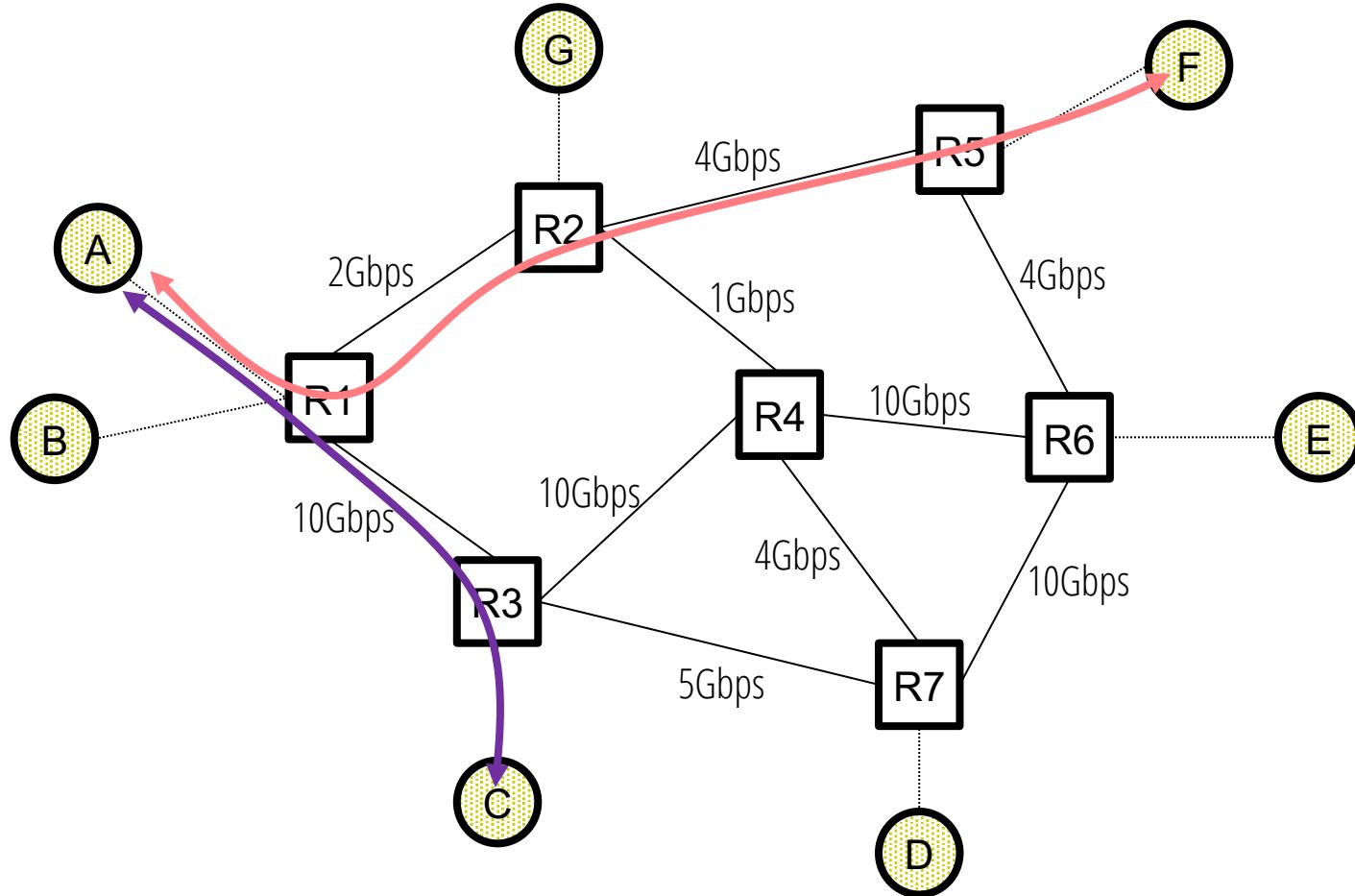
Topics for today

- What makes CC a hard problem?
- Goals for a good solution
- Design space
- Components of a solution
- TCP's approach (high level)
- Next week:
 - TCP CC in detail
 - Advanced topics in CC

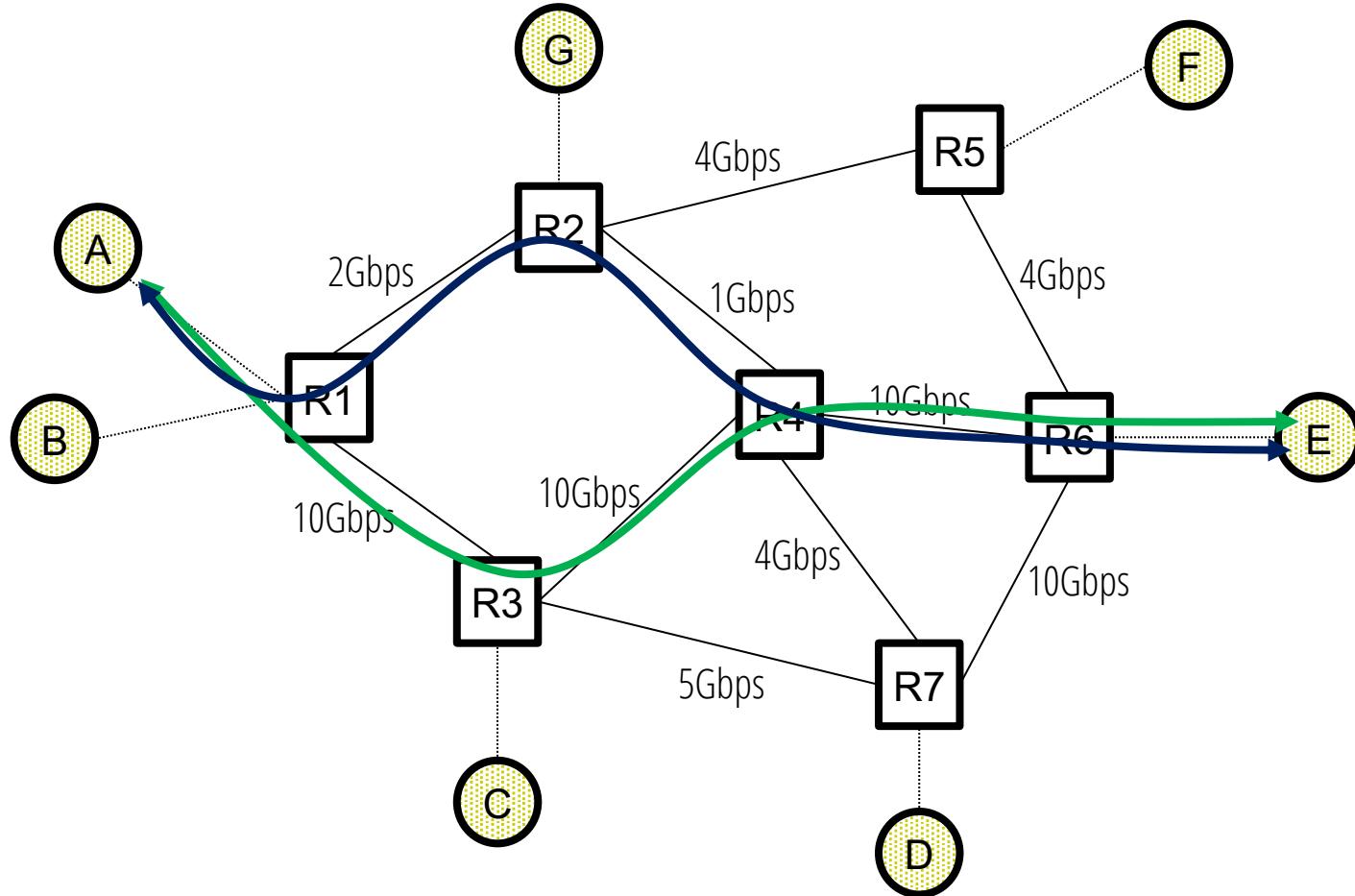


At what rate should Host A send traffic?

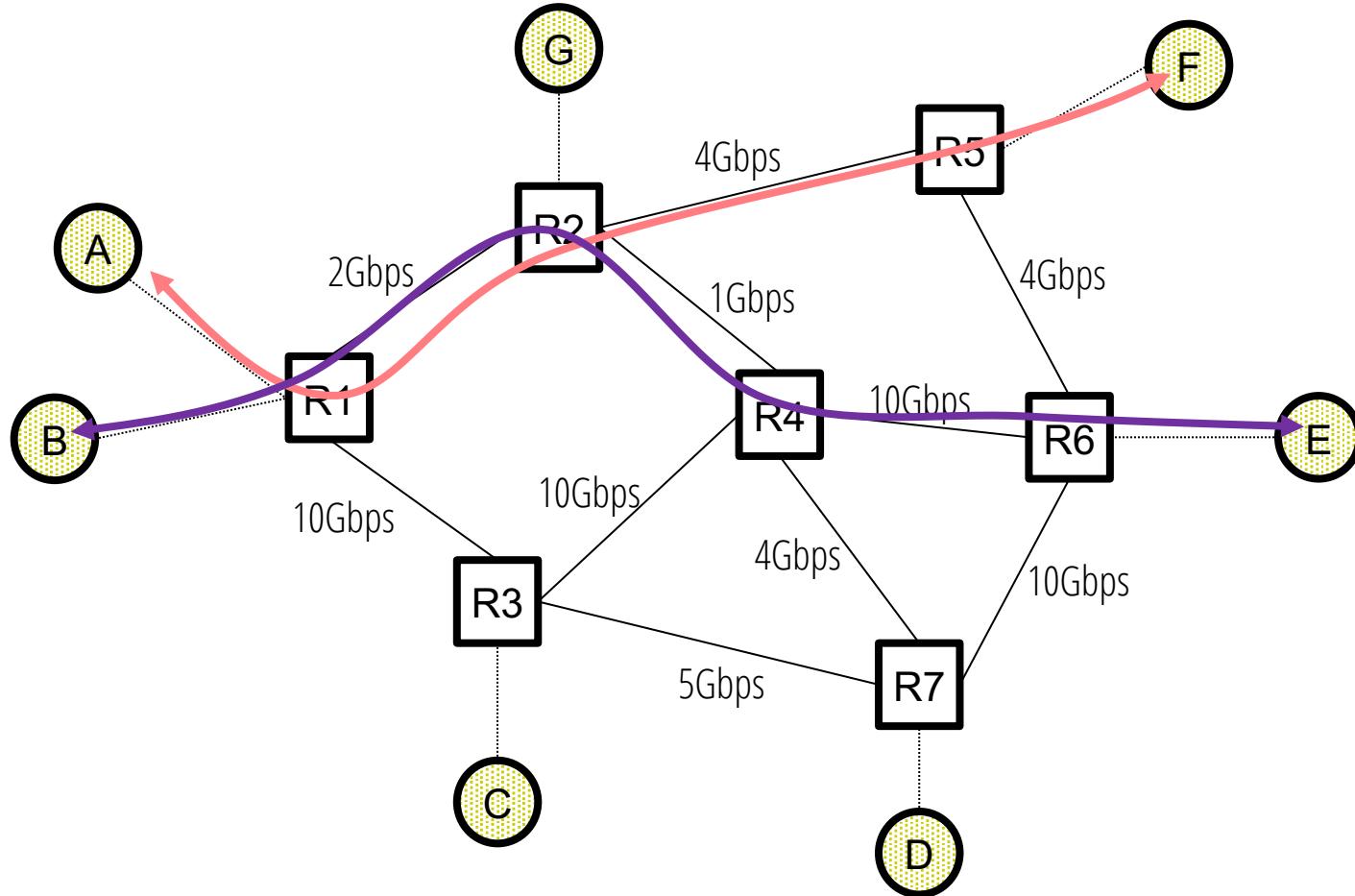
*For this example, we'll ignore the BW of links attaching hosts to routers



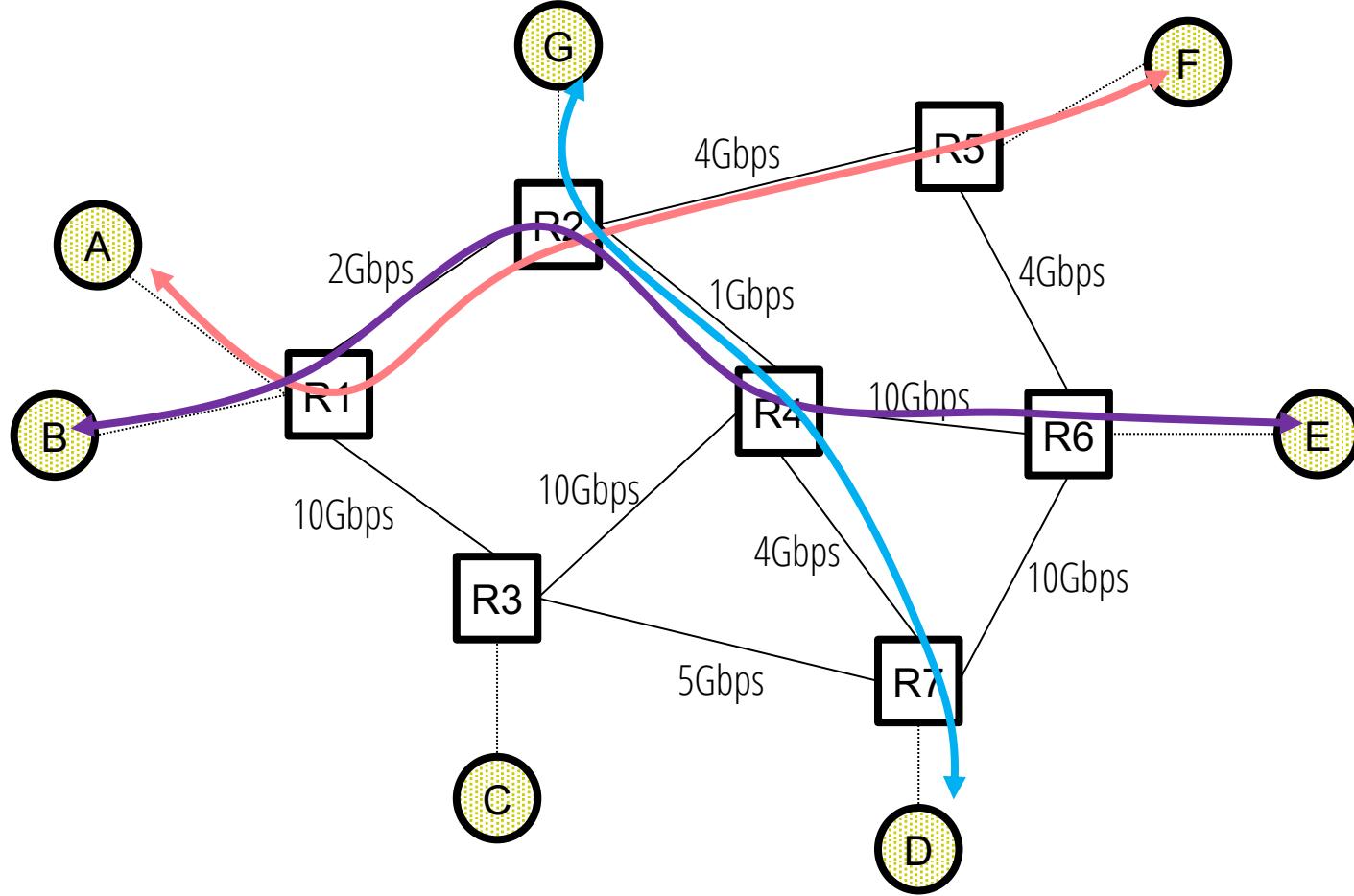
Depends on the destination



Changes with routing dynamics



Depends on “competing” flows



Including “indirect” competition!

Congestion Control

- Fundamentally, a resource allocation problem
 - Flow is assigned a shared of the link BW along a path
- But more complex than traditional resource alloc.
 - Changing one link's allocation can have global impact
 - And we're changing allocations on every flow arrival/exit
 - No single entity has a complete view or complete control!
 - (Exception: within a datacenter)
- Allocations in our context are highly **interdependent**

Outline for today

- What makes CC a hard problem?
- Goals for a good solution
- Design space
- TCP's approach (high level)
- Components of a solution

Goals

- From a resource allocation perspective
 - Low packet delay and loss
 - High link utilization
 - “Fair” sharing across flows

Aim: a good tradeoff between the above goals

Goals

- From a resource allocation perspective
 - Low packet delay and loss
 - High link utilization
 - “Fair” sharing across flows
- From a systems perspective
 - **Practical:** scalable, decentralized, adaptive, etc.

Any questions?

Outline for today

- What makes CC a hard problem?
- Goals for a good solution
- Design space
- TCP's approach (high level)
- Components of a solution

Possible Approaches

(0) Send at will



What happens if A sends at 10Gbps?

Possible Approaches

(1) Reservations

- Pre-arrange bandwidth allocations
- Comes with all the problems we've discussed

Possible Approaches

(1) Reservations

(2) Pricing / priorities

- Don't drop packets for the highest bidders/priority users
- Charge users based on current congestion levels
- Requires payment model

Possible Approaches

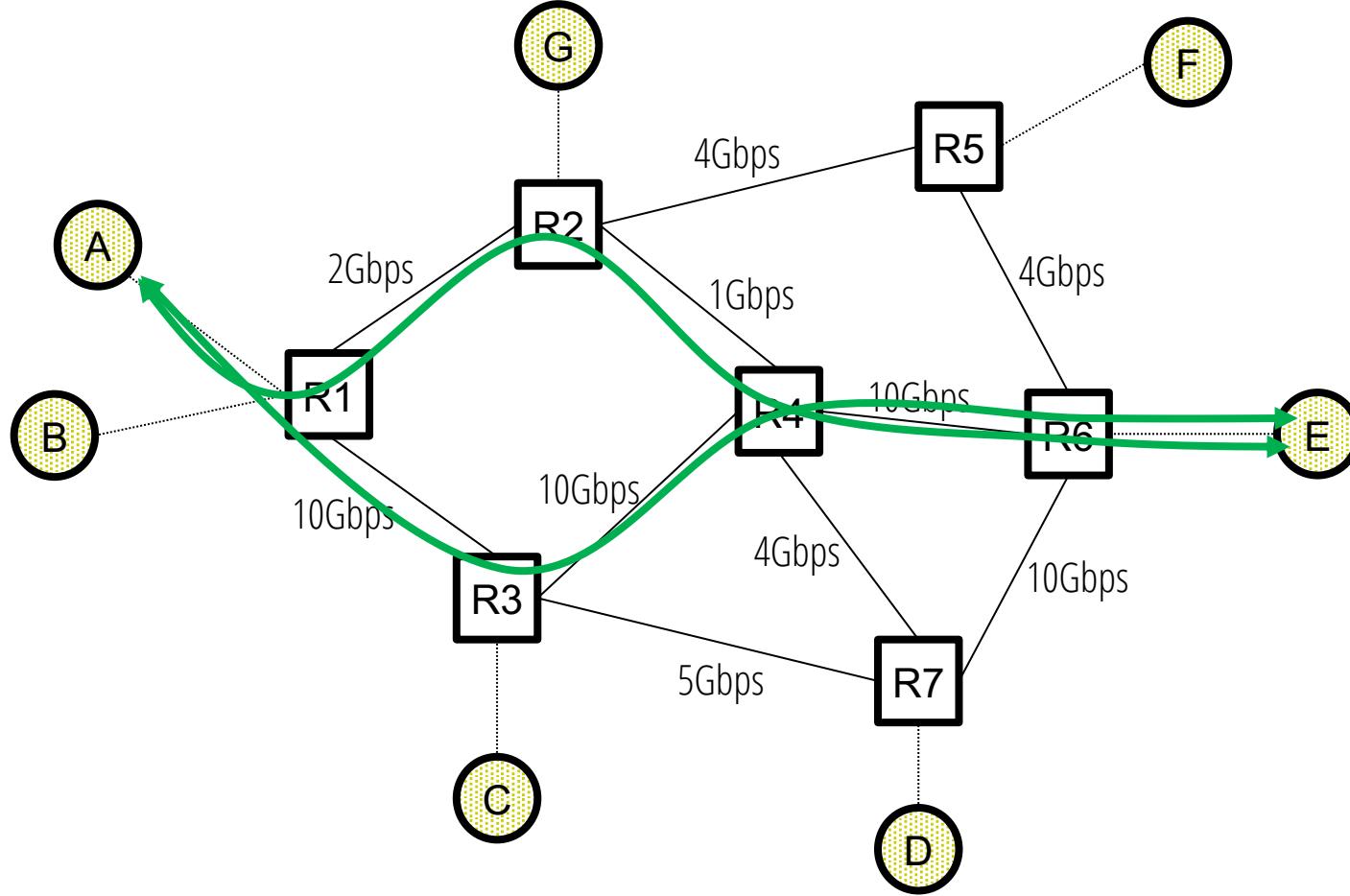
- (1) Reservations
- (2) Pricing / priorities
- (3) Dynamic Adjustment
 - Hosts dynamically learn current level of congestion
 - Adjust their sending rate accordingly
 - Many options for how to implement this basic idea

Possible Approaches

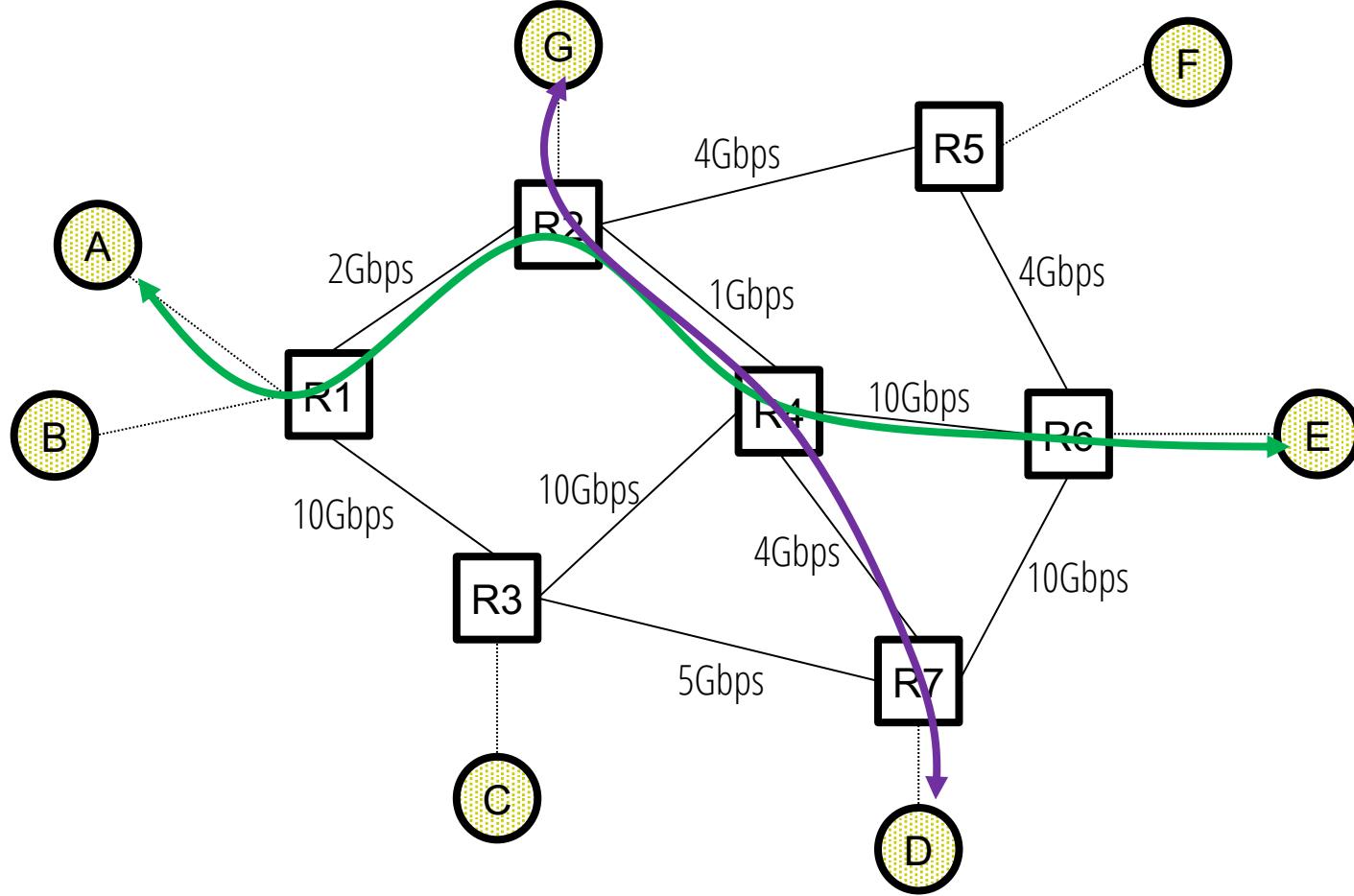
- (1) Reservations
- (2) Pricing / priorities
- (3) Dynamic Adjustment

In practice, the **generality** of dynamic adjustment has proven powerful

- Doesn't presume business model
- Doesn't assume we know app/user requirements
- But does assume good citizenship!



- (1) First, host A discovers it can send at ~10Gbps
- (2) A notices that ~10Gbps is congesting the network
- (3) A figures out it should cut its rate to ~1Gbps



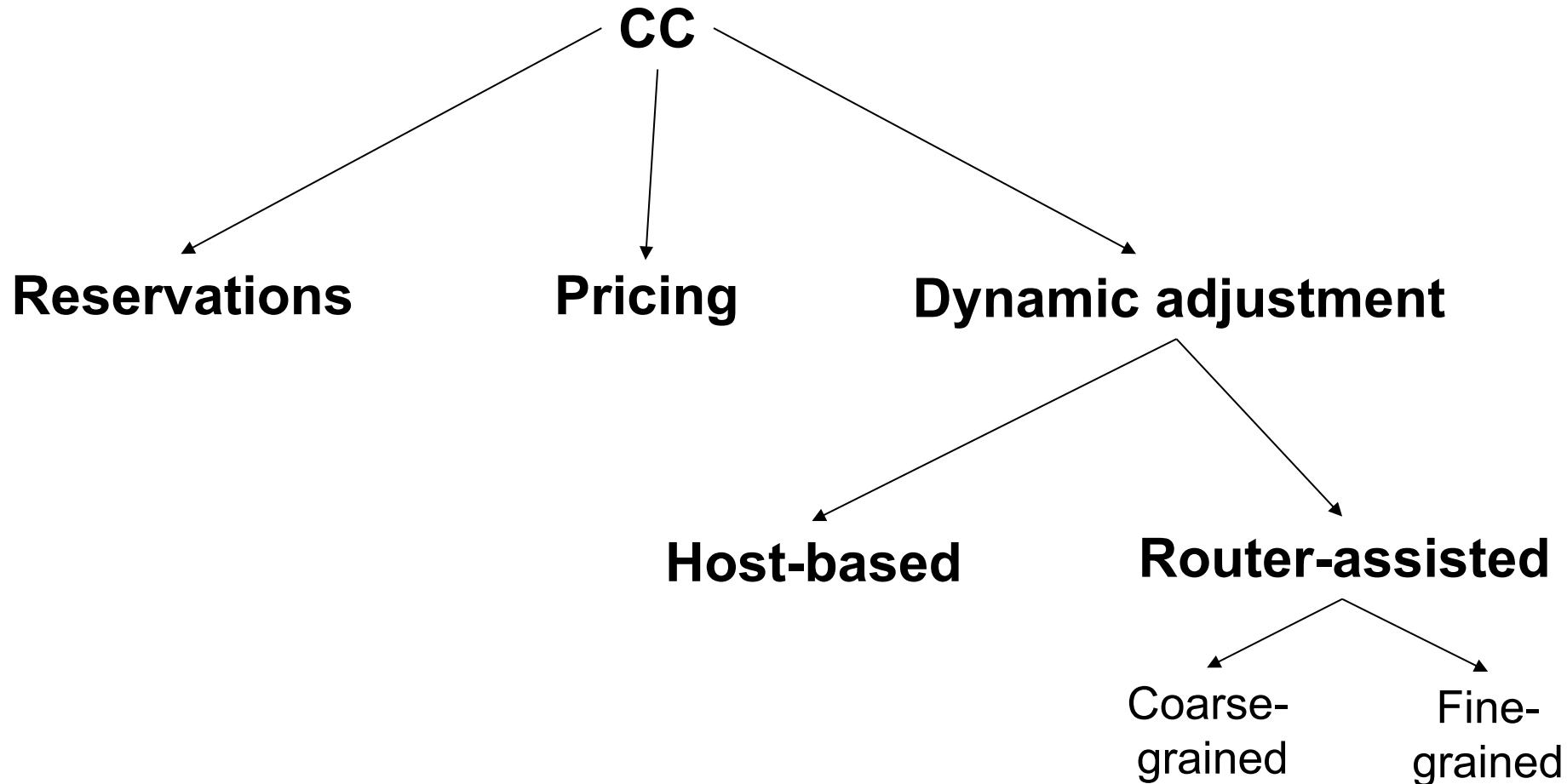
(4) A notices that 1Gbps is congesting the network

(5) A figures out it should cut its rate to (say) $\frac{1}{2}$ Gbps

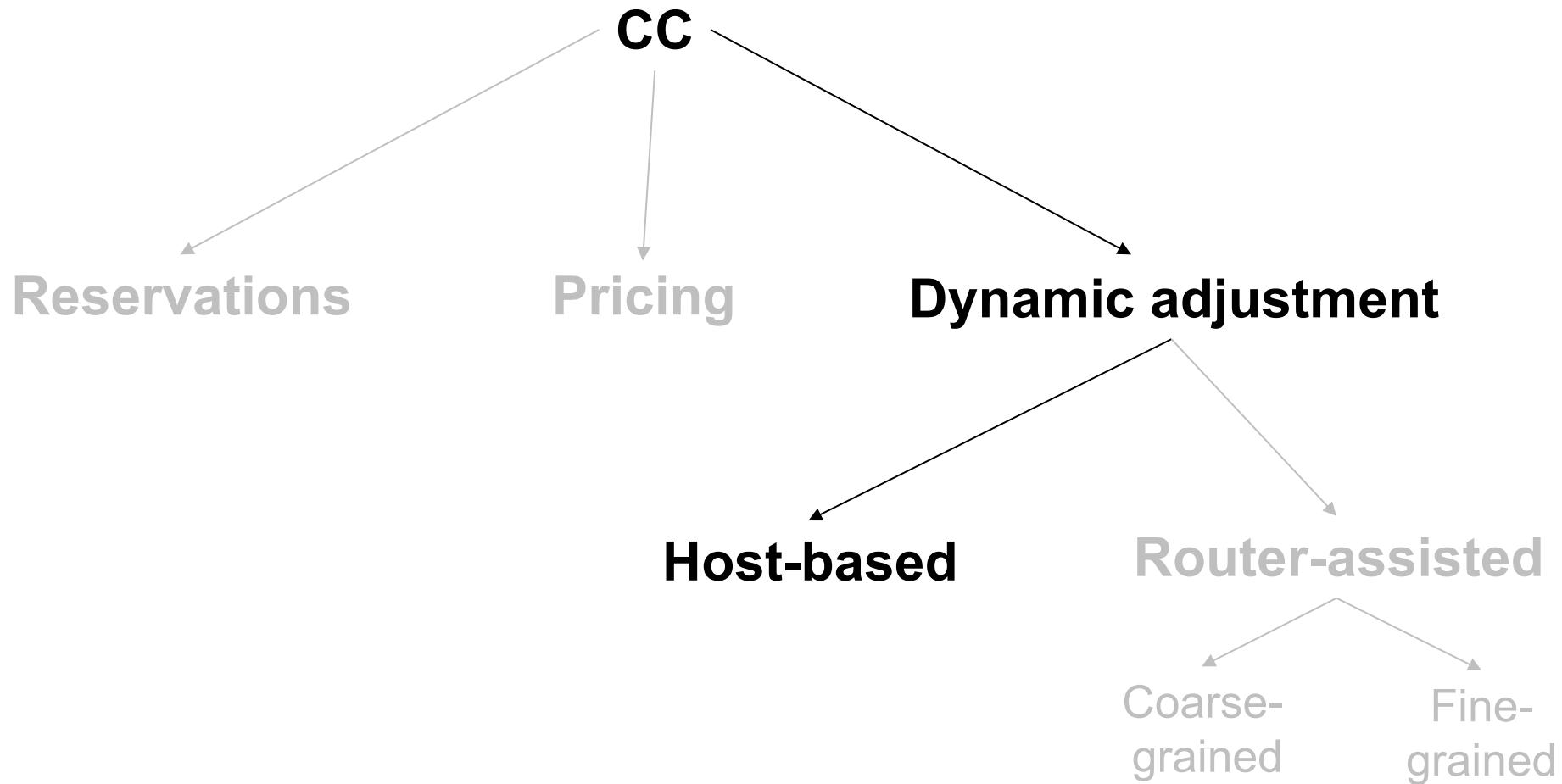
Two broad classes of solutions

- **Host-based CC → Jacobson's original TCP approach**
 - No special support from routers
 - Hosts adjust rate based on implicit feedback from routers
- **Router-assisted CC**
 - Routers signal congestion back to hosts
 - Hosts pick rate based on explicit feedback from routers
- We'll study TCP's host-based approach in detail and a bit of router-assisted CC

Taking stock: where we are in the design space



Taking stock: where we are in the design space



Sketch of a (host-based) solution

How do we pick the initial rate? runs the following:

- **Pick initial rate R**
- **Try sending at a rate R for some period of time**
 - **Did I experience congestion in this time period?**
 - If yes, reduce R
 - If no, increase R
 - **Repeat**

How do we detect congestion

By how much should
we increase/decrease

Components of a Solution

- Discovering an initial rate
- Detecting congestion
- Reacting to congestion (or lack thereof)
 - Increase/decrease rules

Detecting Congestion?

- **Packet loss**
 - Approach commonly used by TCP
- **Benefits**
 - Fail-safe signal
 - Already something TCP detects to implement reliability
- **Cons**
 - Complication: non-congestive loss (e.g., checksum err.)
 - Complication: reordering (e.g., with cumulative ACKs)
 - Detection occurs after packets have experienced delay

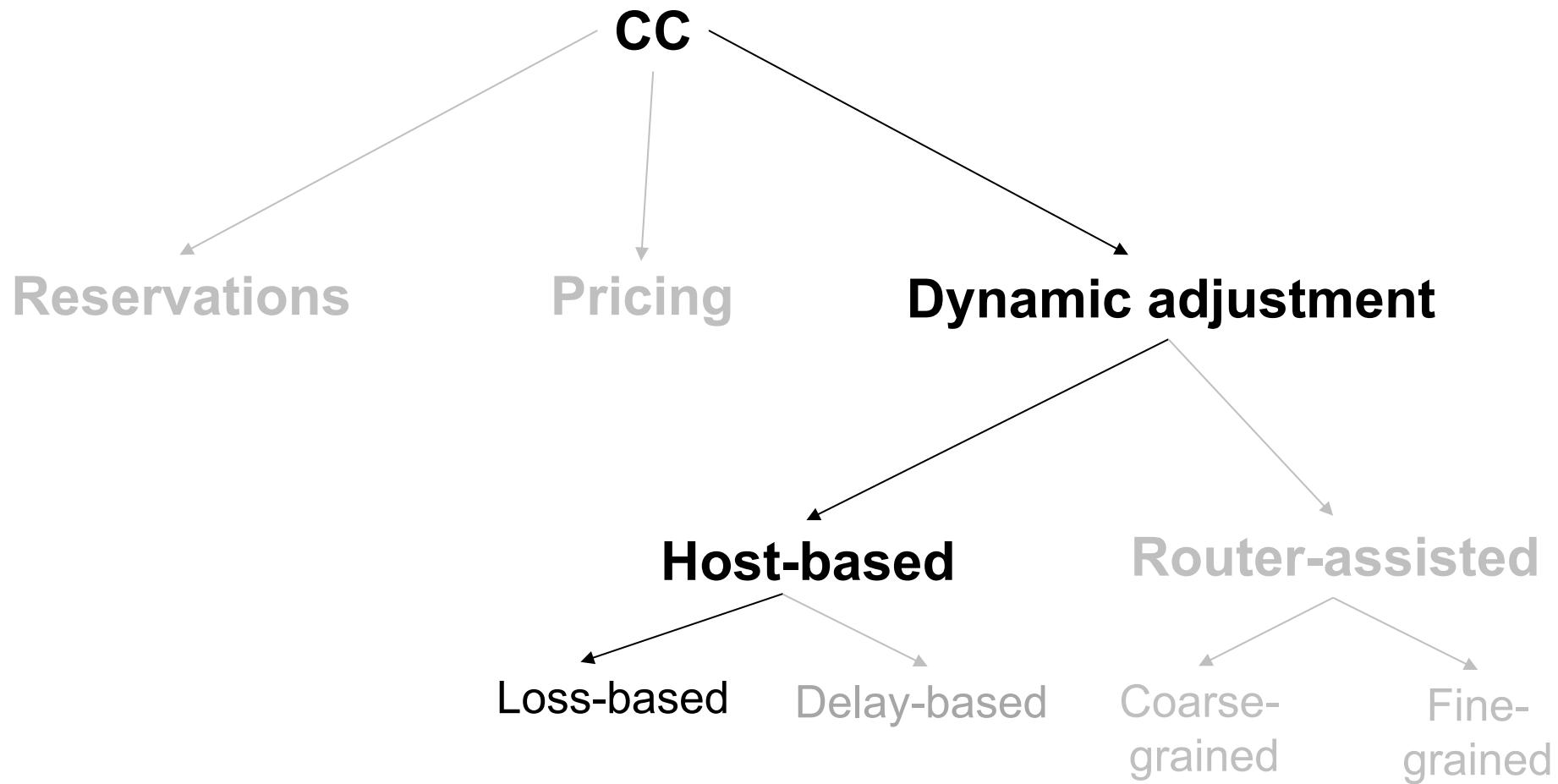
Detecting Congestion?

- **Increase in packet delay**
 - Long considered tricky to get right: packet delay varies with queue size and competing traffic
 - Google's new BBR protocol is now challenging this assumption (next week)

Note: Not All Losses the Same

- **Duplicate ACKs:** isolated loss
 - Packets and ACKs still getting through
 - Suggests mild congestion levels
- **Timeout:** much more serious
 - Not enough packets/dupACKs getting through
 - Must have suffered several losses
- We'll see that TCP reacts differently in each case

Taking stock: where we are in the design space



Discovering an initial rate?

- Goal: estimate available bandwidth
 - Start slow (for safety)
 - But ramp up quickly (for efficiency)
- Toy example (of an inefficient solution)
 - Add $\frac{1}{2}$ Mbps every 100ms until we detect loss
 - If available BW is 1Mbps, will discover rate in 200ms
 - If available BW is 1Gbps, will take 200 seconds
 - Either is possible!

Solution: “Slow Start”

- Start with a small rate (hence the name)
 - Might be much less than actual bandwidth
 - Linear increase takes too long to ramp up
- Increase **exponentially** until first loss
 - E.g., double rate until first loss
- A “safe” rate is half of that when first loss occurred
 - I.e., if first loss occurred at rate R , then $R/2$ is safe rate

Components of a Solution

- Discovering an initial rate
- Detecting congestion
- Reacting to congestion (or lack thereof)
 - Increase/decrease rules

Sketch of a solution

Each source independently runs the following:

- **Pick initial rate R**
- **Try sending at a rate R for some time period**
 - **Did I experience congestion in this time period?**
 - **If yes, reduce R**
 - **If no, increase R**
- **Repeat**

By how much should
we increase/decrease?

Rate adjustment

- This is a critical part of a CC design!
- Determines how quickly a host adapts to changes in available bandwidth
- Determines how effectively BW is consumed
- Determines how BW is shared (fairness)

Goals for rate adjustment

- **Efficiency:** High utilization of link bandwidth
- **Fairness:** Each flow gets equal share

How should we adjust rate?

- Infinite options...
- At the highest level: fast or slow
- Fast: **multiplicative** increase/decrease
 - E.g., increase/decrease by 2x ($R \rightarrow 2R$ or $R/2$)
- Slow: **additive** increase/decrease
 - E.g., increase/decrease by +1 ($R \rightarrow R+1$ or $R-1$)

Leads to four alternatives

- **AIAD**: gentle increase, gentle decrease
- **AIMD**: gentle increase, rapid decrease
- **MIAD**: rapid increase, gentle decrease
- **MIMD**: rapid increase, rapid decrease

Leads to four alternatives

- **AIAD:** gentle increase, gentle decrease
- **AIMD:** gentle increase, rapid decrease
- **MIAD:** rapid increase, gentle decrease
- **MIMD:** rapid increase, rapid decrease

Why AIMD? Intuition

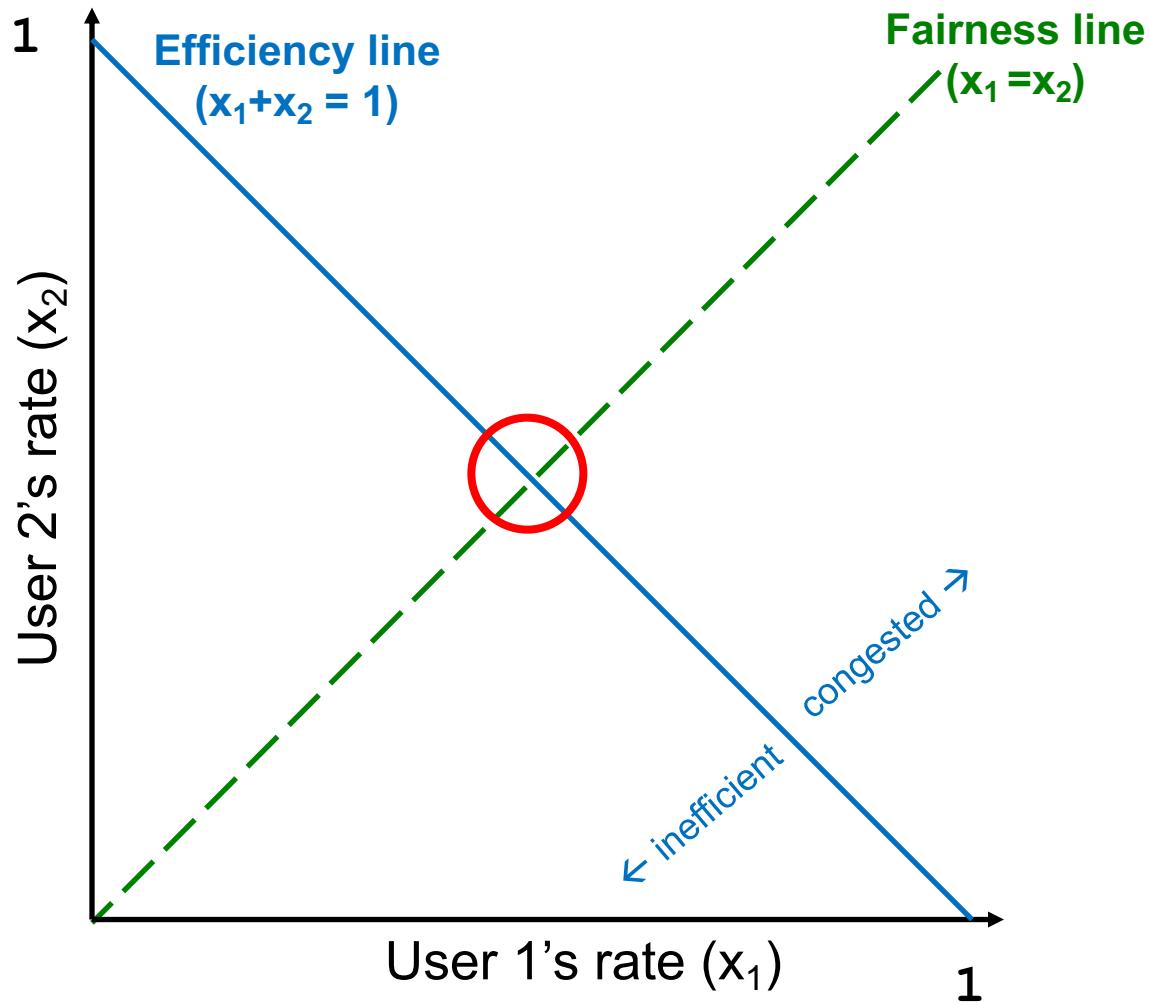
- Consequences of sending too much are worse than sending too little
 - Too much: packets dropped and retransmitted
 - Too little: somewhat lower throughput
- General approach:
 - Gentle increase when uncongested (exploration)
 - Rapid decrease when congested

Why AIMD? In more detail...

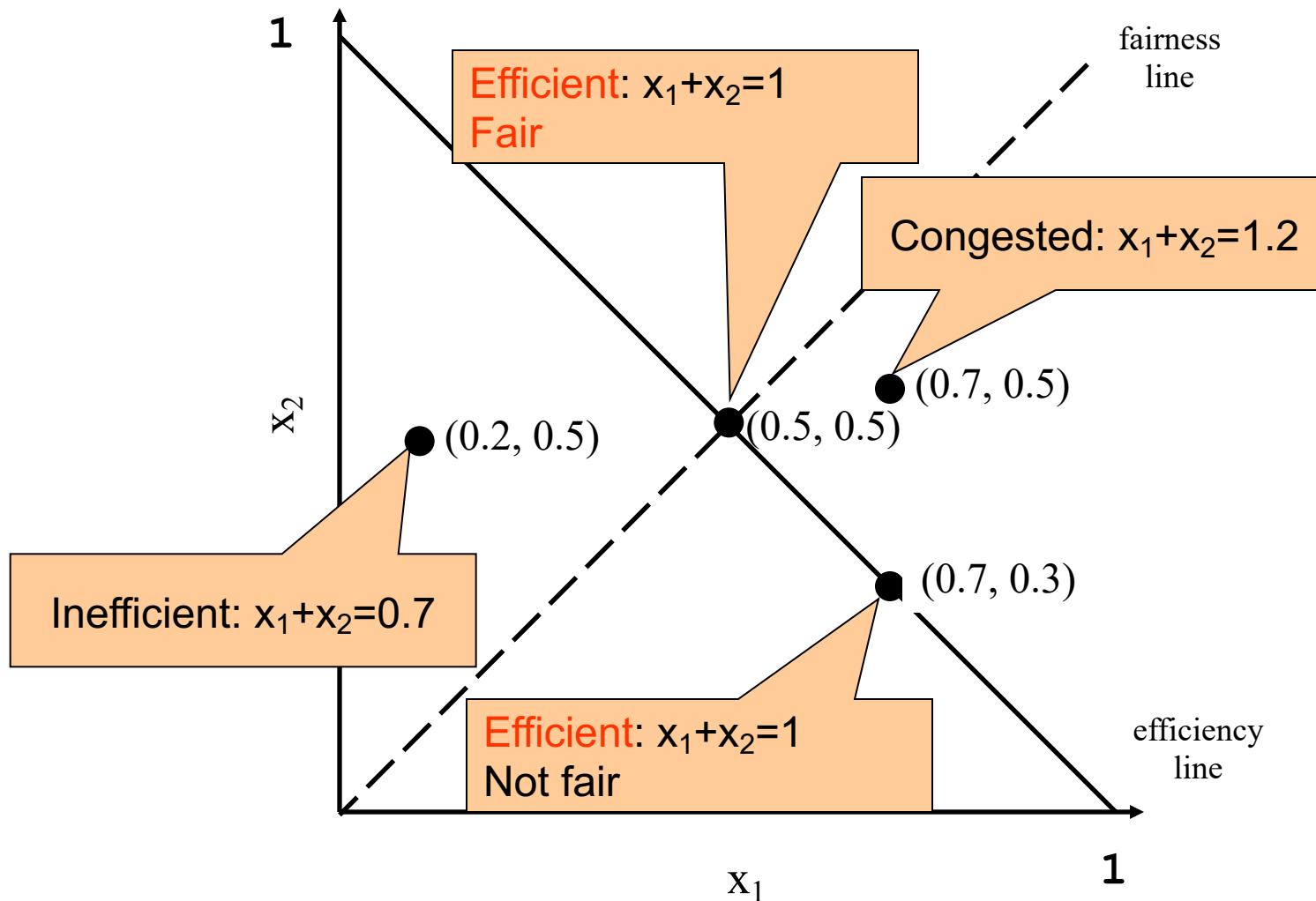
- Consider a simple model
 - Two flows going over single link of capacity C
 - Sending at rates X_1 and X_2 respectively
- When $X_1+X_2 > C$, network is congested
- When $X_1+X_2 < C$, network is underloaded
- Would like *both*:
 - $X_1 + X_2 = C \rightarrow$ link is fully utilized with no congestion
 - $X_1 = X_2 \rightarrow$ sharing is “fair”

Simple Model, C=1

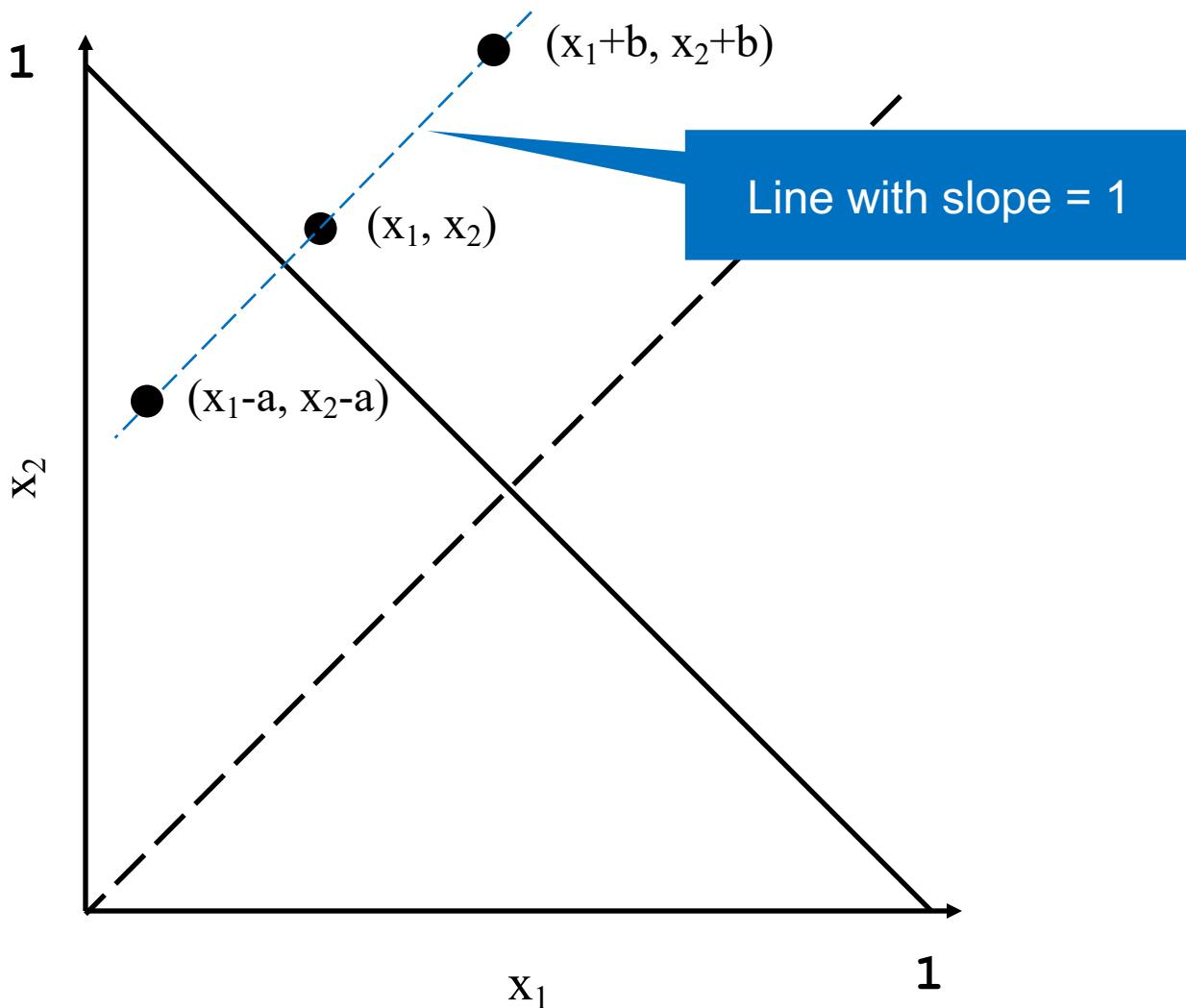
- Two users with rates x_1 and x_2
- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$
- Fair when $x_1 = x_2$



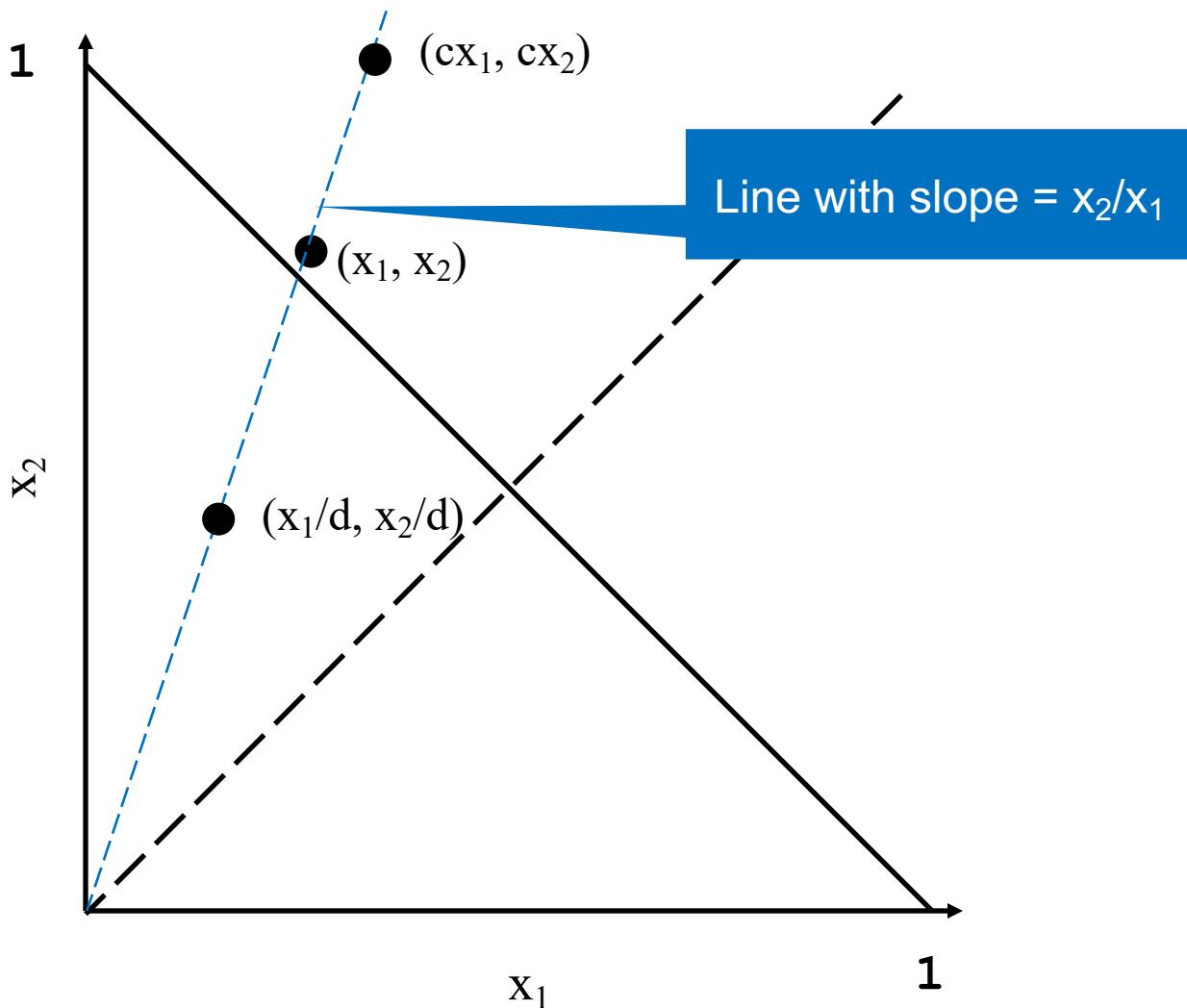
Example Allocations, C=1



Example Adjustments



Example Adjustments



Our Four Options

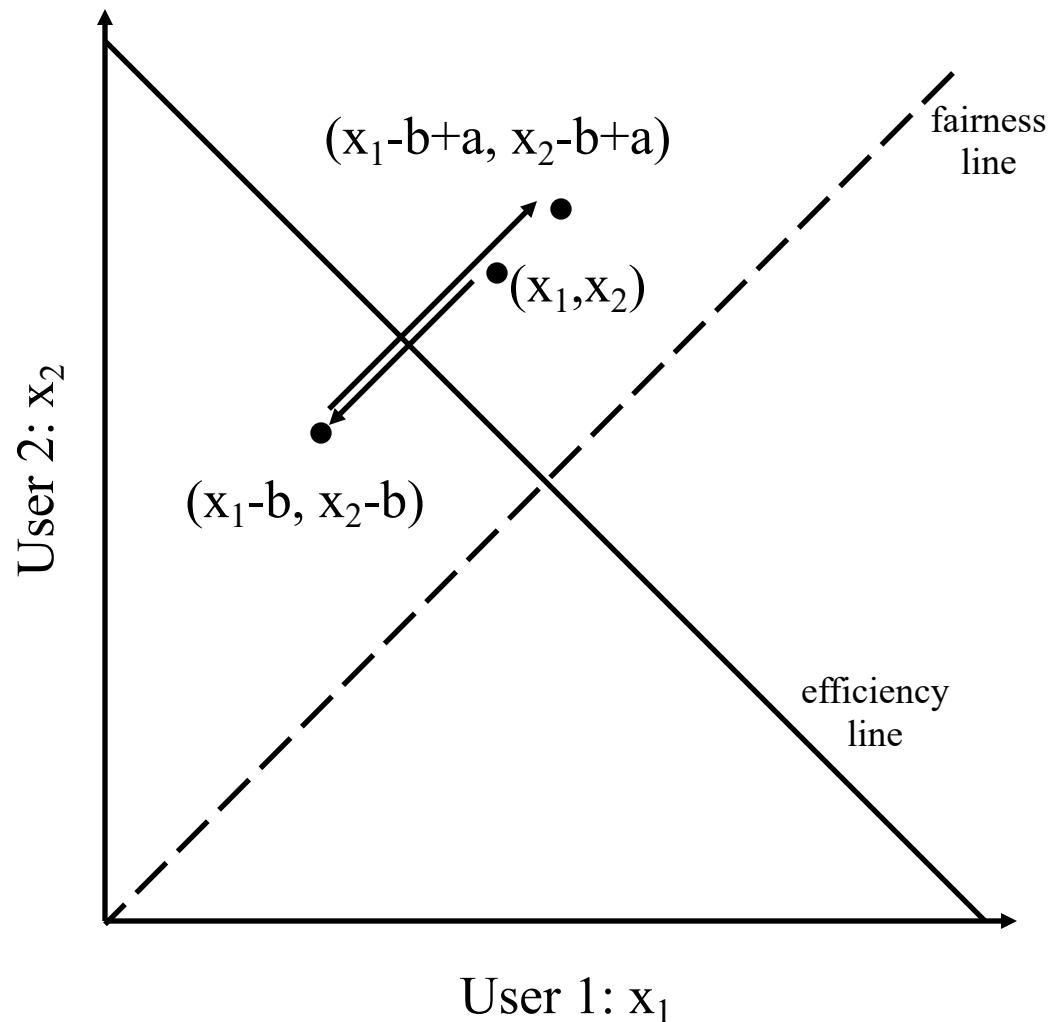
- AIAD: gentle increase, gentle decrease
- AIMD: gentle increase, rapid decrease
- MIAD: rapid increase, gentle decrease
- MIMD: rapid increase, rapid decrease
- **And now apply our simple model!**

AIAD Dynamics

- Consider: Increase: +1 Decrease: -2
 - Start at $X_1 = 1$, $X_2 = 3$, with $C = 5$
 - First iteration: no congestion
 - $X_1 \rightarrow 2$, $X_2 \rightarrow 4$
 - Second iteration: congestion
 - $X_1 \rightarrow 0$, $X_2 \rightarrow 2$
 - Third iteration: no congestion
 - $X_1 \rightarrow 1$, $X_2 \rightarrow 3$
 - ...
- Back where we started!
→ Gap between X_1 and X_2
didn't change at all

AIAD

- Increase: $x + a$
- Decrease: $x - b$
- Does not converge to fairness

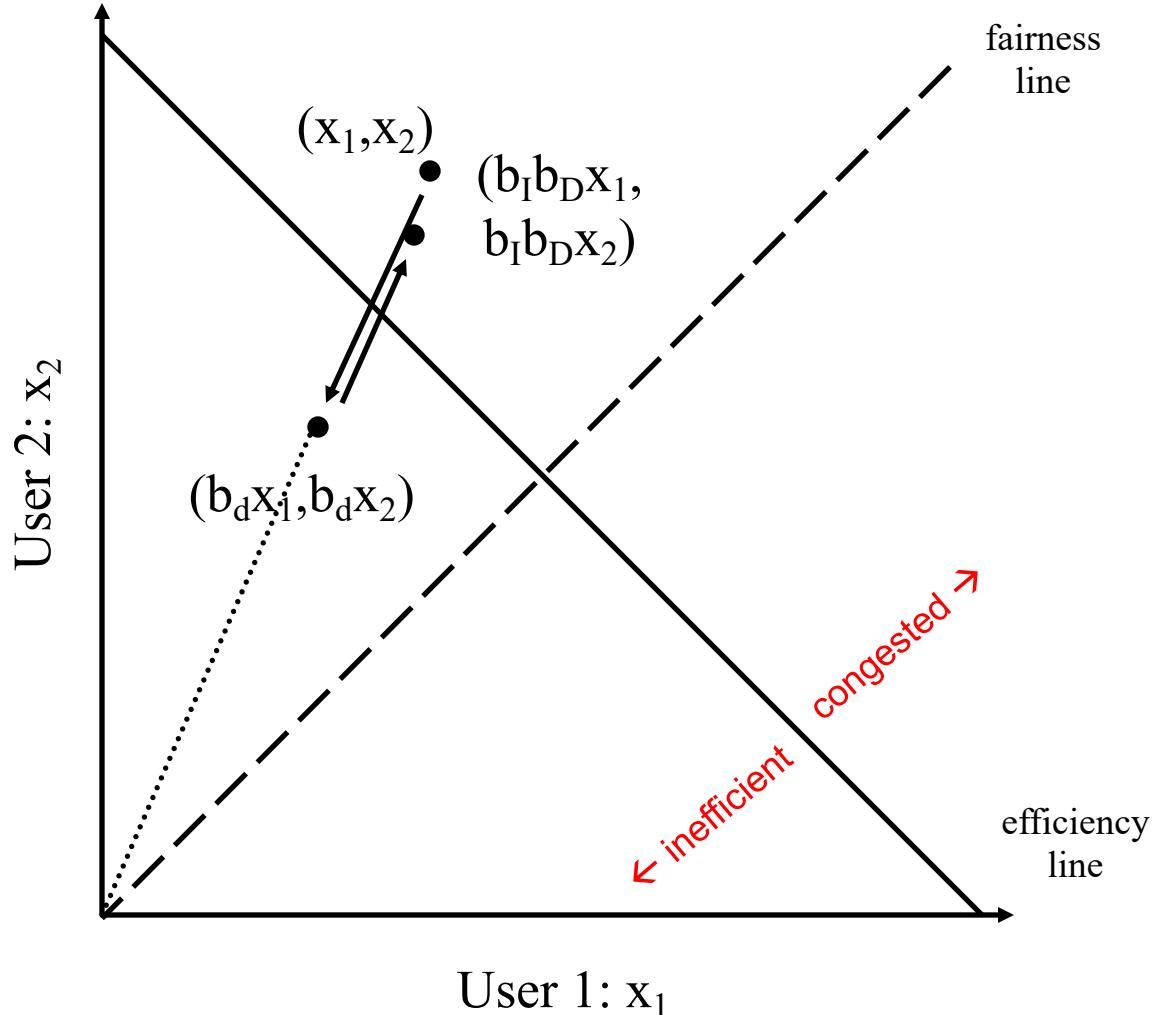


MIMD Dynamics

- Consider: Increase: $\times 2$ Decrease: $\div 4$
- Start at $X_1 = \frac{1}{2}$, $X_2 = 1$, with $C = 5$
- First iteration: no congestion
 - $X_1 \rightarrow 1$, $X_2 \rightarrow 2$
- Second iteration: no congestion
 - $X_1 \rightarrow 2$, $X_2 \rightarrow 4$
- Third iteration: congestion
 - $X_1 \rightarrow \frac{1}{2}$, $X_2 \rightarrow 1$
- ... **Again, no improvement in fairness**

MIMD

- Increase: $x \times b_I$
- Decrease: $x \times b_D$
- Does not converge to fairness



MIAD Dynamics

- Consider: Increase: $\times 2$ Decrease: -1
- Start at $X_1 = 1$, $X_2 = 3$, with $C = 5$
- First iteration: no congestion; $X_1 \rightarrow 2$, $X_2 \rightarrow 6$
- Second iteration: congestion; $X_1 \rightarrow 1$, $X_2 \rightarrow 5$
- Third iteration: congestion; $X_1 \rightarrow 0$, $X_2 \rightarrow 4$
- Fourth iteration: no congestion; $X_1 \rightarrow 0$, $X_2 \rightarrow 8$

X1 pegged at 0; MIAD is maximally unfair!

AIMD Dynamics

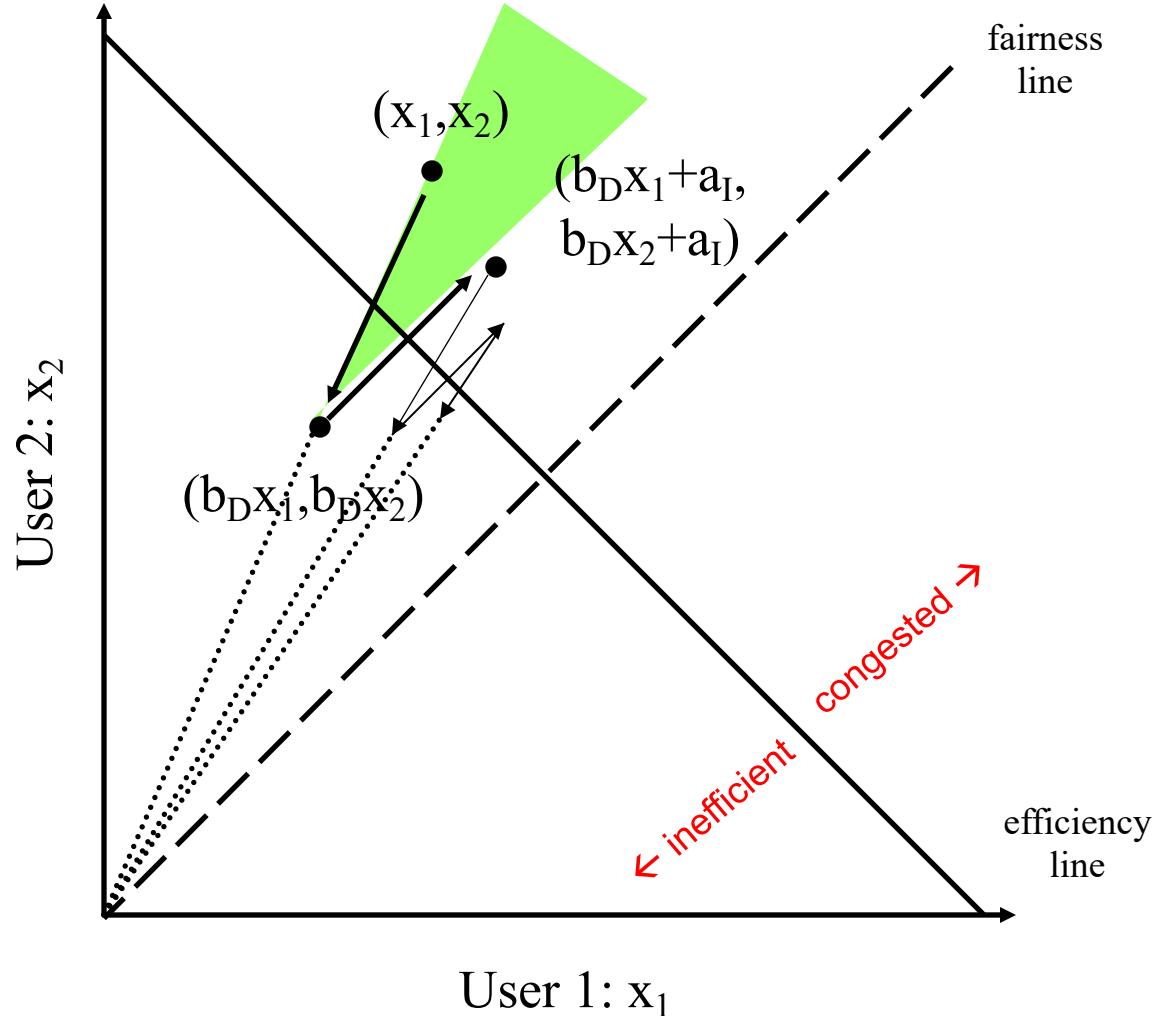
- Consider: Increase: +1 Decrease: $\div 2$
- Start at $X_1 = 1$, $X_2 = 2$, with $C = 5$ Diff = 1
- First iteration: no congestion: $X_1 \rightarrow 2$, $X_2 \rightarrow 3$ Diff = 1
- Second: no congestion: $X_1 \rightarrow 3$, $X_2 \rightarrow 4$ Diff = 1
- Third: congestion: $X_1 \rightarrow 1.5$, $X_2 \rightarrow 2$ Diff = 0.5
- Fourth: no congestion: $X_1 \rightarrow 2.5$, $X_2 \rightarrow 3$ Diff = 0.5
- Fifth: congestion: $X_1 \rightarrow 1.25$, $X_2 \rightarrow 1.5$ Diff = 0.25
- Sixth: no congestion: $X_1 \rightarrow 2.25$, $X_2 \rightarrow 2.5$ Diff = 0.25
- Seventh: no congestion: $X_1 \rightarrow 3.25$, $X_2 \rightarrow 3.5$ Diff = 0.25
- Eighth: congestion: $X_1 \rightarrow 1.625$, $X_2 \rightarrow 1.75$ Diff = 0.125
- Ninth: no congestion: $X_1 \rightarrow 2.625$, $X_2 \rightarrow 2.75$ Diff = 0.125

AIMD

- Difference between X1 and X2 decreasing!
 - Difference stays constant when increasing
 - Halves every time there is a decrease

AIMD

- Increase: $x + a_I$
- Decrease: $x * b_D$
- Converges to fairness



Answer to Why AIMD?

- AIMD embodies gentle increase, rapid decrease
- AIMD only choice that drives us towards “fairness”
- Out of the four options
 - AIAD, MIMD: retain unfairness
 - MIAD: maximally unfair
 - AIMD: fair and appropriate gentle/rapid actions

Any Questions?

Sketch of a solution

Each source independently runs the following:

- **Pick initial rate R**
- **Try sending at a rate R for some time period**
 - **Did I experience congestion in this time period?**
 - **If yes, reduce R**
 - **If no, increase R**
 - **Repeat**

Sketch of TCP's solution

Each source independently runs the following:

- Pick initial rate R
- Try sending at a rate R for some time period
 - Did I experience congestion in this time period?
 - If yes, reduce R
 - If no, increase R
 - Repeat

Sketch of TCP's solution

Each source independently runs the following:

- **Slow-start to find initial rate**
- Try sending at a rate R for some time period
 - Did I experience congestion in this time period?
 - If yes, reduce R
 - If no, increase R
 - Repeat

Sketch of TCP's solution

Each source independently runs the following:

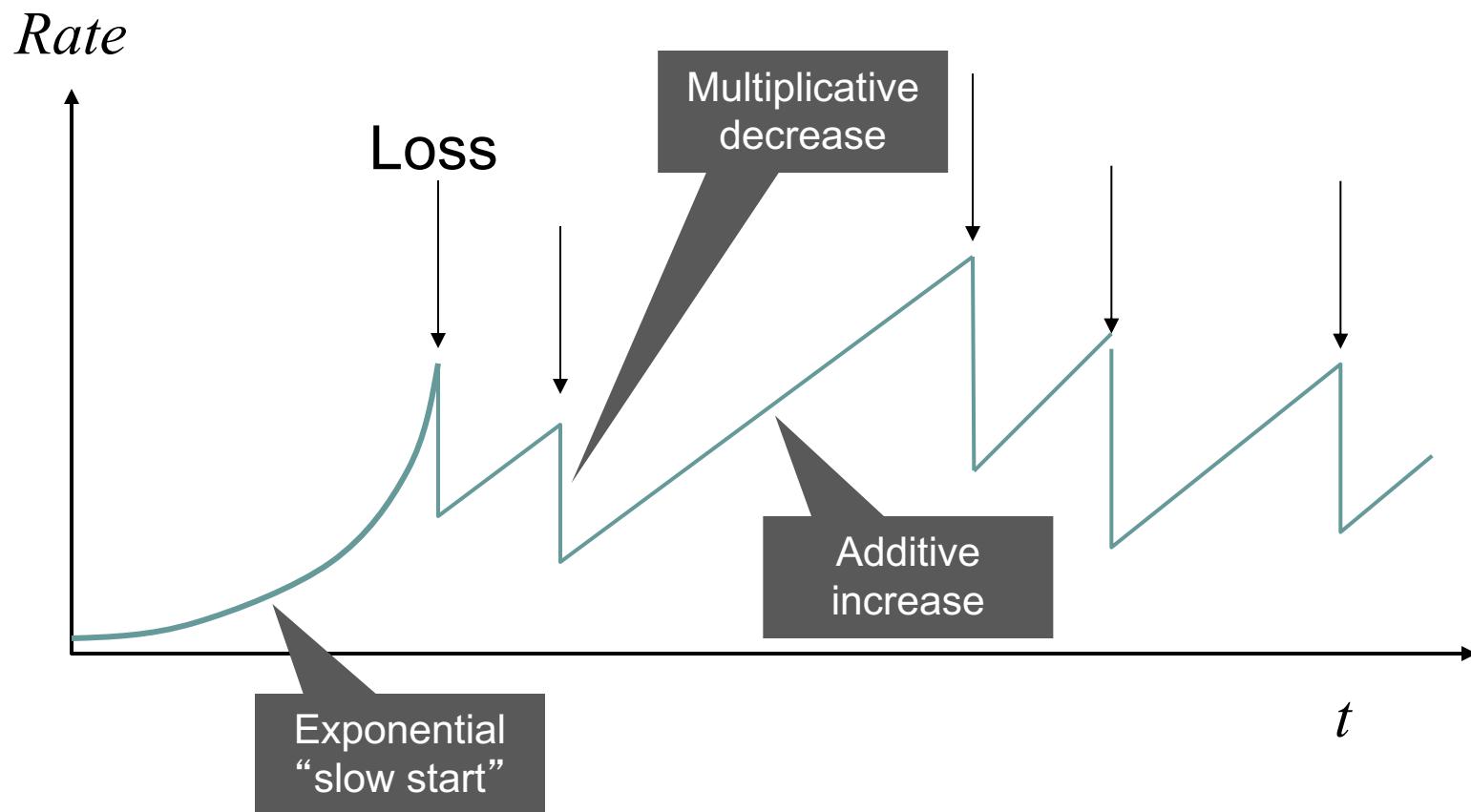
- **Slow-start to find initial rate**
- Try sending at a rate R for some time period
 - Did I experience ~~congestion~~ **loss** in this time period?
 - If yes, reduce R
 - If no, increase R
 - Repeat

Sketch of TCP's solution

Each source independently runs the following:

- **Slow-start to find initial rate**
- Try sending at a rate R for some time period
 - Did I experience ~~congestion~~ loss in this time period?
 - If yes, reduce R multiplicatively ($2x$)
 - If no, increase R additively (+1)
 - Repeat

Leads to the TCP “Sawtooth”



Next time: details of TCP CC

- Our overall approach with a few key differences
 - Based on adjusting window size on timescale of RTT
 - Different reactions for timeouts (severe loss) vs. duplicate ACKs (isolated loss)
 - Slow-start used on timeout as well as at beginning
 - Optimization for the case of isolated loss

TCP Congestion Control

CS 168

<http://cs168.io>

Sylvia Ratnasamy



Last time:

- We narrowed our exploration of the design space to a CC solution that is based on:
 - Implemented only by end-hosts
 - Dynamic rate adjustment
 - Uses loss to detect congestion
- Today: TCP CC
 - An example of the above design



Plan

- Review TCP's window-based operation
- Extending the above for CC



Review:

- Sender maintains a window of packets in flight
- Window size W is picked to balance three goals
 - Take advantage of network capacity (“fill the pipe”)
 - Avoid overloading the receiver (flow control)
 - Avoid overloading links (congestion control)



Review:

- Sender maintains a window of packets in flight
- Window size W is picked to balance three goals
 - Take advantage of network capacity (“fill the pipe”)
 - Avoid overloading the receiver (flow control)
 - Avoid overloading links (congestion control)
- Flow control: sender maintains an **advertised window**; also called a **receiver window (RWND)**
- CC: sender maintains a **congestion window (CWND)**



All These Windows...

- Congestion Window: **CWND**
 - How many bytes can be sent without overloading links
 - Computed by the sender using CC algorithm
- Flow control window: **RWND**
 - How many bytes can be sent without overflowing the receiver's buffers
 - Implemented by having the receiver tell the sender
- **Sender-side window = $\min\{CWND, RWND\}$**
 - Assume for this lecture that **RWND > CWND**

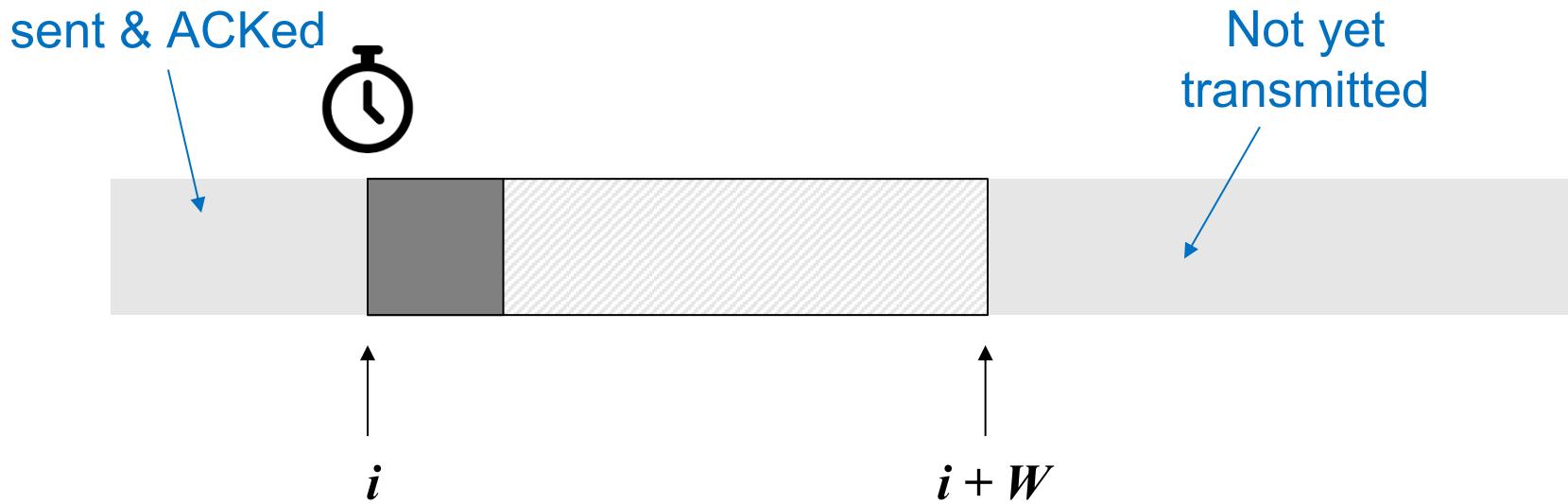


Note

- **Recall: TCP operates on bytestreams**
- **Hence, real implementations maintain CWND in bytes**
- **This lecture will talk about CWND in units of MSS**
 - MSS: Maximum Segment Size, the max number of bytes of data that one TCP packet can carry in its payload
 - This is only for pedagogical purposes



Review:

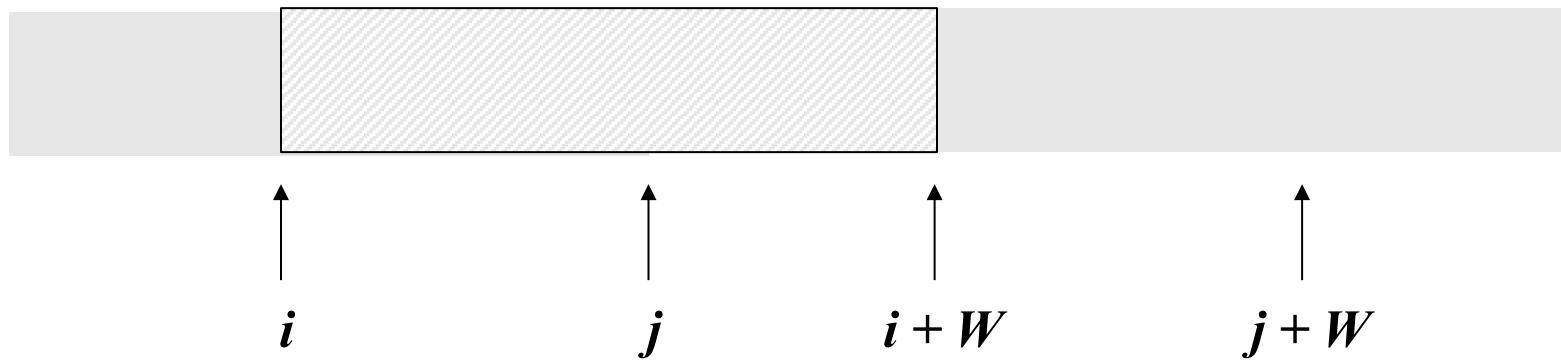


Sender maintains a **sliding window** of **W contiguous bytes**

Sender maintains a single timer, for the LHS of window

On timeout, sender retransmits the packet starting at i

Review:



Receiver sends cumulative ACKs; sender counts #dupACKs

Fast Retransmit: Sender retransmits when #dupACKs = 3

Sender slides window on receiving an ACK for new data ($j > i$)

Extending TCP with CC

- Add a congestion window parameter (CWND)
- Adapt CWND based on current congestion level
- How do we adapt CWND?
 - Last lecture: how sender adapts its transmission *rate*
- In TCP, sender's rate is simply CWND/RTT
 - (Since we're assuming RWND > CWND)
- Adapting CWND every RTT → adapting sender's *rate*



Recall: how we adapt rate

- Detecting congestion
 - **Loss-based**
- Discovering an initial rate
 - **Slow start**
- Adapting rate to congestion (or lack thereof)
 - **AIMD**

What follows is all about how TCP implements the above

Theme: CWND updates driven by ACK arrivals (“ACK clocking”)

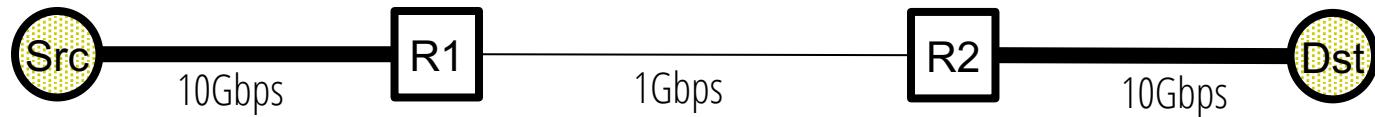


ACK Clocking

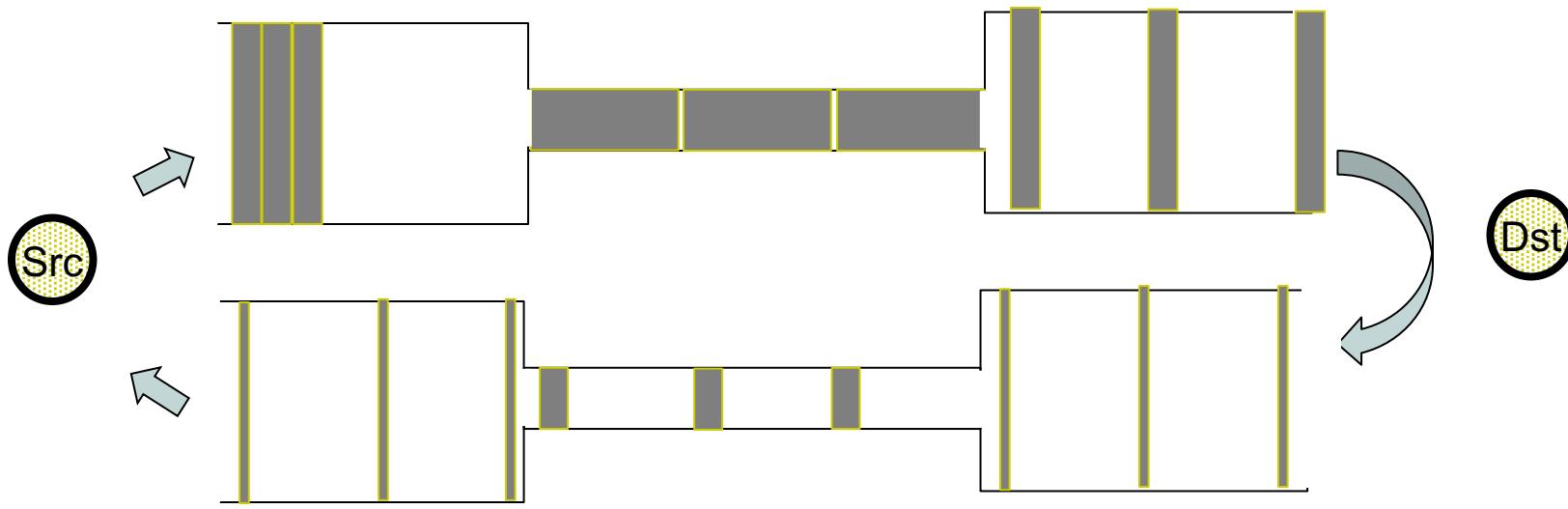
- A new ACK advances the sliding window and lets a new data segment enter the network
 - I.e., ACKs “clock” data segments
- What’s the benefit of ACK clocking?



ACK Clocking



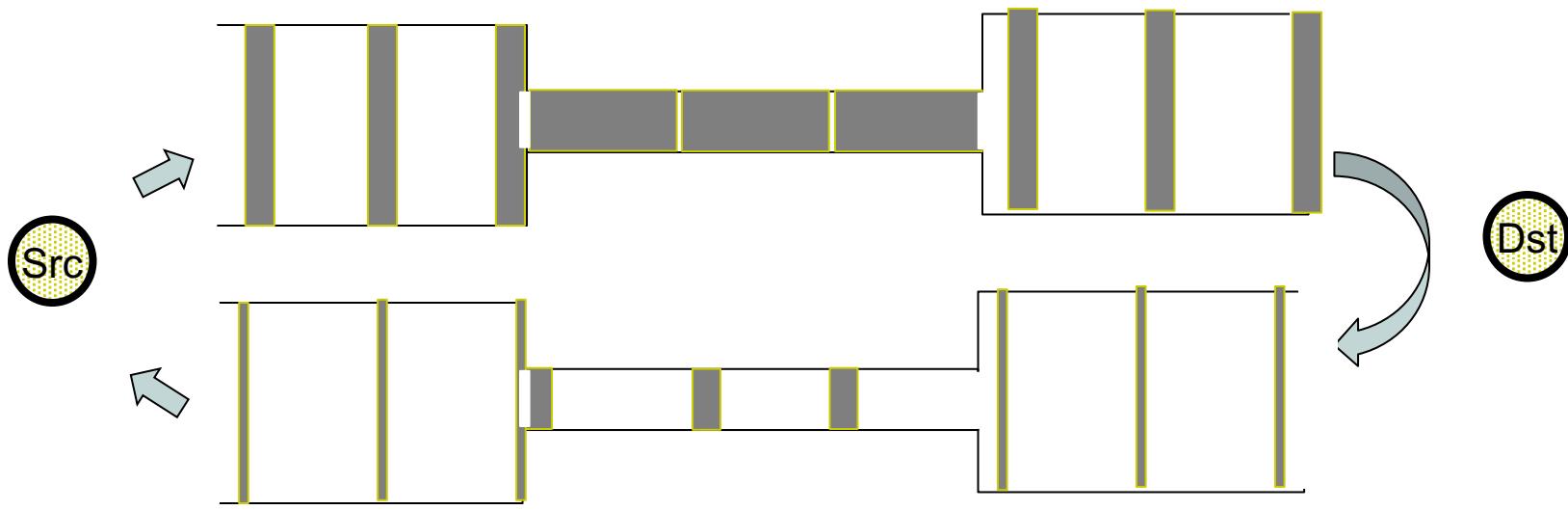
ACK Clocking



Consider: source sends a burst of packets
Packets are queued and “spread out” at slow link
ACKs maintain the spread on the return path



ACK Clocking



Sender clocks new packets with the spread

Now sending without queuing at the bottleneck link!



Recall: how we adapt rate

- Detecting congestion
 - **Loss-based**
- Discovering an initial rate
 - **Slow start**
- Adapting rate to congestion (or lack thereof)
 - **AIMD**

What follows is all about how TCP implements the above

Theme: CWND updates driven by ACK arrivals (“ACK clocking”)



How TCP Detects Loss

- **3 duplicate ACKs:** typically indicates isolated loss
- **Timeout:** typically indicates loss of several packets



How TCP Implements Slow Start

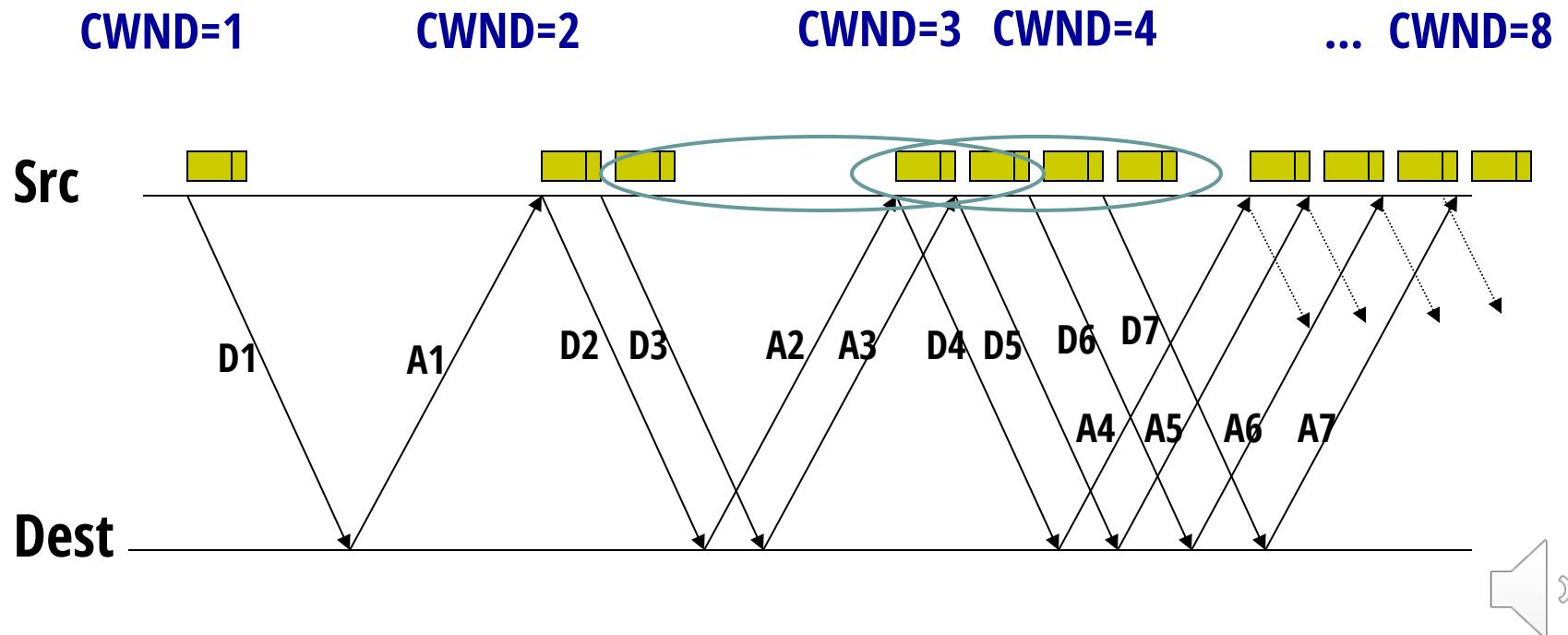
- Sender starts at a slow rate; increases rate **exponentially** until first loss
- In TCP: start with a small CWND = 1 (MSS)
 - So, initial sending rate is MSS/RTT
- Then double CWND every RTT until first loss
- Implemented as: **On each ACK: CWND += 1 (MSS)**



Slow Start in Action

Goal: Double CWND every round-trip time

Simple implementation: On each ACK, CWND += 1 (MSS)



How TCP Implements Slow Start (contd.)

- Double CWND every RTT until first loss
- Introduce a “slow start threshold” parameter
 - **SSTHRESH**, used to remember last “safe” rate
- On first loss: **SSTHRESH = CWND/2**



Recall: how we adapt rate

- Detecting congestion
 - Loss-based
- Discovering an initial rate
 - Slow start
- Adapting rate to congestion (or lack thereof)
 - AIMD



AIMD in TCP

- Additive increase:
 - No loss → increase CWND by **1 MSS every RTT**
- Multiplicative decrease
 - Loss detected by 3 dupACKs → divide CWND in **half**
- What about timeouts? Will **exit AIMD** (coming up)



Implementing Additive Increase

- Implementation works by adding a fraction of an MSS every time we receive an ACK
- On receiving an ACK (for new data)
 - $CWND \rightarrow CWND + \frac{1}{CWND}$
 - $CWND \rightarrow CWND + MSS \times \frac{MSS}{CWND}$ *if counting CWND in bytes*
- NOTE: after full window, CWND increases by 1 MSS
 - Thus, CWND increases by 1 MSS per RTT



Implementing Multiplicative Decrease

- On receiving 3rd dupACK:

- $CWND \rightarrow \frac{CWND}{2}$



On Timeout

- Rationale: lost multiple packets in a window
 - Current CWND may be way off
 - Hence, need to rediscover a good rate from scratch
 - Design decision that errs on the side of caution
- Hence, on timeout:
 - Retransmit first missing packet (as usual)
 - Set SSTHRESH $\leftarrow \frac{CWND}{2}$
 - Set CWND $\leftarrow 1$ MSS & enter **Slow Start** mode



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Determined by **SSTHRESH**
- When $CWND > SSTHRESH$, sender switches from slow-start to AIMD's additive increase



Summary of Decrease

- Cut CWND in **half** on loss detected by dupACKs
- Cut CWND **all the way to 1 (MSS)** on timeout
- Never drop CWND below 1 (MSS)



Summary of Increase

- When in Slow-Start phase
 - Increase CWND by 1 MSS for each new ack
- When in AIMD phase
 - Increase by 1 (MSS) for each window's worth of acked data



TCP Congestion Control Details

In what follows refer to CWND in units of MSS



Implementation

- **State at sender**
 - CWND (initialized to a 1 MSS)
 - STHRESH (initialized to a large constant)
 - dupACKcount (initialized to zero, as before)
 - Timer (as before)
- **Events at sender**
 - ACK (for new data)
 - dupACK (duplicate ACK for old data)
 - Timeout
- What about receiver?
 - Just send ACKs like before



Event: ACK (new data)

- If in slow start
 - $CWND += 1 \text{ (MSS)}$
- $CWND$ packets per RTT
 - Hence after one RTT with no drops:
 $CWND = 2 \times CWND$

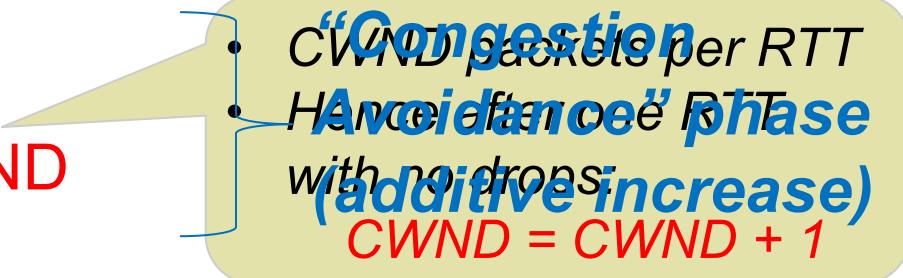


Event: ACK (new data)

- If in slow start
 - CWND += 1 (MSS)
- Else
 - CWND = CWND + 1/CWND
- Plus the usual ...
 - Reset timer, dupACKcount
 - Send new data packets (if CWND allows)



Slow start phase



Event: TimeOut

- On Timeout
 - $SSTHRESH \leftarrow CWND/2$
 - $CWND \leftarrow 1$
 - And retransmit packet (as always)



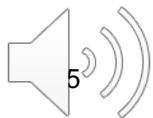
Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
 - SSTHRESH = CWND/2
 - CWND = CWND/2 (but never less than 1)
 - And retransmit packet (as always)

Remain in AIMD
after fast retransmission...

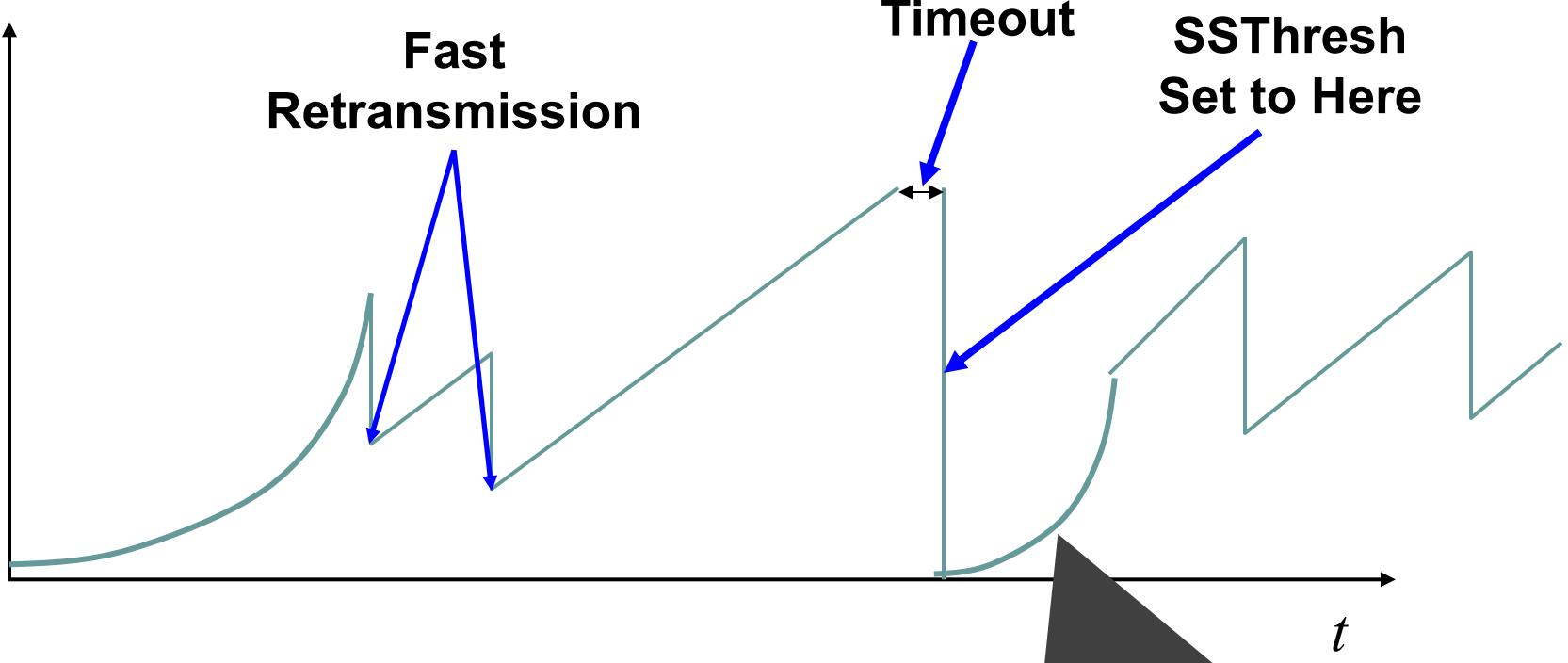


Any Questions?



Time Diagram

Window



Slow start in operation until
CWND crosses STHRESH



One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss
- This last feature is an optimization to improve performance
 - Bit of a hack, but effective



Example

- Again: counting packets, not bytes
 - If you want example in bytes, assume MSS=1000 and add three zeros to all sequence numbers
- Consider a TCP connection with:
 - CWND=10 packets
 - Last ACK was for packet # 101
 - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
 - Packet 101 is dropped
 - What ACKs do they generate and how does the sender respond?



Timeline (at sender)

In flight: ~~101, 102, 103, 104, 105, 106, 107, 108, 109, 110~~ 101

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 (no xmit)
- ACK 101 (due to 106) cwnd=5 (no xmit)
- ACK 101 (due to 107) cwnd=5 (no xmit)
- ACK 101 (due to 108) cwnd=5 (no xmit)
- ACK 101 (due to 109) cwnd=5 (no xmit)
- ACK 101 (due to 110) cwnd=5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK “clocking” stalls for another RTT

Note that you do not restart dupACK counter on same packet!



Two Questions

- Do you understand the problem?
 - Have to wait a long time before sending again
 - When you finally send, you have to send full window
- How would you fix it?



Solution: Fast Recovery

Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight

- If $\text{dupACKcount} = 3$
 - $\text{SSTHRESH} = \text{CWND}/2$
 - $\text{CWND} = \text{SSTHRESH} + 3$
- While in fast recovery
 - $\text{CWND} = \text{CWND} + 1 \text{ (MSS)}$ for each additional duplicate ACK
 - This allows source to send an additional packet...
 - ...to compensate for the packet that arrived (generating dupACK)
- Exit fast recovery after receiving new ACK
 - set $\text{CWND} = \text{SSTHRESH}$



Timeline (at sender)

In flight: ~~101, 102, 103, 104, 105, 106, 107, 108, 109, 110~~ **101 111, 112, ...**

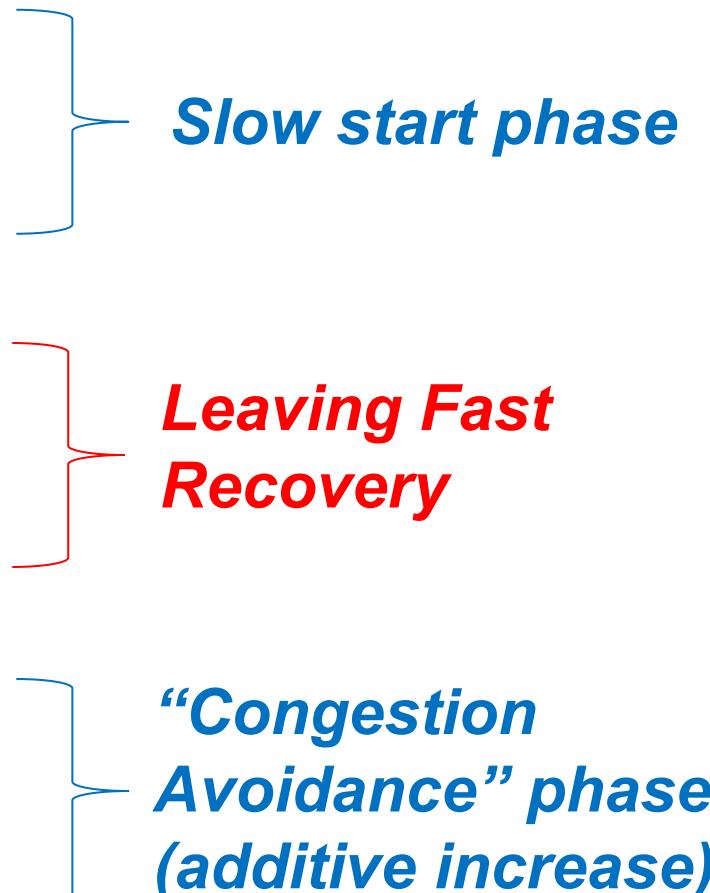
- ACK 101 (due to 102) cwnd=10 dupACK#1
- ACK 101 (due to 103) cwnd=10 dupACK#2
- ACK 101 (due to 104) cwnd=10 dupACK#3
- **REXMIT 101 ssthresh=5 cwnd= 8 (5+3)**
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (**xmit 111**)
- ACK 101 (due to 108) cwnd=12 (**xmit 112**)
- ACK 101 (due to 109) cwnd=13 (**xmit 113**)
- ACK 101 (due to 110) cwnd=14 (**xmit 114**)
- **ACK 111 (due to 101) cwnd = 5 (xmit 115) ↙ exiting fast recovery**
- **Packets 111-114 already in flight (and now sending 115)**
- ACK 112 (due to 111) cwnd = 5 + 1/5 ↙ back in congestion avoidance



Updated Event-Actions



Event: ACK (new data)

- If in slow start
 - CWND += 1 (MSS)
 - If in fast recovery
 - CWND = STHRESH
 - Else
 - CWND = CWND + 1/CWND
 - Plus the usual...
- 
- Slow start phase***
- Leaving Fast Recovery***
- “Congestion Avoidance” phase (additive increase)***



Event: dupACK

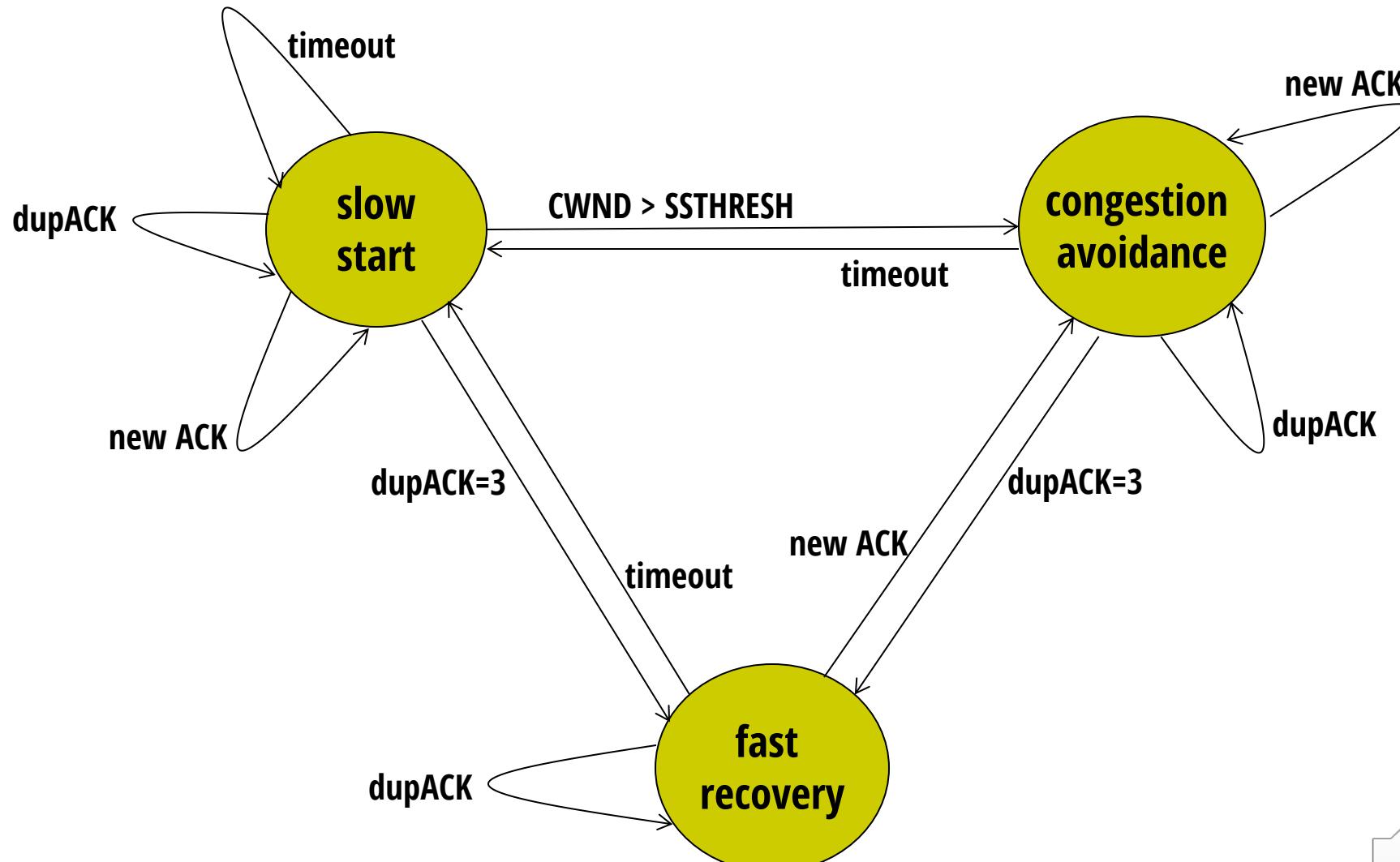
- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
 - ssthresh = CWND/2
 - CWND = CWND/2 +3
 - And retransmit packet
- If dupACKcount > 3 /* fast recovery */
 - CWND = CWND + 1 (MSS)



Next: TCP State Machine



TCP State Machine



Many variants

- TCP-Tahoe
 - CWND =1 on triple dupACK
- TCP-Reno
 - CWND =1 on timeout
 - CWND = CWND/2 on triple dupack
- TCP-newReno
 - TCP-Reno + improved fast recovery
- TCP-SACK
 - incorporates “selective acknowledgements”
 - ACKs describe byte ranges received

Our default assumption



Interoperability

- How can all these algorithms coexist? Don't we need a single, uniform standard?
- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?
- What happens if I'm using Tahoe and you are using SACK?



Next Lecture

- Modeling TCP
- Advanced congestion control techniques



Project 2: Transport

- You will implement the core parts of a TCP socket (Discussion#3)
- Use a network simulator (by Murphy McCauley & others at NetSys) to test, validate, and interact with your socket implementation
- The project is split and scored by (9) stages
- The goal is to guide you through the basic procedures of the TCP protocol, e.g., three-way handshake, reassembly of out-of-order packets, packet retransmission, and passive/active close
- Due: 11:59pm, Nov. 11th. Logistics & OH will be announced on Ed

Announcement#1: Lectures 18-21

- Will release lecture recordings by Murphy
- Topics: DNS, HTTP, Ethernet, discovery protocols
- No in-person lectures: 10/27, 11/1, 11/3
- Flipped lecture on 11/08
- Reminder and details will be posted on Ed

Congestion Control: Advanced Topics

CS 168

<http://cs168.io>

Sylvia Ratnasamy

Last Time

- The gory details of TCP CC

Today

- Modeling TCP
- Critiquing TCP
- Router-assisted CC
- We'll cover a broad range of design ideas
- Focus on the *why* and key insight behind the *how*
- Don't worry about the details

TCP Throughput Equation

TCP Throughput

- Given a path, what TCP throughput can we expect?
- We'll derive a simple model that expresses TCP throughput in terms of path properties:
 - RTT
 - Loss rate, p

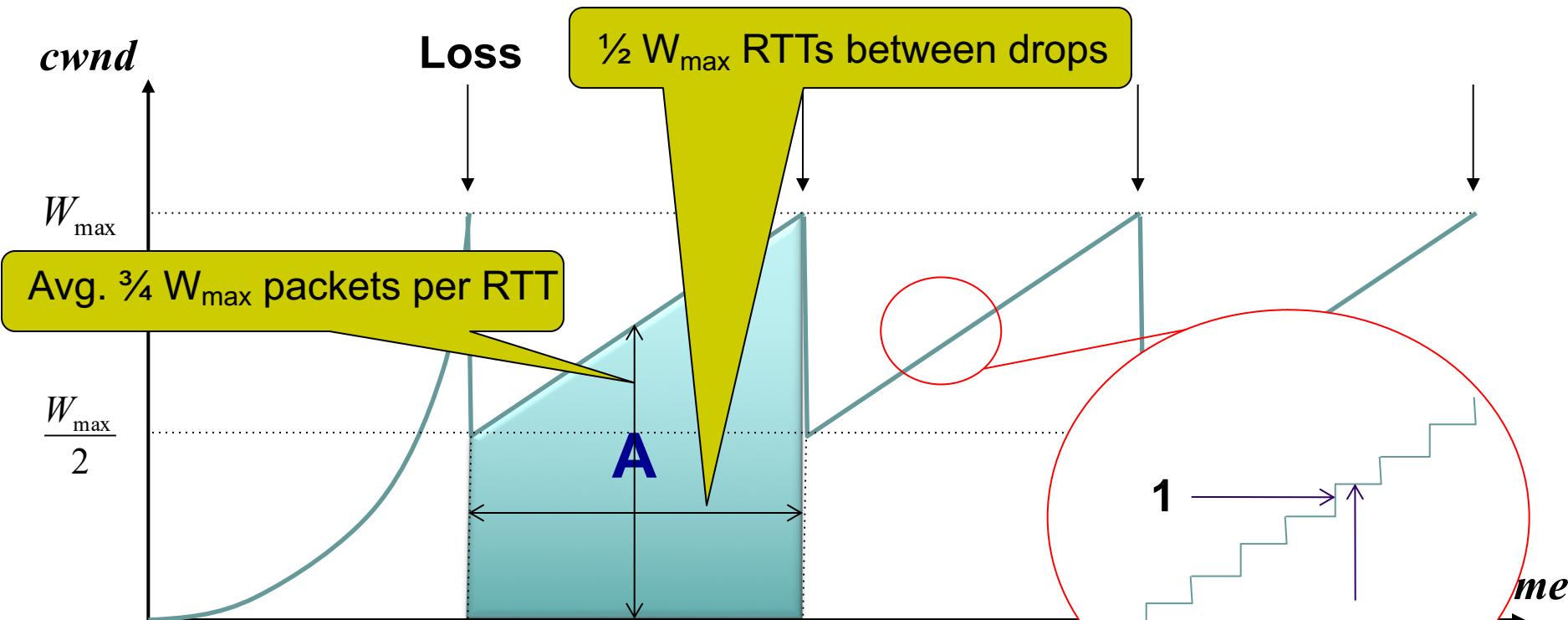
A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)
- Hence, evolution of window size:
 - $\frac{1}{2}W_{max}$ (after detecting loss)
 - $\frac{1}{2}W_{max} + 1$ (one RTT later)
 - $\frac{1}{2}W_{max} + 2$ (two RTTs later)
 - $\frac{1}{2}W_{max} + 3$ (three RTTs later)
 - ...
 - W_{max} [drop]
 - $\frac{1}{2}W_{max}$
 - $\frac{1}{2}W_{max} + 1$

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)
- Hence, evolution of window size:
 - $\frac{1}{2}W_{max}, \frac{1}{2}W_{max}+1, \frac{1}{2}W_{max}+2, \dots, W_{max}$ [drop], $\frac{1}{2}W_{max}, \frac{1}{2}W_{max}+1, \dots$
 - Increase by 1 for $\frac{1}{2}W_{max}$ RTTs, then drop, then repeat
- Average window size per RTT = $\frac{3}{4}W_{max}$
- Average throughput = $\frac{3}{4}W_{max} \times \frac{MSS}{RTT}$
- Remaining step: express W_{max} in terms of loss rate p

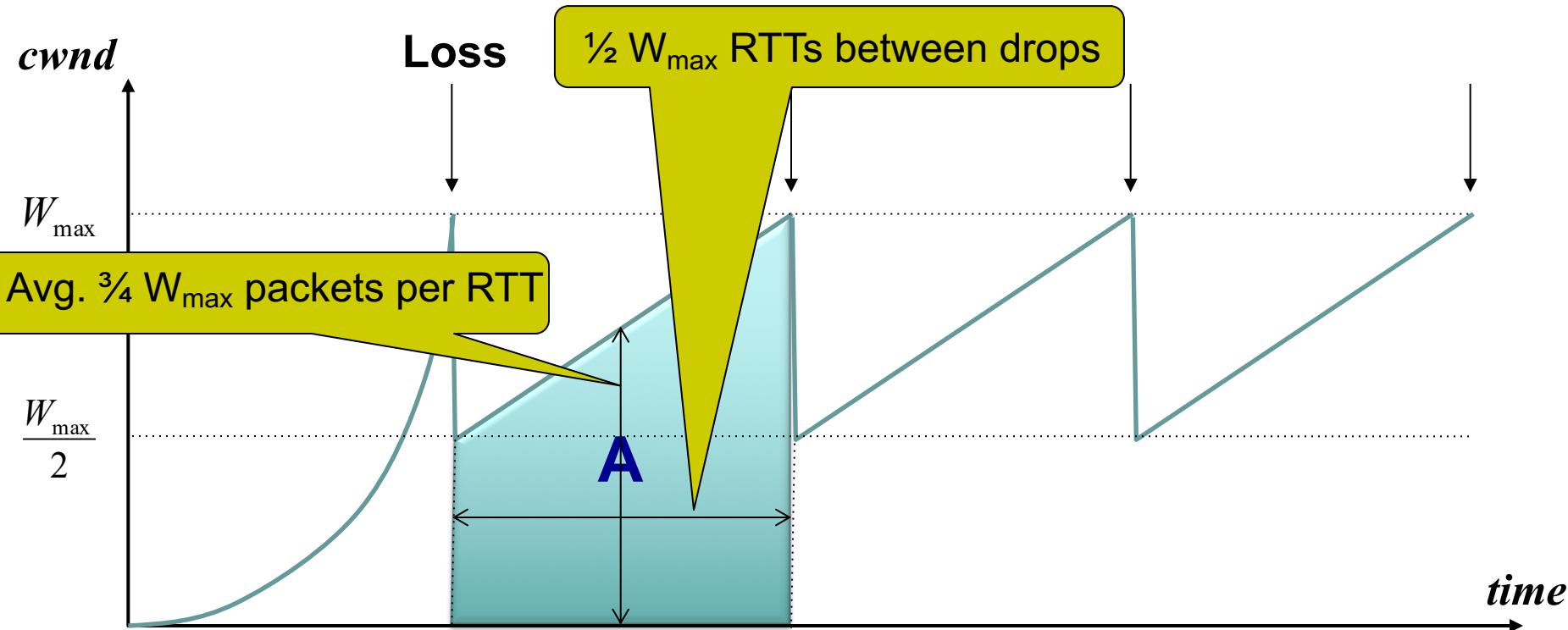
A Simple Model for TCP Throughput



$$\text{Packet drop rate, } p = \frac{1}{A}$$

On average, one of all packets in shaded region is lost
(i.e., loss rate is $1/A$, where A is #packets in shaded region)

A Simple Model for TCP Throughput



$$\text{Packet drop rate, } p = \frac{1}{A} \quad A = \frac{3}{8} W_{max}^2 \quad \rightarrow W_{max} = \frac{2\sqrt{2}}{\sqrt{3p}}$$

$$\text{Average Throughput} = \frac{\frac{3}{4} W_{max} \times \text{MSS}}{\text{RTT}} = \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT} \sqrt{p}}$$

TCP Throughput

- Given a path, what TCP throughput can we expect?
- TCP throughput is proportional to $\frac{1}{\text{RTT}}$ and $\frac{1}{\sqrt{p}}$
 - RTT is path round-trip time and p is the packet loss rate
- Model makes many simplifying assumptions
 - Ignores slow-start, assumes fixed RTT, isolated loss, etc.
- But leads to some insights (coming up)

Taking Stock: TCP CC

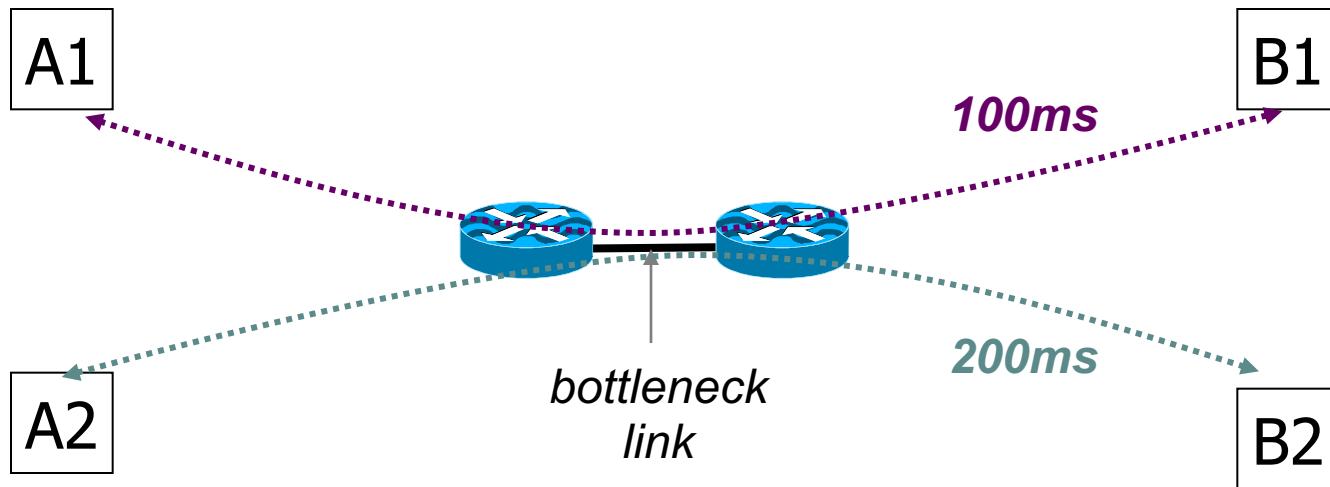
- (Sender) host based
- Loss based
- Adapts every RTT
- Starts out in slow start (start small, double every RTT)
- Adapts based on AIMD (gentle increase, rapid decrease)
- TCP throughput depends on path RTT and loss rate

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT} \sqrt{p}}$$

Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



Implications (2): High Speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT}\sqrt{p}}$$

- Assume **BW=100Gbps**, RTT = 100ms, MSS=1500B
- Value of p required to reach 100Gbps throughput: 2×10^{-12}
 - Requires dropping only one out of 50 billion packets!
 - Going ~ 16.6 hours between drops
- These are not practical numbers
- Problem: scaling a single flow to high throughput is **very** slow with additive increase

HighSpeed TCP [RFC 3649]

- Once past a threshold speed, increase CWND faster
 - Make the increase rule a function of CWND
- Other approaches?
 - Multiple simultaneous connections (workaround)
 - Router-assisted approaches (will see shortly)

Implications (3): *Rate-based CC* [RFC 5348]

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{\text{RTT}\sqrt{p}}$$

- TCP throughput is “choppy”
 - repeated swings between $W/2$ to W
- Some apps would prefer sending at a steady rate
 - e.g., streaming apps
- A solution: Equation-based Congestion Control
 - ditch TCP’s increase/decrease rules and just follow the equation
 - measure RTT and drop percentage p , and set rate accordingly
- Following the TCP equation ensures we’re “TCP friendly”
 - i.e., use no more than TCP does in similar setting

Other Limitations of TCP Congestion Control

(4) Loss not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
 - Throughput $\sim \frac{1}{\sqrt{p}}$ even for non-congestion losses!

(5) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (1): many flows never leave slow start!
 - Short flows never attain their fair share
 - In fact, short flows are likely to suffer unduly long transfer times
- Implication (2): too few packets to trigger dupACKs
 - Isolated loss may lead to timeouts
 - At typical timeout values of ~500ms, might severely impact flow completion time
- A partial fix: use a higher initial CWND [Google IW10]

(6) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Recall: loss follows delay (i.e., queue *must* fill up)
- Means that delays are large, for everyone
 - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B
- Problem exacerbated by the trend towards adding large amounts of memory on routers (a.k.a. “bufferbloat”)

(6) TCP fills up queues → long delays

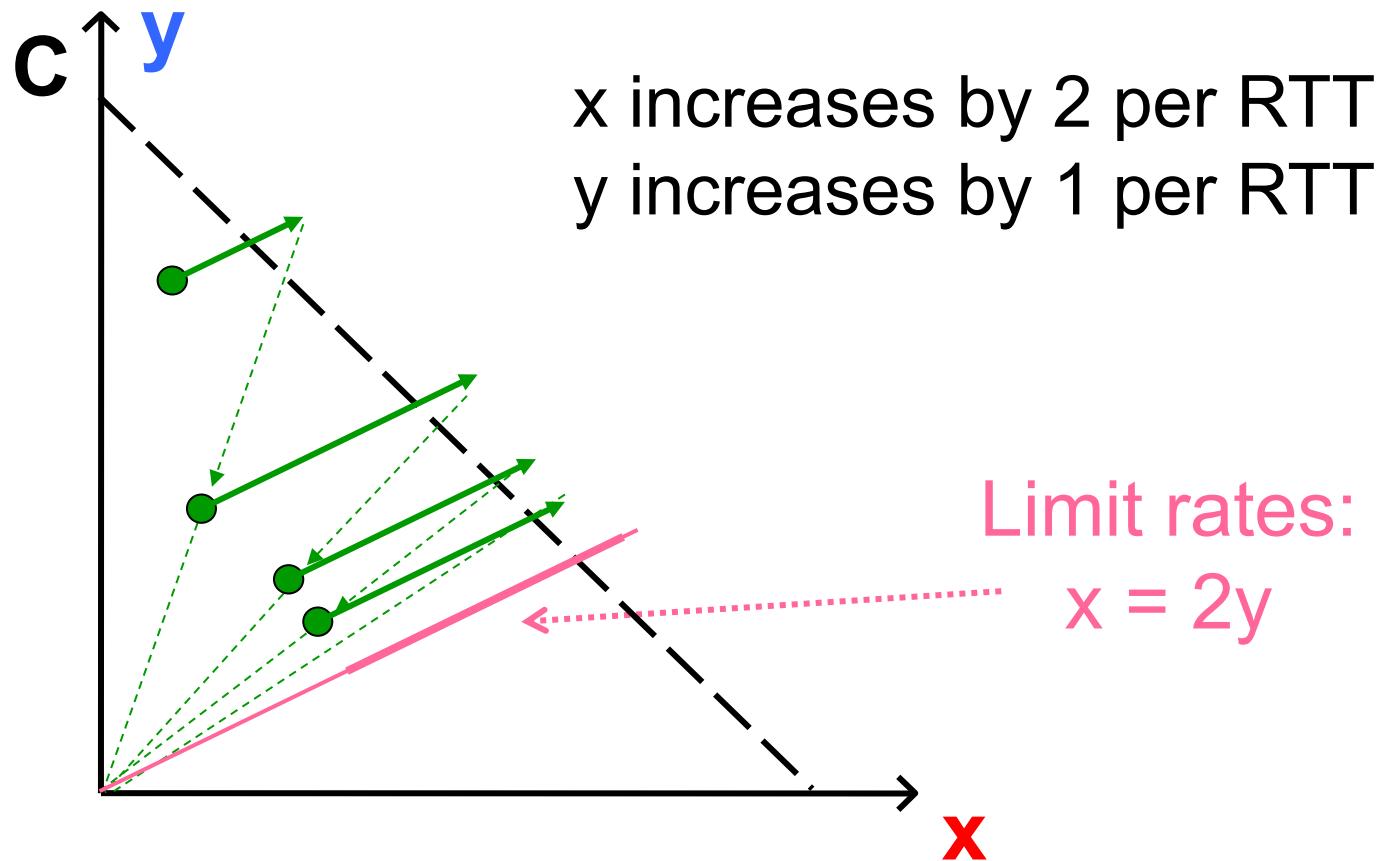
- Focus of Google's BBR algorithm¹
- Basic idea (simplified):
 - Sender learns its minimum RTT (~ propagation RTT)
 - Decreases its rate when the observed RTT exceeds the minimum RTT

¹ [BBR: Congestion-Based Congestion Control; Cardwell et al, ACM Queue 2016](#)

(7) Cheating

- Three easy ways to cheat
 - Increasing CWND faster than +1 MSS per RTT

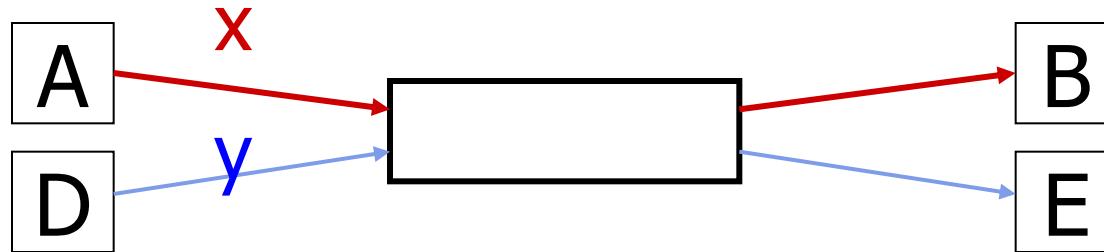
Increasing CWND Faster



(7) Cheating

- Three easy ways to cheat
 - Increasing CWND faster than +1 MSS per RTT
 - Opening many connections

Open Many Connections



Assume

- A starts 10 connections to B
- D starts 1 connection to E
- Each connection gets about the same throughput

Then A gets 10 times more throughput than D

(7) Cheating

- Three easy ways to cheat
 - Increasing CWND faster than +1 MSS per RTT
 - Opening many connections
 - Using large initial CWND

Why hasn't the Internet suffered another congestion collapse?

- Even “cheaters” do back off!
 - Leads to unfairness, not necessarily collapse
- Hard to say whether unfair behavior is common

MOTHERBOARD
TECH BY VICE

**Google's Network Congestion Algorithm
Isn't Fair, Researchers Say**

(8) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
 - CWND adjusted based on ACKs and timeouts
 - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
 - Consider changing from cumulative to selective ACKs
 - A failure of modularity, not layering
- Sometimes we want CC but not reliability
 - e.g., real-time applications
- Sometimes we want reliability but not CC (?)

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-Assisted Congestion Control

- Three ways routers can help
 - Enforce fairness
 - More precise rate adaptation
 - Detecting congestion

How can routers ensure each flow gets its “fair share”?

Fairness: General Approach

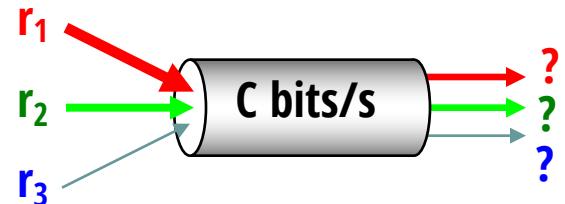
- Consider a single router's actions
- Router classifies incoming packets into “flows”
 - (For now) let's assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router picks a queue (i.e., flow) in a fair order; transmits packet from the front of the queue
- What does “fair” mean exactly?

Max-Min Fairness

- Total available bandwidth C
- Each flow i has bandwidth demand r_i
- What is a fair allocation a_i of bandwidth to each flow i ?
- Max-min bandwidth allocations are:

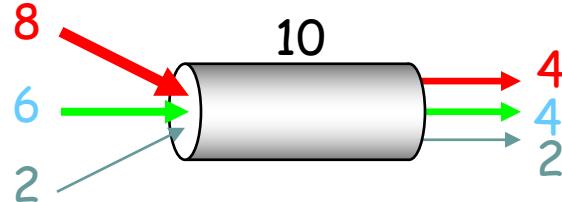
$$a_i = \min(f, r_i)$$

where f is the unique value such that $\text{Sum}(a_i) = C$



Example

- $C = 10; N = 3; r_1 = 8, r_2 = 6, r_3 = 2$
- $C/N = 10/3 = 3.33 \rightarrow$
 - But r_3 's need is only 2
 - Can service all of r_3
 - Allocate 2 to r_3 and remove it from accounting: $C = C - r_3 = 8; N = 2$
- $C/2 = 4 \rightarrow$
 - Can't service all of r_1 or r_2
 - So hold them to the remaining fair share: $f = 4$



$f = 4:$
 $\min(8, 4) = 4$
 $\min(6, 4) = 4$
 $\min(2, 4) = 2$

Max-Min Fairness

- Property:
 - If you don't get full demand, no one gets more than you
- This is what round-robin service gives if all packets are the same size

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)
- Cannot do this in practice!
- But we can approximate it
 - This is what “**fair queuing**” routers do

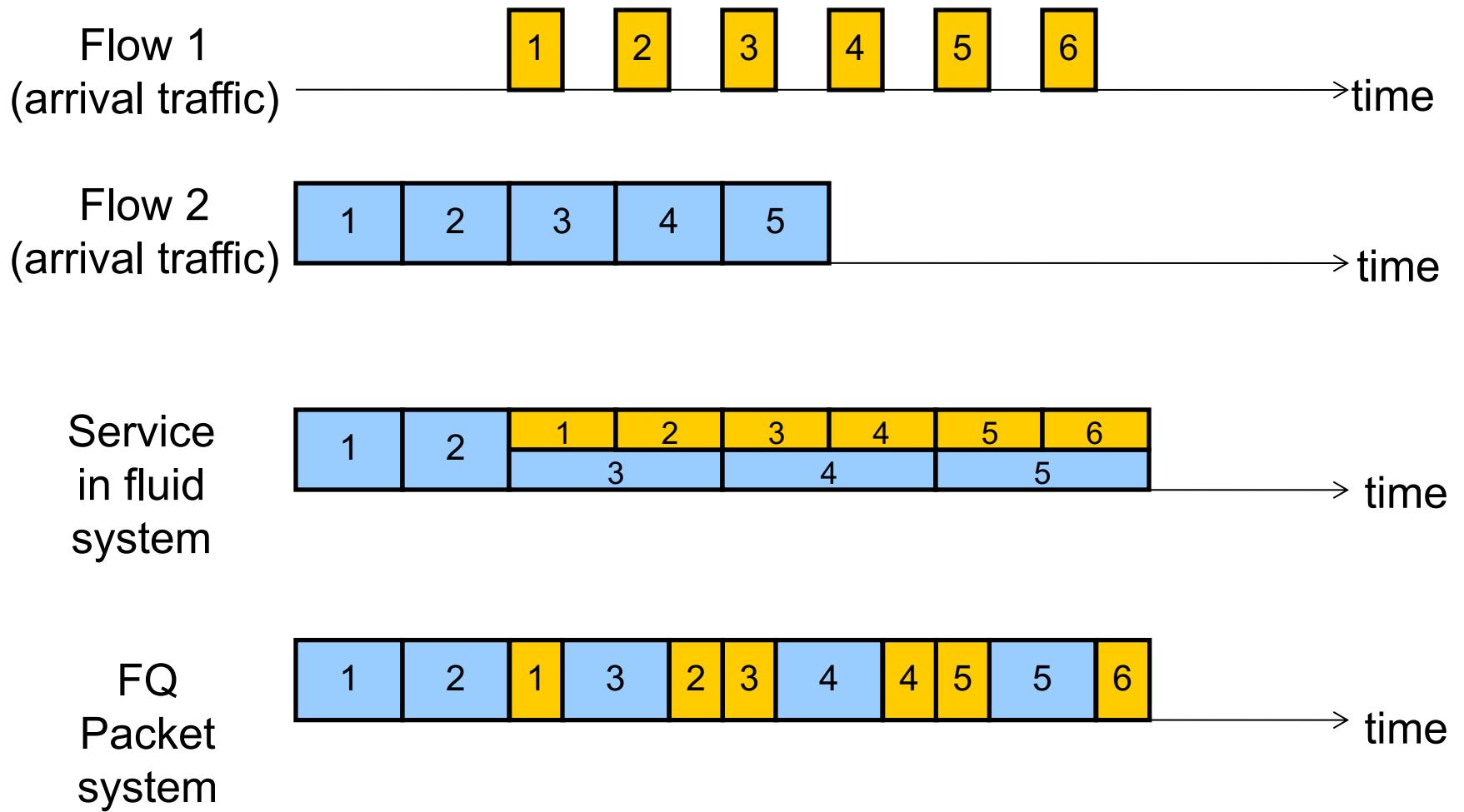
Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit (called “deadlines”)
- Then serve packets in increasing order of their deadlines
- Think of it as an implementation of round-robin extended to the case where not all packets are equal sized

Analysis and Simulation of a Fair Queueing Algorithm

*Alan Demers
Srinivasan Keshav†
Scott Shenker*

Example



FQ vs. FIFO

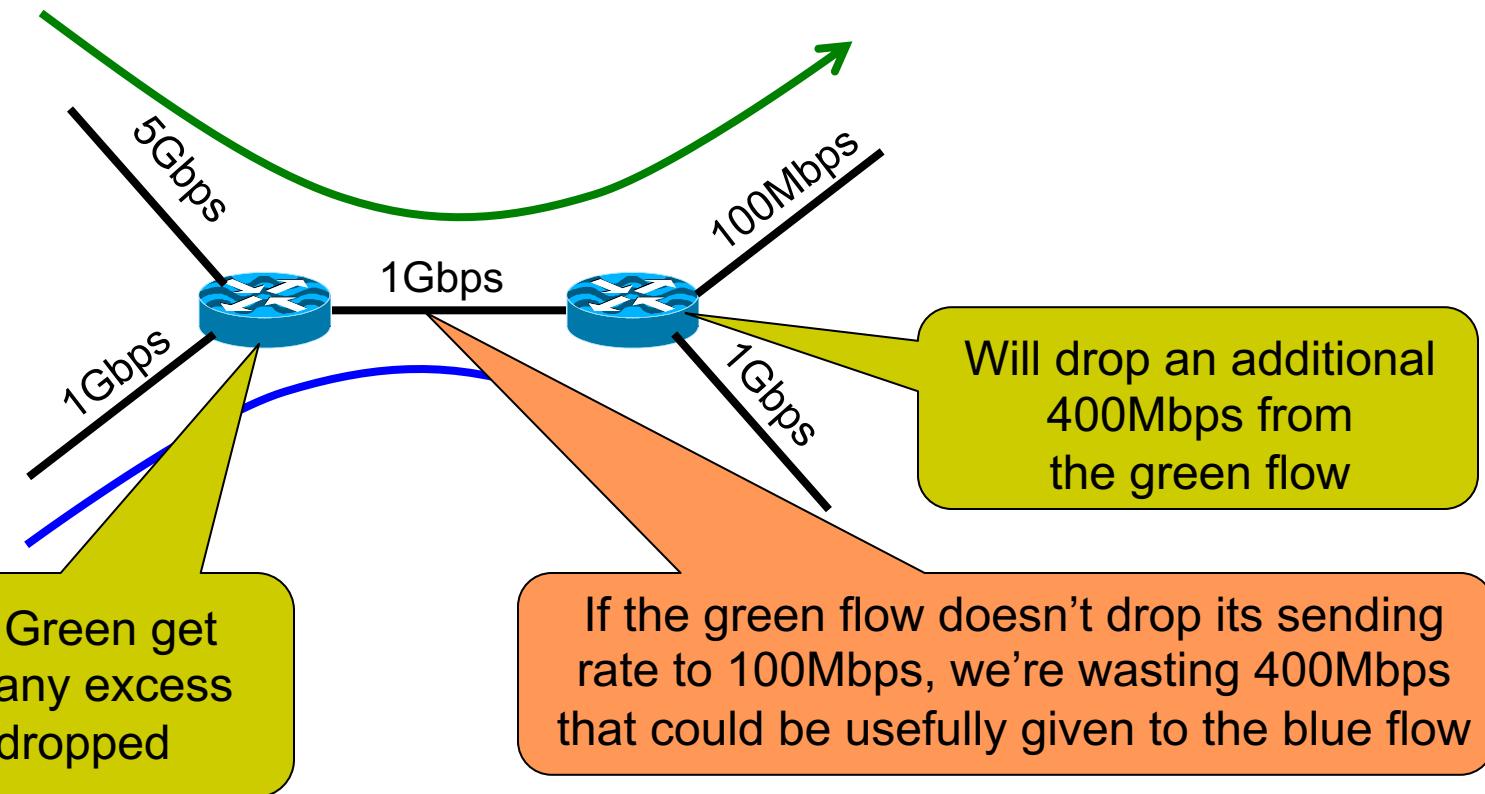
- FQ advantages:
 - Isolation: cheating flows don't benefit
 - Bandwidth share does not depend on RTT
 - Flows can pick any rate adjustment scheme they want
- Disadvantages:
 - More complex than FIFO: per flow queue/state, additional per-packet book-keeping
 - Still only a partial solution (coming up)

Fair Queuing In Practice

- “Pure” FQ too complex to implement at high speeds
- But several approximations exist
 - E.g., Deficit Round Robin (DRR)
- Today:
 - Routers typically implement approximate FQ (e.g., DRR)
 - For a small number of queues
 - Commonly used for coarser-grained isolation (e.g., for select customer prefixes) rather than per-flow isolation

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
- FQ's benefit is its resilience (to cheating, variations in RTT, details of delay, reordering, etc.)
- But congestion and packet drops still occur
- And we still want end-hosts to discover/adapt to their fair share!

Per-flow fairness is a controversial goal

- What if you have 8 flows, and I have 4?
 - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
 - Shouldn't you be penalized for using more of scarce bandwidth?
- And at what granularity do we really want fairness?
 - TCP connection? Source-Destination pair? Source?
- Nonetheless, FQ/DRR is a great way to ensure **isolation**
 - Avoiding starvation even in the worst cases

Router-Assisted Congestion Control

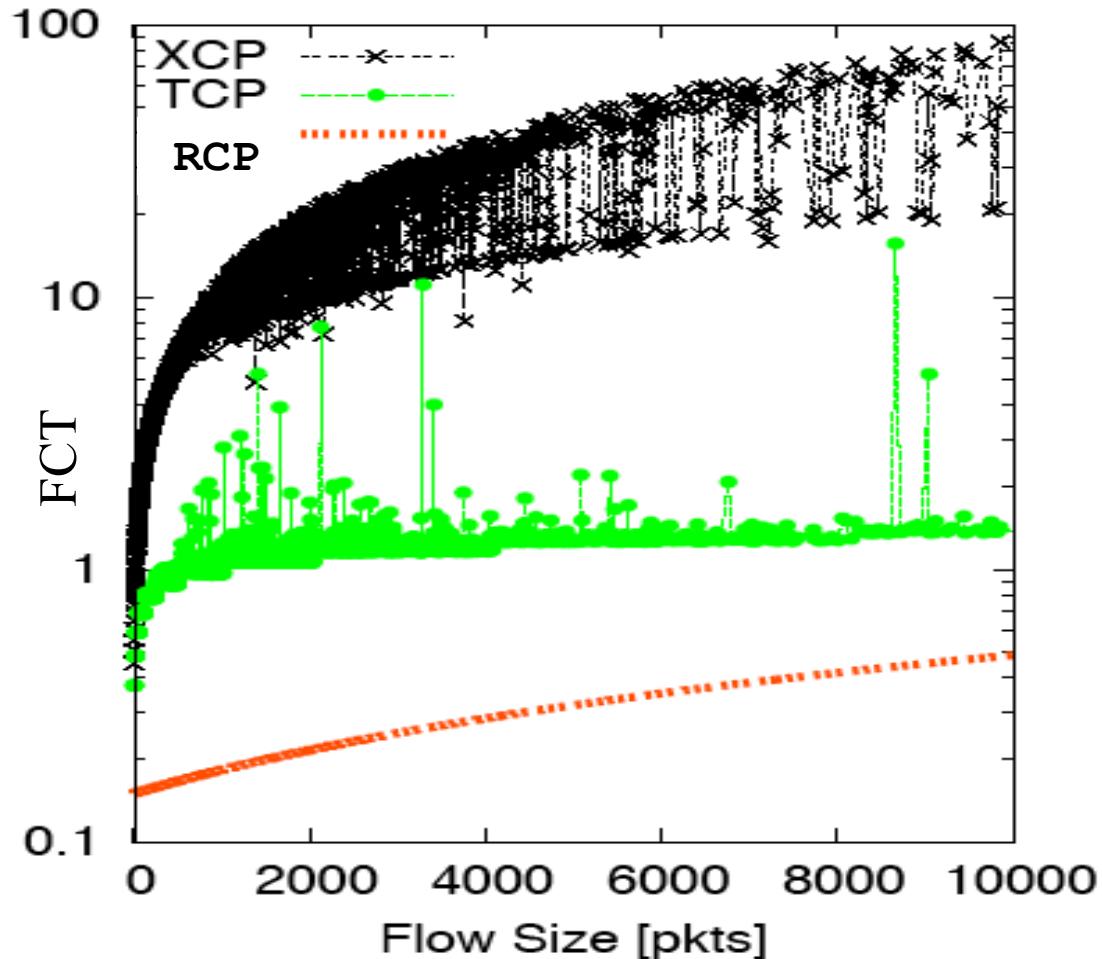
- Three ways routers can help
 - Enforce fairness
 - More precise rate adaptation
 - Detecting congestion

Why not just let routers tell endhosts what rate they should use?

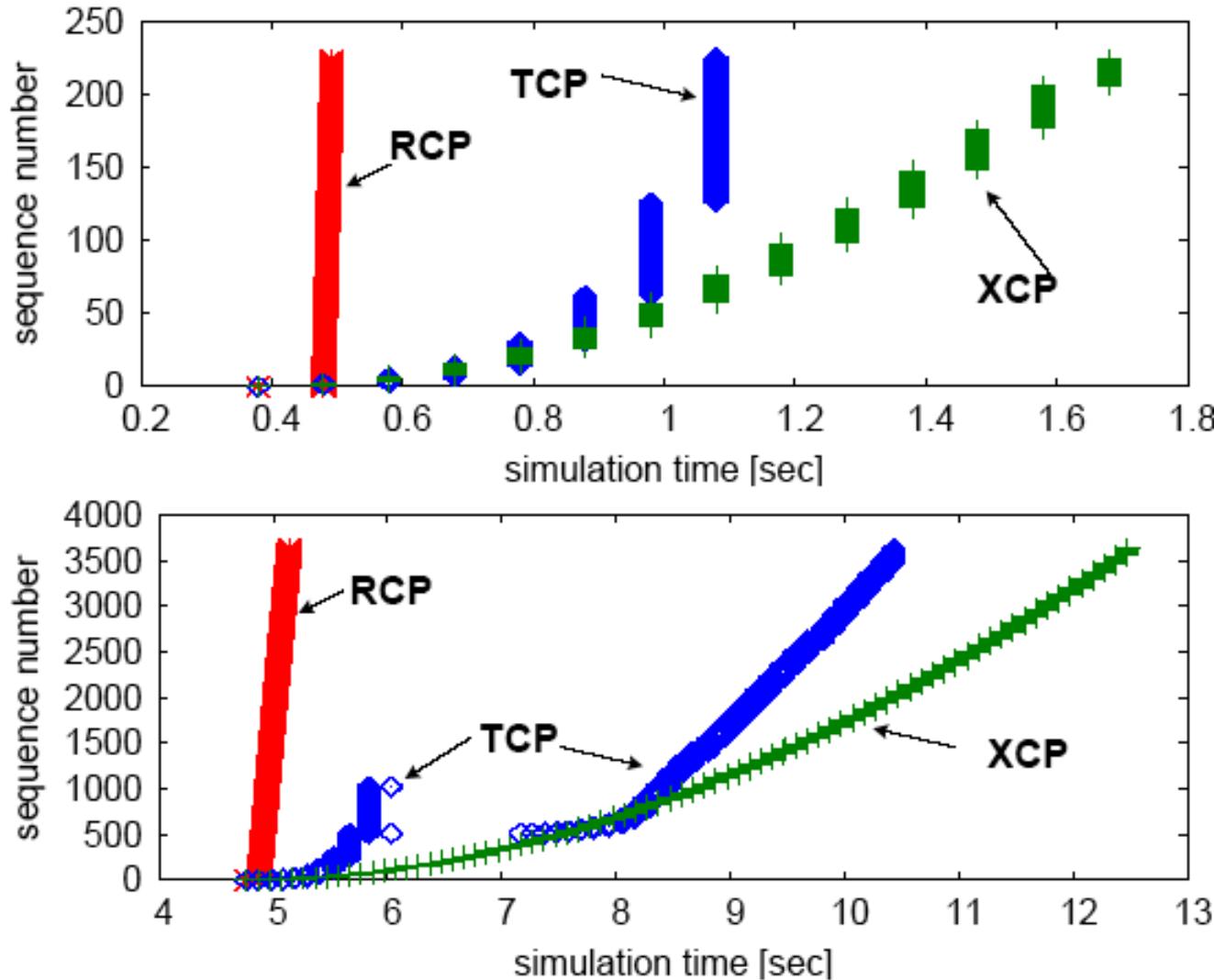
- Packets carry “rate field”
- Routers insert a flow’s fair share f in packet header
- End-hosts set sending rate (or window size) to f
- This is the basic idea behind the “Rate Control Protocol” (RCP) from Dukkipati *et al.* ’07

Flow Completion Time: TCP vs. RCP (Ignore XCP)

Flow Completion Time (secs) vs. Flow Size



Why the improvement?



Router-Assisted Congestion Control

- Three ways routers can help
 - Enforce fairness
 - More precise rate adaptation
 - Detecting congestion

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
 - If data packet has bit set, then ACK has ECN bit set
- Many options for *when* routers set the bit
 - Tradeoff between link utilization and packet delay
- Host can react as though it was a drop
- Advantages:
 - Don't confuse corruption with congestion
 - Early indicator of congestion → avoid delays
 - Lightweight to implement
- Today:
 - Widely implemented in routers
 - Some use in datacenters (e.g., Azure)

Final idea: Congestion-Based Charging

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay \$\$
 - The more congested the network, the more I pay
- No debate over what a flow is, or what fair is...
- Idea started by Frank Kelly at Cambridge
 - “optimal” solution, backed by much math
 - Great idea: simple, elegant, effective
 - But requires an entirely new charging model!

Recap: Router-Assisted CC

- FQ: routers *enforce* per-flow fairness
- RCP: routers *inform* endhosts of their fair share
- ECN: routers set “I’m congested” bit in packets
- Congestion pricing: users pay based on congestion

Perspective: Router-Assisted CC

- Can be highly effective, approaching optimal perf.
- But deployment is more challenging
 - Need support at hosts and routers
 - Some require more complex book-keeping at routers
 - Some require deployment at *every* router
- Though worth revisiting in datacenter contexts

Perspective: TCP CC

- Not perfect, a little ad-hoc
- But deeply practical/deployable
- Good enough to have raised the bar for the deployment of new, more optimal, approaches
- Though datacenters are reshaping the CC agenda
 - different needs and constraints (future lecture)

Next Topics

- The Domain Name System (DNS) and resolving names to addresses
- Remember: no in-person lecture on Thursday

DNS

The Domain Name System

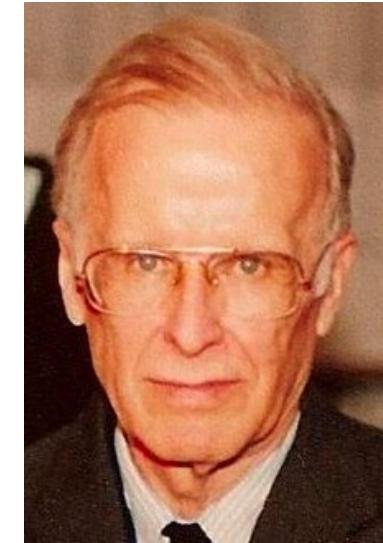
Today in CS

- 31 years ago, John Backus passed away

- Won the National Medal of Science
- Won the ACM Turing Award
- Led the team that developed Fortran
- He's the B in BNF

```
<postal-address>
    <name-part>
    <personal-part>
<street-address>
    <zip-part>
<opt-suffix-part>
    <opt-apt-num> ::= <apt-num> | "
```

```
HTTP-date      = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date   = weekday "," SP date2 SP time SP "GMT"
asctime-date   = wkday SP date3 SP time SP 4DIGIT
date1          = 2DIGIT SP month SP 4DIGIT
                  ; day month year (e.g., 02 Jun 1982)
date2          = 2DIGIT "-" month "-" 2DIGIT
                  ; day-month-year (e.g., 02-Jun-82)
date3          = month SP ( 2DIGIT | ( SP 1DIGIT ) )
                  ; month day (e.g., Jun 2)
time            = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                  ; 00:00:00 - 23:59:59
wkday           = "Mon" | "Tue" | "Wed"
                  | "Thu" | "Fri" | "Sat" | "Sun"
weekday         = "Monday" | "Tuesday" | "Wednesday"
                  | "Thursday" | "Friday" | "Saturday" | "Sunday"
month            = "Jan" | "Feb" | "Mar" | "Apr"
                  | "May" | "Jun" | "Jul" | "Aug"
                  | "Sep" | "Oct" | "Nov" | "Dec"
```



part> <name-part>

Where are we?

- Foundations / principles
 - e.g., Packet switching, end-to-end
- Domain structure of the Internet and...
 - routing within domains
 - routing between domains
- Deep dive on IP and TCP
 - What packets are actually composed of (at L3 and L4)
 - How to make an unreliable network (look) reliable
- Today we start looking at things which are more *user-facing*

The Domain Name System (DNS)

- Overview
 - Introduction
 - Name lookup
- Digging into the Details
 - API, servers, and protocol
 - A and NS records
 - How domain names are born
- More DNS
 - Record types and use cases
- A DNS case study
 - Minecraft and SRV records
- Availability, Scalability, and Performance
 - AKA four ways to add more servers
- DNS skepticism
 - Did we name the right thing?
 - Does this thing work right?
 - Is your privacy safe?
 - Does DNS matter?

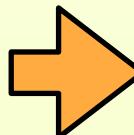
Thinking back...

- Three early “killer apps” of the Internet (or its precursor, the ARPANET)
 - Remote terminal
 - From local machine, log in to a machine somewhere else (like ssh)
 - telnet <remote host>
 - File transfer
 - From local machine, transfer files to/from a remote machine
 - ftp <remote host>
 - Email
 - Send/receive messages to/from user of a remote machine
 - mail <user>@<remote host>
- .. but numerical host addresses are not so nice for humans!
 - Nobody wants to type **telnet 46.0.0.10** !

Numerical addresses vs. humans

- Solution: create an “address book” of host names and their addresses
- Maintained by Elizabeth Jocelyn "Jake" Feinler at the Network Information Center (NIC) at SRI
 - If you wanted a hostname, phone Jake Feinler, she'd enter it in database
- Originally human-readable

Address	Hostname	Computer	Status/System	Address	Hostname	Computer	Status/System
001	UCLA-NMC	PDP-11/45	User 1/1/74/ANTS	148	ETAC-TIP		TIP
65	UCLA-CCn	IBM 360/91	Server	21	LLL-RISOS	PDP-11/45	User/RATS
129	UCLA-CCBS	PDP-10	limited Server	22	ISI-SPEECH11	PDP-11/45	User 1/74
2	SRI-ARC	PDP-10	dedicated Server/TENEX, NLS	86	USC-ISI	PDP-10	Server/TENEX
66	SRI-AI	PDP-10	limited Server/TENEX	150	ISI-DEVTENEX	PDP-10	User 1/74/TENEX
130	SU-DSL	(VDH)->PDP11/20	User 1/74/ANTS	23	USC-44	IBM 360/44	Server
3	UCSB-MOD75	IBM 360/75	Server/OLS	151	USC-TIP		TIP
67	SCRL	(VDH)->PDP11/45	User/ANTS	152	GWC-TIP		TIP
4	UTAH-10	PDP-10	limited Server/TENEX	153	DOC-B-TIP		TIP
132	UTAH-TIP		TIP	26	SDAC-44	IBM 360/44	User
5	BBN-11X	PDP-11	Peripheral processor for #69	154	SDAC-TIP		TIP
69	BBN-TENEX	PDP-10	Server/TENEX	28	ARPA-DMS	PDP-15	User
133	BBN-TENEXB	PDP-10	limited Server/TENEX	156	ARPA-TIP		TIP
6	MIT-MULTICS	H-6180	Server till 12/17/73/Multics	29	BRL	PDP-11/40	User/ANTS
70	MIT-DMS	PDP-10	Server/ITS	158	BBN-TESTIP		TIP
134	MIT-AI	PDP-10	Server/ITS	31	CCA-TENEX	PDP-10	dedicated Server/TENEX
198	MIT-ML	PDP-10	Server/ITS	95	LL-LANTS	PDP-11/40	User 2/74/ANTS
7	RAND-RCC	IBM 370/158	User	159	CCA-TIP		TIP
8	SDC-LAB	IBM 370/145	limited Server	32	PARC-MAXC	(Nova)->MAXC	limited Server/TENEX
9	HARV-10	PDP-10	Server	96	PARC-VTS	Nova 800	User
73	HARV-1	PDP-1	User	160	PARC-11	PDP-11	User 1/74
137	HARV-11	PDP-11	User	33	FNWC	CDC 6500	2/74
10	LL-67	IBM 360/67	limited Server	161	FNWC-TIP		TIP
74	LL-TX2	TX-2	Server	98	UCB	PDP-11/45	User 1/74
138	LL-TSP	TSP	User	35	UCSD-CC	B6700	Server
11	SU-AI	PDP-10	Server/SAIL	36	HAWAII-ALOHA	HP 2100	12/73
12	ILL-CAC	PDP-11/20	User/ANTS	100	HAWAII- 500	BCC 500	1/74
76	ILL-NTS	PDP-11/50	User/ANTS	164	ALOHA-TIP		TIP
140	UNIVAC	UNIVAC 1616	1/15/74	165	RML-TIP		TIP
13	CASE-10	PDP-10	Server/TENEX	40	BBN-NCC	H-316	User
14	CMU-10B	PDP-10	Server	168	NCC-TIP		TIP
78	CMU-10A	PDP-10	Server	232	BBN-1D	PDP-1	User
15	I4-TENEX	PDP-10	limited Server/TENEX	169	NORSAR-TIP		TIP
79	I4-TENEX	PDP-11	Peripheral processor for #15	42	UKICS-360	IBM 360/195	limited Server
16	AMES-67	IBM 360/67	Server	170	UKICS-TIP		TIP
144	AMES-TIP		TIP	43	OFFICE-1	PDP-10	dedicated Server/TENEX, NLS
208	AMES-11	PDP-11/45	User 12/73	171	TYMSHARE-TIP		TIP
145	MITRE-TIP		TIP	44	MIT-MULTICS	H-6180	Server 12/17/73/Multics
146	RADC-TIP		TIP	174	RUTGERS-TIP		TIP
19	NBS-ICST	PDP-11/45	User/ANTS	175	WPAFB-TIP		TIP
147	NBS-TIP		TIP				



Numerical addresses vs. humans

- Solution: create an “address book” of host names and their addresses
- Maintained by Elizabeth Jocelyn "Jake" Feinler at the Network Information Center (NIC) at SRI
 - If you wanted a hostname, phone Jake Feinler, she'd enter it in database
- Originally human-readable
- Eventually a standardized format (“<NETINFO>HOSTS.TXT” aka “hosts.txt”)
 - Machines could consume this directly
 - Everyone periodically uses FTP to fetch HOSTS.TXT from the NIC

HOST : 10.0.0.1 : UCLA-CS,UCLA-CECS : VAX-11/750 : LOCUS : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.16 : AMES-TSS,AMES-67,AMES : IBM-360/67 : TSS/360 : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.22 : ISI-SPEECH11 : PDP-11/45 : EPOS : TCP/TFTP :
HOST : 10.0.0.23 : USC-ECLB,ECLB : DEC-1090B : TOPS20 : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.26 : PENTAGON-TAC : H-316 : TAC : TCP :
HOST : 10.0.0.27 : USC-ISID,ISID : DEC-2060T : TOPS20 : TCP/TELNET,TCP/SMTP,TCP/FTP,TCP/TFTP,TCP/FINGER :
HOST : 10.0.0.32 : PARC-MAXC,PARC : MAXC : TENEX : TCP/FTP,TCP/SMTP,TCP/TELNET :
HOST : 10.0.0.34 : LBL-NMM,NMM : VAX-11/780 : VMS : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.37, 128.10.0.1 : PURDUE,PURDUE-CS,PURDUE-TCP,PURDUE-PVAX,PVAX : VAX-11/780 : UNIX : TCP/FTP,TCP/TELNET,TCP/SMTP :
HOST : 10.0.0.62 : UTEXAS-11 : PDP-11/70 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.68 : USGS1-MULTICS,RESTON,REST : H-60/68 : MULTICS : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.70 : USGS3-MULTICS,MENLO : H-6880 : MULTICS : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.73 : SRI-NIC,NIC : FOOONLY-F3 : TENEX : TCP/TELNET,TCP/SMTP,TCP/TIME,TCP/FTP,NCP/FTP,NCP/TELNET ;Reclama
HOST : 10.0.0.78 : UCB-ARPA : /780 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP,UDP :
HOST : 10.0.0.87 : SANDIA,SNL : DEC-2060T : TOPS20 : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.90 : LANL : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.0.0.91 : WASHINGTON,UDUB,UW-WARD : DEC-2060 : TOPS20 : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.1.0.1 : UCLA-CCN,CCN : IBM-370/3033 : OS/MVS : TCP/TELNET,TCP/FTP,TCP/SMTP,NCP/TELNET,NCP/FTP ;Reclama
HOST : 10.1.0.6 : MIT-DMS,DMS : DEC-1040 : ITS : TCP/TELNET,TCP/FTP,TCP/SMTP,TCP/FINGER :
HOST : 10.1.0.14 : CMU-CS-A,CMU-10A,CMUA : DEC-1080 : TOPS10 : TCP/TELNET,TCP/FTP,TCP/SMTP,TCP/FINGER,ICMP,NCP ;Reclama
HOST : 10.1.0.94, 192.5.2.3 : UWISC,CSNET-SH,CSNETB,CSNET,WISCONSIN : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.2.0.9 : YALE : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.2.0.58 : RUTGERS,RUTGERS-20,RUTGERS-10,RU-RED : DEC-2060T : TOPS20 : TCP/TELNET,TCP/FTP,TCP/SMTP,TCP/FINGER :
HOST : 10.2.0.78 : UCB-VAX,BERKELEY,UCB-C70 : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP,UDP :
HOST : 10.3.0.14 : CMU-CS-C,CMU-20C,CMUC : DEC-2060 : TOPS20 : TCP/TELNET,TCP/FTP,TCP/SMTP,TCP/FINGER,ICMP :
HOST : 10.3.0.24 : WHARTON-10,WHARTON : PLURIBUS : VDA : NCP/TELNET,NCP/FTP,TCP/FTP :
HOST : 10.3.0.96 : CORNELL : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP :
HOST : 10.5.0.53 : MARTIN,MMC : PDP-11/45 : RSX : TCP/TELNET,TCP/FTP ;Reclama

...

Berkeley also had its own network now!

```
HOST : 46.0.0.4 : UCBARPA : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.5 : UCBCAD : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.6 : UCBERNIE : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.7 : UCBMONET : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.9 : UCBESVAX : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.10 : UCBVAX : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.11 : UCBKIM : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.12 : UCBCALDER : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.13 : UCBDALI : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.14 : UCBMATISSE : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.15 : UCBMEDEA : VAX-11/750 : UNIX : TCP/TELNET,TCP/FTP,UDP :  
HOST : 46.0.0.19 : UCBINGRES : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,UDP :
```



You type:

\$ telnet UCBVAX

- UCBVAX looked up in hosts file
- opens connection to 46.0.0.10
- much nicer than telnet 46.0.0.10!

Class A network! Like a /8 !

By 1986, these were all 128.32.0.x instead (Class B — Berkeley still has this /16)

Numerical addresses vs. humans

- Solution: create an “address book” of host names and their addresses
- Maintained by Elizabeth Jocelyn "Jake" Feinler at the Network Information Center (NIC) at SRI
 - If you wanted a hostname, phone Jake Feinler, she'd enter it in database
- Originally human-readable
- Eventually a standardized format (“<NETINFO>HOSTS.TXT” aka “hosts.txt”)
 - Machines could consume this directly
 - Everyone periodically uses FTP to fetch HOSTS.TXT from the NIC
- But this wasn't ideal...

Numerical addresses vs. humans

- Increasing amount of work for Jake Feinler and her team!
- Increasing amount of data transfer!
 - As networks grows (more hosts)
 - file size increases
 - number of hosts fetching it increases
 - frequency with which you fetch to remain up to date increases
 - .. absolute best case is that this is quadratic!
 - .. were starting to be a *lot* more hosts (e.g., due to rise of workstations)
- Longer transfers more likely to fail; may end up with partial hosts file!
- In short:
 - Centralized administration was burdensome and counter to “open” trend
 - Centralized distribution of (increasingly) large file was bad news

The Domain Name System

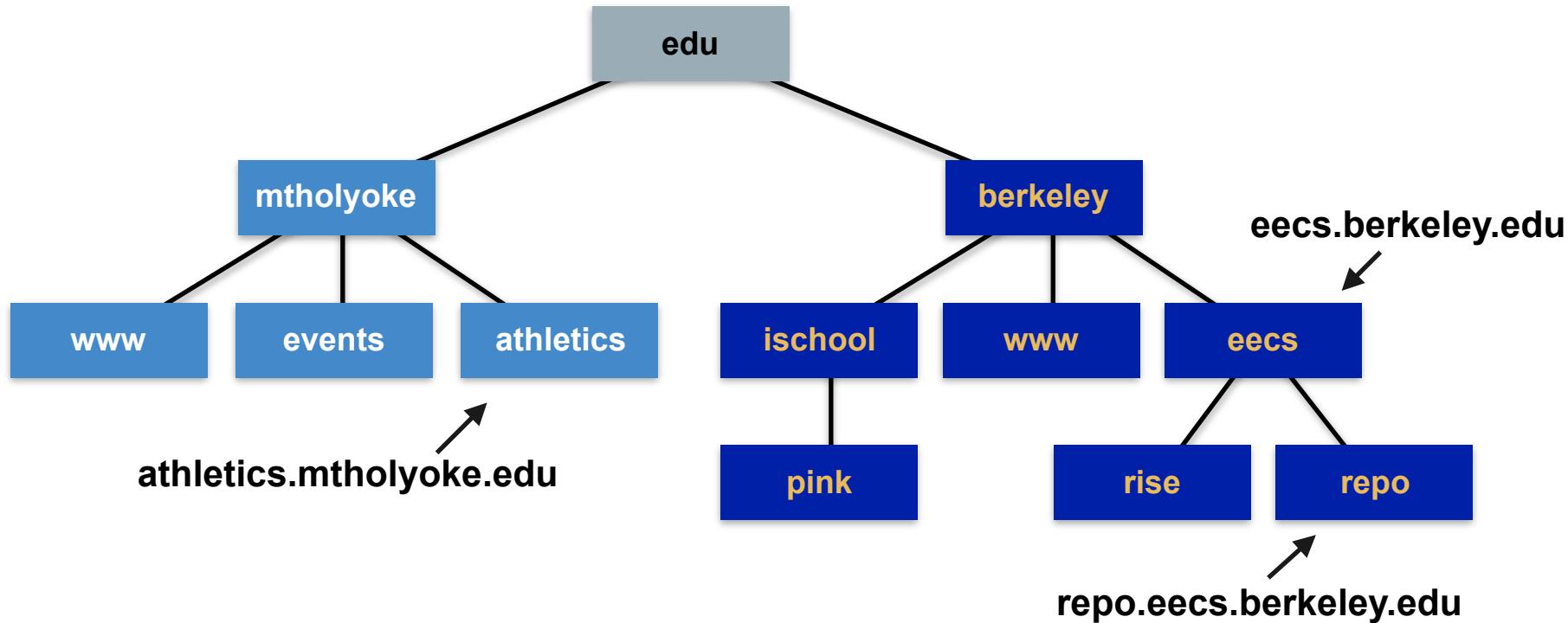
- DNS developed to confront the problems being faced
 - Developed by Paul Mockapetris; RFC in 1983
 - He was given the task of pulling several proposals into a final one...
 - .. just developed his own one instead!
 - .. without much change, we still use it today

The Domain Name System: Goals

- Primary purpose: map from human-friendly names to IP addresses
- Deal with scale!
 - Many hosts/names
 - Many name/address lookups
 - Many updates (can't have bottleneck at NIC)
- Be highly available
 - No single point of failure (what if the NIC's FTP server was down?)
- Perform well
 - Lots of communication starts with a name lookup!
- How do you solve these problems?
 - ~~Hierarchy!~~
 - Three intertwined *hierarchies*!

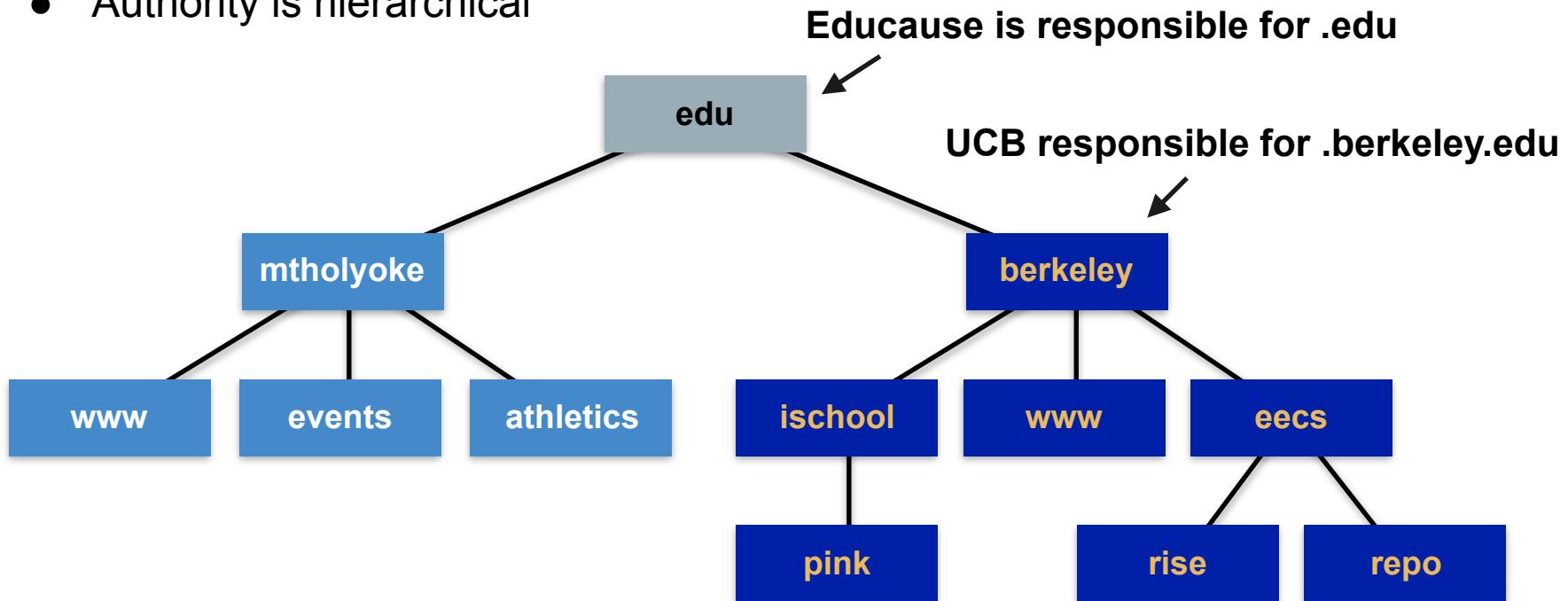
The Domain Name System: Hierarchies

- Names are hierarchical



The Domain Name System: Hierarchies

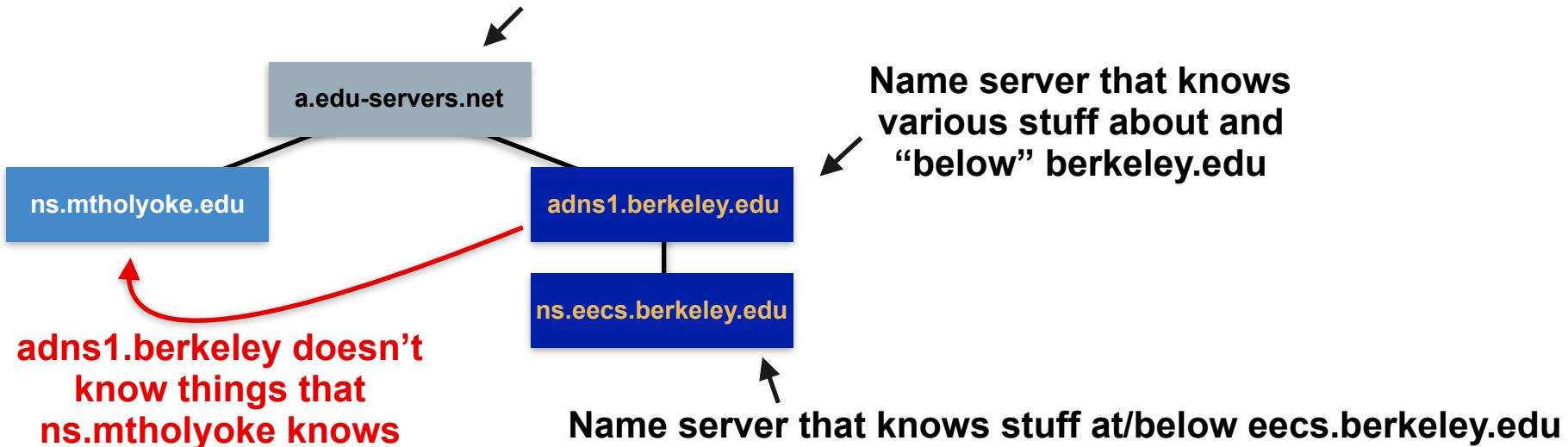
- Authority is hierarchical



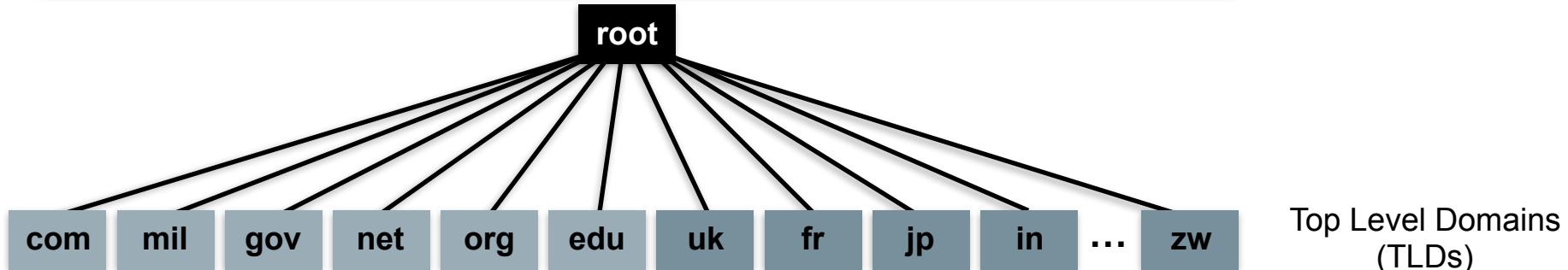
The Domain Name System: Hierarchies

- Infrastructure is hierarchical
 - Infrastructure is not just a single server that knows all the names
 - It's a *hierarchy* of *name servers* which know parts of the hierarchy

Name server that knows about name servers for all *.edu

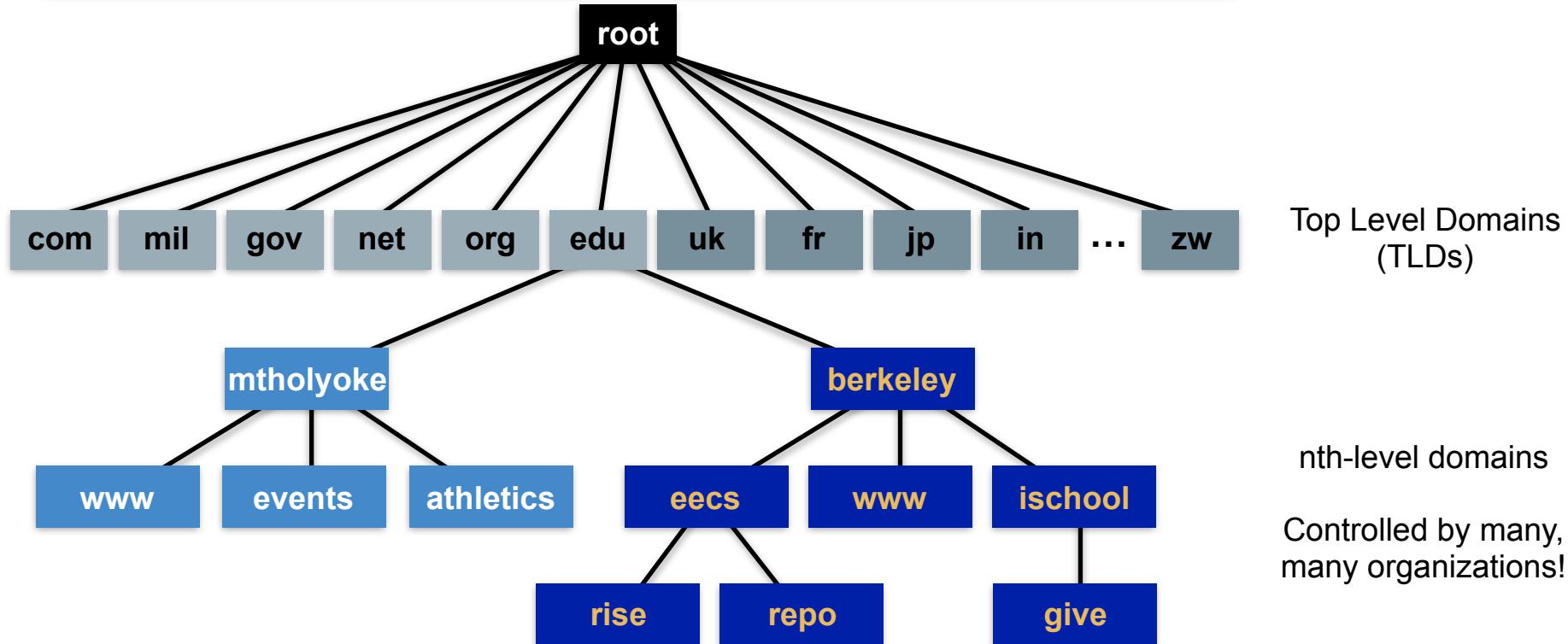


DNS: Bigger Picture



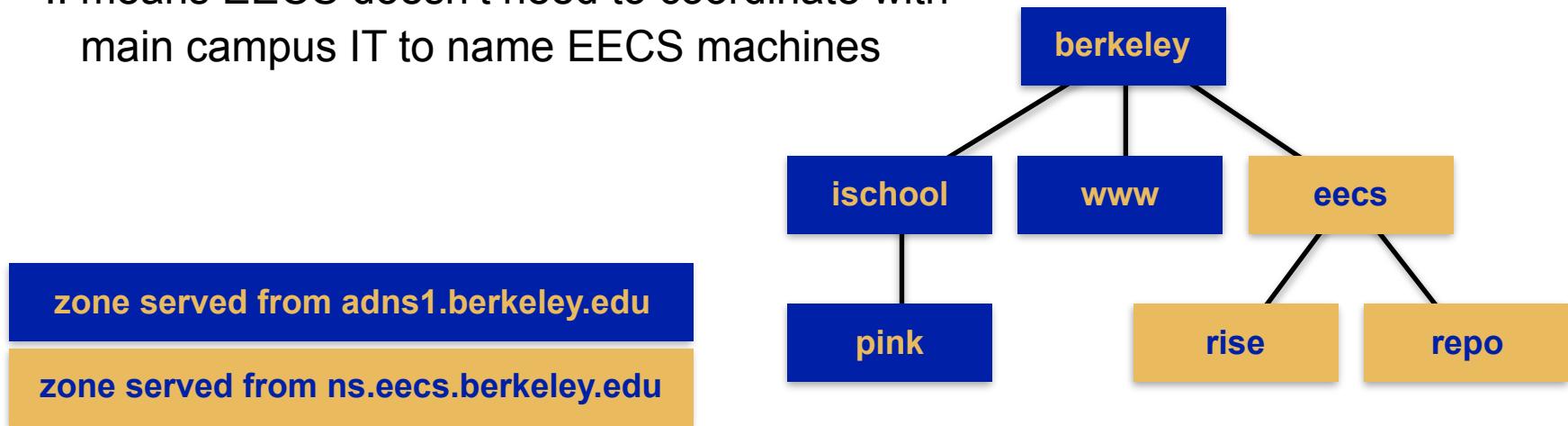
- DNS root
 - Controlled by ICANN
- Top Level Domains (TLDs)
 - Controlled by Educause (.edu), Verisign (.net, .com), AFNIC (.fr), US Government (.gov), etc., etc. (1,515 as of March 2020)

DNS: Bigger Picture



DNS: Zones, Authority, Delegation

- A **zone** corresponds to an administrative authority responsible for contiguous portion of hierarchy
- UCB controls *.berkeley.edu and *.ischool.berkeley.edu
- EECS controls *.eecs.berkeley.edu
- .. you have choice of whether/where to delegate authority of children
- .. means EECS doesn't need to coordinate with main campus IT to name EECS machines



DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want ←
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask

DNS: Name resolution

Example: Let's look up (or *resolve*) **repo.eecs.berkeley.edu**

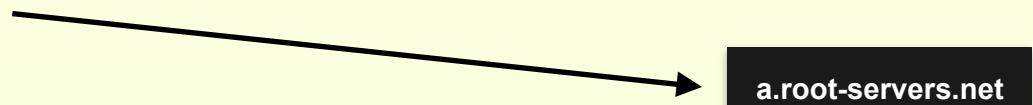
1. Ask [a.root-servers.net](#) for [repo.eecs.berkeley.edu](#)
2. It won't know, but it will tell you who to ask: [a.edu-servers.net](#)
3. Ask [a.edu-servers.net](#) for [repo.eecs.berkeley.edu](#)
4. It won't know, but it will tell you who to ask: [adns1.berkeley.edu](#)
5. Ask [adns1.berkeley.edu](#) for [repo.eecs.berkeley.edu](#)
6. It won't know, but it will tell you who to ask: [ns.eecs.berkeley.edu](#)
7. Ask [ns.eecs.berkeley.edu](#) for [repo.eecs.berkeley.edu](#)
8. It will tell you: 128.32.138.46 !



DNS Sidenote: Classes of name servers

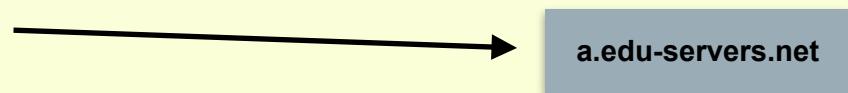
Root server

Knows about all the TLD servers



TLD server

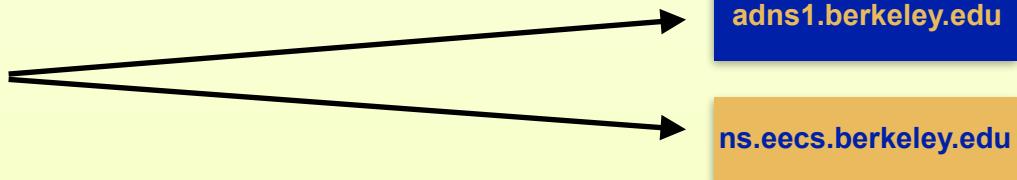
Knows about a particular TLD (e.g., .edu)



Authoritative servers

Know about stuff in their zone

Actually do name to IP mapping!



Can be operated by an organization itself (e.g., UCB), or by a service provider

DNS: Name lookup

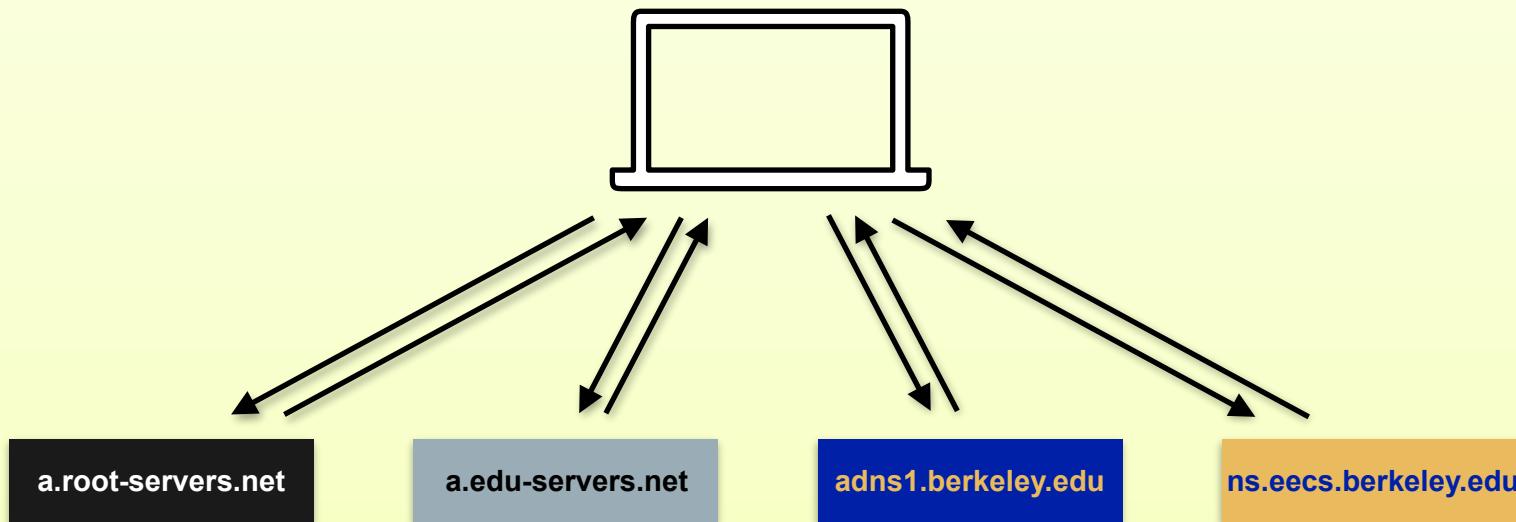
- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want ←
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- 

DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want ←
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- Three important questions here:
 - 1) Who actually does this multi-step lookup process?

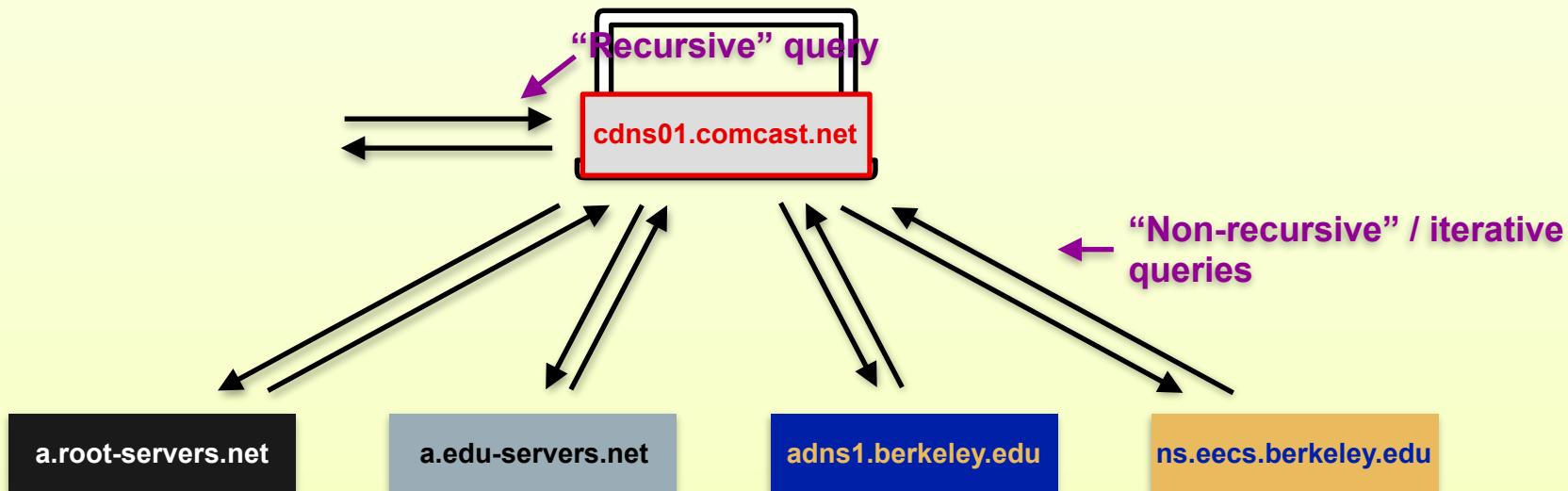
DNS: Name lookup

- Who actually does this multi-step lookup process?
- Originally, likely that host did it directly



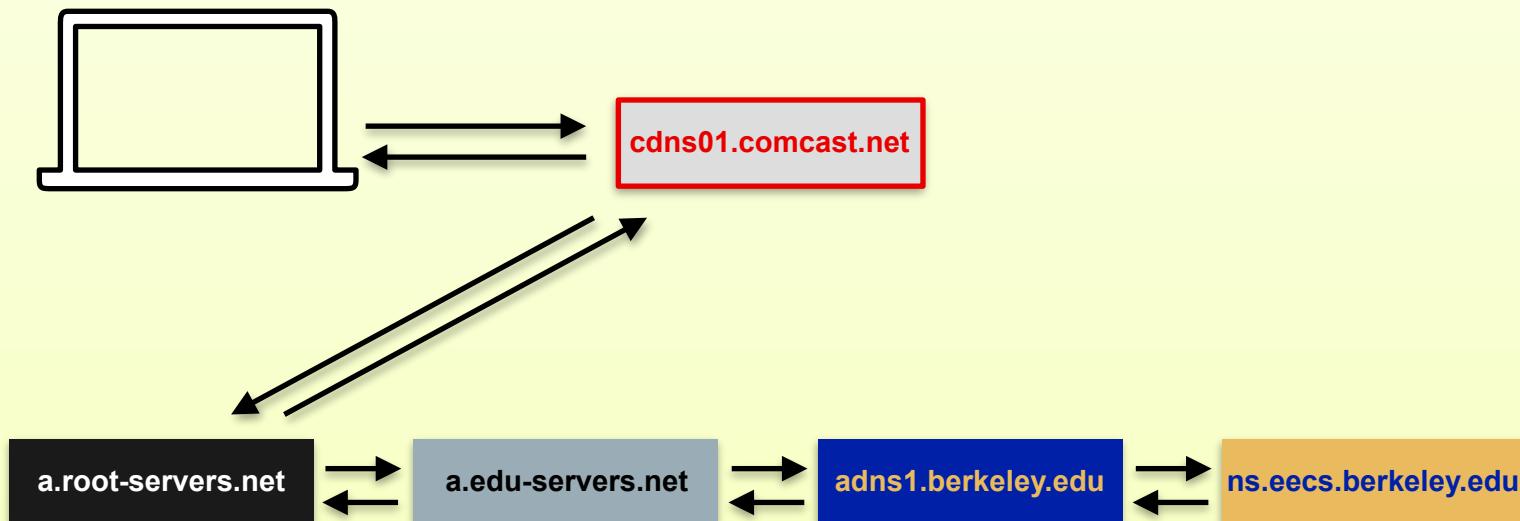
DNS: Name lookup

- Who actually does this multi-step lookup process?
- Today, usually done by a *resolving name server*



DNS: Name lookup

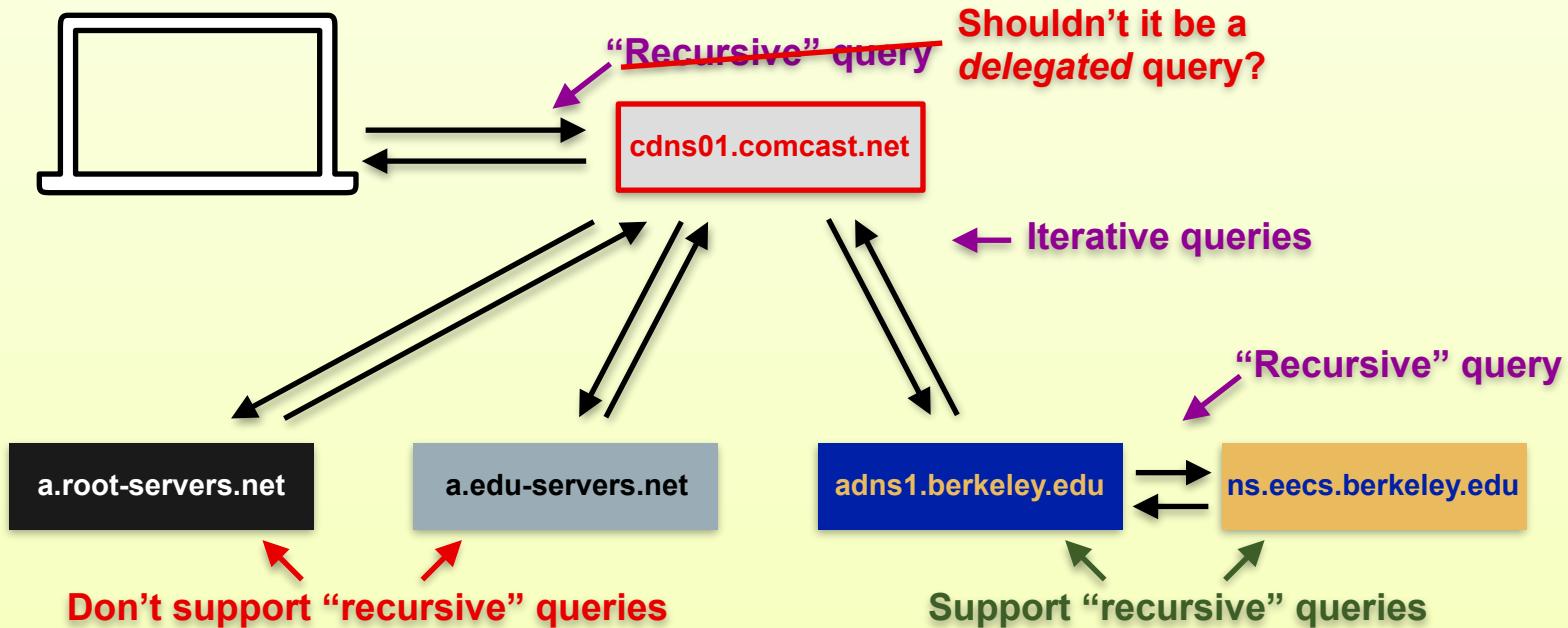
- Who actually does this multi-step lookup process?
- Today, usually done by a *resolving name server*



These servers don't support “recursive” queries — it's harder and they're busy!

DNS: Name lookup

- Who actually does this multi-step lookup process?
- Today, usually done by a *resolving name server*



DNS: Name lookup

- When a server gets a request for a normal/non-recursive query:
 - If server knows the answer — return answer!
 - If not — return reference to next server to query
- When a server gets a request for a recursive query:
 - If server knows the answer — return answer!
 - In theory, *could* perform recursive query on “next” server ← **Truly recursive**
 - More likely this server does the “iterative” process itself ← **Not really recursive?**
 - Even more likely: return an error saying you don’t support “recursion”
- .. usually only specialized resolving servers support these queries
 - Often provided by your ISP
 - Generally aren’t authoritative for any domain (don’t have specific name-to-IP mappings that they’re responsible for)

DNS Sidenote: Classes of name servers

Root server

Knows about all the TLD servers



TLD server

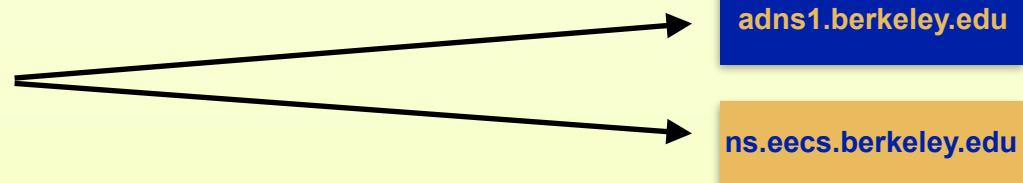
Knows about a particular TLD (e.g., .edu)



Authoritative servers

Know about stuff in their zone

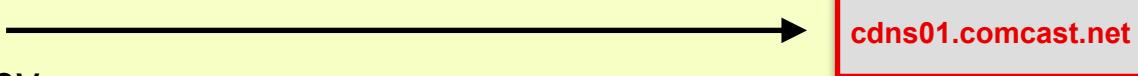
Actually do name to IP mapping!



Resolving DNS servers

Just for delegating lookups

Not really part of the hierarchy

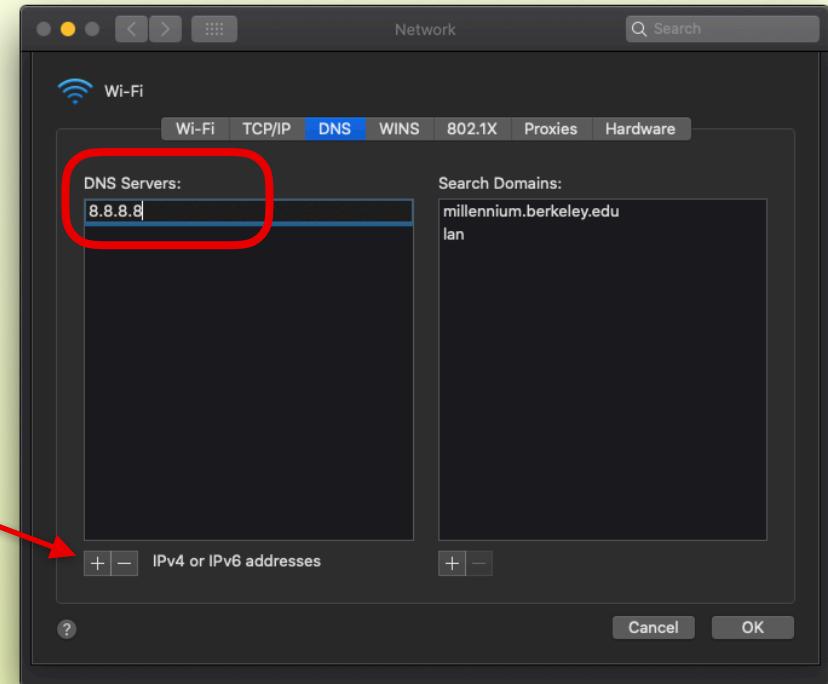


DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- Three important questions here:
 - 1) Who actually does this multi-step lookup process?
 - A host can do it, but probably delegates it to a *resolving DNS server*
 - 2) How do I know the address of my resolving DNS server?

DNS: Name lookup

- How do you know the address of your resolving DNS server?
- Possibly: Configure it manually
- More likely: DHCP
Dynamic Host Configuration Protocol
(We'll cover later)
- Note: You can have more than one!
Depending on OS/config, may cycle
through them, or switch to later one
if earlier one fails, or...



DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- Three important questions here:
 - 1) Who actually does this multi-step lookup process?
 - A host can do it, but probably delegates it to a *resolving DNS server*
 - 2) How do I know the address of my resolving DNS server?
 - Could be manual, but probably via DHCP (more later)
 - 3) How does anyone know the address of the root DNS server?!
 - First, must come clean about a fib...

DNS: Availability (Preview)

- I've been acting like there's one root name server, like Berkeley had one name server, and so on
- This is already somewhat resilient to failure...
 - Berkeley's name server could go down; would not affect UCLA
- But actually, every zone always has *at least two* name servers (replicas)
 - The main Berkeley zone has at least:
 - adns1.berkeley.edu - 128.32.136.3
 - adns2.berkeley.edu - 128.32.136.14
 - adns3.berkeley.edu - 192.107.102.142
 - .edu has 13 ("a" through "m" .edu-servers.net)
 - root also has 13 ("a" through "m" .root-servers.net)
 - .. actually, more. We'll come back to this.

By IETF decree, more or less

DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want ←
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- Three important questions here:
 - 1) Who actually does this multi-step lookup process?
 - A host can do it, but probably delegates it to a *resolving DNS server*
 - 2) How do I know the address of my resolving DNS server?
 - Could be manual, but probably via DHCP (more later)
 - 3) How does anyone know the address of ~~the~~ ^aroot DNS server?!

DNS: Name lookup

- How do you know the address of a root name server?!
- You know it's named a.root-servers.net or b.root-servers.net, etc., but...
 - Where do you look that up to find the IP address?!
 - Bit of a chicken and egg problem here
- Multiple ways, but a decent solution isn't too complicated...
 - Program that does name resolution ships with root server IP addresses (possibly hard coded, possibly in default config file)
 - Try query those pre-configured addresses until you find one that works
 - Ask it for an up-to-date list
 - Called a *priming* query
 - Works as long as at least one of the pre-configured ones still works

DNS: Name lookup

- “Iterative” resolution process:
 - Start with root name server
 - Ask for the name you want
 - If it has an answer — you’re done!
 - If not, it will direct you to next name server to ask
- Three important questions here:
 - 1) Who actually does this multi-step lookup process?
 - A host can do it, but probably delegates it to a *resolving DNS server*
 - 2) How do I know the address of my resolving DNS server?
 - Could be manual, but probably via DHCP (more later)
 - 3) How does anyone know the address of a root DNS server?!
 - Preconfigured addresses; use those to get updated ones (*priming*)

DNS: Name lookup

- A final note on lookup...
- Remember that HOSTS.TXT file from the pre-DNS world?
- Legacy of it remains today on many systems...
 - `/etc/hosts` on Unix-like systems (macOS, Linux, ...)
 - `C:\Windows\System32\Drivers\etc\hosts` on Windows
- .. but there's usually not much in it!

DNS

Digging into the Details

DNS: Digging into the details

- APIs
- Servers
- Protocol
- How a domain is born

DNS: API, servers, and protocol

- The usual API functions:
 - `result = gethostbyname("example.com");`
 - Very old; deprecated for many years
 - Wildly common in real code anyway
 - Limited to IPv4
 - `error = getaddrinfo("example.com", NULL, NULL, &result);`
 - Modern
 - Not limited to IPv4
 - Available in C on Unix-like systems and Windows
 - Available in Python socket module
- These usually just make a request to the configured resolving DNS server

DNS: API, servers, and protocol

- Gold standard DNS server: BIND
 - First DNS server for Unix
 - Written by four Berkeley grad students in 1983 (same year as DNS RFC)
 - Berkeley Internet Name Domain Server
 - .. why not Berkeley Internet Name Daemon?!

The Berkeley Internet Name Domain Server

Douglas B. Terry, Mark Painter, David W. Riggle, and Songnian Zhou

Computer Systems Research Group

Computer Science Division

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

ABSTRACT

The Berkeley Internet Name Domain (BIND) Server allows a host to query in the many types of objects and resources that

Sidenote: Daemons

- Many network server processes are called *daemons*
 - The main program of BIND is called “named” (name daemon)
 - SSH server is “sshd”
- Not strictly just network servers
 - Sometimes referred to as “background” or “non-interactive” processes
 - Sort of misleading — a name server is certainly interactive!
 - .. but not generally run/used directly from command line
 - Roughly: A daemon is a “server process”
 - Generally long lived
 - Generally communicated with via some sort of IPC or network
 - Equivalent programs in Windows world usually called *services*
- Why “daemon”?
 - Goes back at least as far as Descartes...

DNS: API, servers, and protocol

- Gold standard DNS server: BIND
 - First DNS server for Unix
 - Written by four Berkeley grad students in 1983 (same year as DNS RFC)
 - Berkeley Internet Name Domain Server
 - Perhaps it should not be surprising...
 - .. berkeley.edu is the oldest .edu domain on the Internet!

DNS: Protocol

- Client/Server design
 - Client is often a user host; could be another server (e.g., recursive query)
 - Client sends query
 - Server replies with response
- Server typically listens on well-known UDP port 53
- Why UDP?
 - Saves RTT for TCP connection establishment
 - TCP requires servers to keep state per connection... *lots* of connections
 - No real need for ordered stream abstraction; a single packet is often fine
- But wait... UDP is not reliable! What if packets are dropped?
 - Simple timeout/retry mechanism
 - Varies from OS to OS, etc. (but can be fairly slow)

DNS: Protocol

- Some DNS servers also use TCP port 53
 - Not usually used for normal queries
 - Primarily used for “zone transfers” (replicating name database)
 - This is much more data than a normal query!
 - Three-way handshake likely negligible; reliability/ordering important
- We'll talk about some more variants of the protocol later...

DNS: Protocol

- All messages share the same basic format
- Messages may be:
 - Query (“QR” bit in header is 0)
 - Response (“QR” bit in header is 1)
- Queries may *theoretically* be of several different types
 - **IQUERY** obsoleted in 2002 (RFC 3425)
 - "has not been generally implemented and has usually been operationally disabled where it has been implemented."
 - **STATUS** never really defined
 - *Proposed* standard in 2001 (DNS was 18 years old by this time)
 - **QUERY** is used for basically everything
- “RD” bit in header is *recursion desired* — requests “recursive” lookup

See text for more details on message format (or RFC 1035)

DNS: Protocol

- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)

DNS: Protocol

- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)
- Many types!
- Remembering primary goal of DNS (map human-friendly names to IP addrs)...
The two types we need for that are:
 - A records (address)
 - NS records (name server)
- We'll talk about other types later...

DNS: Protocol

- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)
- Name associated with the record
- For **A** records, this is a hostname of interest, e.g., www.google.com

DNS: Protocol

- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)
- Value associated with the record
- For **A** records, this is the IPv4 address associated with name

DNS: Protocol

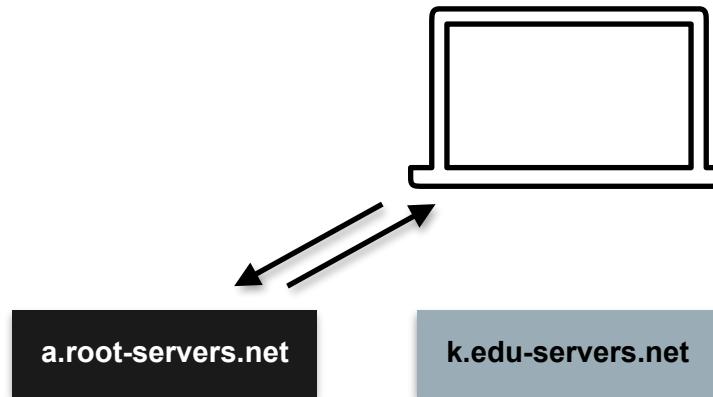
- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)
- How long (in seconds) the record is valid for
- May omit this going forward
- We'll come back to it later

DNS: Protocol

- The actual data stored in the DNS is held in *resource records* (RRs)
- Essentially a tuple: (type, name, value, ttl, class)
- DNS can be used for network types besides the Internet
 - *class* field specifies what network type
- Don't think this was ever used much (*class=Internet* almost always)
- We'll ignore it

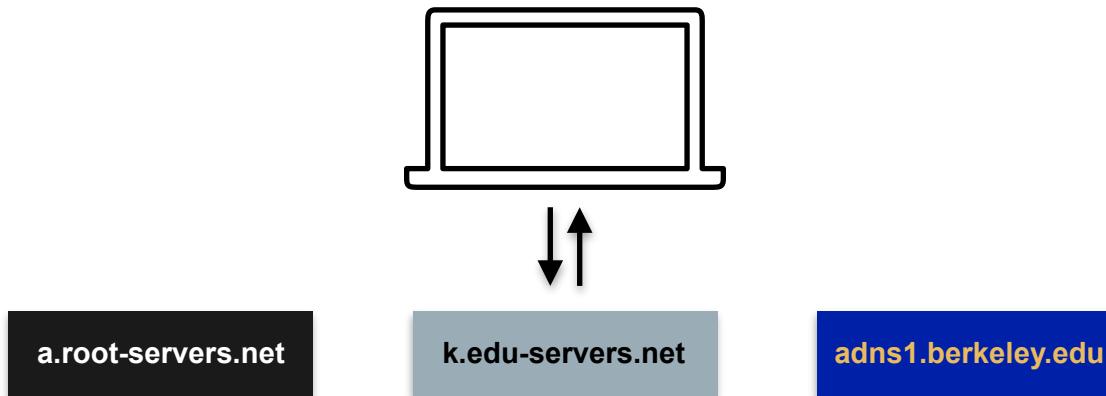
DNS: Protocol example

- Query root server requesting A record for [ischool.berkeley.edu](#)
- It sends back (NS, [edu](#), [k.edu-servers.net](#)), (NS, [edu](#), [l.edu-servers.net](#)), ...
 - Not what we asked for, but tells us our next step!
- Also sends (A, [k.edu-servers.net](#), 192.52.178.30), ...
 - “Additional” record(s)
 - It's probably what we would have asked for next!



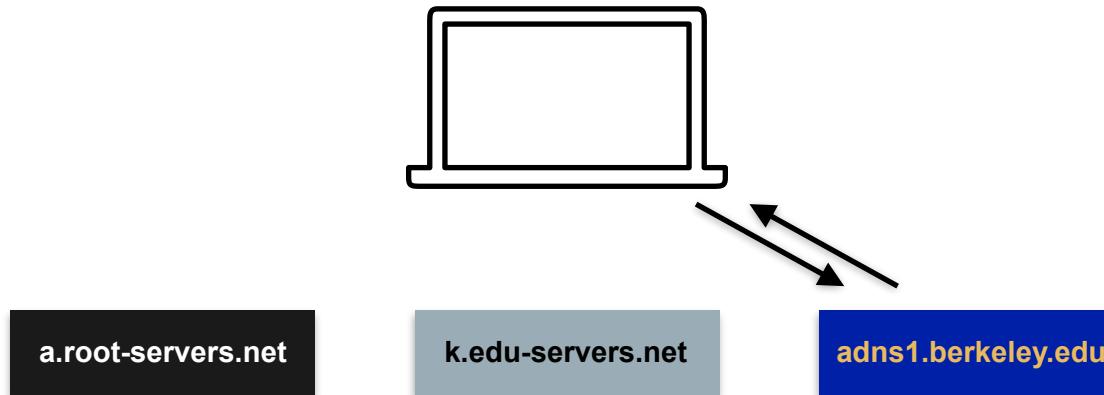
DNS: Protocol example

- Query [k.edu-servers.net](#) requesting A record for [ischool.berkeley.edu](#)
- It sends back (NS, [berkeley.edu](#), [adns1.berkeley.edu](#)), ...
- Also sends (A, [adns1.berkeley.edu](#), 128.32.136.3), ...



DNS: Protocol example

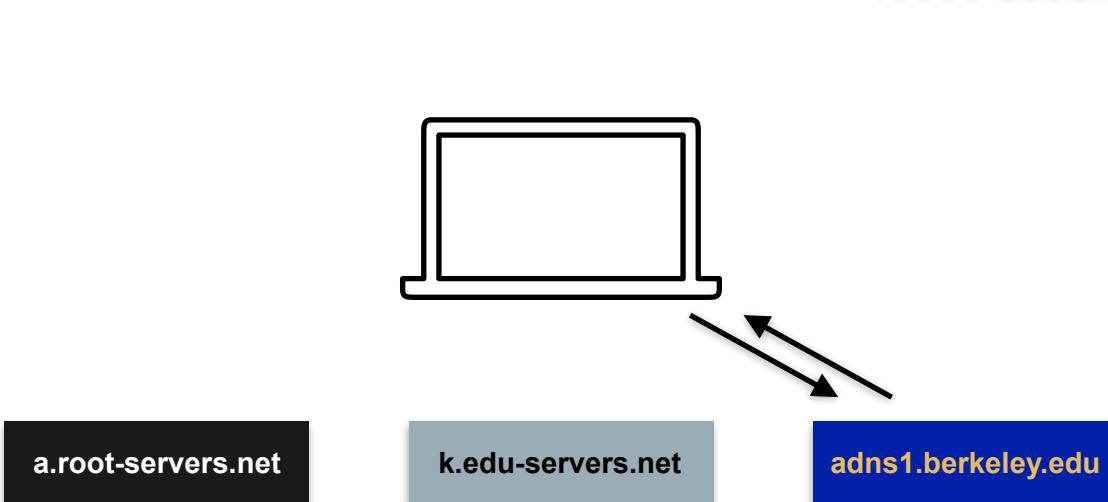
- Query [adns1.berkeley.edu](#) requesting A record for [ischool.berkeley.edu](#)
- It sends back (A, [ischool.berkeley.edu](#), 128.32.78.26)
 - That's what we wanted!



DNS: Protocol example

- Query [adns1.berkeley.edu](#) requesting A record for [ischool.berkeley.edu](#)
- It sends back (A, [ischool.berkeley.edu](#), 128.32.78.26, 10800)
 - That's what we wanted!

Can keep using this IP address for
10800 seconds (3 hours)



The diagram illustrates the DNS query process. At the top is a simple line drawing of a laptop. Three arrows originate from the laptop and point downwards to three colored rectangular boxes representing DNS servers. The first box is black and contains the text "a.root-servers.net". The second box is grey and contains "k.edu-servers.net". The third box is blue and contains "adns1.berkeley.edu".

Moving on...

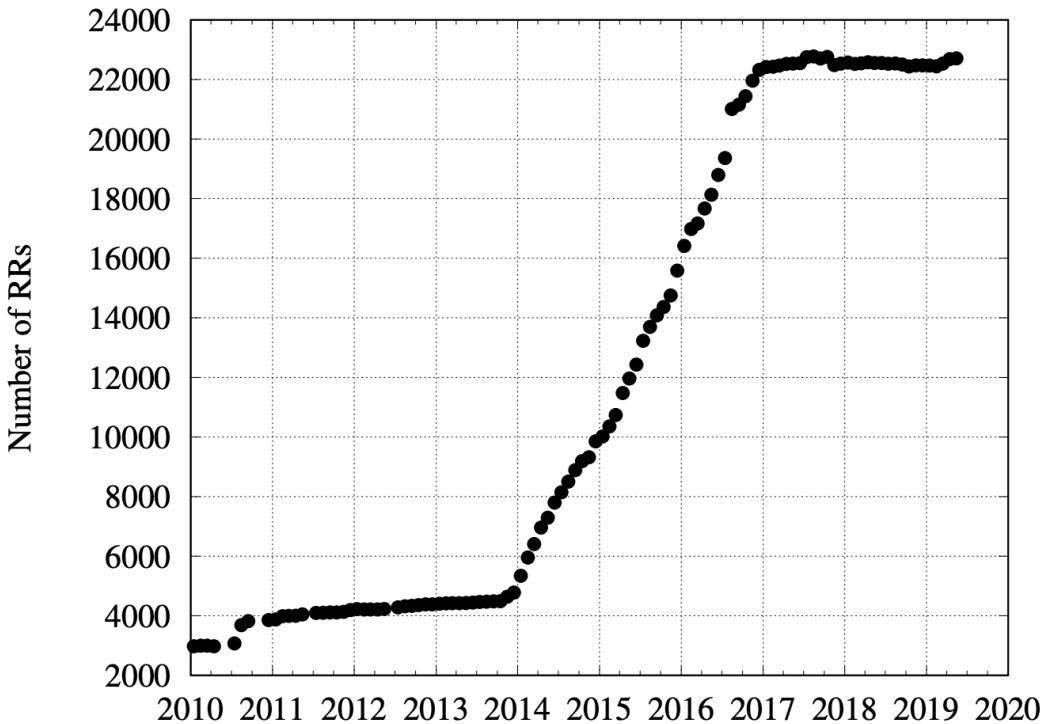
DNS: How is a domain name created?

- Example: you just created company Example Industries
- You get a block of IP addresses from your ISP
 - e.g., 192.0.2.0/25
- Register [example.com](#) with *registrar* (e.g., GoDaddy)
 - Probably less than \$15/year
- Run two authoritative name servers for your domain (or have someone run them for you)
- Give your name server addresses to your registrar
- Registrar inserts pairs of records for them into TLD name servers, e.g.:
 - (NS, [example.com](#), [ns1.example.com](#))
 - (A, [ns1.example.com](#), 192.0.2.6)
- Store resource records in your servers!
 - e.g., type A record for [www.example.com](#) - (A, [www.example.com](#), 192.0.2.1)
 - Costs you basically nothing to create any subdomains you want

DNS: How is a domain name created?

- What if I want my own top level domain?
 - I want to be murphy@awesome.cs168 !
- Talk to ICANN...
 - Get your own for the low, low price of about \$185,000?
 - (If we all chip in, it's only about \$370 per person.)
 - (Just saying.)

Number of records at root over time



[Data from Allman 2019]

DNS: Beyond the Basics

More DNS: Multiple A records

- There might be more than one A record with the same name!
- Server returns multiple A records
- Shuffles the order
- Allows coarse-grained load balancing
 - .. different users look up yahoo.com
 - .. get different IP addresses
 - .. contact different servers
- Allows simple resiliency
 - .. if first one doesn't work, try next

[~]\$

More DNS: IPv6

- Everything we've looked at so far used IPv4 addresses
- Want IPv6?
 - Ask for an **AAAA** record

```
[~]$ dig +short www.google.com A  
172.217.0.36
```

```
[~]$ dig +short www.google.com AAAA  
2607:f8b0:4005:808::2004
```

More DNS: Reverse lookups

- What if I have an address, e.g., 138.110.1.200 ?
 - What's its hostname?
- PTR record
 - Value is an associated hostname
 - Name is:
 - Dot-quad IP address ***listed backwards***
 - 138.110.1.200 → 200.1.110.138
 - Followed by .in-addr.arpa

Similar mechanism for IPv6 using
ip6.arpa

```
[~]$ dig +short 200.1.110.138.in-addr.arpa PTR
ns.mtholyoke.edu.
```

More DNS: Name aliasing

- CNAME record
 - “Canonical name”
 - Allows you to define an alias for another name

```
[~]$ dig www.berkeley.edu | clean | head -1  
www.berkeley.edu. 185 IN CNAME www-production-1113102805.us-west-2.elb.amazonaws.com.
```



- www.berkeley.edu name translates to an amazonaws.com name
 - (Because Berkeley's main website is hosted by Amazon)
- Next step would be to look up the A record for the amazonaws.com name
 - (Actually, server included it — the “head -1” hid it)
- Similar DNAME record maps a whole subtree:
 - WHATEVER.foo.com → WHATEVER.bar.com

More DNS: Email

- Send an email to murphy@berkeley.edu and I get it... at google.com?
- How? Why?
 - berkeley.edu is hosted by Amazon, not Google!
- Even in past, mail server was often separate machine, e.g., mail.berkeley.edu
 - Nobody wants to address messages to murphy@mail.berkeley.edu!
- Email servers look up **MX** record (*mail exchanger*) of recipient domain
 - This tells the mail server(s) to use for mail to that domain

```
[~]$ dig berkeley.edu MX | clean
berkeley.edu.      219 IN  MX  1  aspmx.l.google.com.
berkeley.edu.      219 IN  MX  5  alt2.aspmx.l.google.com.
berkeley.edu.      219 IN  MX  5  alt1.aspmx.l.google.com.
berkeley.edu.      219 IN  MX  10 alt3.aspmx.l.google.com.
berkeley.edu.      219 IN  MX  10 alt4.aspmx.l.google.com.
```

More DNS: TXT records

- **TXT** records were originally meant for human-readable information
- These days, often used for things like *site verification*

```
[~]$ dig +short berkeley.edu TXT | grep veri  
"adobe-idp-site-verification=a113c870-3c49-4b4a-b3a4-31e1cf1860cb"  
"Z00M_verify_RirbP7N1QWC3Zzm02oL4Cw"  
"google-site-verification=fL93jj-VPnl_5wdFDh26YshzKVPrAurHaBCu-k-Xw"  
"google-site-verification=loQrJWyMsMB249uINb-AsRGTwVoLdTc44Td3aMGn-NE"
```

- Adobe, Zoom, Google, Facebook, etc. give you a magic value
- You put it in TXT record on your domain — proves you have control over domain
- Unlocks capabilities on other site
 - Google will show how often your site shows up in results
 - Facebook lets you edit how shared links to your site appear

More DNS: SRV records

- MX was this special-purpose redirection for email
- What about other services?
 - Do they all need their own special record types?
 - Seems silly
 - **SRV** record solves similar problem for arbitrary services

- Record name
 - Records for services
 - See [Minecraft case study video](#) if you want to see how this is used in a real-world scenario.
 - Pros
 - Cons
- Easy to add more services — just create more SRV records



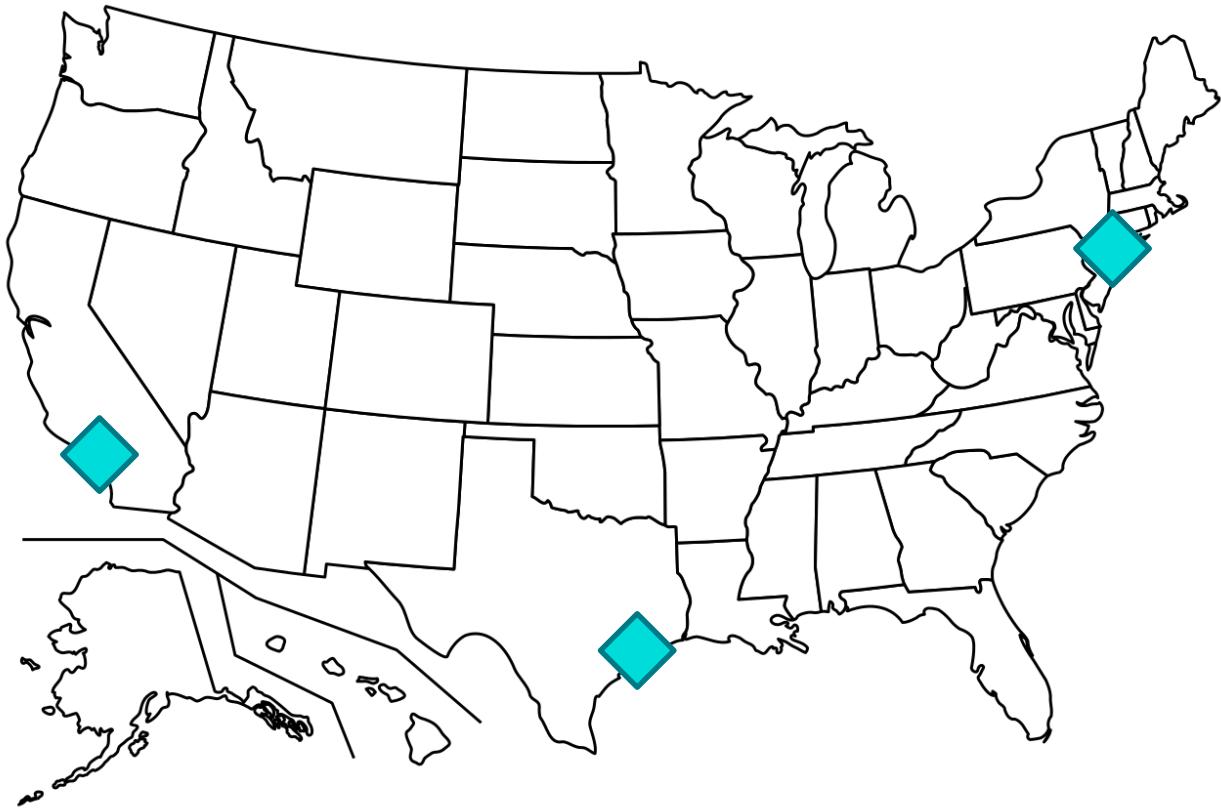
e.here

More DNS: Simple Indirection

- Remember, in an ideal world, your IP address is topologically meaningful!
 - e.g., it's a sub-allocation of your provider's address range
- What if you have a popular website, www.example.com...
 - The server is at 203.0.113.4...
 - And you switch providers...
 - And new provider gives you 198.51.100.88 ?
- .. just update www.example.com's A record to 198.51.100.88
 - .. few people likely to notice

More DNS: Intelligent indirection

- You stream video from three servers across the United States
- Three different IP addresses
- Smart DNS server looks at IP address of client...
- Does GeolP lookup...
- Selects closest server!
- Saves money/latency



More DNS: Summary

- We've looked at a lot of things DNS can do!
- Coarse server load spreading and resiliency (via multiple A records)
- Name-to-IPv6-Address mapping (via AAAA records)
- IP-Address-to-Name (reverse) mapping (via PTR records)
- Alias names (via CNAME record)
- Nice email addresses (via MX records)
- Site verification (via TXT records)
- General name-to-service mapping (via SRV records)
- DNS as an indirection layer
- .. and there are many more record types and DNS tricks!

Attributions

Minecraft PNG, Creative Commons 4.0 BY-NC

<http://pngimg.com/download/59250>

Blank US map borders.svg, Public Domain

https://commons.wikimedia.org/wiki/File:Blank_US_map_borders.svg

DNS

Availability, scalability, and performance

DNS: Availability, scalability, and performance

- DNS should be *highly available*
 - Expectation Internet would be difficult/impossible to use without it
- DNS should be *highly scalable*
 - Expectation that it would be used heavily
- DNS should be *highly performant*
 - Above expectations make this crucial!
- All sort of intertwined because there's basically one trick to accomplishing them
 - Add more servers!
 - Done in four different ways

DNS: Availability, scalability, and performance

- Name servers for different domains are independent
 - Just because [berkeley.edu](#) fails doesn't mean [mtholyoke.edu](#) fails
 - Just because [.edu](#) fails doesn't mean [.com](#) fails
- *Not just one server: servers per domain*
- Availability implication:
 - berkeley can fail; mtholyoke is fine
- Scalability implication:
 - No one server needs to know all info for all domains
- Performance implication:
 - Even if someone else's domain is going viral, yours is fine

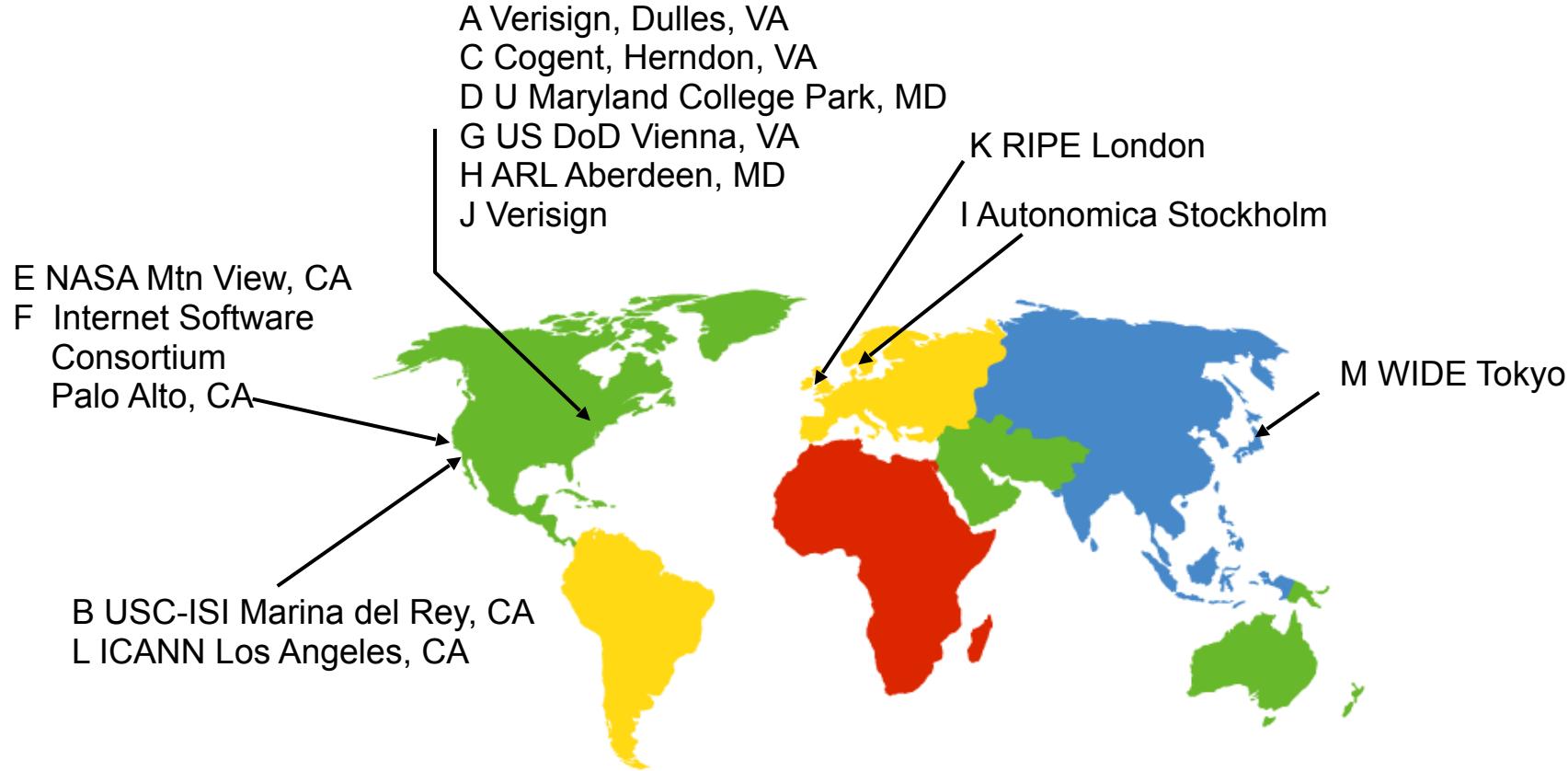
DNS: Availability, scalability, and performance

- Domains have at least two name servers
 - ns1.umass.edu
 - ns2.umass.edu
 - ns3.umass.edu
 - ns.mtholyoke.edu
- *Not just one server per domain: multiple servers per domain*
- Availability implication:
 - One server crashes, you've got others
- Scalability/performance implication
 - Queries get spread across; limits latency due to load

DNS: Looking at the Root Servers

- Already mentioned that there were 13 root name servers...
- a.root-servers.net 198.41.0.4
- b.root-servers.net 199.9.14.201
- c.root-servers.net 192.33.4.12
- d.root-servers.net 199.7.91.13
- e.root-servers.net 192.203.230.10
- f.root-servers.net 192.5.5.241
- g.root-servers.net 192.112.36.4
- h.root-servers.net 198.97.190.53
- i.root-servers.net 192.36.148.17
- j.root-servers.net 192.58.128.30
- k.root-servers.net 193.0.14.129
- l.root-servers.net 199.7.83.42
- m.root-servers.net 202.12.27.33

DNS: Looking at the Root Servers



DNS: Looking at the Root Servers

- 4.5 billion Internet users
- Are 13 root name servers enough?
- .. probably not
 - “j-root” alone got 66K queries per second in April 2018
 - 15us per query?
- So what's the trick?
- There are actually 162 duplicates of j-root in different places!
 - There's actually about a thousand total root servers as of 2020

DNS: J-root server locations

Amsterdam, NL	Cordoba, AR	Kigali City, RW	Oulu, FI	St. George, US
Ashburn, US	Dar Es Salaam, TZ	Klagenfurt, AT	Paris, FR	Stockholm, SE
Athens, GR	Denver, US	Kolkata, IN	Perth, AU	Sydney, AU
Atlanta, US	Des Moines, US	Kuala Lumpur, MY	Philadelphia, US	Tallinn, EE
Bangalore , IN	Dhaka, BD	Lagos, NG	Pirassununga, BR	Tampa, US
Bangkok, TH	Djibouti City, DJ	Leeds, GB	Plano, US	Tamuning, GU
Barcelona, ES	Eau Claire, US	Lisbon, PT	Portland, US	Tel Aviv, IL
Battle Creek, US	Edinburgh, GB	Ljubljana, SI	Prague, CZ	Tokyo, JP
Beijing, CN	Fayetteville, US	London , GB	Princes Town, TT	Turin, IT
Belgrade, RS	Frankfurt , DE	London, GB	Quezon City, PH	Vancouver, CA
Belo Horizonte, BR	Frankfurt Griesheim, DE	Los Angeles, US	Reno, US	Vilnius, LT
Berlin, DE	Frankfurt, DE	Luanda, AO	Reykjavik, IS	Warsaw, PL
Bettembourg, LU	Fremont, US	Madrid, ES	Richmond, US	Wellington, NZ
Bloomington, US	Geneva, CH	Male, MV	Riga, LV	Winnipeg, CA
Boston, US	Guarabira, BR	Melbourne, AU	Rio De Janeiro, BR	Yerevan, AM
Brasilia, BR	Gurgaon, IN	Miami, US	Rome, IT	Zagreb, HR
Bratislava, SK	Halifax, CA	Milan, IT	Saint Petersburg, RU	Zurich, CH
Brisbane, AU	Hong Kong, HK	Montgomery, US	Salzburg, AT	
Bucharest, RO	Honolulu, US	Moscow, RU	San Francisco, US	
Buenos Aires, AR	Istanbul, TR	Mumbai, IN	San Jose, CR	
Cajazeiras, BR	Jakarta, ID	Muscat, OM	San Jose, US	
Calgary, CA	Johannesburg, ZA	Nairobi, KE	San Juan, PR	
Cape Town, ZA	Juazeirinho, BR	New Castle, US	Santiago, CL	
Cebu City, PH	Kansas City, US	New Delhi, IN	Seattle, US	
Chicago, US	Kathmandu, NP	New York, US	Seoul, KR	
Cochabamba, BO	Kaunas, LT	Orlando, US	Singapore, SG	
Colombo, LK	Kiev, UA	Oslo, NO	Sofia, BG	

DNS: Looking at the Root Servers



DNS: E-root server locations

Accra, GH	Brussels, BE	Detroit, US	Istanbul, TR	Luanda, AO
Adelaide, AU	Bucharest, RO	Dhaka, BD	Jacksonville, US	Luxembourg City, LU
Amsterdam, NL	Budapest, HU	Djibouti, DJ	Jakarta, ID	Lyon, FR
Antananarivo, MG	Buenos Aires, AR	Doha, QA	Johannesburg, ZA	Macau, MO
Arica, CL	Buffalo, US	Dubai, AE	Johor Bahru, MY	Madrid, ES
Arusha, TZ	Burbank, US	Dublin, IE	Kampala, UG	Manama, BH
Ashburn, US	Calgary, CA	Durban, ZA	Kansas City, US	Manchester, GB
Athens, GR	Cape Town, ZA	Dusseldorf, DE	Kathmandu, NP	Manchester, UK
Atlanta, US	Castries, LC	Edinburgh, GB	Kigali, RW	Manila, PH
Auckland, NZ	Cebu, PH	Enfidha, TN	Kingston, JM	Maputo, MZ
Baghdad, IQ	Charlotte, US	Fortaleza, BR	Klagenfurt, AT	Mar Del Plata, AR
Baltimore, US	Chennai, IN	Frankfurt, DE	Kuala Lumpur, MY	Marseille, FR
Bangkok, TH	Chicago, US	Geneva, CH	Kuwait City, KW	McAllen, US
Banjul, GM	Chittagong, BD	Gothenburg, SE	Kyiv, UA	Medellin, CO
Barcelona, ES	Colombo, LK	Guayaquil, EC	La Paz, BO	Melbourne, AU
Beirut, LB	Columbus, US	Halifax, CA	Lagos, NG	Memphis, US
Belgrade, RS	Copenhagen, DK	Hamburg, DE	Las Vegas, US	Mexico City, MX
Berlin, DE	Cork, IE	Helsinki, FI	Lausanne, CH	Miami, US
Beverly, US	Curitiba, BR	Hong Kong, HK	Leeds, UK	Milan, IT
Blantyre, MW	Dakar, SN	Honolulu, US	Lima, PE	Minneapolis, US
Bogota, CO	Dallas, US	Houston, US	Lisbon, PT	Mombasa, KE
Boston, US	Dar Es Salaam, TZ	Hyderabad, IN	London, GB	Moncton, CA
Brisbane, AU	Denver, US	Indianapolis, US	Los Angeles, US	Monrovia, LR

DNS: E-root server locations

Montgomery, US	Phnom Penh, KH	Saint George, US	Taipei, TW	Zagreb, HR
Montreal, CA	Phoenix, US	Saint-Denis, RE	Tallahassee, US	Zurich, CH
Moscow, RU	Pittsburgh, US	Saldanha, ZA	Tallinn, EE	
Mountain View, US	Port Louis, MU	Salt Lake City, US	Tampa, US	
Mumbai, IN	Port Vila, VU	San Diego, US	Tampere, FI	
Munich, DE	Port au Prince, HT	San Francisco, US	Tegucigalpa, HN	
Muscat, OM	Port of Spain, TT	San Jose, CR	Tel Aviv, IL	
Nagpur, IN	Port-Au-Prince, HT	San Jose, US	Thessaloniki, GR	
Nairobi, KE	Portland, US	Santa Ana, US	Tokyo, JP	
Nashville, US	Porto Alegre, BR	Santiago, CL	Toronto, CA	
New Delhi, IN	Posadas, AR	Sao Paulo, BR	Tunis, TU	
New York, US	Prague, CZ	Saskatoon, CA	Turin, IT	
Newark, US	Queretaro, MX	Seattle, US	Ulaanbaatar, MN	
Norfolk, US	Quito, EC	Seoul, KR	Vancouver, CA	
Noumea, NC	Ramallah, PS	Seoul, ZA	Vienna, AT	
Omaha, US	Rene Mouawad, LB	Siegerland, DE	Vilnius, LT	
Osaka, JP	Reno, US	Singapore, SG	Warsaw, PL	
Oslo, NO	Reykjavik, IS	Sofia, BG	Washington, US	
Palo Alto, US	Richmond, US	St. Georges, GD	Wellington, NZ	
Panama City, PA	Riga, LV	St. Louis, US	Willemstad, CW	
Paris, FR	Rio De Janeiro, BR	Stockholm, SE	Windhoek, NA	
Perth, AU	Riyadh, SA	Sumbe, AO	Winnipeg, CA	
Philadelphia, US	Rome, IT	Sydney, AU	Yerevan, AM	
	Rosario, AR			
	Sacramento, US			

DNS: Looking at the Root Servers

- There are duplicates of the a,b,c,d,... root servers in different places!
- Each “X-root” duplicate has the same IP address!
- Same address advertised through BGP at each duplicate
- BGP will always just deliver to the “closest” one (subject to BGP policy)
 - This is called *anycast*
 - Delivers to *any* one of the destinations
 - Contrast with *unicast* (our focus until now -- delivers to *single* dest.)
 - .. it “just works” — no change to routing is needed

DNS: Looking at the Root Servers

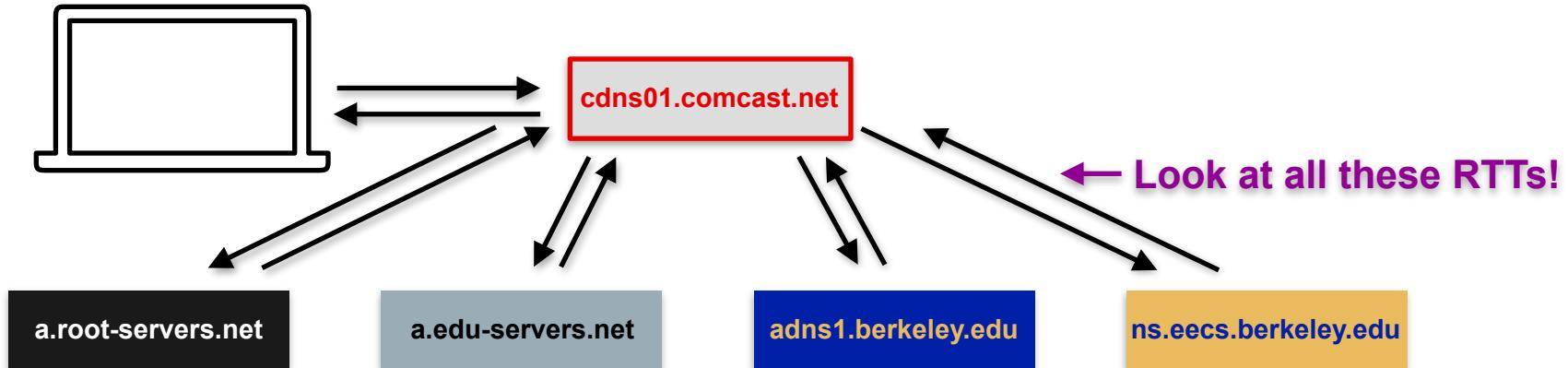
- Anycast “just works”...
- But does it have any drawbacks?
 - Like multihomed prefixes, anycast prefixes can't be aggregated
 - Can't just use it for everything!
 - But maybe really important things like DNS!
 - Doesn't always work well with TCP...
 - Changes to BGP or client mobility may mean your connection suddenly shifts to a different server that knows nothing about it!
 - Luckily... DNS is usually UDP / stateless!

DNS: Looking at the Root Servers

- Availability implication:
 - If network partitioned, BGP will automatically route to reachable server
- Scalability implication:
 - Further spreads load across servers
- Performance implication:
 - Reduce RTT to server because BGP shouldn't pick terrible path

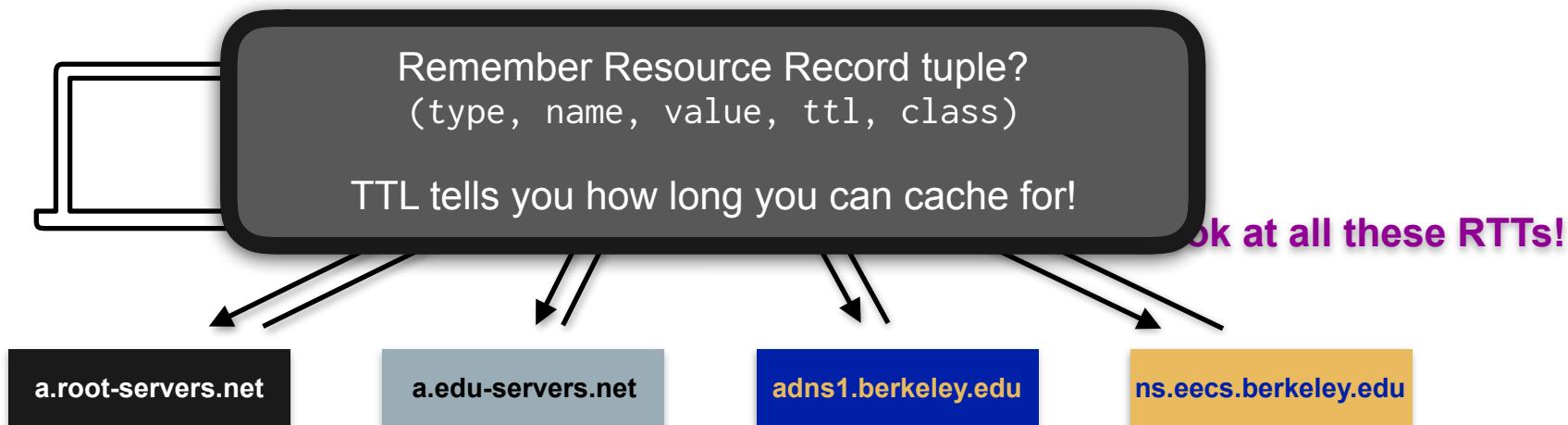
DNS: Availability, scalability, and performance

- Remember this?
- Query procedure not very fast! Many RTTs!
- How can we speed it up?
 - Caching!



DNS: Availability, scalability, and performance

- Servers on bottom never see anything to cache (only their own records)
- But resolving server can cache results of resolution (and share with other hosts)
- Can actually insert *caching servers* just about anywhere!
 - Don't even do iterative lookups -- just forward request and cache reply
- .. and the host can cache too, of course!



DNS: Looking at the Root Servers

- Availability implication:
 - Even during a hiccup (e.g., while BGP converging on new path), common queries can be served out of cache
- Scalability implication:
 - Reduces load on real name servers
- Performance implication:
 - No need to do multi-RTT lookup process for common entries
 - Can cache close to clients to reduce RTT

DNS: Availability, scalability, and performance

- Summing up...
- Four ways in which we add more servers:
 - Different domains have different DNS servers
 - Servers are replicated; minimum two for each domain
 - Each root server is replicated *again* using anycast
 - Add resolving/forwarding/caching servers that cache results
- With more servers:
 - It's more likely one is close to you (faster RTT)
 - They're less likely to be heavily loaded (slow)
 - It's less likely that the only one with the data you want is down
 - The amount of data that needs to be stored on each one is lessened

DNS

A little DNS skepticism

Inquiry and doubt are essential checks against deception, self-deception, and error.
— Richard Carrier
(maybe)

DNS Skepticism

- DNS is usually presented as vital, integral aspect of the Internet
 - It's in every book on the Internet
 - Every network course covers it
 - It has been “.. an essential component of the functionality of the Internet since 1985.” (Wikipedia)
-
- Did we name the right things?
 - Does DNS actually work well?
 - Does DNS put your privacy at risk?
 - Is DNS even important?



Do we name the right things?

- telnet was about logging in to a remote host
 - What was the right thing to name?
 - Hosts!

Do we name the right things?

- FTP is about transferring files
 - File already had names on machines; already had name for the file part
 - “/existing/file/name on hostname” seems reasonable
- Is it ideal?
 - What if you move the file to another machine?
 - What if you want to replicate the file on many hosts so it’s always available? Do you even care *which* host it’s stored at?
- Better solutions? What is the right thing to name?
 - See: Information-Centric Networking, Content-Centric Networking, and **Named Data Networking**

Do we name the right things?

- Email is about communicating between *people*
 - Remember, many users used to share the same computer!
 - Early email was between users on a single computer
 - Just needed to mail <username>
 - Easy to see extension of this to mail <username> on <remote machine>
- Is it ideal?
 - What if the user moves to a different machine?
 - What if you have too many users for single machine to handle (gmail)?
- Better solutions?
 - Maybe unique names for *people* independent of providers?
 - Must map to whoever your current mail provider is
 - Want some sort of accountability (for spammers)?
 - Anonymity / deniability?
 - May be a tricky design challenge?

Do we name the right things?

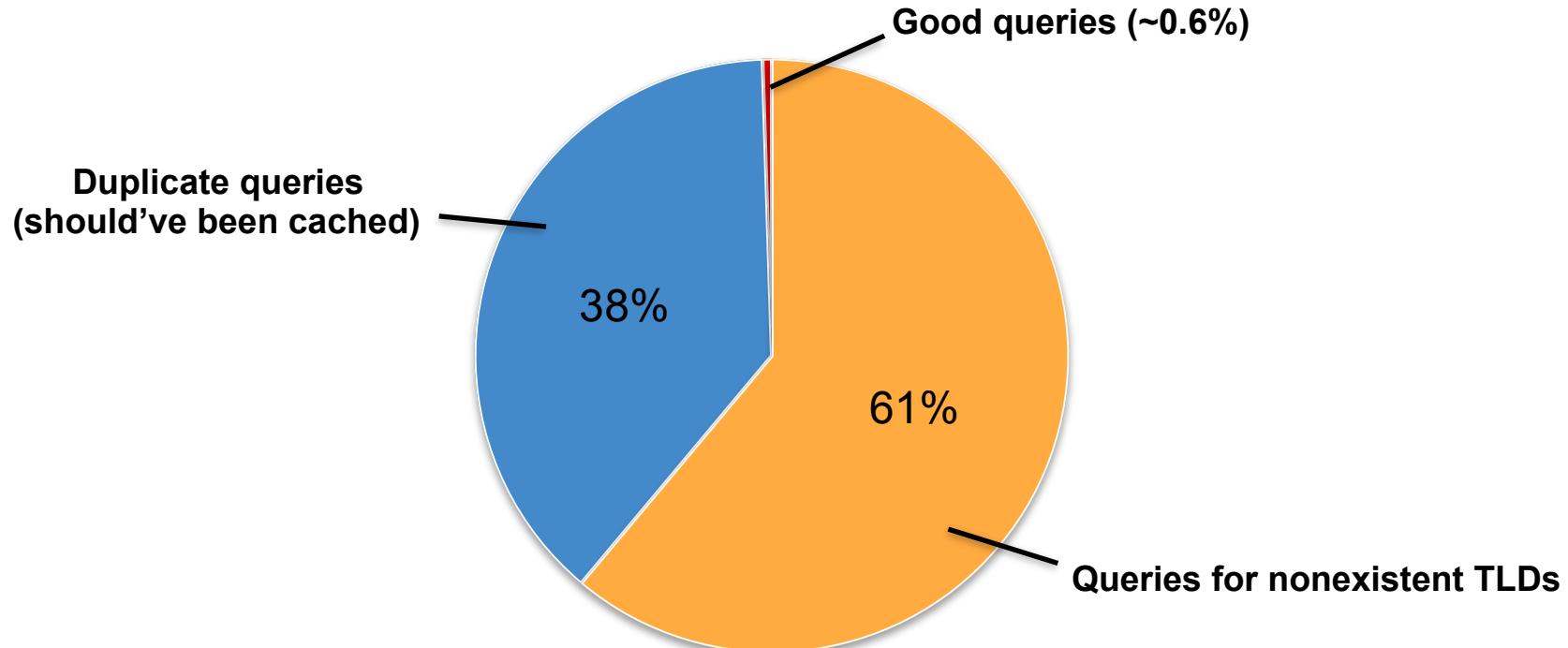
- What about the web?
- URLs basically are hostname plus filename
 - Make sense — early web was (underneath) much like simple file transfer
 - .. but with an important change in how it was used! (hypertext)
- With that in mind, same idea as FTP might apply...
 - Want names driven by content not by which machine they're on!
- But is the web just about transferring files?
 - Is it more about accessing services? (your banking, Facebook, ...)
 - Modern services certainly aren't tied to a specific host!
 - Should we be naming services directly?

Do we name the right things?

- Naming hosts made *perfect* sense for telnet
- For files, people, and services... hosts are less good of a fit
- If we stepped back, could we design better schemes and mechanisms?
 - e.g., A highly available, scalable, performant system for addressing *people* no matter where they are or who their email provider is?
- *My gut says yes.*

Does DNS work well?

- Analyzed 5.7B queries in 24 hours at J-root



[Data from Allman 2019]

Does DNS work well?

- Analyzed 5.7B queries in 24 hours at J-root

Good queries (~0.6%)

Caching isn't super effective for bad TLDs!

Lots of other things don't seem to get cached either!

Root NS infrastructure dedicated entirely to junk.

Moral

If you build a highly resilient and scalable system,
things can be going wrong without you noticing it!

TLDs

Is DNS a privacy problem?

- If you browse sites using https, the data is encrypted
- .. but the DNS lookups aren't
- Nobody knows what you're reading or writing, but everyone knows where you're doing it
- Trivially easy for ISPs to just log requests at DNS server
- They don't even need to "snoop" the traffic -- you send it right to them!

Is DNS a privacy problem?

- Currently two competing projects to make DNS more private:
 - DNS over TLS
 - DNS over HTTPS
- It's causing a fair amount of debate
- Current versions of Firefox are using DNS over HTTPS by default
 - No longer uses your own ISP
 - Uses third party for DNS resolution — Cloudflare
 - Do you know/trust them, or are you just securely giving your DNS info to an untrusted third party?

Is DNS a privacy problem?

- Currently two competing projects to make DNS more private:
 - DNS over TLS
 - DNS over HTTPS



Nick Sullivan @grittygrease · Oct 19, 2018

DNS Queries over HTTPS (DoH) is now RFC 8484. This is a big step forward for DNS security. rfc-editor.org/rfc/rfc8484.txt

15

356

691



Paul Vixie @paulvixie · Oct 20, 2018

Rfc 8484 is a cluster duck for internet security. Sorry to rain on your parade. The inmates have taken over the asylum.

3

25

76

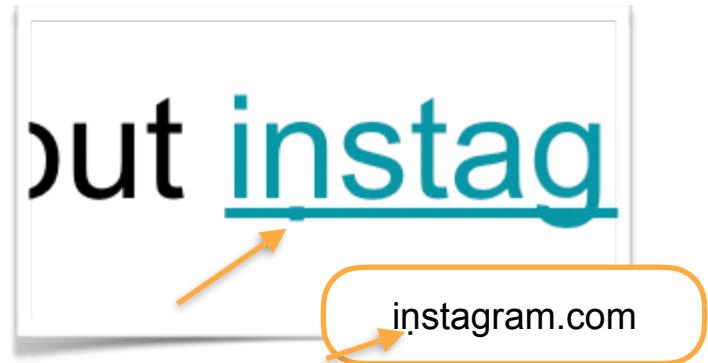


Who needs names anyway?

- When is the last time you typed a hostname?
- Get the weather then (1992):
 - telnet rainmaker.wunderground.com
- Get the weather now:
 - Open Weather Underground app
- Finding HTTP protocol information then (1989):
 - Type <http://info.cern.ch/hypertext/WWW/> into browser
 - Take links for Technical, Protocols, HyperText Transfer Protocol
- Finding HTTP protocol information now:
 - Put “http standard” in search bar of browser

Who needs names anyway?

- Is human-readability useful / secure?
- Wait, is it [examplesite.com](#) or [example-site.com](#)?
- Is it safe to take a link to [wellsfargo.com](#)?
- What about [instagram.com](#)?
- You're probably better off searching than typing an address!



Is DNS actually important?

- DNS maps from hostnames to addresses (mostly)
 - But *people*, *services*, and *data* are in many ways more important than particular hosts today
- Human-supplied names...
 - Matter less today due to web search, apps, etc.
- Human-readable names...
 - Have little guarantee that you're reading what you think you're reading
- .. I'm not convinced original purpose of DNS stands the test of time!

Is DNS actually important?

- But, DNS also provides...
- Load balancing...
 - One name can map to multiple addresses to spread traffic
- An indirection layer...
 - Can keep the same name but map it to a different address
 - Useful if you move your server, switch ISPs, etc.
 - Can do the mapping dynamically — inspect source IP of DNS request, and direct to a server you think is *near* the client!
- .. very important!
 - .. but have nothing to do with the original driver of human-readability!
 - .. could you design a better system if you gave up human-readability?

CS168

Lecture 19

Today in Networking



- 22nd anniversary of Mozilla's official launch (1998)
- The first web browser to really take off was *Mosaic*
 - Developed at National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana–Champaign
 - Funding from “Gore Bill”
- One of its developers (Marc Andreessen) went on to found Netscape
 - Internally, Netscape’s browser (Netscape Navigator) was called “Mozilla”
 - This browser *totally dominated* the web for a crucial period
- In 1998, Mozilla the organization released the browser code under an open license
 - .. this eventually evolved into Firefox
 - .. and all the other things the Mozilla Foundation does for the Internet!

The Web

Where are we?

- Before the break, I said we were starting to look at *user-facing* things
- Started with DNS, which (at least initially) provided a user-facing system for interacting with the network: *names* instead of addresses
- Today:
 - The web — a game changing user-facing killer app

The Web

- Abbreviated history and motivation
- The basics
 - HTML, clients, servers, URLs
 - Basic HTTP
- Availability, scalability, and performance
 - Caching
 - Content Delivery Networks
 - TCP and HTTP
- Back to basics
 - Statelessness

The Web: Abbr. Hist.

The Web: Very abbreviated history

- In 1989, Tim Berners-Lee (then a software engineer at CERN) saw a problem
 - Lots of information
 - Information being added to and *changed* all the time
 - People come and go
- Information gets lost
 - It's often recorded — somewhere!
- CERN had a documentation system — CERNDOC
 - Hierarchical
 - Frustrating — information is not always hierarchical!
- Pitched a solution — “Information Management: A Proposal”

The Web: Very abbreviated history

- In 1989, Tim Berners-Lee (then a software engineer at CERN) saw a problem
 - Lots of information
 - Information being added to and *changed* all the time
 - People come and go
- Information gets lost
 - It's often recorded — somewhere!
- CERN had a documentation system — CERNDODC
 - Hierarchical
 - Frustrating — information is not always hierarchical!
- Pitched a solution — “Information Management: A Proposal”

The actual observed working structure of the organisation is a multiply connected "web" whose interconnections evolve with time.

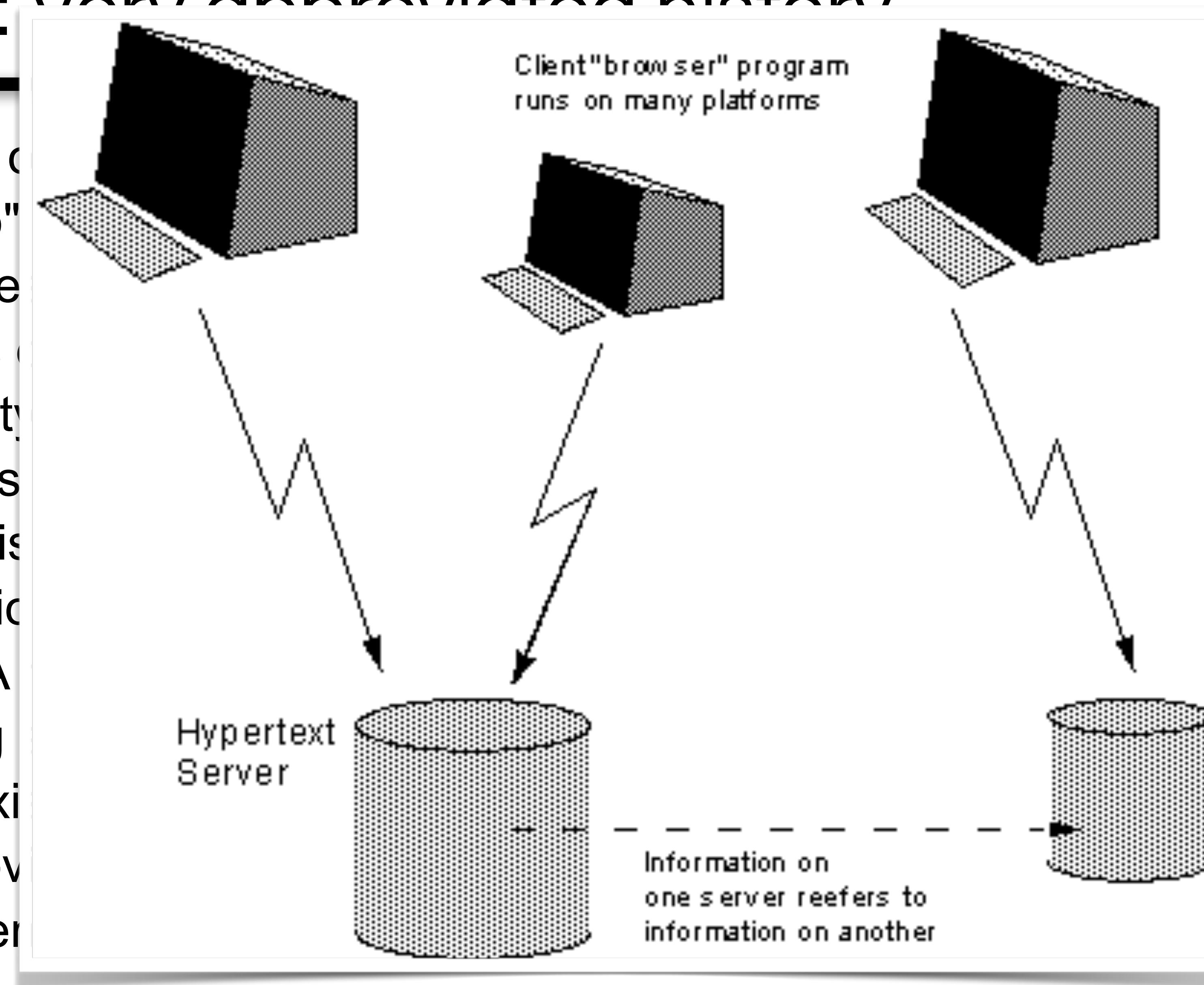
— From "Information Management: A Proposal"

The Web: Very abbreviated history

- The method of storage must not place its own restraints on the information
 - .. a "web" of notes with links ... is far more useful than a fixed hierarchical system.
- Remote access across networks
 - CERN is distributed, and access from remote machines is essential.
- Heterogeneity
 - Access is required to the same data from different types of system
- Non-Centralisation
 - Information systems start small and grow. They also start isolated and then merge. A new system must allow existing systems to be linked together without requiring any central control or coordination.
- Access to existing data
 - If we provide access to existing databases as though they were in hypertext form, the system will get off the ground quicker.

The Web: Very abbreviated history

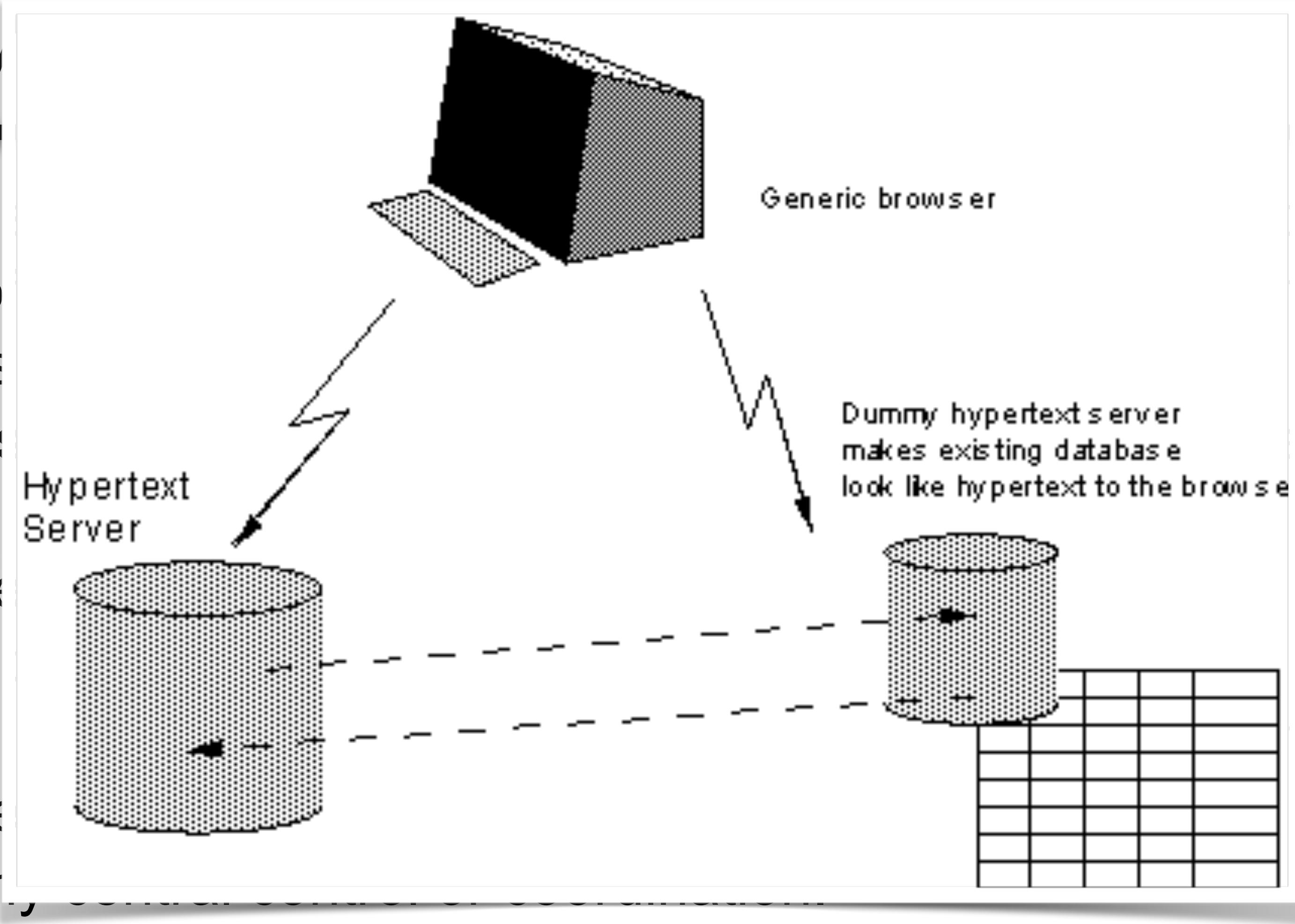
- The method of hypertext
 - .. a "web"
- Remote access
 - CERN is central
- Heterogeneity
 - Access is via a client program
- Non-Centralised
 - Information is stored on many servers and can merge. A user can move between them without requiring a central server
- Access to existing information
 - If we provide a way for users to find what they want, the system will grow



nation
archical system.
tial.
m
d and then
gether without
n hypertext form,

The Web: V

- The method of
 - .. a "web" of
- Remote access
 - CERN is dis
- Heterogeneity
 - Access is re
- Non-Centralisa
 - Information
 - merge. A ne
 - requiring any
- Access to existing data
 - If we provide access to existing databases as though they were in hypertext form, the system will get off the ground quicker.



rmation
ierarchical system.
ential.
em
ed and then
together without

The Web: Why was it so successful?

- It wasn't trying to force anything
 - Didn't need to structure data in a particular way
 - Didn't need to store data in a particular format
 - Didn't need to use a particular computer/database system
 - Didn't need to abandon existing (working) systems
- Had networks in mind from the beginning!
- Provided *integrated* interface to scattered information
- Was designed to be a *practical solution* to a *specific problem*
- They didn't try to charge for the technology
- .. *not all of this was new, but this was where they first all came together*

Every good work of software
starts by scratching a
developer's personal itch.

— Eric Raymond

The Web: Why was it so successful?

- What made it successful in the beginning is what makes it successful now!
 - It gives a lot of leeway for how websites work (didn't over-specify)
 - Not tied to any one underlying system
 - No central authority — you can just add your own server/content
 - The ability to quickly navigate information from different sources

The Web: Why study it?

- The early web was mind-numbingly simple *technically*
- And was not cutting edge *intellectually* in terms of information representation
- But it was/is a successful and practical system that changed the world!

- No professor could design something so simple
 - Enough functionality to be effective
 - Not enough to prove her cleverness

— Professor Scott Shenker

- We couldn't possibly have a class about the Internet that didn't look at it!

The Web: Basics

The Web: Basic requirements

- Something to represent content with links: **HTML**
- Client program to access/navigate/display content (e.g. HTML): **Web browser**
- A way to reference content: **URLs**
 - It's how you link/embed content to/in other content across a network
 - First general “handle” for arbitrary Internet content
 - Not just naming a host/processes (address/port)
- Something to host content: **Web servers**
- A protocol to get content from server to client: **HTTP**
 - Turns web URLs into TCP connections

Web basics

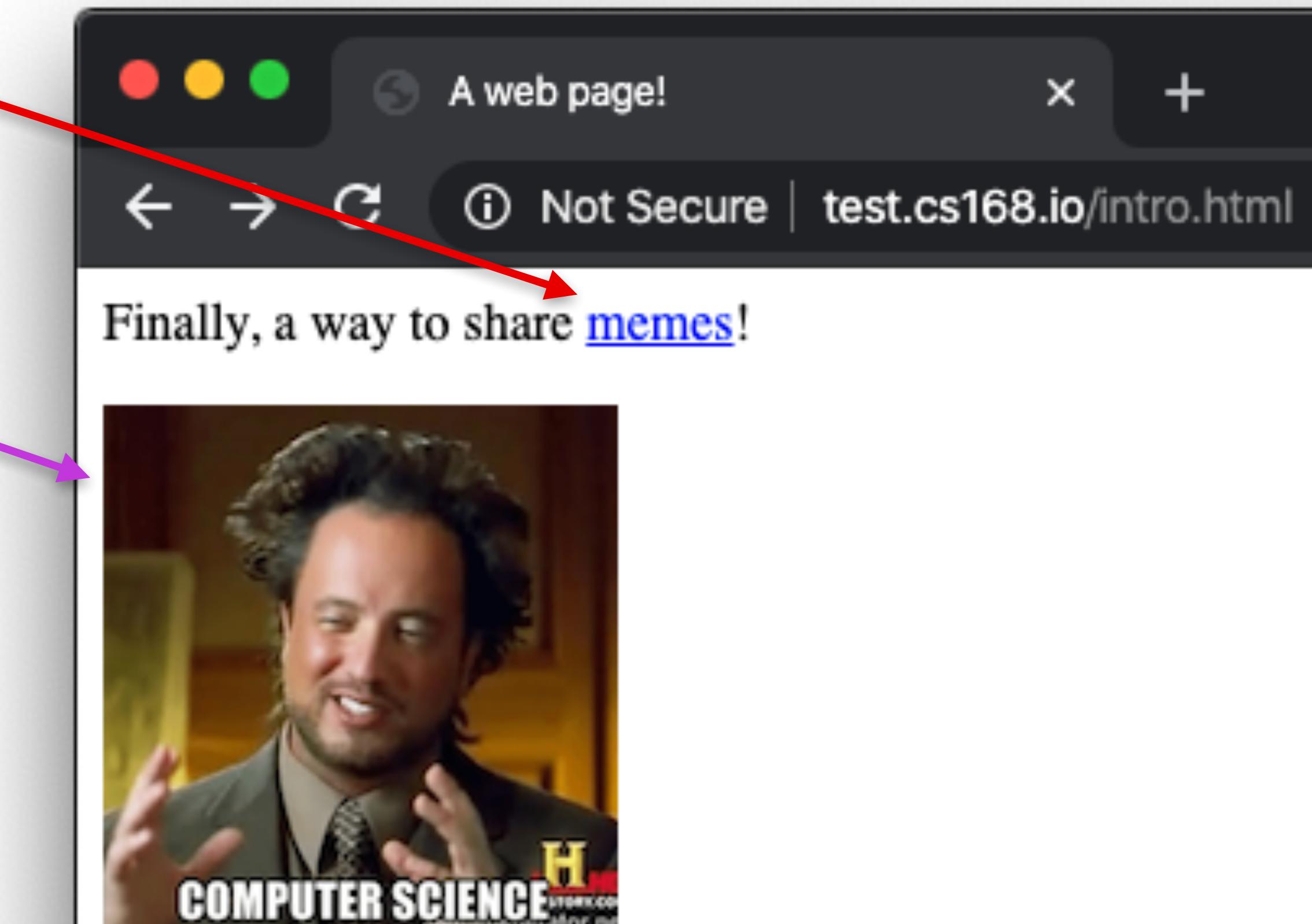
- HTML: HyperText Markup Language - Represent content with links
- Browser: Access/navigate/display content
- Provide *integrated* interface to scattered information

Embed another resource

Link to another resource

```
<html>
  <head>
    <title>A web page!</title>
  </head>

  <body>
    <p>Finally, a way to share
      <a href="about_memes.html">memes</a>!
    </p>
    
  </body>
</html>
```



Web basics: URL Syntax

scheme : //host[:port] /path /resource

<i>scheme</i>	Typically a protocol: http, ftp, https, smtp, rtsp, etc.
<i>host</i>	DNS hostname or IP address
<i>port</i>	Defaults to protocol's standard port e.g. http: 80 https: 443
<i>path</i>	Traditionally reflecting file system
<i>resource</i>	Identifies the desired resource (traditionally a file)
	Can also extend to program executions:  <code>http://us.f413.mail.yahoo.com/ym>ShowLetter? box=%40B%40Bulk&MsgId=2604_1744106_29699_1123_1261_0_28917_3552_128995 7100&Search=&Nhead=f&YY=31454&order=down&sort=date&pos=0&view=a&head=b</code>

Web basics: URL Syntax

scheme://host[:port]/path/resource[?query][#fragment]

<i>scheme</i>	Typically a protocol: http, ftp, https, smtp, rtsp, etc.
<i>host</i>	DNS hostname or IP address
<i>port</i>	Defaults to protocol's standard port e.g. http: 80 https: 443
<i>path</i>	Traditionally reflecting file system
<i>resource</i>	Identifies the desired resource (traditionally a file)
<i>query</i>	e.g., search terms if resource is search program
<i>fragment</i>	Sub-part of resource (e.g., paragraph on web page)

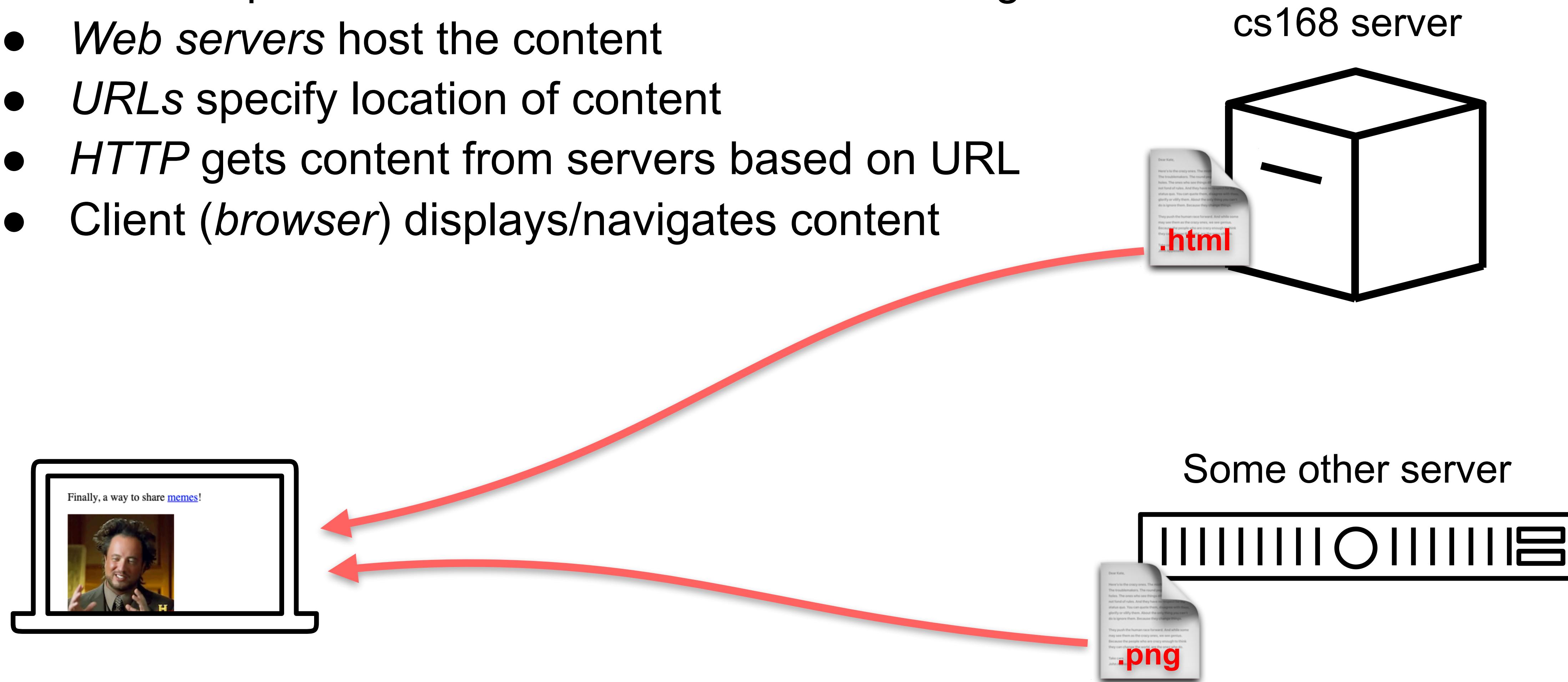
Questions?

Flashback: Do we name the right things?

- URLs basically are hostname plus filename
- Is it ideal?
 - What if you move the file to another machine?
 - What if you want to replicate the file on many hosts so it's always available? Do you even care *which* host it's stored at?
- Should we be naming the *content* directly, rather than server+filename?
 - See: Information-Centric Networking, Content-Centric Networking, and **Named Data Networking**
- Is the web more about accessing services? (your banking, Facebook, ...)
 - Modern services certainly aren't tied to a specific host!
 - And a lot is *dynamic* — ***you're not fetching a file, you're running a program***
 - Should we be naming services directly?

Web basics: putting it all together

- *HTML* represents content with links/embeddings
- *Web servers* host the content
- *URLs* specify location of content
- *HTTP* gets content from servers based on URL
- Client (*browser*) displays/navigates content



Questions?

The Web: Basic requirements

- Something to represent content with links: **HTML**
 - Client program to access/navigate/display content (e.g. HTML): **Web browser**
 - A way to reference content: **URLs**
 - It's how you link/embed content to/in other content across a network
 - First general “handle” for arbitrary Internet content
 - Not just naming a host/processes (address/port)
 - Something to host content: **Web servers**
- A protocol to get content from server to client: **HTTP**
 - Turns web URLs into TCP connections

Basic HTTP

HyperText Transfer Protocol (HTTP)

- Focusing our discussion on common/current versions of HTTP:
 - HTTP 1.0 (1996) and HTTP 1.1 (1997)
 - These are (significant) outgrowth of original “HTTP 0.9”
- HTTP 2 published in 2015
 - Largely based on work by Google
 - As of 2020, 44% of websites use it
 - Significant departure; largely performance optimizations
- HTTP 3 forthcoming standard
 - Largely based on work by Google
 - As of 2020, 5% of websites use it (more or less Google and Facebook?)
 - Significant departure; largely performance optimizations

HyperText Transfer Protocol (HTTP)

- The basics of HTTP:

- Client-server architecture
- Client connects to server on well-known TCP port 80
- Client issues request
- Server issues reply
- Protocol is “stateless”

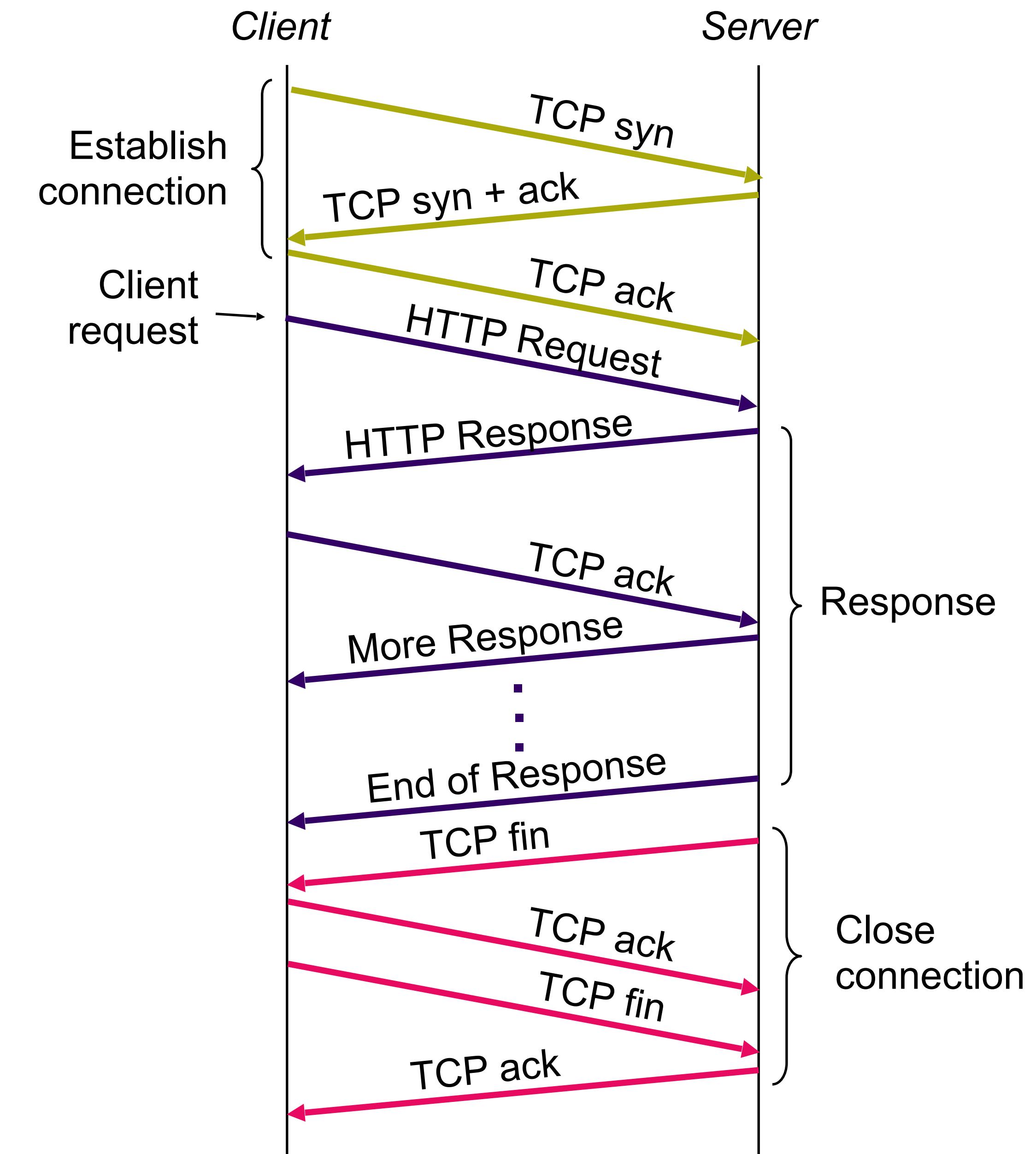
You should basically understand what this is saying.

(We'll go into details, though.)

We'll come back to this.

Inside an HTTP exchange

- (Simple HTTP 1.0 “GET” request)
- Client creates TCP connection (port 80)
- Client sends **request**
- Server sends **response** packets
- Client ACKs them
 - Note: There may be unshown ACKs
- Server closes connection



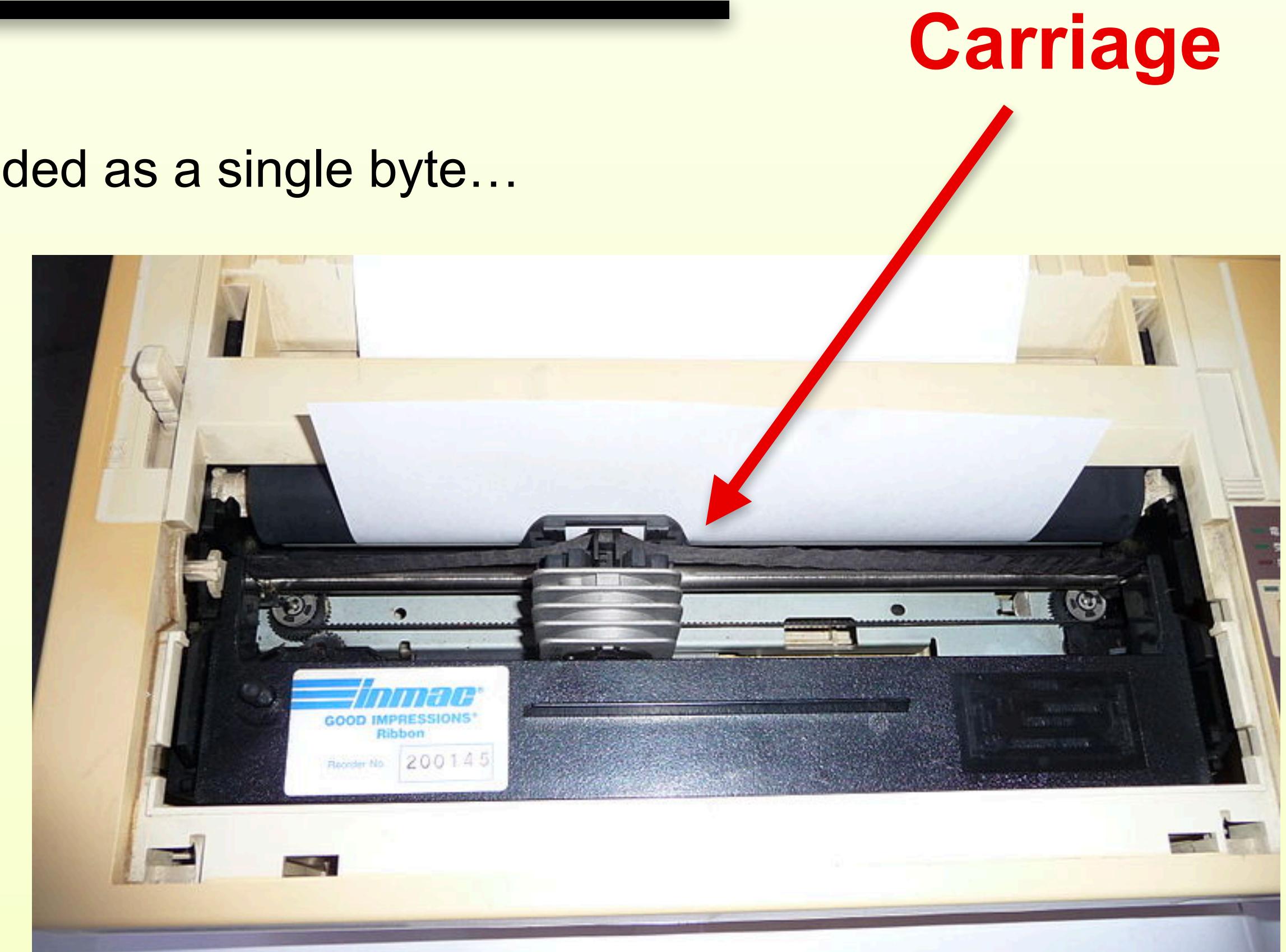
Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
 - Lines separated with CR LF

Request for
<http://www.someschool.edu/main/about.html>

Sidenote: CR and LF

- In common text encodings (ASCII, Latin 1, UTF-8)...
 - Common English letters and punctuation are encoded as a single byte...
 - 65 is “A”, 97 is “a”, 35 is “#”, etc.
 - 0 through 31 are *control characters*
 - 8 is backspace
 - 4 is “end of transmission”
 - 10 is **line feed (LF)**
 - 13 is **carriage return (CR)**
- You’re probably familiar with “\n” — newline
 - On Unix-like systems, this is really LF — does both
 - On Windows, means CR LF
 - Open a file in text mode in Python (and other languages), and it does translation
 - If you ever open up a file and every line ends with “^M” — those are the carriage returns — this was a Windows file and you’re on a Unix-like machine



Carriage

Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
 - Lines separated with CR LF
- Request line:
 - Method - “action” to perform. GET/HEAD/POST/...
 - Resource - e.g., which thing to fetch
 - Protocol version - either 1.0 or 1.1
- Request headers:
 - Provide additional information or modify request
 - Some required; many optional
- Blank line
- Body:
 - Optional data
 - Used when submitting data (e.g., a form via POST)

Request for

<http://www.someschool.edu/main/about.html>

GET /main/about.html HTTP/1.1

Host: www.someschool.edu

User-agent: Mozilla/4.0

Connection: close

Accept-language: en

(blank line)

(body, if there is one)

Inside an HTTP 1.0/1.1 request

- “Plain text” (“Latin 1” encoding)
 - Lines separated with CR LF
- Request line:
 - Method - “action” to perform. GET/HEAD/POST/...
 - Resource - e.g., which thing to fetch
 - Protocol version - either 1.0 or 1.1
- Request headers:
 - Provide additional information or modify request
 - Some required; many optional
- Blank line
- Body:
 - Optional data
 - Used when submitting data (e.g., a form via POST)

Request for
<http://www.someschool.edu/main/about.html>

GET /main/about.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: en
(blank line)
(body, if there is one)

http://www.someschool.edu/main/about.html

Where to connect
(and Host: header)

Request Line

Inside an HTTP 1.0/1.1 response

- Status line:
 - Protocol version - either 1.0 or 1.1
 - Status code - 2xx=success, 4xx=error, ...
 - Reason - Human-readable
 - Response headers:
 - Provide additional information
 - Blank line
 - Body:
 - Optional data — but very common!
 - e.g., it's the content of about.html!
- Request for
<http://www.someschool.edu/main/about.html>
- ```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 2006 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2006 ...
Content-Length: 6821
Content-Type: text/html
(blank line)
<html>
<head>
<title>About Some School</title>
</head>
...

```

# Questions?

# HTTP Methods (Common)

---

- GET
  - The classic!
  - Request to download some object
  - No body on request, body of reply is the requested object
- POST
  - Send data from client to server
  - e.g., submitting a web form, adding item to shopping cart, etc.
  - Body on request and often on response too
- HEAD
  - Basically same as a GET except you *don't want the body* (just headers)
  - Used to, e.g., see if something exists, when it was modified, etc.

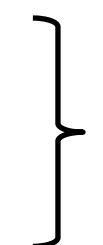
# HTTP Response status codes (selected)

---

- 1xx - Informational
  - None defined in HTTP 1.0, only a couple in HTTP 1.1
- 2xx - Successful
  - ★ ● 200: OK (e.g., here's the web page you requested...)
- 3xx - Redirection
  - 301: Moved Permanently (Location header tells you new URL)
  - ★ ● 304: Not Modified (not really a redirection; we'll revisit this one)
- 4xx - Client Error
  - 400: Bad Request (catchall for when client messes up, e.g., didn't include a required header)
  - 401: Unauthorized (the resource requires a password or something)
  - ★ ● 404: Not Found (the bane of the early 2000s web, though funny/creative ones helped)
- 5xx - Server Error
  - 500: Internal Server Error (catchall for when server configuration goes wrong)

# HyperText Transfer Protocol (HTTP)

---

- The basics of HTTP:
  - Client-server architecture
  - Client connects to server on well-known TCP port 80
  - Client issues request
  - Server issues reply
  - Protocol is “stateless”
- 

We'll come back to this.

Questions?

# Where do we go from here?

---

- We've described the basics...
  - .. what else do you want?!
- Users
  - Fast! (Performant)
  - Highly available!
  - .. nobody likes a slow or broken site!
- Content provider
  - Fast and highly available (make users happy!)
  - Scalable (stay fast and highly available even with lots of users/content)

Do these goals sound really familiar?

.. they're basically the same as DNS!

Solve them using same ideas:  
*replication and caching!*

Plus: Make up for some TCP issues...

# HTTP

Availability, scalability, and performance

# HTTP: Availability, scalability, and performance

---

- Like with DNS, these topics are somewhat intertwined!
- We'll discuss three things here:
  - Caching
  - Content Delivery Networks (CDNs)
  - Interplay of HTTP and TCP

# Web Caching

# HTTP caching: Why does caching work?

---

- *Why* does caching work?
  - Exploits *locality of reference* AKA *principle of locality*
    - *Spatial locality* — If something is accessed, something near it will also probably be accessed
    - *Temporal locality* — If something is accessed, it'll probably be accessed again soon
  - Both were a factor if you took CS61C
  - One is much more relevant to web caching

# HTTP caching: How well does caching work?

---

- *How well* does caching work?

- Very well up to a point...
- .. file popularity has high peak but long tail
  - Large overlap in highly popular content
  - But many unique requests
  - .. common to many types of cache
- In the real world...
  - Content is increasingly dynamic (personalized feeds, many updates)
- But there's still a lot of static content worth caching
  - Images, CSS stylesheets, JavaScript libraries, ...

Everyone downloads the same viral memes...

.. but everyone has their own weird interests.

# HTTP caching: How does caching work?

---

- *How does caching work in HTTP?*
- The key is in the headers...
- Response headers:
  - Cache-Control
  - Expires

# HTTP caching: the Cache-Control header

---

- Cache-Control header used for lots of cache-related things
- Used for both requests and responses
- Most important use is for server (response) to specify max-age
  - It's just a TTL in seconds — how long response can be cached
  - Cache-Control: max-age=<seconds>

# HTTP caching: the Expires header

---

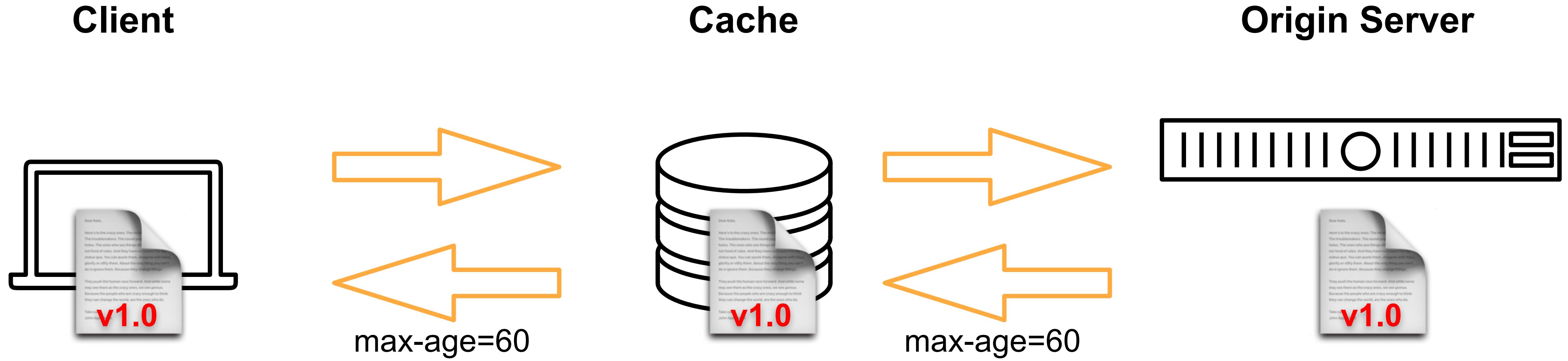
- Cache-Control is only available in HTTP 1.1
- HTTP 1.0 uses Expires response header
  - It's just a TTL in absolute time — when cached response becomes invalid
  - Expires: Thu, 31 Dec 2037 23:55:55 GMT
- Servers often send both Cache-Control: max-age and Expires

# HTTP caching: How does caching work?

---

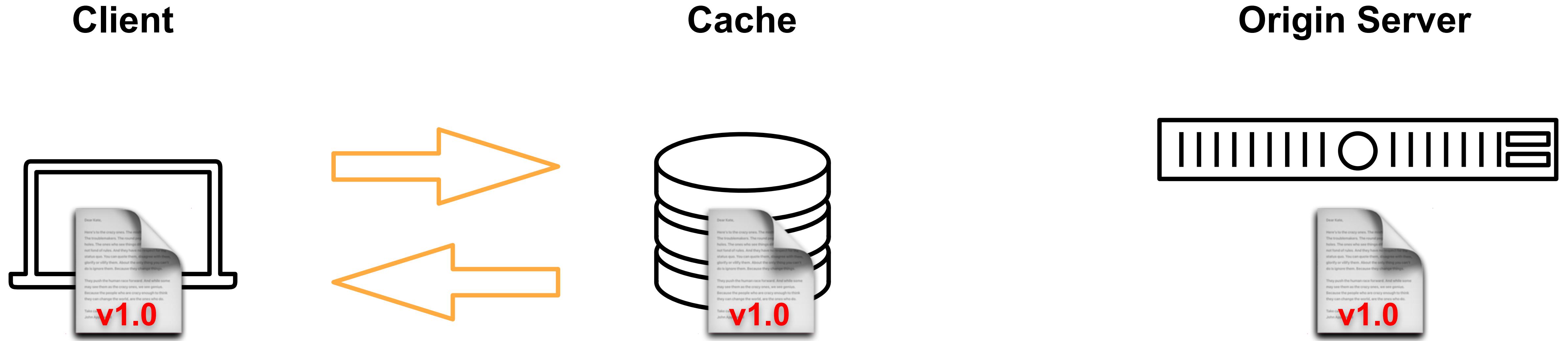
- *How does caching work in HTTP?*
- The key is in the headers...
- Response headers providing TTLs
  - Cache-Control: max-age (HTTP 1.1)
  - Expires (HTTP 1.0)
- But TTLs aren't always good enough!
  - .. server doesn't necessarily *really* know when content will be updated
  - .. clients need a way to force skipping of caches!
    - Cache-Control: no-cache and Pragma: no-cache request headers

# HTTP caching: How does caching work?



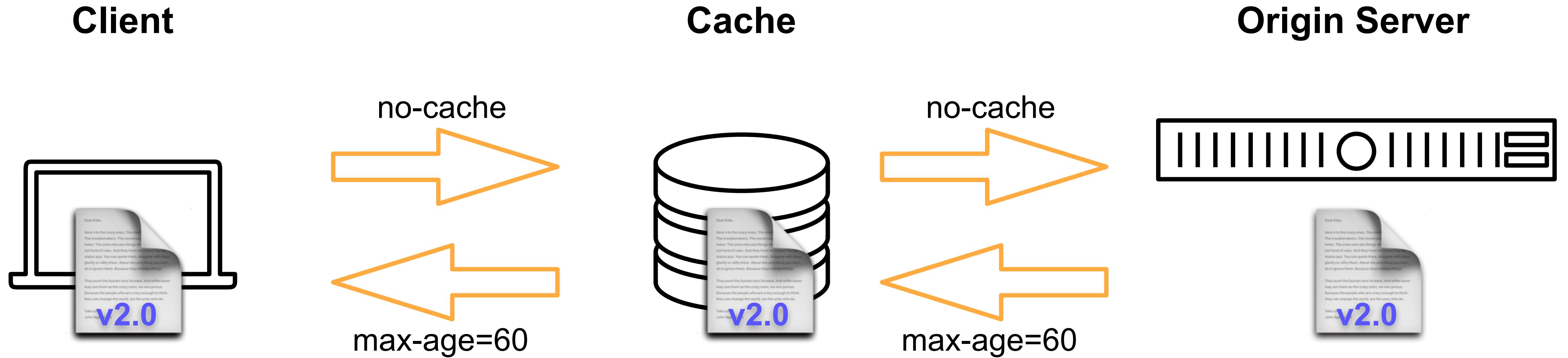
Client requests document; cached for one minute at t=0

# HTTP caching: How does caching work?



Client requests document; cached for one minute at t=0  
Document updated on server at t=1 — refresh would be stale until t=60!

# HTTP caching: How does caching work?



Client requests document; cached for one minute at t=0

Document updated on server at t=1 — refresh would be stale until t=60!

User does “hard refresh” at t=10 (shift-click refresh in browser)

What if document hadn't been updated? We just transferred it again for nothing!  
v1.0 was already in the caches!

# HTTP caching: How does caching work?

---

- Request header If-Modified-Since: <date>
  - If resource **has** changed since <date>:
    - Respond with latest version
  - If resource **has not** changed:
    - Respond with 304 - Not Modified
      - Includes headers
      - But not the body
    - .. lets you know you're up-to-date, but doesn't waste bandwidth

# HTTP caching: Typical caching interaction

---

- Client issues request for resource
- If resource in browser cache:
  - If cached version not expired ( $\text{TTL} > 0$ )
    - Assumed to be current — use version in browser cache
  - Else, cached version is expired
    - Send request using `If-Modified-Since: <date of cached version>`
    - If server's version is newer:
      - Respond with new version (200 response)
    - If server's version has same date:
      - Respond with Not Modified (304 response)
- Else, resource not in browser cache
  - Send request to server (with no `If-Modified-Since`)

# HTTP caching: Typical caching interaction

---

- Client issues request for resource
- If resource in browser cache:
  - If cached version not expired ( $\text{TTL} > 0$ )
    - Assumed to be current — use version in browser cache
  - Else, cached version is expired
    - Send request using `If-Modified-Since: <date of cached version>`
    - If server's version is newer:
      - Respond with new version (200 response)
    - If server's version has same date:
      - Respond with Not Modified (304 response)
- Else, resource not in browser cache
  - Send request to server (with no `If-Modifi`

What if server's version is **older**?

# HTTP caching: Typical caching interaction

- Client issues request for resource
- If resource in browser cache:
  - If cached version not expired
    - Assumed to be current — use version in browser cache
    - Else, cached version is expired
      - Send request using `If-Modified-Since: <date of cached version>`
      - If server's version is newer:
        - Respond with new version (200 response)
        - If server's version has same date:
          - Respond with Not Modified (304 response)
    - Else, resource not in browser cache
      - Send request to server (with no `If-Modified-Since`)

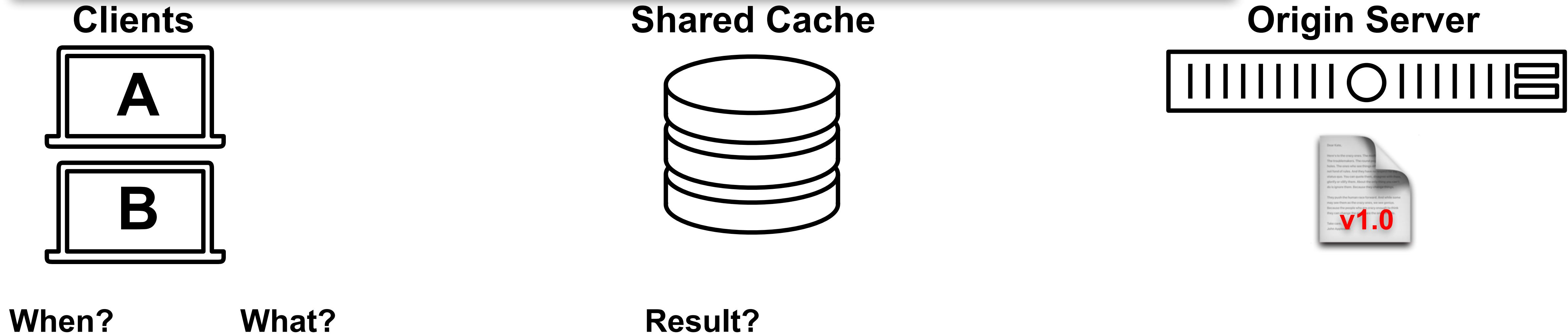
This example just looked at browser cache.

When browser actually makes requests, they may pass through other caches that use a similar algorithm!

# Questions?

Doc TTL  
is 5

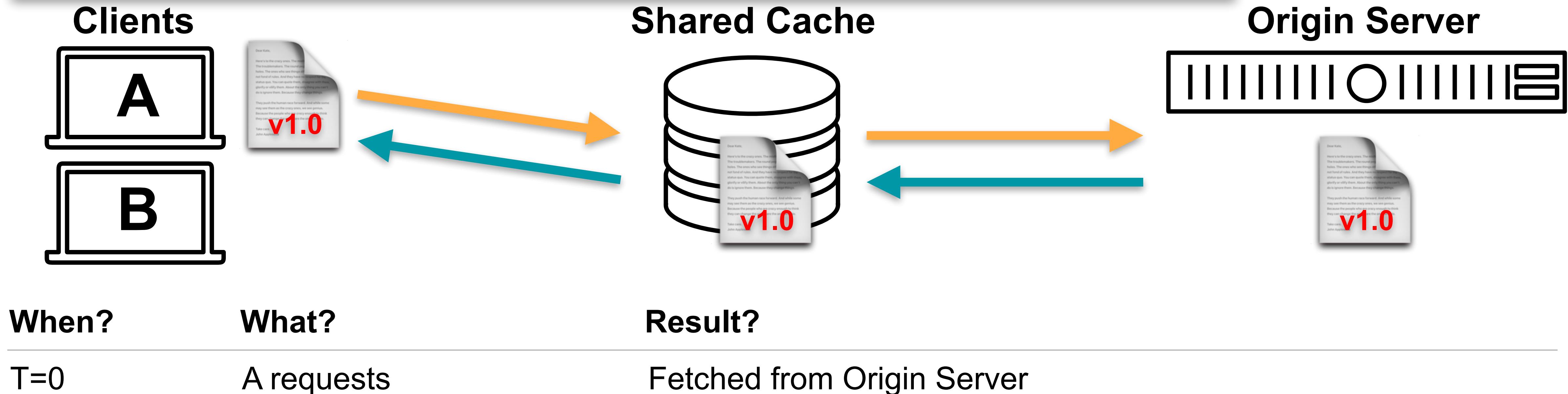
# HTTP caching: Two-client example



- Clients are downloading a document.
- Clients have local (browser) caches.
- Clients also share cache in network.
- Document TTL is 5 (minutes).

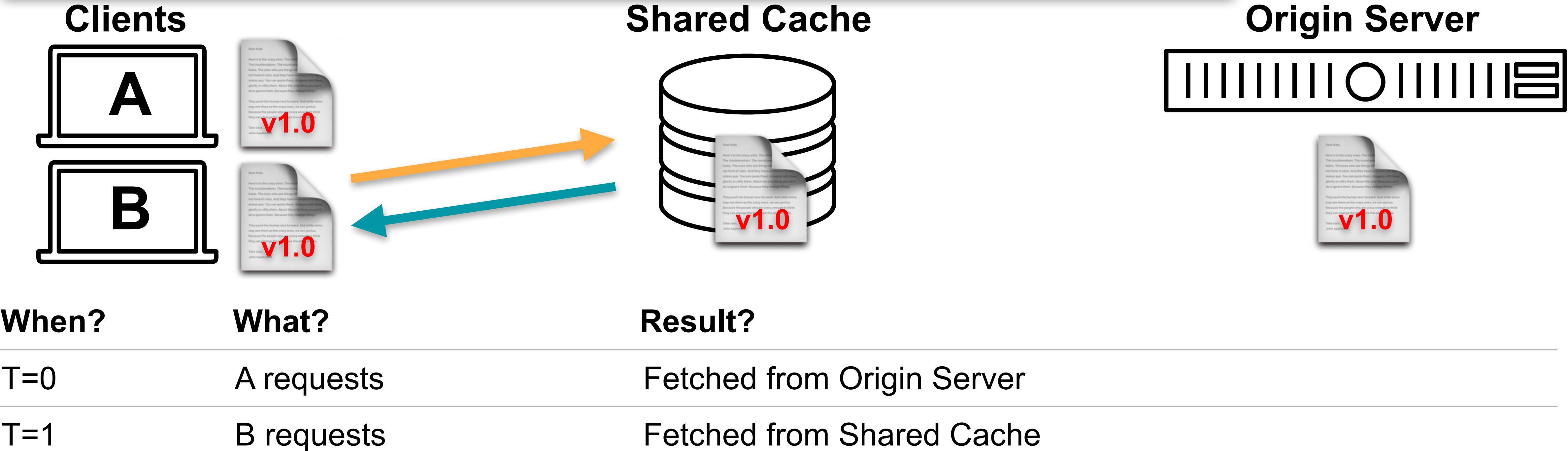
Doc TTL  
is 5

# HTTP caching: Two-client example



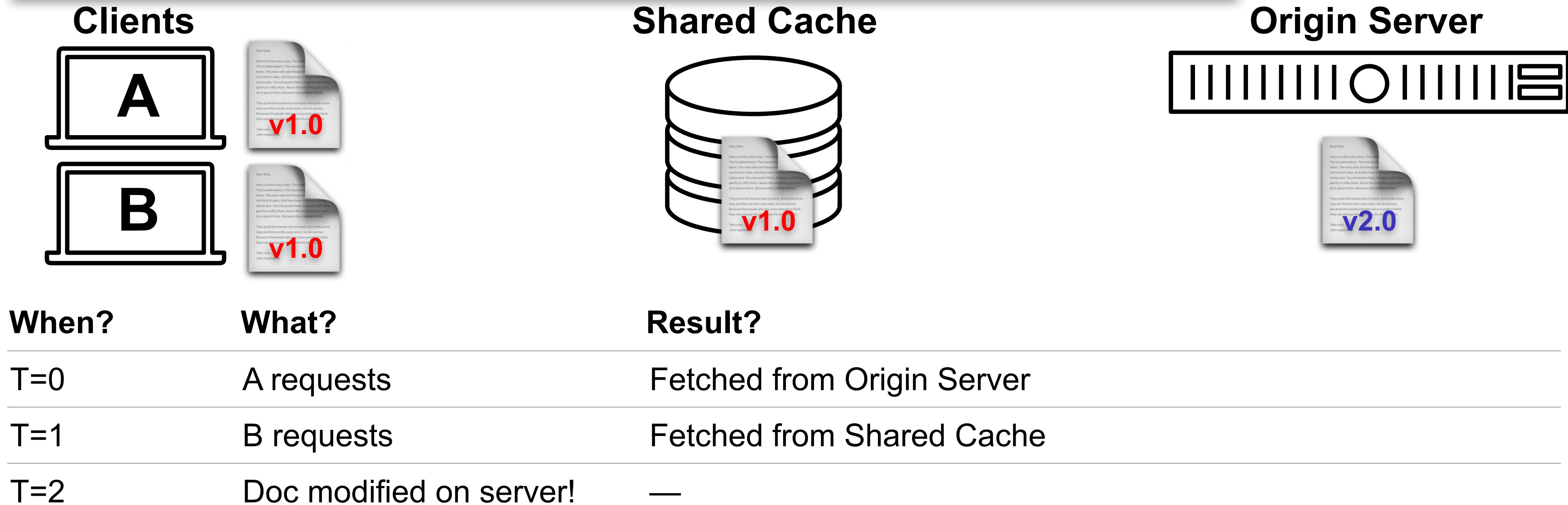
Doc TTL  
is 5

# HTTP caching: Two-client example



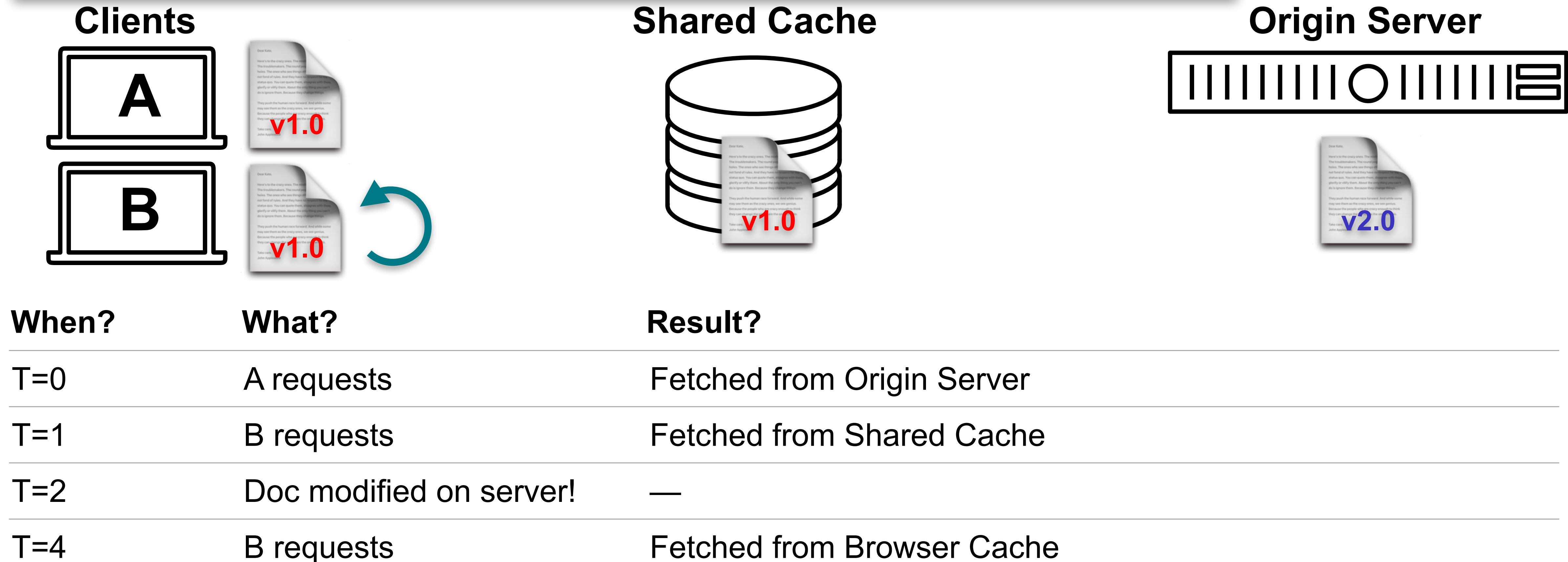
Doc TTL  
is 5

# HTTP caching: Two-client example



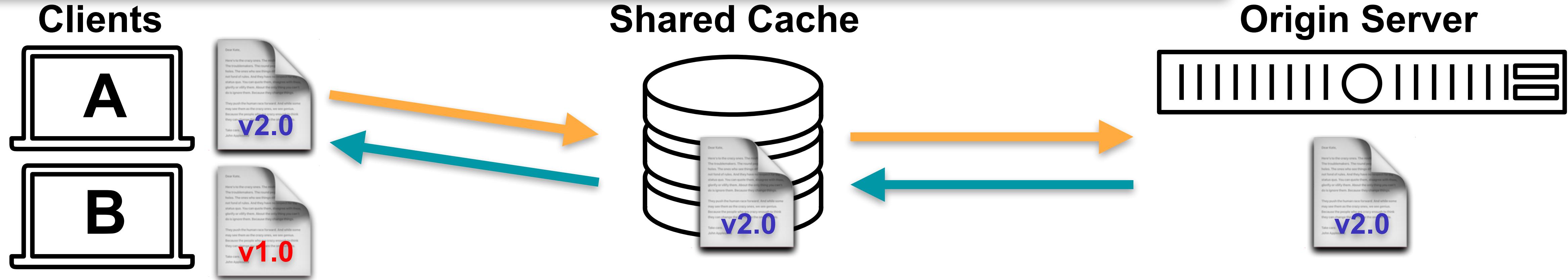
# HTTP caching: Two-client example

# Doc TTL is 5



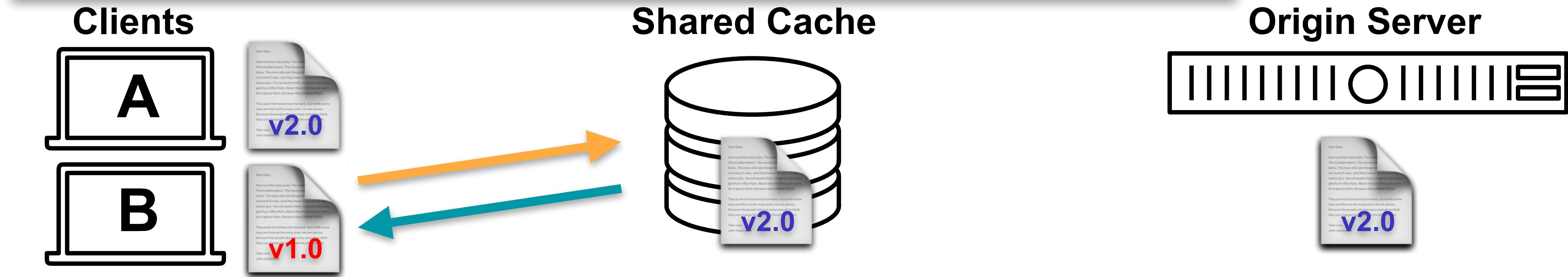
Doc TTL  
is 5

# HTTP caching: Two-client example



| When? | What?                   | Result?                                                                                                                                          |
|-------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| T=0   | A requests              | Fetched from Origin Server                                                                                                                       |
| T=1   | B requests              | Fetched from Shared Cache                                                                                                                        |
| T=2   | Doc modified on server! | —                                                                                                                                                |
| T=4   | B requests              | Fetched from Browser Cache                                                                                                                       |
| T=6   | A requests              | Client A sends If-Modified-Since (to Shared Cache)<br>Shared Cache sends If-Modified-Since (to Origin Server)<br>v2.0 fetched from Origin Server |

# HTTP caching: Two-client example



| When? | What?                   | Result?                                                                                                                                                 |
|-------|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| T=0   | A requests              | Fetched from Origin Server                                                                                                                              |
| T=1   | B requests              | Fetched from Shared Cache                                                                                                                               |
| T=2   | Doc modified on server! | —                                                                                                                                                       |
| T=4   | B requests              | Fetched from Browser Cache                                                                                                                              |
| T=6   | A requests              | Client <b>A</b> sends If-Modified-Since (to Shared Cache)<br>Shared Cache sends If-Modified-Since (to Origin Server)<br>v2.0 fetched from Origin Server |
| T=7   | B requests              | Client <b>B</b> sends If-Modified-Since (to Shared Cache)<br>v2.0 fetched from Shared Cache                                                             |

# Questions?

# HTTP caching: Summary of important cache headers

---

- **Response** headers providing TTLs:
  - Cache-Control: max-age and Expires
- **Request** headers allowing overriding of TTLs:
  - Cache-Control: no-cache and Pragma: no-cache
  - Can be triggered by “shift-refresh”
- Allow requests that skip body if cache is up to date:
  - Request header If-Modified-Since: <date>
- Remember: you can have multiple caches along paths!

# Questions?

# HTTP caching: Final word on Cache-Control header

---

- A couple other important uses of Cache-Control in response...
  - Cache-Control: no-store
    - Don't cache this!
    - Always request from origin server
    - e.g., for things like banking data
  - Cache-Control: private
    - Content only meant for one user
    - Okay to store in private (browser) cache
    - .. but don't store it in shared proxy server cache!

# HTTP caching: Where?

---

- We've discussed how caching works...
- .. but *where* are the caches?
- The client!
- Proxy servers

# Proxy Servers

---

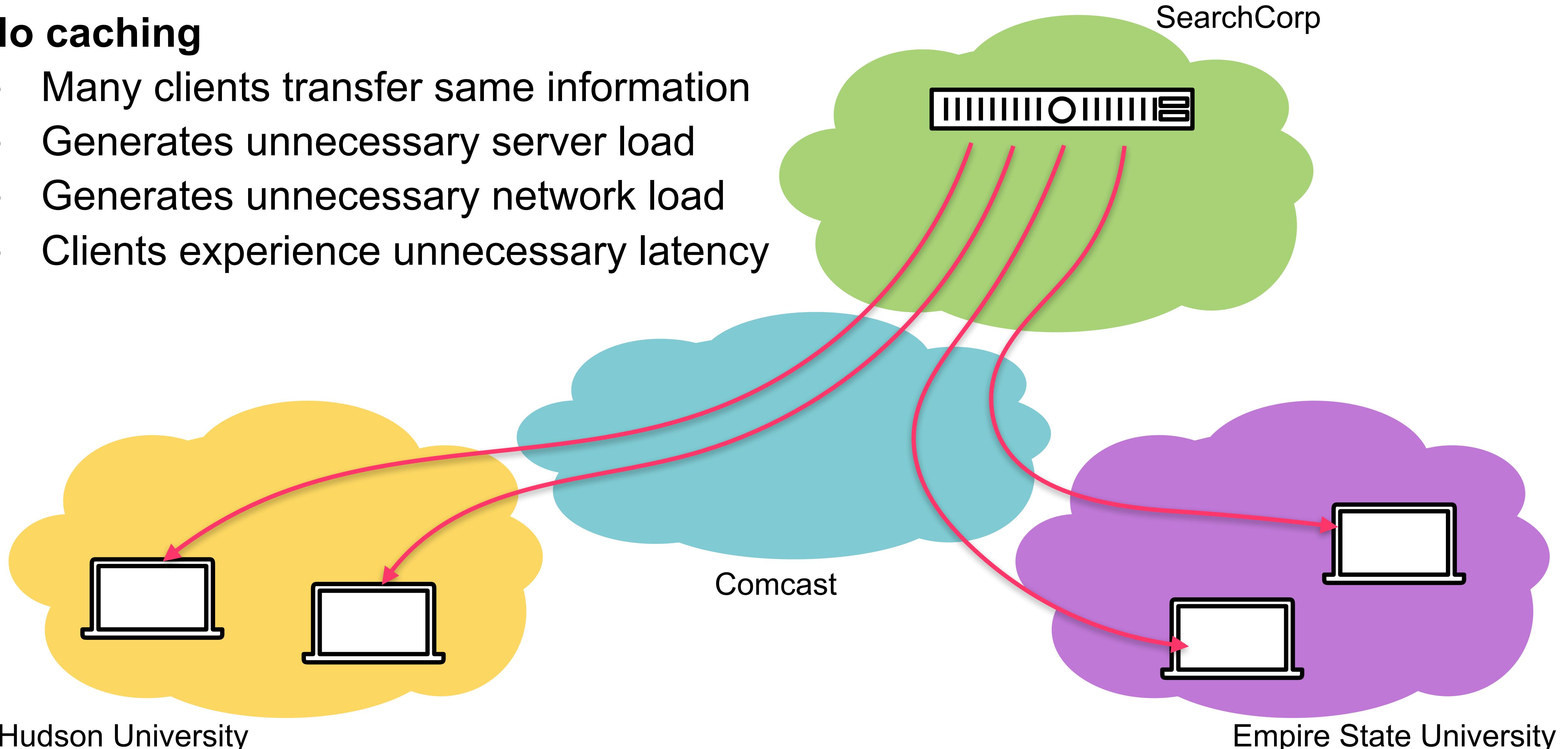
- *Proxy server*: A server that makes requests on behalf of a client
  - The caches we saw in previous examples fit that definition
    - .. *caching* is a major feature of web proxy servers
  - Also often used to *enforce policy*
    - .. company blocks all traffic except through proxy
    - .. proxy has whitelist/blacklist
  - Also often used to do *load balancing*
    - .. request arrives at proxy
    - .. it redirects it to one of several equivalent servers
  - Note: our focus is the web, but other protocols have proxy servers too

# HTTP caching: Where?

---

## No caching

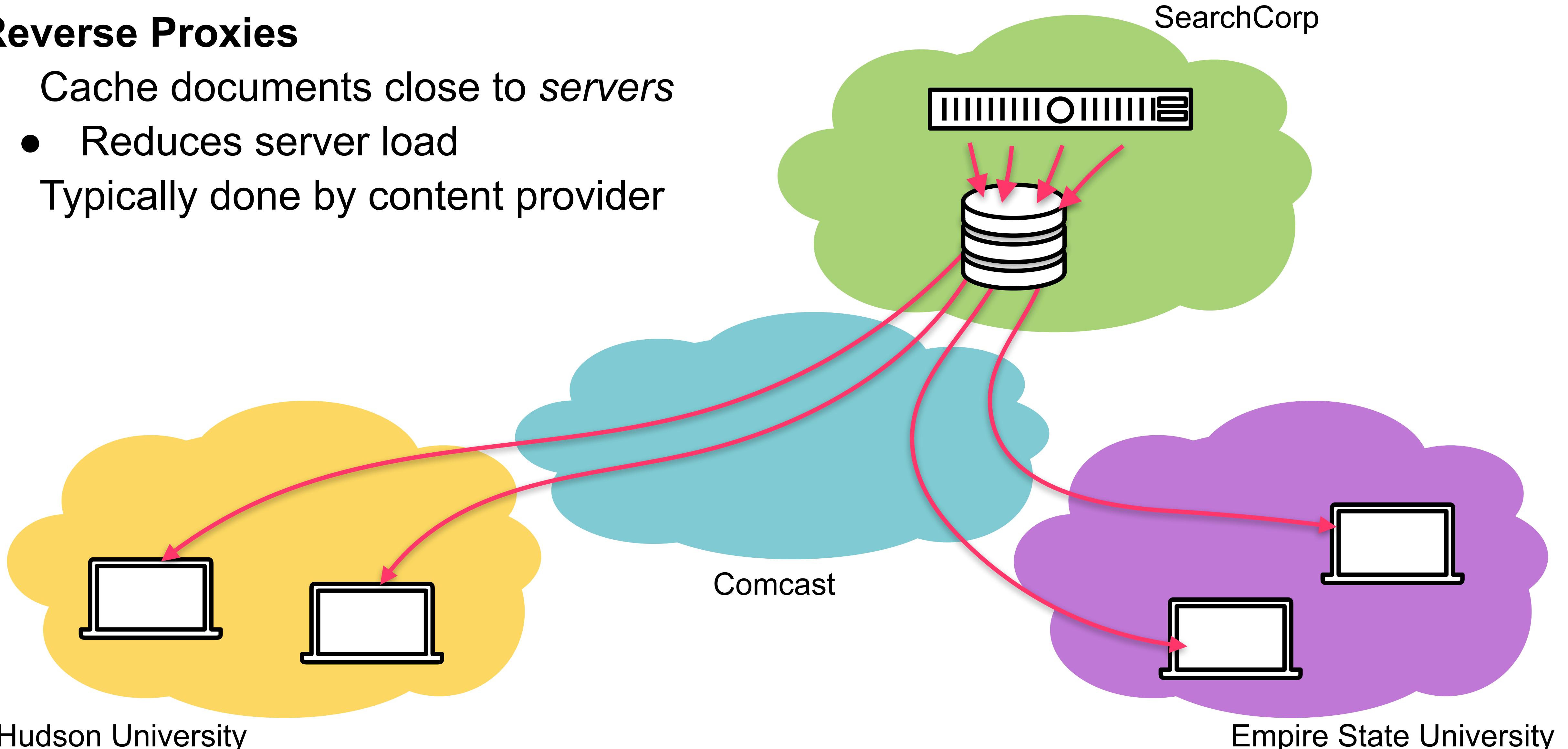
- Many clients transfer same information
- Generates unnecessary server load
- Generates unnecessary network load
- Clients experience unnecessary latency



# HTTP caching: Where?

## Reverse Proxies

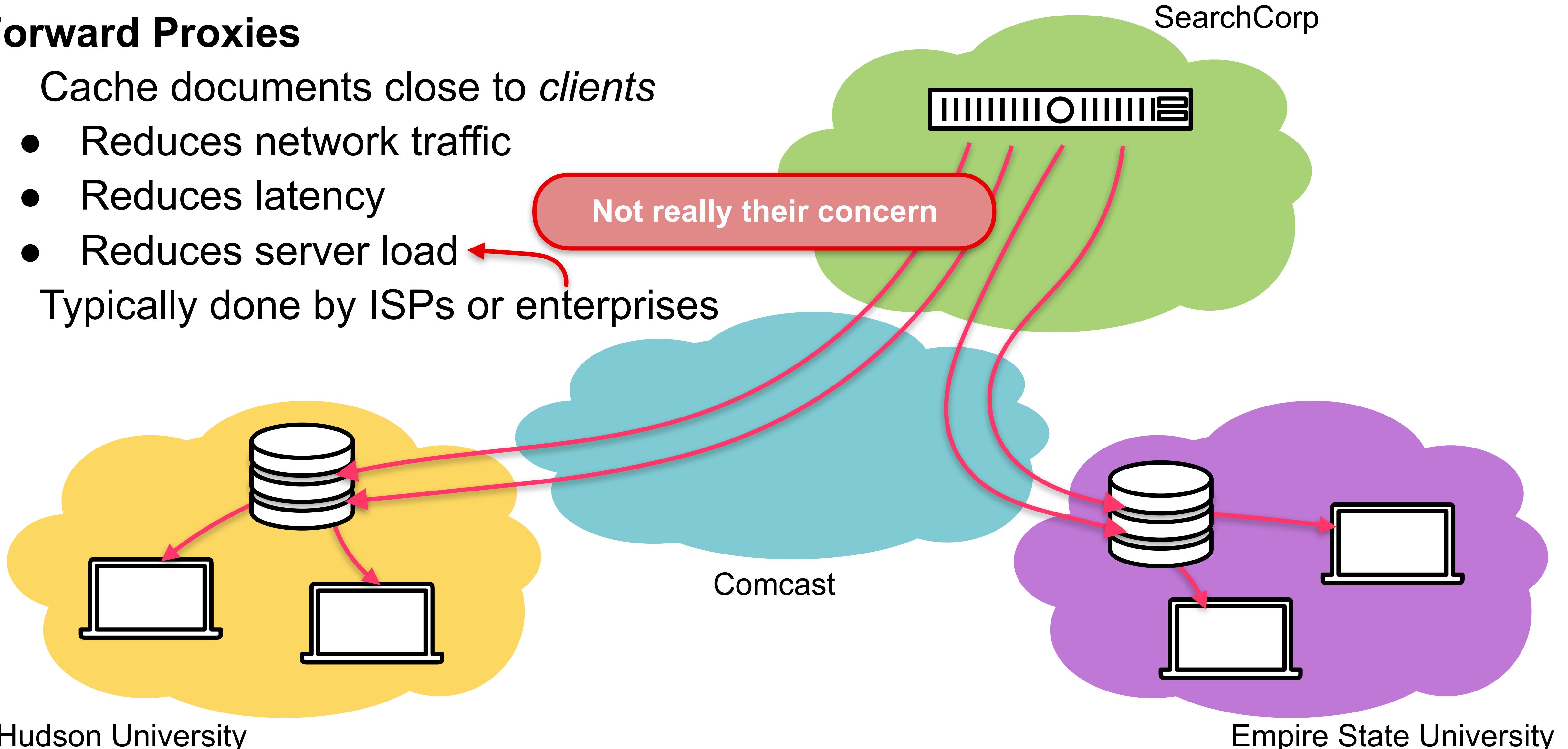
- Cache documents close to servers
  - Reduces server load
- Typically done by content provider



# HTTP caching: Where?

## Forward Proxies

- Cache documents close to *clients*
  - Reduces network traffic
  - Reduces latency
  - Reduces server load
- Typically done by ISPs or enterprises



# HTTP caching: Where?

---

- We've discussed how caching works...
- .. but *where* are the caches?
- The client!
- Proxy servers
  - Forward proxies (near client)
  - Reverse proxies (near server)
- Content Delivery Networks (CDNs)
  - This is its own subtopic!
  - Any questions before we move on to it?

# Content Delivery Networks

# CDNs

---

- *Replication* is a huge benefit to availability, scalability, and performance
  - We saw this with DNS!
  - Can spread the load
  - Places content closer to clients (less latency)
- Caching is a form of opportunistic replication
  - .. but what if a given organization doesn't have a forward proxy?
  - .. what if content provider wants its content *always* replicated?
- Idea: *Caching and replication as a service* — “CDNs 1.0”

# CDNs “1.0”

---

- Large-scale distributed storage infrastructure
  - (Usually) administered by one entity
  - e.g., Akamai has 275,000+ servers in 136 countries
- How does content provider get its data onto Akamai’s servers?
- Two major ways
  - Pull
  - Push
  - .. we’ll come back to these in a moment
- Both typically used with DNS trick mentioned in previous lecture

# CDNs “1.0”: The basic idea

---

- Content provider buys service from a CDN, e.g., Akamai
- CDN creates new domain names for the customer content provider
  - e.g., [e12596.dscj.akamaiedge.net](http://e12596.dscj.akamaiedge.net) for [cnn.com](http://cnn.com)
  - The CDN’s DNS servers are authoritative for the new domains
- Content provider modifies its content so that embedded URLs reference the new domains
  - “Akamaize” content
  - e.g.: <http://www.cnn.com/some-photo.jpg> becomes <http://e12596.dscj.akamaiedge.net/some-photo.jpg>
- Initial request goes to CNN (e.g., for main <http://www.cnn.com> page)
  - .. but embedded links go to Akamai, which handles DNS resolution for URL
  - .. Akamai DNS servers pick one of their 275,000+ servers to serve it  
(based on IP geolocation, server load, etc. — see Lecture 17 - Intelligent indirection)

# CDNs “1.0”: The basic idea

---

- Content provider buys service from a CDN, e.g., Akamai
- CDN creates new domain names for the customer content provider
  - e.g., [e12596.dscj.akamaiedge.net](#)
- The CDN creates new domains
  - Content provider can still use their own domain name
- Content provider can still use their own domain name
  - “Akamai” domain is just a CNAME record
  - e.g.: <http://cdn.cnn.com> (CNAME, [cdn.cnn.com](http://cdn.cnn.com), [e12596.dscj.akamaiedge.net](http://e12596.dscj.akamaiedge.net))
- Initial request goes to CNN (e.g., <http://www.cnn.com> page),
  - .. but embedded links go to Akamai, which handles DNS resolution for URLs
  - .. Akamai DNS servers pick one of their 275,000+ servers to serve it  
(based on IP geolocation, server load, etc. — see Lecture 17 - Intelligent indirection)

# CDNs “1.0”: The basic idea

---

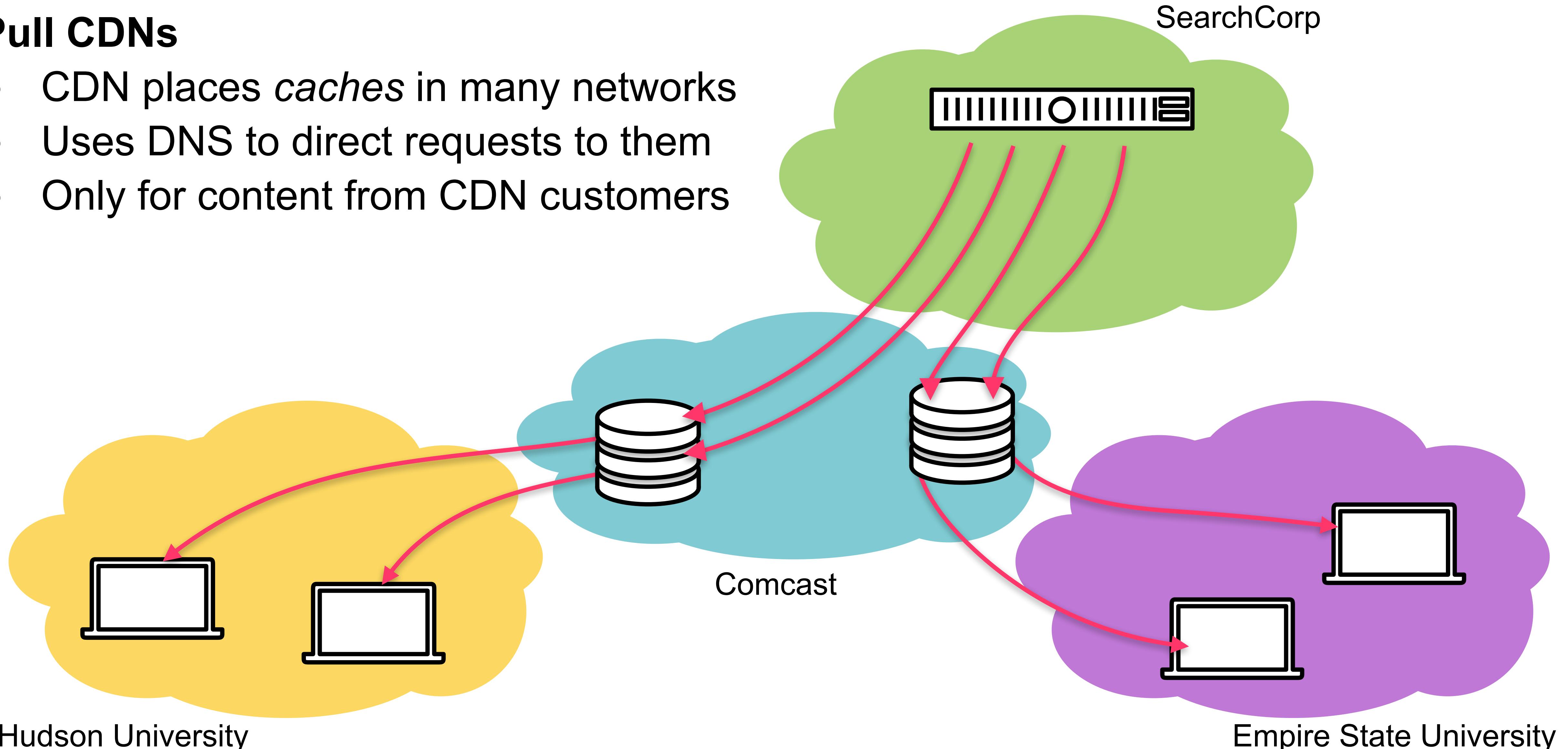
- How does content provider get data onto CDN’s servers?
- *Pull*
  - Akamai servers act like a cache
  - Content provider gives CDN “origin” URL
  - When a client requests from Akamai
    - .. if cached, serve it
    - .. if not cached, request (“pull”) from origin, cache it, serve it
- *Push*
  - Akamai servers just act like normal servers
  - Content provider uploads content to CDN (“pushes” their content)
  - When a client requests from Akamai, just serve like any web server
- Various tradeoffs
  - Short version: pull is less work for content provider but push gives more control

# CDNs “1.0”: The basic idea

---

## Pull CDNs

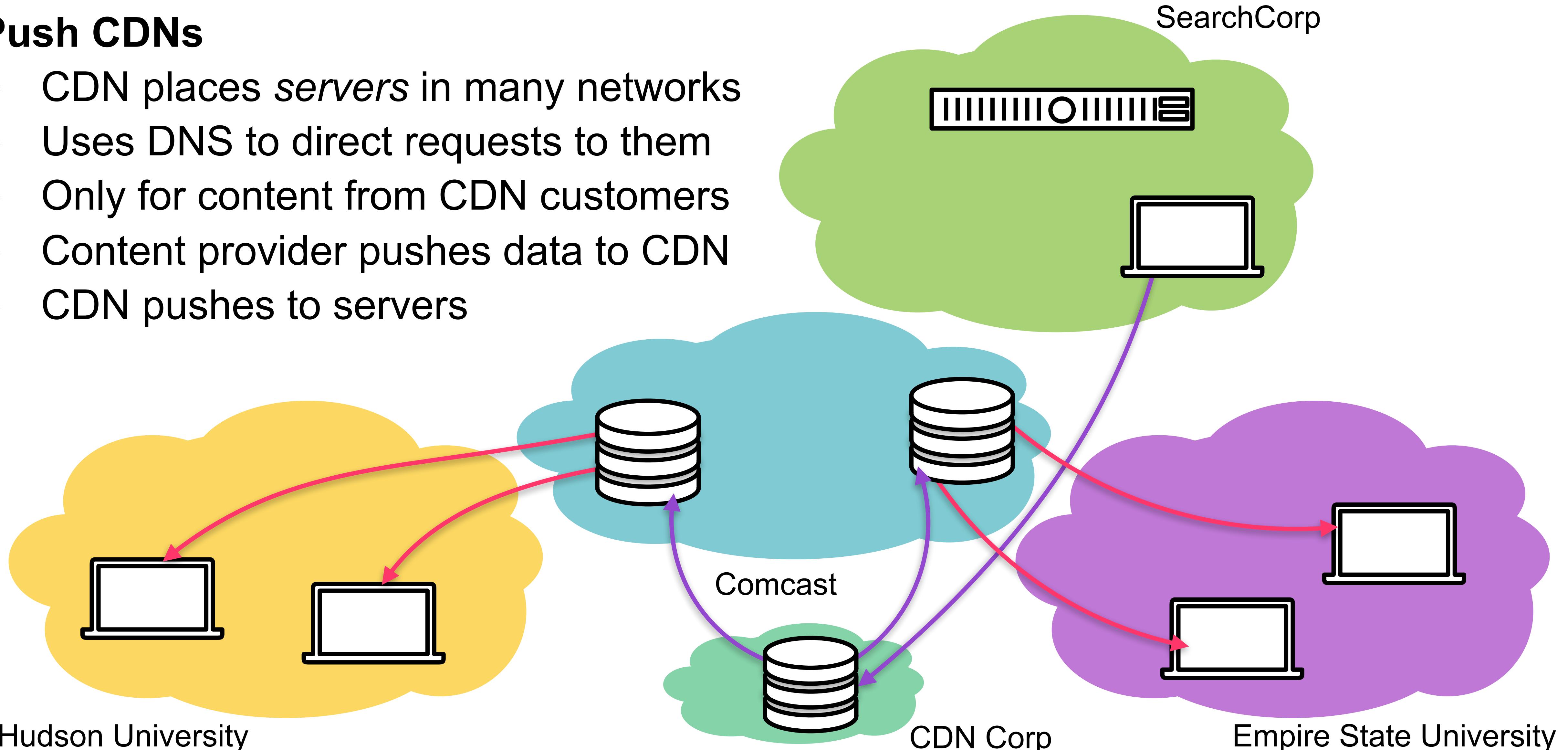
- CDN places *caches* in many networks
- Uses DNS to direct requests to them
- Only for content from CDN customers



# CDNs “1.0”: The basic idea

## Push CDNs

- CDN places *servers* in many networks
- Uses DNS to direct requests to them
- Only for content from CDN customers
- Content provider pushes data to CDN
- CDN pushes to servers



# CDNs

---

- Clear to see how this works for static content (I called this “CDN 1.0”)
  - Replicate/cache on demand (pull)
  - Replicate manually by content provider (push)
  - Pick replica/cache server via clever DNS server
- What about dynamic content/features?
  - Constant evolution in this direction
  - A relatively hot commercial area!

Questions?

# TCP and HTTP

# TCP and HTTP

---

- Caching can be a big performance boost!
- But the way HTTP uses TCP also makes a big difference!
  - What am I talking about?
  - Let's see...

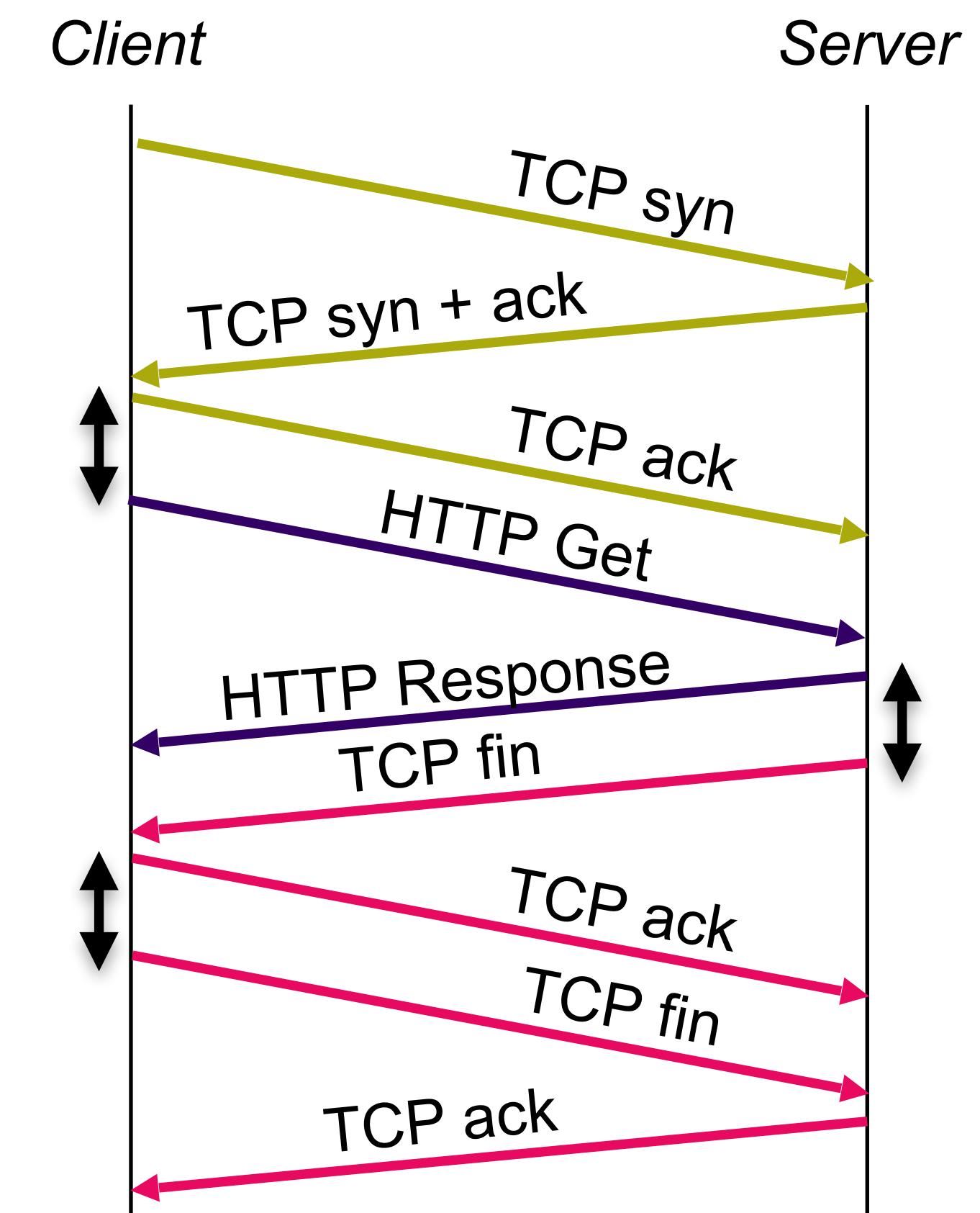
# TCP and HTTP: Observations

---

- Many web pages composed of multiple objects/resources
  - e.g., HTML file and a bunch of embedded images
- Many of the resources are pretty small — only a few packets
  - Small images
  - 304 responses (just checking if cache is up to date)
  - Etc.
- Loading [cnn.com](http://cnn.com) resulted in about 40 responses that fit in a packet!
- TCP overheads fetching these can be very large!

# HTTP Performance: TCP and HTTP

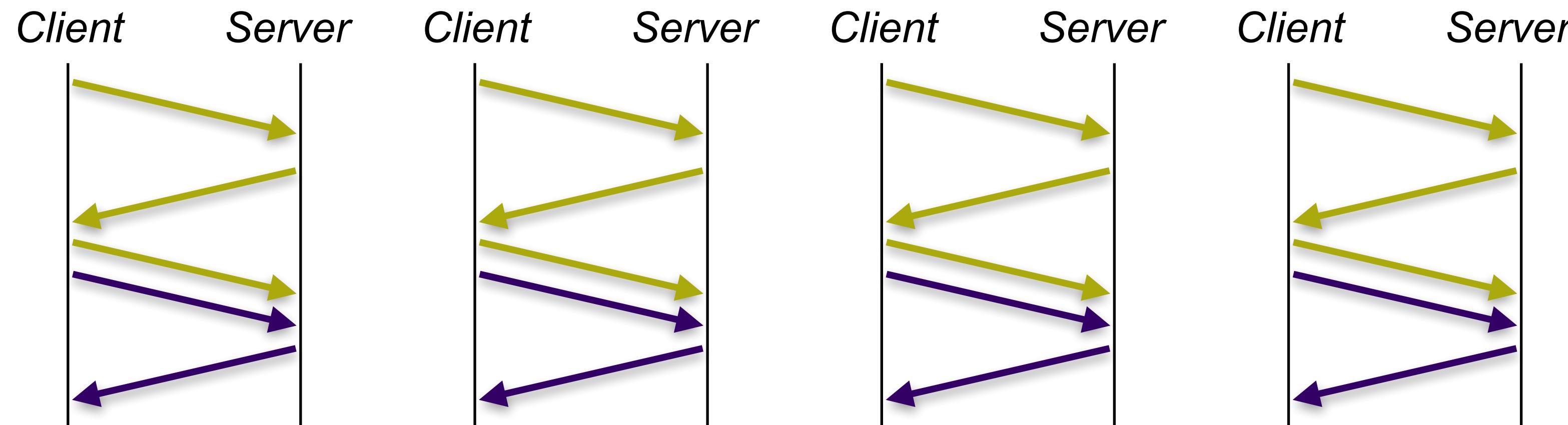
- Naive approach — one object at a time
  - Client creates TCP connection
  - Client sends **request**
  - Server sends **response**
  - Server closes connection
- Transmission delay is not the issue (<3ms at 5Mbps)
- Time dominated by RTTs (30ms RTT to Google)
- How many RTTs to download 40 small objects?
  - $2 \cdot 40 = 80$  RTTs = 2.4 seconds
  - Why not 2 RTTs per object? Why not 3?



# HTTP Performance: TCP and HTTP

---

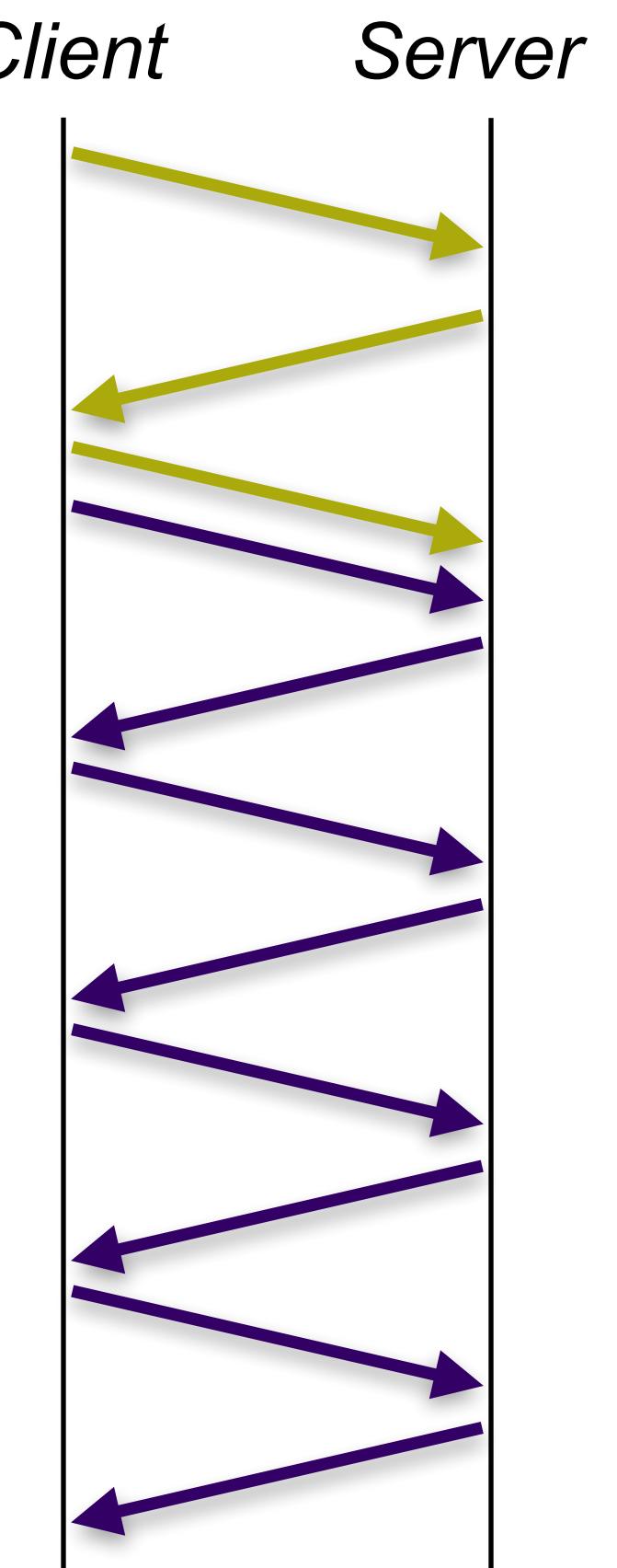
- **Concurrent requests**
  - Make several requests *in parallel*
- How many RTTs to download 40 small objects, 4 at once?
  - $2 \cdot 40/4 = 20$  RTTs = 600ms (4x improvement)
- Browsers do this — limit has changed (was 6 per site for a long time?)



# HTTP Performance: TCP and HTTP

---

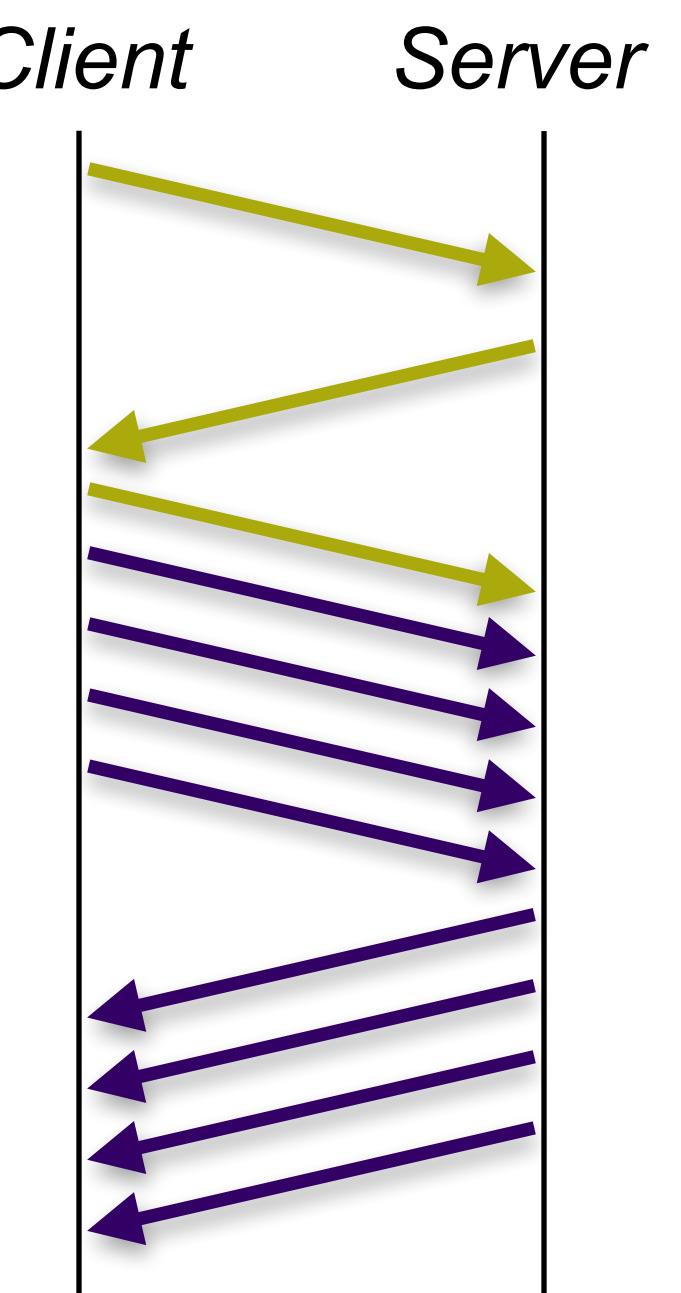
- **Persistent connections**
  - Maintain TCP connection across multiple requests
  - Client or server can tear down connection after idle period
- Performance advantages:
  - Avoid overhead of connection set-up and tear-down
  - Allow TCP congestion window to increase (next lectures)
- How many RTTs to download 40 small objects?
  - $40 + 1 = 41$  RTTs = 1.23 seconds
  - With four concurrent persistent connections? 330ms
- Browsers do it — optional in HTTP 1.0; default in HTTP 1.1



# HTTP Performance: TCP and HTTP

---

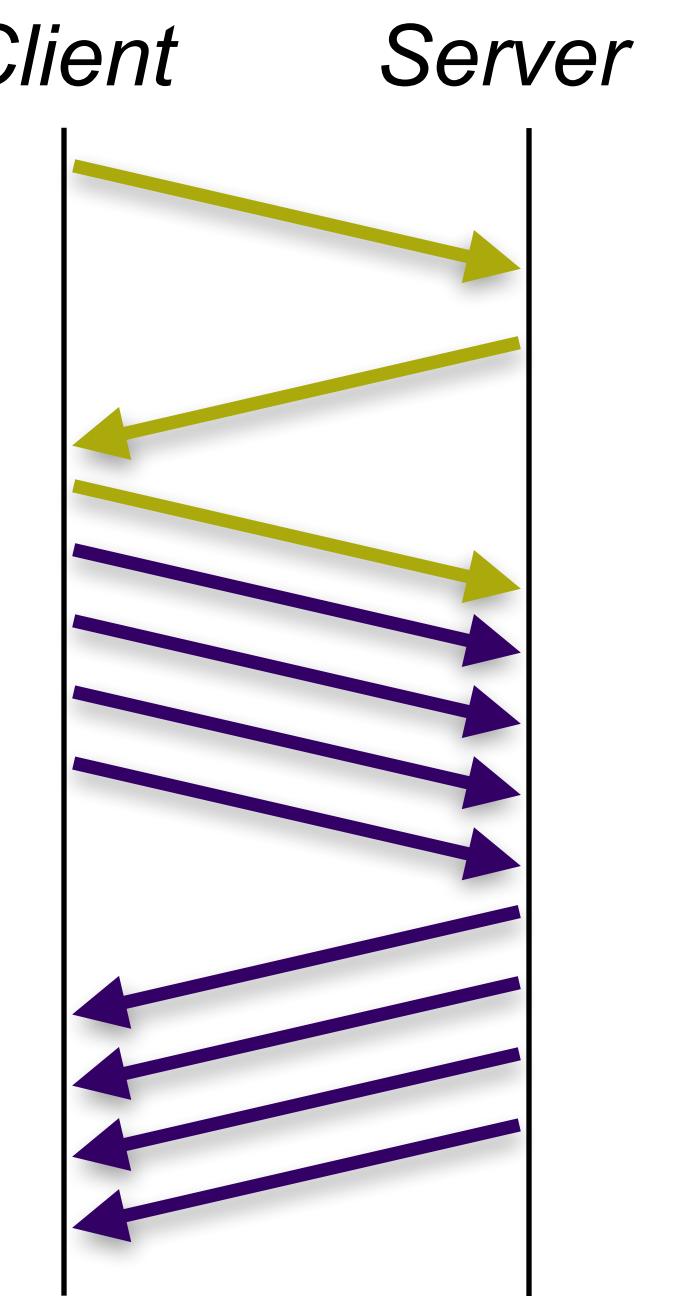
- ***Pipelined connections***
  - Persistent connections to the next level!
  - Send multiple requests at once
- Performance advantages:
  - Reduces the RTTs
  - Multiple very small requests/responses can be coalesced into smaller number of larger packets
- How many RTTs to download 40 small objects?
  - 2! Probably dominated by transmission delay now!
- Appeared in HTTP 1.1
  - .. and promptly disabled



# HTTP Performance: TCP and HTTP

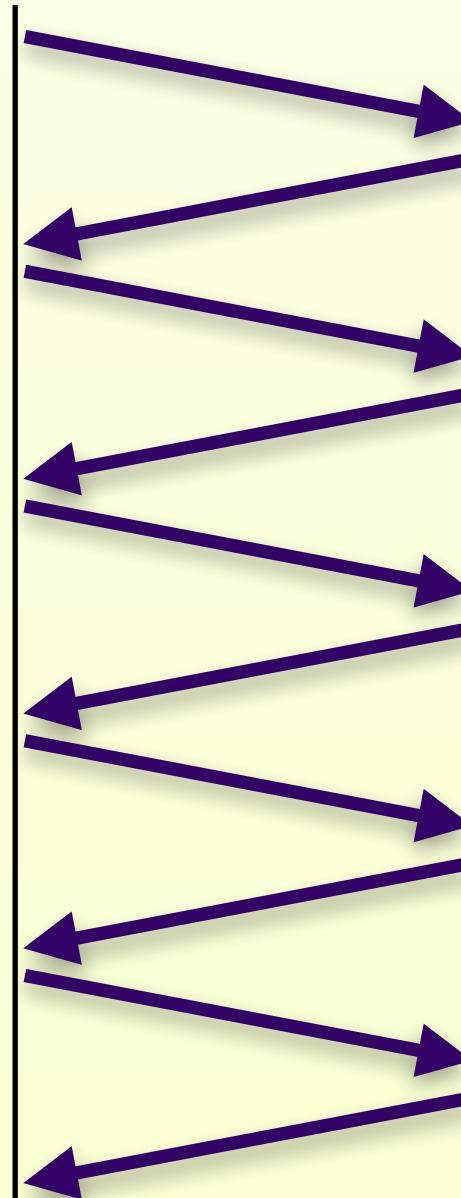
---

- Pipelined connections *aren't actually used*
  - But they seemed like a huge win!
  - What happened?!
    - .. primarily two reasons
- Reason 1: Bugs!
  - One manifestation: images on page are swapped!
  - Often blamed on proxy servers
  - My guess: bad adaptation of multithreaded non-pipelined version
- Reason 2: *Head-of-line blocking*

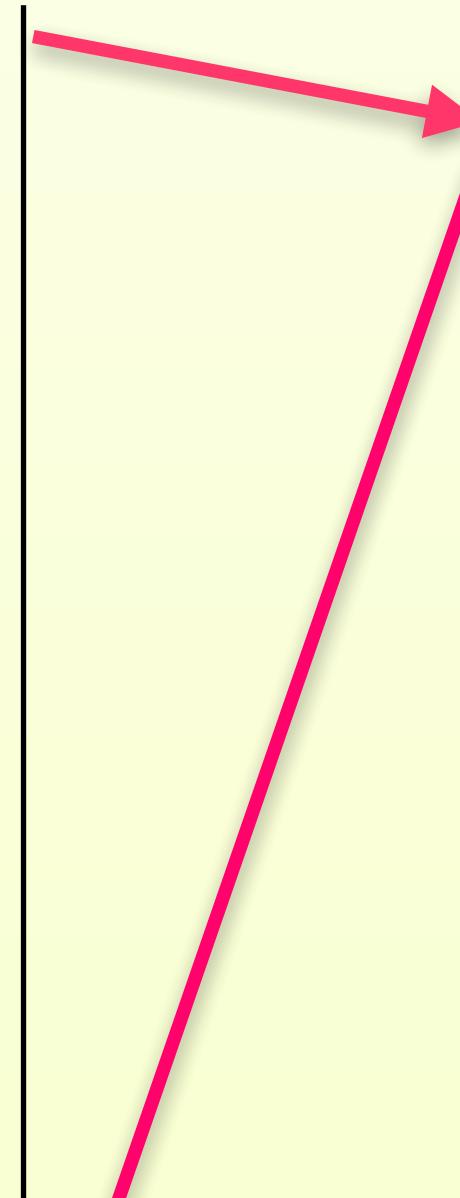


# HTTP Performance: TCP and HTTP

Client      Server



Client      Server



Downloading six objects

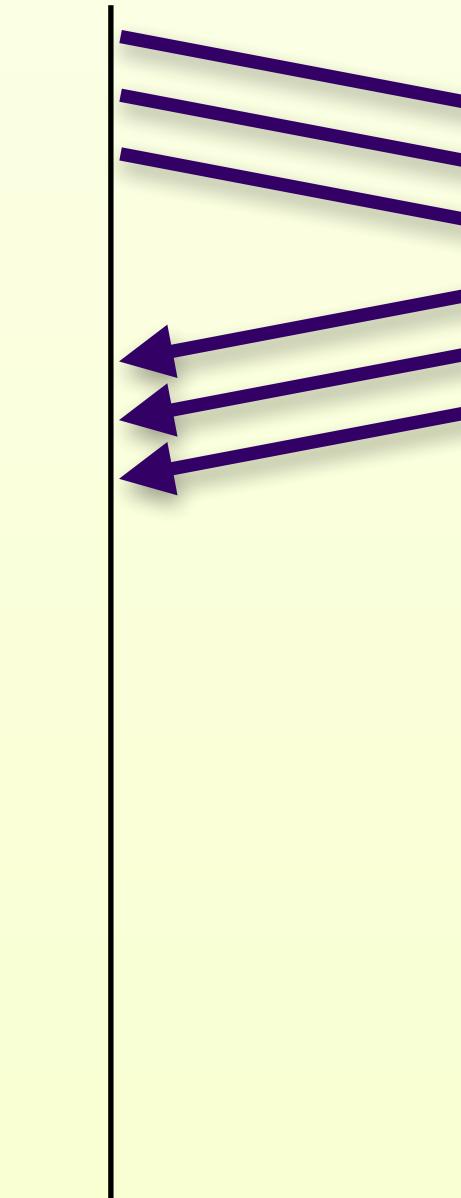
Persistent finishes 5 fairly quick, 1 slow

Pipelined finishes 3 very quick, 3 slow

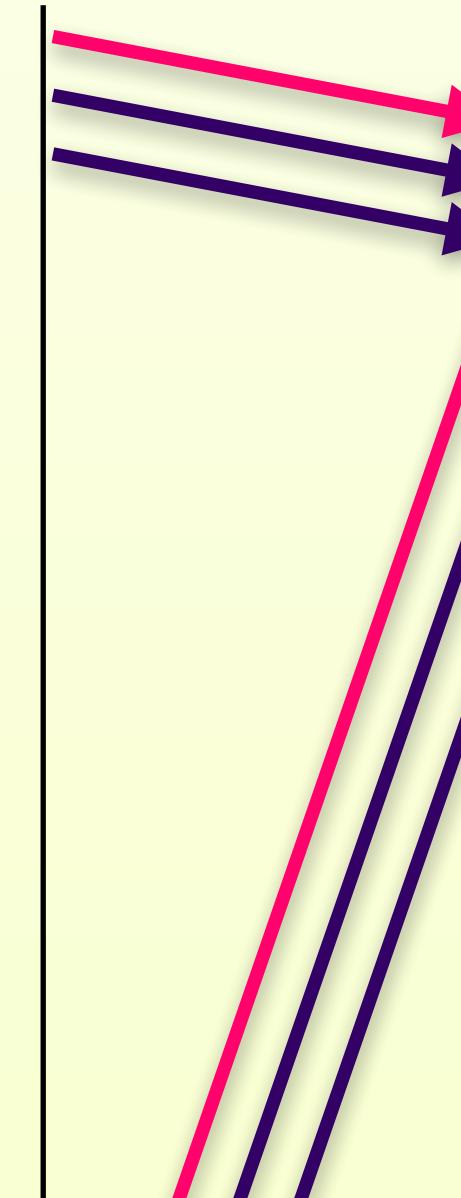
...

1 slow at the “head of the line”  
blocked 2 others

Client      Server



Client      Server



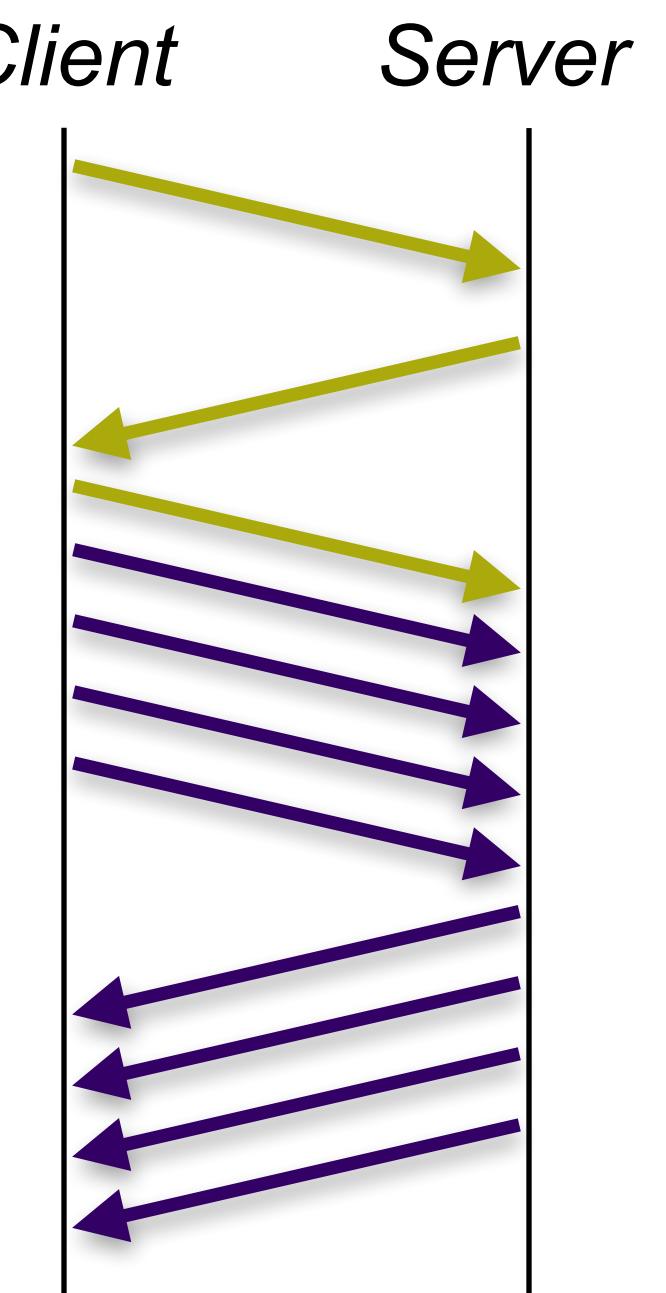
Two Persistent Connections

Two Pipelined Connections

# HTTP Performance: TCP and HTTP

---

- Pipelined connections *aren't actually used*
  - But they seemed like a huge win!
  - What happened?
    - .. primarily two reasons
- Reason 1: Bugs!
  - One manifestation: images on page are swapped!
  - Often blamed on proxy servers
  - My guess: bad adaptation of multithreaded non-pipelined version
- Reason 2: *Head-of-line blocking*
  - Small requests get stuck behind big one
- HTTP 2 replaced this with *multiplexing* with better results



# HTTP Performance: TCP and HTTP

---

- Summing up...
- Single connection per small download can leaves performance on the floor!
  - RTTs kill your performance!
- Things you can do about it:
  - Concurrent connections
  - Persistent connections
  - Pipelined connections
  - .. and combinations thereof!

} Actually used today
- (And multiplexed connections in HTTP 2&3)
- Why doesn't this apply to large downloads?
  - If transmission time dominates, only solution is get more bandwidth!

|                        |       |
|------------------------|-------|
| www.berkeley.edu (AWS) | 50ms  |
| eecs.berkeley.edu      | 30ms  |
| cs.umass.edu           | 100ms |
| www.umass.edu (Akamai) | 25ms  |
| www.usp.br             | 300ms |

# Questions?

Have a good week!

# Attributions

---

File:Mozilla dinosaur head logo.png, [CC BY 3.0](#)

[https://commons.wikimedia.org/wiki/File:Mozilla\\_dinosaur\\_head\\_logo.png](https://commons.wikimedia.org/wiki/File:Mozilla_dinosaur_head_logo.png)

File:Printer\_dot\_matrix\_EPSON\_VP-500.jpg, [CC BY-SA 2.0](#)

[https://commons.wikimedia.org/wiki/File:Printer\\_dot\\_matrix\\_EPSON\\_VP-500.jpg](https://commons.wikimedia.org/wiki/File:Printer_dot_matrix_EPSON_VP-500.jpg)

Many slides borrowed/adapted from earlier Berkeley CS168/EE122

# Putting The Pieces Together

## Ethernet, DHCP, ARP, etc.

This content is protected and may not be shared, uploaded or distributed.

# Today in Internet history...

---

- April 14, 1998 (22 years ago)...
- Netflix website launched!
- 925 movies
- .. mailed to you on DVD; no streaming until 2007 (nine years later)
- Pay-per-movie; subscription started the following year
- A year after that, it offers itself to Blockbuster for \$50 million
  - .. Blockbuster probably should have taken them up on that
  - 2019 Netflix: \$20 billion gross, \$1.86 billion net
  - 2020 Blockbuster: One remaining store in Bend, Oregon... maybe? 😞

# Putting The Pieces Together

## Ethernet, DHCP, ARP, etc.

# In the past...

---

- We've talked a lot about L3; specifically IP!
  - Common routing
    - Intradomain (D-V and L-S)
    - Interdomain (BGP)
  - Addresses
    - Structure, properties (CIDR, aggregatable, etc.)
- We've talked some about L2; mostly Ethernet
  - Common routing
    - L-S
    - Learning switches and STP
  - Addresses
    - ... ?

# Today...

---

- Fill in some gaps!
  - Bias towards Ethernet (L2) and IPv4 (L3)
  - Generally similarities with other L2/L3 (e.g., WiFi and IPv6)
- Ethernet
  - History and background:
    - Multiple access, ALOHA, CSMA, CSMA/CD, and exponential backoff
    - Addresses, broadcast and multicast service types
    - Modern Ethernet
- How do L2 and L3 really fit together?
  - Routing
  - Addresses (ARP)
  - How does a host know its own IP address? (DHCP)
- Example — all together now!
- Bonus topic: Network Address Translation (NAT)

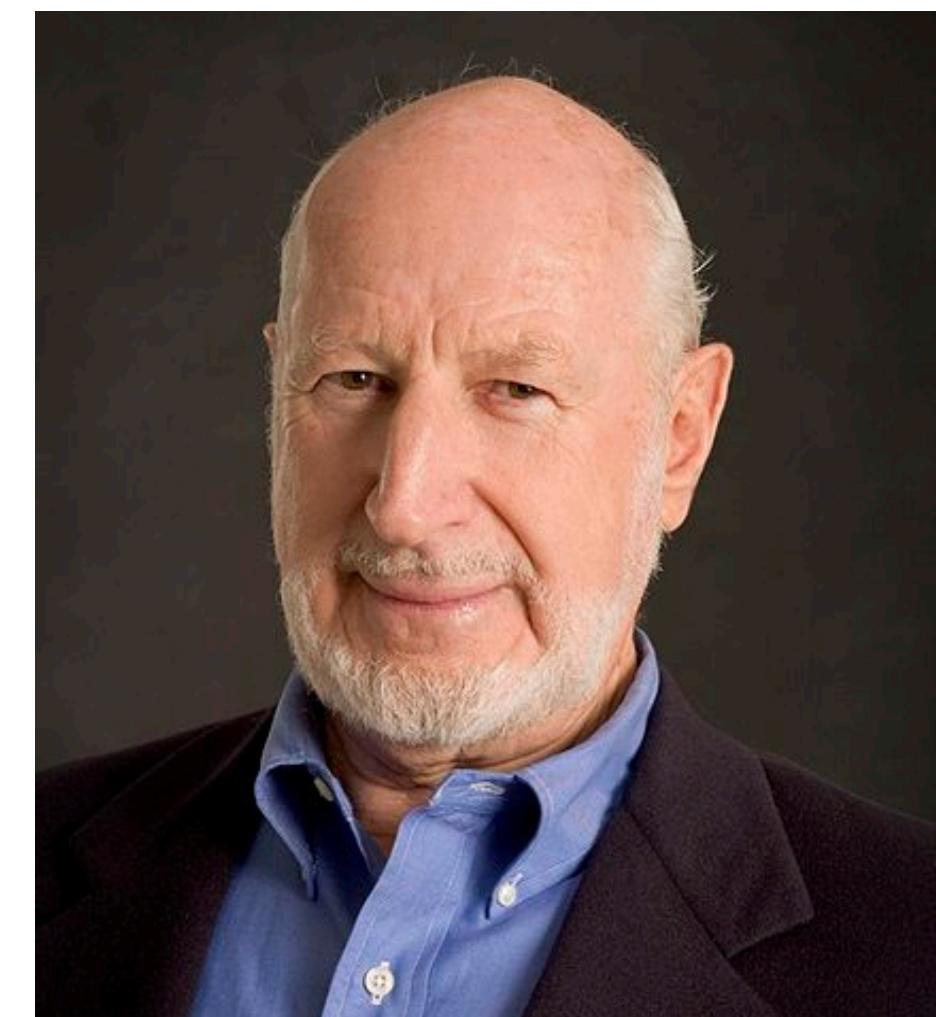
# Digging into Ethernet

## Our L2 technology of choice

# ALOHA

---

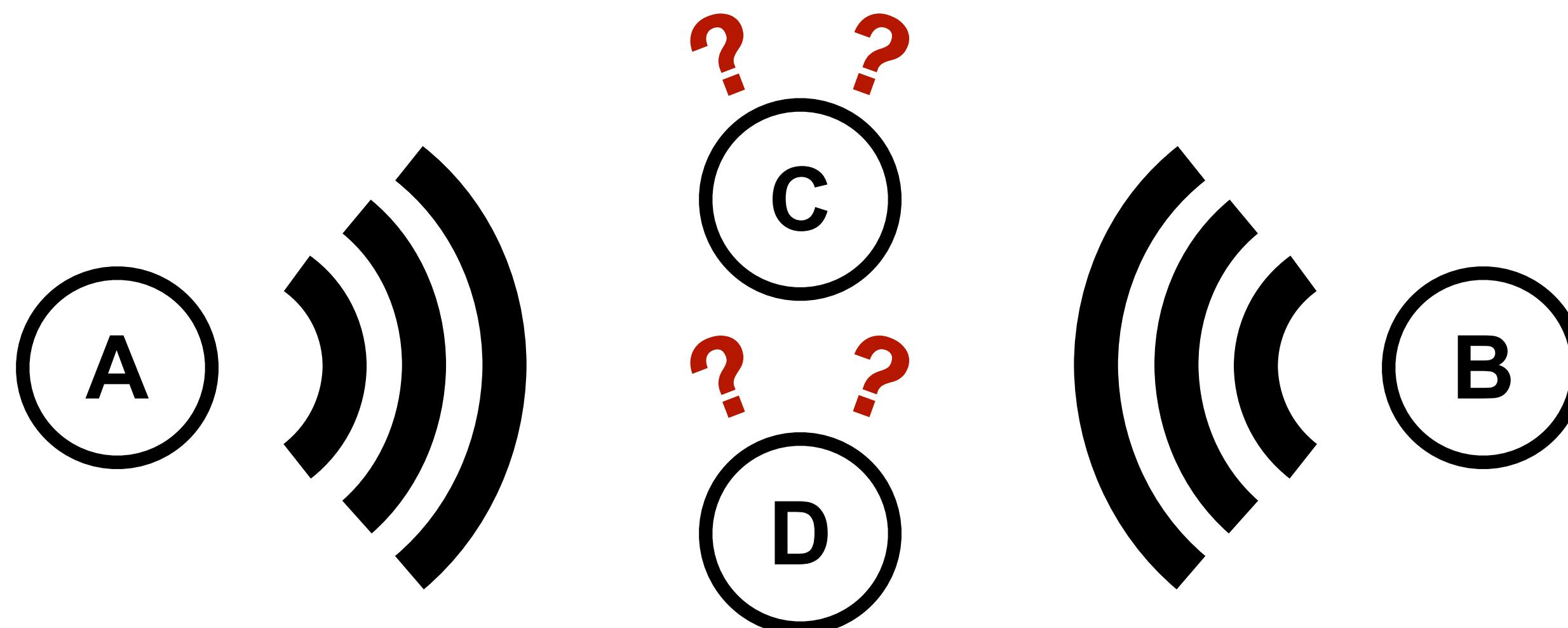
- In 1968, Norman Abramson had a problem at the University of Hawaii...
  - How to allow people on the other islands access to the U of H computer?
- His solution: ALOHAnet
  - Additive Links On-line Hawaii Area
  - Wireless connection from terminals on the other islands!
  - *Hugely* influential!
- We'll return to ALOHA; first let's talk about *shared media*  
(And no, I don't mean BitTorrenting the full works of Abba)



# Shared Media

---

- In a radio network, nodes utilize a *shared medium* (electromagnetic spectrum in some locality)
- Transmissions from different nodes may interfere or *collide* with each other!
- We need a system for allocating the medium to everyone wanting to use it
  - .. a *multiple access protocol*



# Common Multiple Access Protocol approaches

---

- **Divide medium up by frequency** (*Frequency Division Multiplexing*)
  - Can be wasteful! Only so much EM spectrum to go around, and many frequencies likely to be idle often (traffic is bursty)
- **Divide medium up by time** (several ways)
  - Divide time into fixed-sized “slots”, each sender gets their own slot (*Time Division Multiplexing*); same drawback as FDM
  - Take turns...

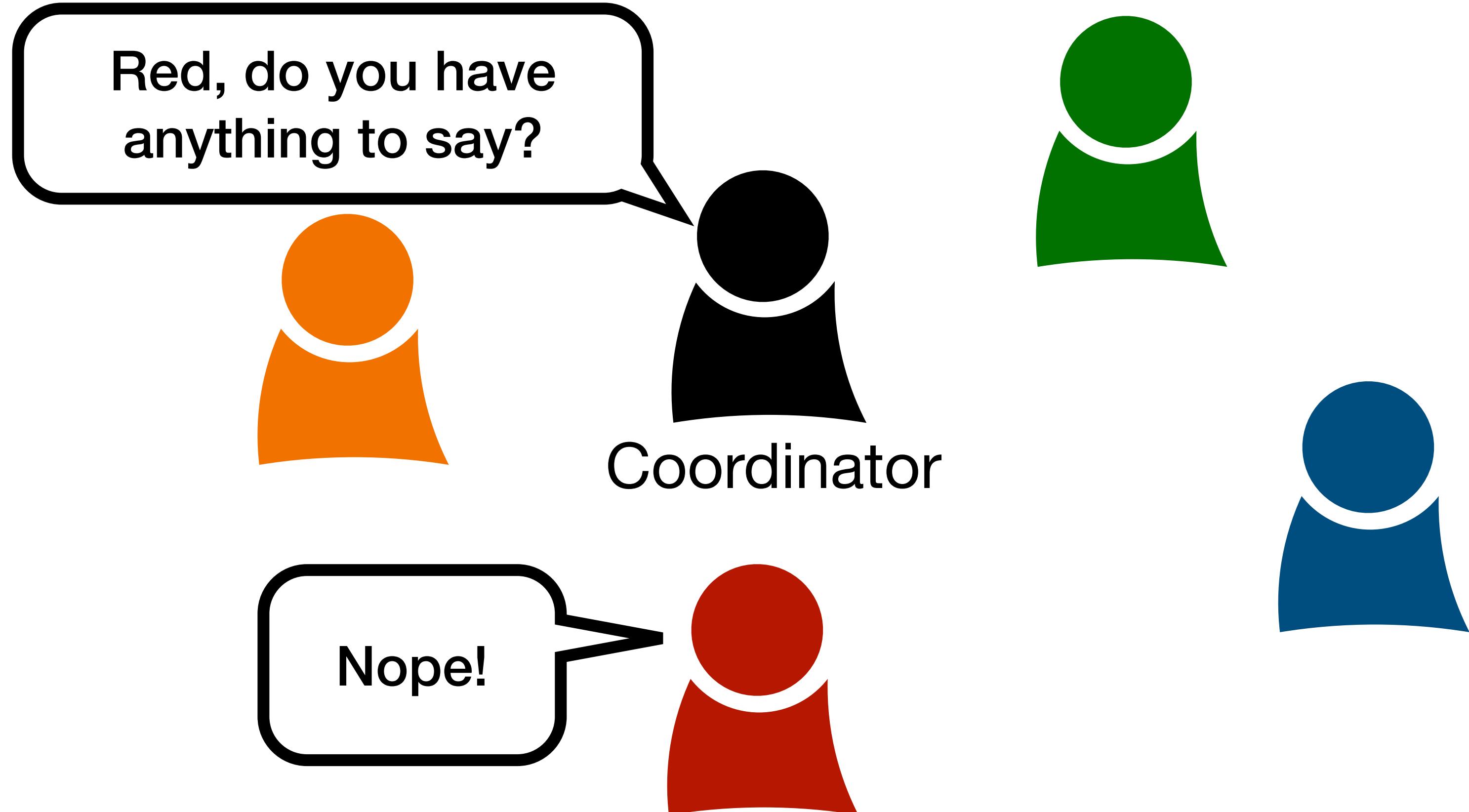


*Partitioning approaches*

# Turn-Taking schemes

---

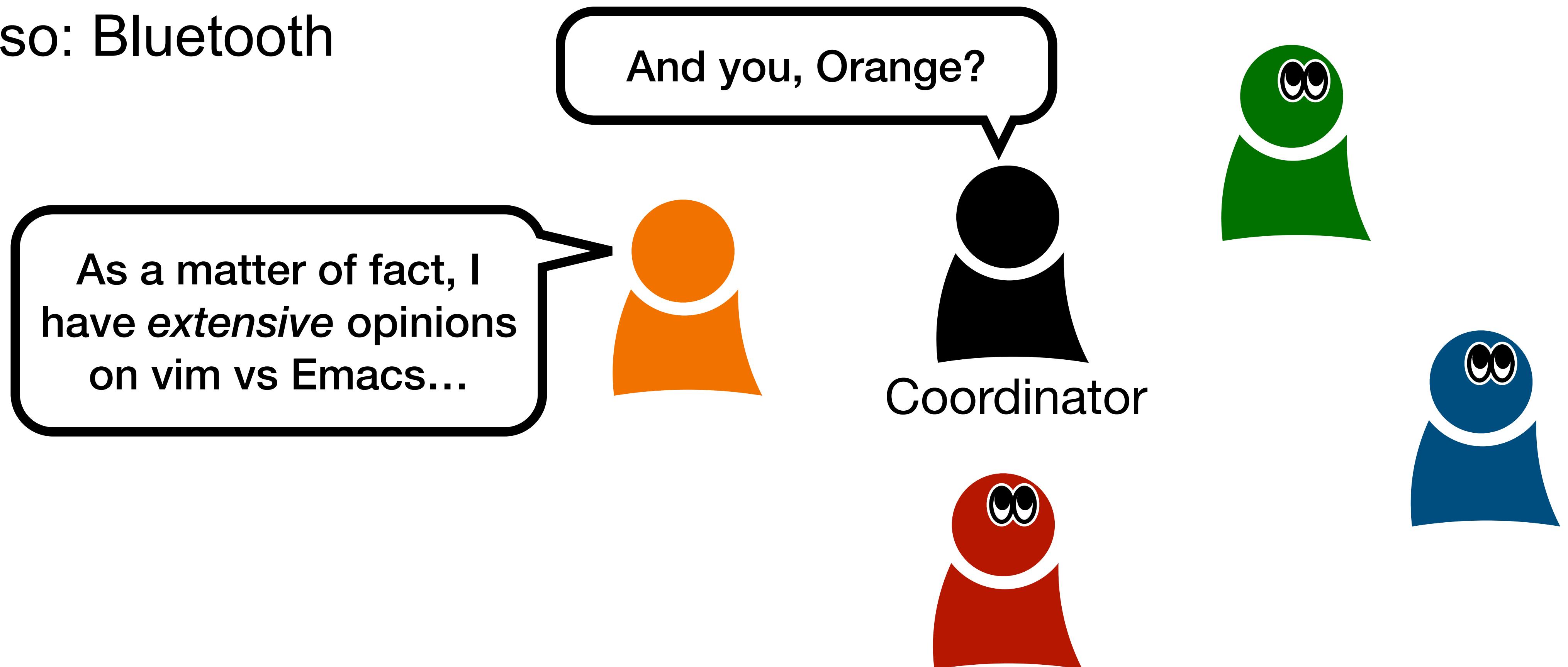
- Polling protocols
  - A coordinator decides who gets to speak when
  - Like congress: “The Chair recognizes the Senator from California...”
  - Also: Bluetooth



# Turn-Taking schemes

---

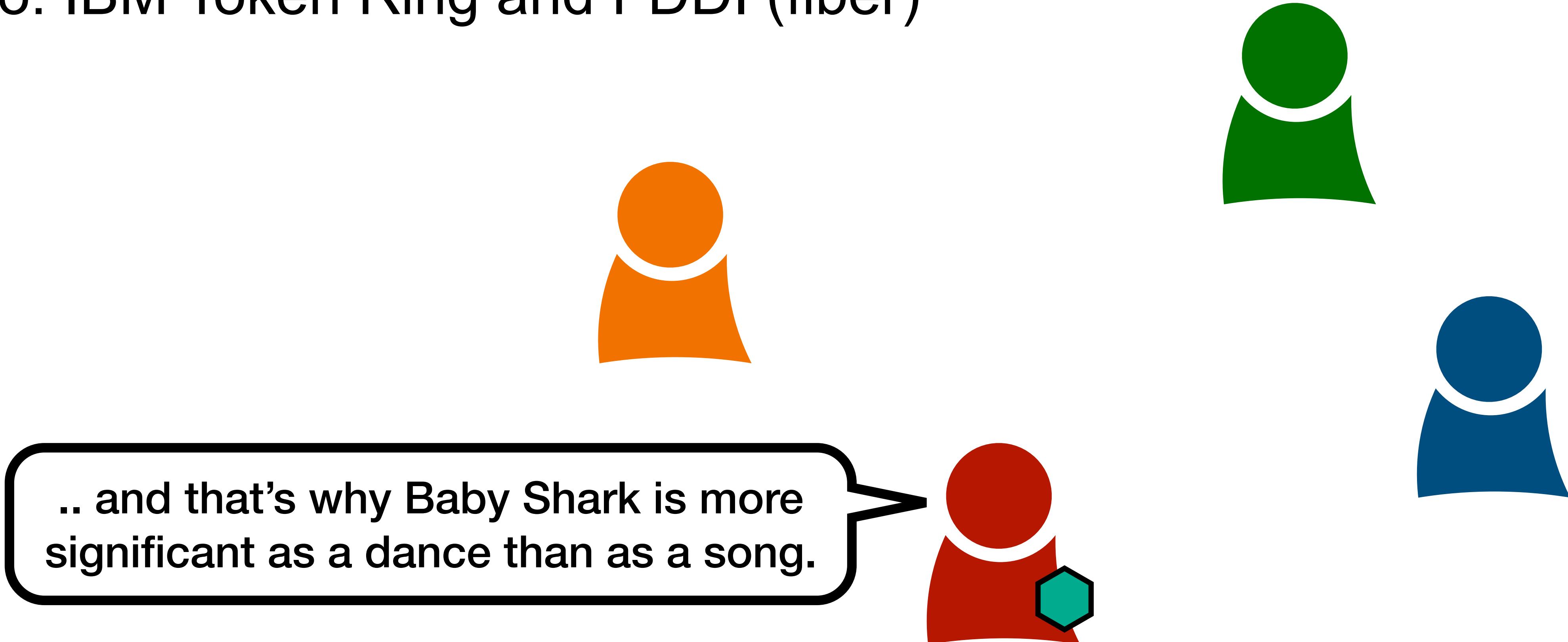
- Polling protocols
  - A coordinator decides who gets to speak when
  - Like congress: “The Chair recognizes the Senator from California...”
  - Also: Bluetooth



# Turn-Taking schemes

---

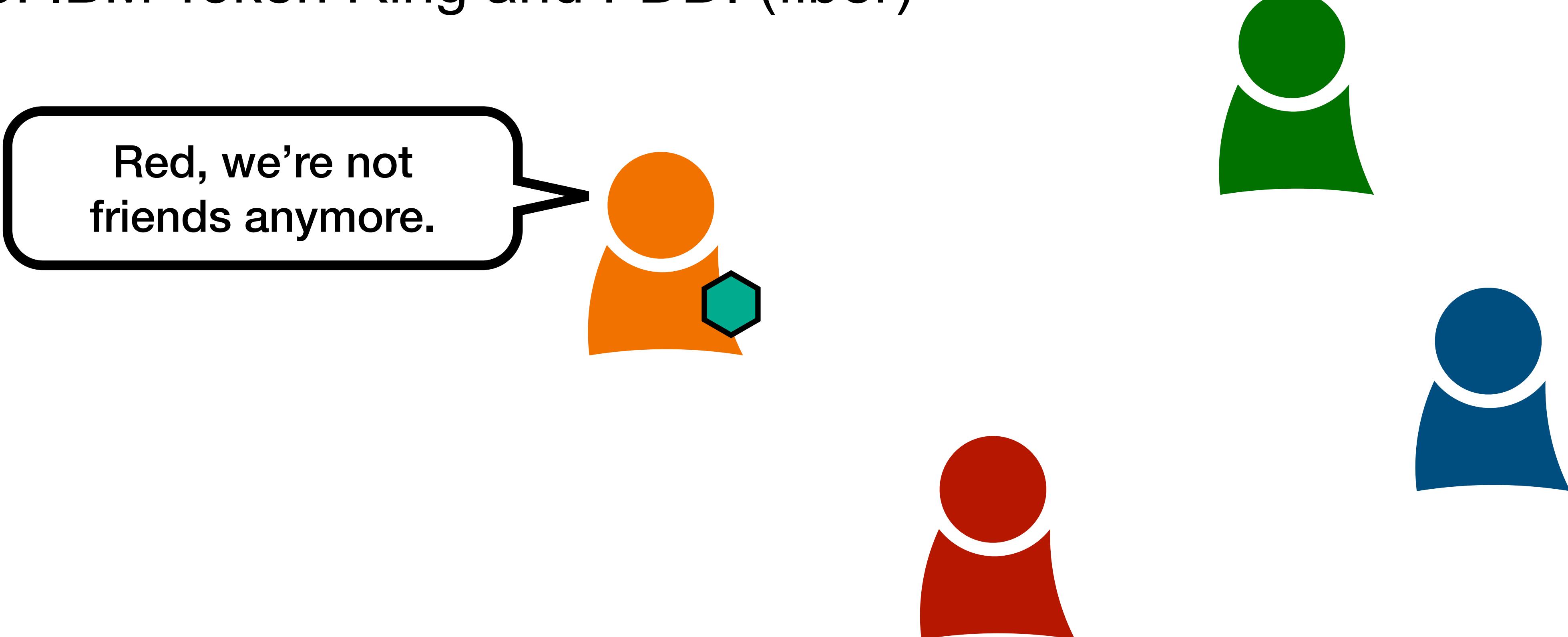
- Token-passing
  - Virtual “token” passed around, only holder can transmit
  - Like a “talking stick”
  - Also: IBM Token Ring and FDDI (fiber)



# Turn-Taking schemes

---

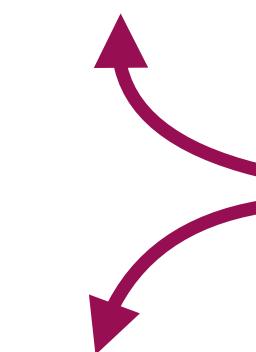
- Token-passing
  - Virtual “token” passed around, only holder can transmit
  - Like a “talking stick”
  - Also: IBM Token Ring and FDDI (fiber)



# Common Multiple Access Protocol approaches

---

- **Divide medium up by frequency** (*Frequency Division Multiplexing*)
  - Can be wasteful! Only so much EM spectrum to go around, and many frequencies likely to be idle often (traffic is bursty)
- **Divide medium up by time** (several ways)
  - Divide time into fixed-sized “slots”, each sender gets their own slot (*Time Division Multiplexing*); same drawback as FDM
  - Take turns
    - e.g., by *polling* or *token-passing*
  - Random access
    - Introduced by *ALOHA*
    - Also used by *CSMA*, *CSMA/CD*, ....

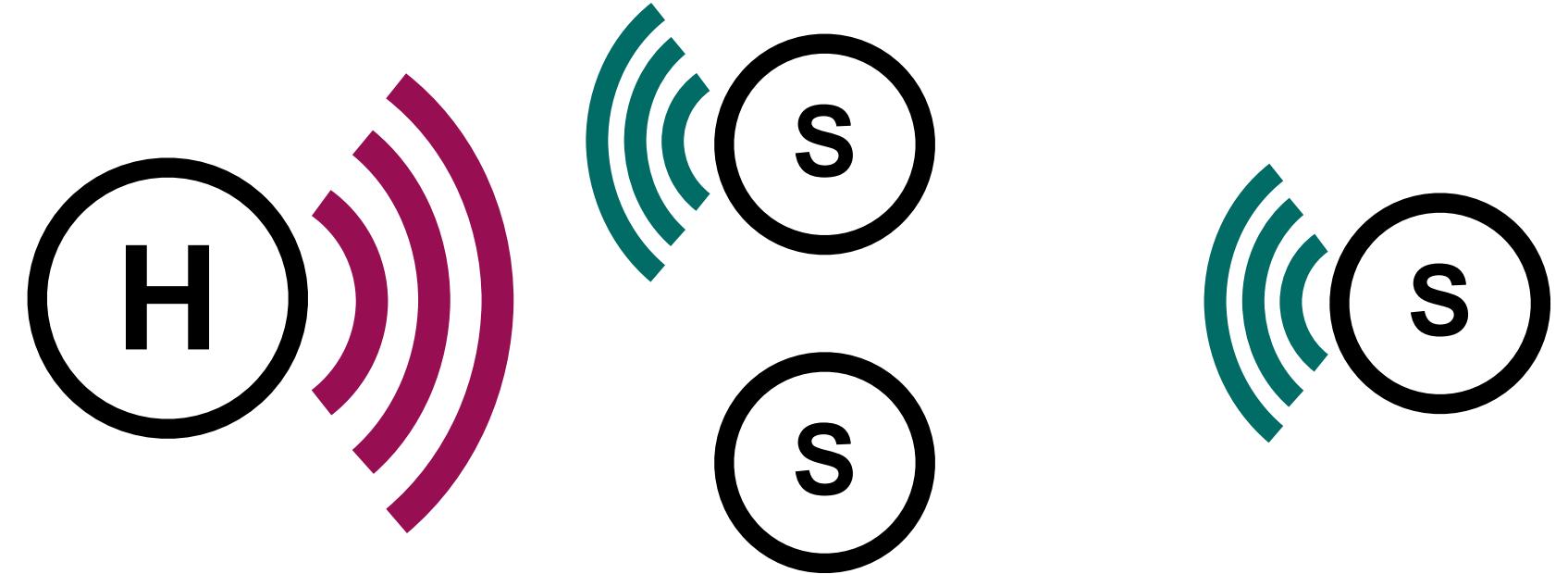


*Partitioning approaches*

# ALOHA<sub>net</sub>: Context

---

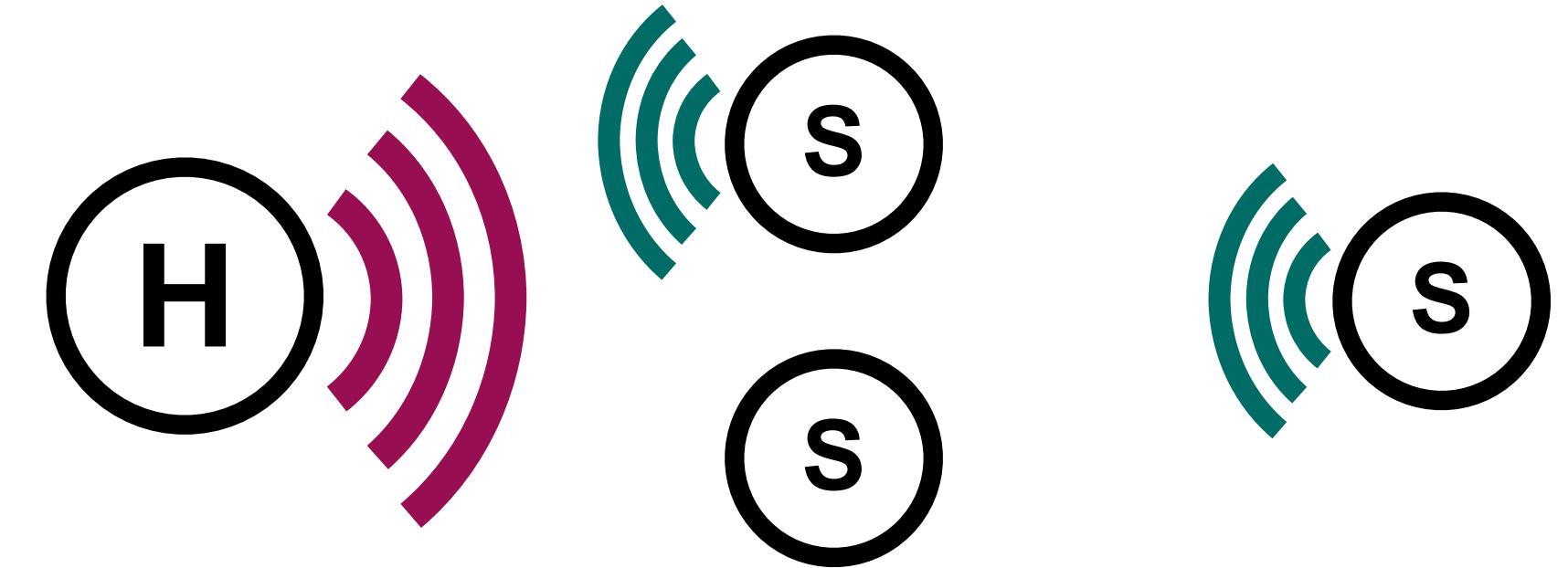
- “Hub” node on Oahu
- “Remote” nodes across Hawaii
- Two frequencies:
  - Hub transmits on its own frequency
    - Only one sender — no collisions
    - (All remotes listen to it)
  - All remote sites transmit on shared frequency
    - May collide — use random access scheme
    - (Only hub listens to it)



# ALOHA<sub>net</sub>: “Pure ALOHA” random access scheme

---

- If remote has a packet — just send it
  - No *a priori* coordination among remote sites
- When hub gets a packet — send ack
- If two remote sites transmitted at once, collision will have garbled packet...
  - .. hub will not send an ack!
- If remote does not get expected ack...
  - Wait a *random amount of time*
  - Then resend — probably won’t collide this time!
- .. it’s so simple!

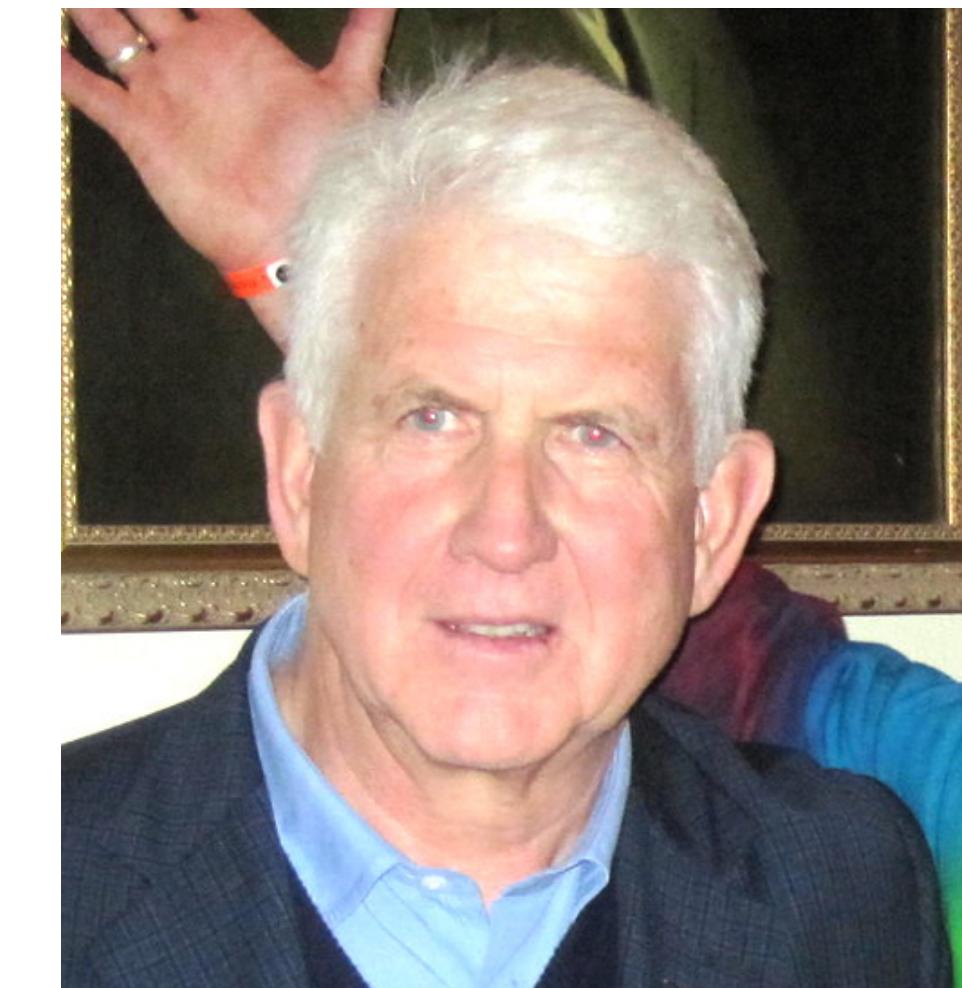


That's all great, but...  
aren't we supposed to be talking about  
**Ethernet**  
?

# From ALOHA to Ethernet

---

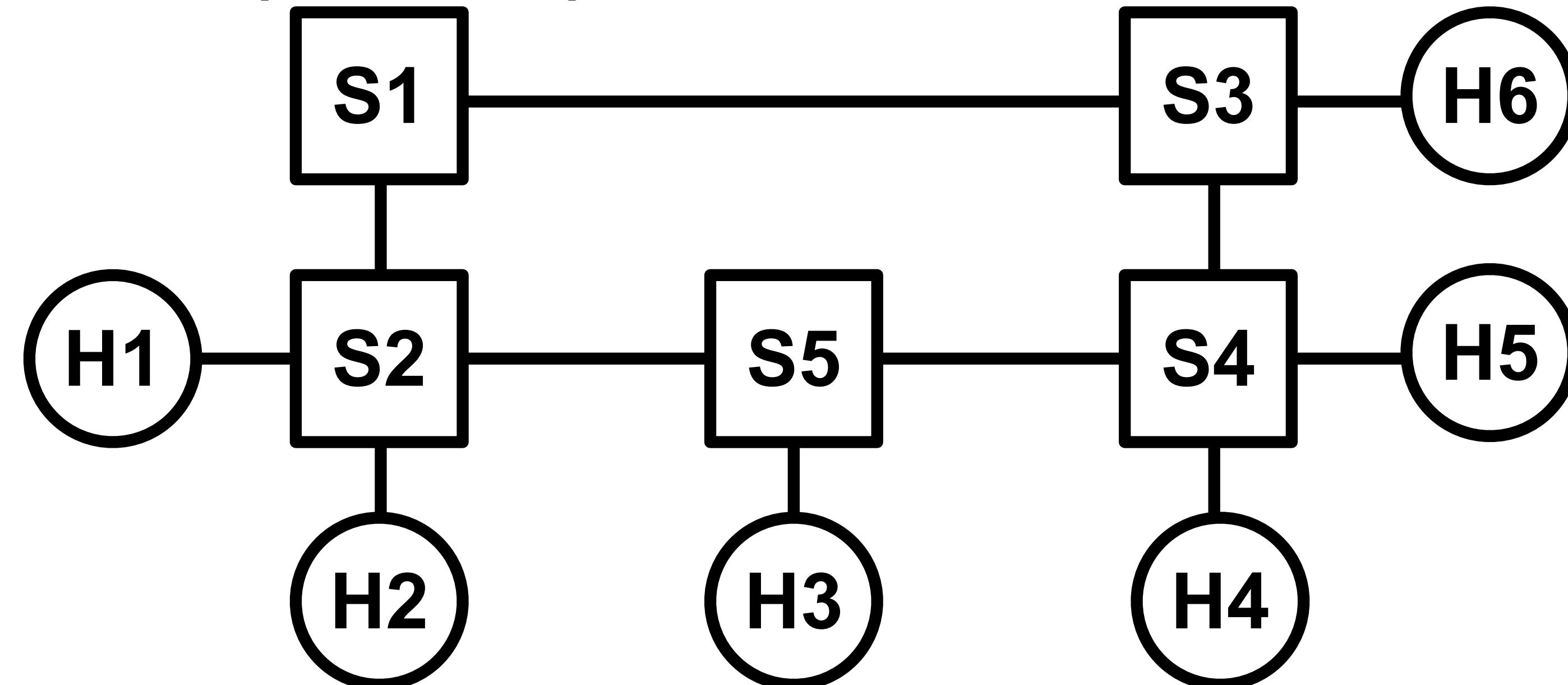
- Robert “Bob” Metcalfe worked at Xerox in 1972 while Xerox was developing the Xerox Alto computer
  - This was a totally groundbreaking computer — the first attempt at a “personal” computer or workstation
- When you’ve got tens (hundreds?!?) of computers in one building, how do you connect them all?!
- Didn’t want a centralized “rat’s nest” of cables in a wiring closet
- Wanted something “maximally distributed”... and cheap
- .. just run *one* two-conductor cable; connect *all* the computers to it!
  - .. a shared medium!
  - Part of his PhD thesis was on ALOHAnet — used similar ideas



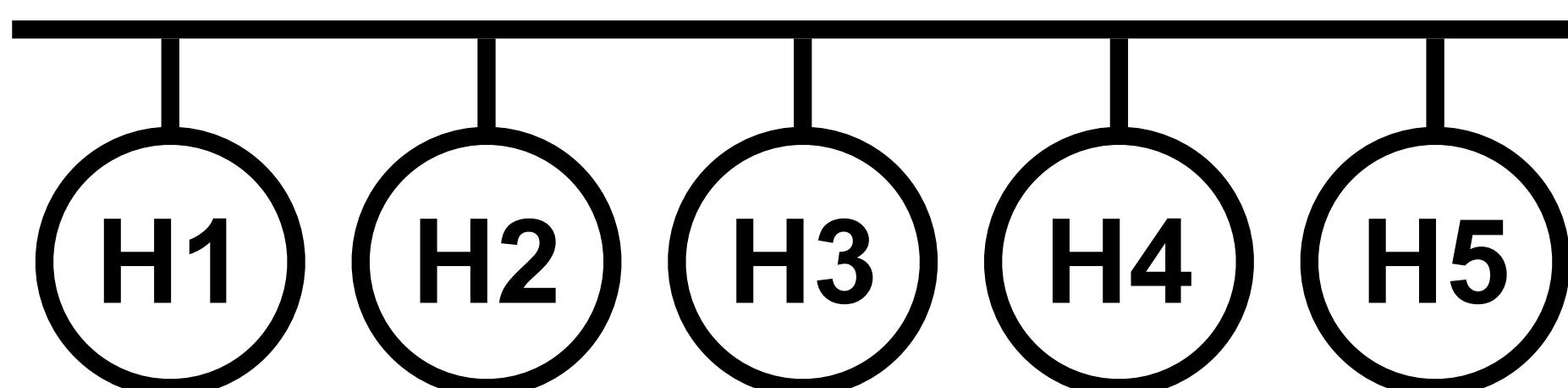
# Ethernet

---

- Early ARPANET (and almost everything we've looked at this semester) were all point-to-point links with switches:



- Bob Metcalfe's Ethernet looked like this:



# Ethernet: CSMA

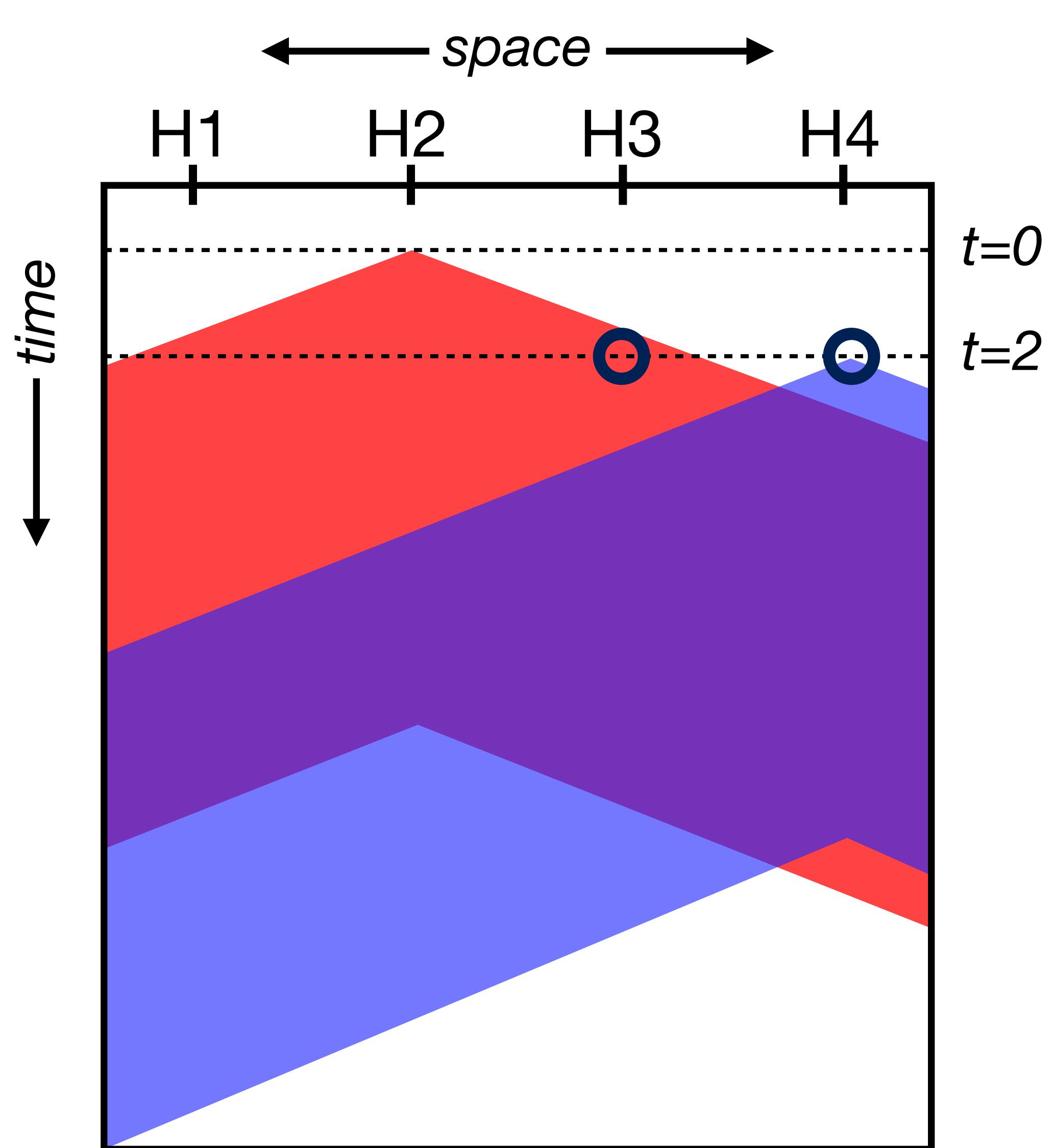
---

- Refined ALOHA multiple access protocol:
- *Carrier Sense Multiple Access - CSMA*
  - ALOHA is “rude” — nodes just start talking; figure out collisions later
  - CSMA is “polite” — listen first, start talking when it’s quiet
    - Listen = *sense* the signal (*carrier*)
  - .. this is a nice improvement but doesn’t completely avoid collisions
  - .. why not?
    - Propagation delay!

# Ethernet: CSMA and propagation delay

---

- At  $t=0$ ...
  - H2 transmits
    - Signal propagates as time goes by
- At  $t=2$ ...
  - H3 has heard it; won't transmit
  - H4 has no idea yet; starts transmitting
    - Signal propagates as time goes by
    - .. and collides with H2's signal!
- Solution: CSMA/CD



# Ethernet: CSMA/CD

---

- *Carrier Sense Multiple Access with Collision Detection (CSMA/CD)*
  - Listen *while* you talk
    - If you start hearing someone else while you're talking, shut up (Detect the collision)
    - Don't bother continuing to transmit the whole packet!
    - .. there's a bit more to it, but this is the basic idea

# Ethernet: A final word on retransmission

---

- After a collision, we wait a random amount of time and retransmit
- If link has high contention (many wanting to send), may keep colliding
- Use randomized *binary exponential backoff*
  - If retransmit after collision also collides, wait up to twice as long
  - Continue doubling for every subsequent collision
  - Retransmits fast when possible, slows down when necessary

# Ethernet: Summary so far

---

- Ethernet
  - Used a shared medium (coaxial cable)
  - .. with a random medium access protocol (CSMA/CD)
  - .. inspired by ALOHA
- Key ideas:
  - Carrier sense
    - **Listen before speaking, and don't interrupt**
    - Check if someone else is already sending data; waiting for them to finish
  - Collision detection
    - **If someone else starts talking at the same time, stop**
    - Realizing when two nodes are transmitting at once (detect data on wire is garbled)
  - Retransmission randomness
    - **Don't start talking again right away**
    - Waiting a random amount of time
  - Exponential backoff
    - **When link is highly contended, be increasingly conservative**
    - On subsequent collisions, upper bound of random wait gets longer and longer (doubles)

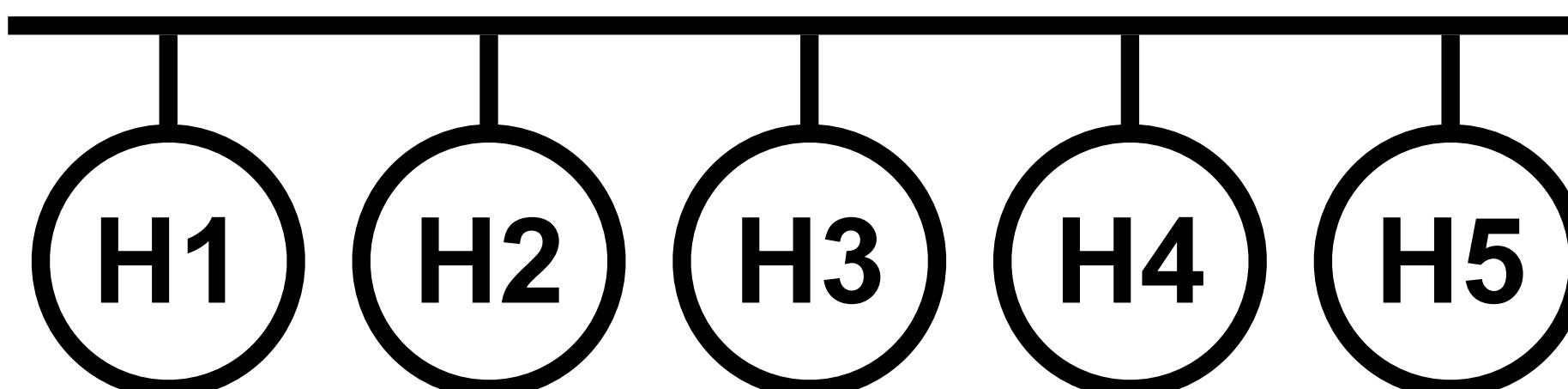
# Questions?

# Ethernet Addresses & Service Types

# Ethernet: Addresses and Service Types

---

- On this shared medium, if you transmit, everyone receives!
  - L2 “address” is not useful as a *locator*
    - .. but it’s useful as an *identifier*
  - We’ve used postal metaphor in this class
    - But there’s no need to find the right street or anything here
    - It’s like everyone is in the same room — you talk, they’ll hear
    - But you still need to say their name so they know who you mean
  - .. there’s no routing or aggregation here — *flat* addresses



# Ethernet: Addresses

---

- 48 bits
- Usually shown as six two-digit hex numbers with colons (or dashes)
- Typically *stored permanently in network interface hardware (“burned in”)*
  - Can often be overridden by software
  - Often found printed on the device
- Structure (simplified)
  - Two bits of flags (we won't discuss)
  - 22 bits identifying company/organization (e.g. device manufacturer)
  - 24 bits identifying device
- Usually supposed to be globally unique
  - Not because they're all reachable as in IP!
  - .. just because they're hardcoded and you don't know if they will be or not

# Ethernet: Service Types

---

- We've talked about two service types:
  - *Unicast* — send to one recipient
  - *Anycast* — send to any one member of a group

# Ethernet: Service Types

---

- We've talked about two service types:
  - *Unicast* — send to one recipient
  - *Anycast* — send to any one member of a group
- On classic Ethernet, it is trivial to support:
  - *Broadcast* — send to everyone

# Ethernet: Broadcast

---

- Broadcast — send to everyone
  - Specifically, we mean everyone in the specific Ethernet network
    - .. everyone on the same cable!
  - The packet already reaches them, they just need to listen!
  - Implemented using all-ones address:
    - FF:FF:FF:FF:FF:FF
  - In classic Ethernet, only really influences receiver
    - It's just listening to something besides just its normal address
    - Network itself behaves just the same

# Ethernet: Service Types

---

- We've talked about two service types:
  - *Unicast* — send to one recipient
  - *Anycast* — send to any one member of a group
- On classic Ethernet, it is trivial to support:
  - *Broadcast* — send to everyone
  - *Multicast* — send to all members of a group

# Ethernet: Multicast

---

- Multicast — send to all members of a group
  - Again, trivial on classic Ethernet
    - .. just a matter of whether you're listening for it or not
  - Implemented by setting one of the flags in address to 1:
    - 01:00:00:00:00:00 (the one here is the flag bit)
    - Thus, in all normal addresses, first byte is even
    - This is actually the first bit on the wire; bytes are sent low bit first
    - .. note that broadcast is really just a special case

# Ethernet

---

## Real-world multicast example

- Multicast  
• Aggregation  
• ...  
• Implications  
• Other  
• The  
• The  
• The  
• ...  
• Multicast
- How does a Mac know when there are things around to AirPlay to? Or network printers nearby?
  - They're all communicating via multicast using multicast Ethernet address 01:00:5E:00:00:FB !
- Your computer sends queries to that address (“I'm looking for printers!”).
  - Relevant devices are listening on that address and answer back (“I'm a printer named foo!”).
- These messages are formatted as DNS records (PTR, SRV, TXT).
  - But there's no central server! Each device responds when it sees a query relevant to it.
- (Windows does similar using 01:00:5E:00:00:FC.)

bit first

# Ethernet: Multicast

---

- Multicast — send to all members of a group
  - Again, trivial on classic Ethernet
    - .. just a matter of whether you're listening for it or not
  - Implemented by setting one of the flags in address to 1:
    - 01:00:00:00:00:00 (the one here is the flag bit)
    - Thus, in all normal addresses, first byte is even
    - This is actually the first bit on the wire; bytes are sent low bit first
    - .. note that broadcast is really just a special case
  - Multicast in IP is much more complex to implement!

# Ethernet: Service Types

---

- We've talked about two service types:
  - *Unicast* — send to one recipient
  - *Anycast* — send to any one member of a group
- On classic Ethernet, it is trivial to support:
  - *Broadcast* — send to everyone
  - *Multicast* — send to all members of a group
  - .. basically just a matter of receiver listening to broadcast/multicast addresses and not just their own address

# Ethernet: Service Types

---

- We've talked about two service types:
  - *Unicast* — send to one recipient
  - *Anycast* — send to any one member of a group
- On classic Ethernet, it is trivial to support:
  - *Broadcast* — send to everyone
  - *Multicast* — send to all members of a group

- .. basically  
addresses

Quiz!

Does Ethernet support unicast? (Yes)

Does Ethernet support anycast? (Not directly)

ast/multicast

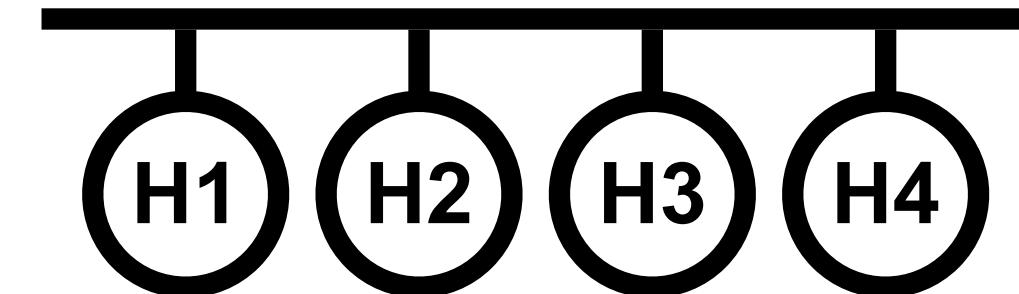
# Questions?

# Modern Ethernet

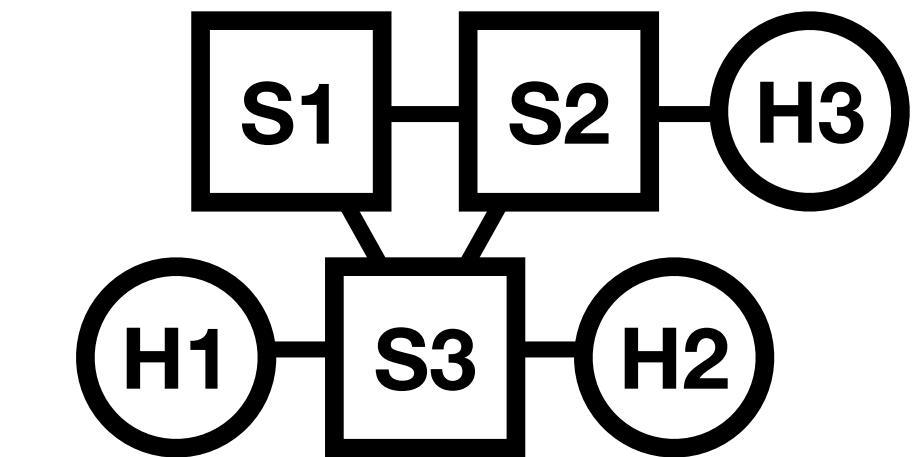
# Ethernet: From classic to modern

---

- I've been sort of hedging here, talking about "classic Ethernet"
  - Shared media with CSMA/CD



- Modern Ethernet rarely uses shared media — "switched Ethernet"
  - Links have exactly two nodes
    - *Nodes transmit on separate wires*
    - It's actually like two unidirectional links
    - No possibility of collision on a single link
  - And no collisions at switches; they queue packets from each link
  - But switched Ethernet still mostly *acts like* shared media Ethernet



# Ethernet: From classic to modern

---

- Classic Ethernet
  - Infrastructure is a single cable
  - You send a packet, and everyone gets it
- Switched Ethernet:
  - Essential primitive: **flooding**
  - You send a packet, and everyone gets it
  - Same basic model meant easy transition from single-cable Ethernet
    - No big new element required (e.g., address assignment, routing...)
  - Learning switches are just an optimization:
    - Once you learn where an address is, don't flood for that address

# Ethernet: From classic to modern

---

- Classic Ethernet
  - Infrastructure is a single cable
  - You send a packet, and everyone gets it
- Switched Ethernet:
  - Essential primitive: **flooding**
  - You send a packet, and everyone gets it

- Same as classic Ethernet?
  - No
- Learn how to support broadcast/multicast on switched Ethernet?
  - One way: Just flood it

## Quiz!

Broadcast/multicast on classic Ethernet:  
Just send the packet

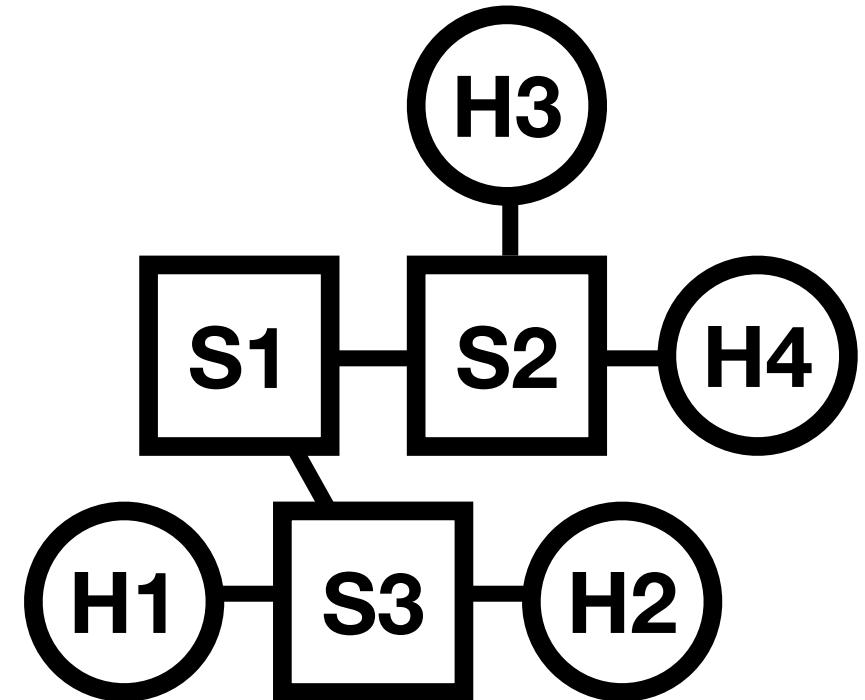
# Questions?

# The Interplay of L2 and L3

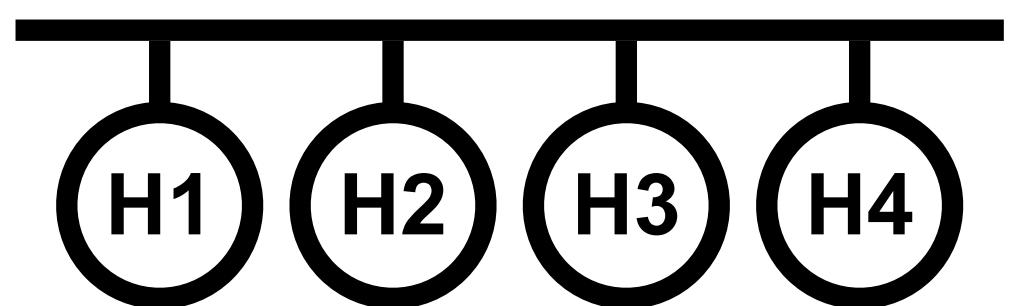
# A note on notation

---

- Super important note!
- If the switches in this diagram are all L2 switches...



- Then this network is logically equivalent to...

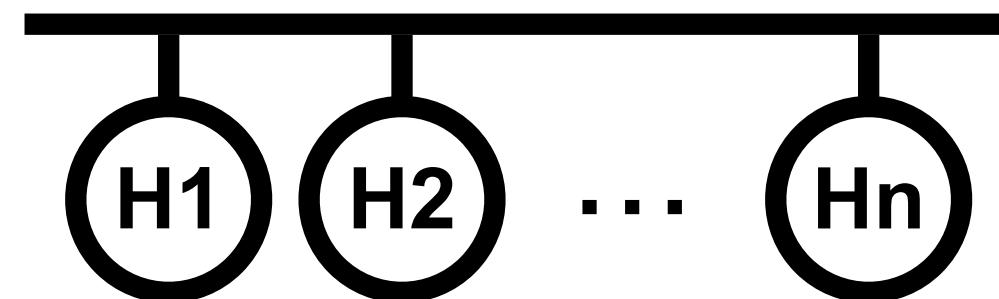


- Right???

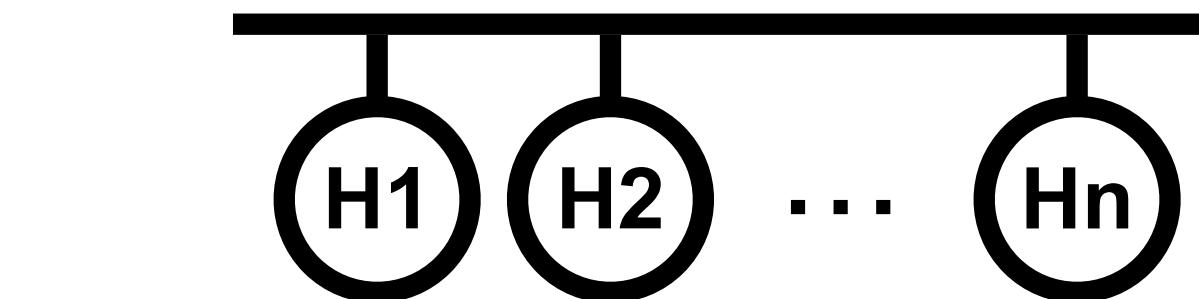
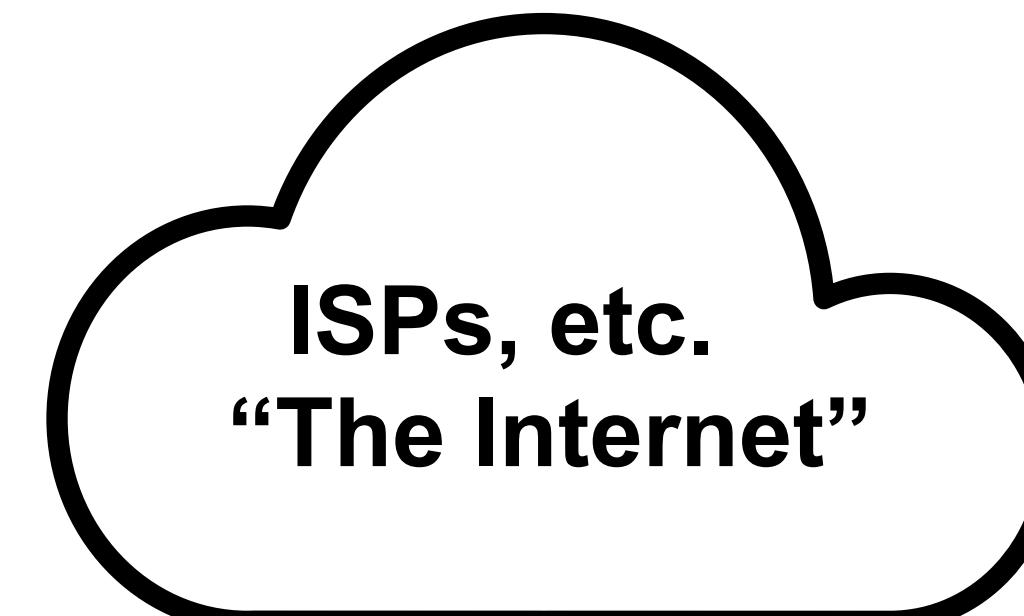
# L2 and L3 together

---

- Remember that IP is the *Internet Protocol*
  - Its purpose is to compose many networks into one Internet!
- What are those networks?
  - Many are local networks built with Ethernet! (Or some other L2)



Dunder Mifflin

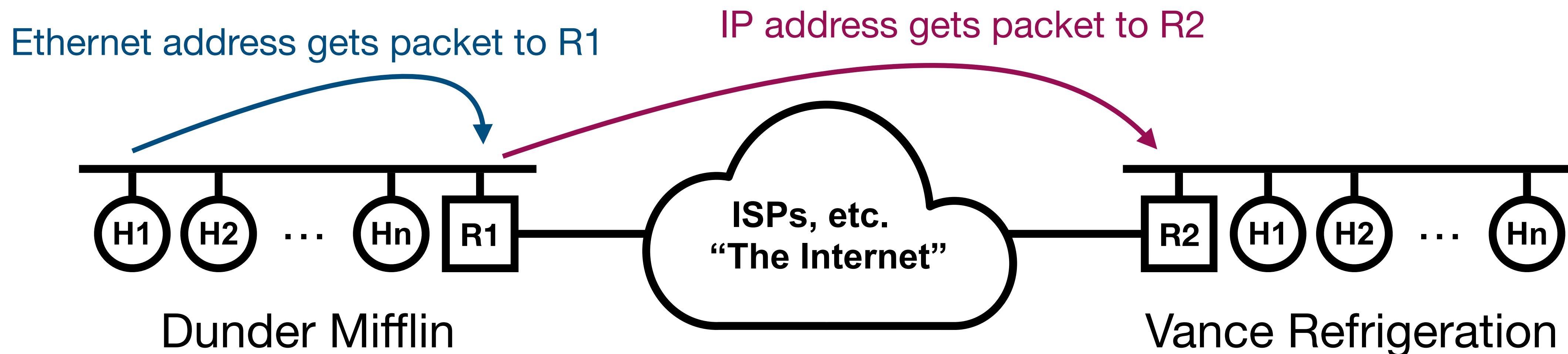


Vance Refrigeration

# L2 and L3 together

---

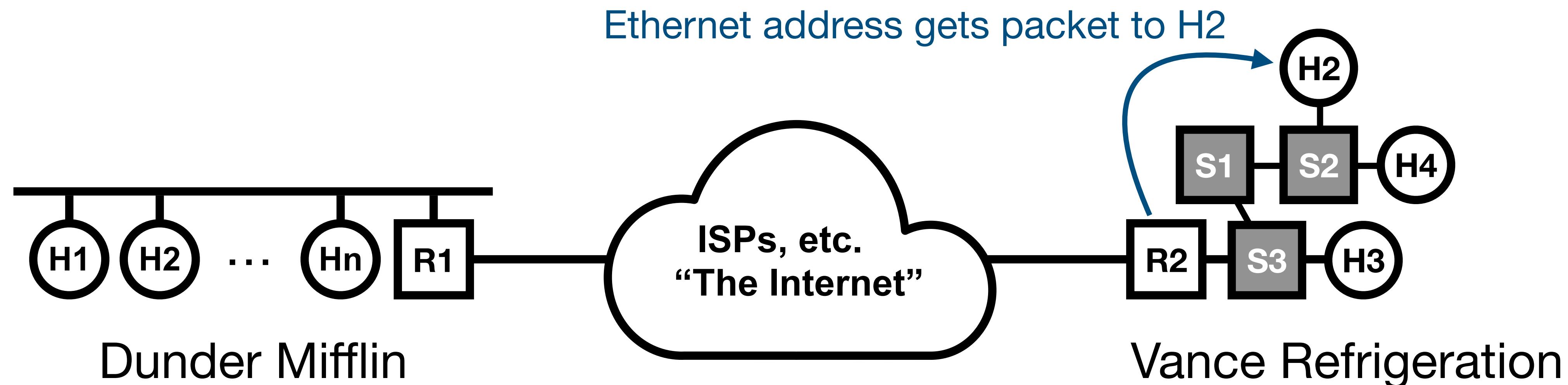
- Remember that IP is the *Internet Protocol*
  - Its purpose is to compose many networks into one Internet!
- What are those networks?
  - Many are local networks built with Ethernet! (Or some other L2)



# L2 and L3 together

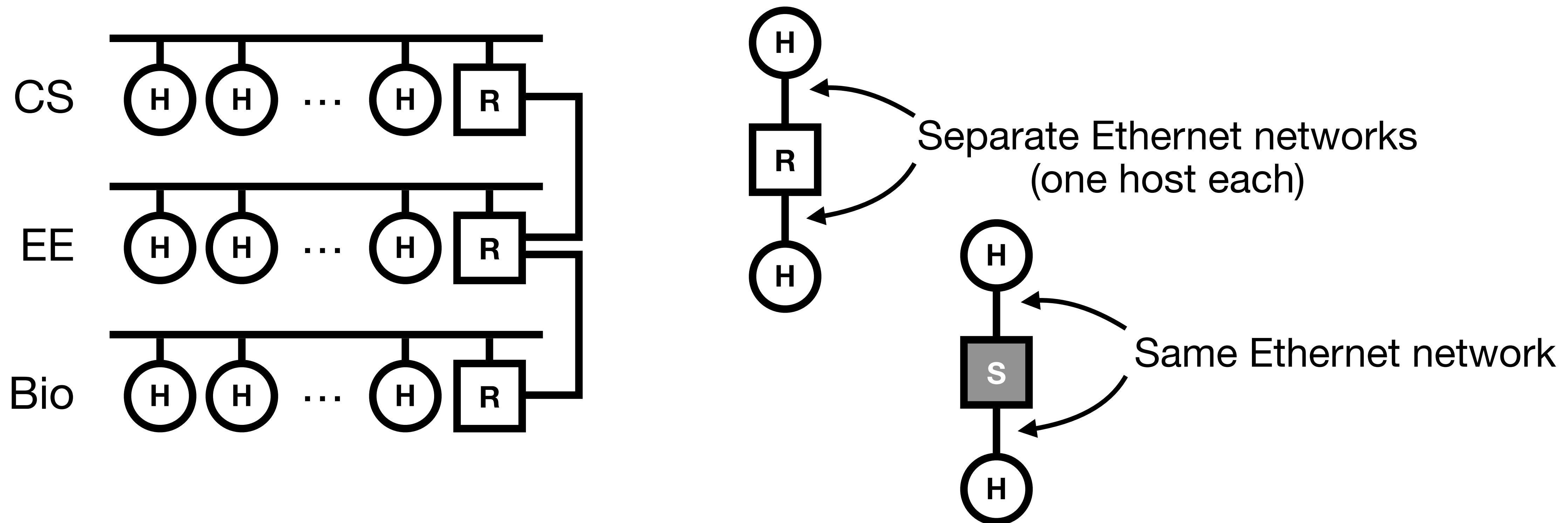
---

- Remember that IP is the *Internet Protocol*
  - Its purpose is to compose many networks into one Internet!
- What are those networks?
  - Many are local networks built with Ethernet! (Or some other L2)



# L2 and L3 together

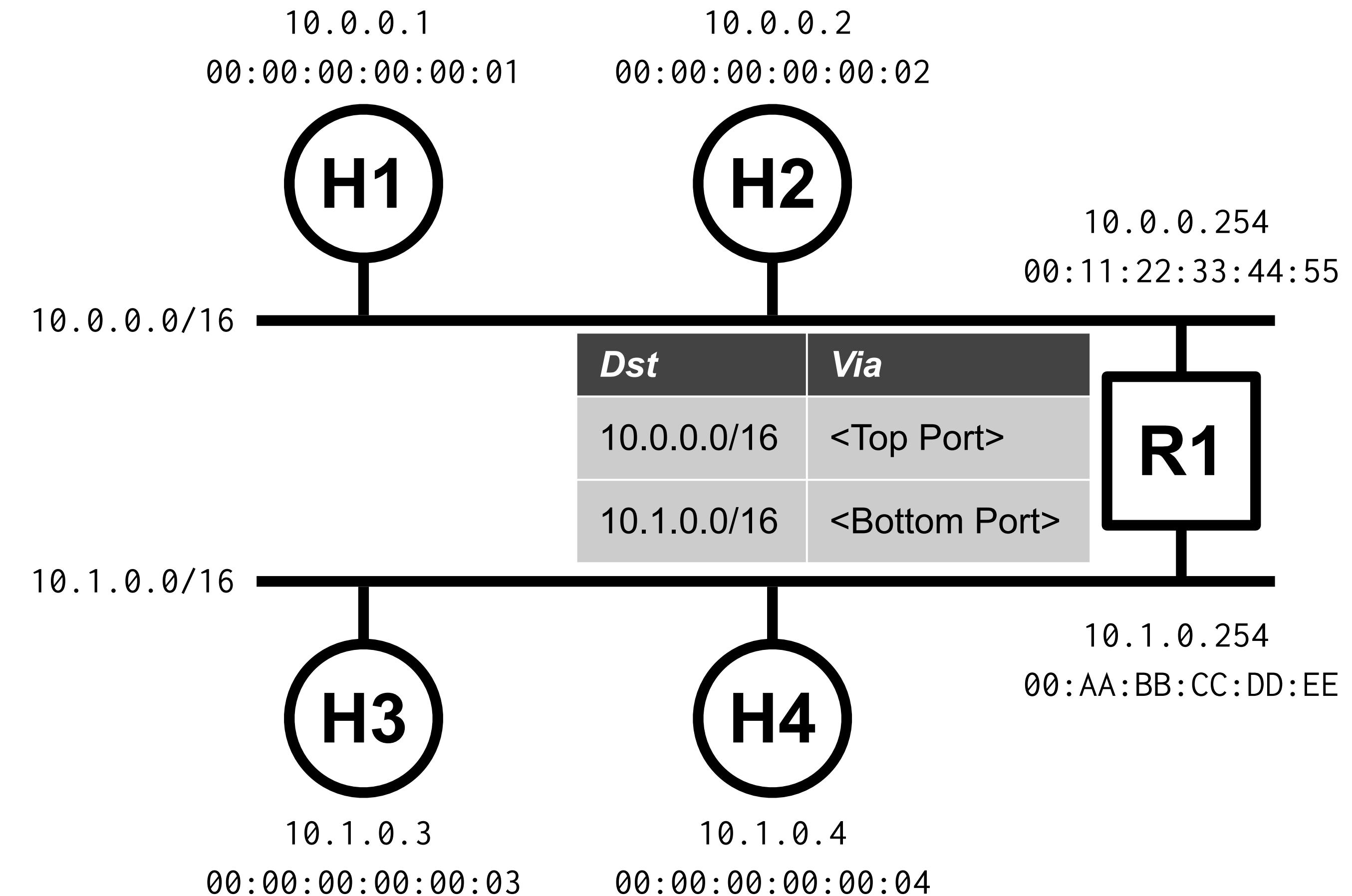
- Note: no reason you can't use IP routers to connect Ethernets in a private part of network (without going through public Internet)!
  - Note: no reason your Ethernet needs to have more than two nodes!



# Questions?

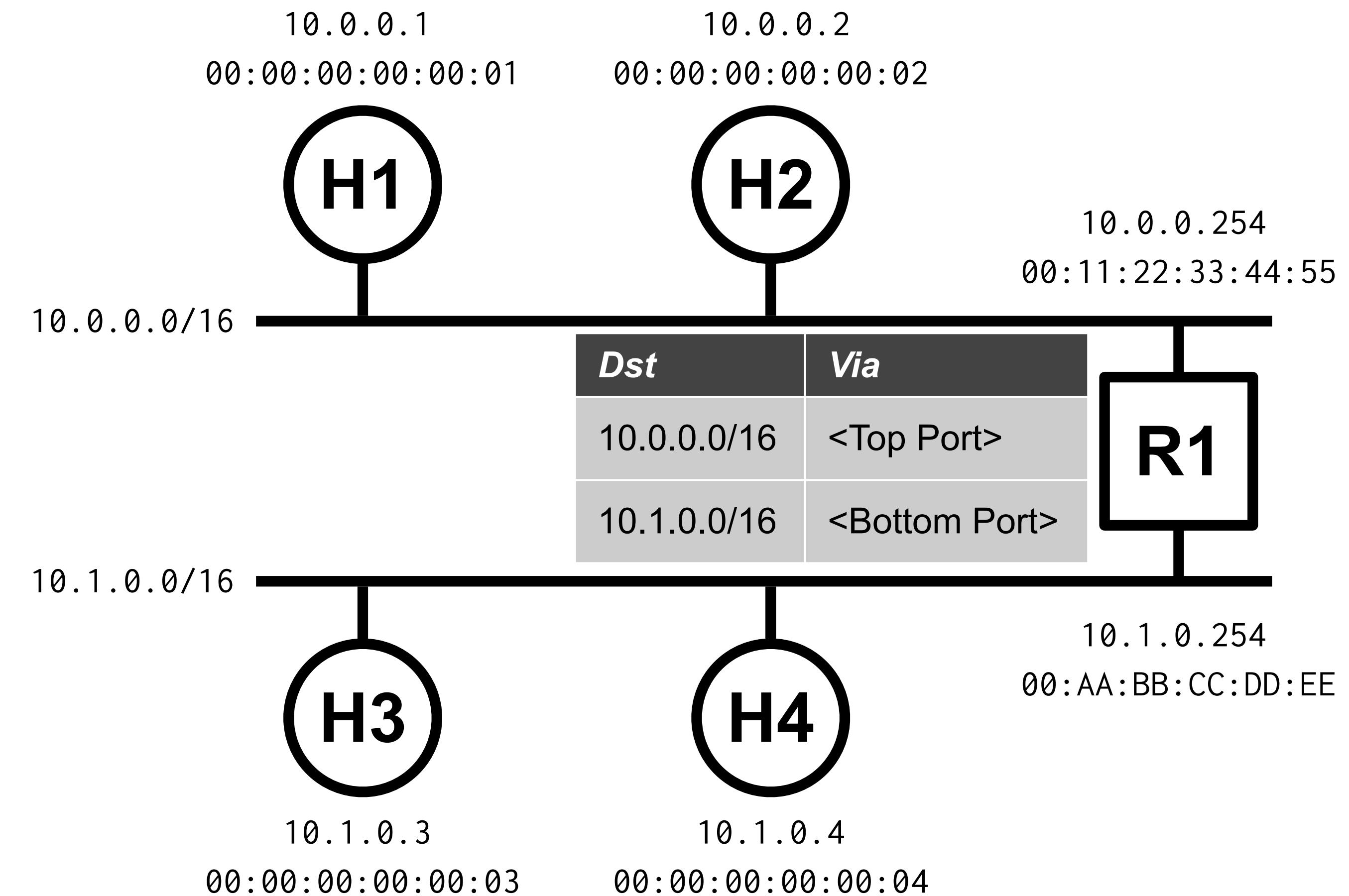
# L2 and L3 together: sending packets

- Two subnets connected by IP router
- Subnets use different IP prefixes
- IP table populated with static routes
- Router has appropriate IP address for each port
- Note: real Ethernet addresses would be very arbitrary!  
(Assigned by manufacturer)



# L2 and L3 together: sending packets

- Ex: H1 is sending an IP packet to H2
- They're on the same subnet, so H1 can just put the packet to 10.0.0.2 on the wire, and it'll get to H2
- Is it that easy? Missing something?
  - What Ethernet address should it use?
  - .. without right one, H2 will ignore it!
- Option 1: FF-FF-FF-FF-FF-FF
  - Doesn't allow learned paths
  - Annoys other nodes on network
  - Doesn't always work!
- Option 2: 00-00-00-00-00-02
  - But how do we find that?
  - ARP!



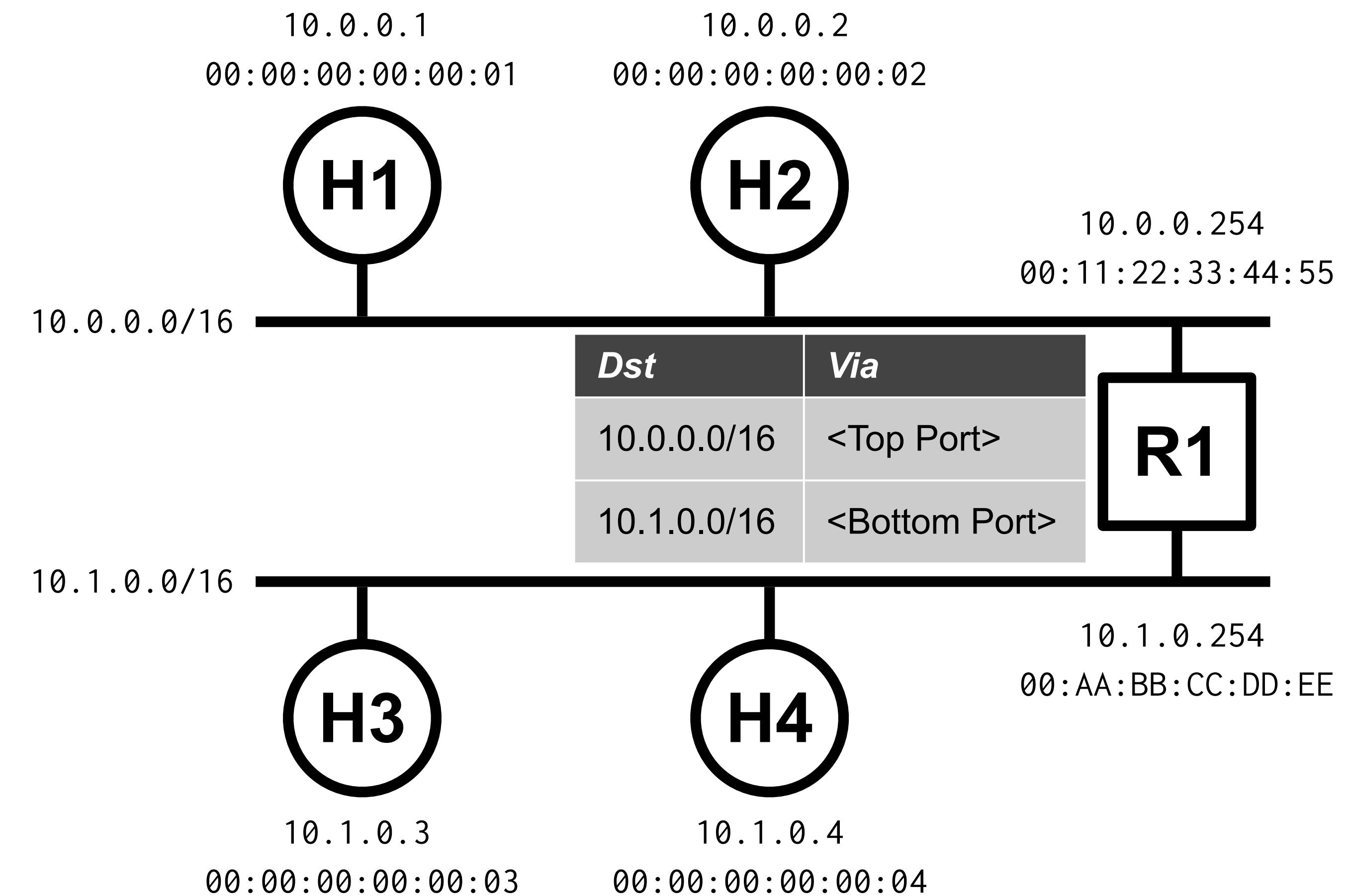
# ARP: the Address Resolution Protocol

---

- Given an IP address, want to know corresponding Ethernet address
- ARP runs directly atop L2 (not part of IP!)
- Host *broadcasts* query:
  - **Who has IP address w.x.y.z?**
- Host with address w.x.y.z hears query and responds (unicast):
  - **I am w.x.y.z, and my Ethernet address is a1:b2:c3:d4:e5.**
- Hosts cache results in “ARP table” / “neighbor table”
  - Refresh occasionally (resend queries)

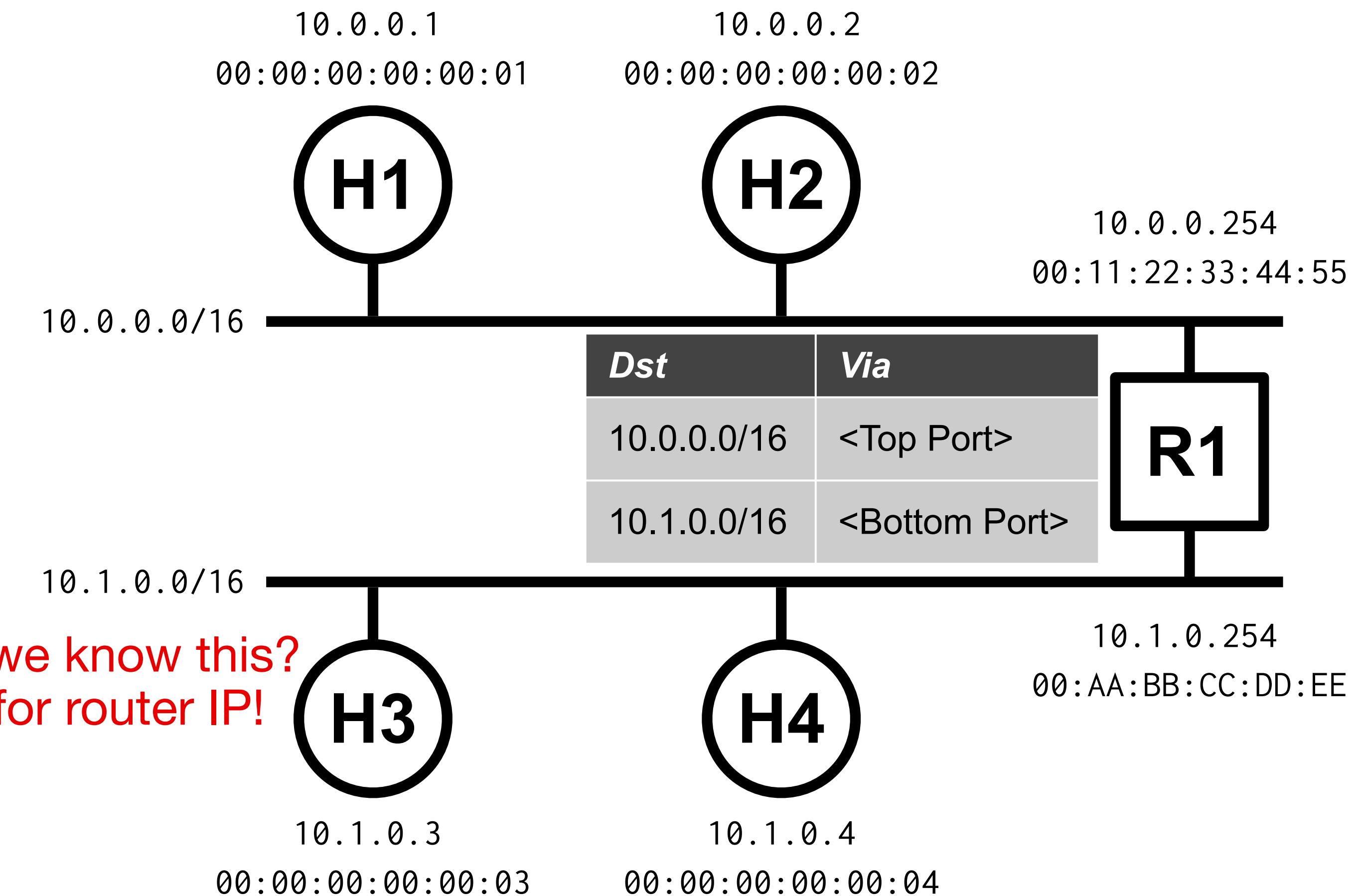
# L2 and L3 together: sending packets

- Ex: H1 is sending an IP packet to H2
- They're on the same subnet, so H1 can just put the packet to 10.0.0.2 on the wire, and it'll get to H2
- Use ARP to find Ethernet address
- How do we know H2 is on same subnet?
  - Check netmask/prefix:
    - $10.0.0.0/16 = 10.0.0.0/255.255.0.0$
    - $(10.0.0.2 \& 255.255.0.0)$   
=  
 $(10.0.0.1 \& 255.255.0.0)$
- How did we know our netmask?
  - Hold that thought...



# L2 and L3 together: sending packets

- Ex: H1 is sending an IP packet to H3
- Not on the same subnet
  - We must be sending via a router!
  - Assume host knows router's IP
- Packet headers when H1 sends it...
  - src IP: 10.0.0.1
  - src Eth: 00:00:00:00:00:01
  - dst IP: 10.1.0.3
  - dst Eth: 00:11:22:33:44:55
- Packet headers when R1 sends it...
  - src IP: 10.0.0.1
  - src Eth: 00:AA:BB:CC:DD:EE
  - dst IP: 10.1.0.3
  - dst Eth: 00:00:00:00:00:03



# L2 and L3 together: things a host must know...

---

- Its own IP address
- Subnet mask (network size) of directly attached network
  - So that we know if another host is directly reachable (at L2) or needs to be reached via router
- IP address of router
  - We didn't need this directly...
  - .. but we used it to get Ethernet address of router
- We're about to discuss how we know all this, but first...

# Questions?

# DHCP

How to know the things you need to know.

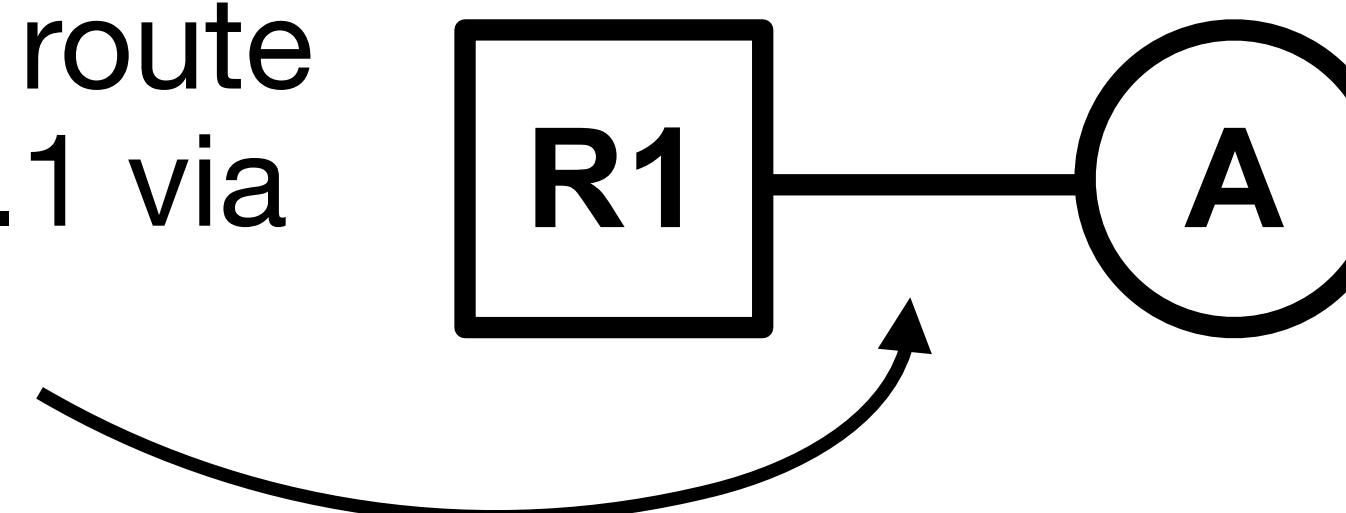
(Assuming you're a host.)

# IP Addresses

---

- The source of “ground truth” for Ethernet addresses is that addresses are burned into the hardware!
  - **Switch state / routing adapts to hosts (learning).**
- What’s the source of ground truth for IP addresses?
  - Answer 1: Static routes on routers (from network designer/operator)
  - Answer 2: Allocation of addresses from a registrars, e.g., ARIN)
  - **Hosts must adapt to switch state / routing / network authority.**

R1 has static route  
to 192.168.1.1 via  
this link



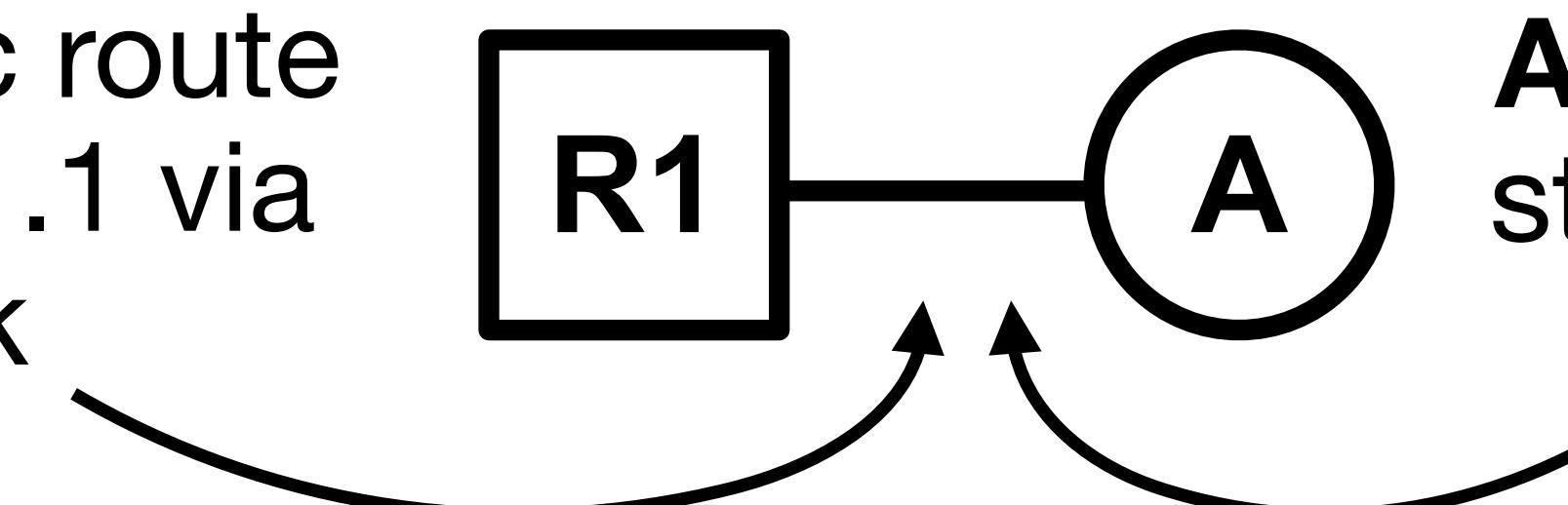
But how does A  
know that it is  
192.168.1.1 ?!

# IP Addresses

---

- Possible solution 1:
  - Manual — statically assign address to hosts
  - Static works well for networks (that don't move/change much)
  - Static worked fine for hosts when computers were big and few
  - .. works less well today
    - Discounting COVID-19, we often move our hosts around several times per day!
  - Doing it manually would be a pain!

R1 has static route  
to 192.168.1.1 via  
this link



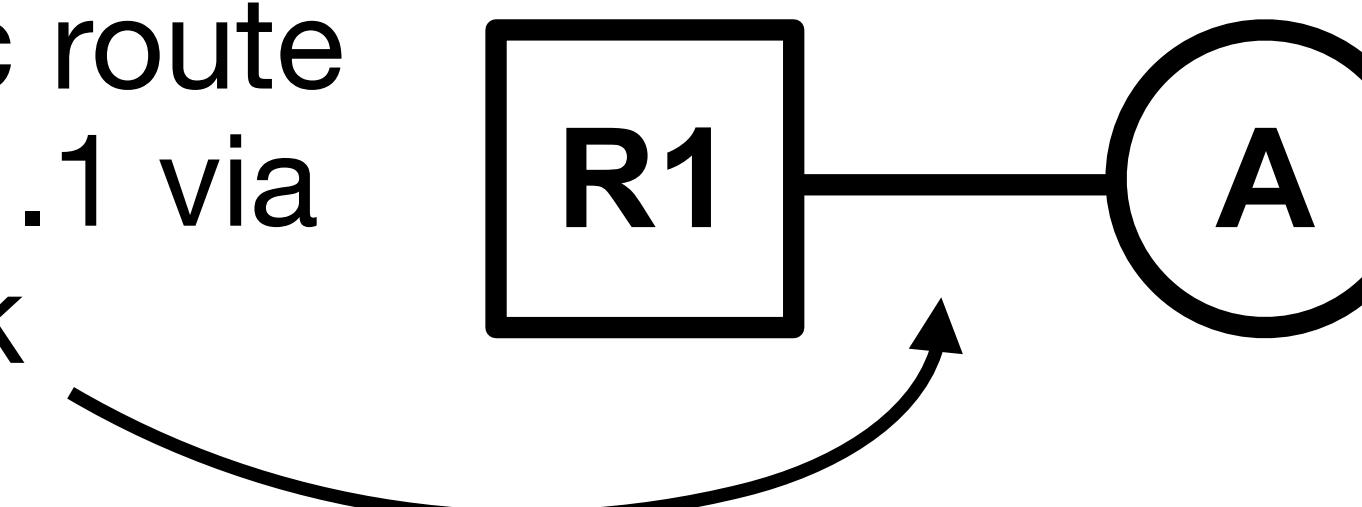
A has 192.168.1.1  
statically assigned  
for this link

# IP Addresses

---

- Possible solution 2:
  - Observation: “The network” already knows valid addresses
    - .. operators got the block of addresses from ARIN or whoever
    - .. operators configured routers with those addresses
  - So... design a protocol so that the network can tell the hosts!
  - DHCP!

R1 has static route  
to 192.168.1.1 via  
this link



A queries network  
& **something** tells it  
address is 192.168.1.1

# IP Addresses: DHCP

---

- DHCP is the *Dynamic Host Configuration Protocol*
- Provides a way for hosts to query “the network” for local configuration information
- Crucial IP configuration stuff:
  - **IP address**
  - **Netmask**
  - “Default gateway” = **first hop router**
- Also important:
  - **Local DNS resolving server**
- Much less important: lots of other assorted stuff (all optional)

Exactly the three things we said we wanted to know a moment ago

# DHCP

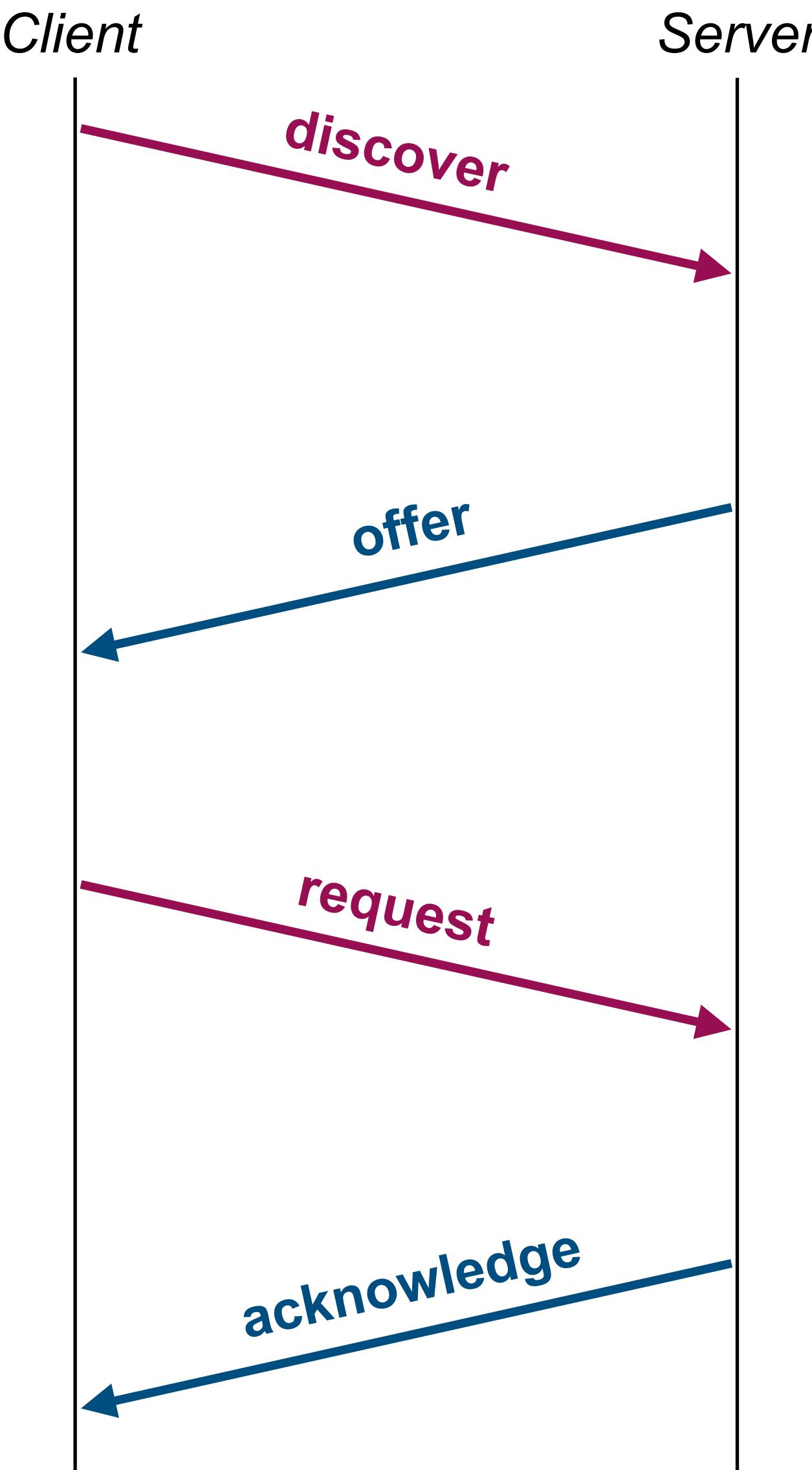
---

- One or more *DHCP servers* are added to network
  - Can be separate machine or built into a router (e.g., your wifi router)
  - Listen on well-known UDP port 67
  - Configured with required information
    - First hop router address, local DNS server
    - A *pool* of usable IP addresses
  - Servers *lease* hosts an IP address
    - Only valid for a limited time (often hours or a day)
    - Host must renew if it wants to keep it
    - Server won't offer it to another host if it's currently leased!

# DHCP

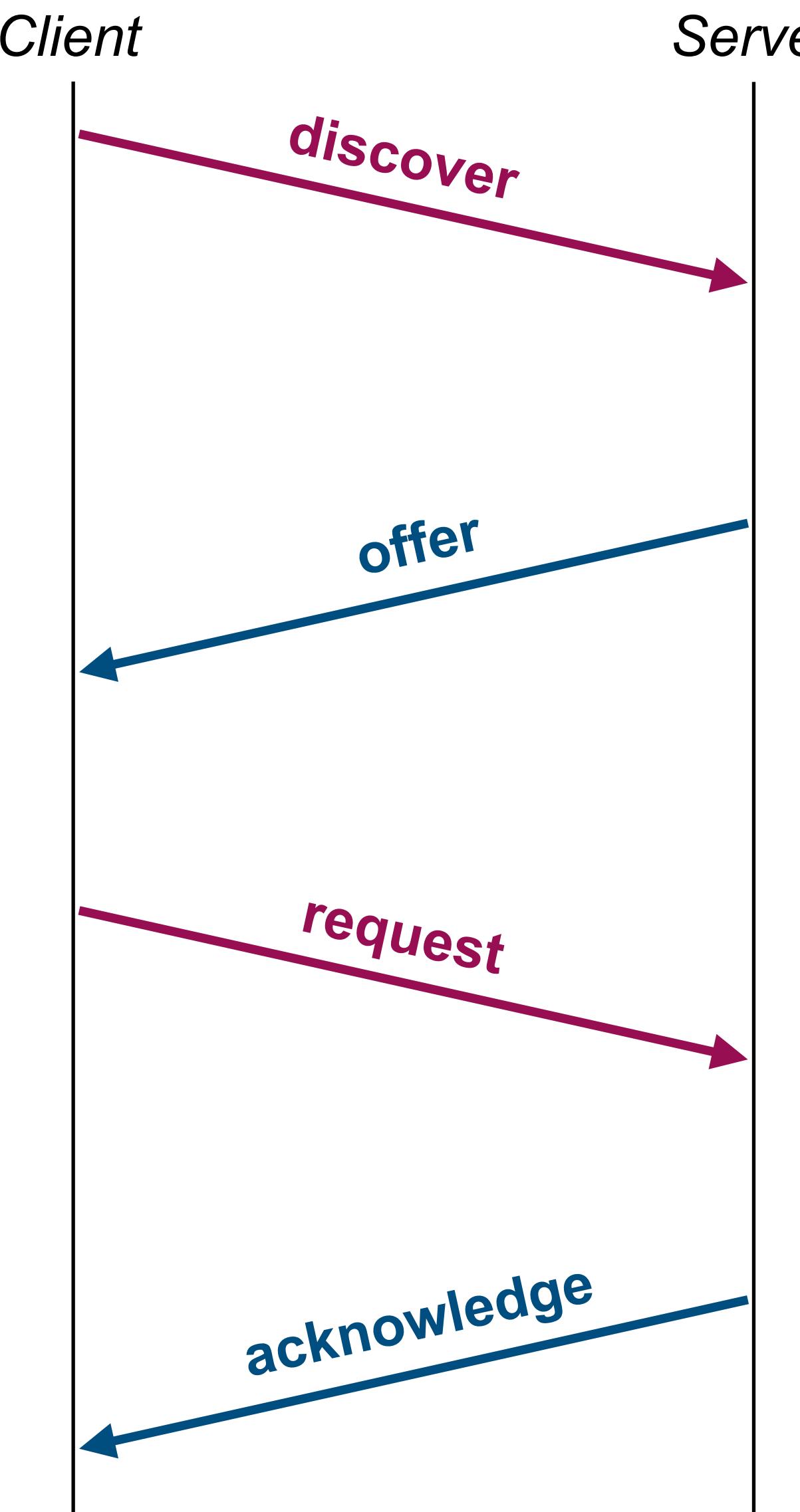
---

- Client sends a **discover message** — asks for config info
- Server(s) send(s) **offer message** with config info (e.g., particular IP)
- Client sends **request message** to accept a particular offer
- Server sends **acknowledge message** to confirm request granted



# DHCP

- Client sends a **discover message** — asks for config info
- Server(s) send(s) **offer message** with config info (e.g., particular IP)
- Client sends **request message** to accept a particular offer
- Server sends **acknowledge message** to confirm request granted



- DHCP built on UDP (built on IP)...
- How does client know server IP?
  - It might not!
  - What do you do about that?
  - **Broadcast** messages to it
  - Eth/L2 - FF:FF:FF:FF:FF
  - IP/L3 - 255.255.255.255
- What IP does server use for client?
  - Client doesn't have one yet!
  - **Broadcast** messages to it
- Source IP in packets from client?
  - 0.0.0.0

# DHCP

- Client sends a **discover message** — asks for config info



- Server(s) send **offer message** with (e.g., particular)

- Client sends **request message** to a particular offer

- Server sends **acknowledgment** to confirm request granted

- DHCP built on UDP (built on IP)...

new server IP?

out that?

ges to it

FF : FF : FF

5. 255

use for client?

one yet!

ges to it

from client?

## Quiz!

Q: What does broadcasting imply about the location of the DHCP server?

A: It's got to be available on the L2 network (within “broadcast range” of the client)!

Broadcast doesn't generally extend beyond that!

*DHCP relays* (generally part of a router) can do special forwarding across L2 networks if necessary.

• 0.0.0.0

# Questions?

# DHCP

---

- Final DHCP question:
  - Why doesn't DHCP just give us the router's Ethernet address?
  - .. did we actually need the router's IP address?
- It's cleaner — IP configuration all in terms of IP
  - There must be some mechanism for mapping from L3 to L2 addr
    - Just use it, whatever it is
    - Means that DHCP (and IP config in general) is the same even when used with different L2s

**Everything together now!**  
We've been building toward  
this all semester!

# First a quick recap...

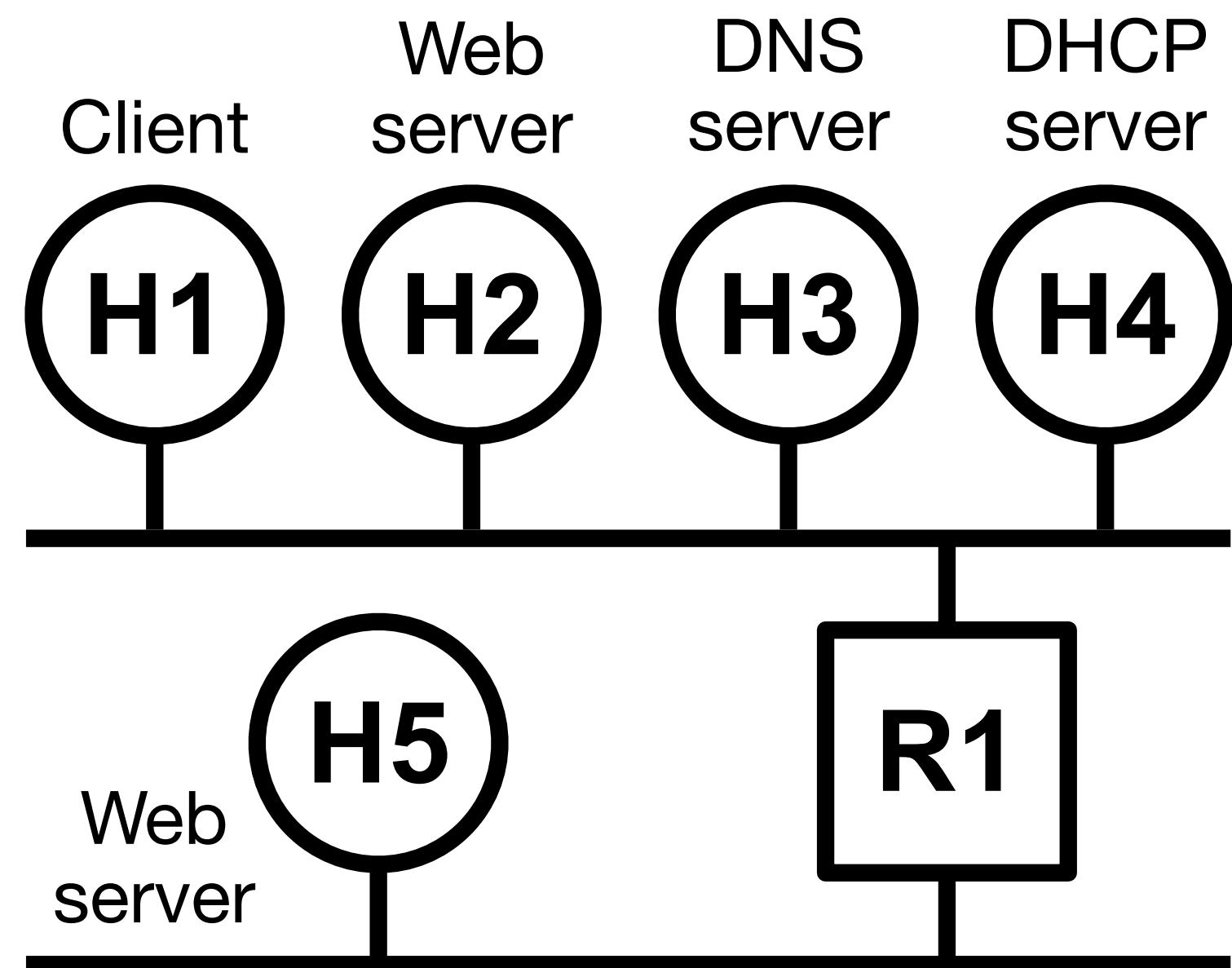
---

- Hosts know their Ethernet address...
  - .. because it's *burnt in* to hardware
- Hosts know their IP address...
  - .. via DHCP
- Hosts learn mapping from IP to Ethernet addresses...
  - .. via ARP
- Other things you learn from DHCP...
  - Subnet mask
  - First hop router IP address
  - Local DNS resolving server
- DHCP and ARP use a lot of *broadcast*
  - Scalability is okay, because only broadcasts to local L2 network
  - Solves chicken/egg addressing problems (i.e., don't know who ask so ask everyone)

# The Setup

---

- Scenario:
  - Two subnets connected by router R1
  - Host H1...
    - Boots up (all state cleared)
    - Fetches a small file from H5.com
    - Goes idle for five minutes
    - Fetches two small files from H2.com
- The Task:
  - List (in order) the packets H1 sends/receives



# Assumptions

---

- HTTP uses persistent connections
  - Browser times out after one minute
  - Server times out after two minutes
- HTTP requests/responses fit in single packets
- No TCP “piggybacking” — i.e., no data on returning ACKs (next slide)

# When to piggyback?

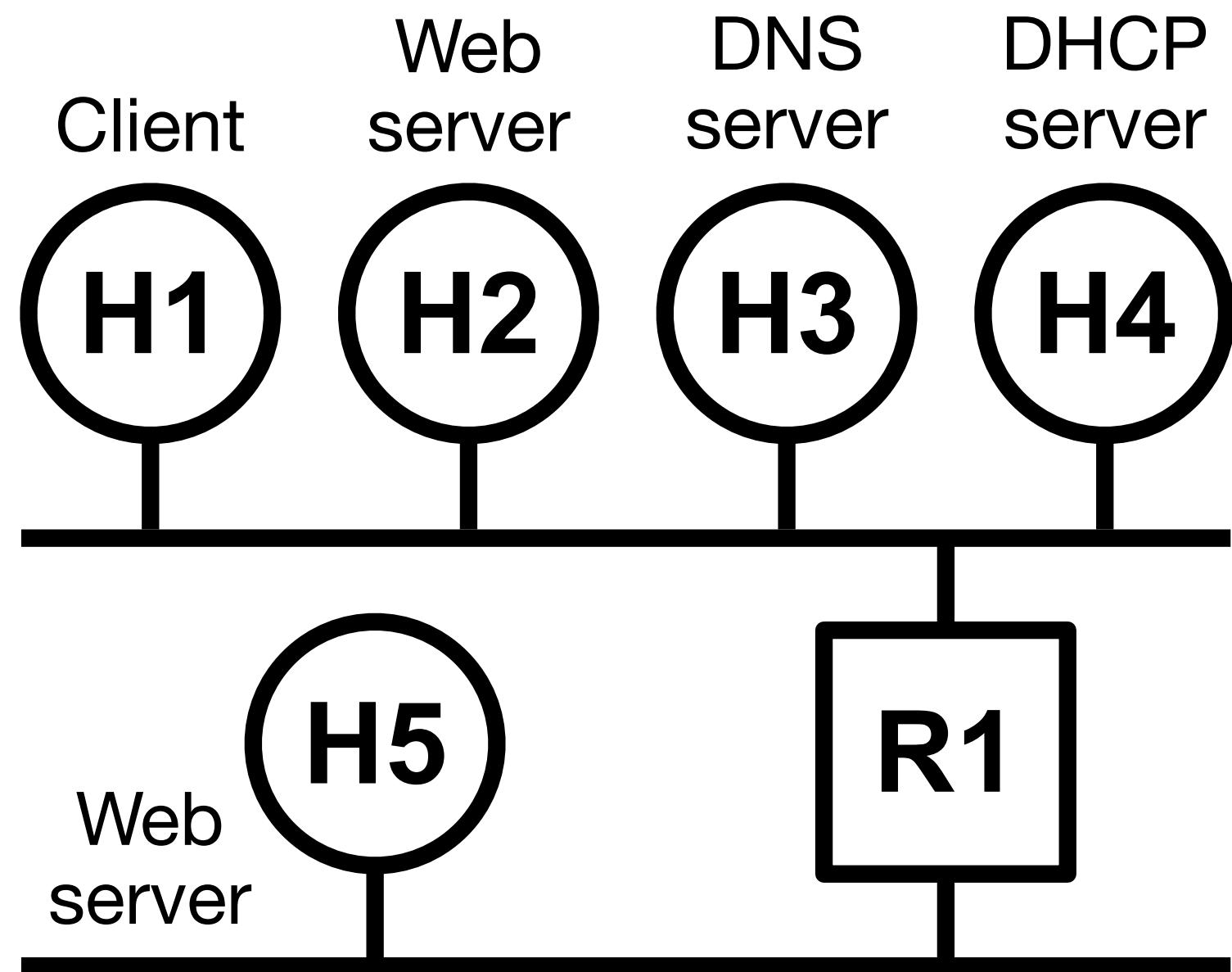
---

- TCP implementation is typically in kernel
  - Returning ACKs are generated in kernel
- Applications (HTTP and above) are in userspace
  - Application responses generated in userspace
- ACKs are often generated before application has chance to respond
  - Kernel creates ACK and *schedules application to run later*
  - Exception if kernel is delaying ACKs (which is a thing, but not in our example)
- Similar with TCP close:
  - Generally see FIN, ACK, FIN, ACK — not FIN, FIN+ACK, ACK
  - Generally, one application side sees other side close, then closes its side
- In what follows, do not use any piggybacking...
  - .. except in SYN+ACK (because done in kernel)

# The Setup

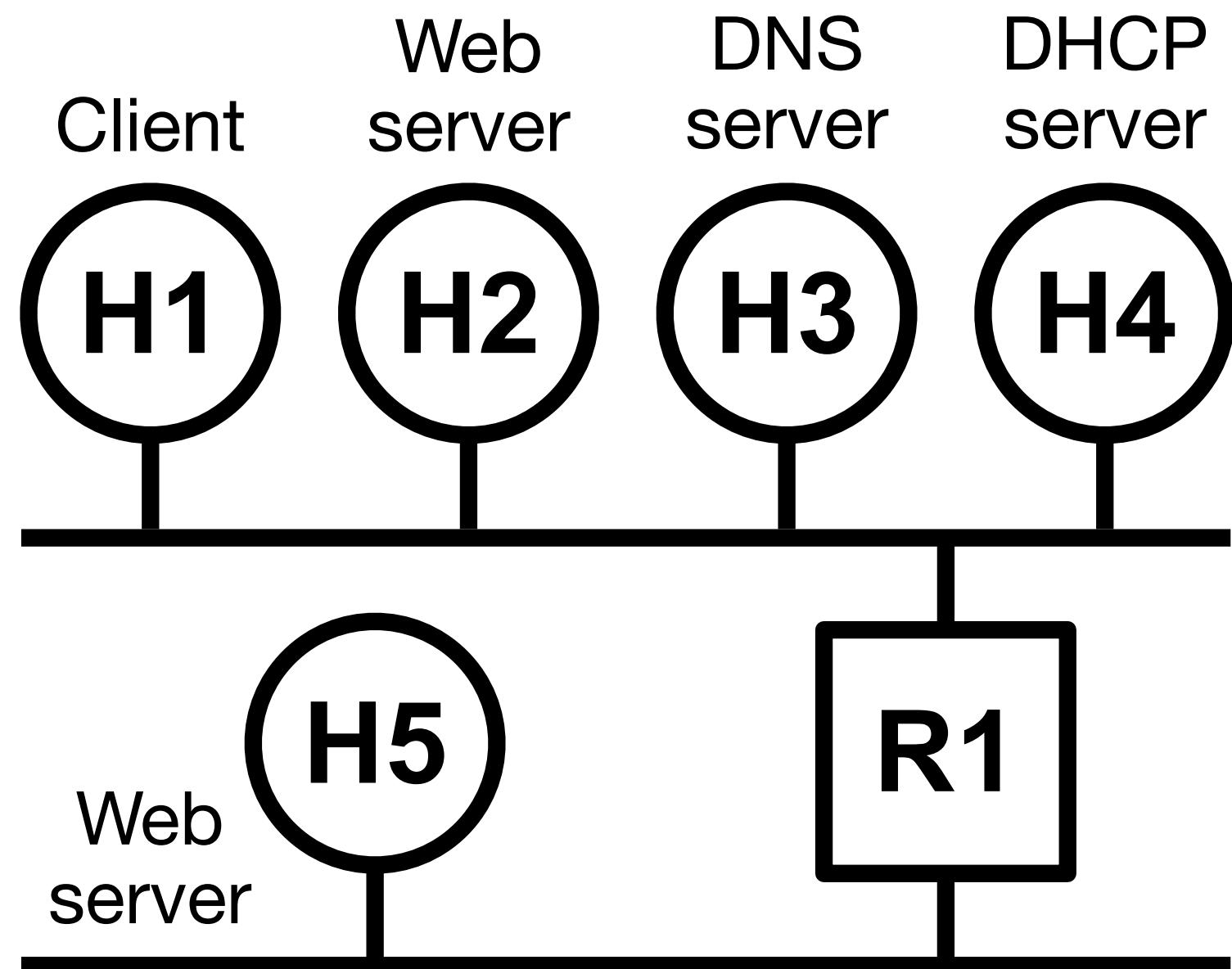
---

- Host H1...
  - Boots up (all state cleared)
  - Fetches a small file from H5.com
  - Goes idle for five minutes
  - Fetches two small files from H2.com
- To do list:
  - DHCP (get configured)
  - ARP for DNS server
  - Resolve H5.com
  - ARP for R1
  - TCP connection to H5
  - HTTP request to H5
  - TCP disconnect from H5
  - Resolve H2.com
  - ARP for H2
  - TCP connection to H2
  - HTTP request to H2
  - HTTP request to H2
  - TCP disconnect from H2



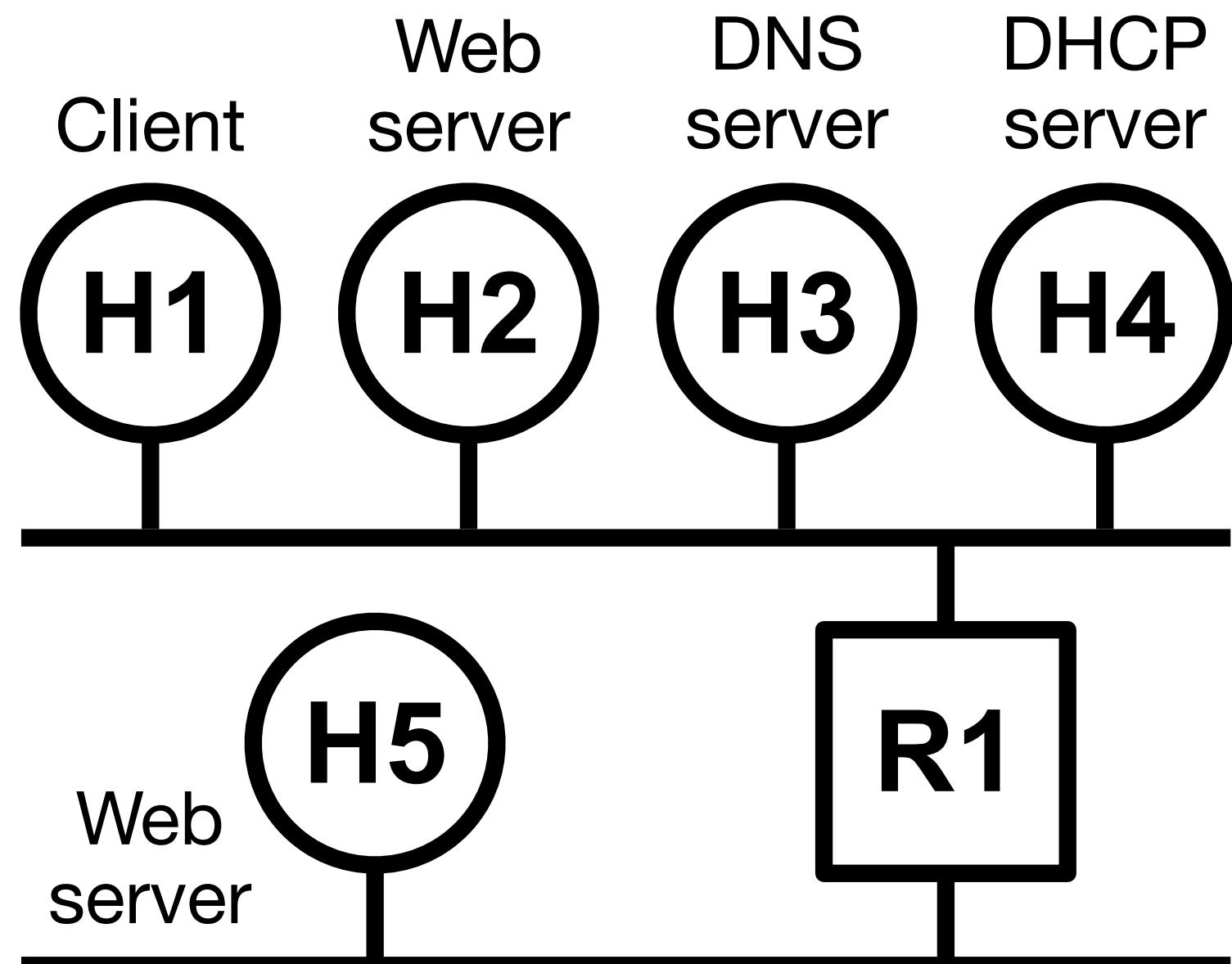
| Dir | O  | Trnsp | Message                 |
|-----|----|-------|-------------------------|
| ⇒   | *  | UDP   | DHCP discover           |
| ⇐   | H4 | UDP   | DHCP offer              |
| ⇒   | *  | UDP   | DHCP request            |
| ⇐   | H4 | UDP   | DHCP acknowledge        |
| ⇒   | *  | -     | ARP request for H3      |
| ←   | H3 | -     | ARP response from H3    |
| →   | H3 | UDP   | DNS request for H5.com  |
| ←   | H3 | UDP   | DNS response for H5.com |
| ⇒   | *  | -     | ARP request for R1      |
| ←   | R1 | -     | ARP response from R1    |
| →   | H5 | TCP   | SYN                     |
| ←   | H5 | TCP   | SYN+ACK                 |
| →   | H5 | TCP   | ACK                     |
| →   | H5 | TCP   | HTTP GET                |
| ←   | H5 | TCP   | ACK                     |
| ←   | H5 | TCP   | HTTP response           |
| →   | H5 | TCP   | ACK                     |
| →   | H5 | TCP   | FIN                     |
| ←   | H5 | TCP   | ACK                     |
| ←   | H5 | TCP   | FIN                     |
| →   | H5 | TCP   | ACK                     |

- ✓ 1. DHCP (get configured)
- ✓ 2. ARP for DNS server
- ✓ 3. Resolve H5.com
- ✓ 4. ARP for R1
- ✓ 5. TCP connection to H5
- ✓ 6. HTTP request to H5
- ✓ 7. TCP disconnect from H5
- 8. Resolve H2.com
- 9. ARP for H2
- 10. TCP connection to H2
- 11. HTTP request to H2
- 12. HTTP request to H2
- 13. TCP disconnect from H2



| Dir | O  | Trnsp | Message                 |
|-----|----|-------|-------------------------|
| →   | H3 | UDP   | DNS request for H2.com  |
| ←   | H3 | UDP   | DNS response for H2.com |
| ⇒   | *  | -     | ARP request for H2      |
| ←   | R1 | -     | ARP response from H2    |
| →   | H5 | TCP   | SYN                     |
| ←   | H5 | TCP   | SYN+ACK                 |
| →   | H5 | TCP   | ACK                     |
| →   | H5 | TCP   | HTTP GET                |
| ←   | H5 | TCP   | ACK                     |
| ←   | H5 | TCP   | HTTP response           |
| →   | H5 | TCP   | ACK                     |
| →   | H5 | TCP   | HTTP GET                |
| ←   | H5 | TCP   | ACK                     |
| ←   | H5 | TCP   | HTTP response           |
| →   | H5 | TCP   | ACK                     |
| →   | H5 | TCP   | FIN                     |
| ←   | H5 | TCP   | ACK                     |
| ←   | H5 | TCP   | FIN                     |
| →   | H5 | TCP   | ACK                     |

- ✓ 1. DHCP (get configured)
- ✓ 2. ARP for DNS server
- ✓ 3. Resolve H5.com
- ✓ 4. ARP for R1
- ✓ 5. TCP connection to H5
- ✓ 6. HTTP request to H5
- ✓ 7. TCP disconnect from H5
- ✓ 8. Resolve H2.com
- ✓ 9. ARP for H2
- ✓ 10. TCP connection to H2
- ✓ 11. HTTP request to H2
- ✓ 12. HTTP request to H2
- ✓ 13. TCP disconnect from H2



# Questions?

Thank you!

Good luck on the project and final!

# Attributions

---

Norman Abramson

Public Domain, <https://commons.wikimedia.org/wiki/File:NormanAbramson.jpg>

Kris Krug, Bob Metcalfe and Tim Berners Lee (Cropped), by Shashi Bellamkonda

CC-BY 2.0, <https://www.flickr.com/photos/drbeachvacation/8543090629>

Many slides borrowed/adapted from earlier CS168/EE122, with thanks to Nick McKeown, Sylvia Ratnasamy, Jennifer Rexford, Scott Shenker, Ion Stoica, and others

# CS 168

# **Software-Defined Networking (SDN)**

Fall 2022

*Guest Lecture: Scott Shenker*

[CS168.io](http://CS168.io)

# Be Forewarned....

- First classroom lecture in four years...
  - ...so this could be very rough
- Please ask questions when I'm floundering
  - To save us all the embarrassment
- When I ask a question, not looking for an answer
  - I'm asking you to think!
- Lecture will start slowly, with lots of generalities
  - This is to establish the necessary context
  - But things get more specific towards the end

# Goal For Today

- Provide the “why” of software-defined networking
  - Why was it needed, why did it come about,...
  - Some history, some gossip, and the post-hoc rationale
  - ***An exercise in retrospective architectural thinking***
- Almost none of the real “how”
  - Go read papers (RCP, 4D, NOX, ONIX, NetVirt, Fabric, ...)
  - Sylvia is actively working on new ways of doing SDN
  - But the main point of SDN isn’t in the details of how...
- I am presenting the “canonical” version of SDN
  - Which absolutely no one uses in its pure form
  - But is the best way to understand what SDN is conceptually

# We Begin With Two Questions

# Q#1: What Is SDN?

- SDN is a way of managing networks
  - Will clarify what that means later in lecture
- However, SDN is not a revolutionary technology...
  - ...just a way of organizing network functionality
- But that's all the Internet architecture is....
  - The Internet architecture isn't clever, but it is deeply wise
- *SDN isn't clever, and we can only hope it is wise*
  - *We'll find out in thirty or forty years...*

# Q#2: Where did SDN come from?

- ~2004: Research on new management paradigms
  - RCP, 4D [Princeton, CMU,....]
  - SANE, Ethane [Stanford/Berkeley]
  - Industrial efforts with similar flavor (most not published)
- 2008: Software-Defined Networking (SDN)
  - NOX Network Operating System [Nicira]
  - OpenFlow switch interface [Stanford/Nicira]
- 2011: Open Networking Foundation (ONF)
  - **Board:** Google, Yahoo, Verizon, DT, Msft, Fbook, NTT, GS
  - **Members:** Cisco, Juniper, HP, Dell, Broadcom, IBM,.....

# Where did SDN really come from?



**Martín Casado**

# Current Status Of SDN

- SDN accepted as **right way to do networking**
  - Commercialized, in production use, growing revenue
    - E.g., in use at Google/MSoft/Amazon, carriers partially adopted
  - Not fully adopted by router vendors, so this talk will often refer to pre-SDN practices in the present tense
- Was an insane level of SDN hype, and backlash...
  - SDN doesn't work miracles, merely makes things easier
- But the real question is: *why the rapid adoption?*
  - 2004: **idea**, 2008: **design**, 2011: industry **frenzy**
  - *This is incredibly fast for networking!*

# Why The Rapid Adoption?

- When a technology is adopted so quickly, it must be addressing a significant pain point.
  - Especially true in networking, which changes very slowly
- SDN was addressing ***two*** huge pain points
- #1: Cisco's extreme market power
  - Explains why ***vendors*** jumped on SDN
- #2: The poor state of network management
  - Explains why ***customers*** cared about SDN

# **Network Management**

# What is network management?

- Recall the two “planes” of networking
- **Data plane:** forwarding packets
  - Based on local forwarding state
- **Control plane:** computing that forwarding state
  - Involves coordination with rest of system
- Broad definition of “network management”:
  - *Everything having to do with the control plane*

# Original Goals For The Control Plane

- **Basic connectivity:** route packets to destination
  - Forwarding state computed by routing protocols
  - Globally (intradomain) distributed algorithms
- **Interdomain policy:** find policy-compliant paths
  - Done by globally (interdomain) distributed BGP
- For long time, these were the only relevant goals!
- ***What other goals are relevant now?***
- Here are a few examples...

# Isolation Of Logical LANs

- L2 bcast protocols often used for discovery
  - Useful, unscalable, invasive
- Want multiple logical LANs on a physical network
  - Retain usefulness, cope with scaling, provide isolation
- Use VLANs (virtual LANs) tags in L2 headers
  - Controls where broadcast packets go
  - Can create multiple logical L2 networks
  - Routers connect these logical L2 networks
- No universal method for setting VLAN state

# Access Control

- Operators want to limit access to various hosts
  - “Don’t let laptops access backend database machines”
  - Crucial for security
- This can be imposed by routers using ACLs
  - ACL: Access Control List
- Example entry in ACL: <header template; drop>
  - If not port 80, drop
  - If source address = X, drop
- These are typically configured manually
  - And often implemented in firewalls

# Traffic Engineering

- Choose routes to spread traffic load across links
- Two main methods:
  - Setting up MPLS tunnels (*MPLS is layer 2.5*)
  - Adjusting weights in OSPF
- Often done with centralized computation
  - Take snapshot of topology and load
  - Compute appropriate MPLS/OSPF state
  - Send state to network

# Net management now has many goals

- Achieving these goals is job of the control plane...
- ...which currently involves many mechanisms
- **Globally distributed:** routing algorithms
- **Manual/scripted configuration:** ACLs, VLANs
- **Centralized computation:** Traffic engineering

# Bottom Line

- Many different control plane mechanisms
- Each designed from scratch for their intended goal
- Encompassing a wide variety of implementations
  - Distributed, manual, centralized,...
- And none of them particularly well designed
- **Network control plane was a complicated mess!**
  - With mediocre functionality...
- Big contrast with simple and functional dataplane

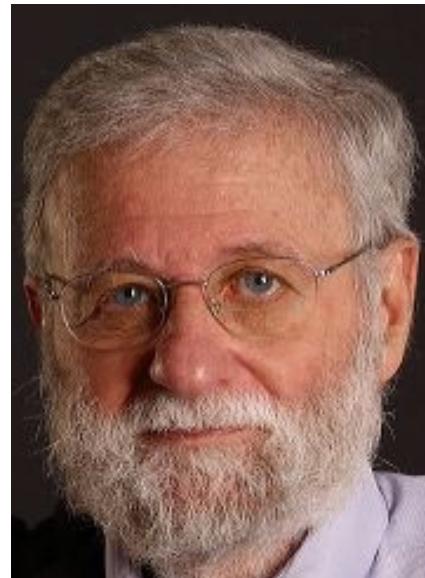
# Questions?

# How Have We Managed To Survive?

- Network admins must master this complexity
  - Understand all aspects of networks
  - Must keep myriad details in mind
- Networks require large expert admin staffs
  - Much larger than compute admin staffs
  - This is how we survive, by mastering complexity
- This ability to master complexity is both a blessing
  - ...and a curse!

# A Simple Story About Complexity...

- ~1985: Don Norman visits Xerox PARC
  - Talks about user interfaces and stick shifts
  - Do you even know what a stick shift is?



# What Was His Point?

- The ability to **master complexity** is valuable
  - But not the same as the ability to **extract simplicity**
- Each has its role:
  - When first getting systems to work, *master complexity*
    - ***Stick shifts!***
  - When making system easy to use, *extract simplicity*
    - ***Automatic transmissions!***
- You will never succeed in extracting simplicity
  - ***If you don't recognize it is a different skill set than mastering complexity!***

# What Is My Point?

- Networking had never made the distinction...
  - And therefore never made the transition from mastering complexity to extracting simplicity for control plane
- Until SDN, focused on mastering complexity
  - Networking “experts” are those that know all the details
- Network management had suffered as a result
- *Simplify network mngmt requires extracting simplicity*
  - And we had never bothered to do that for control plane

# Forcing People To Make Transition?

- We are really good at mastering complexity
  - And it had worked for us for decades, why change?
- How do you make people change?
  - Make them cry!
- A personal story about algebra and complexity
  - School problems:
$$3x + 2y = 8$$
$$x + y = 3$$
  - My father's problems:
$$327x + 26y = 8757$$
$$45x + 57y = 7776$$
  - My response: (1) I cried, (2) I learned algebra

# How Do You Make Network Operators Cry?

What convinced network operators  
that they needed SDN?

# Step 1: Large datacenters

- 100,000s machines; 10,000s switches
- Pushing the limits of what we could handle....

# Step 2: Multiple tenancy

- Large datacenters can host many customers
  - Gave rise to the modern public cloud
- Each customer gets their own logical network
  - Customer should be able to set policies on this network
  - ACLs, VLANs, etc.
- If there are 1000 customers, that adds 3 oom
  - Where oom = orders of magnitude
- This went way beyond what we could handle
  - Because our control plane is so primitive!

# Net Operators Were Now Weeping...

- They had been beaten by complexity
- The era of ad hoc control mechanisms was over
- We needed a simpler, more systematic design
  - We needed algebra, not arithmetic...
- But note the contrast between banks and multitenant datacenters:
  - One group willing to continue mastering complexity
  - The other was defeated, and needed something new
    - And they were desperate, which is why the rapid adoption

# What Do We Do Now?

- We had been defeated by complexity in DCs
- So we had to “extract simplicity”!
- ***So how do you “extract simplicity”?***

# An Example Transition: Programming

- Machine languages: no abstractions
  - Had to deal with low-level details
  - Mastering complexity was crucial
- Higher-level languages: OS and other abstractions
  - File system, virtual memory, abstract data types, ...
- Modern languages: even more abstractions
  - Object orientation, garbage collection, ...

**Abstractions key to extracting simplicity**

# “The Power of Abstraction”

“

“Modularity based on abstraction  
is the way things get done”

–Barbara Liskov

**Abstractions → Interfaces → Modularity**

# What About Network Abstractions?

- Consider the data and control planes separately
- Different tasks, so naturally different abstractions

# Abstractions for Data Plane: Layers

**Applications**

**...built on...**

**Reliable (or unreliable) transport**

**...built on...**

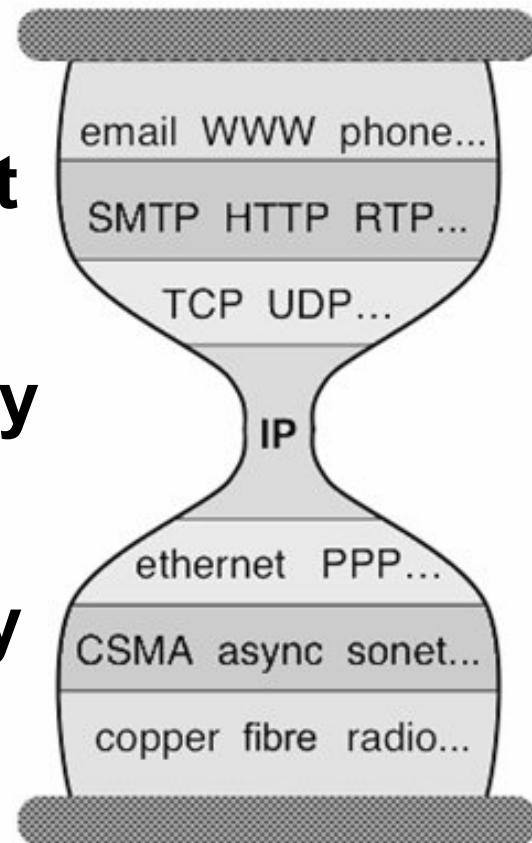
**Best-effort global packet delivery**

**...built on...**

**Best-effort local packet delivery**

**...built on...**

**Physical transfer of bits**



# Control Plane Abstractions



# Many Control Plane Mechanisms

- Variety of goals, no modularity:
  - **Routing:** distributed routing algorithms
  - **Isolation:** ACLs, VLANs, Firewalls,...
  - **Traffic engineering:** adjusting weights, MPLS,...
- **Control Plane: mechanism without abstraction**
  - *Too many mechanisms, not enough functionality*

# **SDN: An Exercise in Finding Control Plane Abstractions**

# Lecture So Far

- We have motivated the need for SDN
  - Networks admins were using arithmetic...
  - ...but suddenly needed algebra
- We now talk about how SDN met that need
  - What is the “algebra” of network management

# How do you find abstractions?

- You start with a task you need to perform
- You then decompose the task....
- ...and define abstractions for each subtask
- Let's do that for the control plane
- Basic task is to compute forwarding state
- But this task has several subtasks or constraints

# Task: Compute Forwarding State

- Consistent with low-level hardware/software
  - Which might depend on particular vendor
- Based on entire network topology
  - Because many control decisions depend on topology
- For all routers/switches in network
  - Every router/switch needs forwarding state

# The Pre-SDN Approach

- Design one-off mechanisms that deal with all three
  - E.g., routing protocols deal with all three subproblems
- A sign of how much we love complexity
- No other field would do it this way!
- They would define abstractions to handle each subtask independently
- ...and so should we!
- And that is what leads to SDN

# Separate Concerns With Abstractions

1. Be compatible with low-level hardware/software  
Need an abstraction for general **forwarding model**
2. Make decisions based on entire network  
Need an abstraction for **network state**
3. Compute configuration of each physical device  
Need an abstraction that **simplifies configuration**

# Abs#1: Forwarding Abstraction

- Express intent independent of implementation
  - Don't want to deal with proprietary HW and SW
- OpenFlow is one proposal for forwarding
  - Standardized interface to switch
  - Configuration in terms of flow entries: <header, action>
- Design details concern exact nature of:
  - Header matching
  - Allowed actions

# Separate Concerns With Abstractions

1. Be compatible with low-level hardware/software  
Need an abstraction for general **forwarding model**
  
2. **Make decisions based on entire network**  
**Need an abstraction for network state**
  
3. Compute configuration of each physical device  
Need an abstraction that simplifies configuration

# Abs#2: Network State Abstraction

- Abstract away various distributed mechanisms
- Abstraction: **global network view**
  - Annotated network graph provided through an API
- Implementation: “Network Operating System”
  - Runs on servers in network (“controllers”)
  - Replicated for reliability
- Information flows both to and from NOS
  - Information from routers/switches to form “view”
  - Configurations to routers/switches to control forwarding

# Network Operating System

- Think of it as a centralized link-state algorithm
- Switches send connectivity info to controller
- Controller computes forwarding state
  - Some control program that uses the topology as input
- Controller sends forwarding state to switches
  - Using forwarding abstraction (OpenFlow)
- Controller is replicated for resilience
  - System is only “logically centralized”

# Network of Functions and Network Function Slicing

routing, access control, etc.

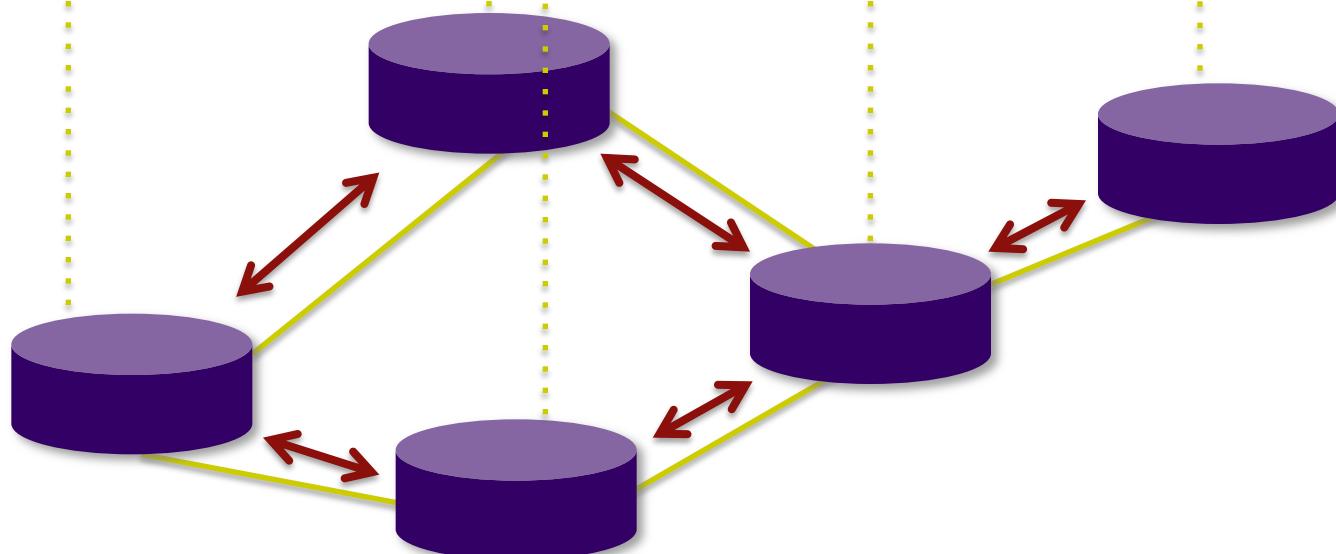
Control Program

Distributed algorithm running between neighbors

Global Network View

*Complicated task-specific distributed algorithm*

Network OS



# Major Change In Paradigm

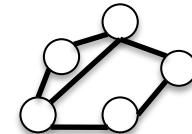
- Control program: **Configuration = Function(view)**
  - Configuration means set of forwarding entries
- Control mechanism now program using NOS API
- Not a distributed protocol, just a graph algorithm
  - All distributed algorithms in NOS
- Configurations are passed to switches by NOS

# Software Defined Network (So Far)

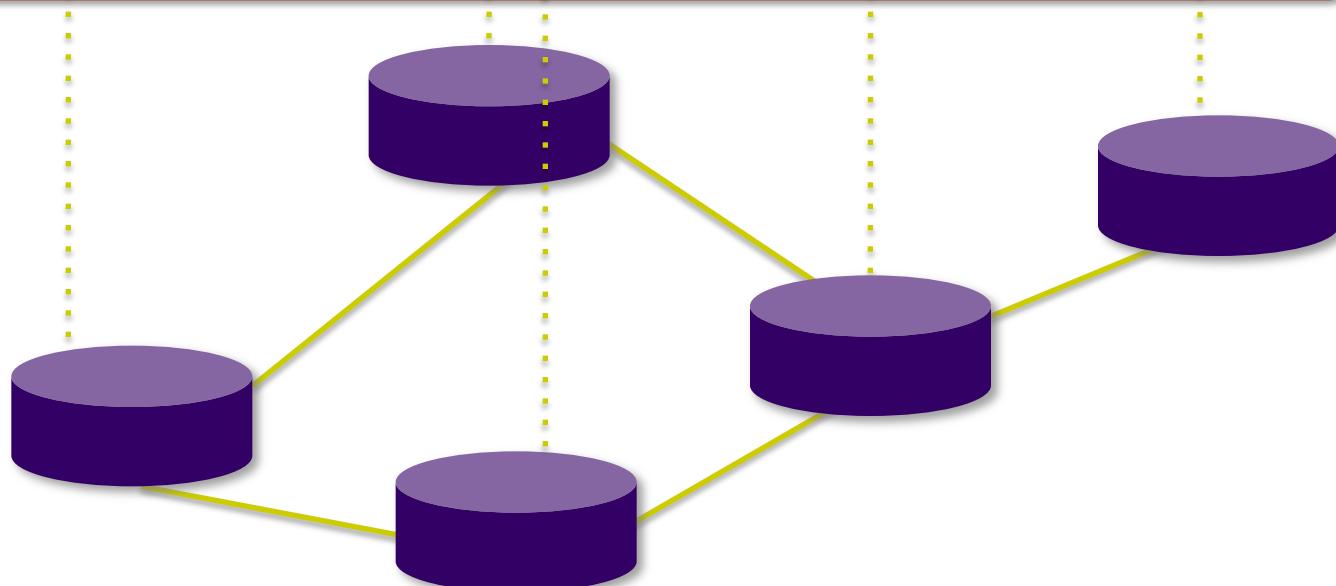
**routing, access control, etc.**

Control Program

Global Network View



Network OS



# Questions?

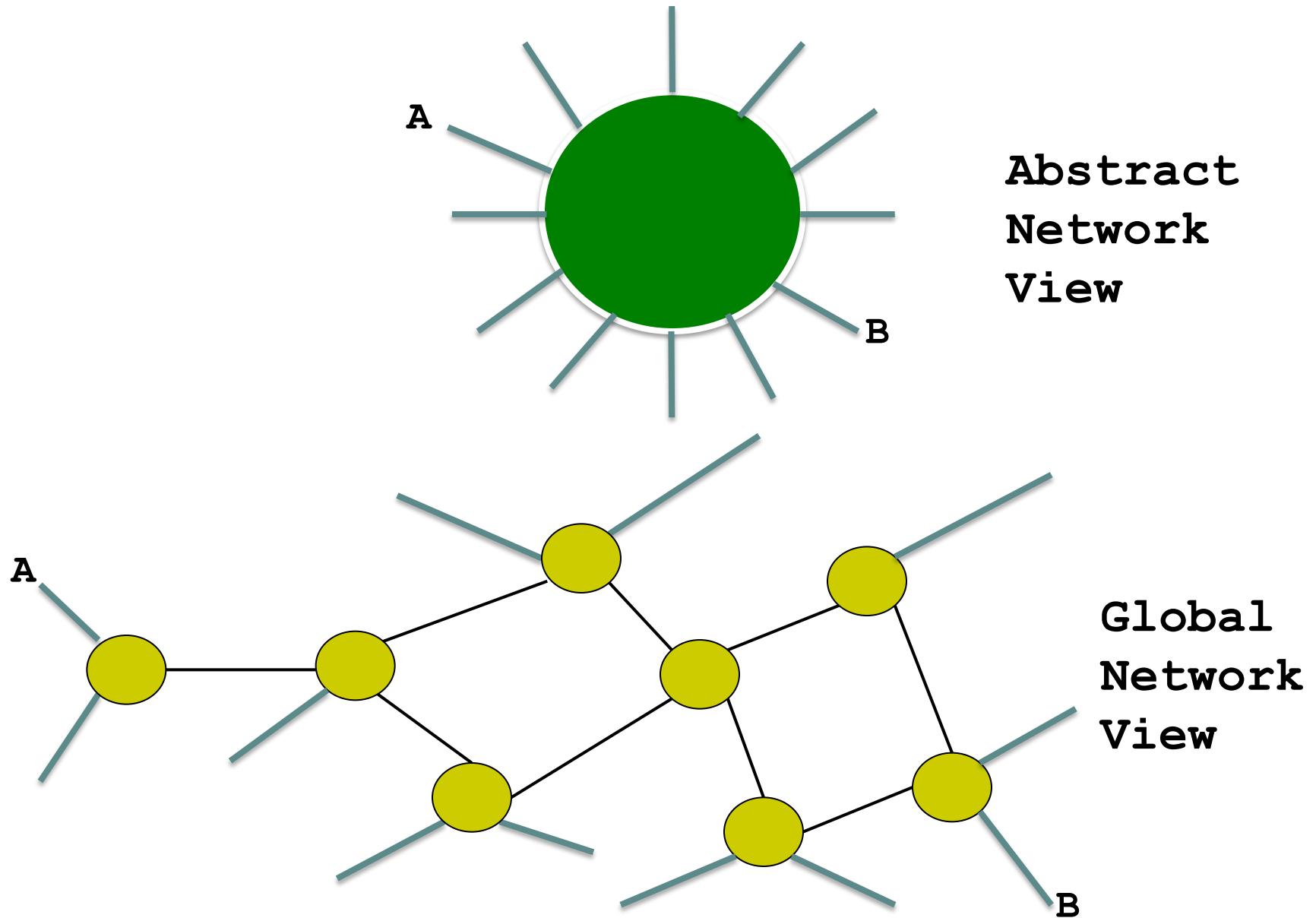
# Separate Concerns With Abstractions

1. Be compatible with low-level hardware/software  
Need an abstraction for general forwarding model
2. Make decisions based on entire network  
Need an abstraction for network state
3. **Compute configuration of each physical device**  
**Need an abstraction that simplifies configuration**  
Otherwise, control program must compute forwarding entries for every switch in network....

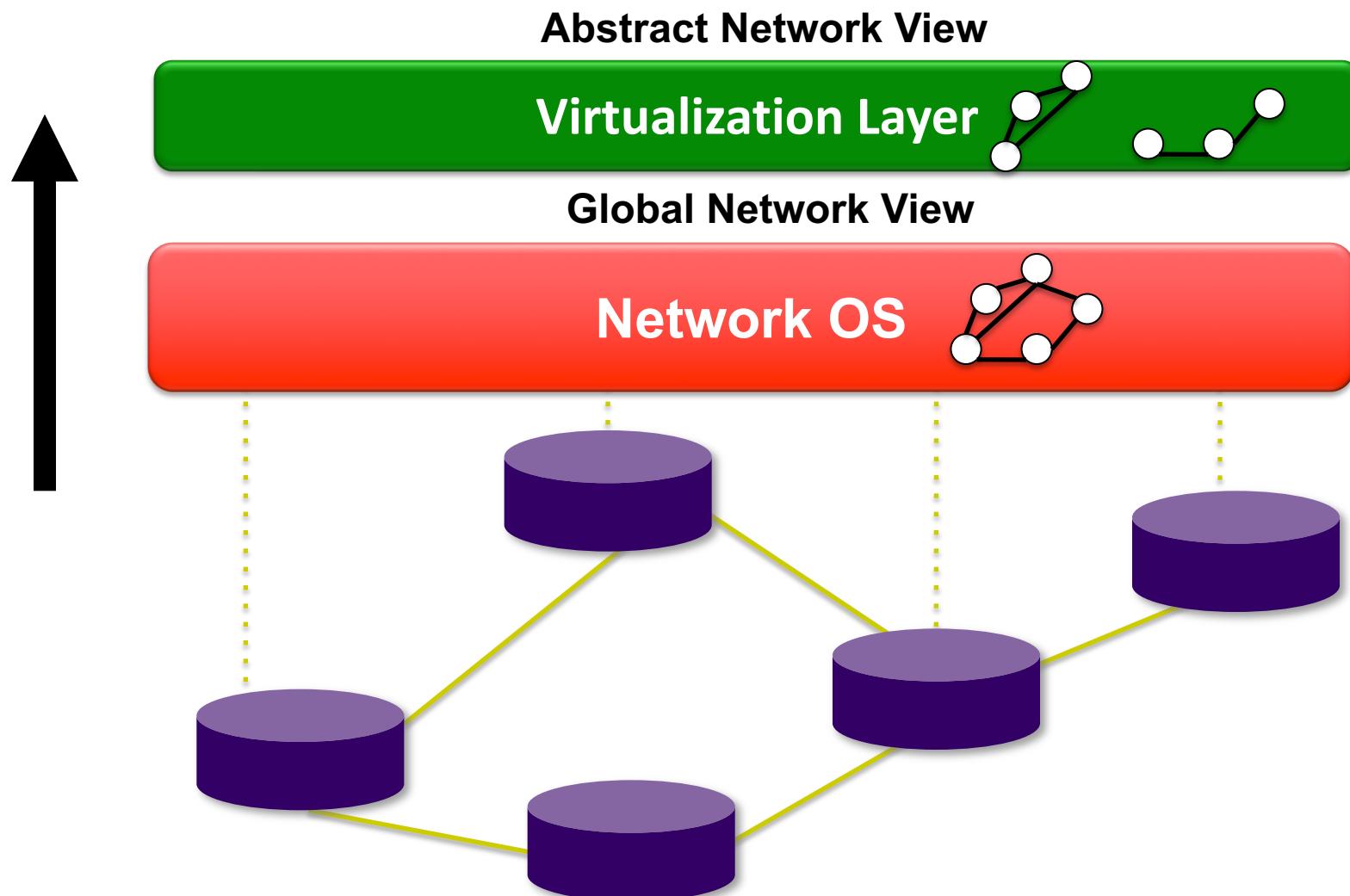
# Abs#3: Specification Abstraction

- Control mechanism specifies desired behavior
  - Whether it be isolation, access control, or QoS
- It should not be responsible for *implementing* that behavior on physical network infrastructure
  - Requires configuring the forwarding tables in each switch
- Proposed abstraction: **abstract view** of network
  - Abstract view models only enough detail to specify goals
  - Will depend on task semantics
  - Now called “intention-based” networking
  - Think of this as providing a compiler...

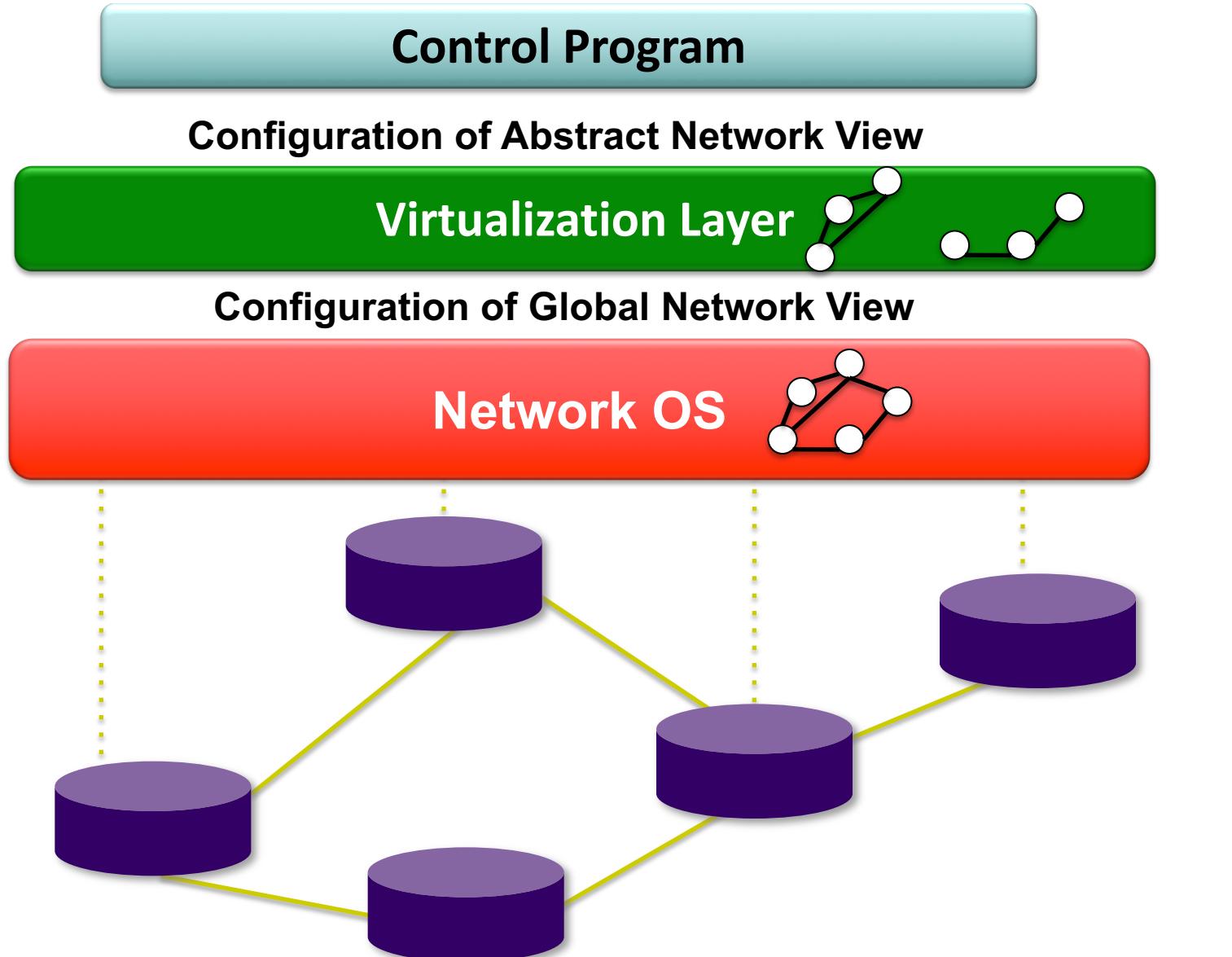
# Simple Example: Access Control



# Software Defined Network



# Software Defined Network



# **What This Really Means**

# Routing Application

- Look at graph of network
- Compute routes
- Give to SDN platform, which passes on to switches
- *Graph algorithm, not distributed protocol*

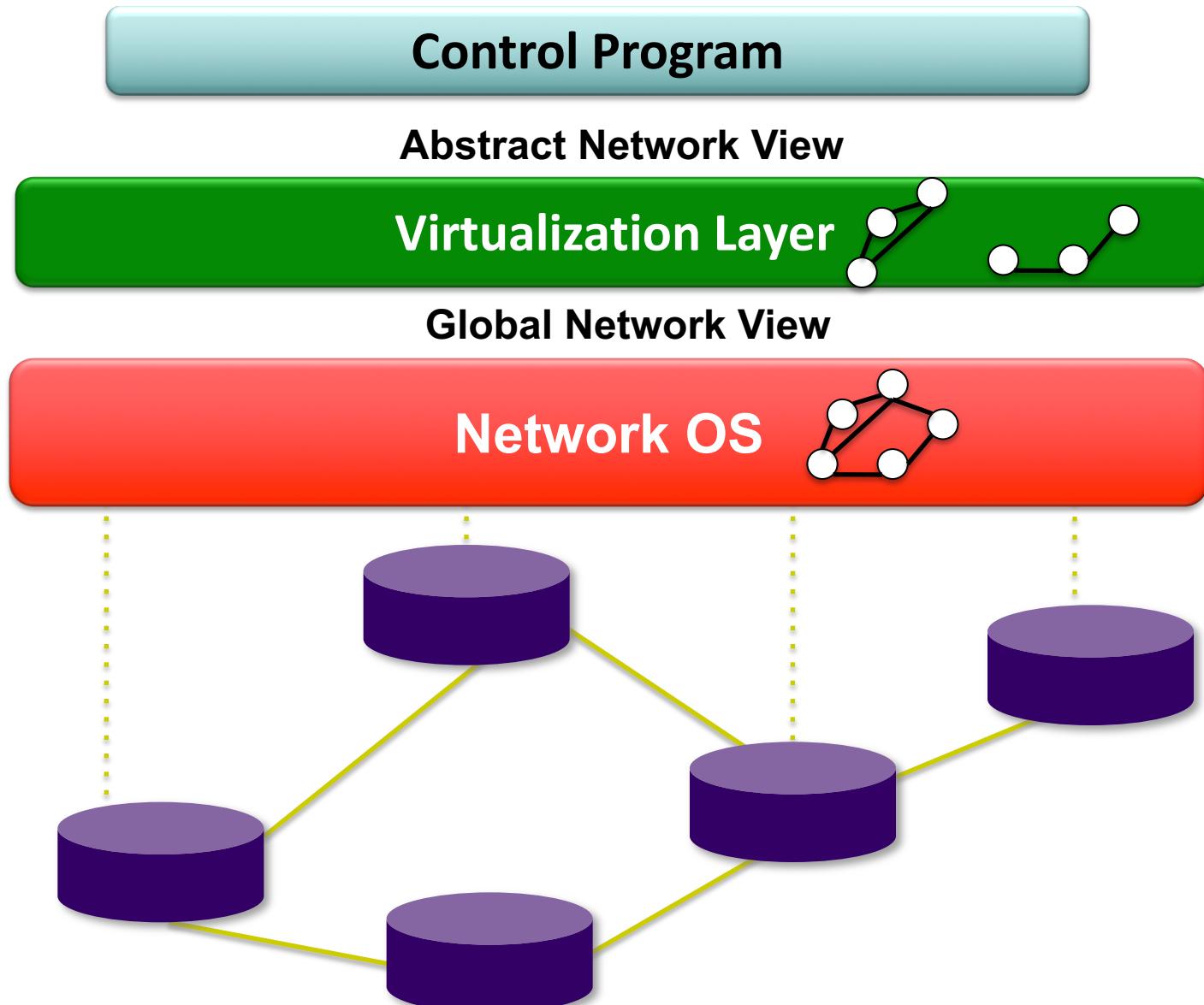
# Access Control Application

- Control program decides who can talk to who
  - E.g., based on security category
- Pass this information to SDN platform
- Appropriate ACL flow entries are added to network
  - In the right places (based on the topology)
- *The control program that decides who can talk to whom doesn't care what the network looks like!*

# Clean Separation Of Concerns

- **Control program:** specify goals on abstract view
  - Driven by **Operator Requirements**
- **Virt. Layer:** abstract view  $\leftrightarrow$  global view
  - Implements goals on network (as in global view)
  - Driven by **Specification Abstraction** for particular task
- **NOS:** global view  $\leftrightarrow$  physical switches
  - API: driven by **Network State Abstraction**
  - Switch interface: driven by **Forwarding Abstraction**

# SDN: Layers For The Control Plane



# Questions?

# Abstrns Don't Remove Complexity

- NOS, Virtualization are complicated pieces of code
- SDN merely localizes the complexity:
  - Simplifies interface for control program (use-specific)
  - Pushes complexity into **reusable** code (SDN platform)
  - This is the trajectory of computer science
- This is the big payoff of SDN: modularity!
  - The core distribution mechanisms can be reused
  - Control programs only deal with their specific function

# Why Is SDN Important?

- As a design:
  - It is more modular, enabling faster innovation
  - Control programs become very simple!
- As an academic endeavor:
  - Provides abstractions that enable systematic reasoning
  - Can reason about control program, without looking at each switch...
- As a change in the ecosystem:
  - Open switch interfaces reduce vendor lock-in
  - Not clear that this will happen (why?)

# Common Questions About SDN?

- Is SDN less scalable, secure, resilient,...?
- Can SDN be extended to the WAN?
- Is OpenFlow the right fwding abstraction?
- Is SDN incrementally deployable?

# Common Questions About SDN?

- Is SDN less scalable, secure, resilient,...? **No**
- Can SDN be extended to the WAN? **Yes**
- Is OpenFlow the right fwding abstraction? **No**
- Is SDN incrementally deployable?  
**Yes**  
*How can this be?*

# What About Deployment?

- Most of SDN's design is in software on servers
  - NOS and virtualization layer run on servers
  - Deploying these components is easy!
- But all routers must support OpenFlow
  - To provide information to the SDN controllers
  - To receive flow entries from the SDN controllers
- Requires replacing all routers in network
  - Routers are closed/proprietary, vendors won't upgrade
- So the question is...

# How Did We Get This Deployed?

- Get everyone to buy new OpenFlow switches?
- That is a completely ludicrous approach
  - *Though one we believed in at Nicira for a while*
- So, how did we deploy SDN?
  - Without them buying new switching hardware
  - And in some cases not even talking to the networking team at the company....
- Think about it....

# Fact #1

- Most additional control plane functionality can be implemented at the edge
  - Access control, LAN Isolation, traffic engineering,...
  - Think about this for a second..
- Network core merely needs to deliver packets
  - Pre-SDN networking technologies pretty good at this
  - i.e., control plane for core only has its original task
- So only need to add SDN at network edge...
- This edge/core split arises in other contexts
  - E.g., MPLS, which has been widely adopted

## Fact #2

- The operators who were crying were from large multitenant datacenters....
- They run hypervisors on their hosts, to support VMs initiated by tenants
- These two facts gave us an opening....

# Deployment In Virtualized Datacenters

- Virtualization (VMs) is supported by hypervisors
- Hypervisors use virtual switches to connect VMs
- Make this virtual (software) switch SDN-compatible
  - And you'll be able to deploy SDN without any new HW
- Open vSwitch was an OpenFlow-capable vSwitch
  - Developed by Nicira, inserted into in Linux, Xen, etc.
- SDN now deployable without any HW deployment!
- This applies only to multitenant datacenters
  - *But they were our only customers!*

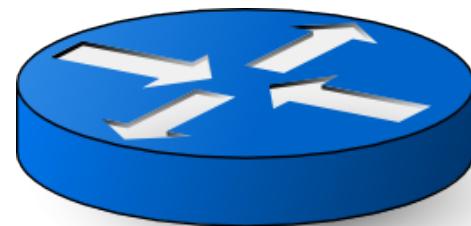
# Network in Regular Setting

Host

Host

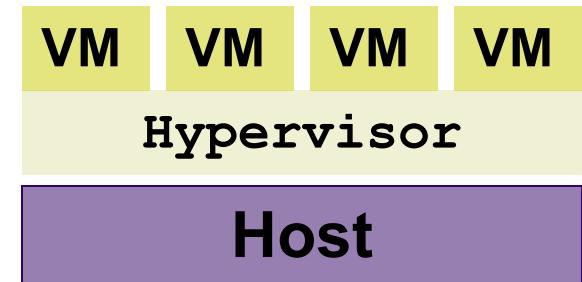
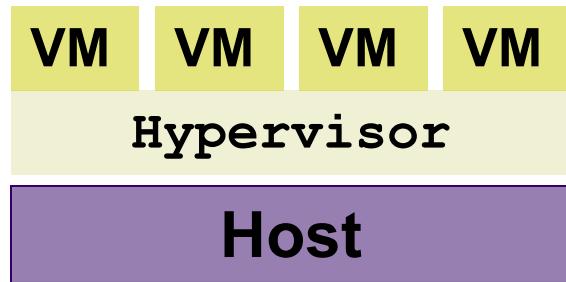
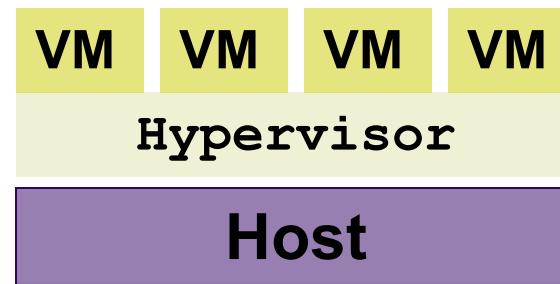
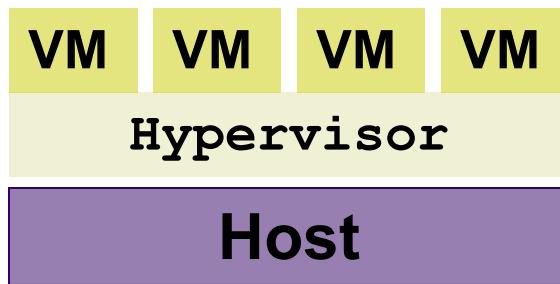
Host

Host

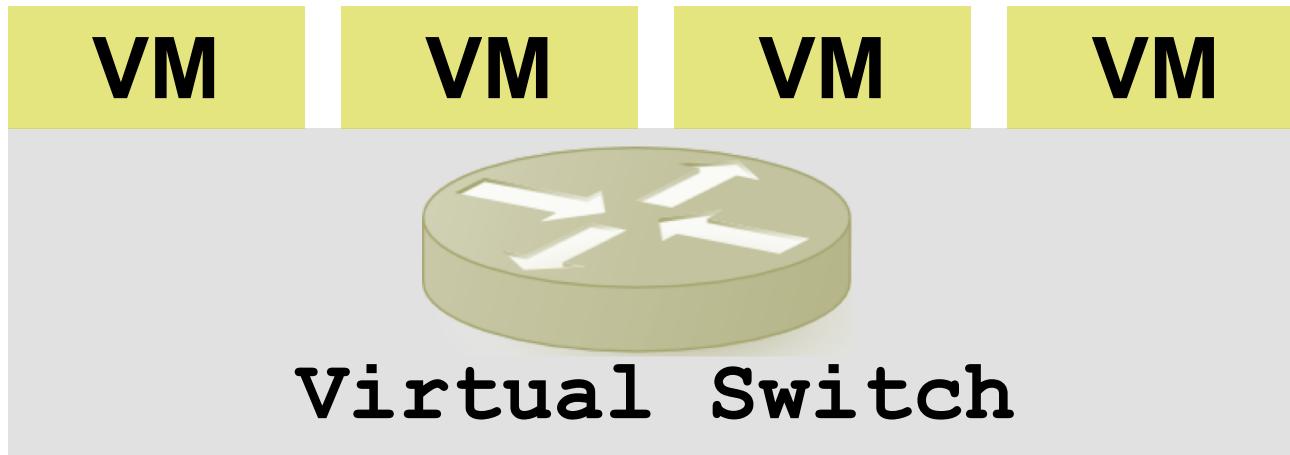


Physical Switches

# Network in Virtualized Setting

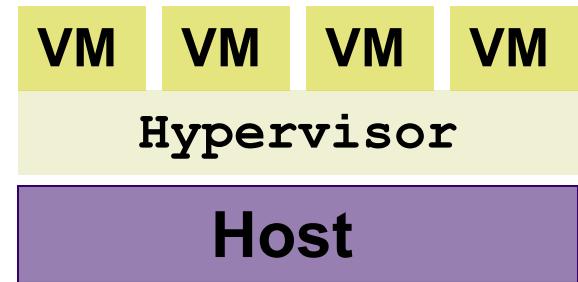
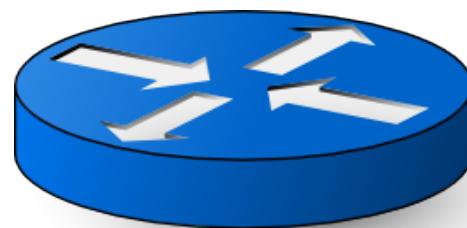
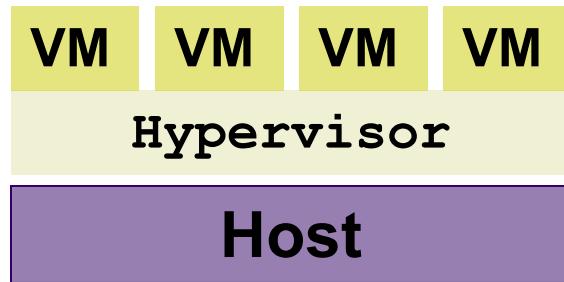
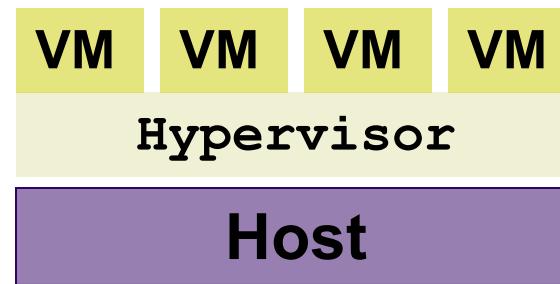
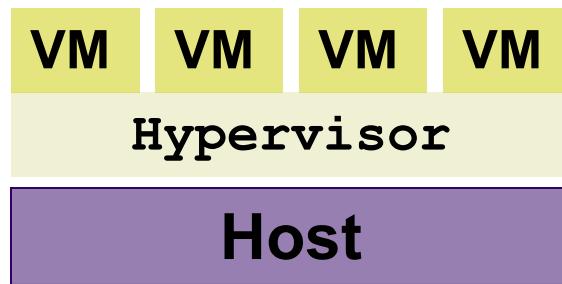


# Virtual Switches (vSwitch)

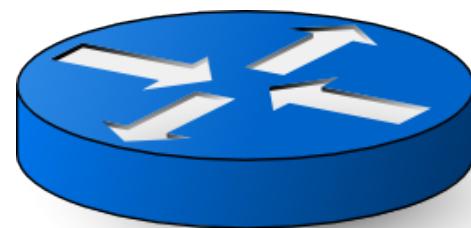


- vSwitch is first-hop switch for all VMs
  - vSwitch sends packet to other VMs, or to physical network
- vSwitch is a software switch
  - If it supports OpenFlow, can be controlled by NOS

# Physical View of Virtualized Network



# Logical View of Virtualized Network



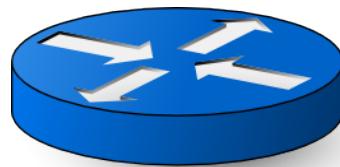
All edge switches are vSwitches

# vSwitches are Sufficient in VDCs

- vSwitches enough to implement most CP functions
  - Access control, QoS, mobility, migration, monitoring,...
- Physical network becomes static crossbar
  - Crossbar: just delivers packets from edge-to-edge
  - Simple to implement and manage
  - Mostly static (only responds to changes inside core)
- Edge handles all dynamic/configured functions
  - Tracking VM movement
  - Access control policies
  - ...

# Managing Physical Network

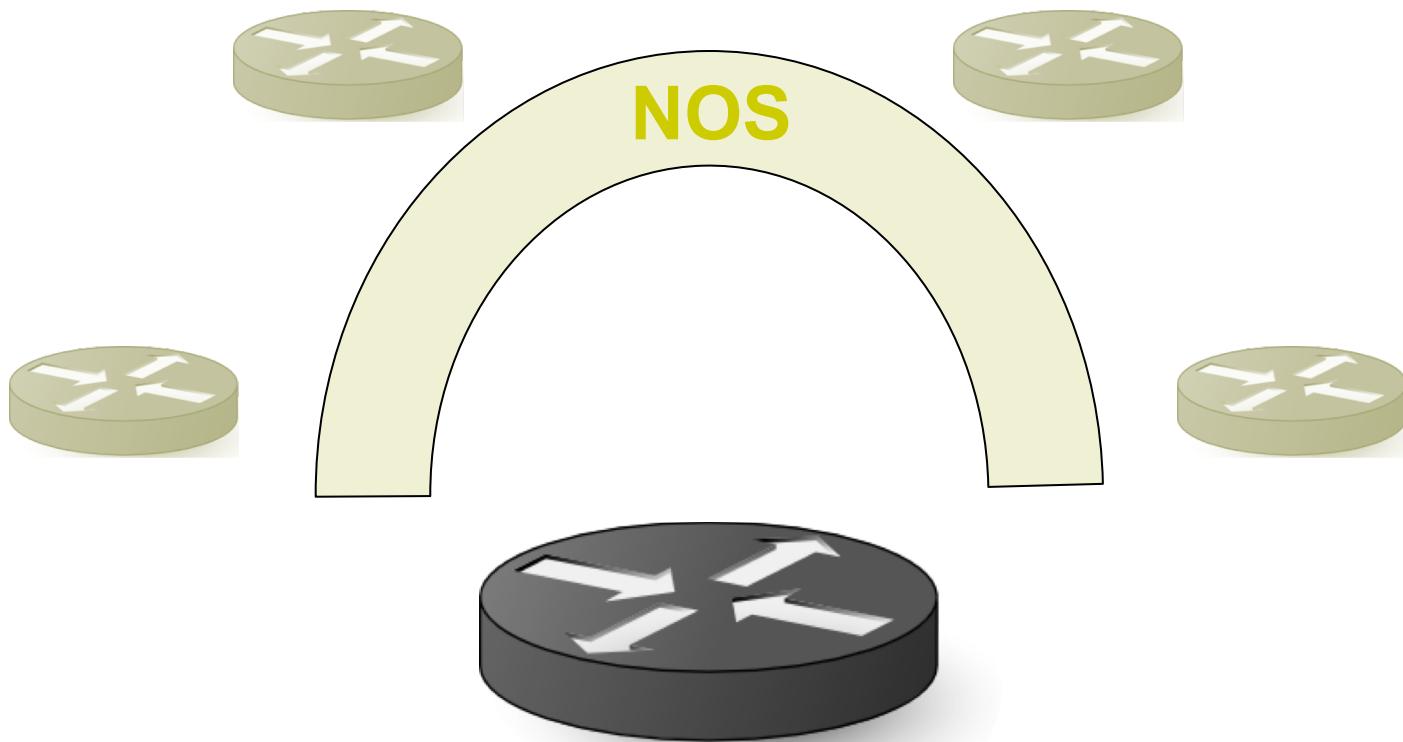
NOS



Physical Switches

# Managing Virtualized Network

**NOS only needs to control vSwitches at edge**



**Physical network is logical crossbar**

# vSwitches as Insertion Point

- Can insert new functionality into datacenters with
  - Hypervisors with OpenFlow-enabled vSwitch
  - Network Operating System (on servers)
- No change to physical infrastructure
  - Legacy hosts
  - Legacy network components
- This last issue isn't just a technical point
  - The network remaining completely unchanged is huge!

# Deploying SDN in VDCs

- Because the network is completely unchanged, the deployment can be managed by the compute team, not the networking team
  - Which have very different perspectives
- Networking team:
  - Very conservative, need to “not fail”, HW-oriented
- Compute team:
  - More nimble, need to deliver functionality, SW-oriented
- Initial SDN deployments were not network-driven
  - Which is what made them possible!

# Questions?

About this lecture, or anything else...