

# CS305

# Computer Networks

*2023 Spring*

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Course information

## *Lecture:*

- Zhuozhao Li, [lizz@sustech.edu.cn](mailto:lizz@sustech.edu.cn)
  - Office: RM 516, South Tower, CoE Building (工学院南楼516)
- Lectures: Wednesday 2:00 PM – 3:50 PM
- Location: Room 208, The Third Teaching Building

## *Lab:*

- Qing Wang, [wangq9@mail.sustech.edu.cn](mailto:wangq9@mail.sustech.edu.cn)
  - Office: Room 110, South Tower, CoE Building
- Location: Room 504, The Third Teaching building

# Introduction

- Zhuozhao Li, Ph.D.
- Assistant Professor, Department of Computer Science and Engineering
- Homepage: <https://zhuozhaoli.github.io/>
- Office hour: Friday 3-5pm or by email appointment
- Research interests:
  - Distributed systems and cloud computing

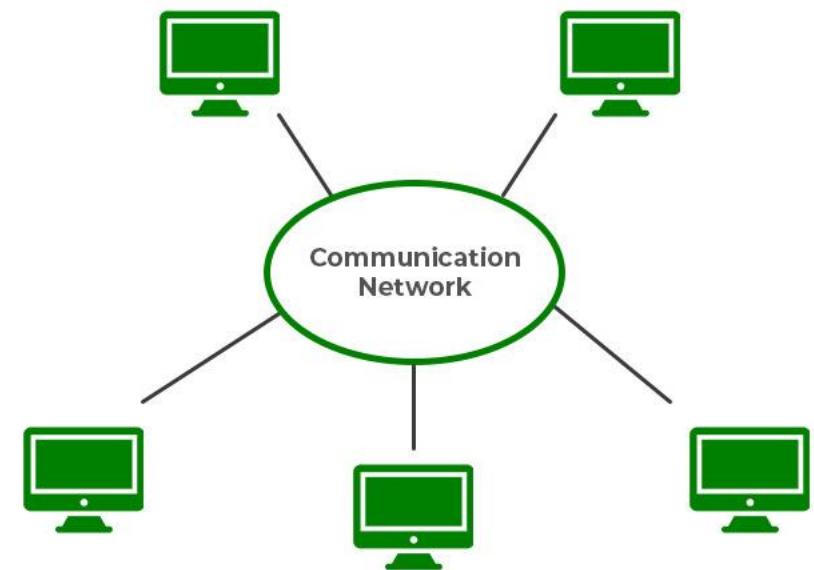
# Distributed system

A system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another

- Computers
- Networks

Examples of distributed systems

- Internet, websites, video content services...
- Supercomputers, cloud datacenters...



# Course information

- Sakai: <https://sakai.sustech.edu.cn/portal/site/db0d4bc0-03ad-4b69-83bd-0dd2ec67c8d1>
  - Name: CS305-spring2023
- QQ group: 181896966
- Syllabus (Tentative)
  - <https://zhuozhaoli.github.io/courses/CS305A/2023Spring/>

# Grading policy

- Homework and programming assignments – 15%
- Attendance and lab practice – 10%
- Project – 15%
  - CS students
  - Non-cs students
- Midterm Examination - 30%
  - Closed book
- Final Examination - 30%
  - Closed book

# Assignments

- No late assignment will be accepted
  - Unless some special situations (e.g., medical leave) which will be reviewed by all the instructors
- The following excuses will NOT be approved for late submissions: computer crashes, disk crashes, accidental file deletions, lab computer unavailability, and the like
- Linux, Python, Wireshark, possibly C/C++ for the project (CS students only)

# Rules about plagiarism

- No plagiarism is allowed
  - If plagiarism on homework or project is found for the first time, the plagiaristic part is graded as 0 and warning is given to the students
  - If plagiarism is found for the second time, the course is graded as 0
  - For project report, any sentence that is copied from other paper or article should cite the original source as the reference. Otherwise, the report is considered as plagiarism
- Submit the [commitment letter](#) on Sakai system
- [Don't](#) put your assignments or projects on any open-source website (e.g., Github). Otherwise, you have the same responsibility in case of plagiarism

# What will we learn in this course?

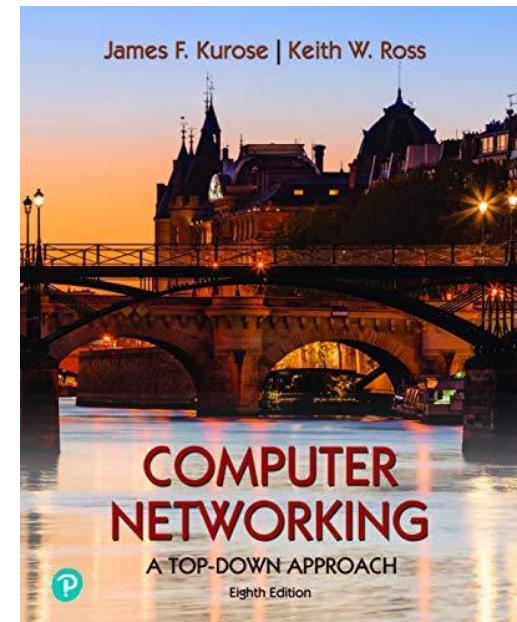
- What is network? What is communication?
  - Consider the analogy in practice: communication vs. social network
- What networks do we use in daily life? Any other network ever heard?
  - Mobile network, WWW, social network, etc.
- What applications require network access?
  - WeChat, games, websites, etc.

# What is this course about?

- Introductory (first) course in computer networking
  - learn **principles** of computer networking
  - learn **practice** of computer networking
  - Internet architecture/protocols as case study
- Goals:
  - learn a lot (not just factoids, but principles and practice)
  - have fun (well, it should be interesting, at least)

# Course materials

- Course materials:
  - Textbook: Computer Networking: A Top-Down Approach
    - J. Kurose & K. Ross, Pearson, 8th ed., 2020
  - Slides
- Online resources:
  - <https://sakai.sustech.edu.cn>
  - Textbook in pdf
  - Homework and assignments
  - Projects
  - Search

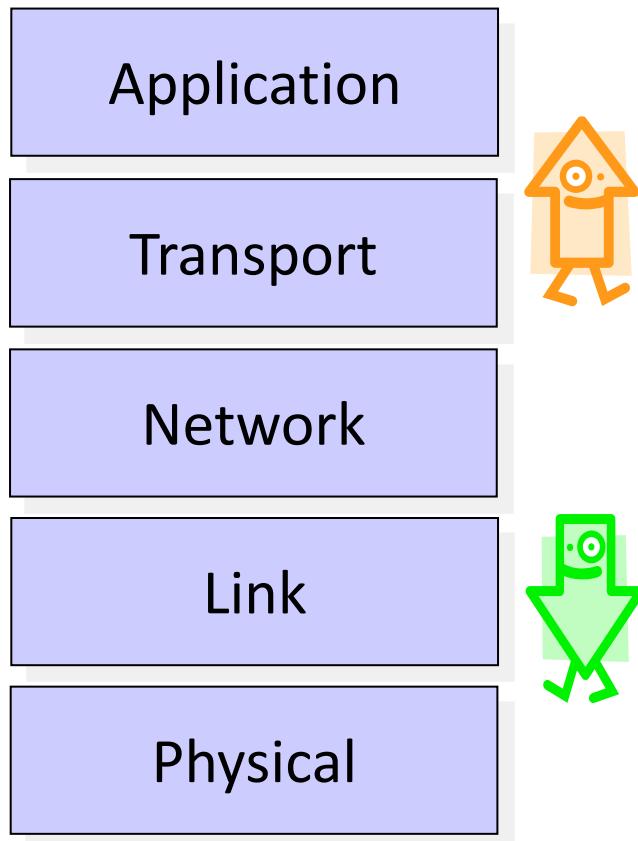


# How to use the textbook?

- For each lecture:
  - Read corresponding content after class
  - Go through the review questions
  - Finish homework
- After each chapter
  - Read summary and interview if interested

# Textbook information

- Computer Networking: A Top-Down Approach, James Kurose and Keith Ross, Pearson (8th Ed.)



- ❑ **Bottom Up:** Start with physical (e.g., wires) layer and move up to applications (e.g., mail, web browsers) layer explaining how functions are implemented
- ❑ **Top Down:** Start with application layer and move down to physical layer, explaining what expectations from applications, and how such services are implemented

# Course overview:

## Introduction (2 classes, text: Chapter 1)

- what is the Internet, what is a protocol?
- network edge, network core, network access
- physical media
- delay, loss, throughput in packet-switched networks
- protocol layers, service models
- Internet backbones, ISPs, IXPs
- brief history of networking, Internet

<https://zhuozhaoli.github.io/courses/CS305A/2023Spring/#schedule>

# Course overview:

## Application layer (*3 classes, text: Ch. 2*)

- principles of application-layer protocols
- World Wide Web: HTTP
- video streaming and content distribution networks
- electronic mail in the Internet
- the Internet's directory service: DNS
- P2P: Skype
- socket programming

# Course overview:

## Transport layer (*3 classes, text Ch. 3*)

- transport-layer services and principles
- multiplexing and demultiplexing applications
- connectionless transport: UDP
- principles of reliable of data transfer
- TCP case study
- principles of congestion control
- TCP congestion control

MIDTERM EXAM  
(approx)

# Course overview:

## Network layer (*4 classes, text: Ch. 4-5*)

- introduction and network service model
- what's inside a router?
- routing principles (algorithms)
- hierarchical routing
- IP: the Internet Protocol
- Internet routing: RIP, OSPF, BGP

In Textbook 8<sup>th</sup> edition:

**Network layer – Data Plane**

**Network layer – Control Plane**

- Software defined network (SDN)

# Course overview:

## Link layer, LANs (*2 classes, text: Ch. 6*)

- introduction, services
- error detection, correction
- multiple access protocols, LANs
- LAN addresses, ARP
- Ethernet
- network as a link layer: MPLS
- a day in the life of a web request (synthesis)

We will add more physical layer content in this chapter

# Course overview:

## Wireless and mobile networks (1 class, Ch 7)

- wireless link characteristics
- the wireless link:
- 802.11
- cellular Internet access
- mobility principles
- mobility in practice:
- mobile IP
- mobility in cellular networks

# Lab

## Basic content:

- Basic network commands
- Packet capture using Wireshark
- Protocol analysis
- Socket programming

## Make your hands dirty!

- Setup switch and router
- Setup wireless networks
- Analyze network performance

# Tips for attending lectures

- Having around 100 students in one room is difficult
- To get the best use of lectures
  - Interactive
  - Interrupt and ask whenever you have any question
  - Ask immediately after the class if you are shy
  - Give me suggestions and feedback frequently
- Get the main ideas in class, read the details after class

# Tips for this course

- Computer network is a human-invented object
  - No strict right or wrong, science vs. technology
  - limited by many factors -> trade-off
- We can meet almost all the content in our daily life
  - Think about: where do we use it when we learn a new application or protocol?  
What's your own experience?
- Take yourself as the designer of the internet
  - Think how to design the protocol before learn it
  - Try every idea out
- Computer network often mimics social network
  - Computer vs. people
  - Protocol vs. people communication

# Tips for this course

- Computer network is a human-invented object
  - No strict right or wrong, science vs. technology
  - limited by many factors -> trade-off
- We can meet almost all the content in our daily life
  - Think about: where do we use it when we learn a new application or protocol?  
What's your own ~~experience~~?
- Take yourself as the
  - Think how to design
  - Try every idea out
- Computer network often mimics social network
  - Computer vs. people
  - Protocol vs. people communication

Be active!

You can change the world!

# Chapter 1: Introduction

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Chapter 1: introduction

## *Chapter goal:*

- Get “feel,” “big picture,” introduction to terminology
  - more depth, detail *later* in course
- Approach:
  - use Internet as example



## *Overview/roadmap:*

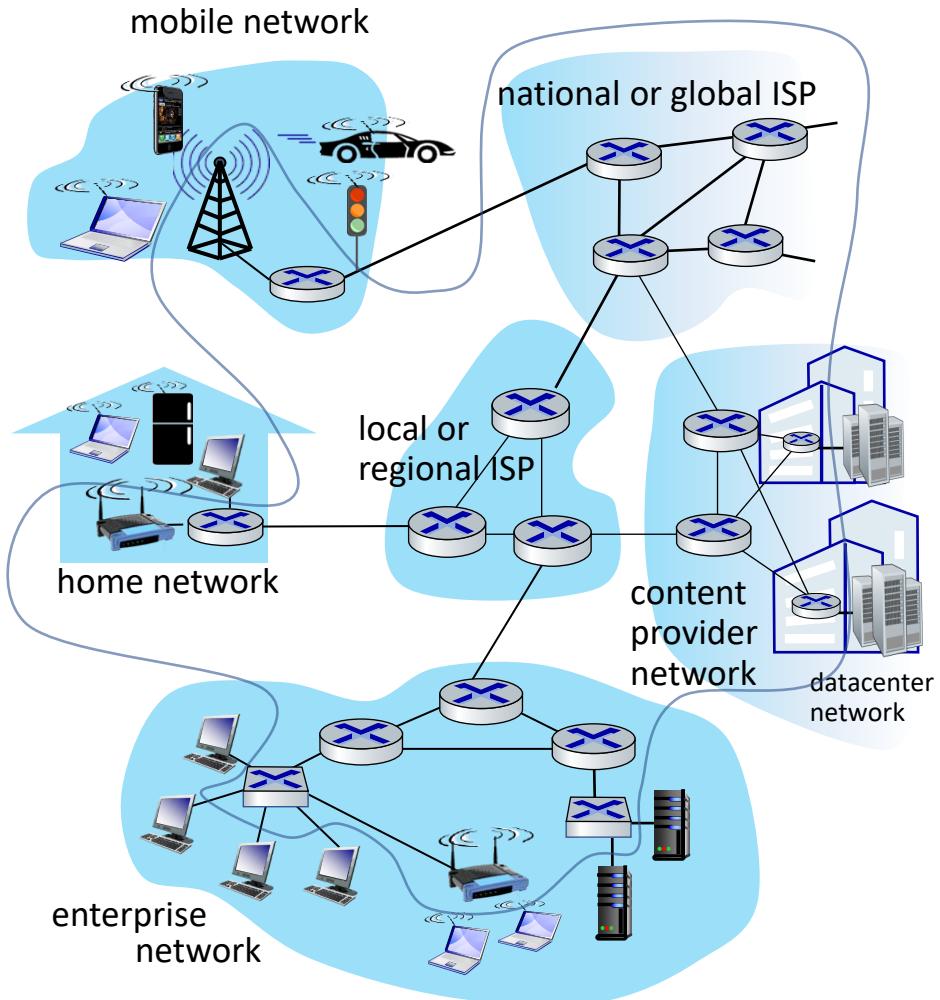
- What *is* the Internet?
- What *is* a protocol?
- **Network edge:** hosts, access network, physical media
- **Network core:** packet/circuit switching, internet structure
- Protocol layers, service models
- **Performance:** loss, delay, throughput
- Security
- History

# The Internet: a “nuts and bolts” view



Billions of connected computing *devices*:

- *hosts* = end systems
- running *network apps* at Internet's “edge”



# “Fun” Internet-connected devices



Amazon Echo



Internet refrigerator



Security Camera



Internet phones



IP picture frame



Slingbox: remote control cable TV



Pacemaker & Monitor



Web-enabled toaster + weather forecaster



sensorized, bed mattress



Tweet-a-watt: monitor energy use



AR devices



Fitbit

*Others?*

# The Internet: a “nuts and bolts” view



Billions of connected computing *devices*:

- *hosts* = end systems
- running *network apps* at Internet's “edge”

*Packet switches*: forward packets (chunks of data)

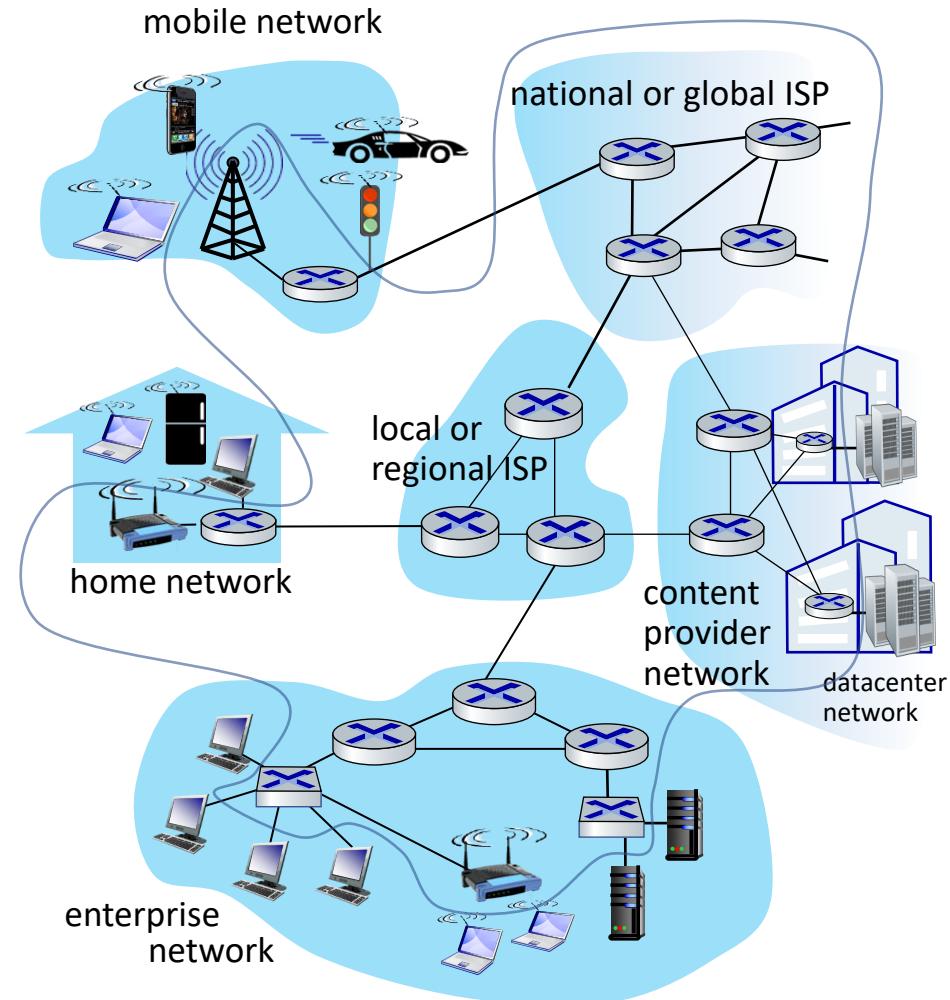
- routers, switches

*Communication links*

- fiber, copper, radio, satellite
- transmission rate: *bandwidth*

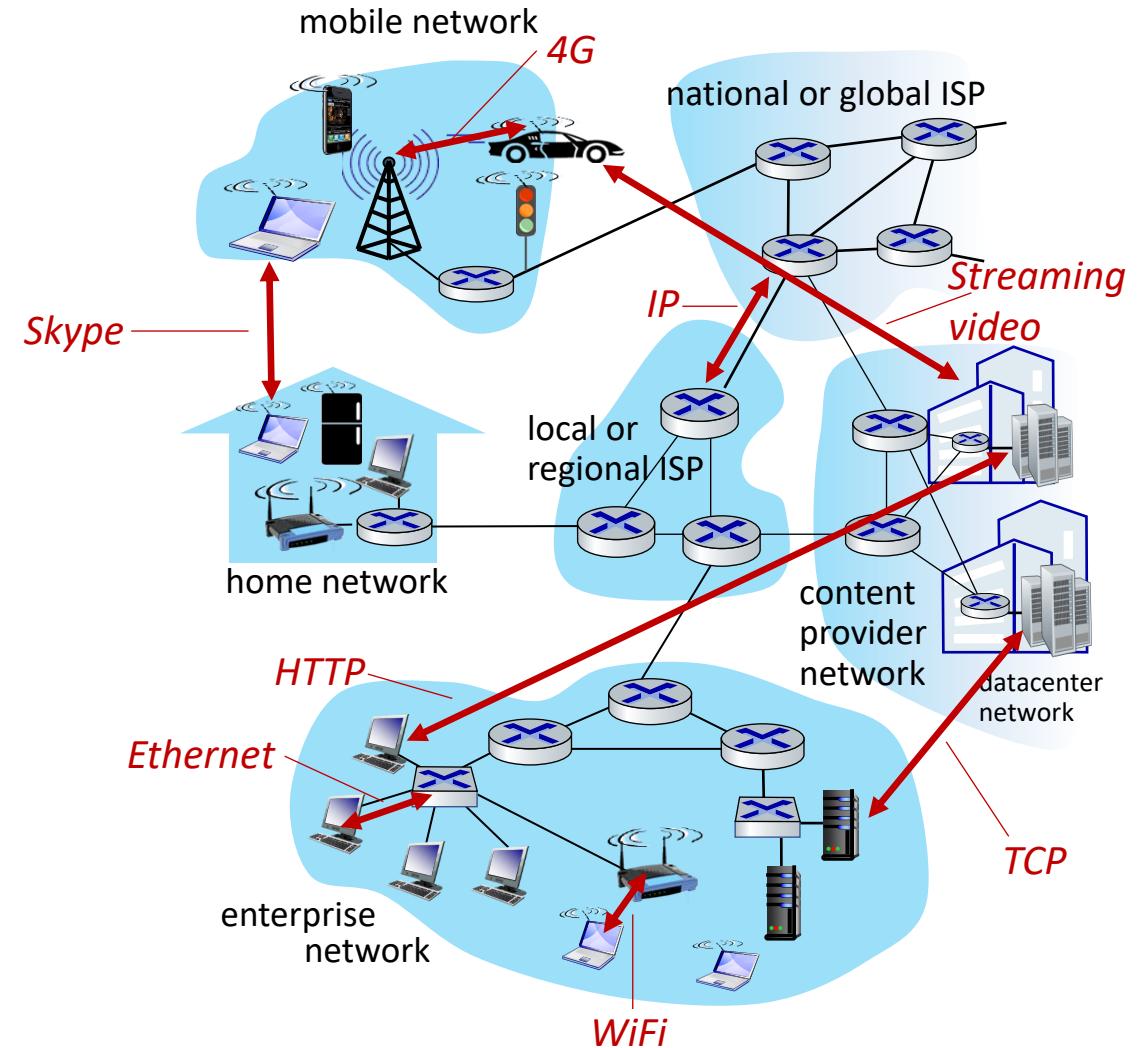
*Networks*

- collection of devices, routers, links: managed by an organization



# The Internet: a “nuts and bolts” view

- *Internet: “network of networks”*
  - Interconnected ISPs
- *protocols are everywhere*
  - control sending, receiving of messages
  - e.g., HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet
- *Internet standards*
  - RFC: Request for Comments
  - IETF: Internet Engineering Task Force



# 有多个标准组织参与数据通信领域的标准制定

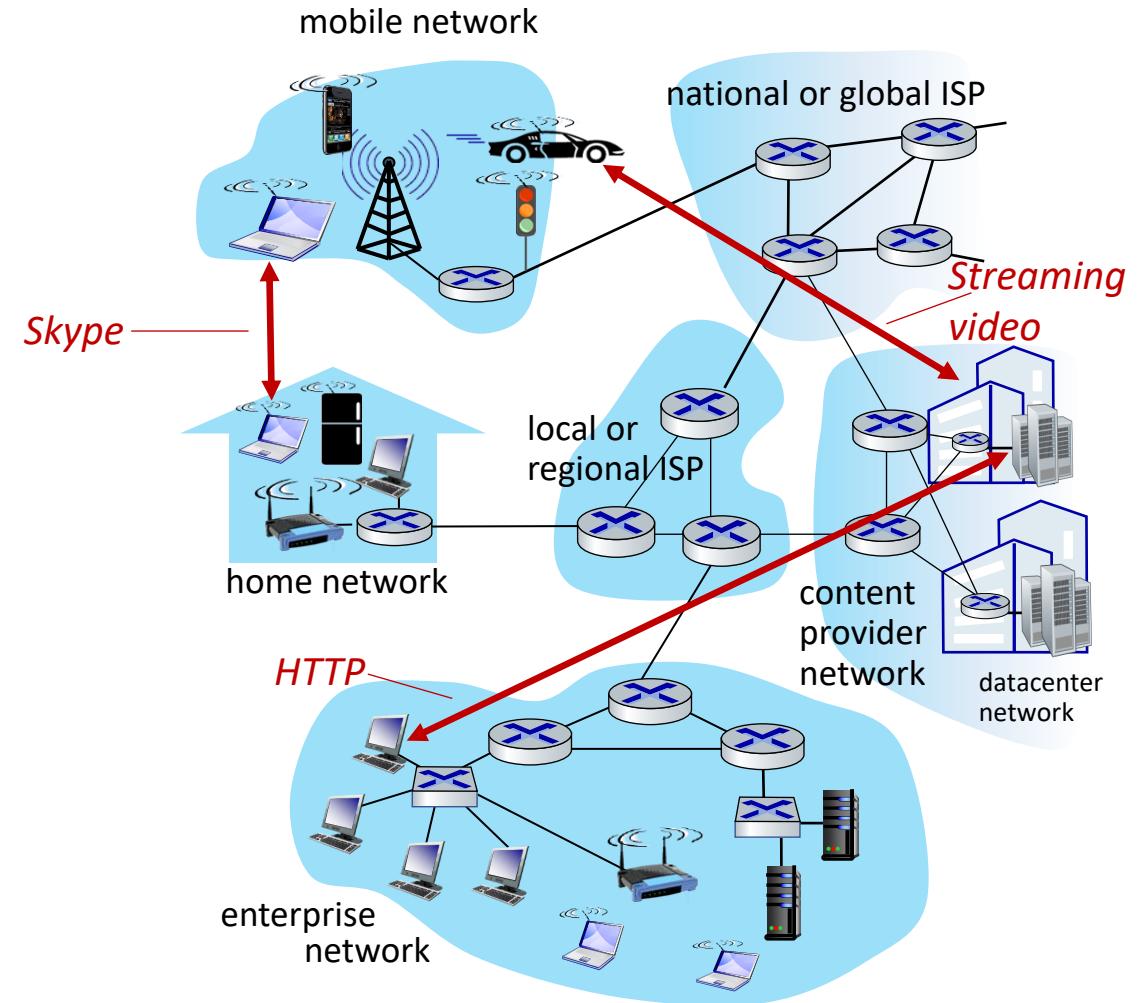
标准组织	在数据通信领域的主要工作	
<b>三大官方国际标准组织</b>		
ITU-T	国际电信联盟	电信业务 IP 化、物联网
ISO	国际标准组织	网络互联模型
IEC	国际电工委员会	机械电气接口和互换性
<b>三大民间国际标准组织</b>		
IETF	互联网工程任务组	网络互联协议，标准主导者
IEEE	电气与电子工程师协会	Ethernet、WLAN
3GPP	第三代合作伙伴计划	无线IP
<b>三大区域性标准组织</b>		
CCSA	中国通信标准协会	设备形态、接口、标准支持
ETSI	欧洲电信标准化协会	宽带接入、终端
ANSI	美国国家标准局	美国标准审批



Source: Huawei

# The Internet: a “service” view

- *Infrastructure* that provides services to applications:
  - Web, streaming video, multimedia teleconferencing, email, games, e-commerce, social media, interconnected appliances, ...
- provides *programming interface* to distributed applications:
  - “hooks” allowing sending/receiving apps to “connect” to, use Internet transport service
  - provides service options, analogous to postal service



# What's a protocol?

## *Human protocols:*

- “what’s the time?”
- “I have a question”
- introductions

... specific messages sent  
... specific actions taken  
when message received,  
or other events

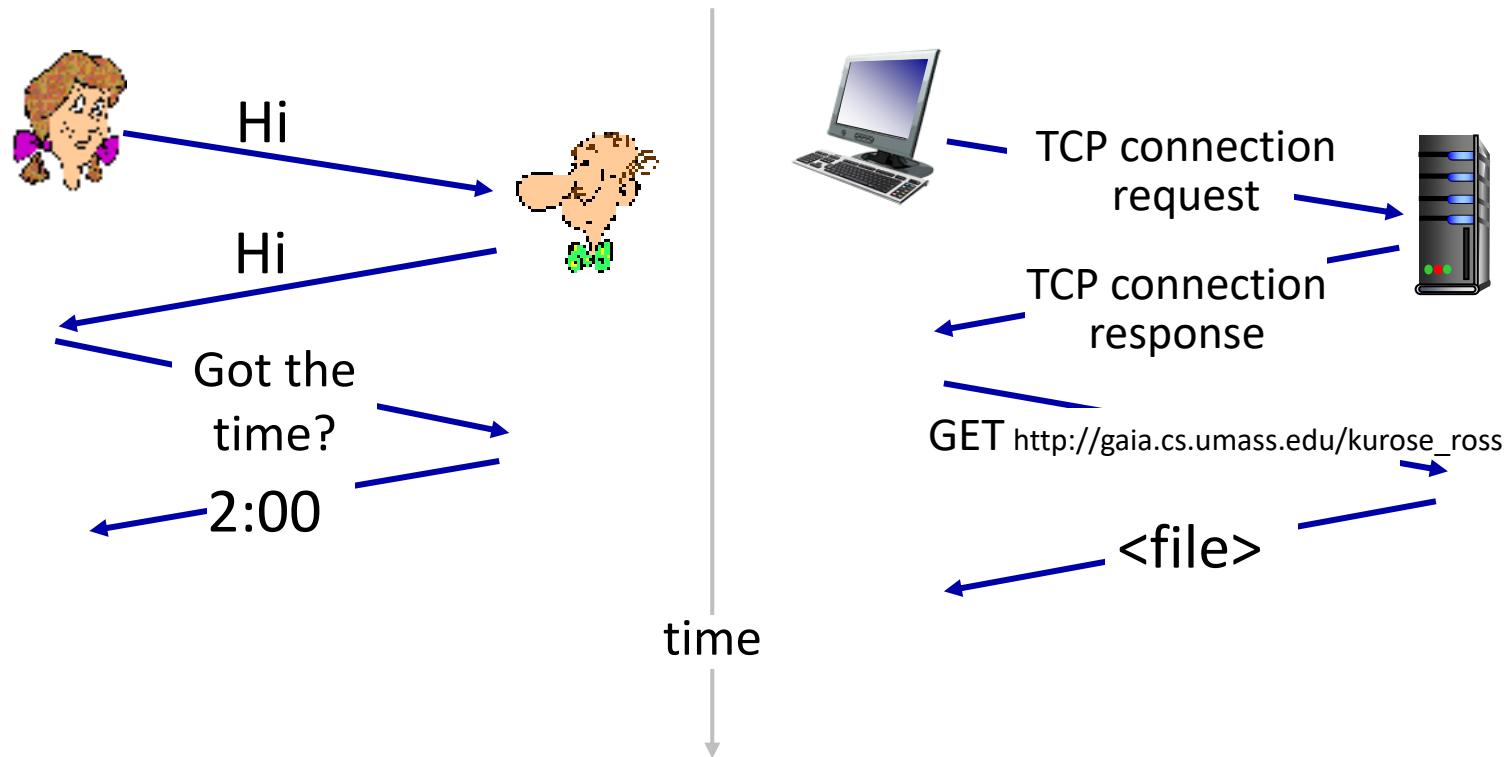
## *Network protocols:*

- computers (devices) rather than humans
- all communication activity in Internet governed by protocols

*Protocols define the **format, order** of messages sent and received among network entities, and **actions taken** on msg transmission, receipt*

# What's a protocol?

A human protocol and a computer network protocol:



*Q:* other human protocols?

# Chapter 1: roadmap

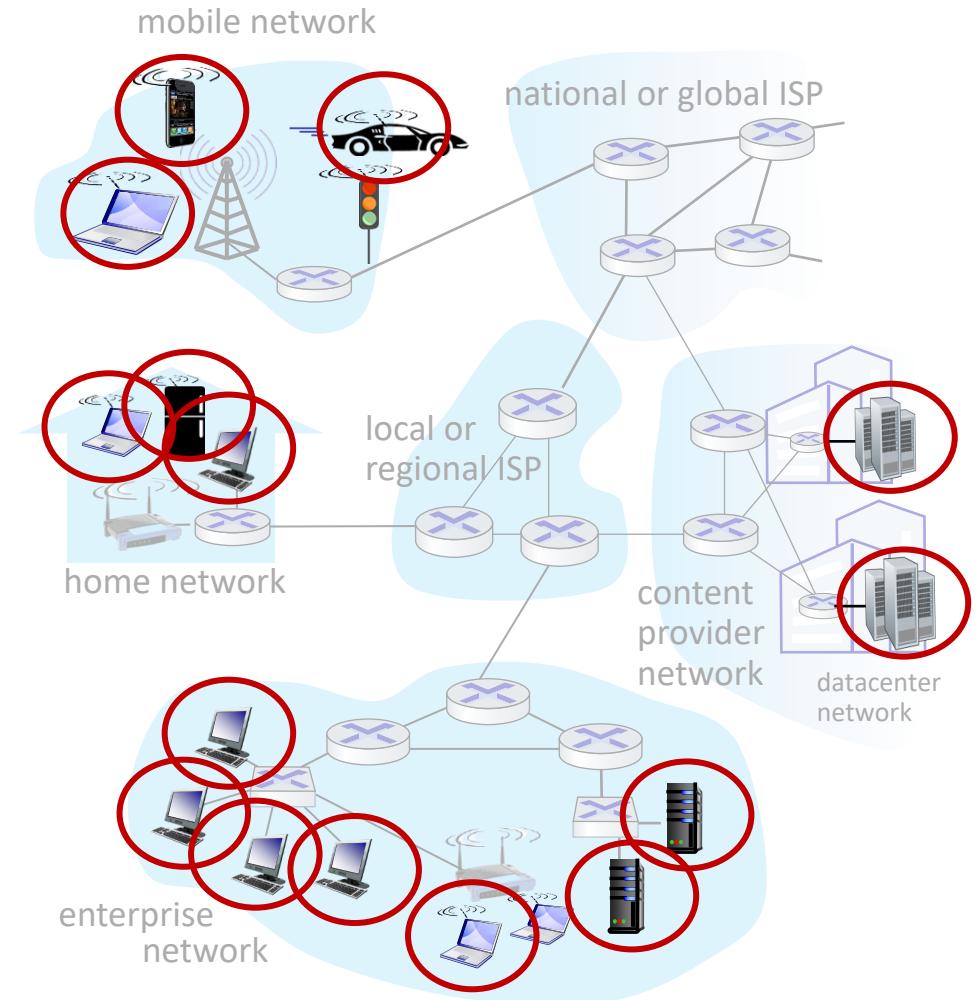
- What *is* the Internet?
- What *is* a protocol?
- **Network edge:** hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Protocol layers, service models
- Performance: loss, delay, throughput
- Security
- History



# A closer look at Internet structure

## Network edge:

- hosts: clients and servers
- servers often in data centers



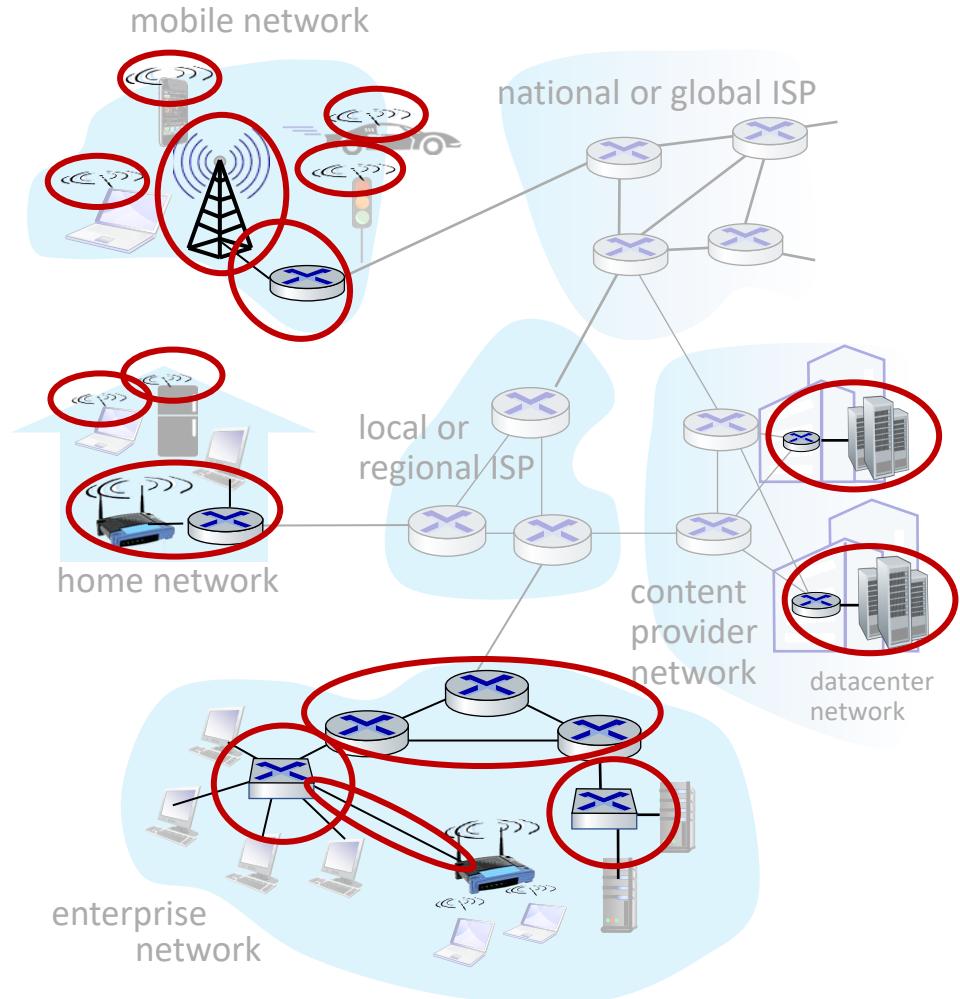
# A closer look at Internet structure

## Network edge:

- hosts: clients and servers
- servers often in data centers

## Access networks, physical media:

- wired, wireless communication links



# A closer look at Internet structure

## Network edge:

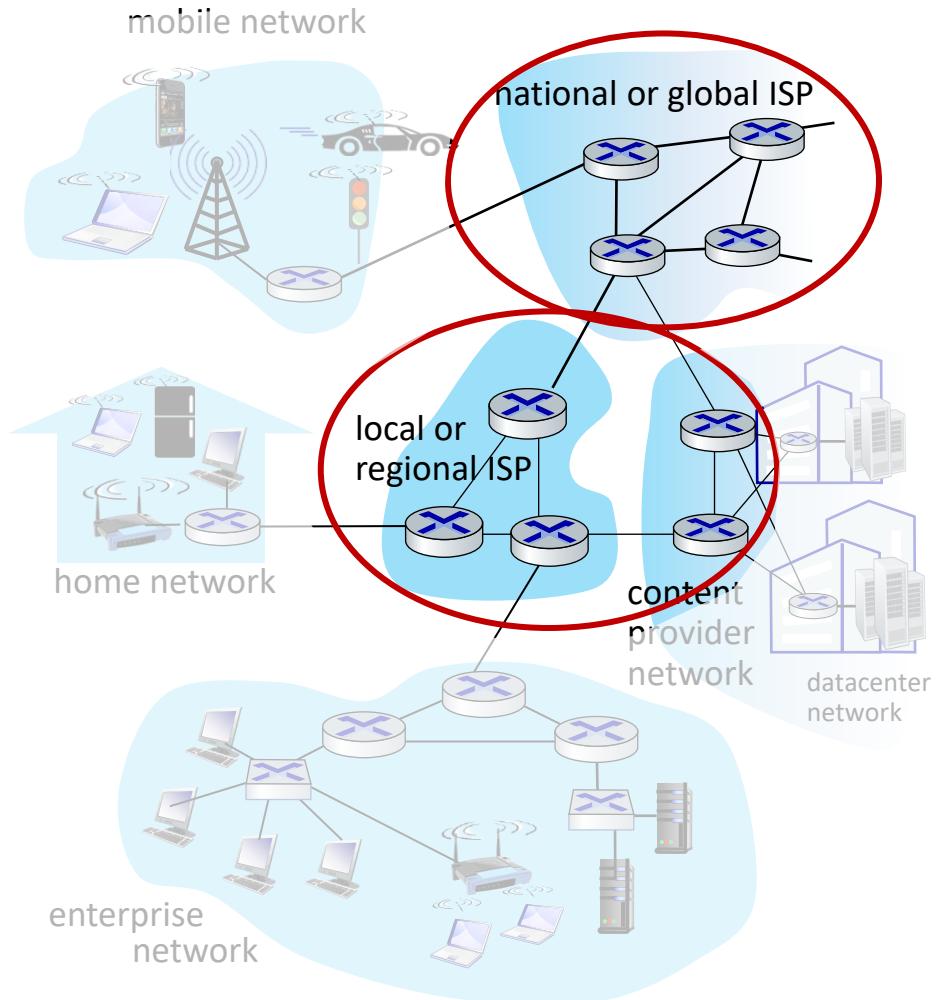
- hosts: clients and servers
- servers often in data centers

## Access networks, physical media:

- wired, wireless communication links

## Network core:

- interconnected routers
- network of networks



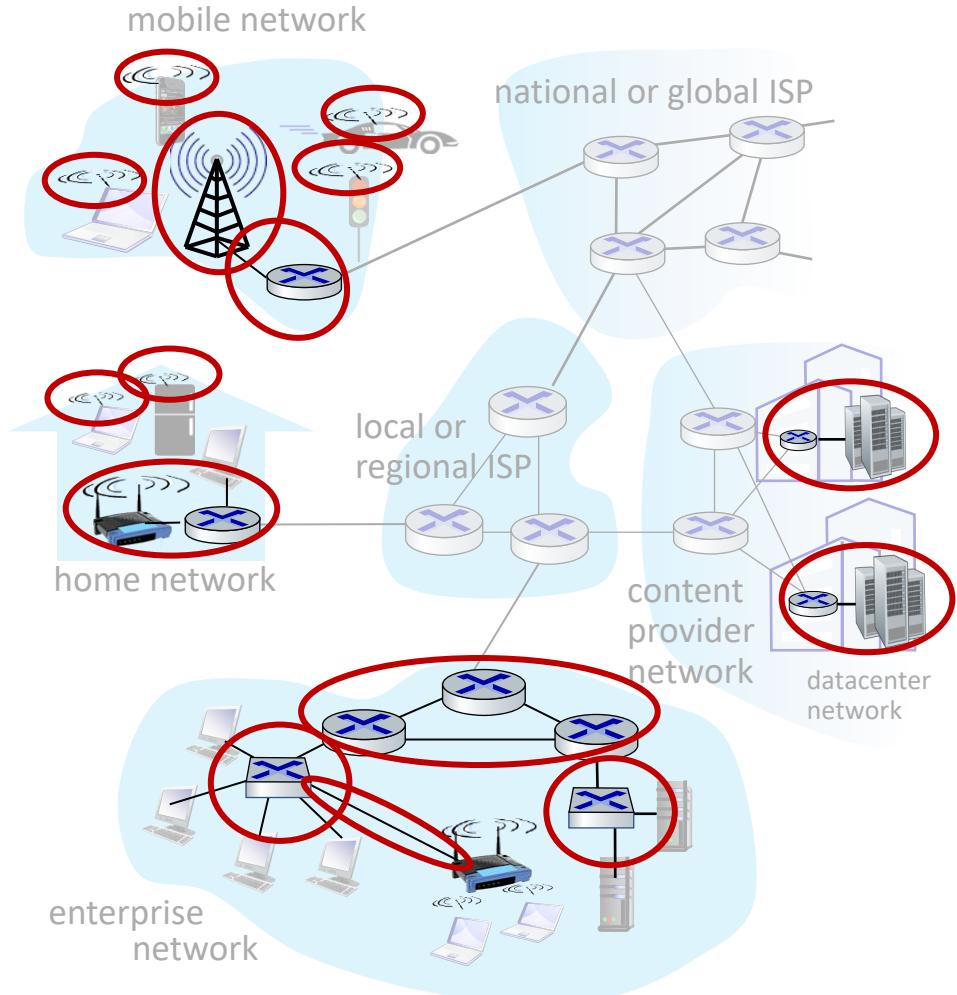
# Access networks and physical media

*Q: How to connect end systems  
to edge router?*

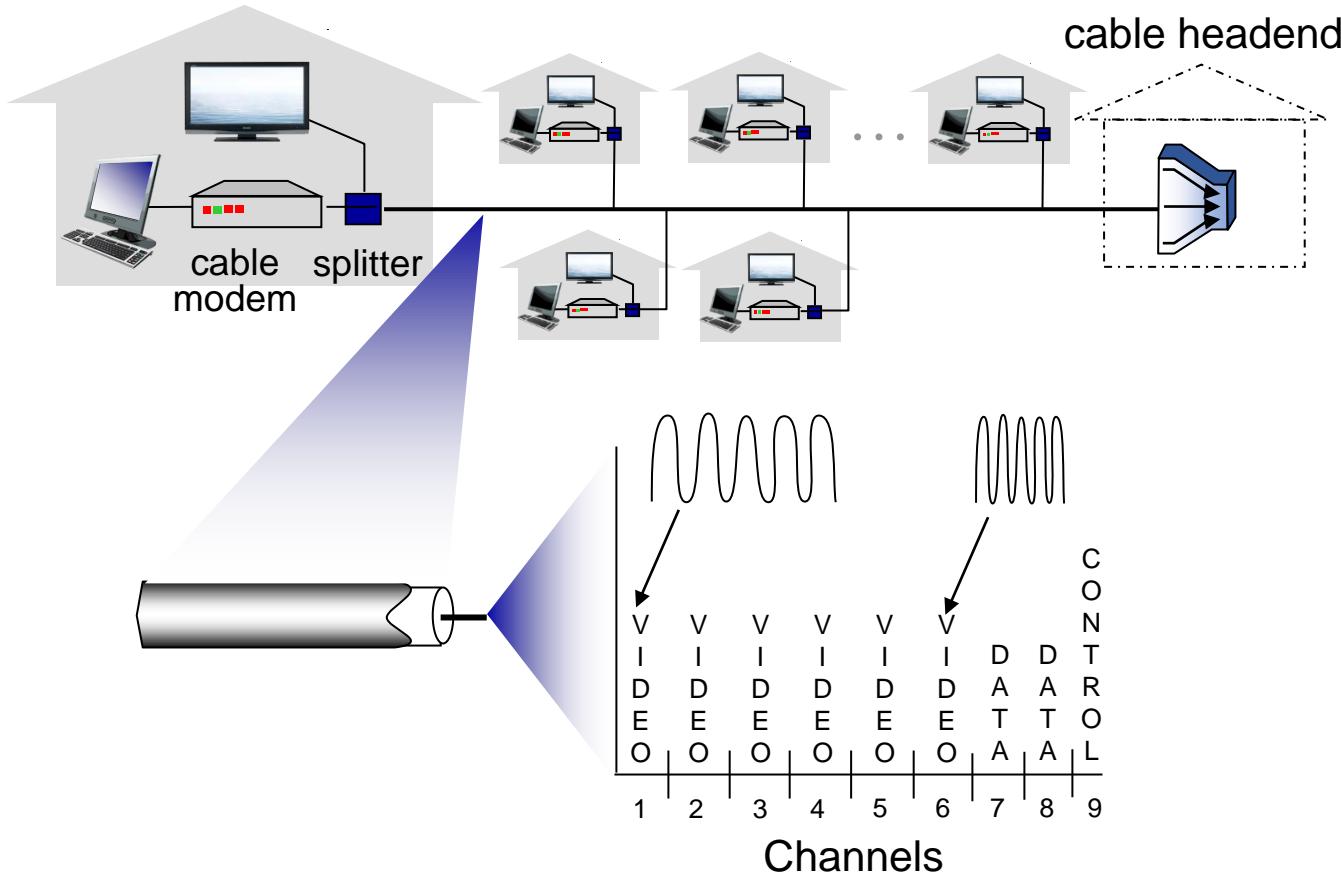
- residential access nets
- institutional access networks (school, company)
- mobile access networks (WiFi, 4G/5G)

*What to look for:*

- transmission rate (bits per second) of access network?
- shared or dedicated access among users?

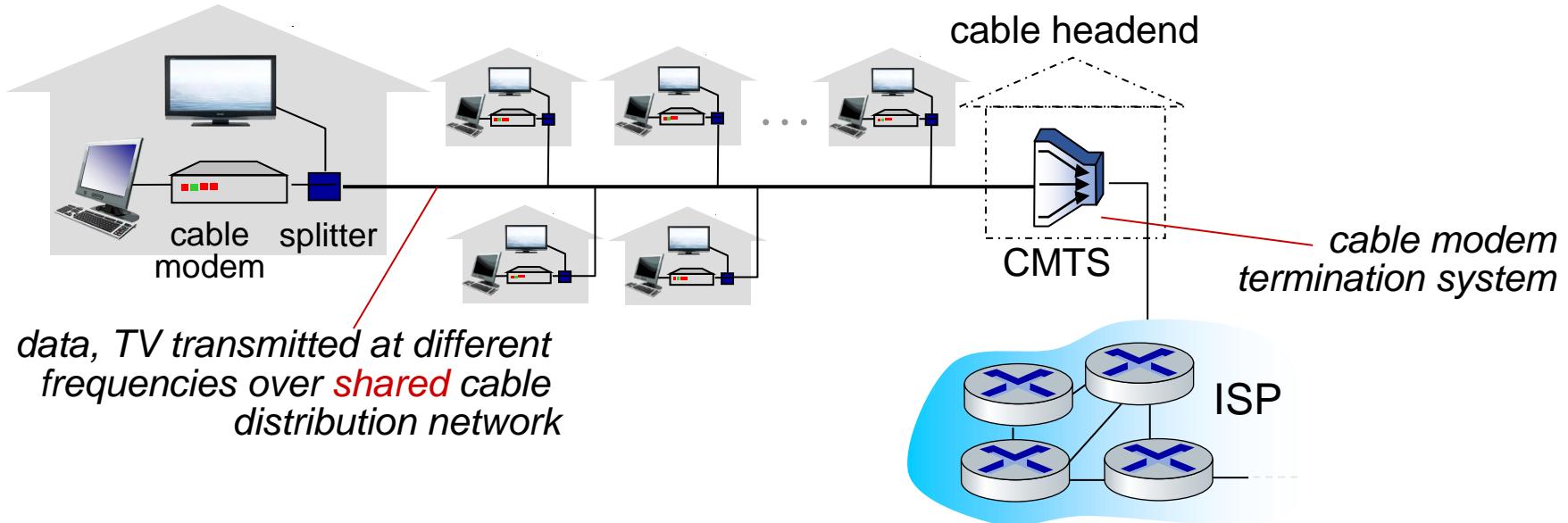


# Access networks: cable-based access



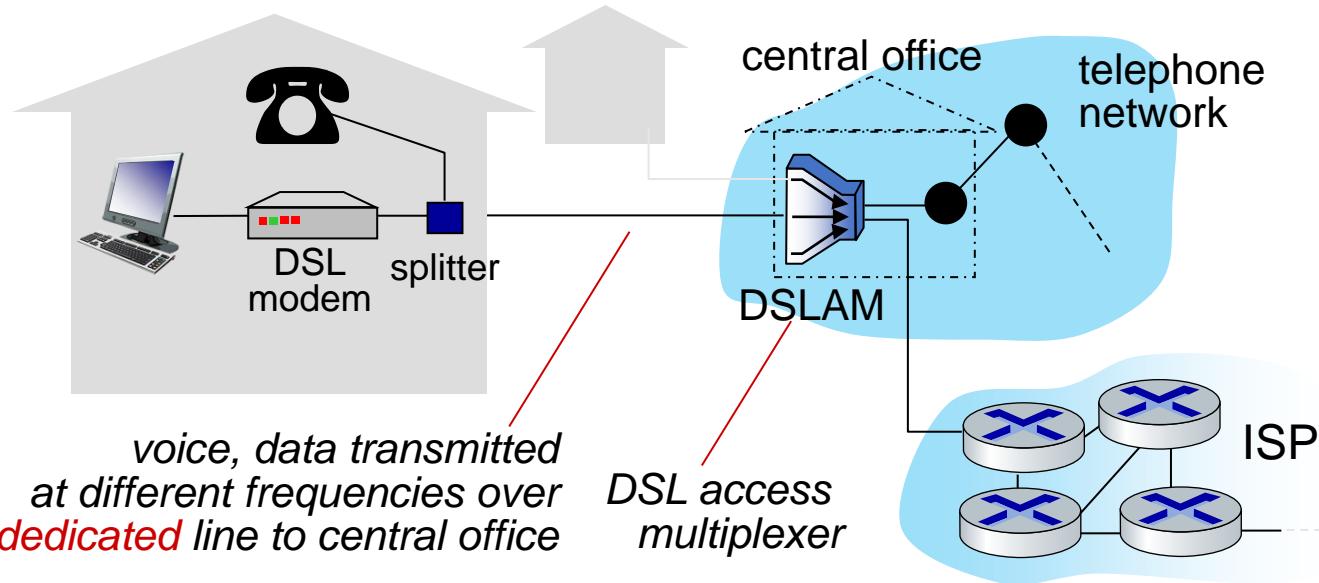
*frequency division multiplexing (FDM)*: different channels transmitted in different frequency bands

# Access networks: cable-based access



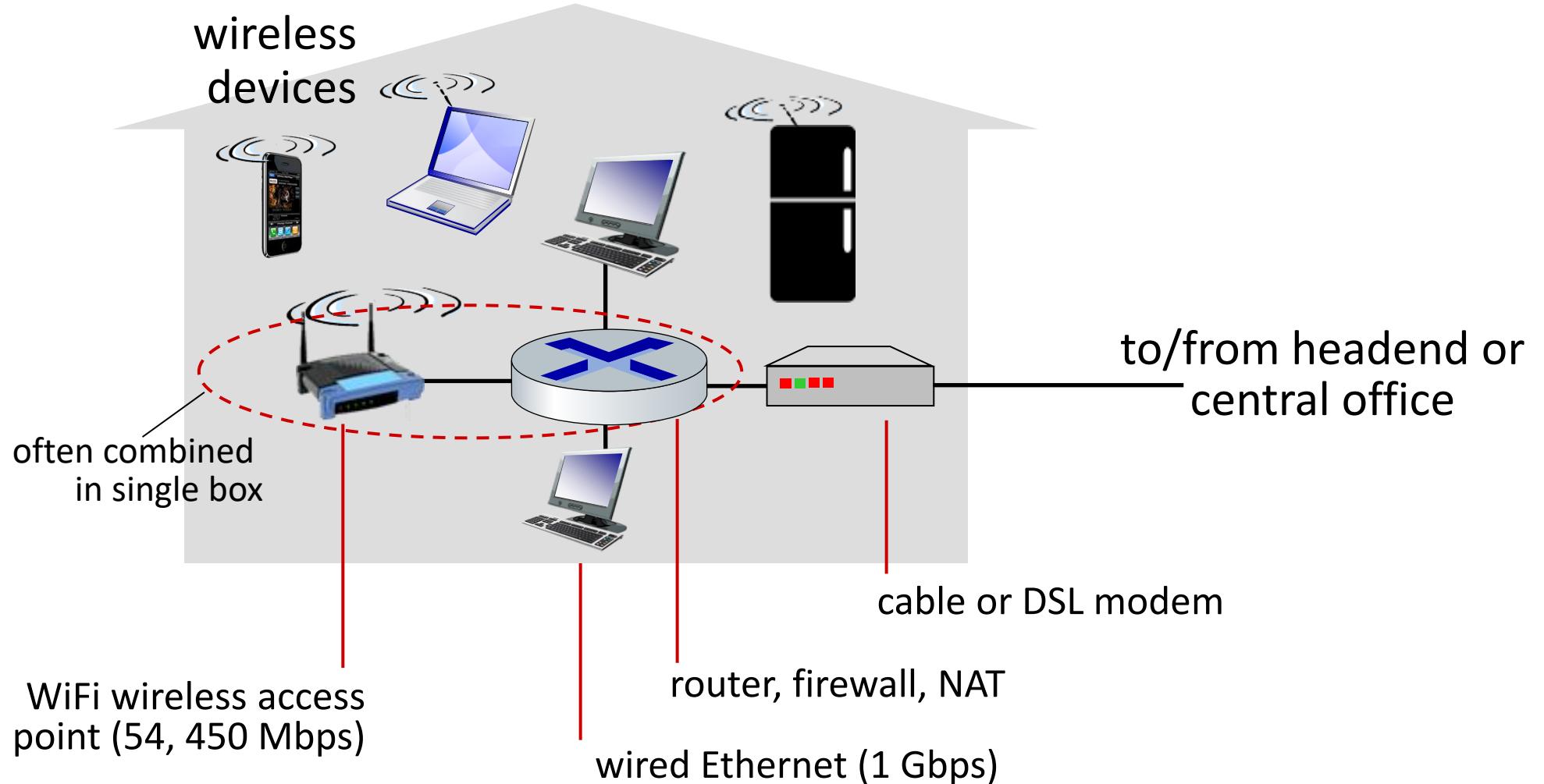
- HFC: hybrid fiber coax
  - *asymmetric*: up to 40 Mbps – 1.2 Gbps downstream transmission rate, 30-100 Mbps upstream transmission rate
- network of cable, fiber attaches homes to ISP router
  - homes *share access network* to cable headend

# Access networks: digital subscriber line (DSL)



- use *existing* telephone line to central office DSLAM
  - data over DSL phone line goes to Internet
  - voice over DSL phone line goes to telephone net
- 24-52 Mbps dedicated downstream transmission rate
- 3.5-16 Mbps dedicated upstream transmission rate

# Access networks: home networks



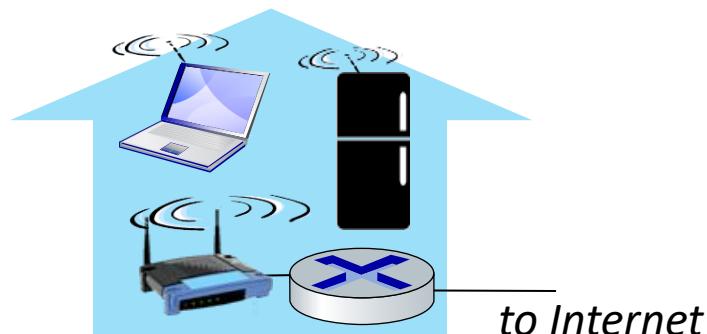
# Wireless access networks

Shared *wireless* access network connects end system to router

- via base station aka “access point”

## Wireless local area networks (WLANs)

- typically within or around building (~100 ft)
- 802.11b/g/n (WiFi): 11, 54, 450 Mbps transmission rate



## Wide-area cellular access networks

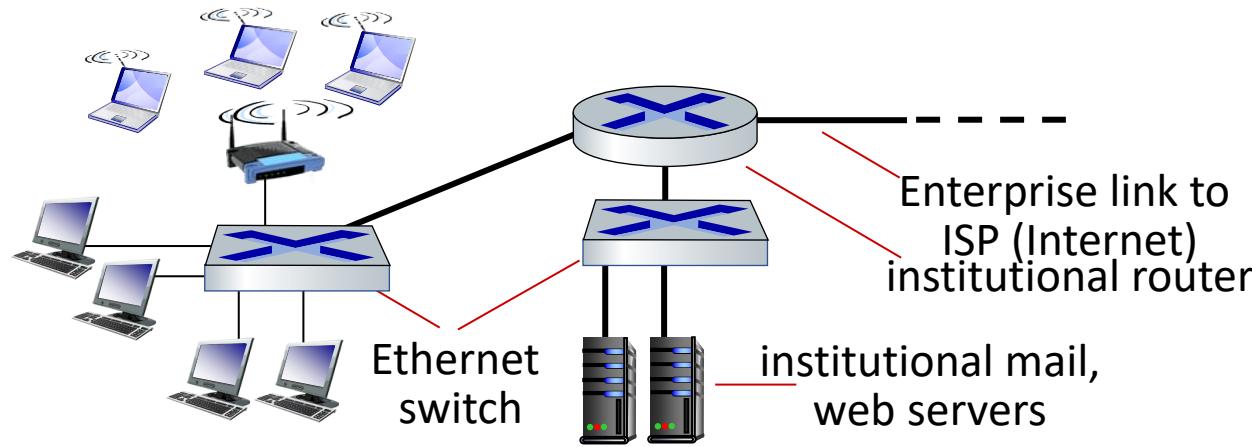
- provided by mobile, cellular network operator (10's km)
- 10's Mbps
- 4G cellular networks (5G coming)



# Wireless access networks

Protocol	Frequency	Channel Width	MIMO	Maximum data rate (theoretical)
802.11ax	2.4 or 5GHz	20, 40, 80, 160MHz	Multi User (MU-MIMO)	2.4 Gbps <sup>1</sup>
802.11ac wave2	5 GHz	20, 40, 80, 160MHz	Multi User (MU-MIMO)	1.73 Gbps <sup>2</sup>
802.11ac wave1	5 GHz	20, 40, 80MHz	Single User (SU-MIMO)	866.7 Mbps <sup>2</sup>
802.11n	2.4 or 5 GHz	20, 40MHz	Single User (SU-MIMO)	450 Mbps <sup>3</sup>
802.11g	2.4 GHz	20 MHz	N/A	54 Mbps
802.11a	5 GHz	20 MHz	N/A	54 Mbps
802.11b	2.4 GHz	20 MHz	N/A	11 Mbps
Legacy 802.11	2.4 GHz	20 MHz	N/A	2 Mbps

# Access networks: enterprise networks

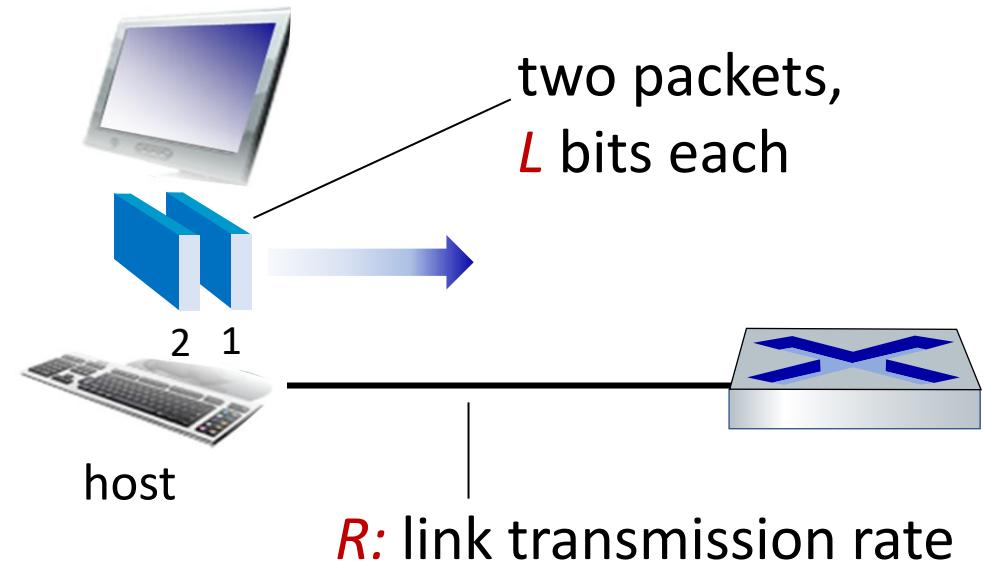


- companies, universities, etc.
- mix of wired, wireless link technologies, connecting a mix of switches and routers (we'll cover differences shortly)
  - Ethernet: wired access at 100Mbps, 1Gbps, 10Gbps
  - WiFi: wireless access points at 11, 54, 450 Mbps

# Host: sends *packets* of data

host sending function:

- takes application message
- breaks into smaller chunks, known as *packets*, of length  $L$  bits
- transmits packet into access network at *transmission rate R*
  - link transmission rate, aka link *capacity, aka link bandwidth*



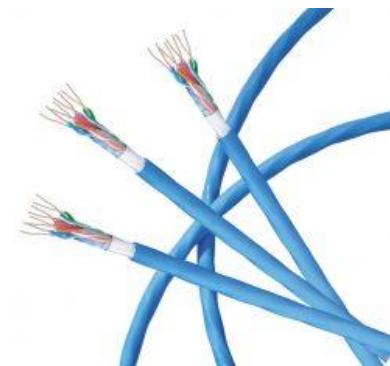
$$\text{packet transmission delay} = \frac{\text{time needed to transmit } L\text{-bit packet into link}}{R \text{ (bits/sec)}}$$

# Links: physical media

- **bit**: propagates between transmitter/receiver pairs
- **physical link**: what lies between transmitter & receiver
- **guided media**:
  - signals propagate in solid media: copper, fiber, coax
- **unguided media**:
  - signals propagate freely, e.g., radio

## Twisted pair (TP)

- two insulated copper wires
  - Category 5: 100 Mbps, 1 Gbps Ethernet
  - Category 6: 10Gbps Ethernet



# Links: physical media

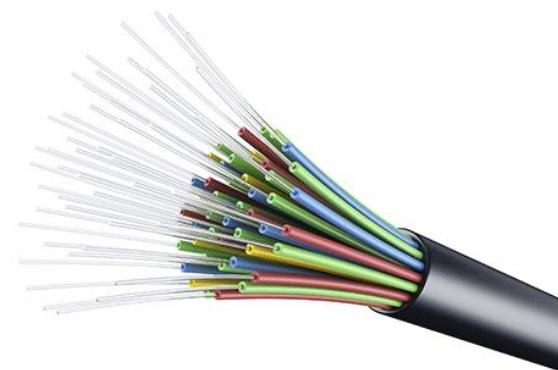
## Coaxial cable:

- two concentric copper conductors
- bidirectional
- broadband:
  - multiple frequency channels on cable
  - 100's Mbps per channel



## Fiber optic cable:

- glass fiber carrying light pulses, each pulse a bit
- high-speed operation:
  - high-speed point-to-point transmission (10's-100's Gbps)
- low error rate:
  - repeaters spaced far apart
  - immune to electromagnetic noise



# Links: physical media

## Wireless radio

- signal carried in electromagnetic spectrum
- no physical “wire”
- broadcast and “half-duplex” (sender to receiver)
- propagation environment effects:
  - reflection
  - obstruction by objects
  - interference

## Radio link types:

- terrestrial microwave
  - up to 45 Mbps channels
- Wireless LAN (WiFi)
  - Up to 100's Mbps
- wide-area (e.g., cellular)
  - 4G cellular: ~ 10's Mbps
- satellite
  - up to 45 Mbps per channel
  - 270 msec end-end delay
  - geosynchronous versus low-earth-orbit

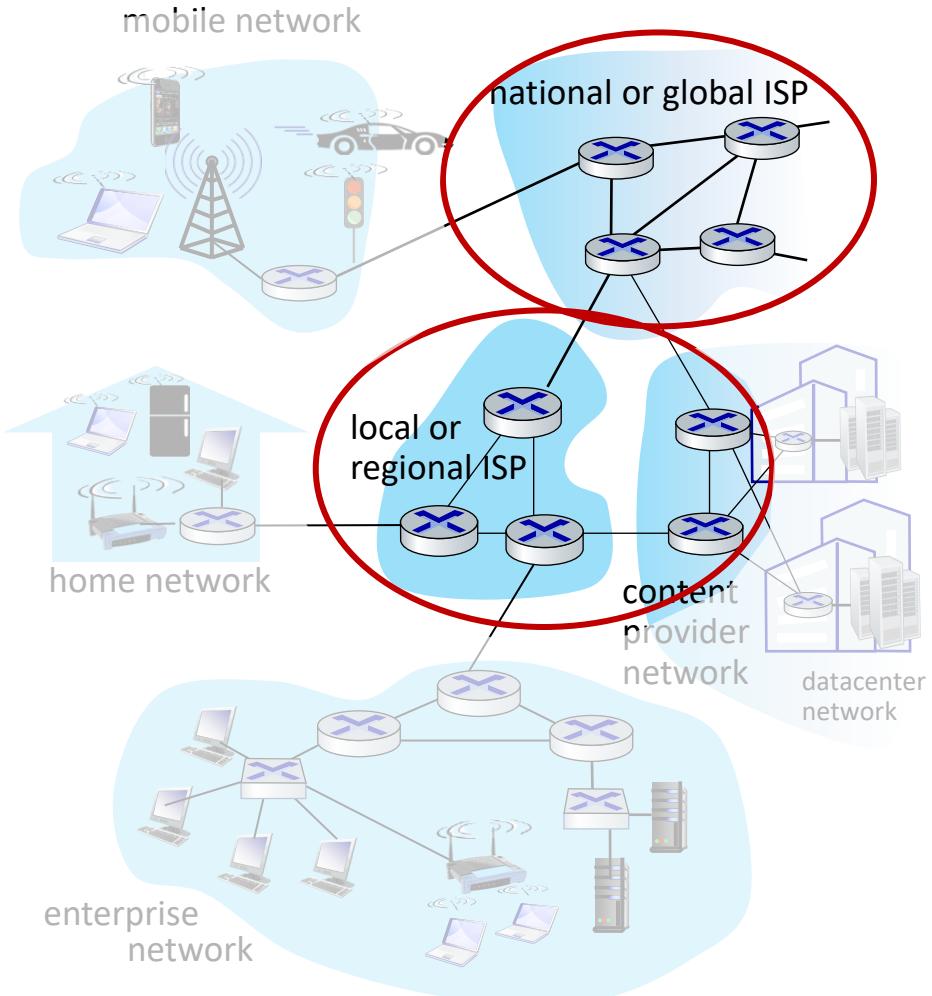
# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- **Network core:** packet/circuit switching, internet structure
- Protocol layers, service models
- Performance: loss, delay, throughput
- Security
- History



# The network core

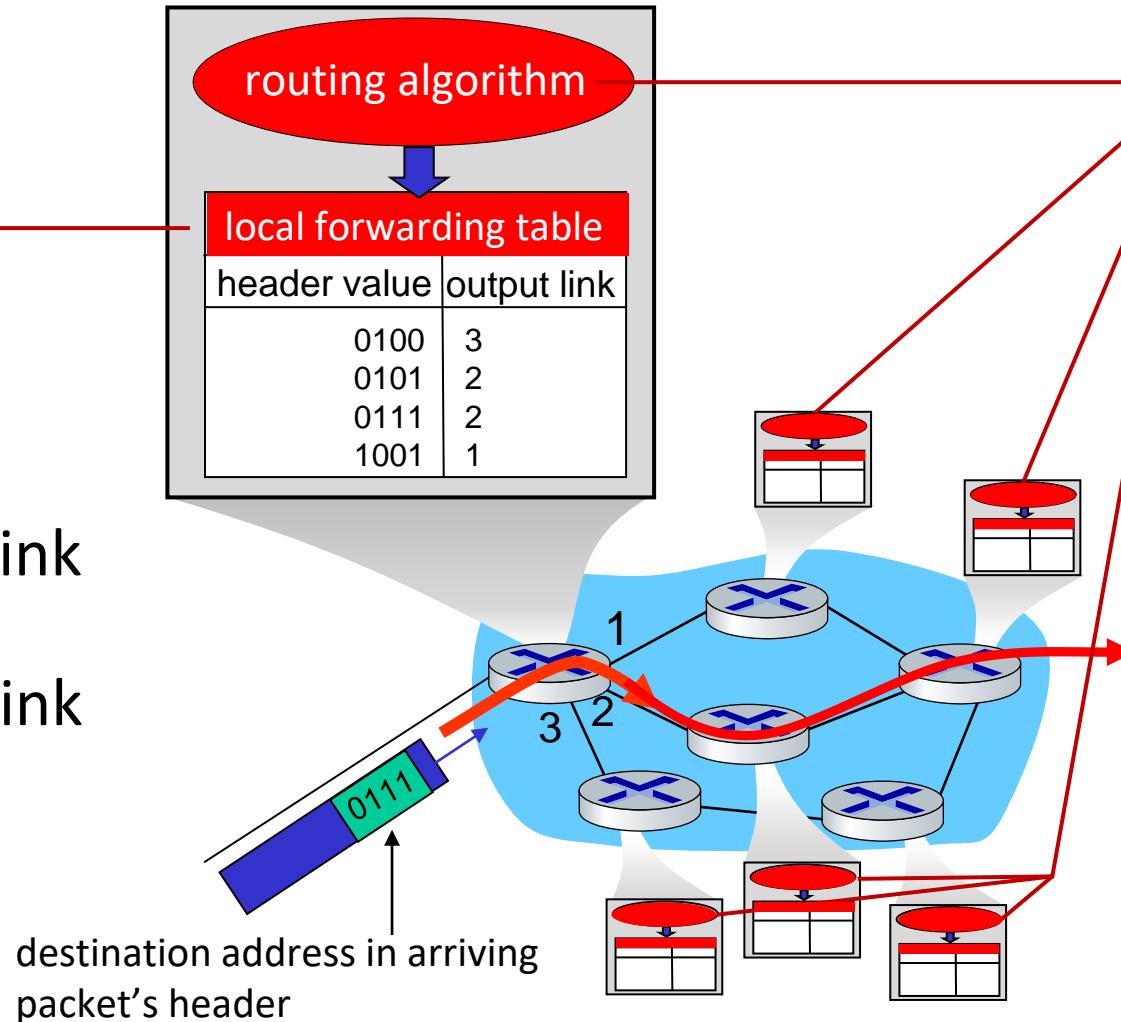
- mesh of interconnected routers
- **packet-switching:** hosts break application-layer messages into *packets*
  - forward packets from one router to the next, across links on path from **source to destination**
  - each packet transmitted at full link capacity



# Two key network-core functions

*Forwarding:*

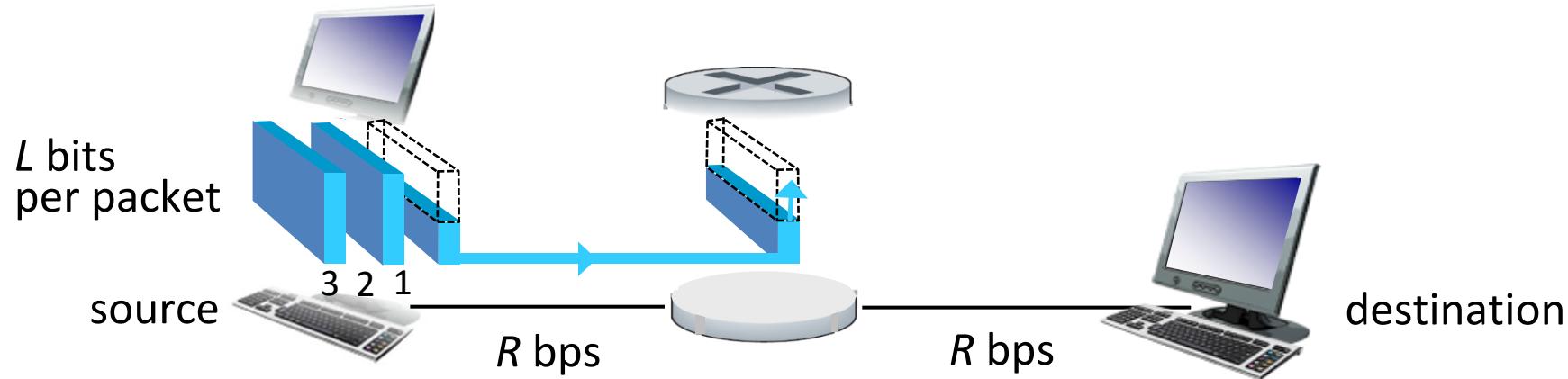
- *local* action:  
move arriving  
packets from  
router's input link  
to appropriate  
router output link



*Routing:*

- *global* action:  
determine source-  
destination paths  
taken by packets
- routing algorithms

# Packet-switching: store-and-forward

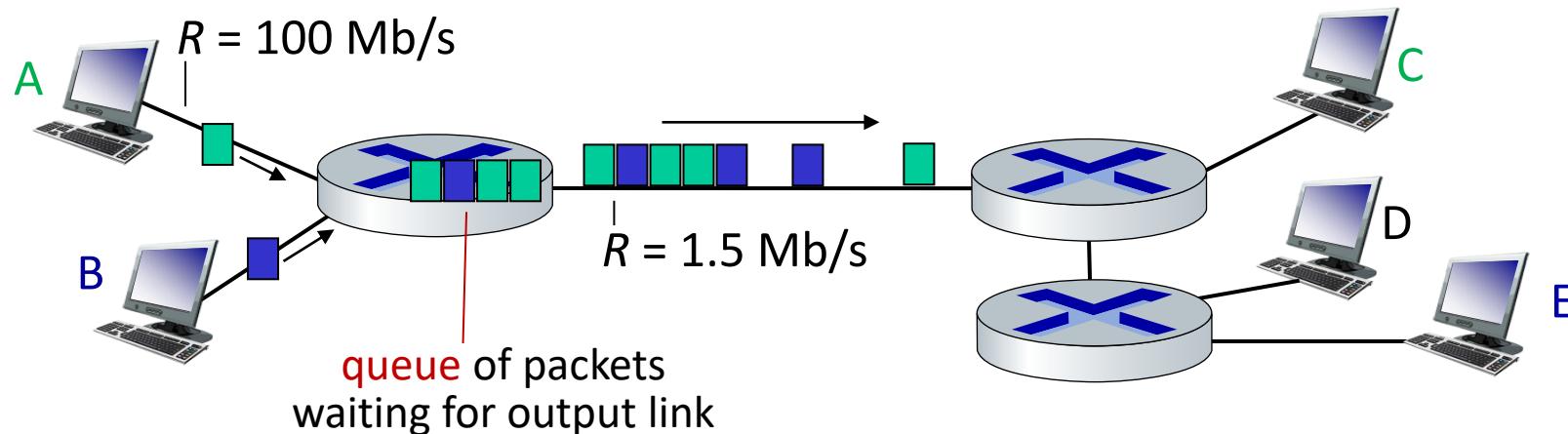


- **Transmission delay:** takes  $L/R$  seconds to transmit (push out)  $L$ -bit packet into link at  $R$  bps
- **Store and forward:** entire packet must arrive at router before it can be transmitted on next link

*One-hop numerical example:*

- $L = 10 \text{ Kbits}$
- $R = 100 \text{ Mbps}$
- one-hop transmission delay = 0.1 msec

# Packet-switching: queueing delay, loss



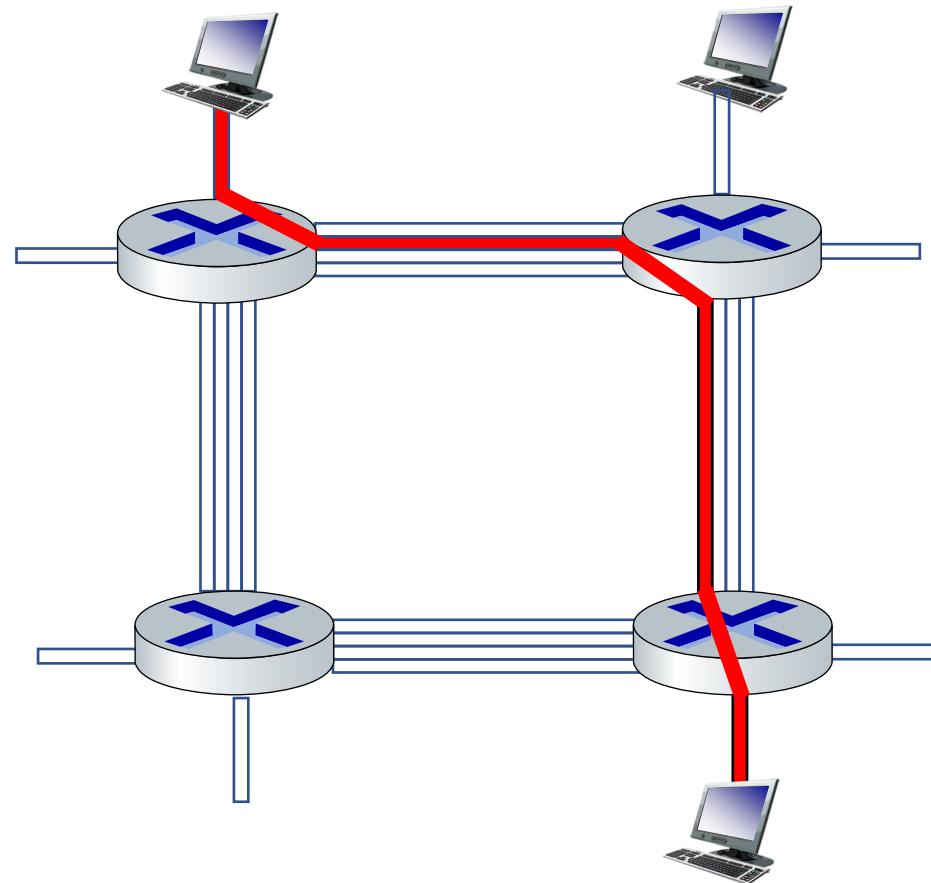
*Packet queuing and loss:* if arrival rate (in bps) to link exceeds transmission rate (bps) of link for a period of time:

- packets will queue, waiting to be transmitted on output link
- packets can be dropped (lost) if memory (buffer) in router fills up

# Alternative to packet switching: circuit switching

end-end resources allocated to,  
reserved for “call” between source  
and destination

- in diagram, each link has four circuits.
  - call gets 2<sup>nd</sup> circuit in top link and 1<sup>st</sup> circuit in right link.
- dedicated resources: no sharing
  - circuit-like (guaranteed) performance
- circuit segment idle if not used by call (no sharing)
- commonly used in traditional telephone networks



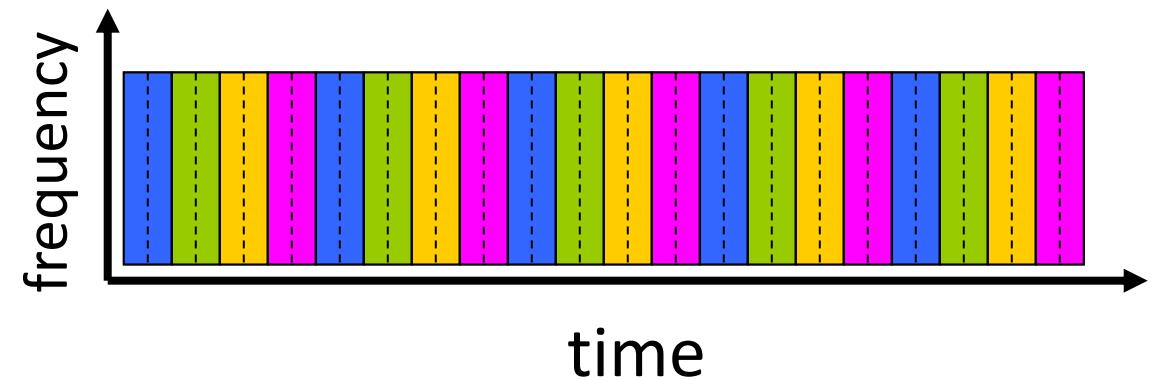
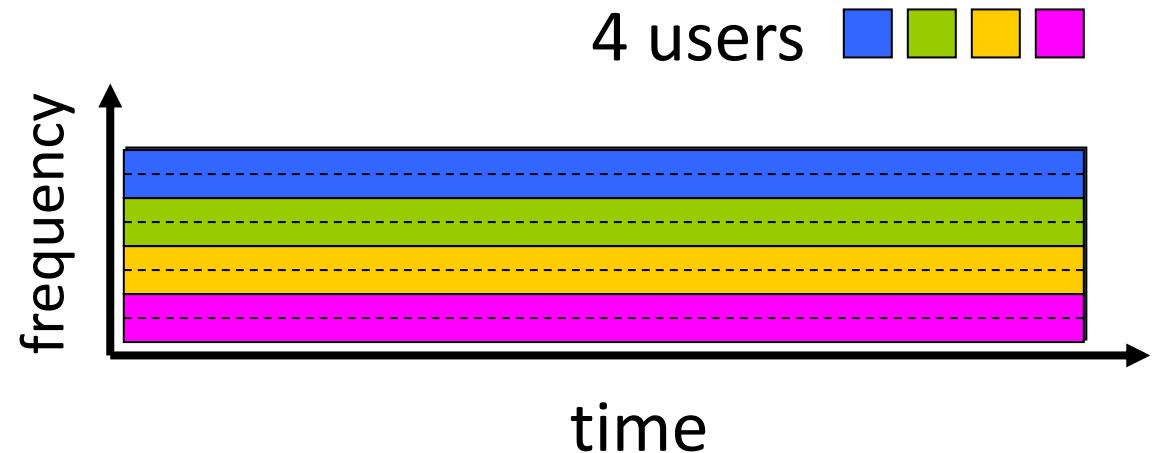
# Circuit switching: FDM and TDM

## Frequency Division Multiplexing (FDM)

- optical, electromagnetic frequencies divided into (narrow) frequency bands
- each call allocated its own band, can transmit at max rate of that narrow band

## Time Division Multiplexing (TDM)

- time divided into slots
- each call allocated periodic slot(s), can transmit at maximum rate of (wider) frequency band, but only during its time slot(s)

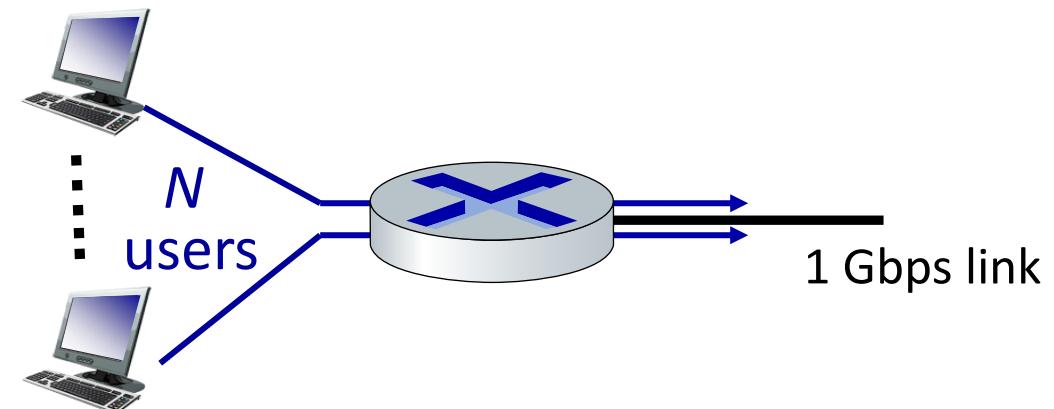


# Packet switching versus circuit switching

*packet switching allows more users to use network!*

Example:

- Capacity: 1 Gb/s link
- each user:
  - 100 Mb/s when “active”
  - active 10% of time
- *circuit-switching*: 10 users
- *packet switching*: with 35 users, probability > 10 active at same time is less than .0004 \*



*Q:* how did we get value 0.0004?

*Q:* what happens if > 35 users ?

# Packet switching versus circuit switching

Is packet switching a “slam dunk winner”?

- great for “bursty” data – sometimes has data to send, but at other times not
  - resource sharing
  - simpler, no call setup
- **excessive congestion possible:** packet delay and loss due to buffer overflow
  - protocols needed for reliable data transfer, congestion control

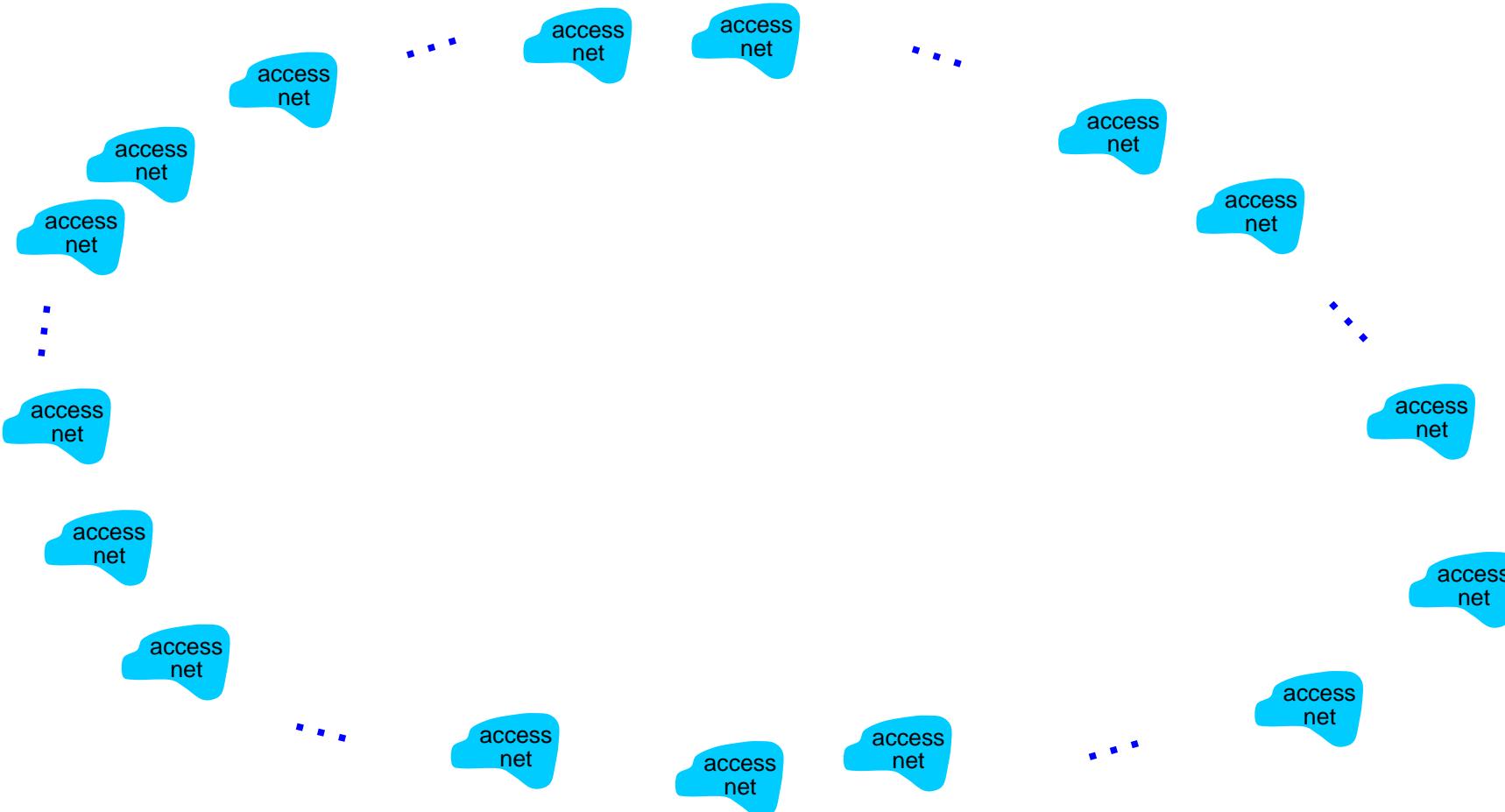
*Q:* human analogies of reserved resources (circuit switching) versus on-demand allocation (packet switching)?

# Internet structure: a “network of networks”

- Hosts connect to Internet via **access** Internet Service Providers (ISPs)
  - residential, enterprise (company, university, commercial) ISPs
- Access ISPs in turn must be interconnected
  - so that any two hosts can send packets to each other
- Resulting network of networks is very complex
  - evolution was driven by **economics** and **national policies**
- Let's take a stepwise approach to describe current Internet structure

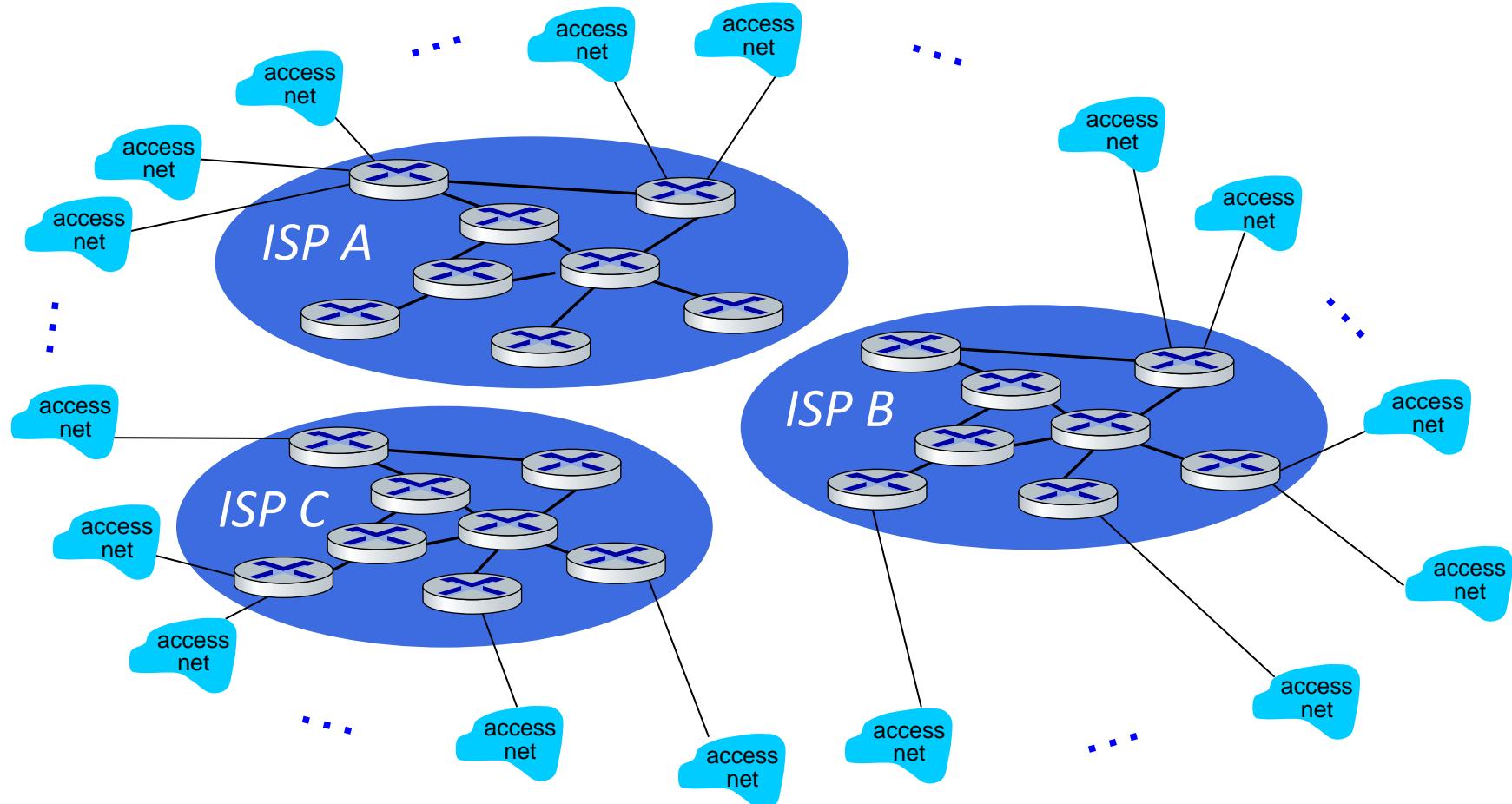
# Internet structure: a “network of networks”

*Question:* given *millions* of access ISPs, how to connect them together?



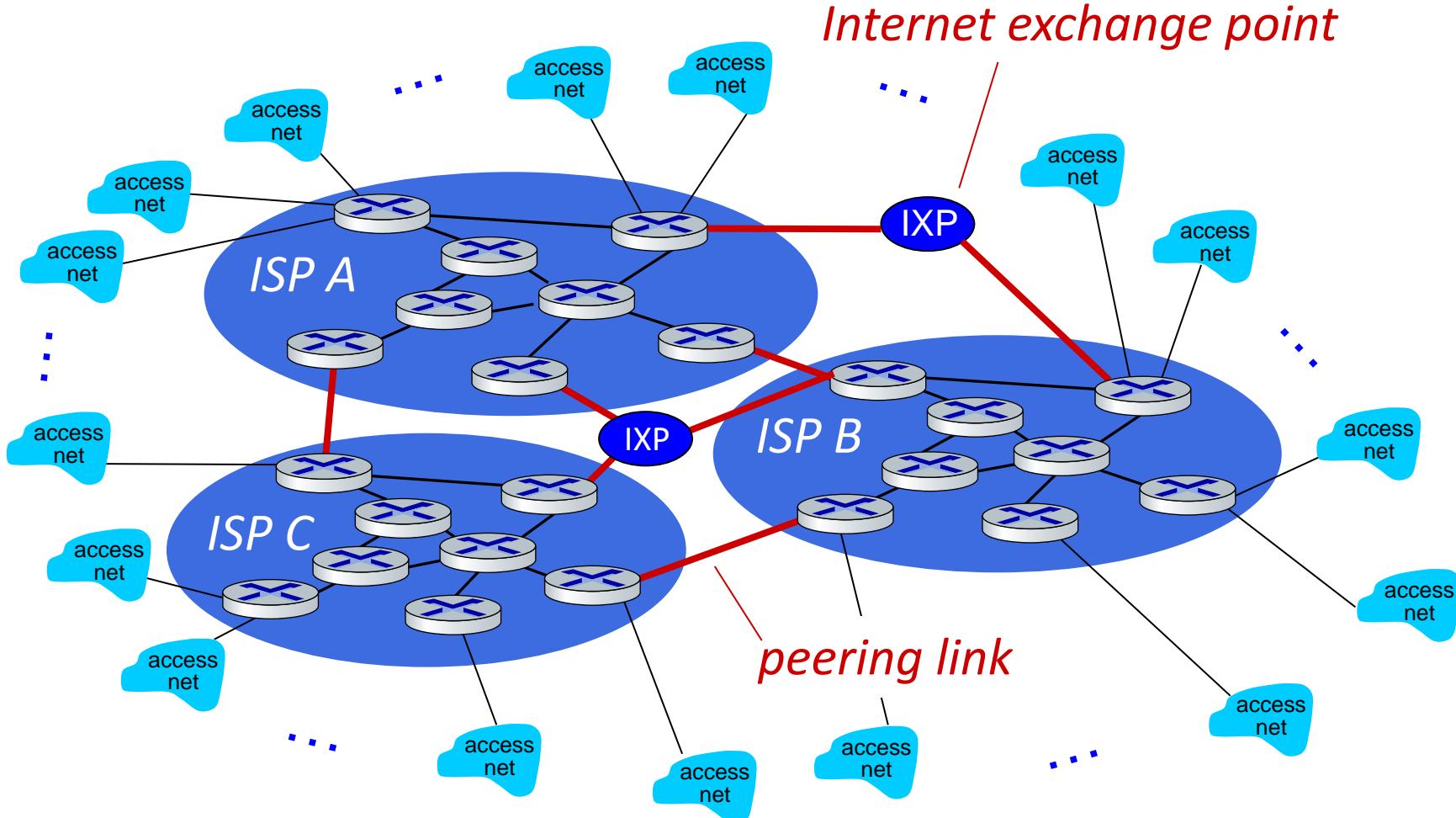
# Internet structure: a “network of networks”

Connected via multiple global ISPs



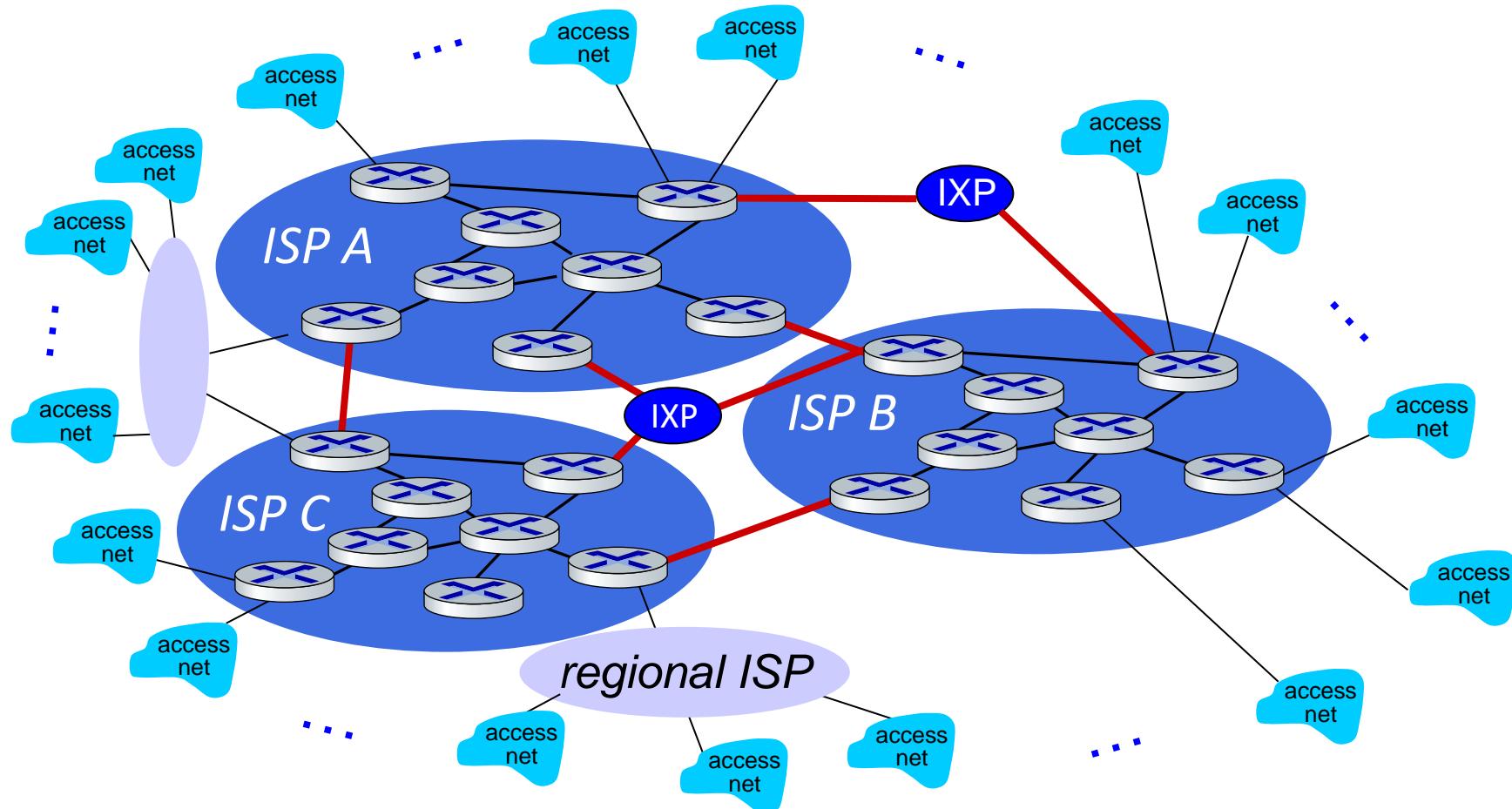
# Internet structure: a “network of networks”

these global ISPs also need to be connected, so that each host can send packets to any hosts



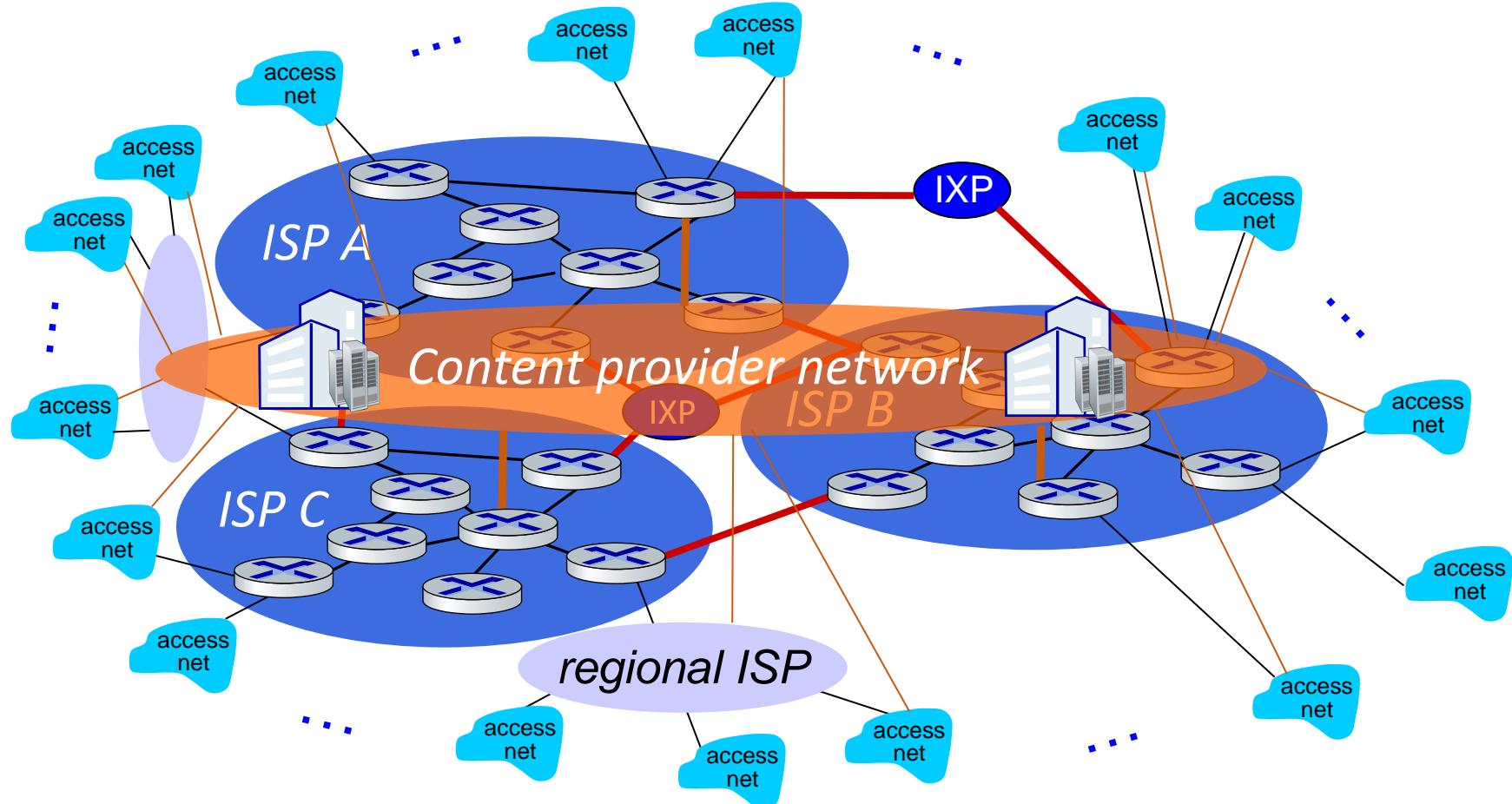
# Internet structure: a “network of networks”

... and regional networks may arise to connect access nets to ISPs

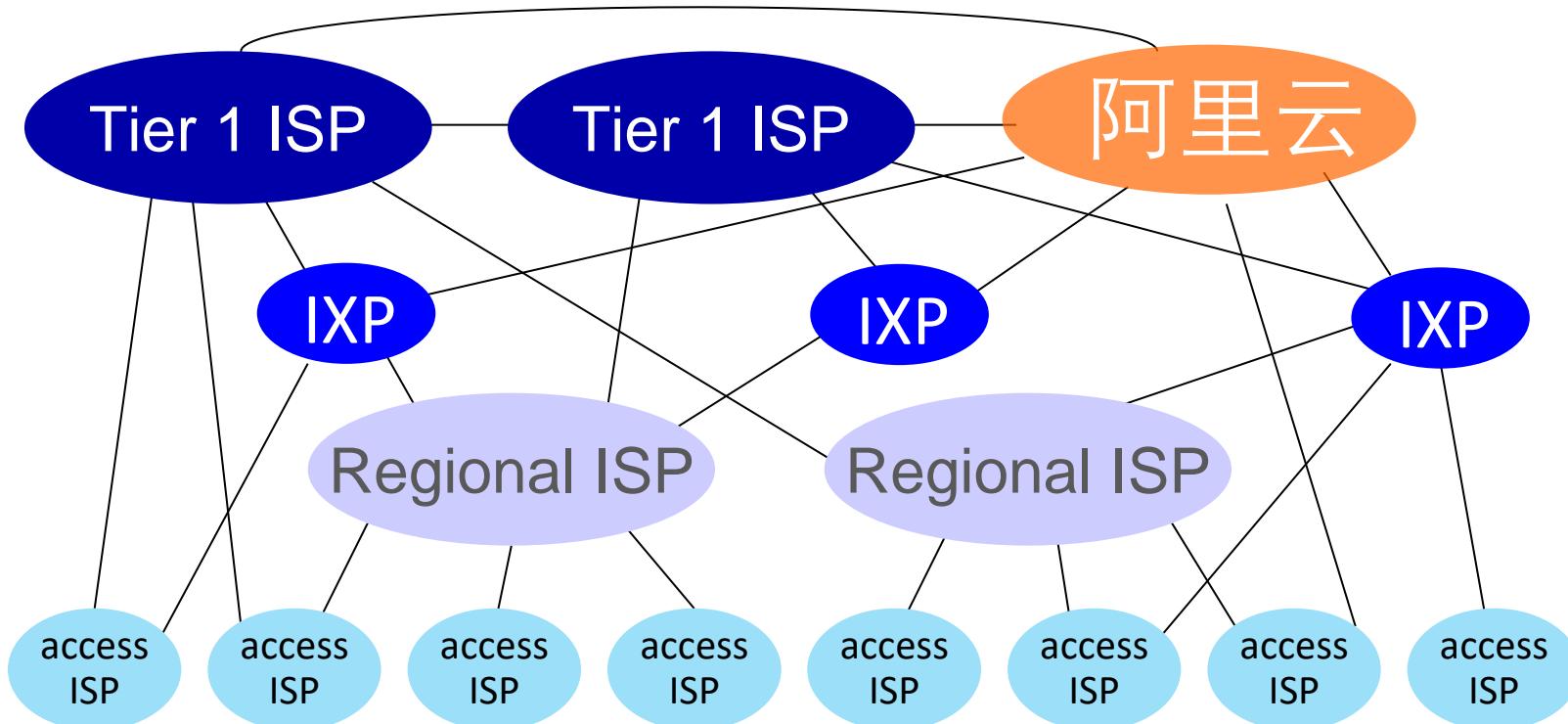


# Internet structure: a “network of networks”

... and content provider networks (e.g., Google, Microsoft, Akamai) may run their own network, to bring services, content close to end users



# Internet structure: a “network of networks”



At “center”: small # of well-connected large networks

- “tier-1” commercial ISPs (e.g., 移动、联通、电信), national & international coverage
- content provider networks (e.g., 阿里云、腾讯云): private network that connects its data centers to Internet, often bypassing tier-1, regional ISPs

# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- **Protocol layers, service models**
- Performance: loss, delay, throughput
- Security
- History



# Protocol “layers” and reference models

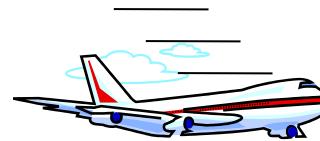
*Networks are complex,  
with many “pieces”:*

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

*Question:*

is there any hope of  
*organizing* structure of  
network?

# Example: organization of air travel



ticket (purchase)  
baggage (check)  
gates (load)  
runway takeoff  
airplane routing

ticket (complain)  
baggage (claim)  
gates (unload)  
runway landing  
airplane routing

airplane routing

airline travel: a series of steps, involving many services

# Example: organization of air travel



*layers:* each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

*Q:* describe in words  
the service provided  
in each layer above

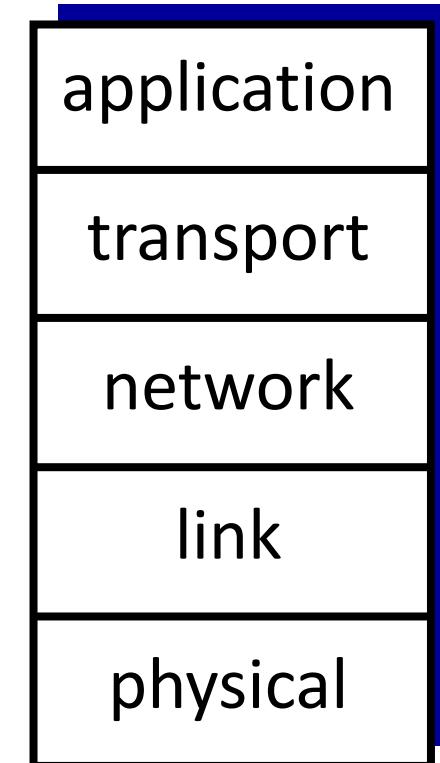
# Why layering?

dealing with complex systems:

- explicit structure allows identification, relationship of complex system's pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change in layer's service *implementation*: transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system

# Internet protocol stack

- *application*: supporting network applications
  - IMAP, SMTP, HTTP
- *transport*: process-process data transfer
  - TCP, UDP
- *network*: routing of datagrams from source to destination
  - IP, routing protocols
- *link*: data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- *physical*: bits “on the wire”



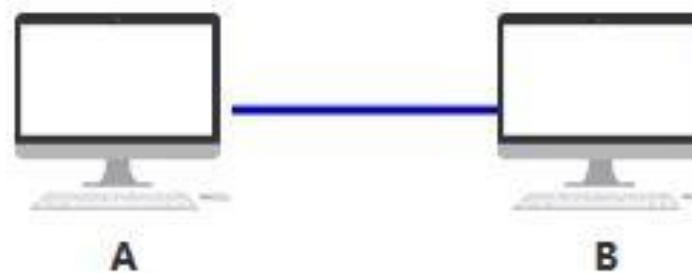
# Internet protocol stack – an example

Source: 低并发编程 WeChat Channel  
<https://mp.weixin.qq.com/s/jiPMUk6zUdOY6eKxAjNDbQ>

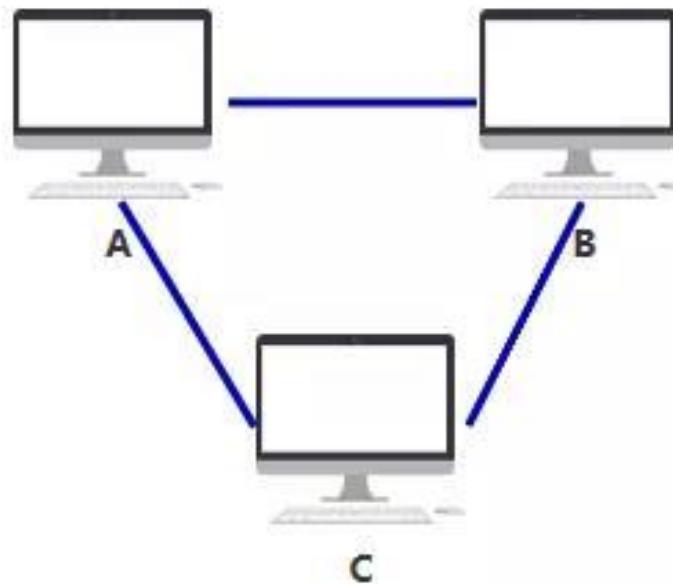
Use this example to understand why the layers are designed in this way!



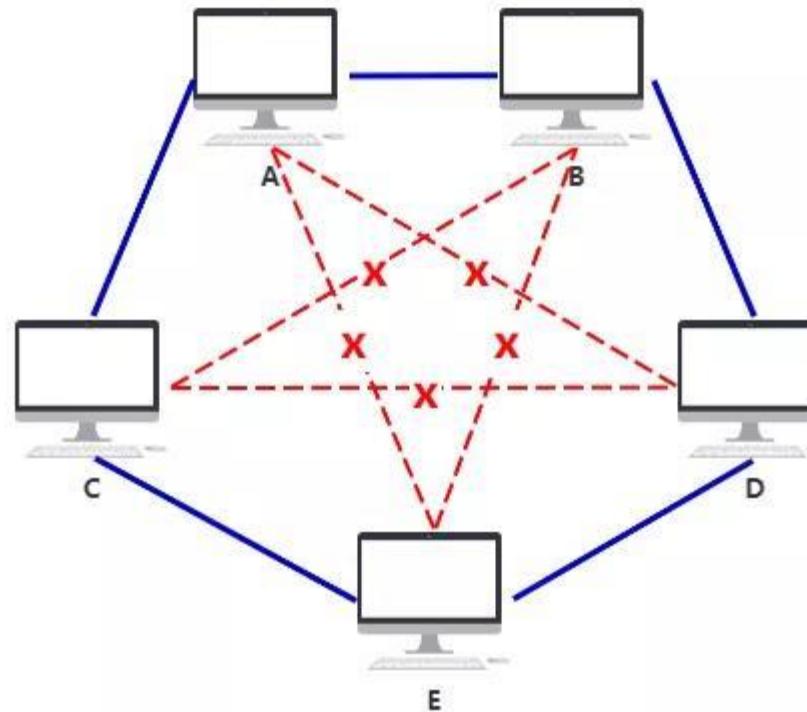
# Internet protocol stack – an example



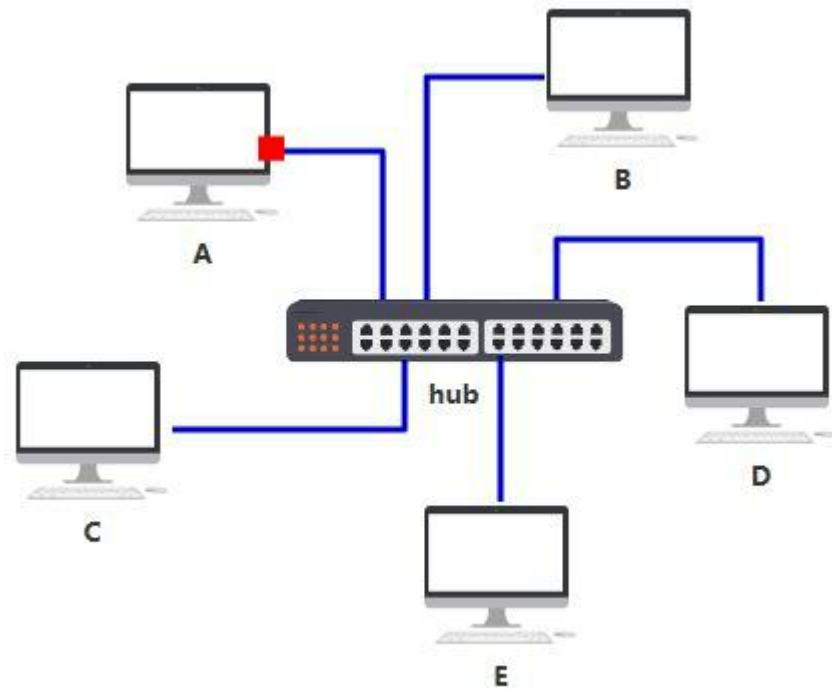
# Internet protocol stack – an example



# Internet protocol stack – an example



# Internet protocol stack – an example



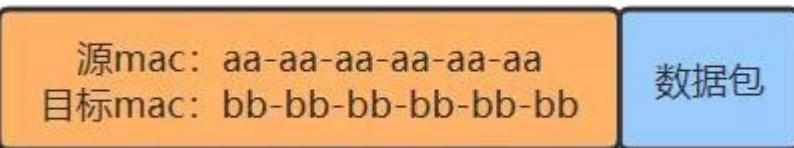
Hub (集线器)

- Broadcast
- Signal **amplification** and signal **regeneration**

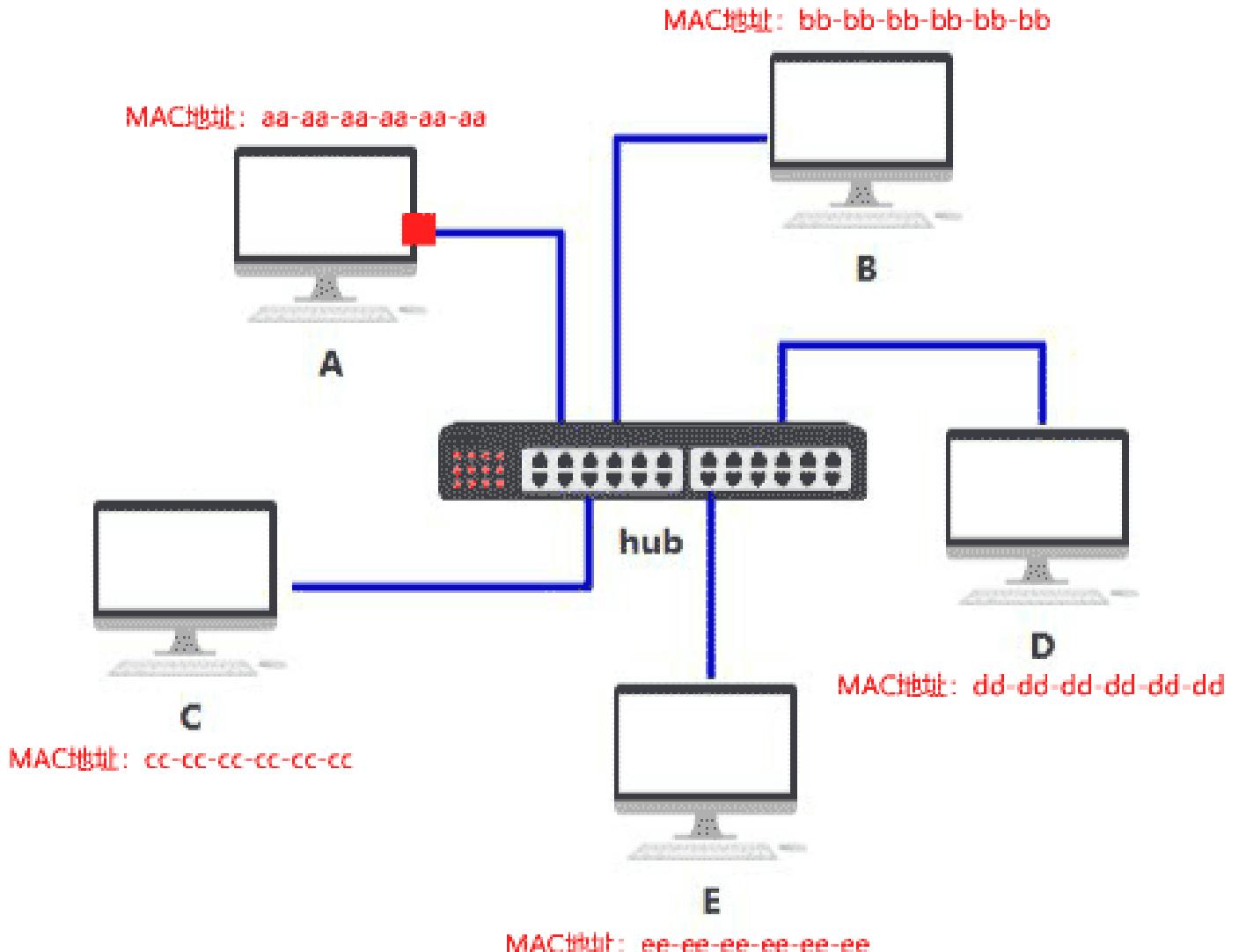
Summary of physical layer

- Bits “on the wire”

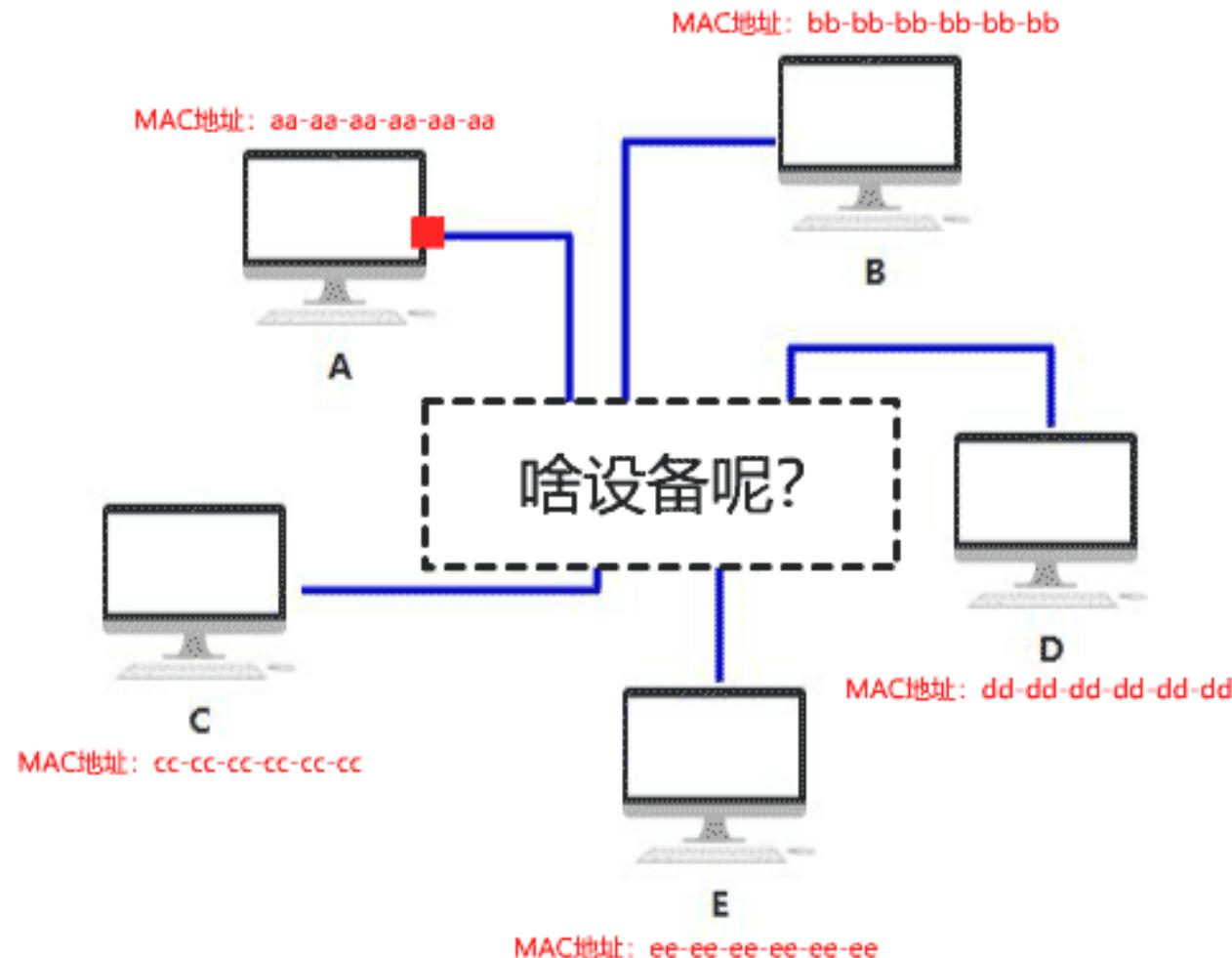
# Internet protocol stack – an example



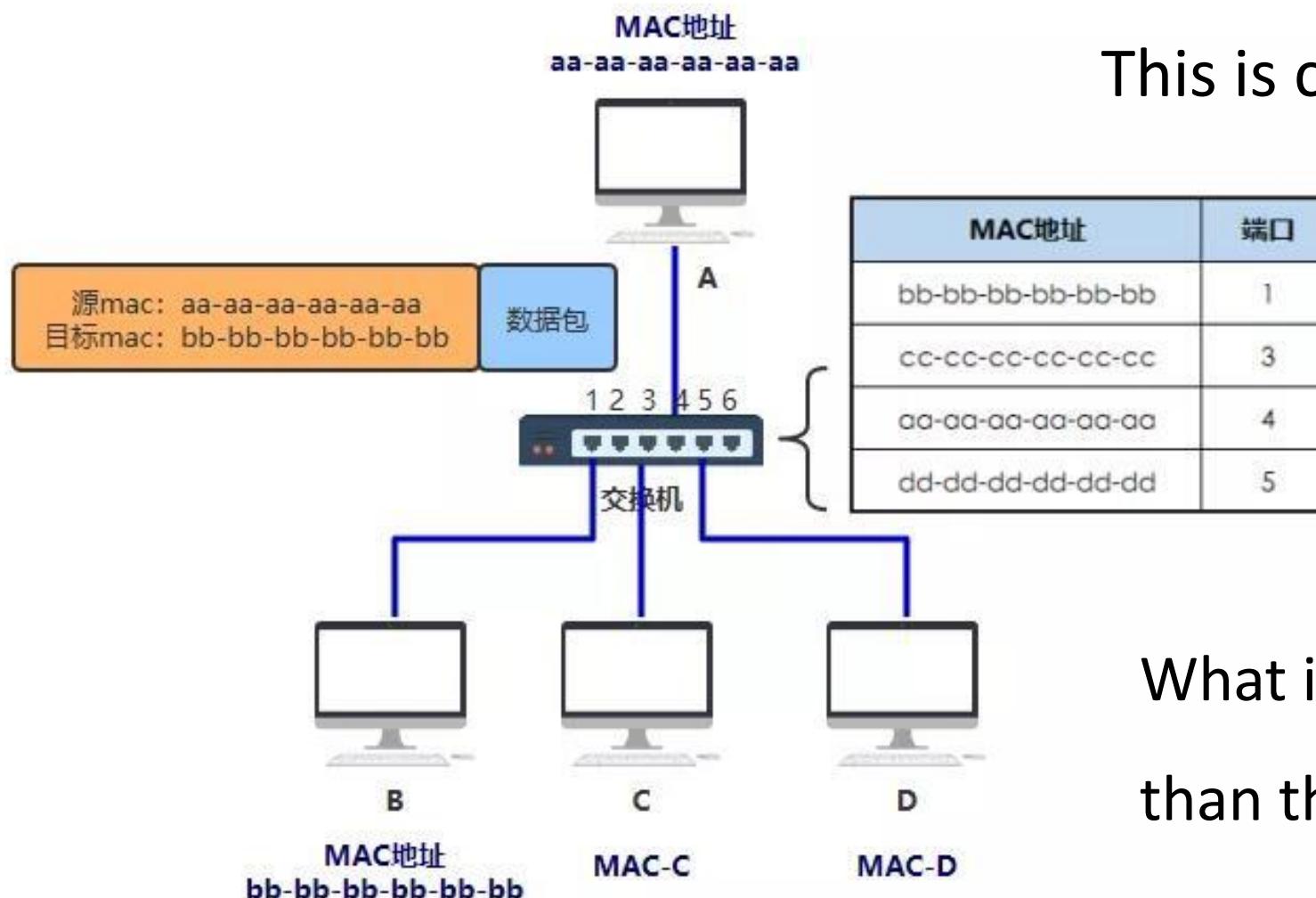
- Media Access Control (MAC) address
- Each network interface has a MAC address



# Internet protocol stack – an example



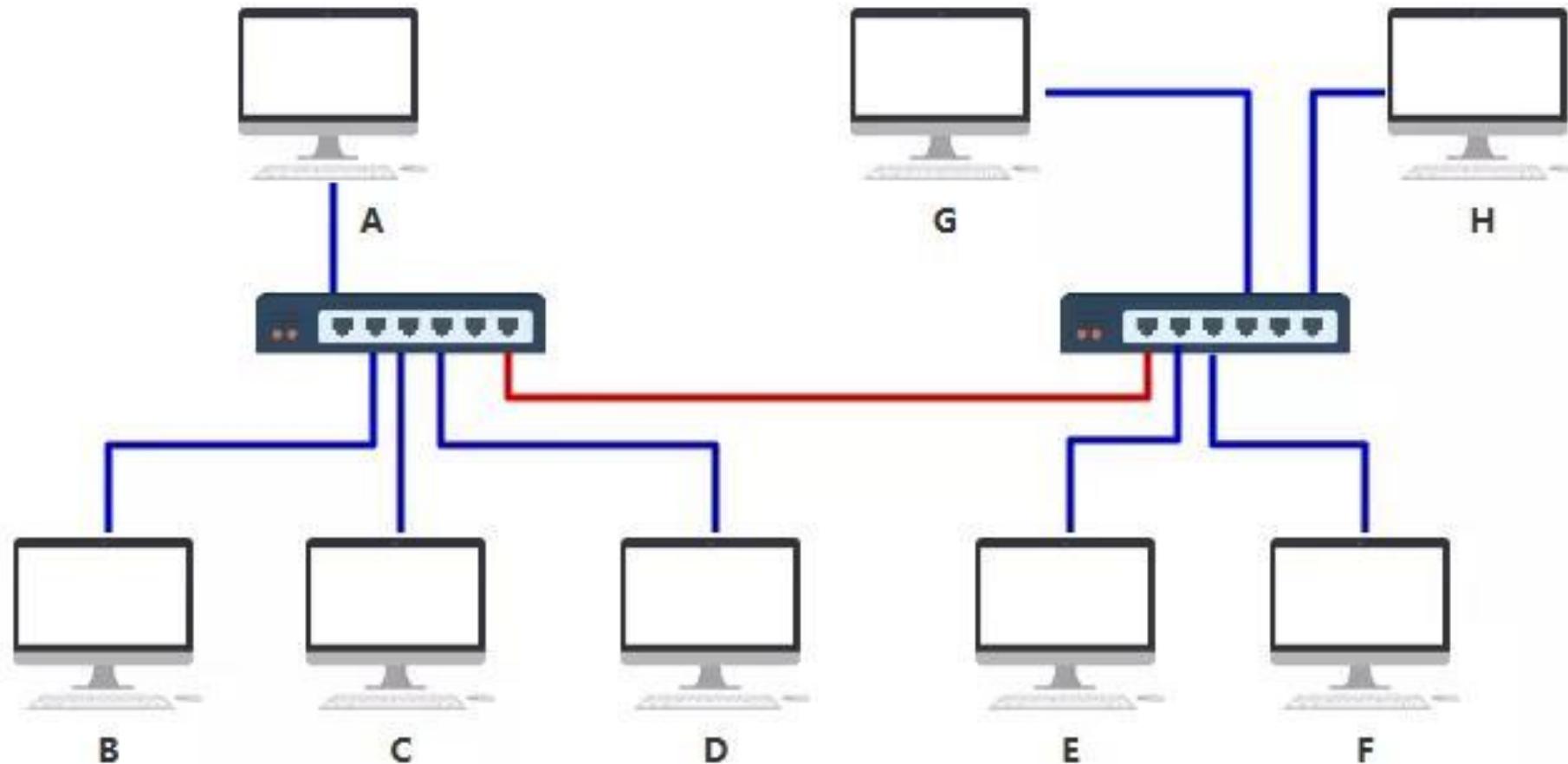
# Internet protocol stack – an example



This is called **Ethernet** (以太网)

What if there are more machines  
than the number of ports?

# Internet protocol stack – an example



# Internet protocol stack – an example

MAC Address	Port
bb-bb-bb-bb-bb-bb	1
cc-cc-cc-cc-cc-cc	3
aa-aa-aa-aa-aa-aa	4
dd-dd-dd-dd-dd-dd	5
ee-ee-ee-ee-ee-ee	6
ff-ff-ff-ff-ff-ff	6
gg-gg-gg-gg-gg-gg	6
hh-hh-hh-hh-hh-hh	6

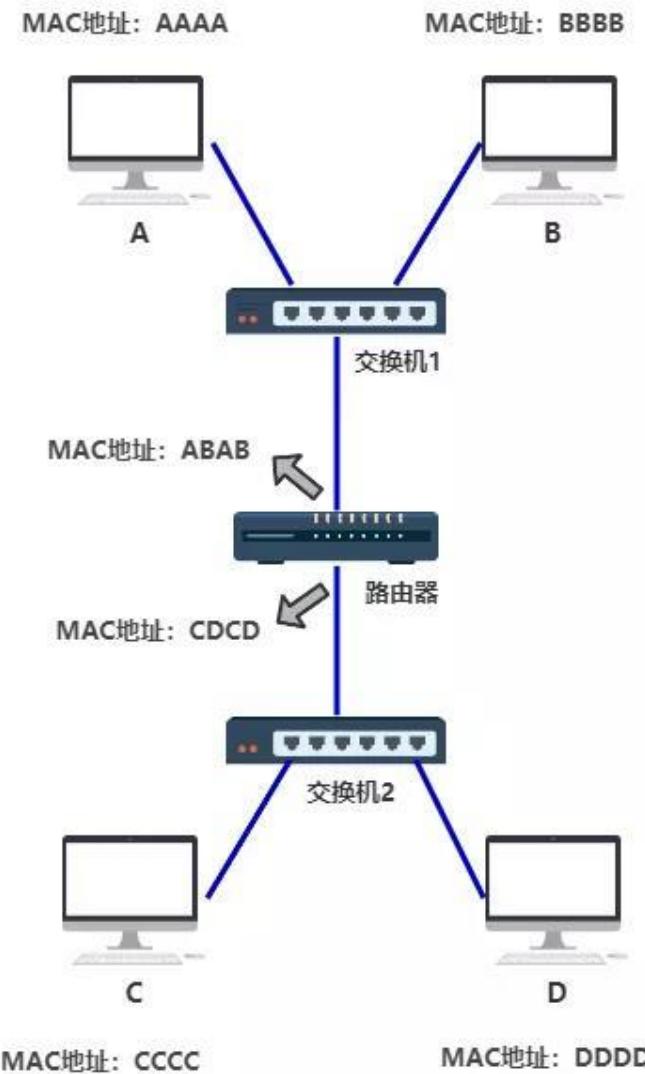
## Summary of link layer

- Data communication with neighbor

# Internet protocol stack – an example

What if there are more local networks to interconnect?

Question: how does a host know when it should send a packet to the router?



# Internet protocol stack – an example

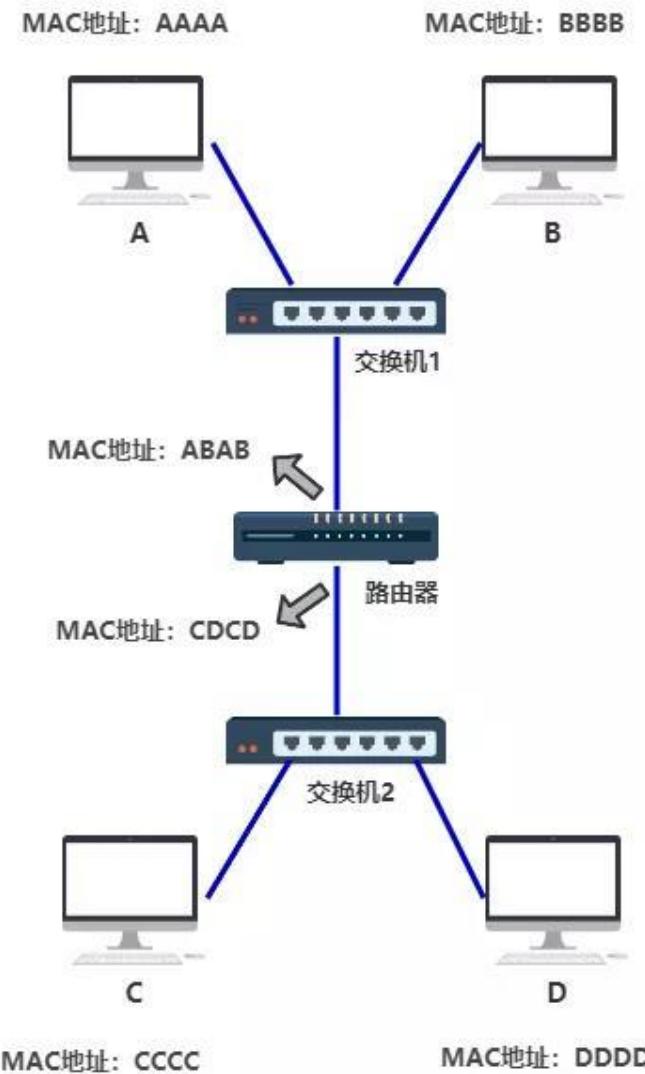
How about routing packets with  
the same MAC prefix the router?

For example:

C's MAC address: 0.0.0.0-FFFF-CCCC

D's MAC address: 0.0.0.0-FFFF-DDDD

It is hard and not quite practical



# Internet protocol stack – an example

A new type of address

- Internet Protocol address (IP address)
- An example IP address: 11000000101010000000000000000001
- 192.168.0.1
- 0.0.0.0 – 255.255.255.255



A



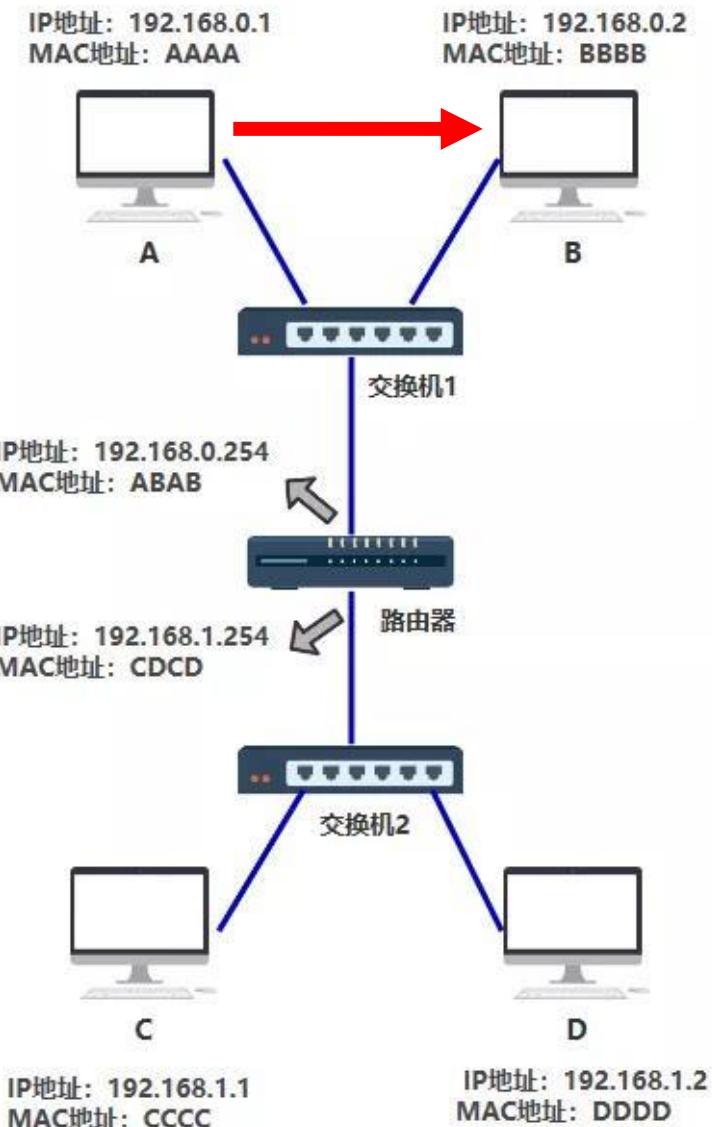
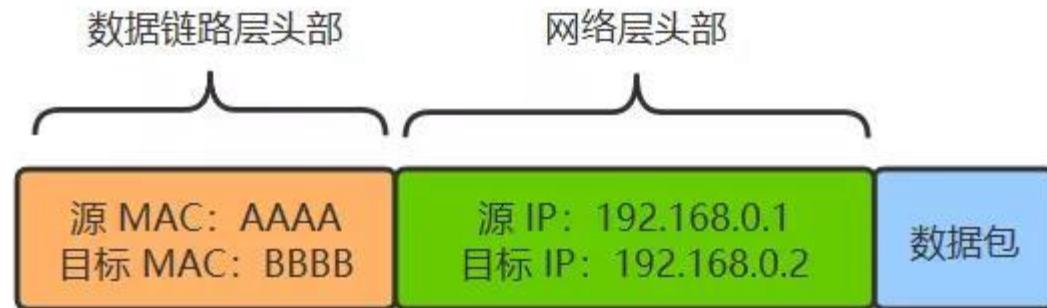
B

MAC: aa-aa-aa-aa-aa-aa  
IP: 192.168.0.1

MAC: bb-bb-bb-bb-bb-bb  
IP: 192.168.0.2

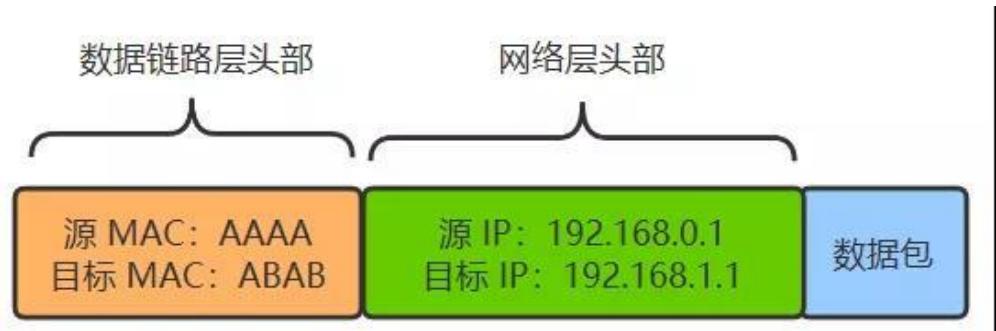
# Internet protocol stack – an example

A to B:

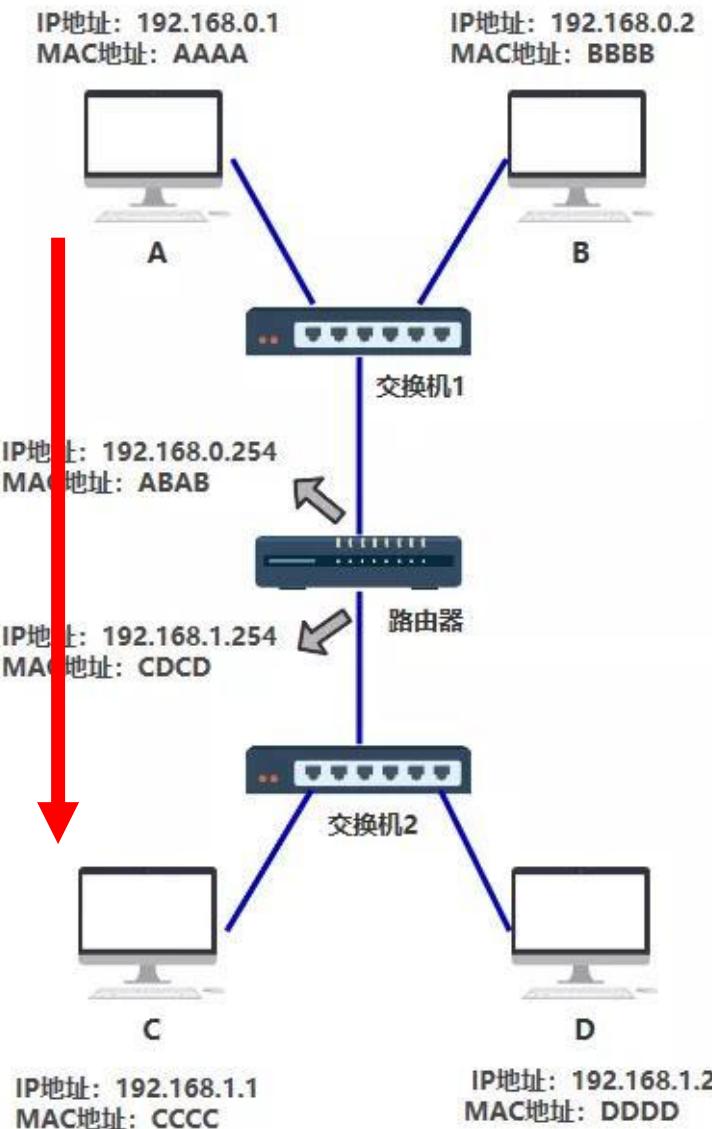
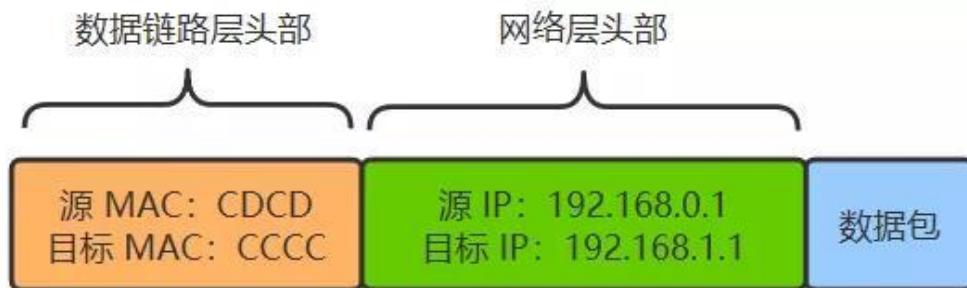


# Internet protocol stack – an example

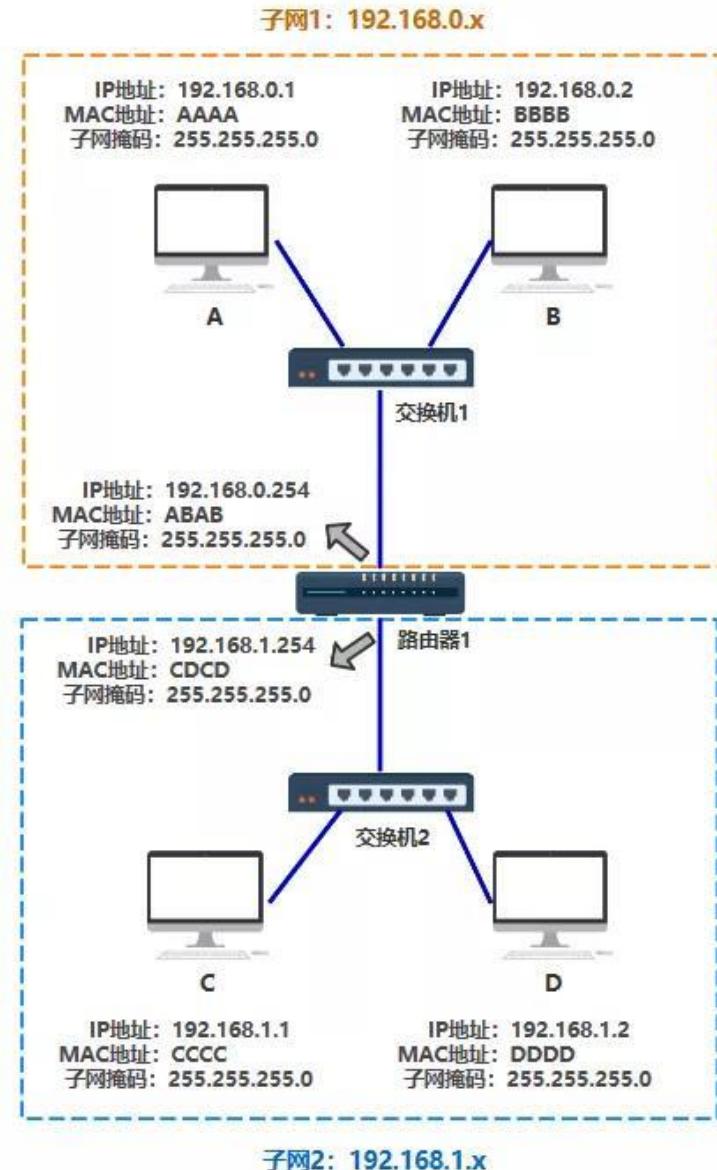
A to 路由器:



路由器 to C:



# Internet protocol stack – an example



## Subnet

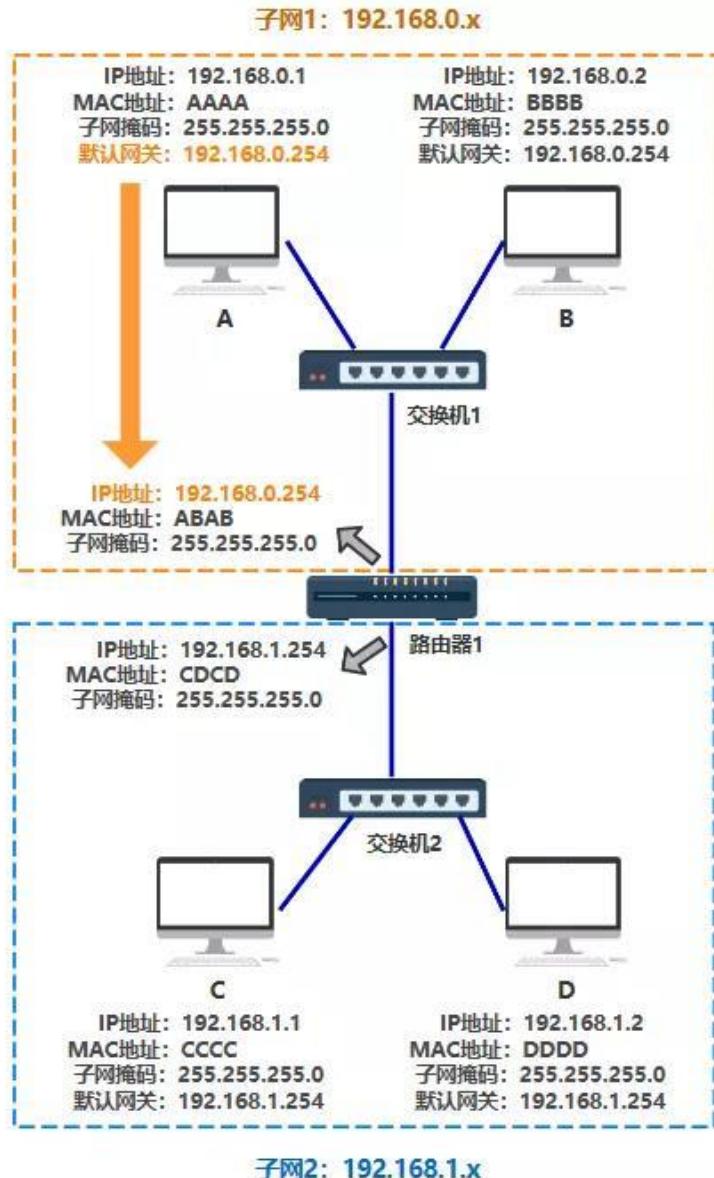
- 192.168.0.1 and 192.168.0.2: same subnet
- 192.168.0.1 and 192.168.1.1: different subnet

## Subnet mask to find the same subnet

- A:  $192.168.0.1 \& 255.255.255.0 = 192.168.0.0$
- B:  $192.168.0.2 \& 255.255.255.0 = 192.168.0.0$
- C:  $192.168.1.1 \& 255.255.255.0 = 192.168.1.0$
- D:  $192.168.1.2 \& 255.255.255.0 = 192.168.1.0$

PS: 255 in binary: 11111111

# Internet protocol stack – an example



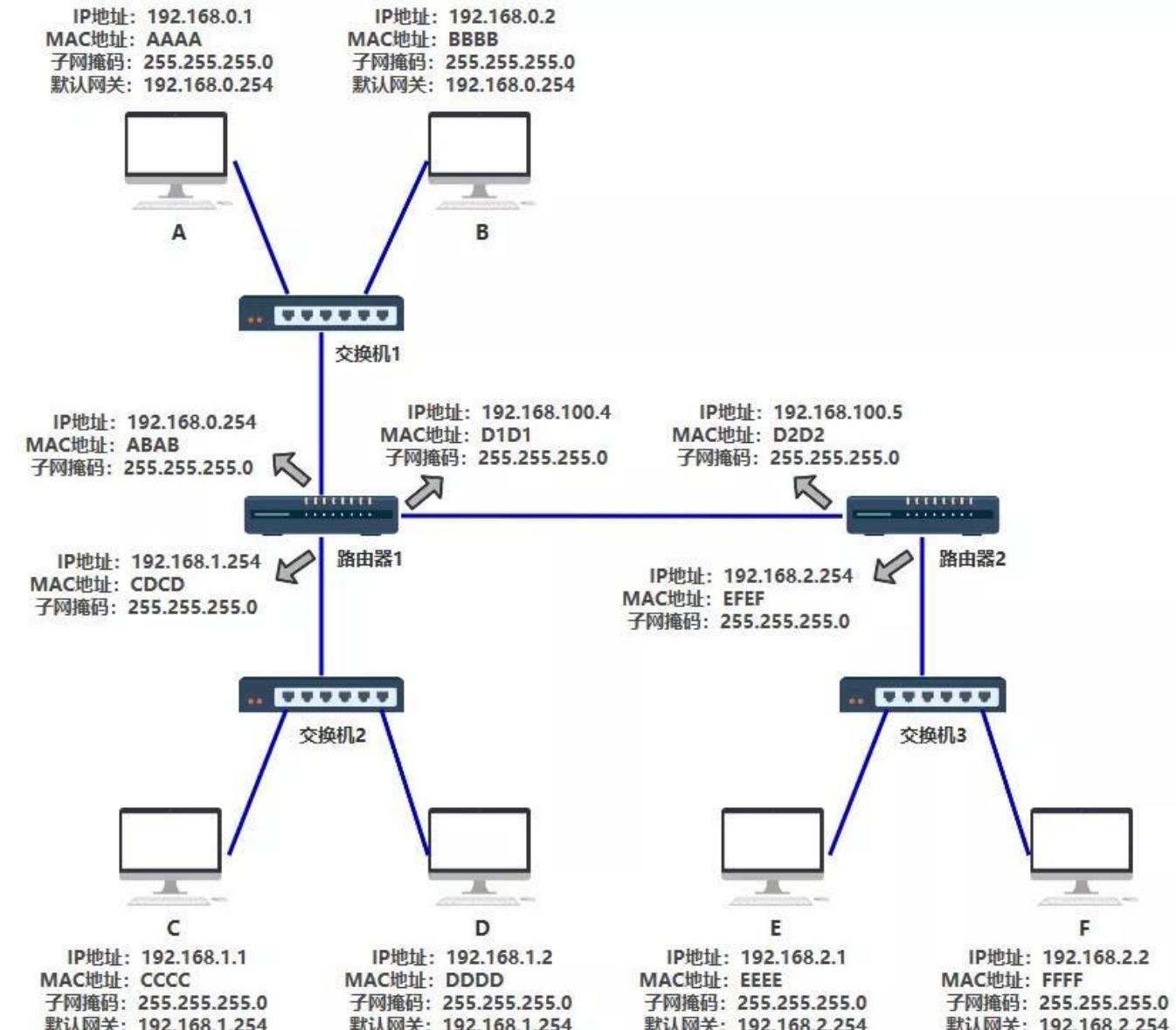
## Gateway

- 192.168.0.254 is the router

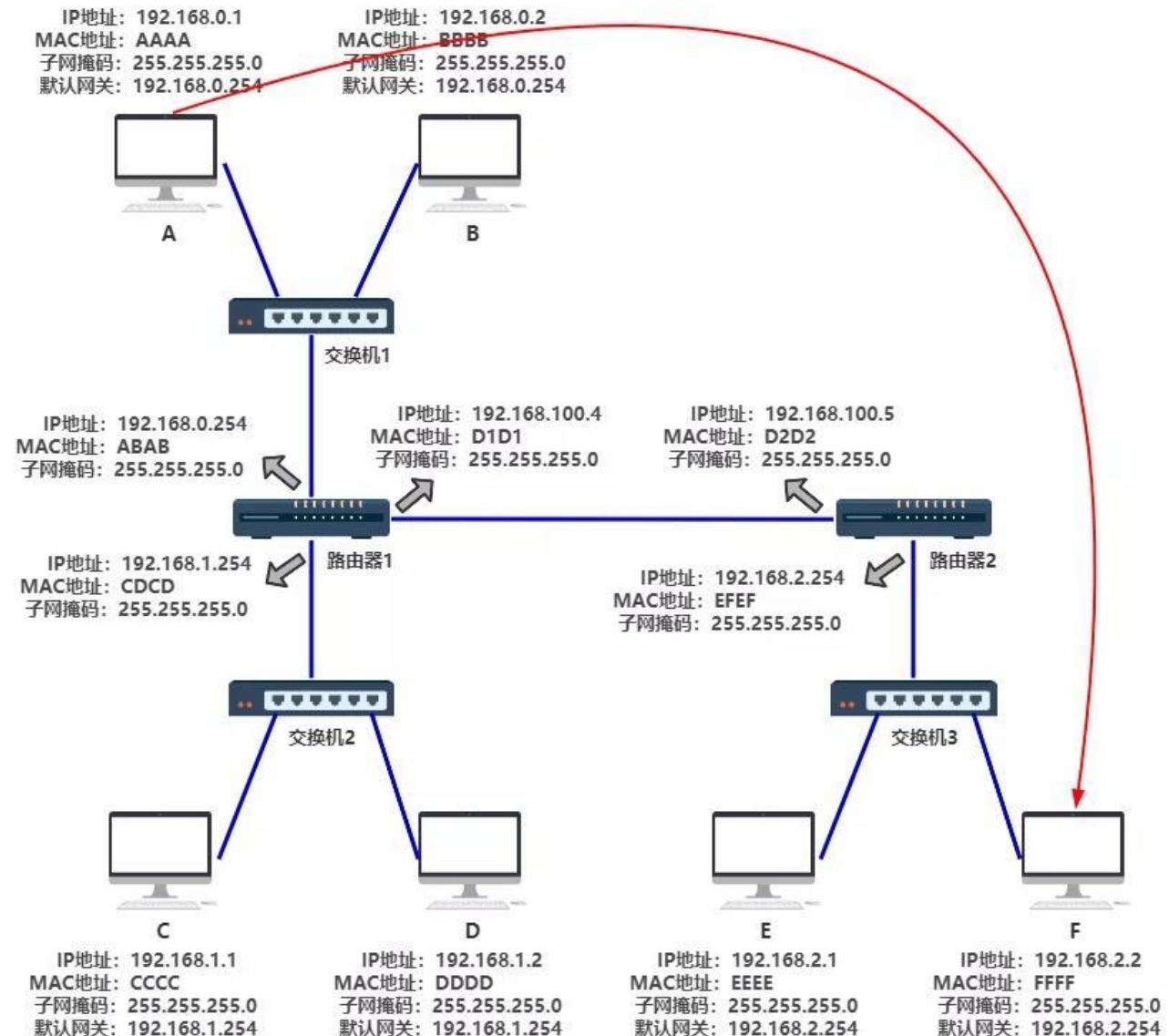
## Summary of network layer

- Route datagram from source to destination

# Internet protocol stack – an example



# Internet protocol stack – an example

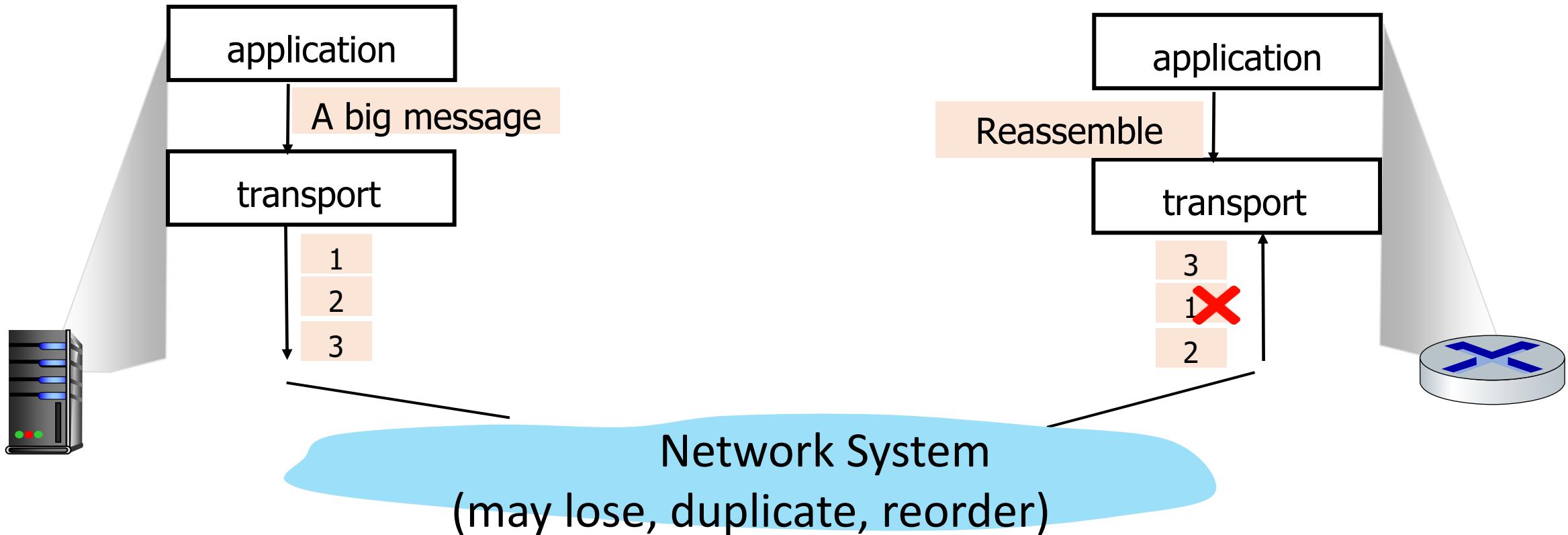


# Transport and application layers

- Build on top of the bottom three layers
  - Use the functionalities of the bottom layers to provide services
- Transport layer
  - sender: **breaks** application messages into *segments*, passes to network layer
  - receiver: **reassembles** segments into messages, passes to application layer
  - E.g., TCP and UDP protocols
- Application layer
  - supporting network applications

# Transport and application layers

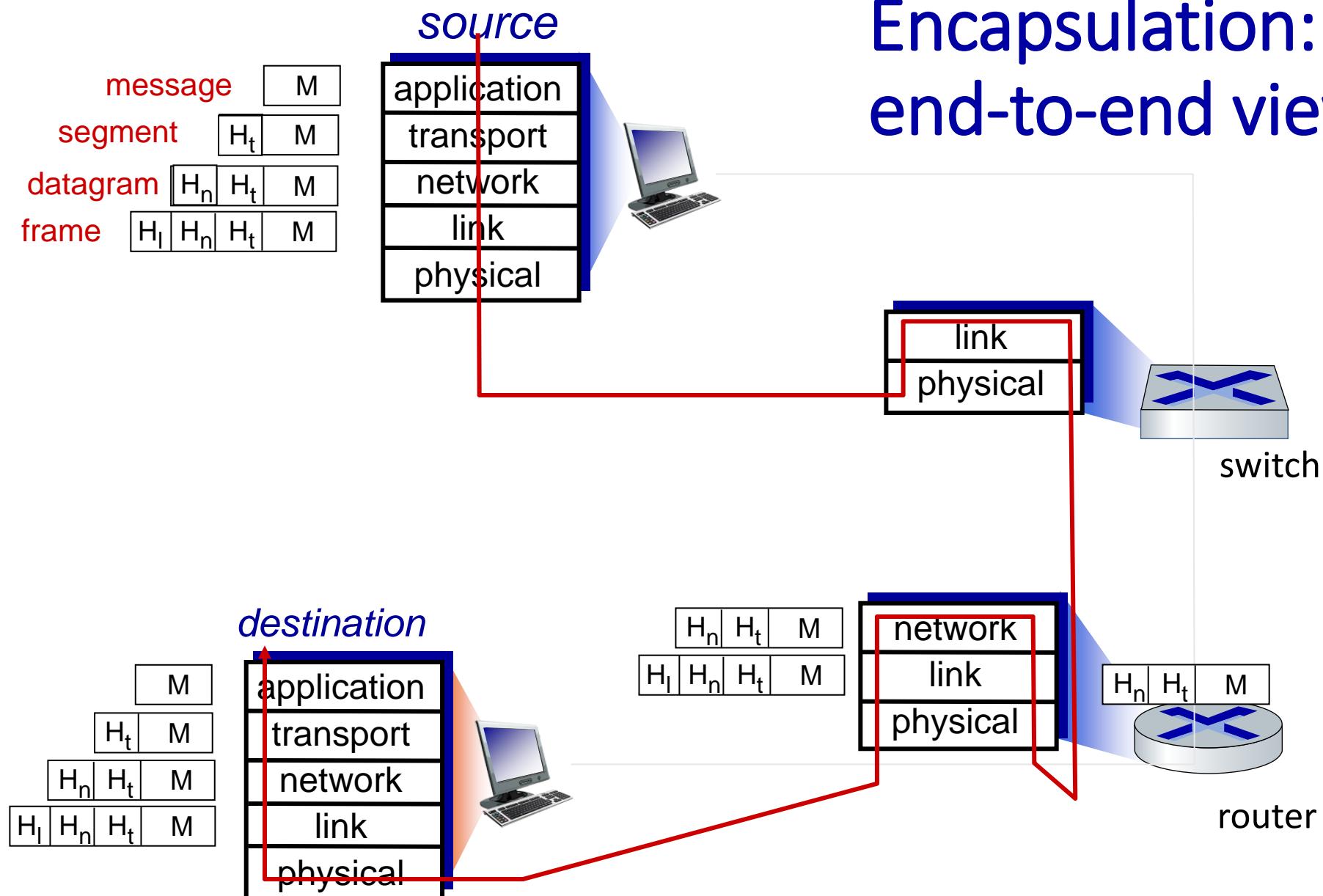
- Transport layer: provide *logical communication* between application processes running on different hosts (e.g., TCP and UDP)



# Notes on the example

- A high-level brief overview of the main functionalities of each layer
  - The example still misses many other functionalities
  - More details on other functionalities later in the course
  - E.g., how to create the MAC table and routing table
- Please keep this example in mind when you study the course later in this semester
  - Have a big picture on why we need to design the network like this

# Encapsulation: an end-to-end view



# Chapter 1: roadmap

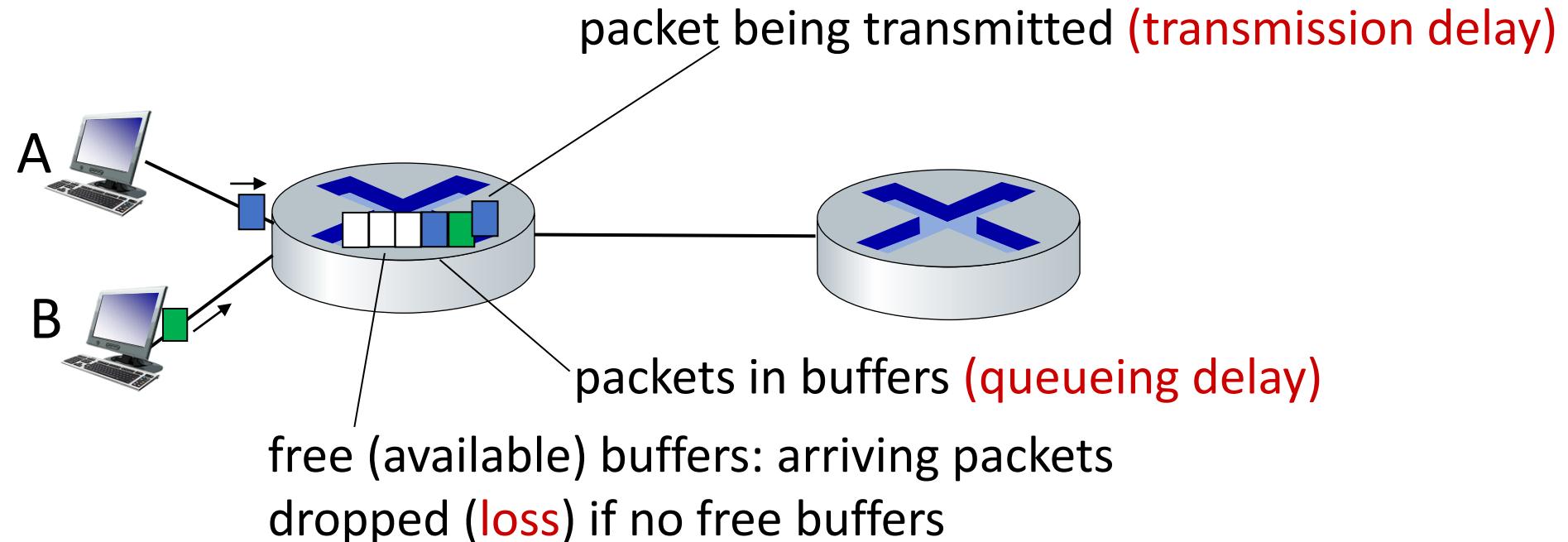
- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Protocol layers, service models
- **Performance: loss, delay, throughput**
- Security
- History



# How do packet loss and delay occur?

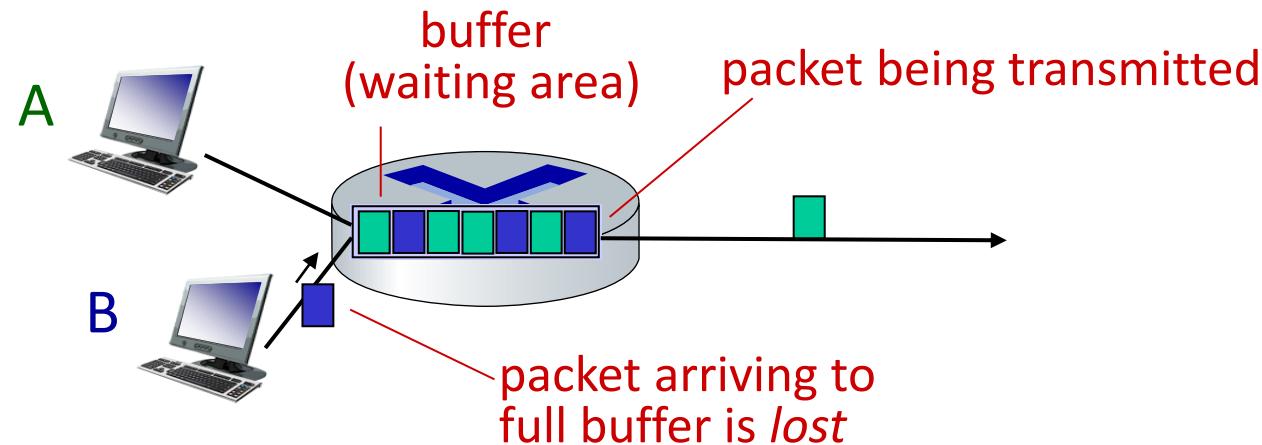
packets *queue* in router buffers

- packets queue, wait for turn
- arrival rate to link (temporarily) exceeds output link capacity: packet loss



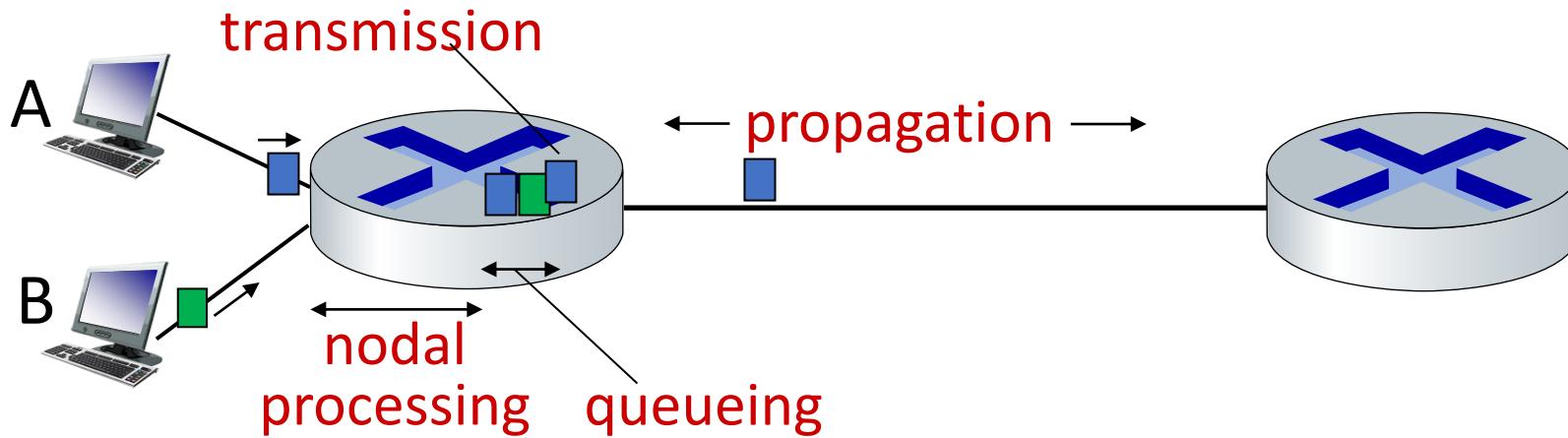
# Packet loss

- queue (aka buffer) preceding link in buffer has finite capacity
- packet arriving to full queue dropped (aka lost)
- lost packet may be retransmitted by previous node, by source end system, or not at all



\* Check out the Java applet for an interactive animation on queuing and loss

# Packet delay at a router: four sources



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

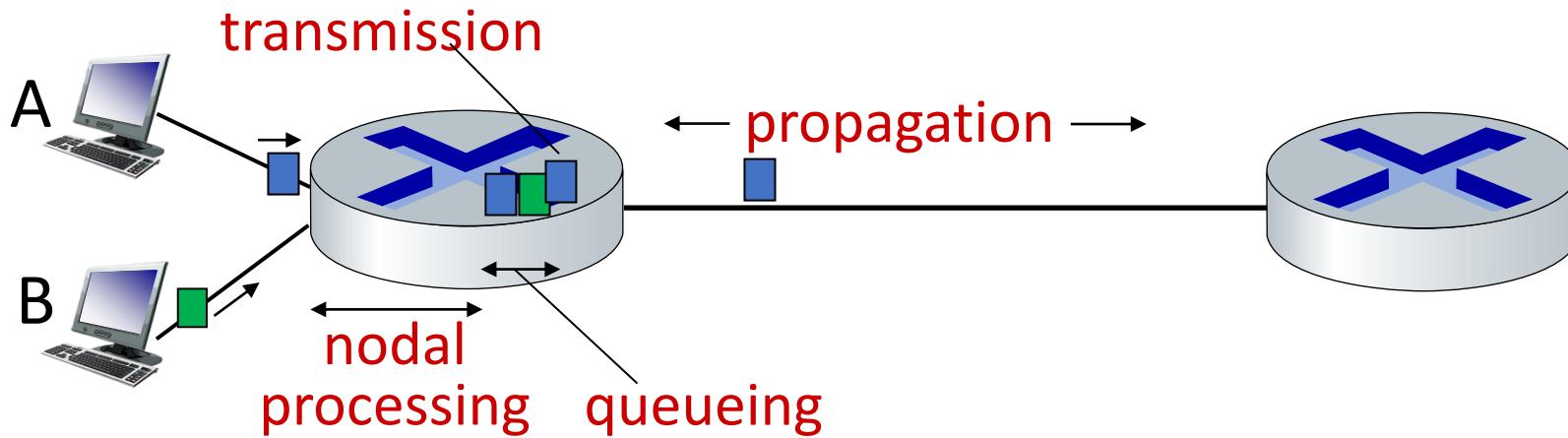
$d_{\text{proc}}$ : nodal processing

- check bit errors
- determine output link
- typically < msec

$d_{\text{queue}}$ : queueing delay

- time waiting at output link for transmission
- depends on congestion level of router

# Packet delay: four sources



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

$d_{\text{trans}}$ : transmission delay:

- $L$ : packet length (bits)
- $R$ : link *transmission rate (bps)*

$$\boxed{\mathbf{d}_{\text{trans}} = L/R}$$

$d_{\text{trans}}$  and  $d_{\text{prop}}$   
very different

$d_{\text{prop}}$ : propagation delay:

- $d$ : length of physical link
- $s$ : propagation speed ( $\sim 2 \times 10^8$  m/sec)

$$\boxed{\mathbf{d}_{\text{prop}} = d/s}$$

\* Check out the online interactive exercises:  
[http://gaia.cs.umass.edu/kurose\\_ross](http://gaia.cs.umass.edu/kurose_ross)

# Transmission delay vs. propagation delay

$d_{\text{trans}}$ : transmission delay (传输时延):

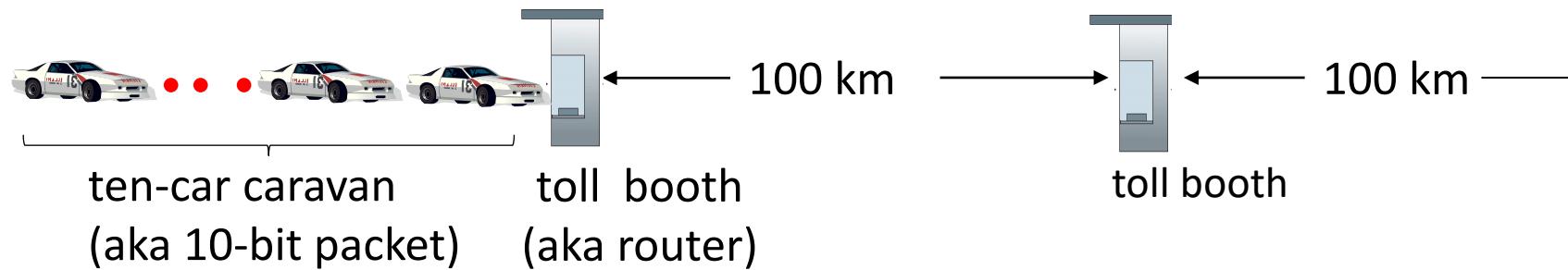
- $L$ : packet length (bits)
- $R$ : link *transmission rate (bps)*
- $d_{\text{trans}} = L/R$



$d_{\text{prop}}$ : propagation delay (传播时延):

- $d$ : length of physical link
- $s$ : propagation speed ( $\sim 2 \times 10^8$  m/sec)
- $d_{\text{prop}} = d/s$

# Caravan analogy

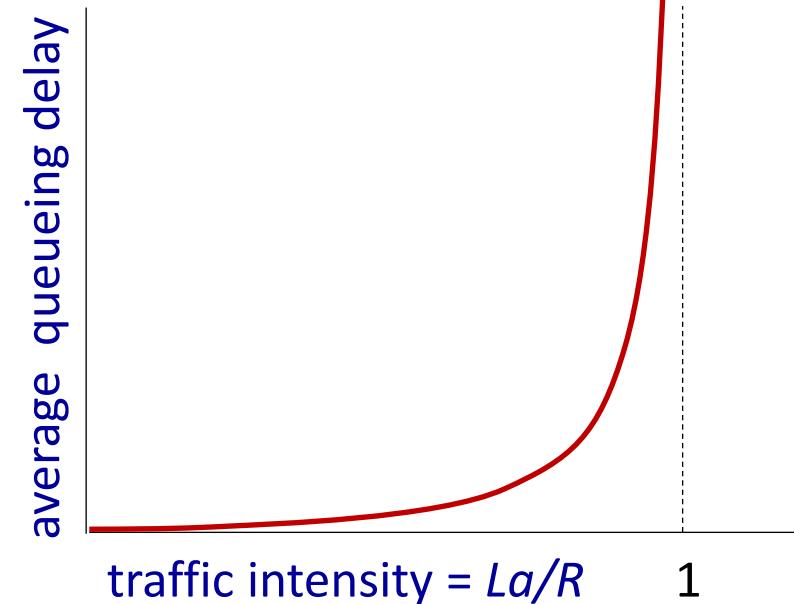


- cars “propagate” at 100 km/hr
- toll booth takes 12 sec to service car (bit transmission time)
- car ~ bit; caravan ~ packet
- **Q: How long until caravan is lined up before 2nd toll booth?**

- time to “push” entire caravan through toll booth onto highway =  $12 * 10 = 120$  sec
- time for last car to propagate from 1st to 2nd toll both:  $100\text{km}/(100\text{km/hr}) = 1$  hr
- **A: 62 minutes**

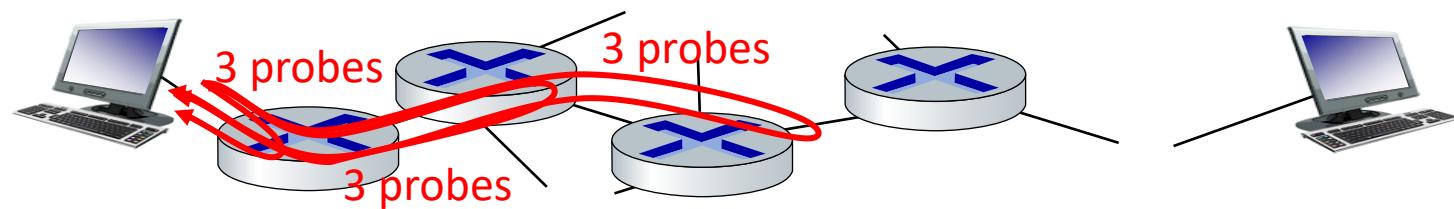
# Packet queueing delay (revisited)

- $R$ : link bandwidth (bps)
  - $L$ : packet length (bits)
  - $a$ : average packet arrival rate
  - Total arrival of bits:  $L \cdot a$
  - Traffic intensity:  $La/R$
- 
- $La/R \sim 0$ : avg. queueing delay small
  - $La/R \rightarrow 1$ : avg. queueing delay large
  - $La/R > 1$ : more “work” arriving is more than can be serviced - average delay infinite!



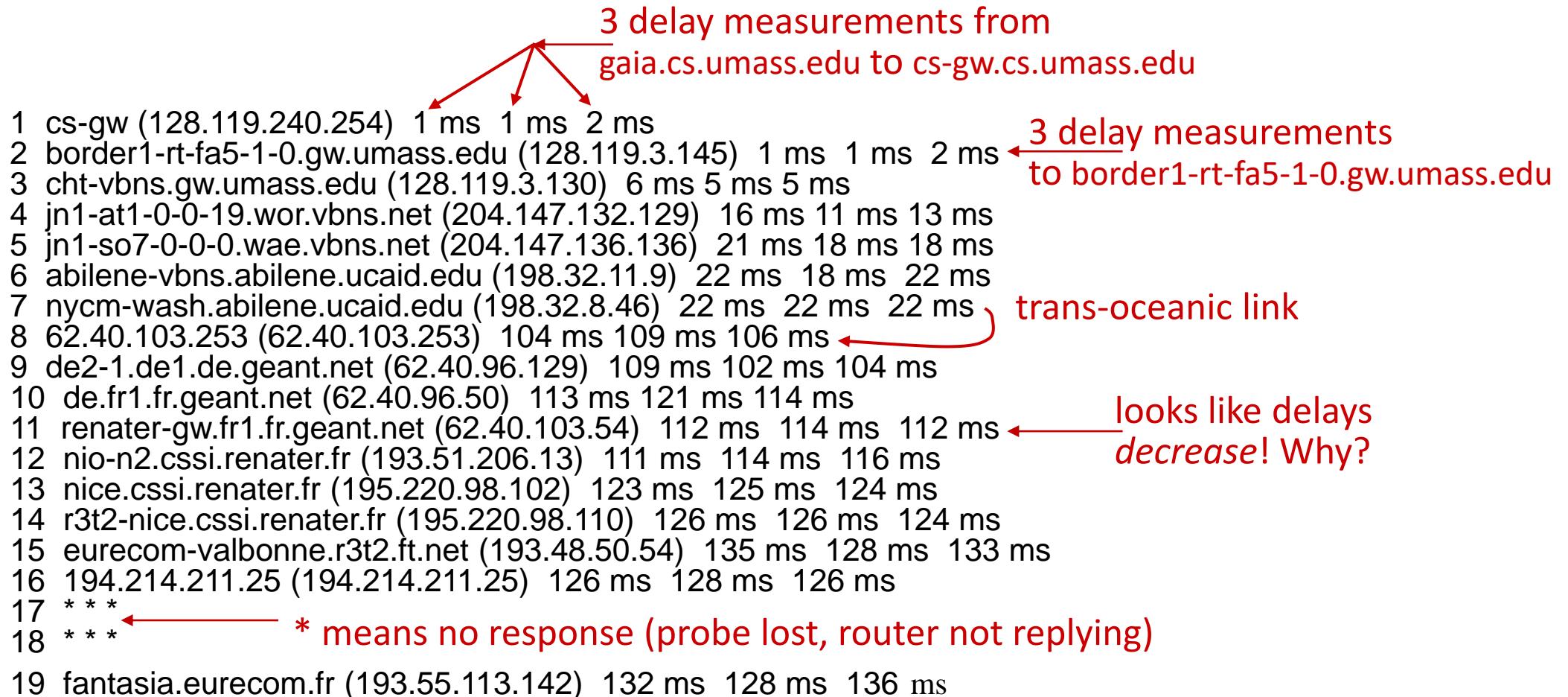
# “Real” Internet delays and routes

- what do “real” Internet delay & loss look like?
- **traceroute** program: provides delay measurement from source to router along end-end Internet path towards destination. For all  $i$  (*router*):
  - sends three packets that will reach router  $i$  on path towards destination (with time-to-live field value of  $i$ )
  - router  $i$  will return packets to sender
  - sender measures time interval between transmission and reply



# Real Internet delays and routes

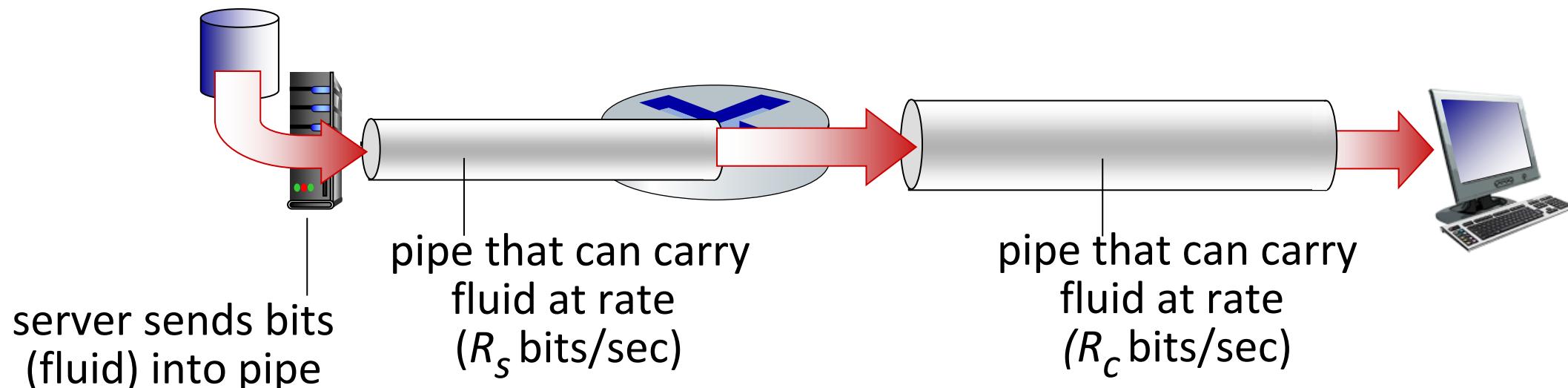
traceroute: gaia.cs.umass.edu to www.eurecom.fr



\* Do some traceroutes from exotic countries at [www.traceroute.org](http://www.traceroute.org)

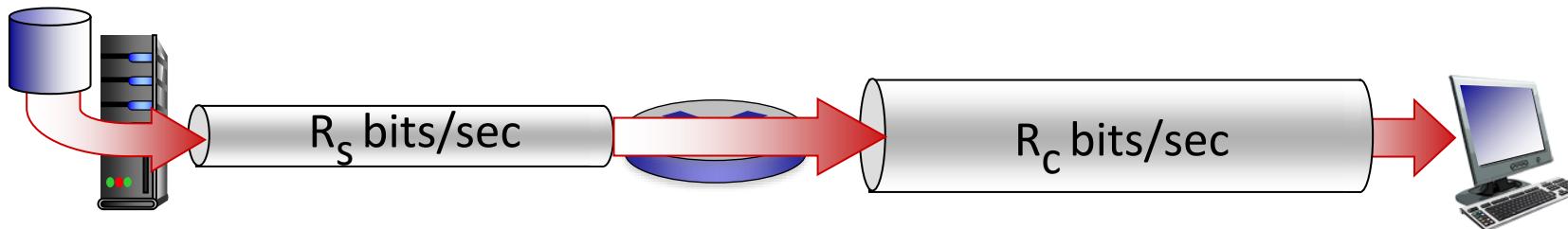
# Throughput

- *throughput*: rate (bits/time unit) at which bits are being sent from sender to receiver
  - *instantaneous*: rate at given point in time
  - *average*: rate over a long period of time

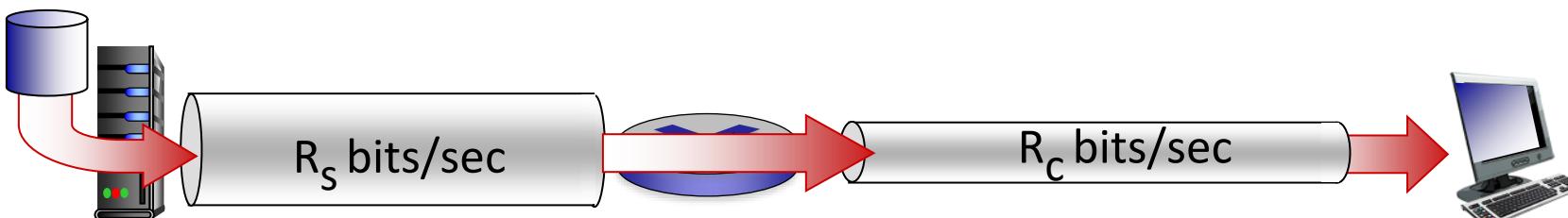


# Throughput

$R_s < R_c$  What is average end-end throughput?



$R_s > R_c$  What is average end-end throughput?

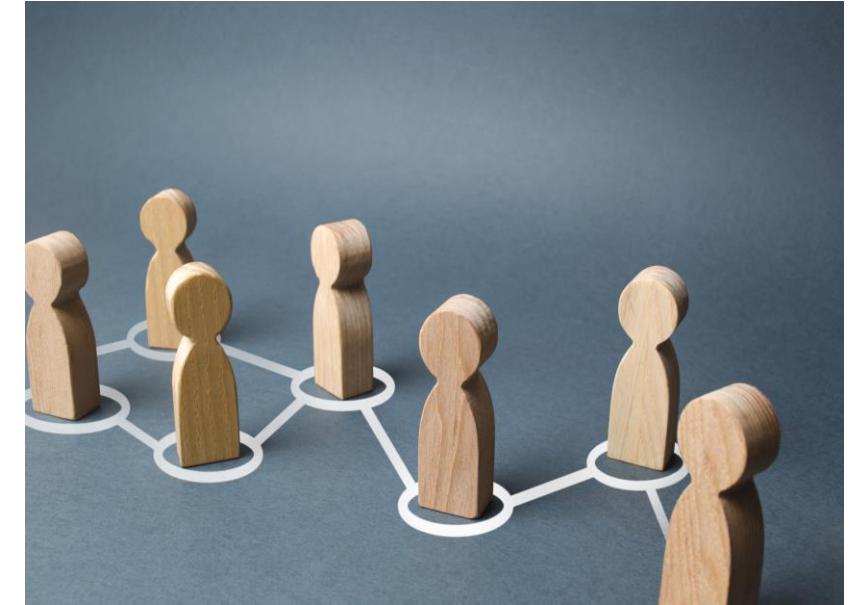


*bottleneck link*

link on end-end path that constrains end-end throughput

# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Protocol layers, service models
- Performance: loss, delay, throughput
- **Security**
- History



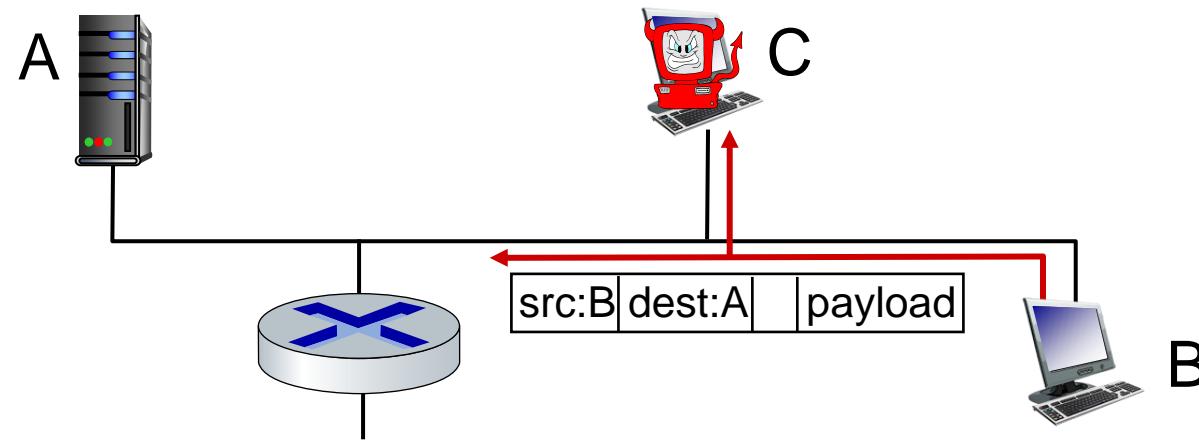
# Network security

- **field of network security:**
  - how bad guys can attack computer networks
  - how we can defend networks against attacks
  - how to design architectures that are immune to attacks
- **Internet not originally designed with (much) security in mind**
  - *original vision*: “a group of mutually trusting users attached to a transparent network” ☺
  - Internet protocol designers playing “catch-up”
  - security considerations in all layers!

# Bad guys: packet interception

*packet “sniffing”:*

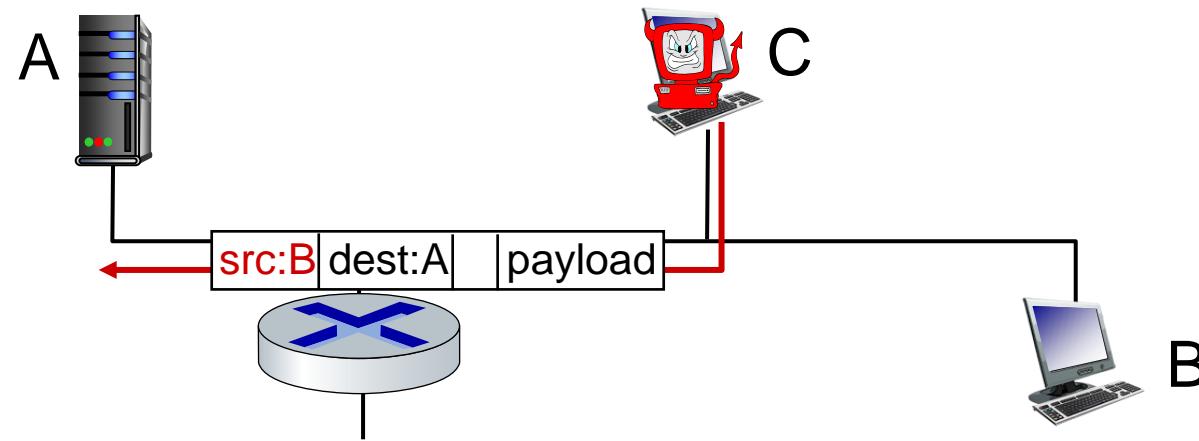
- broadcast media (shared Ethernet, wireless)
- promiscuous network interface reads/records all packets (e.g., including passwords!) passing by



Wireshark software used for our end-of-chapter labs is a (free) packet-sniffer

# Bad guys: fake identity

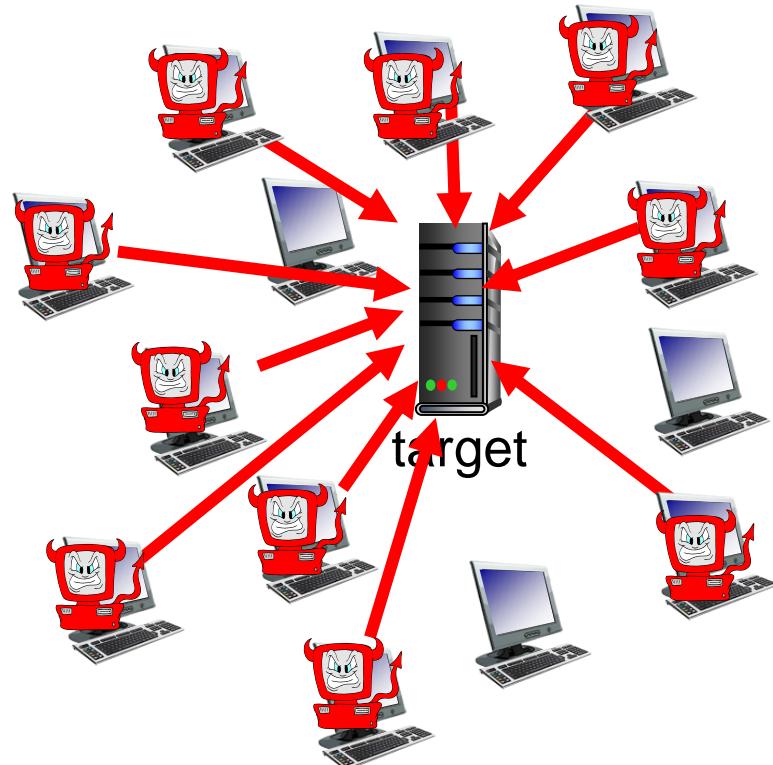
*IP spoofing:* send packet with false source address



# Bad guys: denial of service

*Denial of Service (DoS):* attackers make resources (server, bandwidth) unavailable to legitimate traffic by overwhelming resource with bogus traffic

1. select target
2. break into hosts  
around the network  
(see botnet)
3. send packets to target  
from compromised  
hosts



# Lines of defense:

- **authentication**: proving you are who you say you are
  - cellular networks provides hardware identity via SIM card; no such hardware assist in traditional Internet
- **confidentiality**: via encryption
- **integrity checks**: digital signatures prevent/detect tampering
- **access restrictions**: password-protected VPNs
- **firewalls**: specialized “middleboxes” in access and core networks:
  - off-by-default: filter incoming packets to restrict senders, receivers, applications
  - detecting/reacting to DOS attacks

*... lots more on security (throughout, Chapter 8)*

# Chapter 1: roadmap

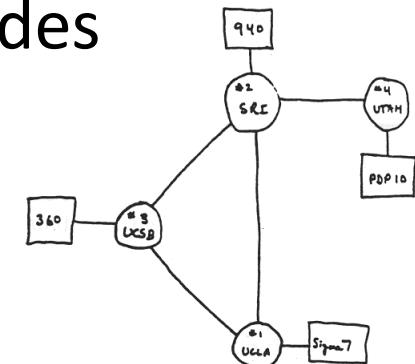
- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Protocol layers, service models
- Performance: loss, delay, throughput
- Security
- History



# Internet history

## 1961-1972: Early packet-switching principles

- 1961: Kleinrock - queueing theory shows effectiveness of packet-switching
- 1964: Baran - packet-switching in military nets
- 1967: ARPAnet conceived by Advanced Research Projects Agency
- 1969: first ARPAnet node operational
- 1972:
  - ARPAnet public demo
  - NCP (Network Control Protocol) first host-host protocol
  - first e-mail program
  - ARPAnet has 15 nodes



THE ARPA NETWORK

# Internet history

## 1972-1980: Internetworking, new and proprietary nets

- 1970: ALOHAnet satellite network in Hawaii
- 1974: Cerf and Kahn - architecture for interconnecting networks
- 1976: Ethernet at Xerox PARC
- late 70's: proprietary architectures: DECnet, SNA, XNA
- late 70's: switching fixed length packets (ATM precursor)
- 1979: ARPAnet has 200 nodes

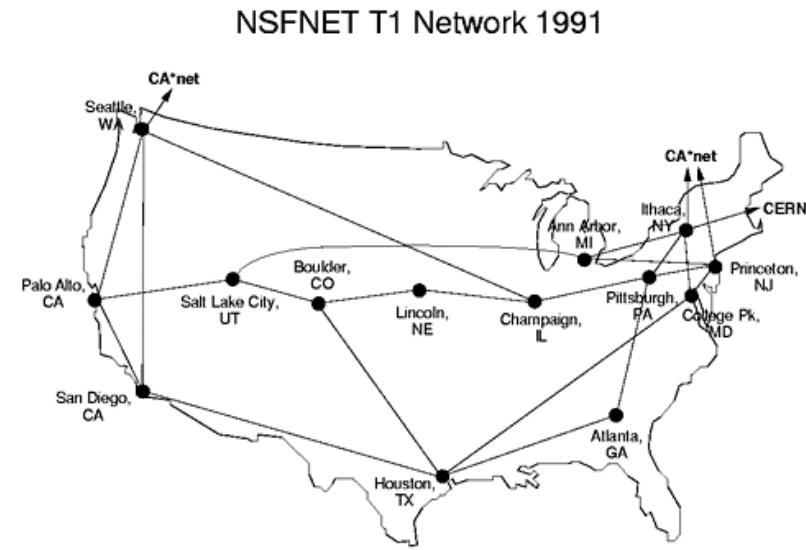
Cerf and Kahn's internetworking principles:

- minimalism, autonomy - no internal changes required to interconnect networks
  - best-effort service model
  - stateless routing
  - decentralized control
- define today's Internet architecture

# Internet history

## *1980-1990: new protocols, a proliferation of networks*

- 1983: deployment of TCP/IP
- 1982: smtp e-mail protocol defined
- 1983: DNS defined for name-to-IP-address translation
- 1985: ftp protocol defined
- 1988: TCP congestion control
- new national networks: CSnet, BITnet, NSFnet, Minitel
- 100,000 hosts connected to confederation of networks



# Internet history

## *1990, 2000s: commercialization, the Web, new applications*

- early 1990s: ARPAnet decommissioned
  - 1991: NSF lifts restrictions on commercial use of NSFnet (decommissioned, 1995)
  - early 1990s: Web
    - hypertext [Bush 1945, Nelson 1960's]
    - HTML, HTTP: Berners-Lee
    - 1994: Mosaic, later Netscape
    - late 1990s: commercialization of the Web
- late 1990s – 2000s:
- more killer apps: instant messaging, P2P file sharing
  - network security to forefront
  - est. 50 million host, 100 million+ users
  - backbone links running at Gbps

# Internet history

*2005-present: more new applications, Internet is “everywhere”*

- ~18B devices attached to Internet (2017)
  - rise of smartphones (iPhone: 2007)
- aggressive deployment of broadband access
- increasing ubiquity of high-speed wireless access: 4G/5G, WiFi
- emergence of online social networks:
  - Facebook: ~ 2.5 billion users
- service providers (Google, FB, Microsoft) create their own networks
  - bypass commercial Internet to connect “close” to end user, providing “instantaneous” access to search, video content, ...
- enterprises run their services in “cloud” (e.g., Amazon Web Services, Microsoft Azure)

# Chapter 1: summary

*We've covered a "ton" of material!*

- Internet overview
- what's a protocol?
- layering, service models
- network edge, access network, core
  - packet-switching versus circuit-switching
  - Internet structure
- performance: loss, delay, throughput
- security
- history

*You now have:*

- context, overview, vocabulary, "feel" of networking
- more depth, detail, *and fun* to follow!

# Chapter 2: Application Layer

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Application layer: overview

## Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

# Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video  
(bilibili, Migu)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- remote login
- ...

*Q: your favorites?*

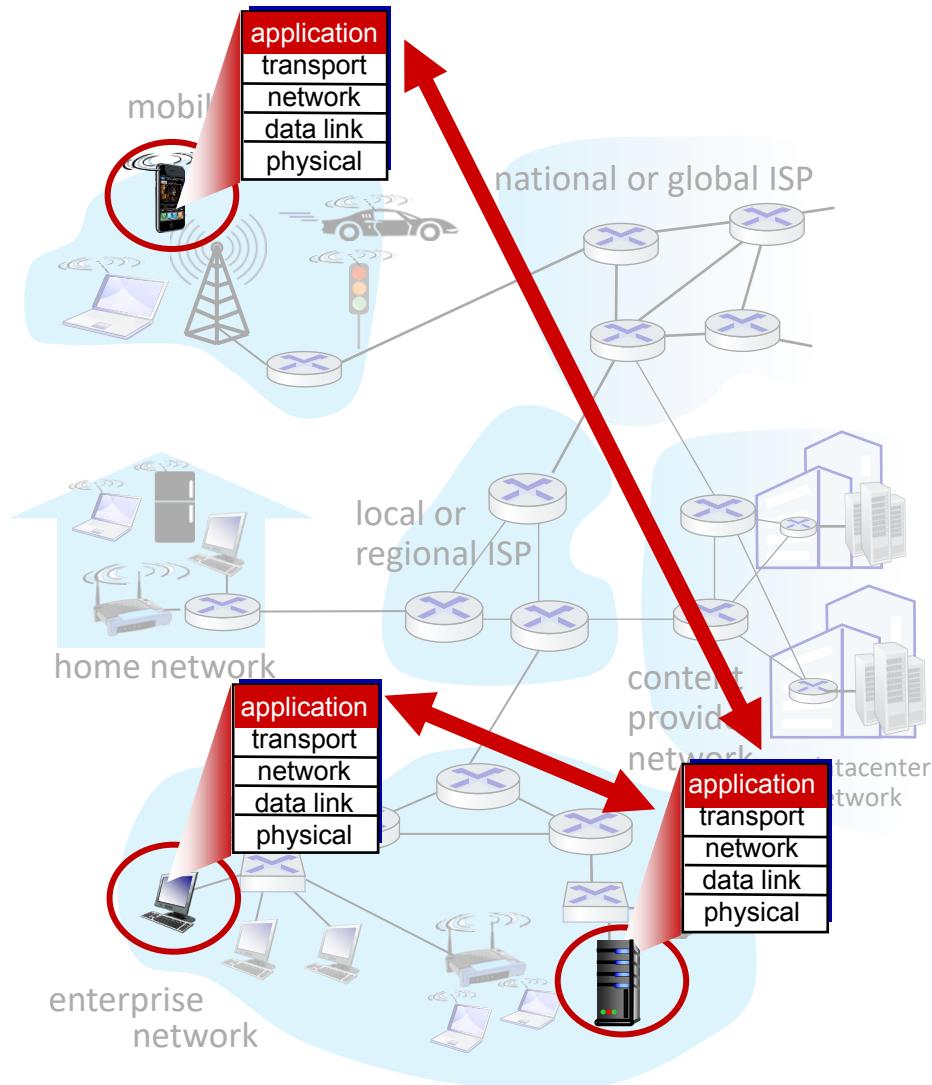
# Creating a network app

write programs that:

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

no need to write software for  
network-core devices

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



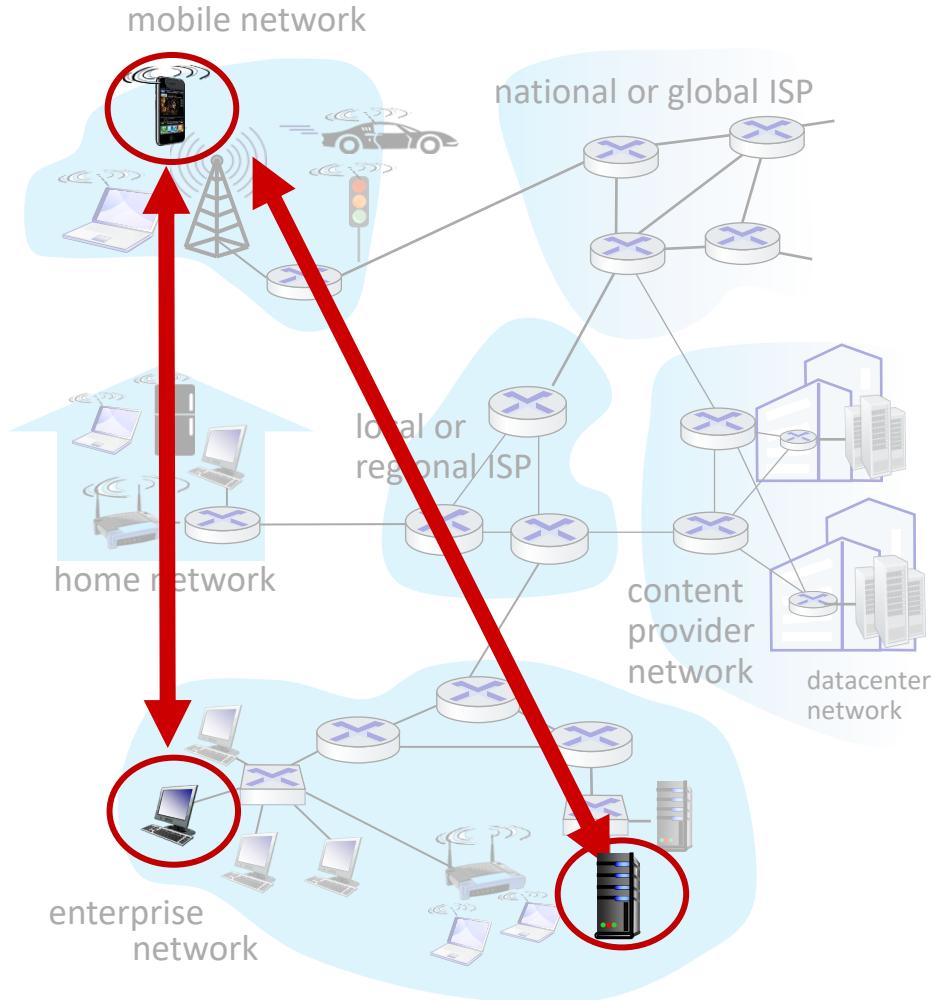
# Client-server paradigm

## server:

- always-on – long-running
- permanent IP address
- often in data centers, for scaling

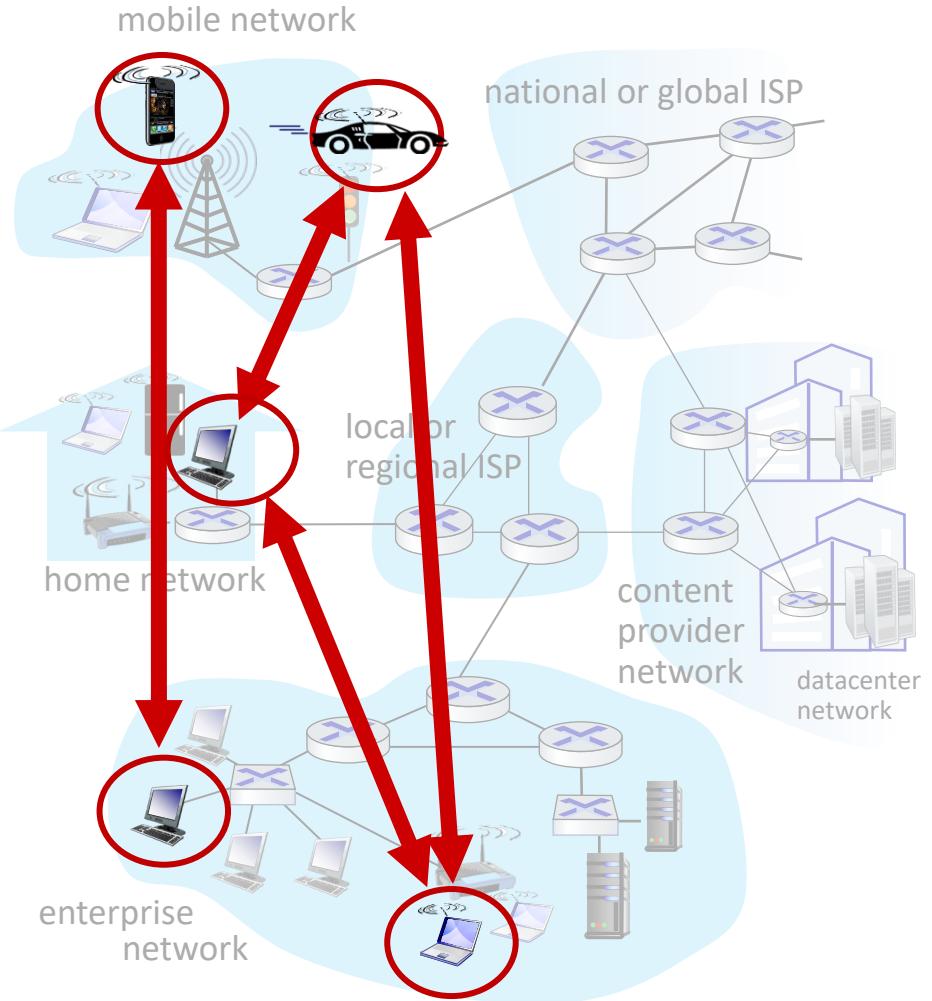
## clients:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP



# Peer-to-peer (P2P) paradigm

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



# Processes communicating

*process*: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

clients, servers

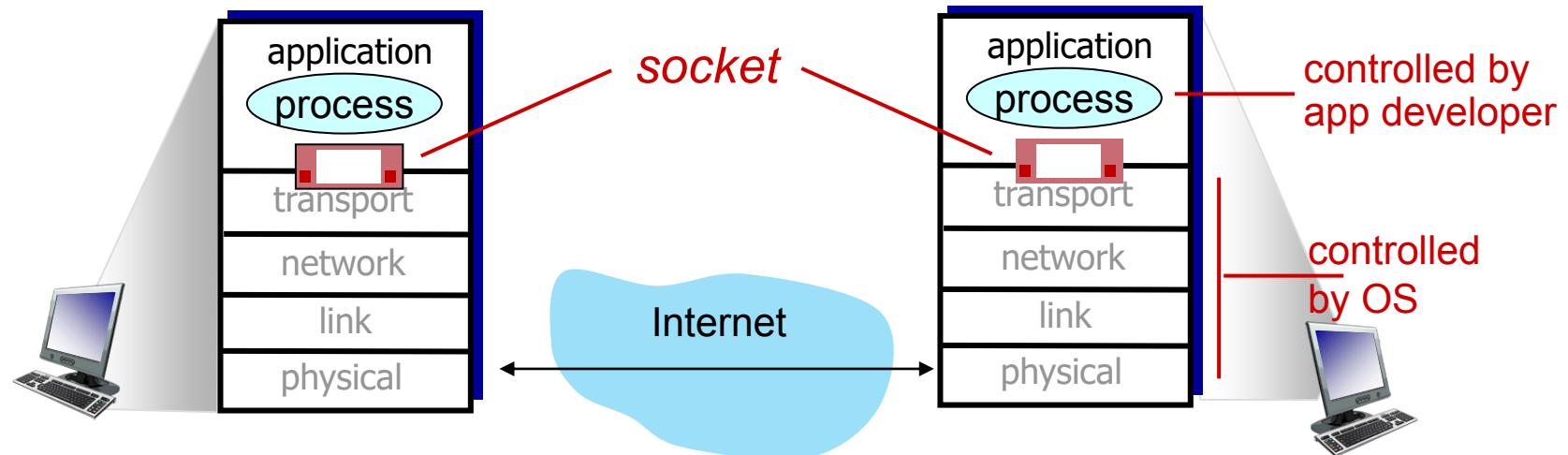
*client process*: process that initiates communication

*server process*: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

# Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to courier (快递员)
  - sending process shoves message out courier
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



# Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - **IP address:** 128.119.245.12
  - **port number:** 80
- more shortly...

# An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability

e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

# What transport service does an app need?

## data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

## throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

## timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

## security

- encryption, data integrity, ...

# Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

# Internet transport protocols services

## TCP service:

- ***reliable transport*** between sending and receiving process
- ***flow control***: sender won't overwhelm receiver
- ***congestion control***: throttle sender when network overloaded
- ***does not provide***: timing, minimum throughput guarantee, security
- ***connection-oriented***: setup required between client and server processes

## UDP service:

- ***unreliable data transfer*** between sending and receiving process
- ***does not provide***: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

# Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

# Securing TCP

## Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

## Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

## TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn

## TLS socket API

- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8

# Application layer: overview

- Principles of network applications
- **Web and HTTP**
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Web and HTTP

*First, a quick review...*

- web page consists of *objects*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL (Uniform Resource Locator)*, e.g.,

www . someschool . edu / someDept / pic . gif

host name

path name

# HTTP overview

## HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model:
  - *client*: browser that requests, receives, (using HTTP protocol) and “displays” Web objects
  - *server*: Web server sends (using HTTP protocol) objects in response to requests



# HTTP overview (continued)

## *HTTP uses TCP:*

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

## *HTTP is “stateless”*

- server maintains *no* information about past client requests

*aside*  
protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

# HTTP connections: two types

## *Non-persistent HTTP (HTTP 1.0)*

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

## *Persistent HTTP (HTTP 1.1)*

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client and that server
- TCP connection closed

# Non-persistent HTTP: example

User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80



1b. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80 “accepts” connection, notifying client

time  
↓

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

# Non-persistent HTTP: example (cont.)

User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects



4. HTTP server closes TCP connection.

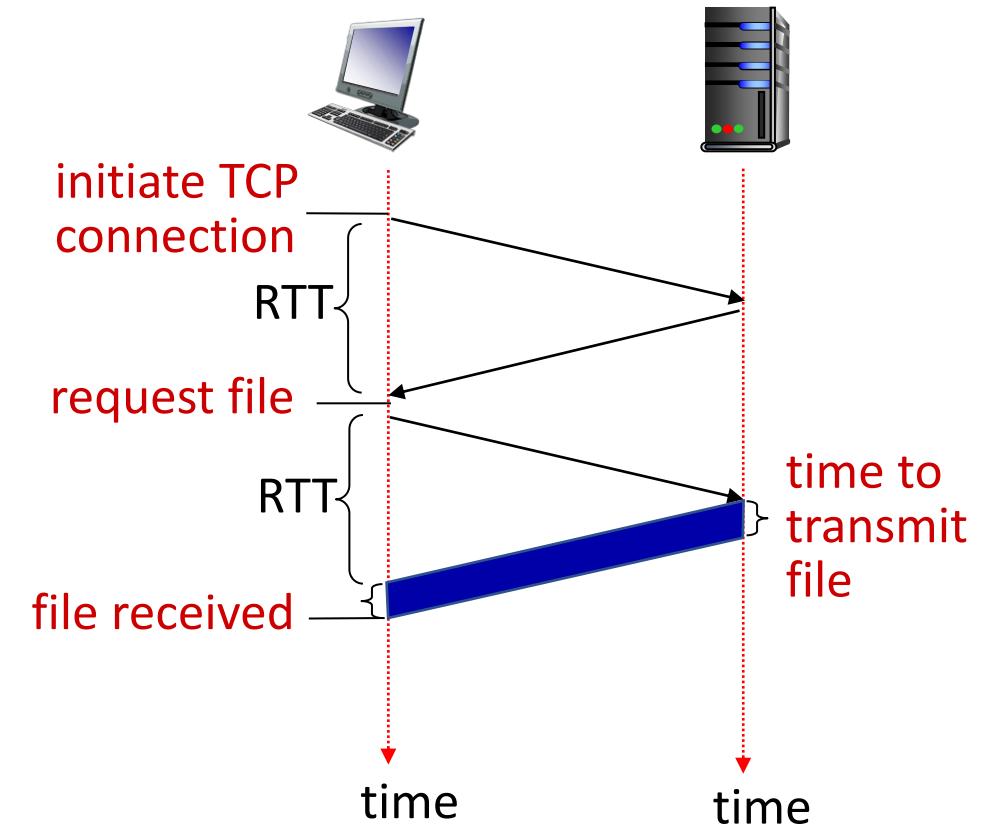
time

# Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP response time (per object):

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



$$\text{Non-persistent HTTP response time} = 2\text{RTT} + \text{file transmission time}$$

# Persistent HTTP (HTTP 1.1)

## *Non-persistent HTTP issues:*

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

## *Persistent HTTP (HTTP1.1):*

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

# HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:

- ASCII (human-readable format)

post data to 服务器

request line (GET, POST,  
HEAD commands) → GET /index.html HTTP/1.1\r\n  
Host: www-net.cs.umass.edu\r\nUser-Agent: Firefox/3.6.10\r\n  
Accept: text/html,application/xhtml+xml\r\nAccept-Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7\r\nKeep-Alive: 115\r\nConnection: (keep-alive)\r\n\r\n

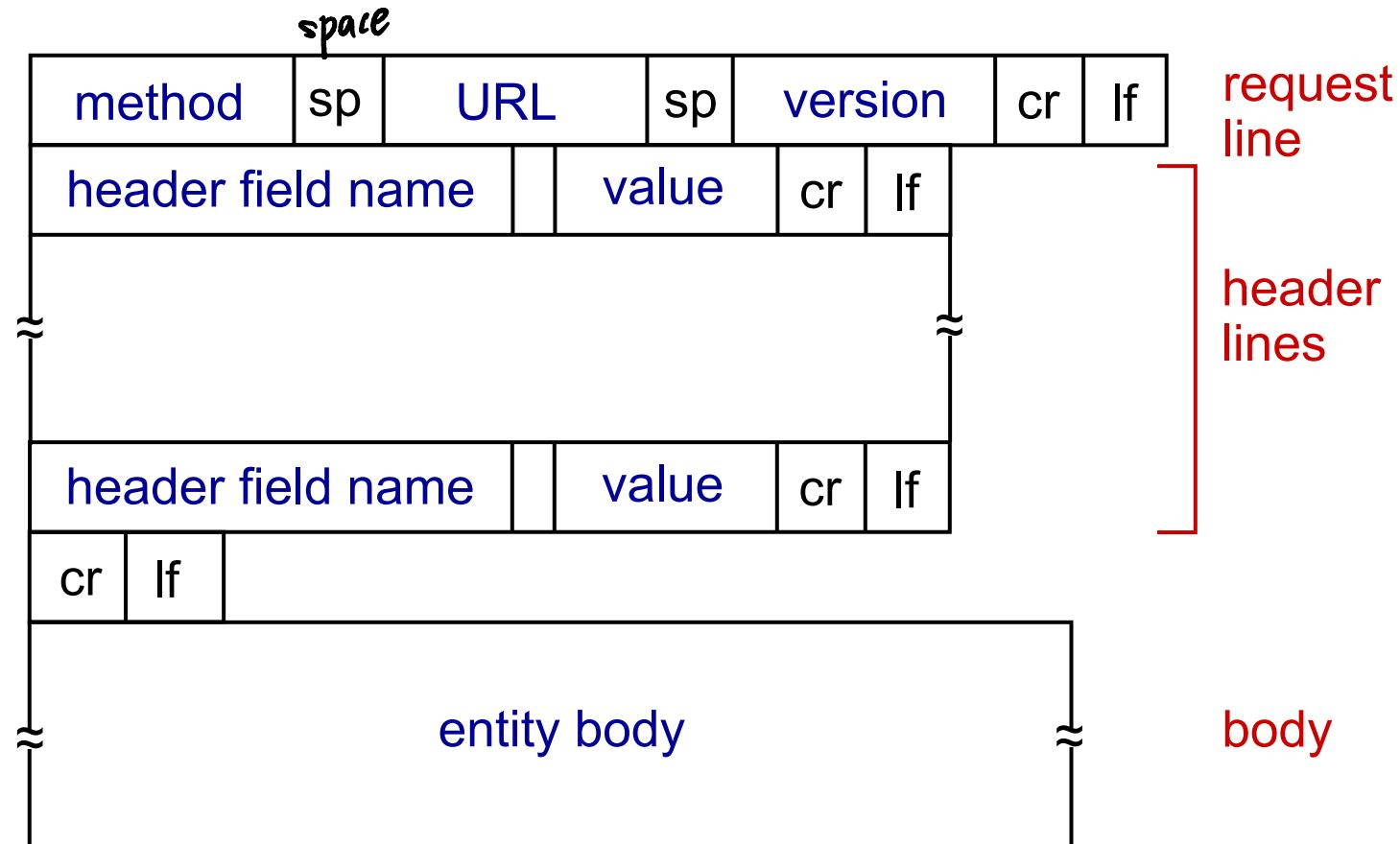
header lines

carriage return, line feed  
at start of line indicates  
end of header lines

carriage return character  
line-feed character

[https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)

# HTTP request message: general format



# Other HTTP request messages

## POST method:

在服务器产生影响不同

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

## GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

[www.somesite.com/animalsearch?monkeys&banana](http://www.somesite.com/animalsearch?monkeys&banana)

仅用于知道 data 大小作内存准备

## HEAD method:

get 的变种，不返回服务器的数据

- requests headers (only) that would be returned if specified URL were requested with an HTTP GET method.

## PUT method:

post 的变种 在服务器产生影响相同

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

# Other HTTP request messages

## POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

## GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

[www.somesite.com/animalsearch?monkeys&banana](http://www.somesite.com/animalsearch?monkeys&banana)

## HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

## PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

# Other HTTP request messages

Properties of request methods

Request method ↴	RFC ↴	Request has payload body ↴	Response has payload body ↴	Safe ↴	Idempotent ↴	Cacheable ↴
<b>GET</b>	RFC 9110 ↗	Optional	Yes	Yes	Yes	Yes
<b>HEAD</b>	RFC 9110 ↗	Optional	No	Yes	Yes	Yes
<b>POST</b>	RFC 9110 ↗	Yes	Yes	No	No	Yes
<b>PUT</b>	RFC 9110 ↗	Yes	Yes	No	Yes	No
<b>DELETE</b>	RFC 9110 ↗	Optional	Yes	No	Yes	No
<b>CONNECT</b>	RFC 9110 ↗	Optional	Yes	No	No	No
<b>OPTIONS</b>	RFC 9110 ↗	Optional	Yes	Yes	Yes	No
<b>TRACE</b>	RFC 9110 ↗	No	Yes	Yes	Yes	No
<b>PATCH</b>	RFC 5789 ↗	Yes	Yes	No	No	No

# HTTP response message

status line (protocol  
status code status phrase)

header  
lines

data, e.g., requested  
HTML file

狀態碼 → 簡單描述  
HTTP/1.1 200 OK\r\nDate: Sun, 26 Sep 2010 20:09:20 GMT\r\nServer: Apache/2.0.52 (CentOS)\r\nLast-Modified: Tue, 30 Oct 2007 17:00:02  
GMT\r\nETag: "17dc6-a5c-bf716880"\r\n每個資源有唯一id. 帮助浏览器判断内容  
Accept-Ranges: bytes\r\nContent-Length: 2652\r\nKeep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\nContent-Type: text/html; charset=ISO-8859-1\r\n\r\ndata data data data data ...

[https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)

# HTTP response status codes

需要继续进行操作

- status code appears in 1st line in server-to-client response message.
- some sample codes:

## 200 OK

- request succeeded, requested object later in this message

## 301 Moved Permanently 重定向.更换网址.

- requested object moved, new location specified later in this message (in Location: field)

## 400 Bad Request 客户端出错.

- request msg not understood by server

## 404 Not Found

- requested document not found on this server

## 505 HTTP Version Not Supported

服务器端发生错误

1. Informational responses ( 100 – 199 )

2. Successful responses ( 200 – 299 )

3. Redirection messages ( 300 – 399 )

4. Client error responses ( 400 – 499 )

5. Server error responses ( 500 – 599 )

http://本是 stateless 的，只是通过 cookie 使之可以识别状态

请求发送时会带上 cookie 的信息。  
cookie 被保存在本地文件夹。

# Maintaining user/server state: cookies

Web sites and client browser use  
**cookies** to maintain some state  
between transactions

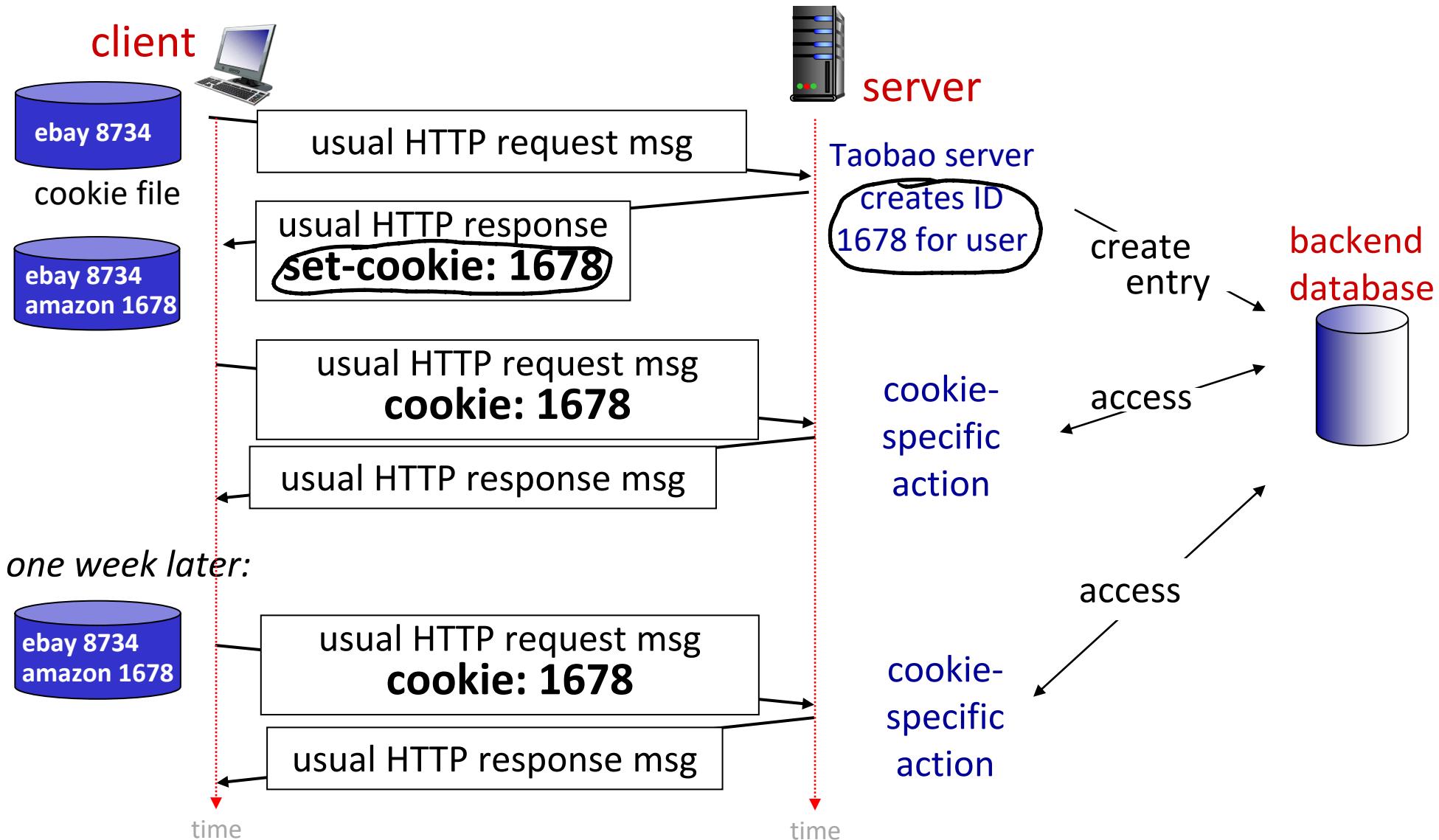
*four components:*

- 1) cookie header line of HTTP *response* message
- 2) cookie header line in next HTTP *request* message
- 3) cookie file kept on user's host,  
managed by user's browser
- 4) back-end database at Web site

**Example:**

- Susan uses browser on laptop,  
visits specific e-commerce site  
for first time
- when initial HTTP requests  
arrives at site, site creates:
  - unique ID (aka “cookie”)
  - entry in backend database  
for ID
  - subsequent HTTP requests  
from Susan to this site will  
contain cookie ID value,  
allowing site to “identify”  
Susan

# Maintaining user/server state: cookies



# HTTP cookies: comments

## *What cookies can be used for:*

- authorization - 部分 cookie 有期限.
- shopping carts
- recommendations
- user session state (Web e-mail)

## *Challenge: How to keep state:*

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: HTTP messages carry state

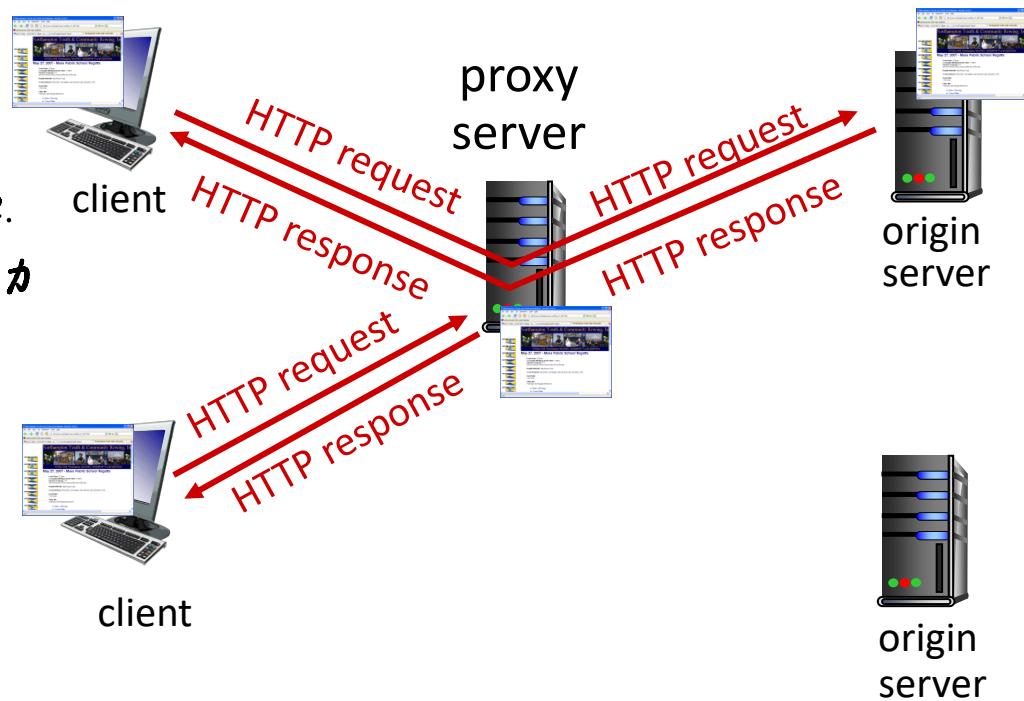
aside  
*cookies and privacy:*

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

# Web caches (proxy servers)

**Goal:** satisfy client request without involving origin server

- user configures browser to point to a Web cache 容量有限 布署在接入网附近
- browser sends all HTTP requests to cache 先缓存在本地 least commonly use.
  - if object in cache: cache returns object to client
  - else cache requests object from origin server, caches received object, then returns object to client



# Web caches (proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

## *Why* Web caching?

- reduce response time for client request
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables “poor” content providers to more effectively deliver content

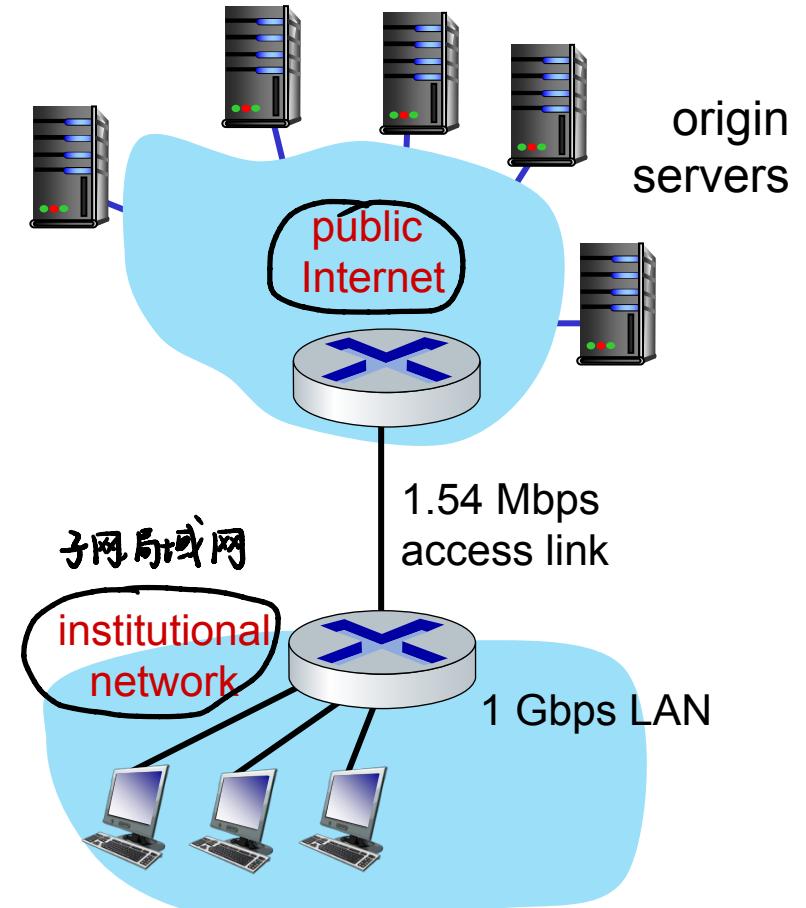
# Caching example

## Scenario:

- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Average request rate from browsers to origin servers: 15/sec
  - average data rate to browsers: 1.50 Mbps

## Performance:

- round trip time*
- access link utilization = .97 *queuing delay 非常高*
  - LAN utilization: .0015 *问题*
  - end-end delay = Internet delay +  
access link delay + LAN delay  
= 2 sec + minutes + usecs



# Caching example: buy a faster access link

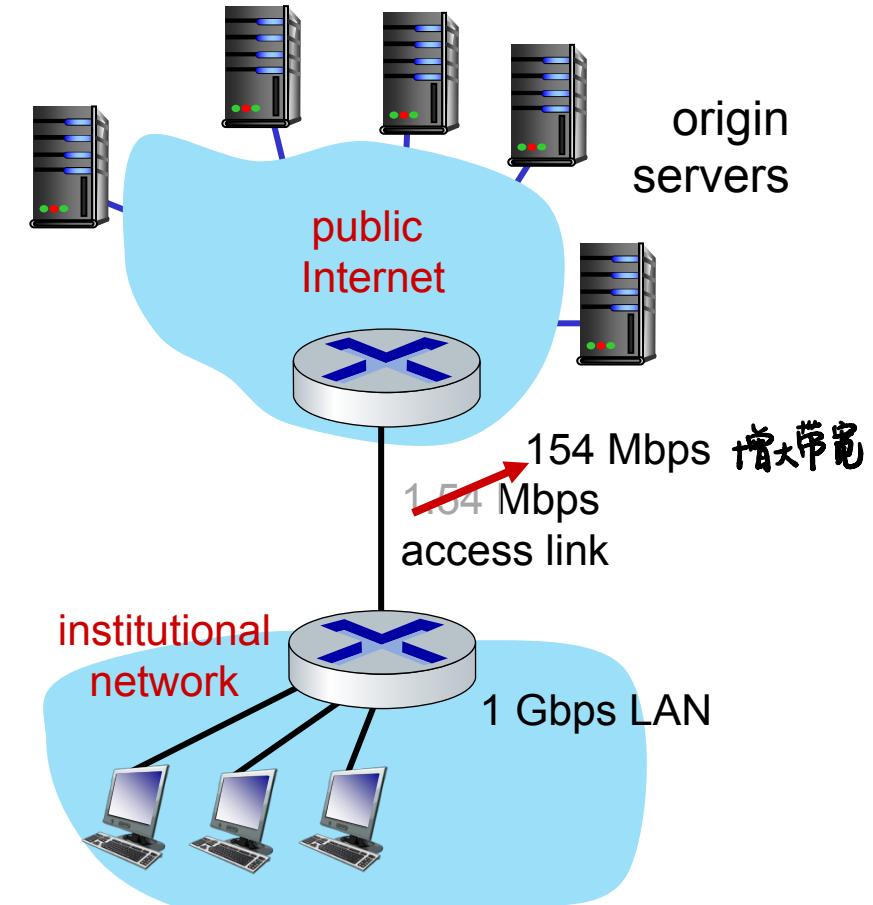
## Scenario:

- access link rate: ~~1.54~~ Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

## Performance:

- LAN utilization: .0015
- access link utilization = ~~.07~~ → .0097
- end-end delay = Internet delay +  
access link delay + LAN delay  
= 2 sec + ~~minutes~~ + usecs

*Cost:* faster access link (expensive!) → msecs



# Caching example: install a web cache

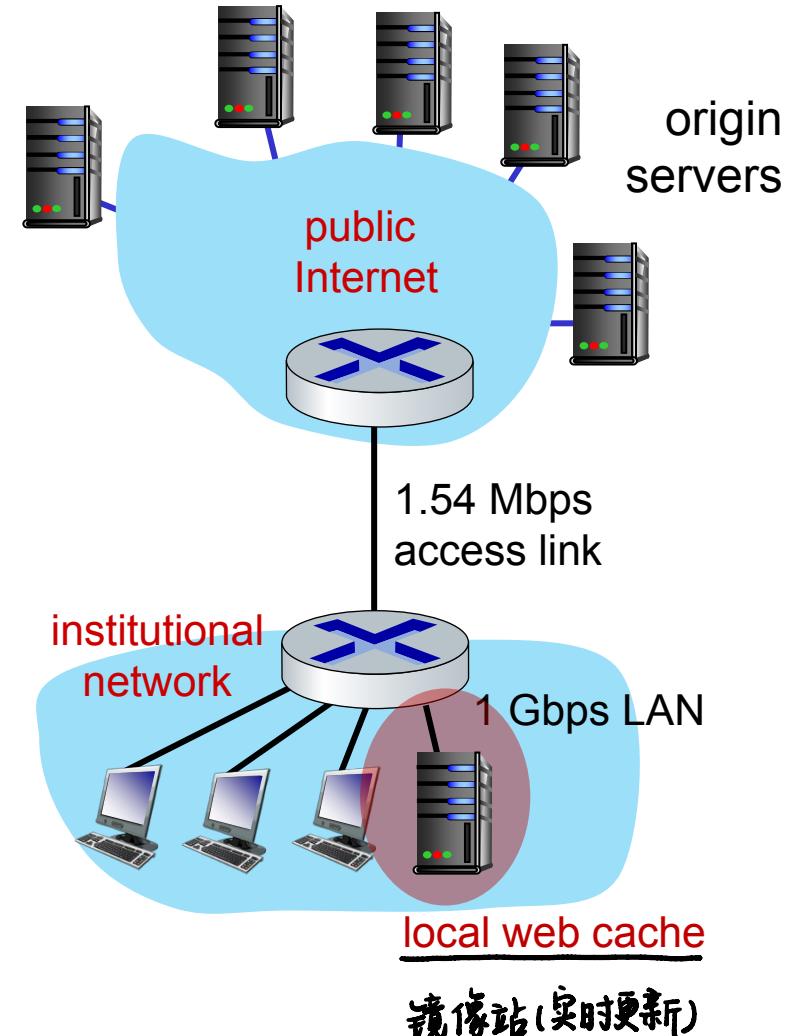
## Scenario:

- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

## Performance:

- LAN utilization: .?
- access link utilization = ? *How to compute link utilization, delay?*
- average end-end delay = ?

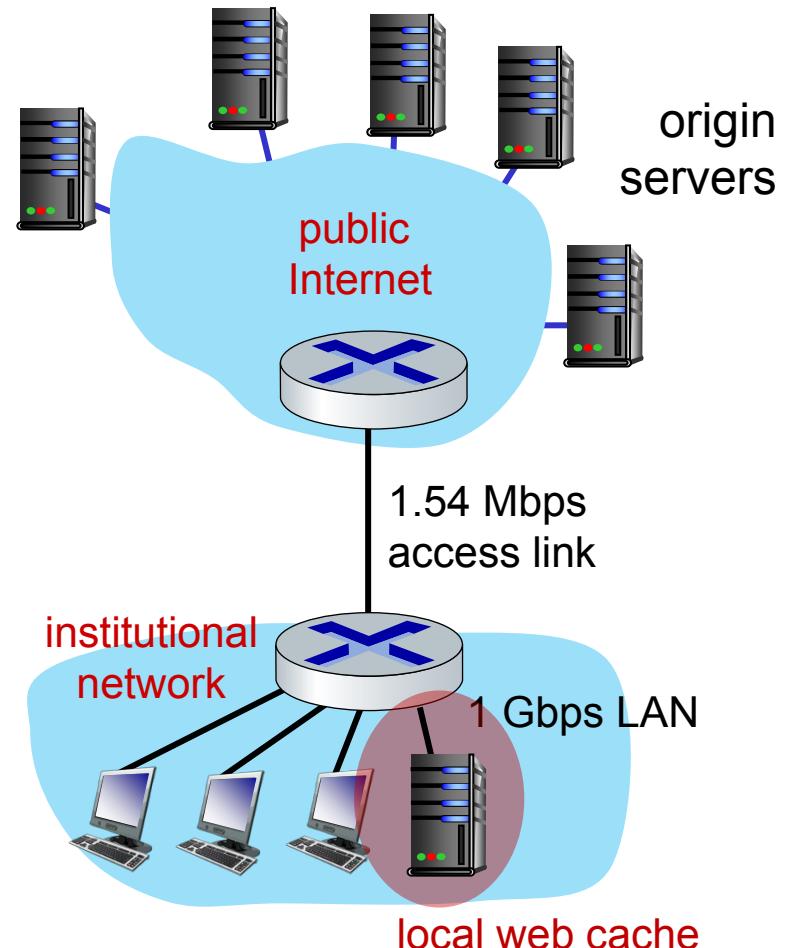
*Cost:* web cache (cheap!)



# Caching example: install a web cache

Calculating access link utilization, end-end delay with cache: *查询时已经在 web cache 里面 -- hit*

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link  
 $= \textcircled{0.6} * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
- utilization =  $0.9/1.54 = .58$
- average end-end delay (estimation)  
 $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$   
 $= 0.6 * (2.01) + 0.4 * (\text{~msecs}) = \sim 1.2 \text{ secs}$



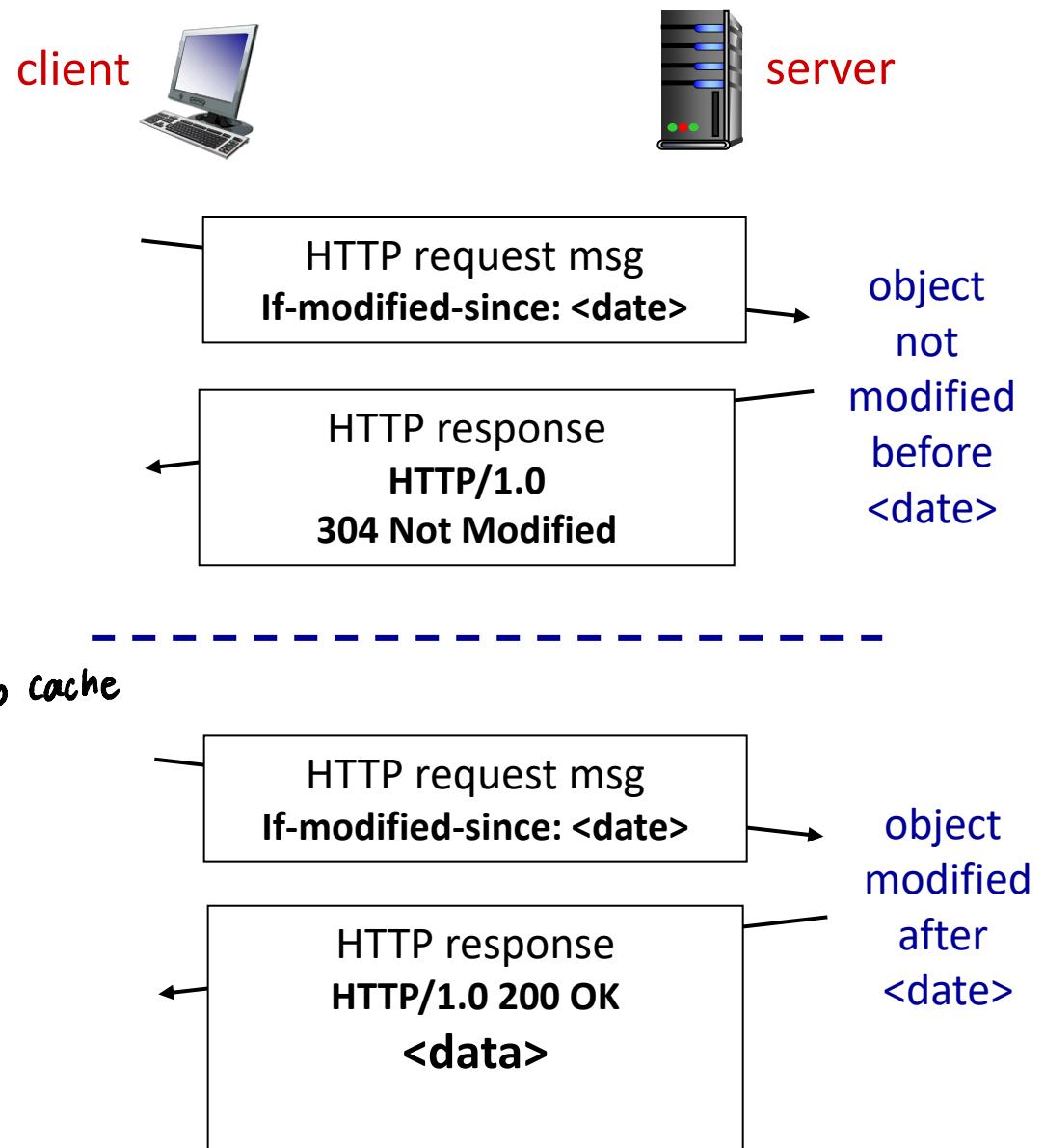
*lower average end-end delay than with 154 Mbps link (and cheaper too!)*

# Conditional GET

web cache 额外常用

**Goal:** don't send object if cache has up-to-date cached version

- no object transmission delay
- lower link utilization
- **cache:** specify date of cached copy in HTTP request
  - If-modified-since: <date> 何时更新, 未更新则 web cache  
而已有, 无须重复获取.
- **server:** response contains no object if cached copy is up-to-date:  
**HTTP/1.0 304 Not Modified**



# HTTP/2

*Key goal:* decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection

- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s) 都要按顺序.
- loss recovery (retransmitting lost TCP segments) stalls object transmission

# HTTP/2

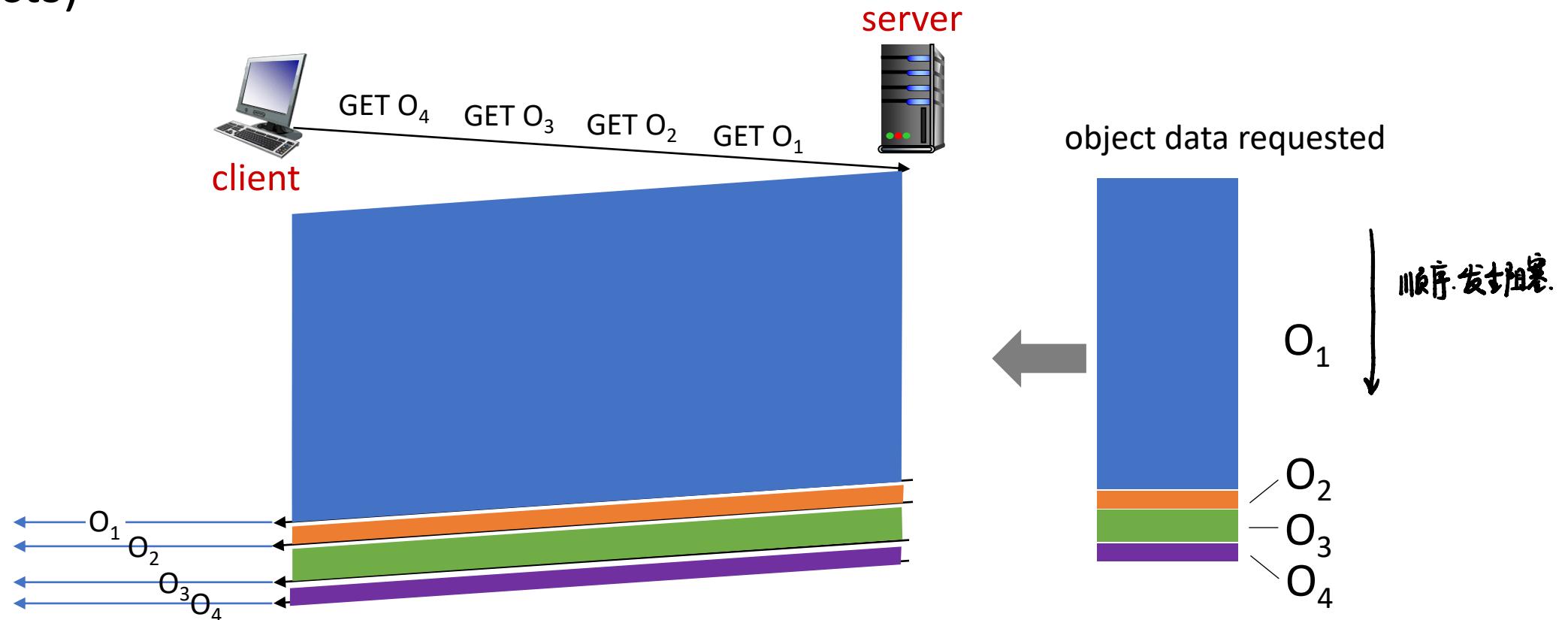
*Key goal:* decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- ② ■ transmission order of requested objects based on client-specified object priority (not necessarily FCFS) 每个请求得到一定资源
- ③ ■ push unrequested objects to client 识别请求A后可能继续请求B，并把B提前push出来。
- ④ ■ divide objects into frames, schedule frames to mitigate HOL blocking

# HTTP/2: mitigating HOL blocking

HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



*objects delivered in order requested:  $O_2$ ,  $O_3$ ,  $O_4$  wait behind  $O_1$*

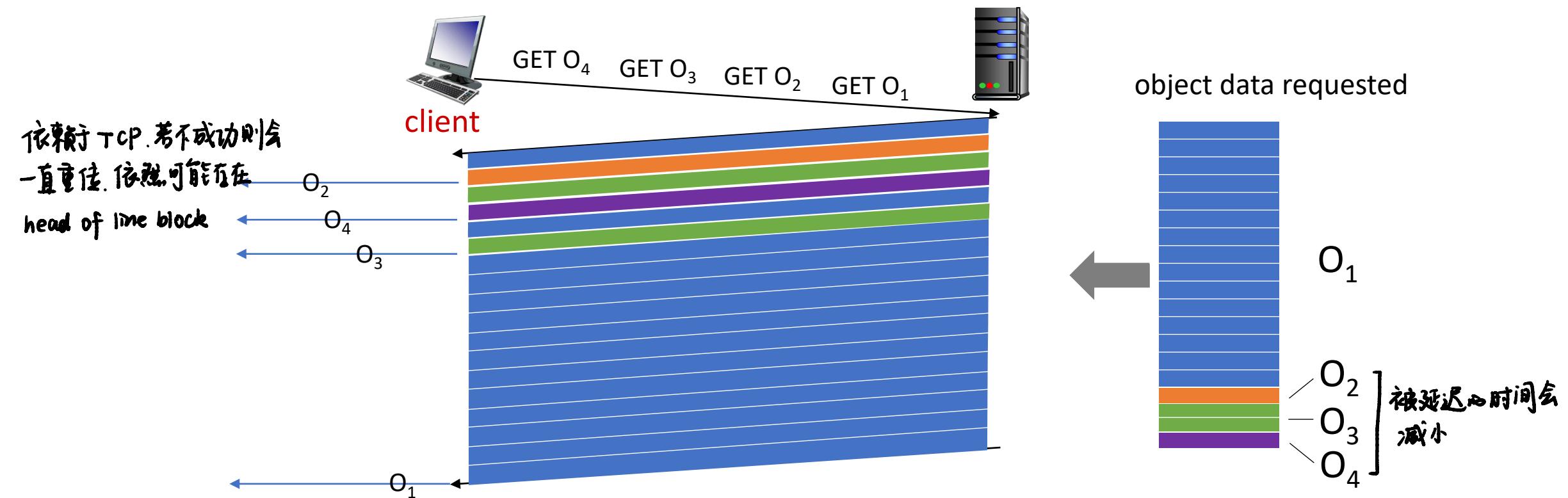
# HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, **frame** transmission interleaved

固定帧大小

帧 每帧的调度不再遵循 FCFS

server



# HTTP/2 to HTTP/3

*Key goal:* decreased delay in multi-object HTTP requests

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over vanilla TCP connection
- **HTTP/3:** adds security , per object error- and congestion-control (more pipelining) over **UDP** 可以不继续重传.
  - more on HTTP/3 in transport layer 将可靠的部分单独移出.

https 默认端口号为 443.

# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



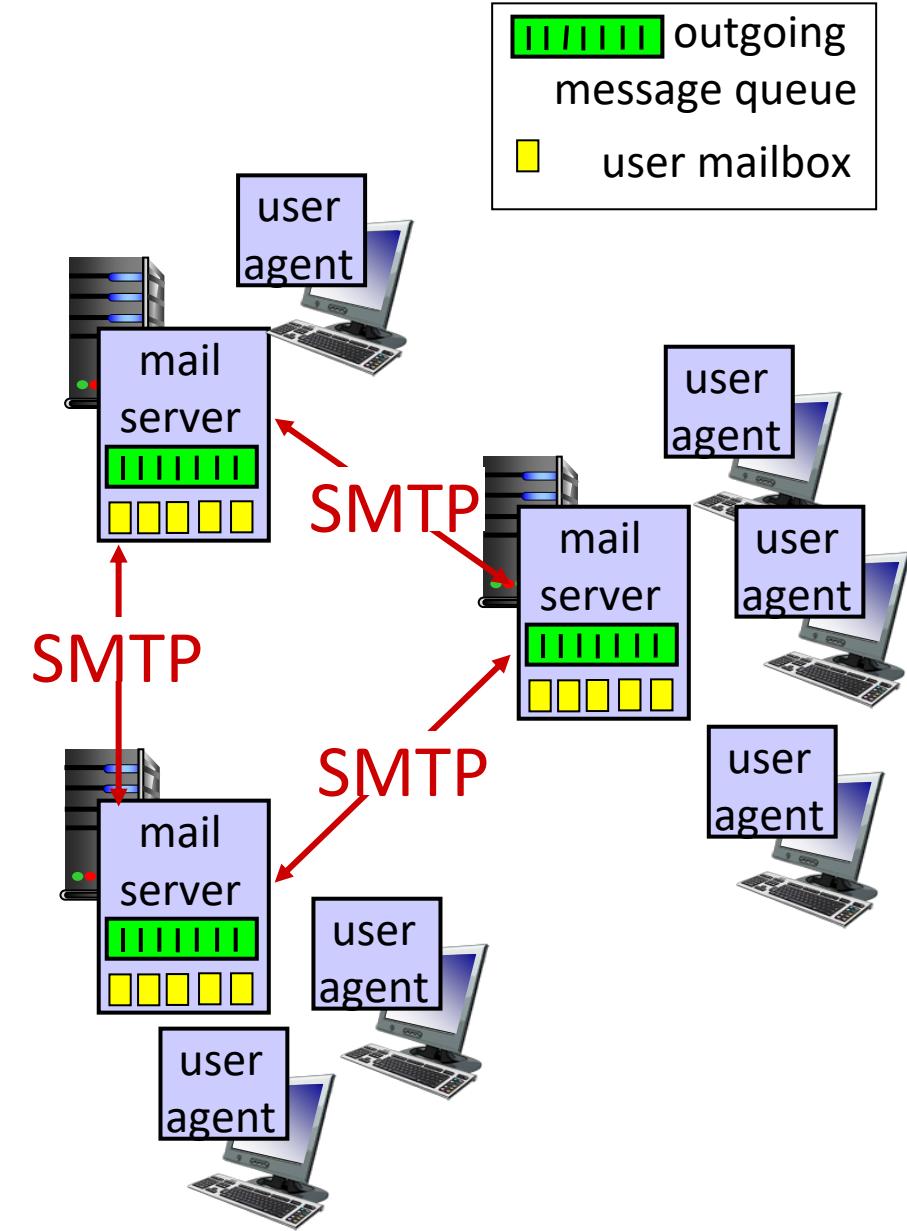
# E-mail

Three major components:

- user agents
- mail servers 朝向公网的服务器.
- simple mail transfer protocol: SMTP

## User Agent

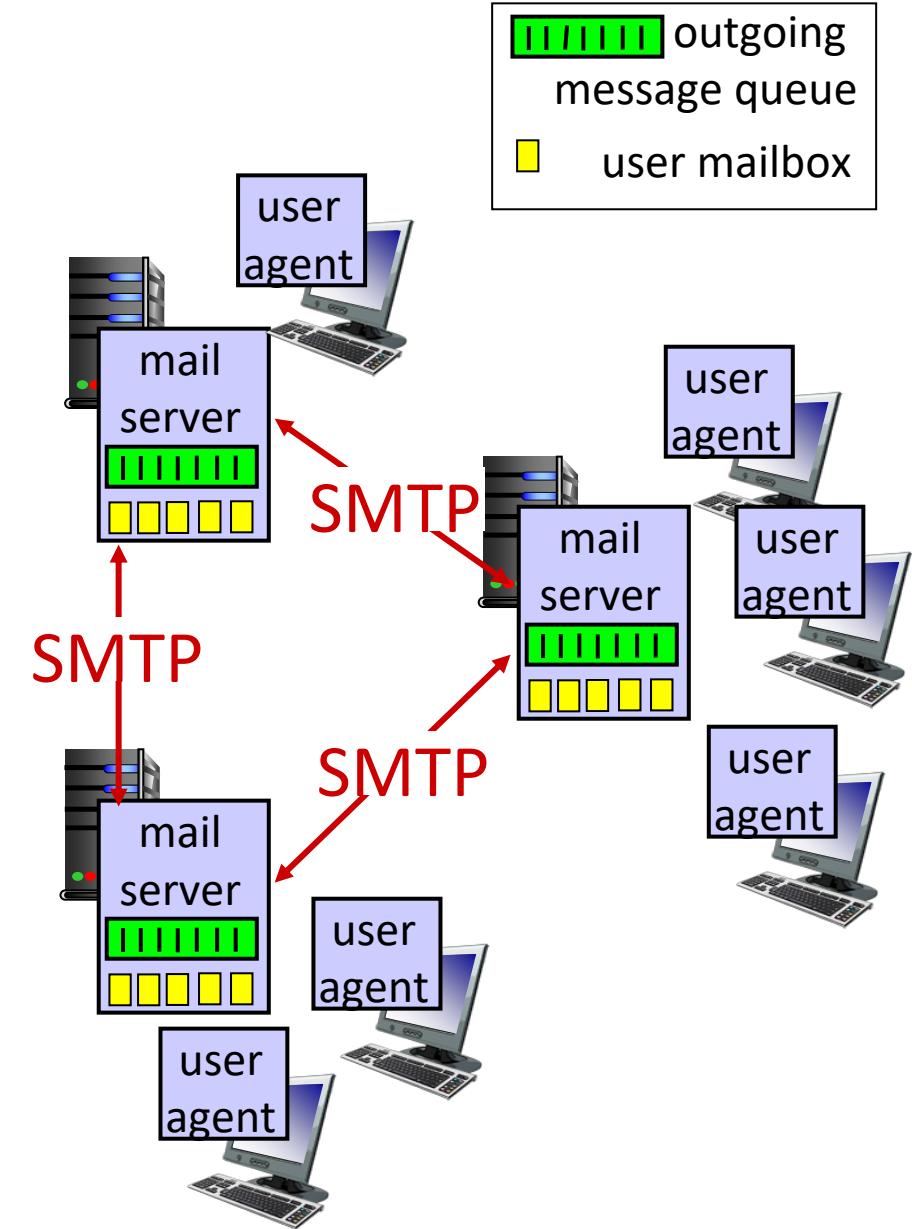
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, phone mail client
- outgoing, incoming messages stored on server



# E-mail: mail servers

## mail servers:

- mailbox contains incoming messages for user
- message queue of outgoing (to be sent) mail messages 向外发送的消息队列
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - “server”: receiving mail server



# Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message “to” bob@someschool.edu

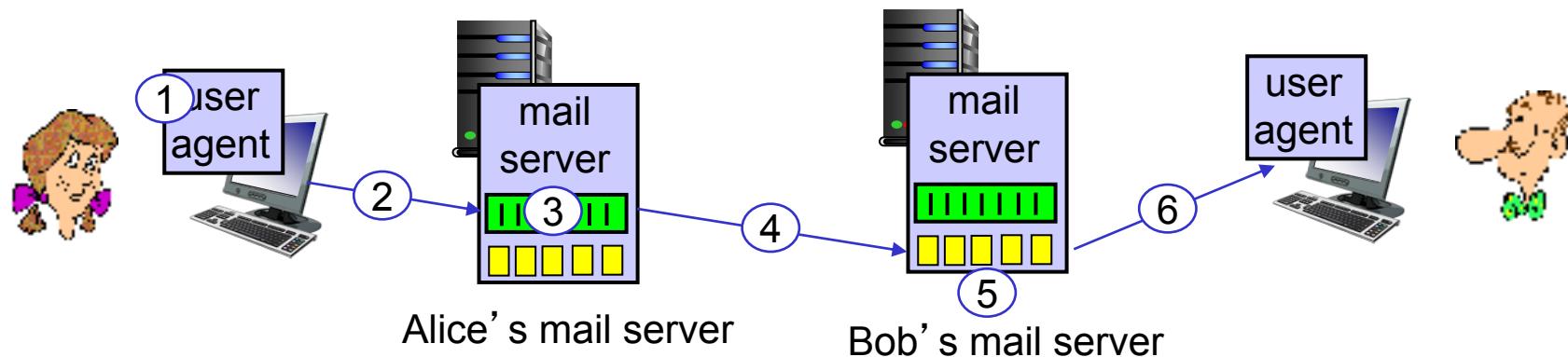
2) Alice’s UA sends message to her mail server; message placed in message queue

3) client side of SMTP opens TCP connection with Bob’s mail server

4) SMTP client sends Alice’s message over the TCP connection

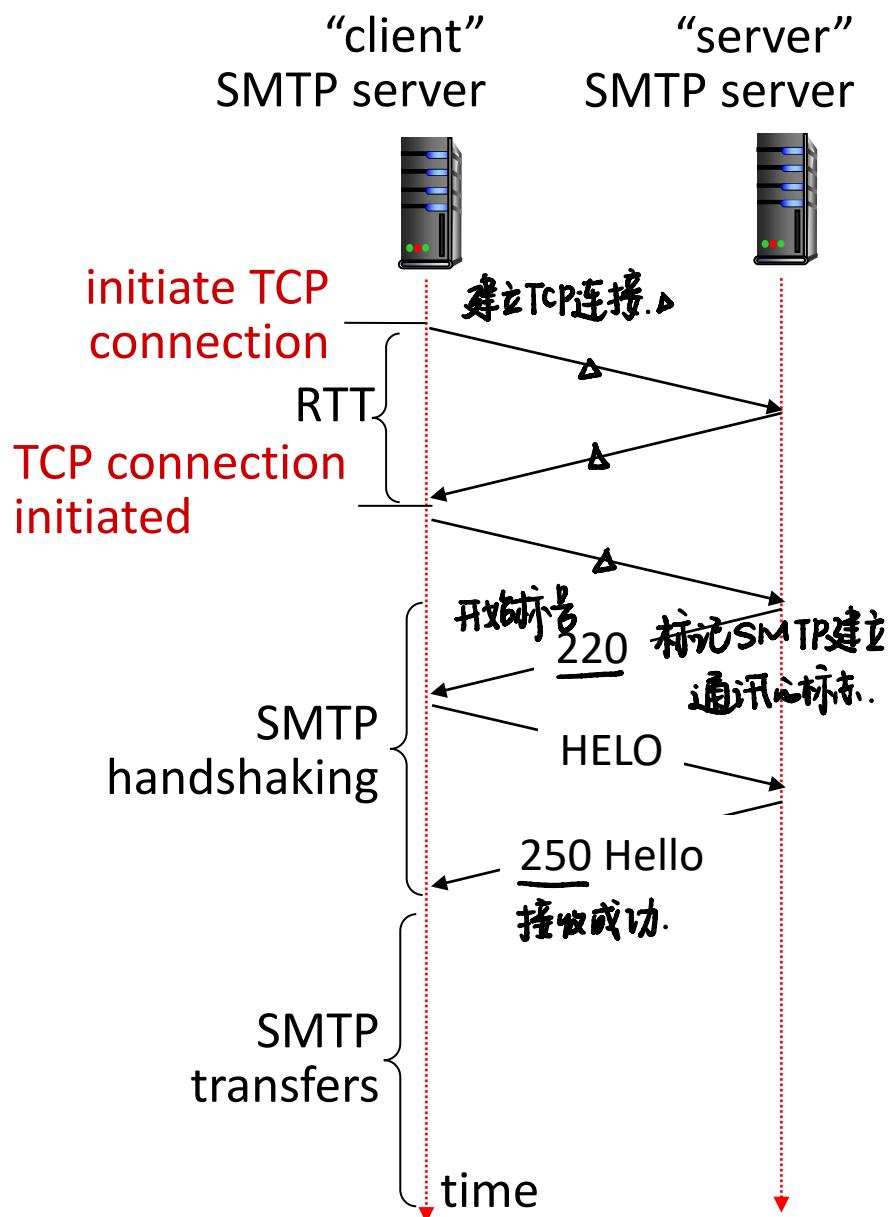
5) Bob’s mail server places the message in Bob’s mailbox

6) Bob invokes his user agent to read message



# SMTP RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
  - direct transfer: from sending server (acting like client) to receiving server
- three phases of transfer
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase



# Sample SMTP interaction

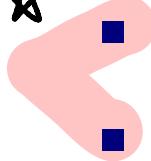
本文 package の内容.

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
发自 → C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
发给 → C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: () 以点或新行结束.
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

# SMTP: closing observations

## *comparison with HTTP:*

区别



- HTTP: pull 拉下来 请求什么给什么
- SMTP: push (反向) 内容推到服务器处

- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in a multipart message

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

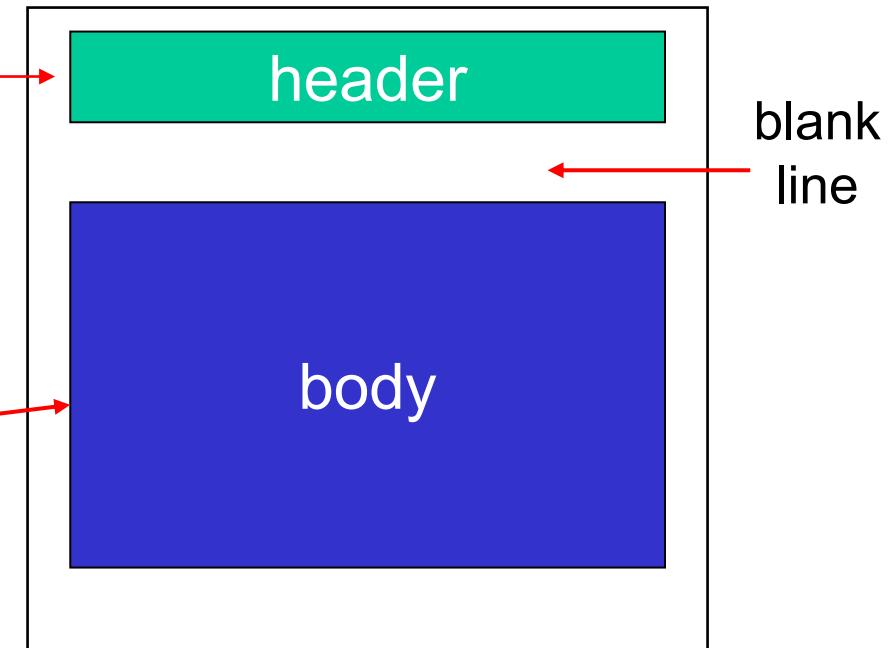
# Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

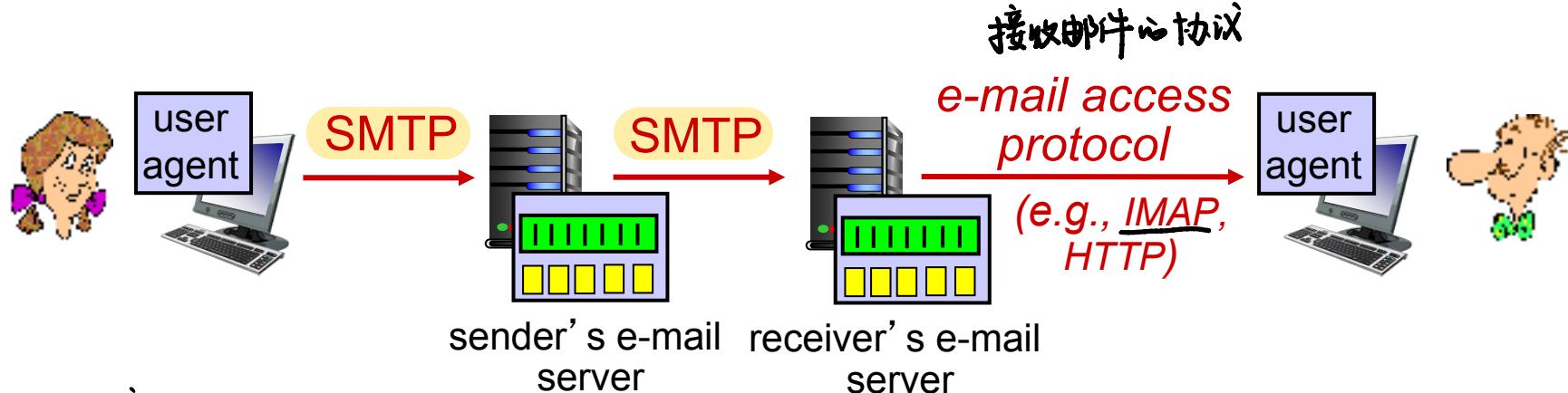
RFC 822 defines *syntax* for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the “message”, ASCII characters only



# Mail access protocols



默认端口是25，加密后端口为465。

- **SMTP:** delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - **IMAP:** Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- **HTTP:** gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

浏览器上发邮件直接使用HTTP协议。内部服务器那端用SMTP。

# Application Layer: Overview

- Principles of network applications
  - Web and HTTP
  - E-mail, SMTP, IMAP
  - The Domain Name System DNS
- P2P applications
  - video streaming and content distribution networks
  - socket programming with UDP and TCP



# DNS: Domain Name System

分布式数据库

提供域名和 IP 地址的转换服务。

*people:* many identifiers:

- student ID, name, passport #

*Internet hosts, routers:*

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

*Domain Name System:*

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol*: hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol* 甚于传输层的 UDP 协议。
  - complexity at network’s “edge”

# DNS: services, structure

## DNS services 浏览器会先访问 DNS 服务器.

- hostname to IP address translation
- host aliasing 别名.
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

顶级域名服务器

distributing & hierarchical

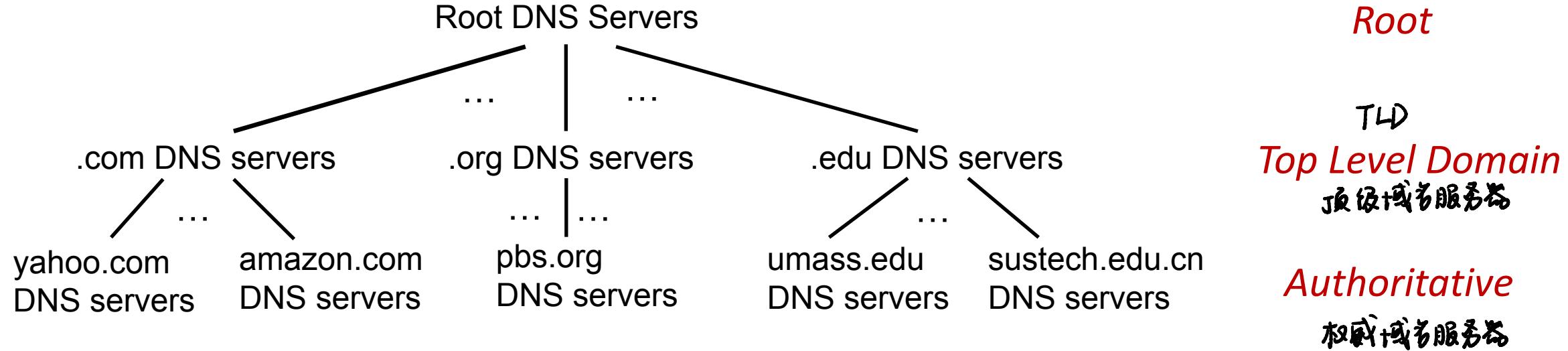
## Q: Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

## A: doesn't scale!

- Comcast DNS servers alone: 600B DNS queries per day

# DNS: a distributed, hierarchical database



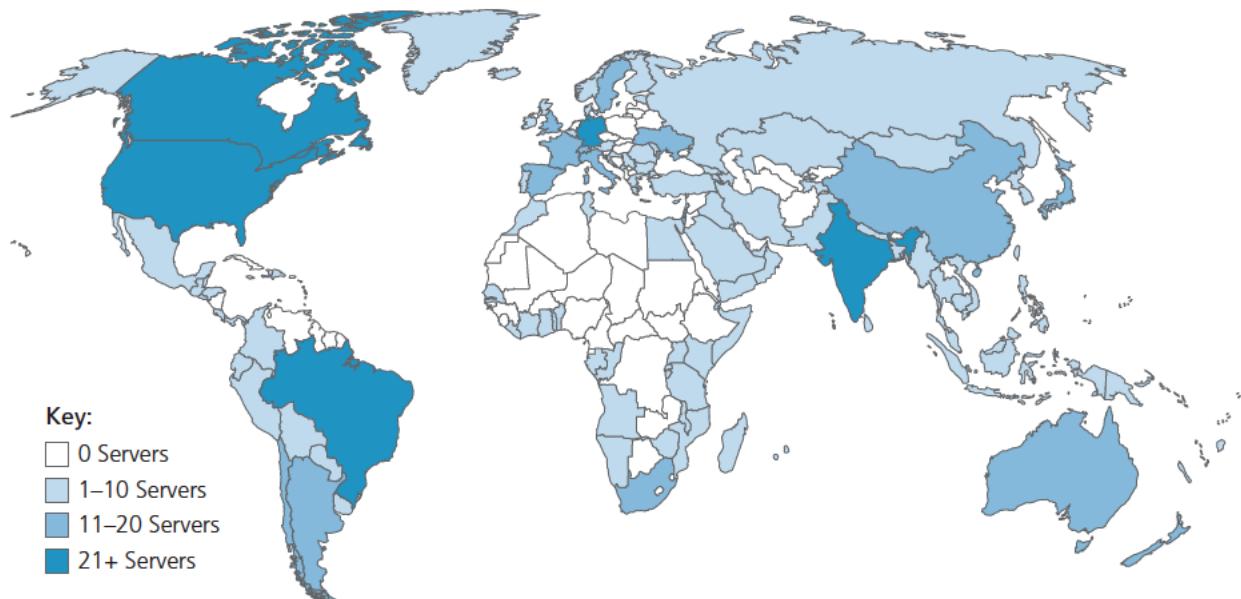
Client wants IP address for [www.amazon.com](http://www.amazon.com); 1<sup>st</sup> approximation:

- client queries root server to find [.com](http://.com) DNS server 返回 TLD 服务器
- client queries .com DNS server to get [amazon.com](http://amazon.com) DNS server 返回 ns.amazon.com
- client queries amazon.com DNS server to get IP address for [www.amazon.com](http://www.amazon.com)

# DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name “servers”  
worldwide each “server” replicated  
many times (~200 servers in US)



# Top-Level Domain, and authoritative servers

## Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

## Authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

# Local DNS name servers

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

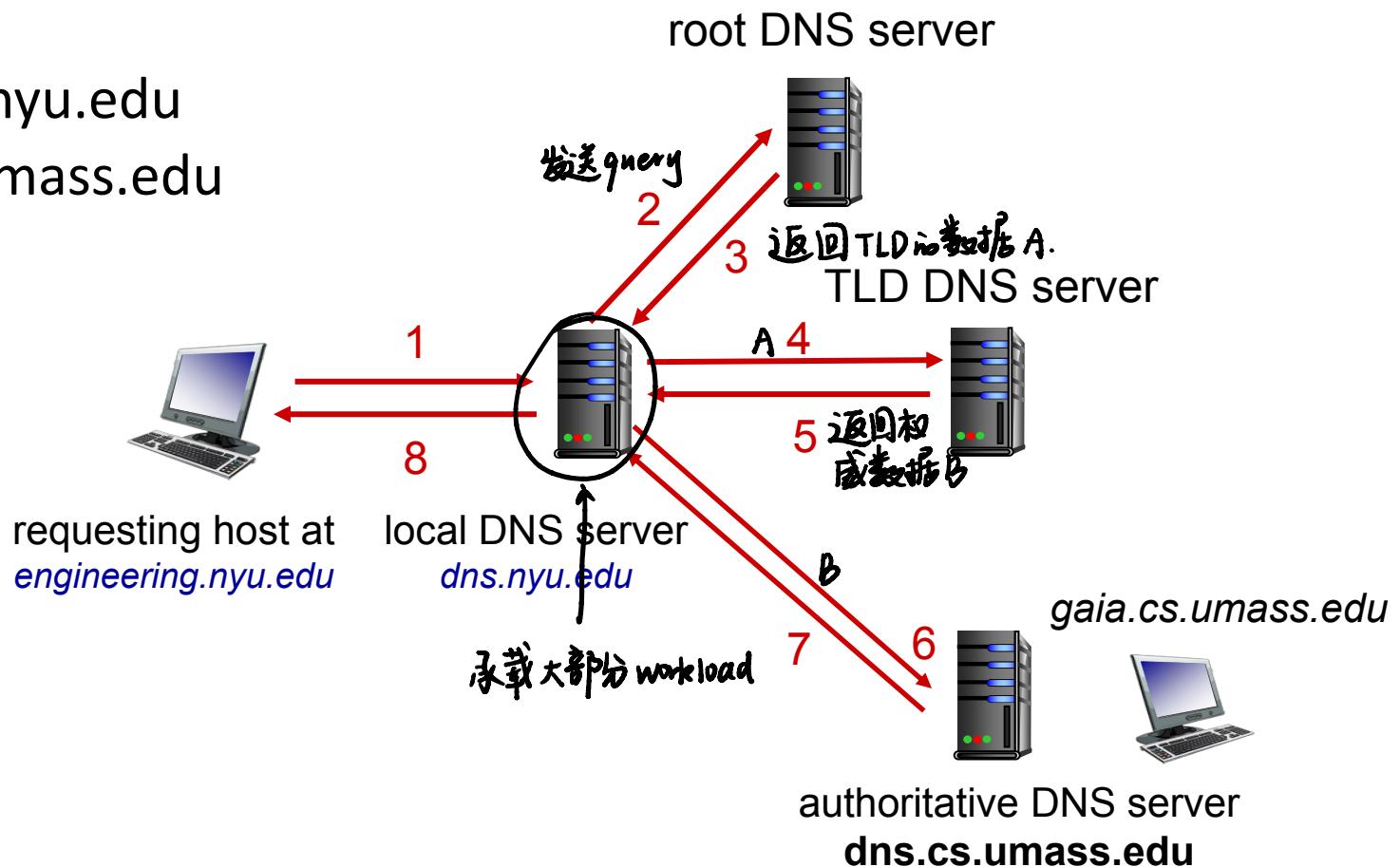
# DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

大部分时间：  
不断地提交请求

## Iterated query:

- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”

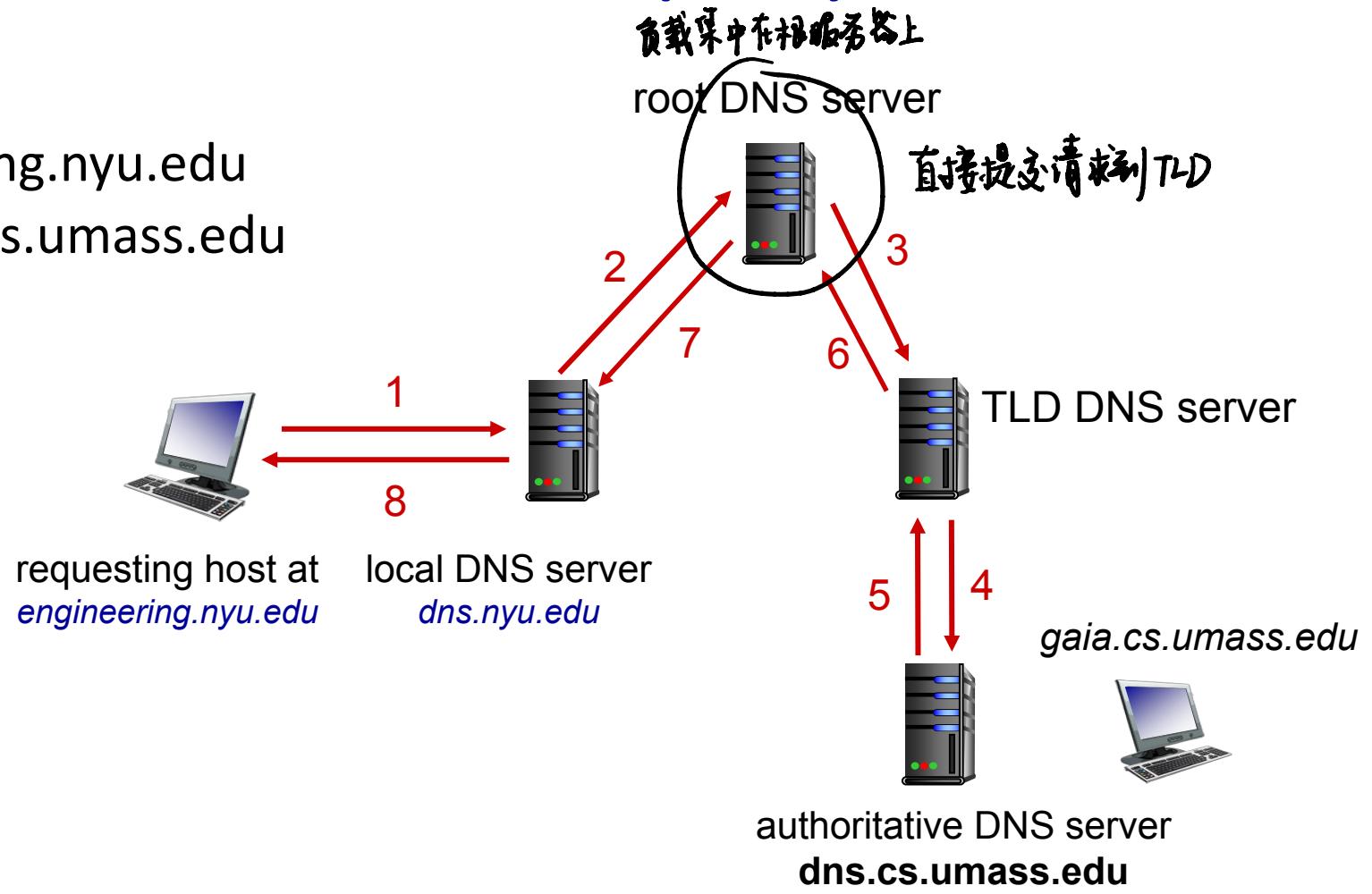


# DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

## Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



# Caching, Updating DNS Records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL) time to leave 何时有效
  - TLD servers typically cached in local name servers 过期则需重新查询.
  - thus root name servers not often visited
- cached entries may be *out-of-date* (*best-effort name-to-address translation!*)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!

example 下载 Winbind  
dig.exe 直接运行  
dig +trace sakai.sustech.edu.cn.  
根域名服务器：本地配置文件

# DNS records

**DNS:** distributed database storing **resource records (RR)**

RR format: (name, 域名, value, 遵回值, type, ttl)  
 {  
 IP地址  
 TLD服务器地址  
 Authority  
 }

## type=A

- name is hostname
- value is IP address

## type=NS name server. 允许直接继续查询

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

## type=CNAME

- name is alias name for some “canonical” (the real) name 别名.
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name 计算机集群内部有内部

## type=MX 外显的IP, P, IT.

- value is name of **mailserver** associated with name

dig +noerror @某服务器 网址

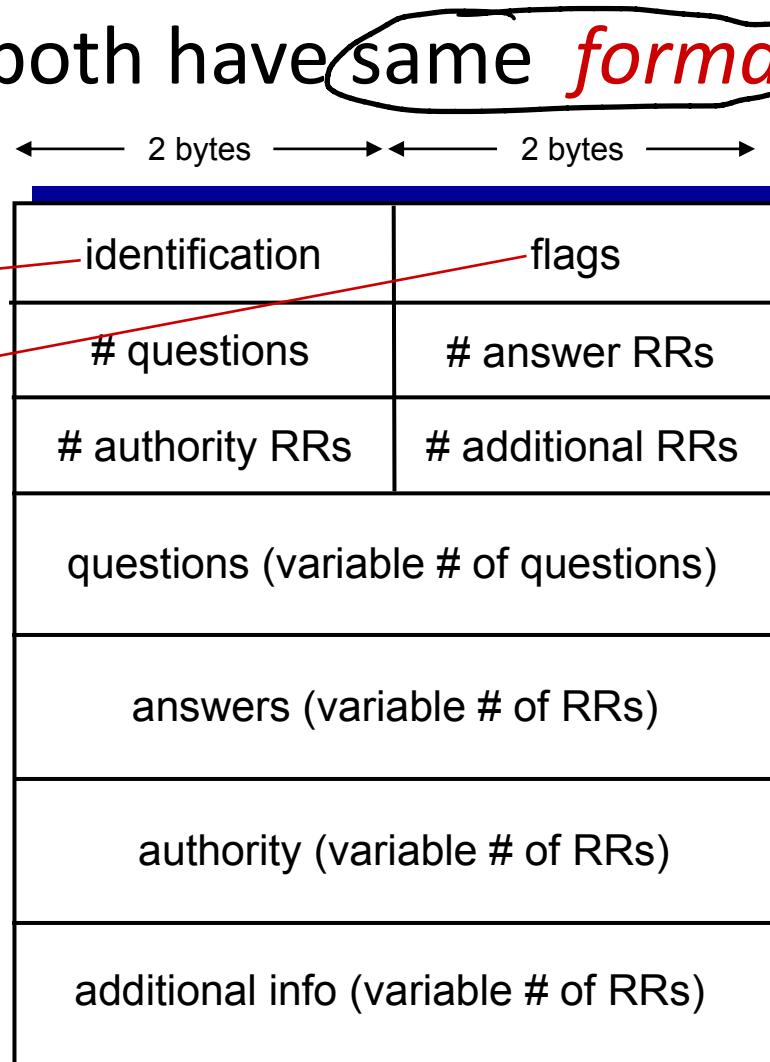
A IPV4 32位.  
AAAA IPV6 128位

# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*.

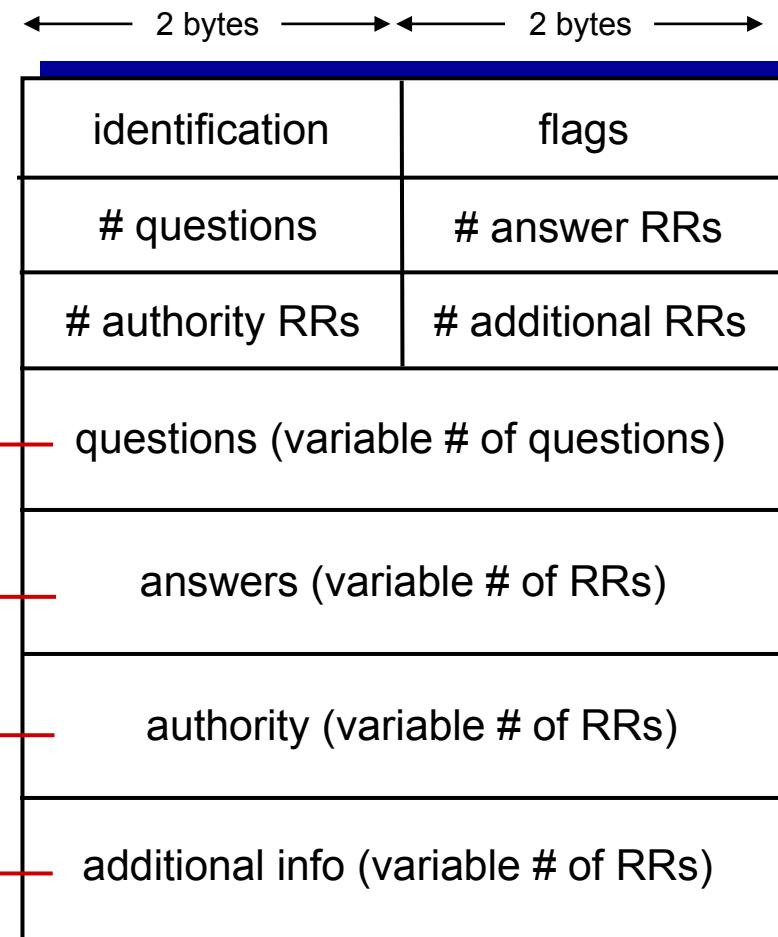
message header:

- **identification:** 16 bit # for query,  
reply to query uses same #
- **flags:**
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:



name, type fields for a query

RRs in response to query

records for authoritative servers

additional “helpful” info that may be used

*dig/wireshark.*

# Inserting records into DNS

Example: new startup “Network Utopia”

ICANN组织来管理.

- register name networkuptopia.com at DNS registrar (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:  
(networkutopia.com, dns1.networkutopia.com, NS)  
(dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server locally with IP address 212.212.212.1
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com

# DNS security

## DDoS attacks

- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

## Redirect attacks

- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus replies to DNS server, which caches

## Exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

DNSSEC  
[RFC 4033]

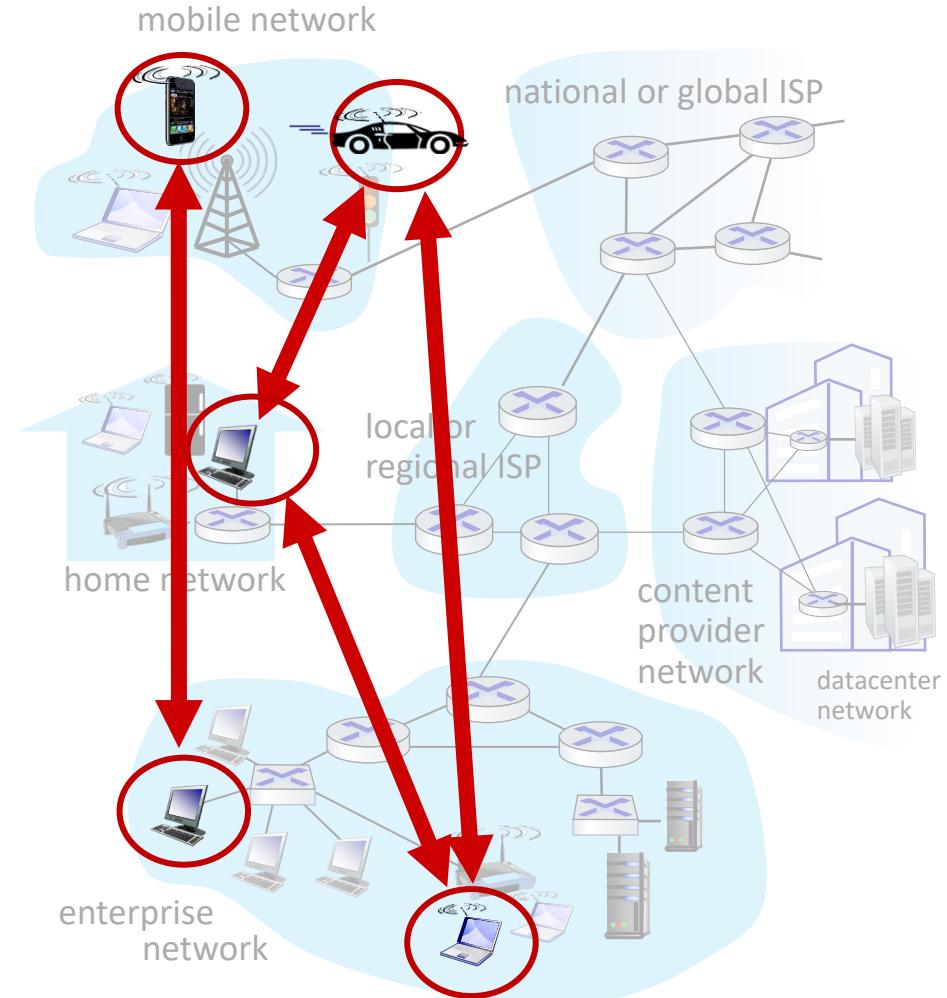
# Application Layer: Overview

- Principles of network applications
  - Web and HTTP
  - E-mail, SMTP, IMAP
  - The Domain Name System DNS
- P2P applications
  - video streaming and content distribution networks
  - socket programming with UDP and TCP



# Peer-to-peer (P2P) architecture

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)

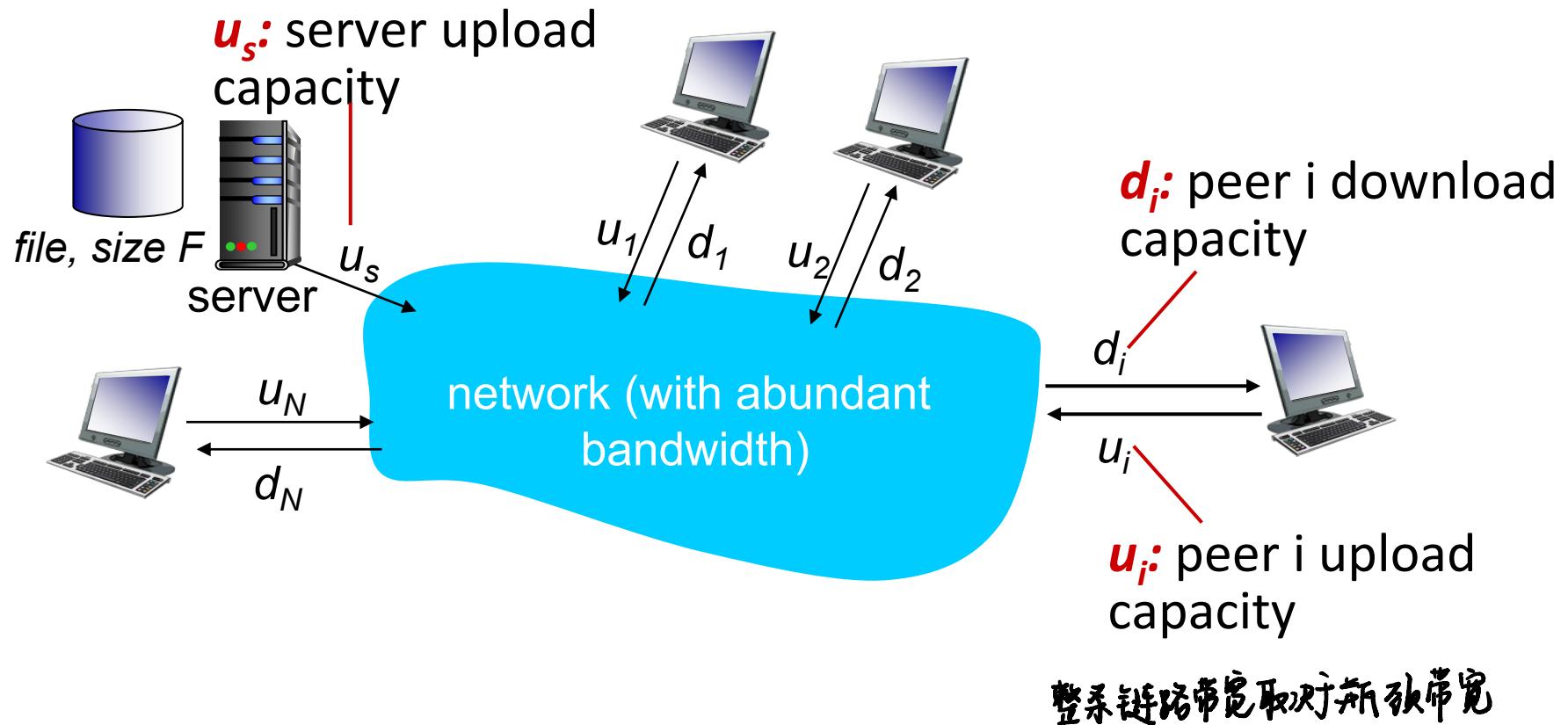


# File distribution: client-server vs P2P

throughput

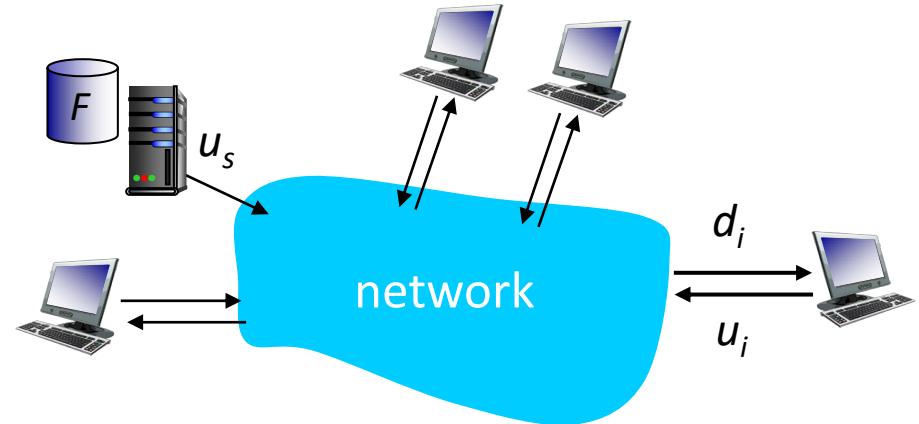
**Q:** how much time to distribute file (size  $F$ ) from one server to  $N$  peers?

- peer upload/download capacity is limited resource



# File distribution time: client-server

- *server transmission*: must sequentially send (upload)  $N$  file copies:
  - time to send one copy:  $F/u_s$
  - time to send  $N$  copies:  $NF/u_s$
- *client*: each client must download file copy
  - $d_{min}$  = min client download rate
  - min client download time:  $F/d_{min}$



上传一个文件需要的时间).

time to distribute  $F$   
to  $N$  clients using  
client-server approach

$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

下载

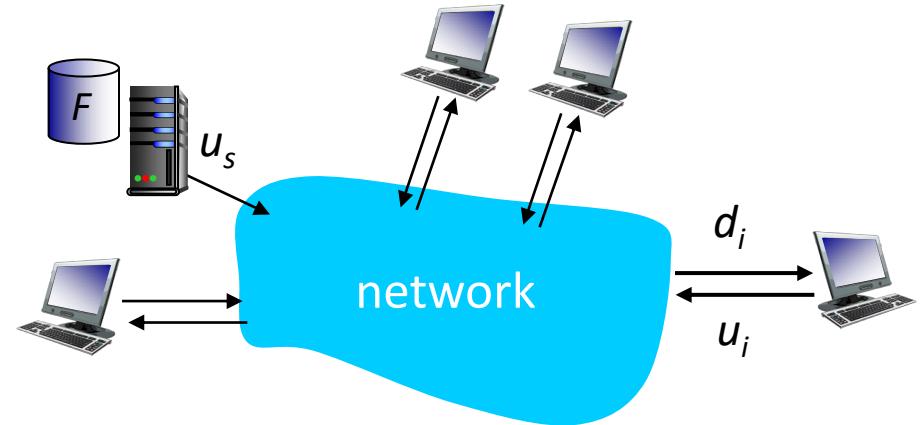
increases linearly in  $N$

# File distribution time: P2P

- *server transmission*: must upload at least one copy:
  - time to send one copy:  $F/u_s$

- *client*: each client must download file copy
  - min client download time:  $F/d_{min}$

- *clients*: as aggregate must download  $NF$  bits
  - max upload rate (limiting max download rate) is  $u_s + \sum u_i$



time to distribute  $F$   
to  $N$  clients using  
P2P approach

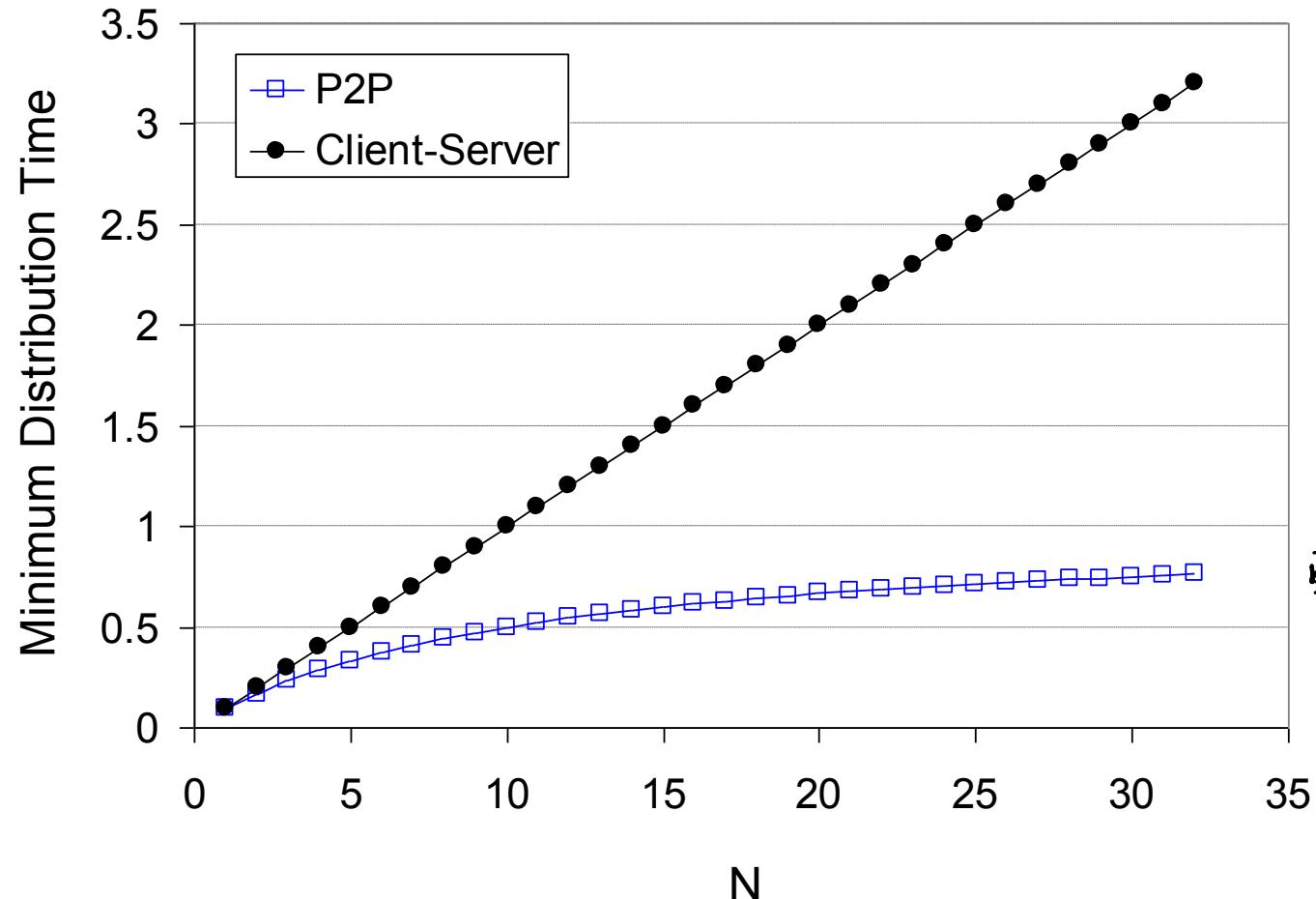
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in  $N$  ...

... but so does this, as each peer brings service capacity

# Client-server vs. P2P: example

client upload rate =  $u$ ,  $F/u = 1$  hour,  $u_s = 10u$ ,  $d_{min} \geq u_s$



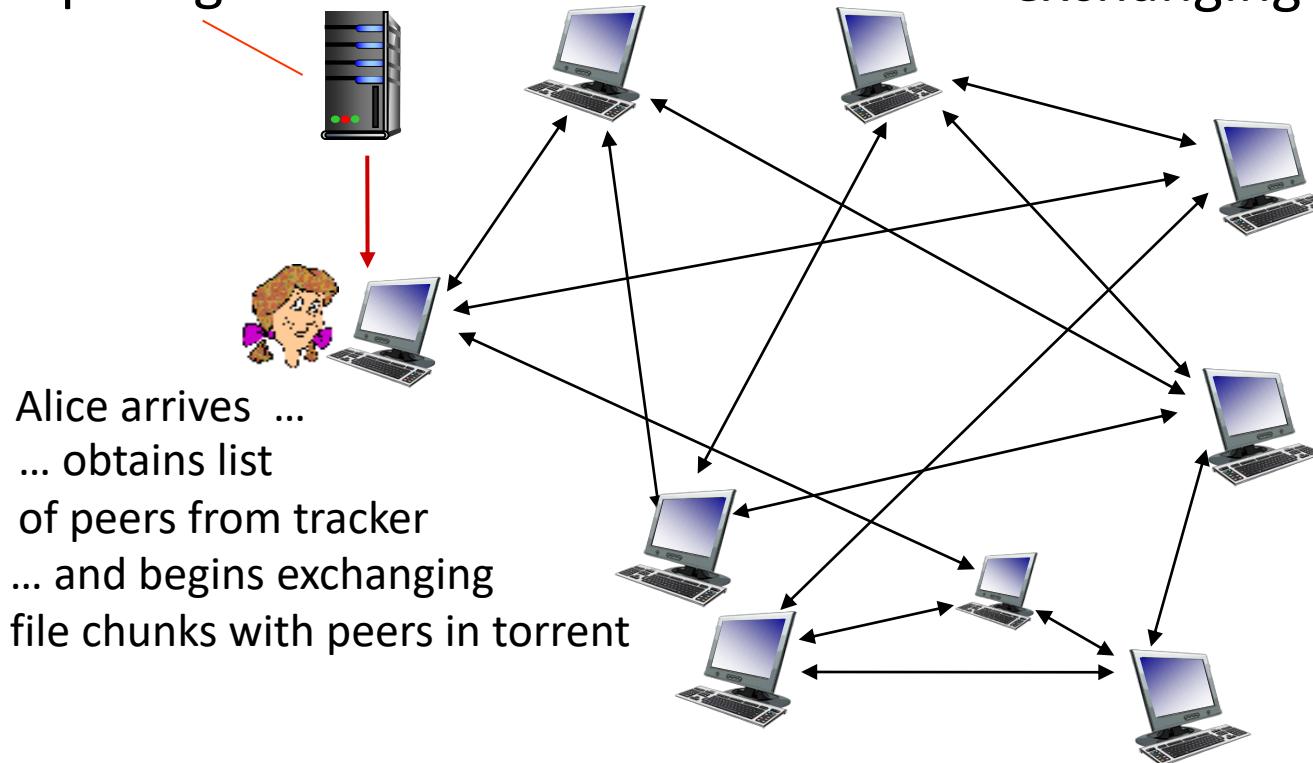
# P2P file distribution: BitTorrent

文件共享软件.

文件块

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

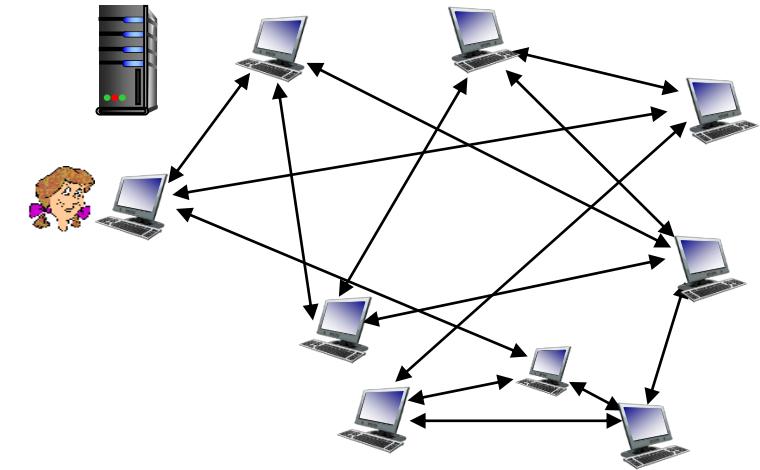
*tracker:* tracks peers  
participating in torrent



*torrent:* group of peers  
exchanging chunks of a file

# P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn*: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent



# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution: distributed, application-level infrastructure*



# Multimedia: video

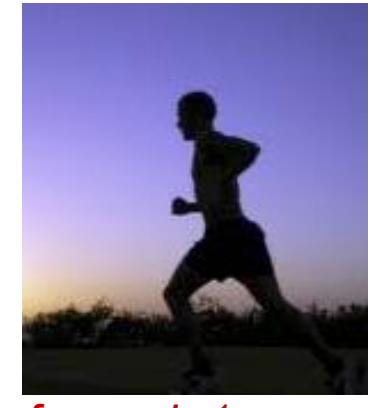
- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image 时~~间~~
  - spatial (within image) 空~~间~~
  - temporal (from one image to next)

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and *number of repeated values* ( $N$ )



frame  $i$

*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$

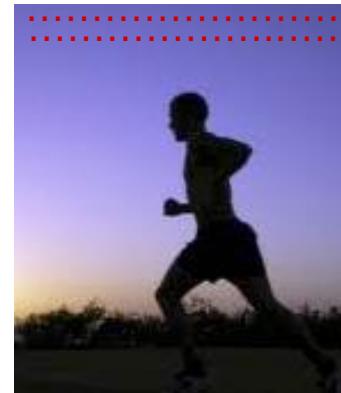


frame  $i+1$

# Multimedia: video

高 bit rate → 视频质量更好.

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples: 编码方式.
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps, e.g., mp4) mp4 运用上是 MPEG4.



frame *i*



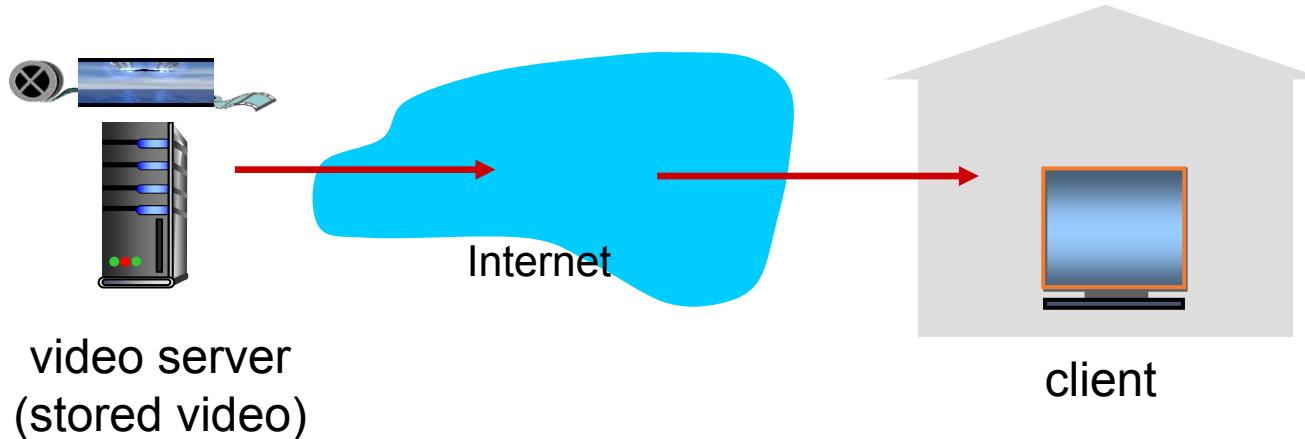
frame *i+1*

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and number of repeated values ( $N$ )

*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$

# Streaming stored video

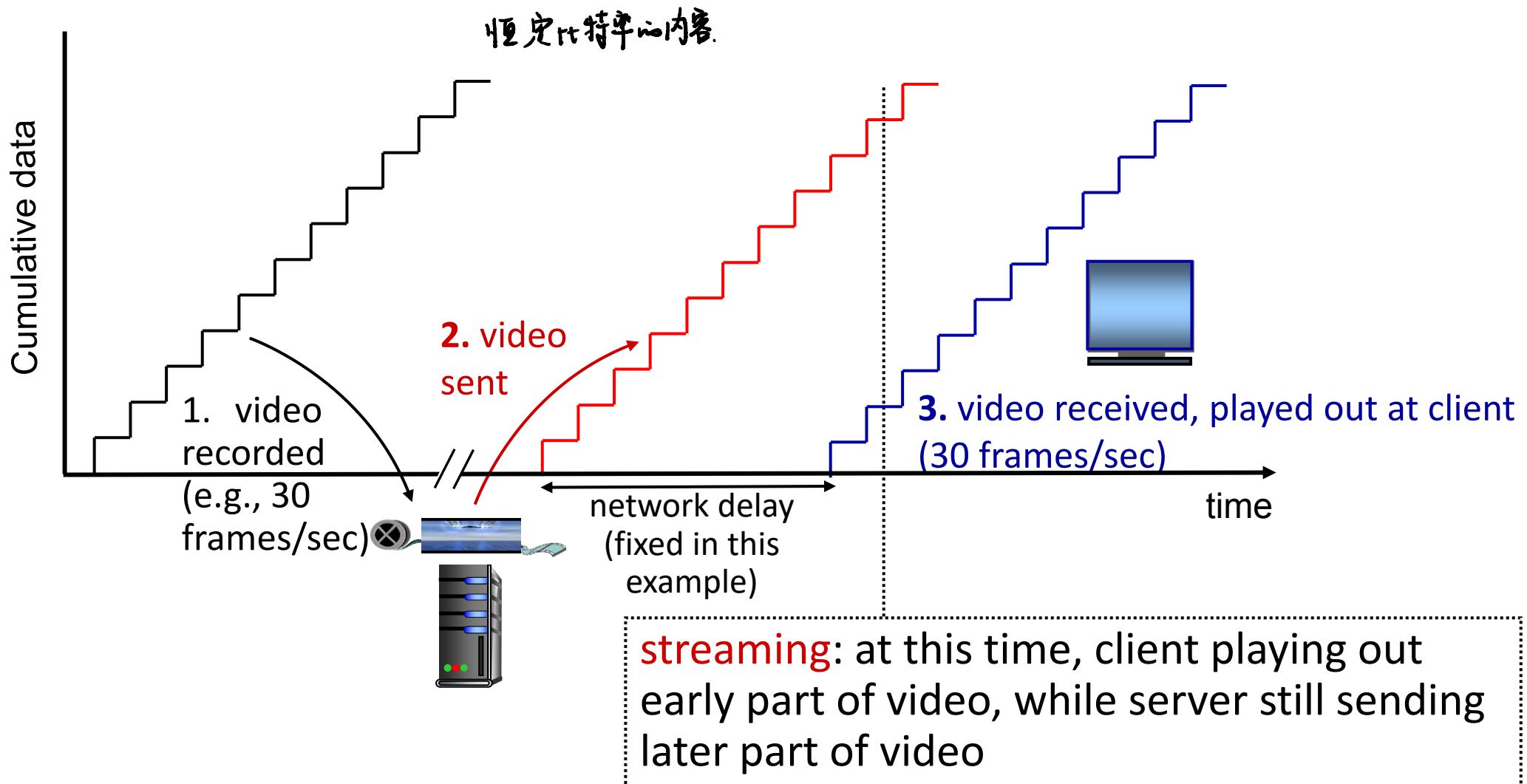
simple scenario:



Main challenges:

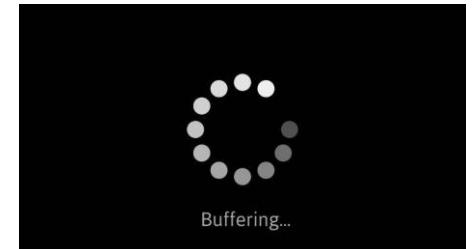
- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

# Streaming stored video

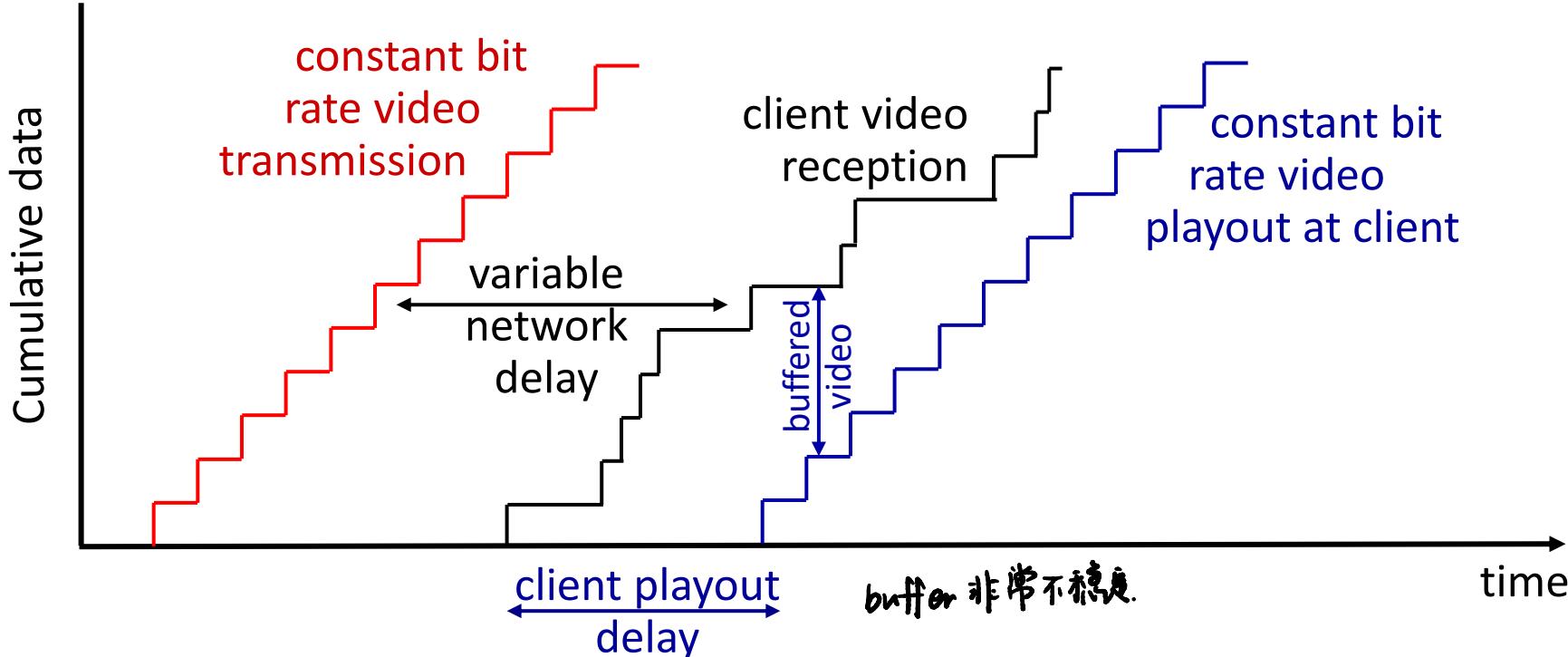


# Streaming stored video: challenges

- continuous playout constraint (连续播放限制):  
once client playout begins, playback must match original timing
  - ... but network delays are variable (jitter晃动), so will need client-side buffer to match playout requirements
- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted



# Streaming stored video: playout buffering



- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

# Streaming multimedia: DASH

- **DASH: Dynamic, Adaptive Streaming over HTTP**

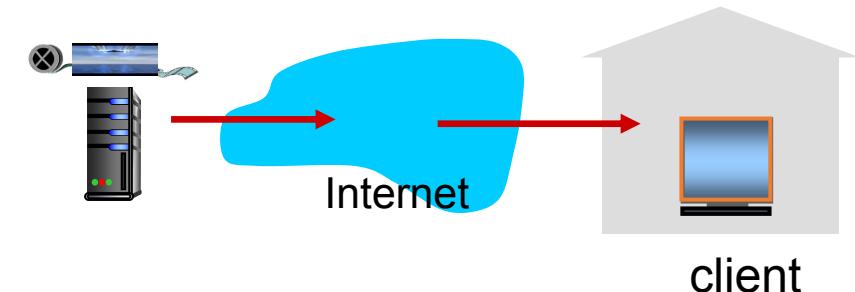
- **server:**

- divides video file into multiple chunks
- each chunk stored, encoded at different rates
- manifest file: provides URLs for different chunks

- **client:** 所有文件的比特率.

带宽测试.

- periodically measures server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate sustainable given current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time)

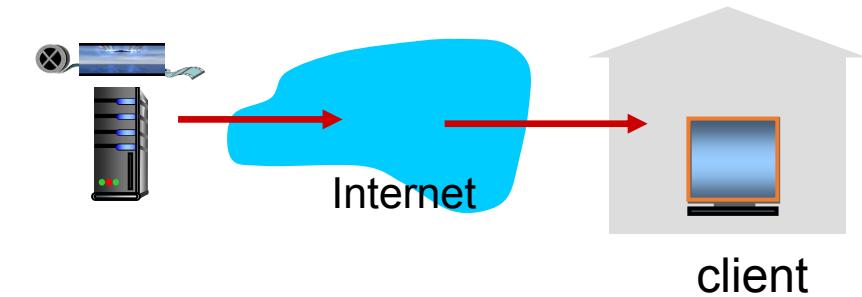


# Streaming multimedia: DASH

决策,请求都在客户端完成.

- “*intelligence*” at client: client determines

- *when* to request chunk (so that buffer starvation, or overflow does not occur)
- *what encoding rate* to request (higher quality when more bandwidth available)
- *where* to request chunk (can request from URL server that is “close” to client or has high available bandwidth)



Streaming video = encoding + DASH + playout buffering

三种技术同时实现.

# Content distribution networks (CDNs) 单服多点

- **challenge:** how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- **option 1:** single, large “mega-server”
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

....quite simply: this solution *doesn't scale*

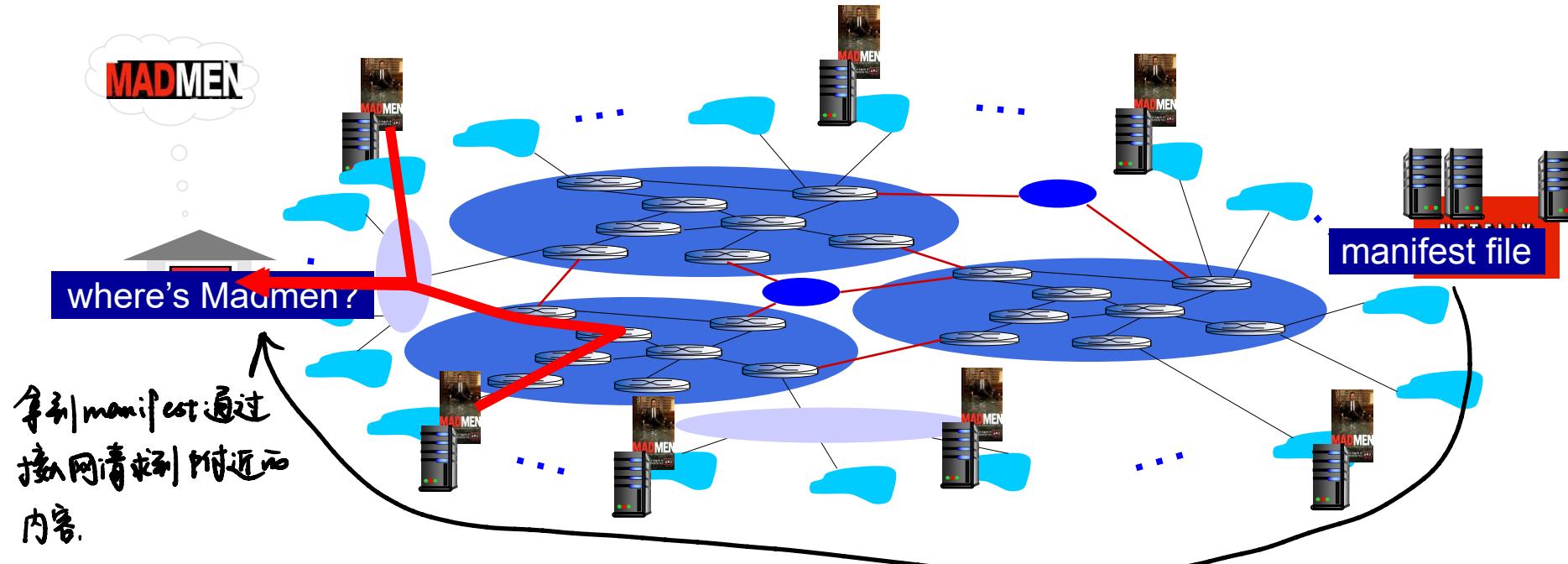
# Content distribution networks (CDNs)

- **challenge:** how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- **option 2:** store/serve multiple copies of videos at multiple geographically distributed sites (**CDN**)
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in more than 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in Points of Presence (POPs) near (but not within) access networks
    - used by Limelight



# Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



# Content distribution networks (CDNs)



*OTT challenges:* coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

DNS 酸解到不同服务器  
实现负载均衡

卫星信号/电视信号

# Application Layer: Overview

- Principles of network applications
  - Web and HTTP
  - E-mail, SMTP, IMAP
  - The Domain Name System DNS
- P2P applications
  - video streaming and content distribution networks
  - **socket programming with UDP and TCP**

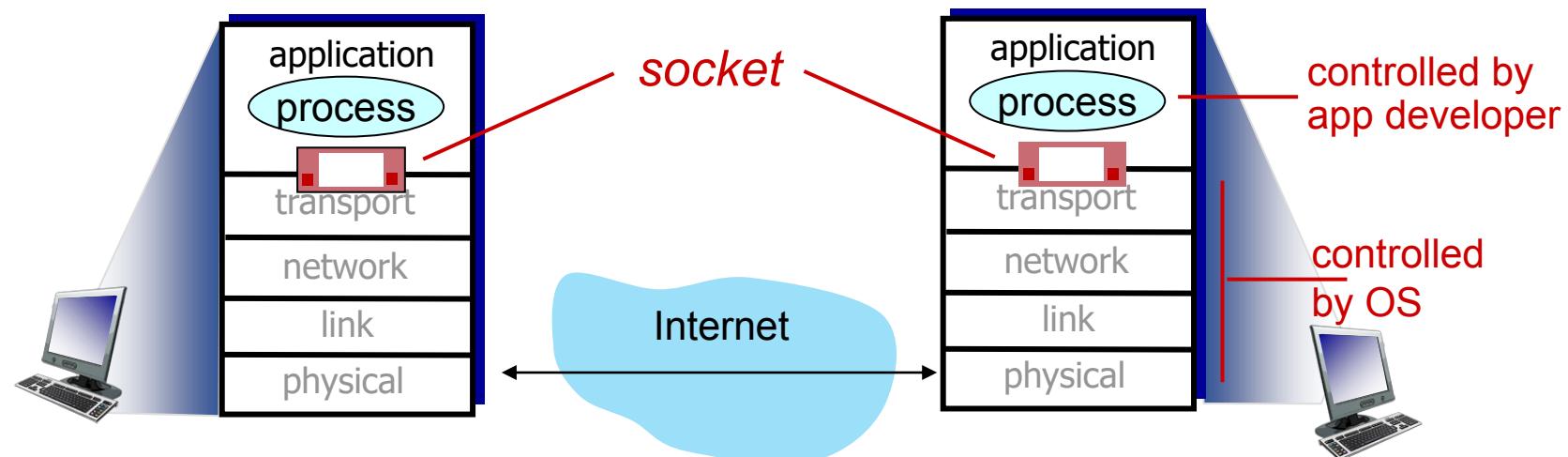


传输层

# Socket programming (套接字编程) <<sub>TCP</sub><sup>UDP</sup>

*goal:* learn how to build client/server applications that communicate using sockets

*socket:* like a courier between application process and end-end-transport protocol



# Socket programming

Two socket types for two transport services:

- **UDP**: unreliable datagram  
    **有明确定义的边界**  
    **不需要建立连接**.
- **TCP**: reliable, byte stream-oriented  
    **以 receive 和 send 形式存在**.

## Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

# Socket programming with UDP

**UDP:** no “connection” between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port # from received packet

**UDP:** transmitted data may be lost or received out-of-order

**Application viewpoint:**

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”数据报) between client and server

# Client/server socket interaction: UDP



**server** (running on serverIP)

```
create socket, port= x:  
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
read datagram from  
serverSocket
```

```
write reply to  
serverSocket  
specifying  
client address,  
port number
```



**client**

```
create socket:  
clientSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
Create datagram with server IP and  
port=x; send datagram via  
clientSocket
```

```
read datagram from  
clientSocket  
close  
clientSocket
```

# Example app: UDP client

## *Python UDPCClient*

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
create UDP socket for server → clientSocket = socket(AF_INET,
                                                    SOCK_DGRAM)
get user keyboard input → message = raw_input('Input lowercase sentence:')
attach server name, port to message; send into socket → clientSocket.sendto(message.encode(),
                           (serverName, serverPort))
read reply characters from socket into string → modifiedMessage, serverAddress =
                                               clientSocket.recvfrom(2048)
print out received string and close socket → print(modifiedMessage.decode())
                                               clientSocket.close()
```

# Example app: UDP server

## *Python UDPServer*

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(("", serverPort))
                                         print ("The server is ready to receive")
loop forever → while True:
Read from UDP socket into message, getting →   message, clientAddress = serverSocket.recvfrom(2048)
client's address (client IP and port)           modifiedMessage = message.decode().upper()
                                               serverSocket.sendto(modifiedMessage.encode(),
send upper case string back to this client →   clientAddress)
```

# Socket programming with TCP

## Client must contact server

- server process must first be running
- server must have created socket that welcomes client's contact

## Client contacts server by:

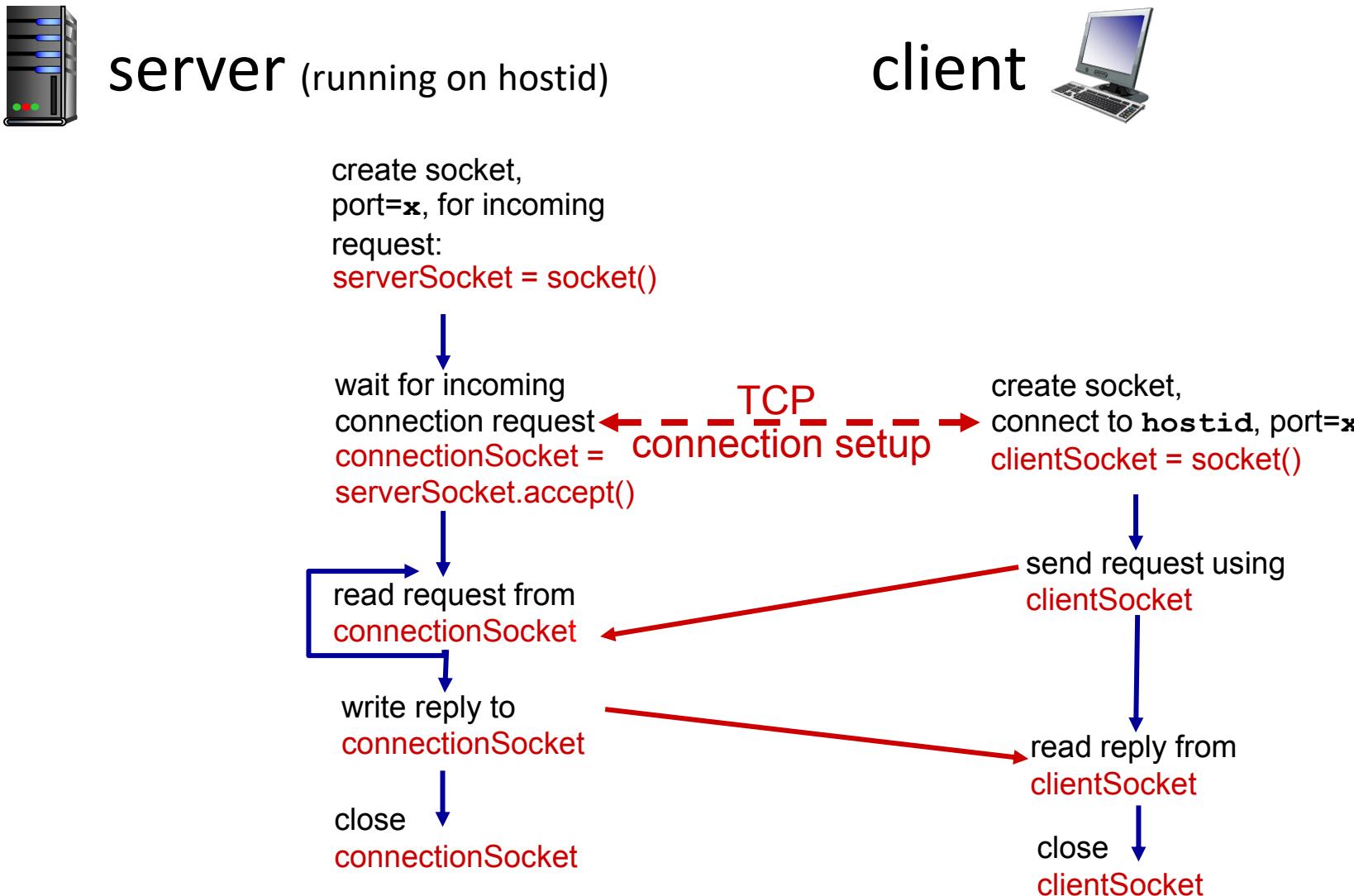
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

## Application viewpoint

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

# Client/server socket interaction: TCP



# Example app: TCP client

## *Python TCPClient*

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000 → clientSocket = socket(AF\_INET, SOCK\_STREAM)

No need to attach server name, port → clientSocket.connect((serverName, serverPort))

# Example app: TCP server

## *Python TCP Server*

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                           encode())
connectionSocket.close()
```

create TCP welcoming socket → from socket import \*

server begins listening for incoming TCP requests → serverPort = 12000  
→ serverSocket = socket(AF\_INET,SOCK\_STREAM)  
→ serverSocket.bind(("","serverPort"))  
→ serverSocket.listen(1)

loop forever → print 'The server is ready to receive'  
→ while True:

server waits on accept() for incoming requests, new socket created on return → connectionSocket, addr = serverSocket.accept()

read bytes from socket (but not address as in UDP) → sentence = connectionSocket.recv(1024).decode()  
→ capitalizedSentence = sentence.upper()  
→ connectionSocket.send(capitalizedSentence.  
 encode())

close connection to this client (but *not* welcoming socket) → connectionSocket.close()

# Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:
  - TCP, UDP sockets

# Chapter 2: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data*: info (payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”

*from lecture 6*

# Chapter 3: Transport Layer

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Transport layer: overview

*Our goal:*

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

# Transport layer: roadmap

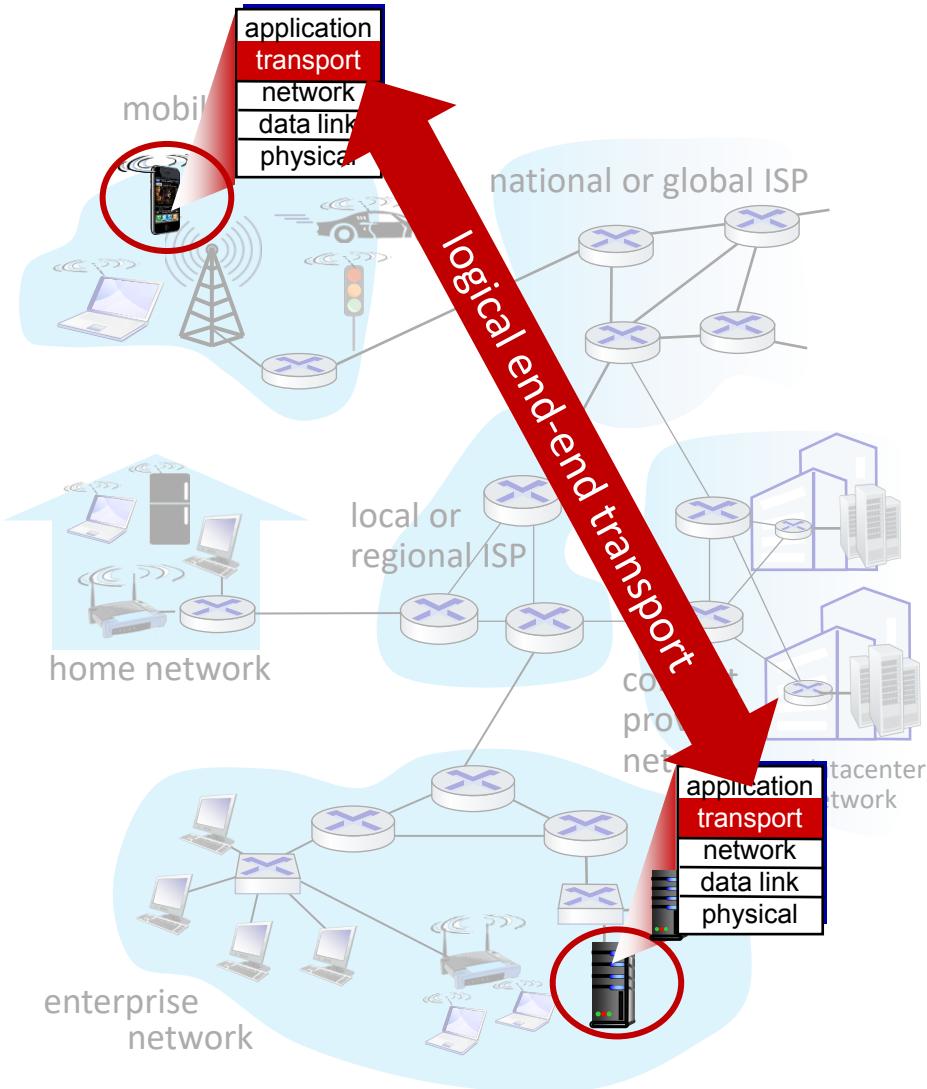
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control  
阻塞
- Evolution of transport-layer functionality



# Transport services and protocols

每个应用至少跑在一个进程上面。(虚拟的抽象连接)

- provide logical communication between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into segments, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



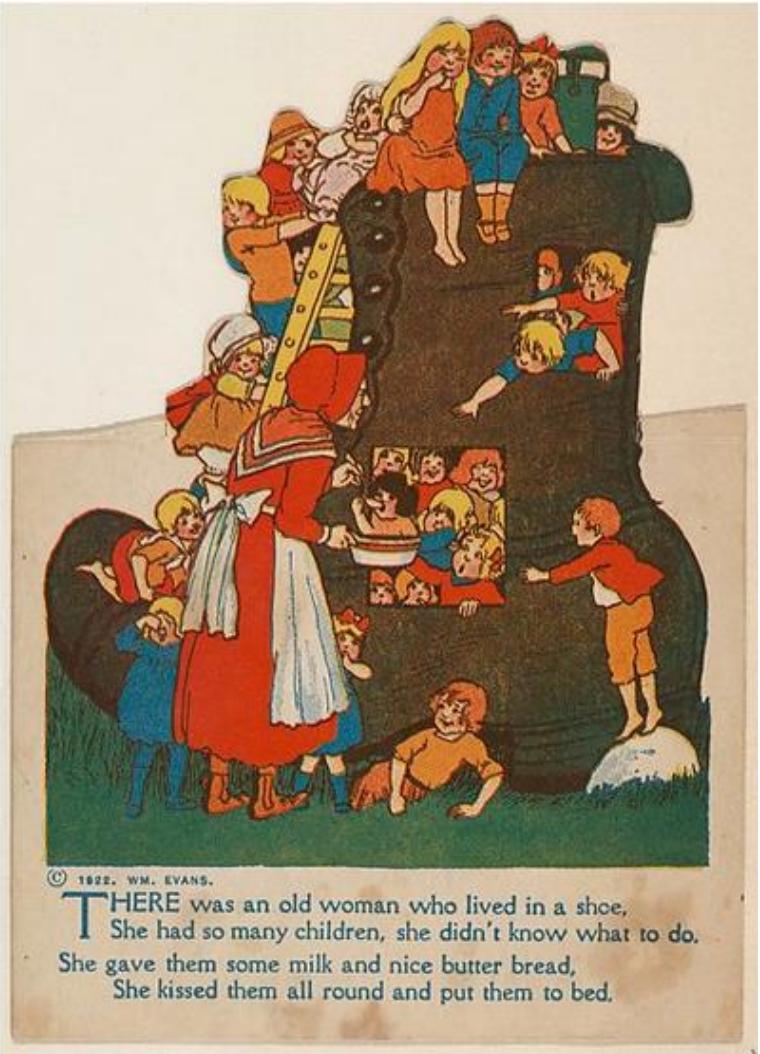
# Transport vs. network layer services and protocols

网络层：邮局：房子 → 房子：传输层：房子 → 小孩 应用

*household analogy:*

*12 kids in Ann's house  
sending letters to 12 kids in  
Bill's house:*

- hosts = houses
- processes = kids
- app messages = letters in envelopes



# Transport vs. network layer services and protocols

- **network layer:** logical communication between *hosts*
- **transport layer:** logical communication between *processes*
  - relies on, enhances, network layer services

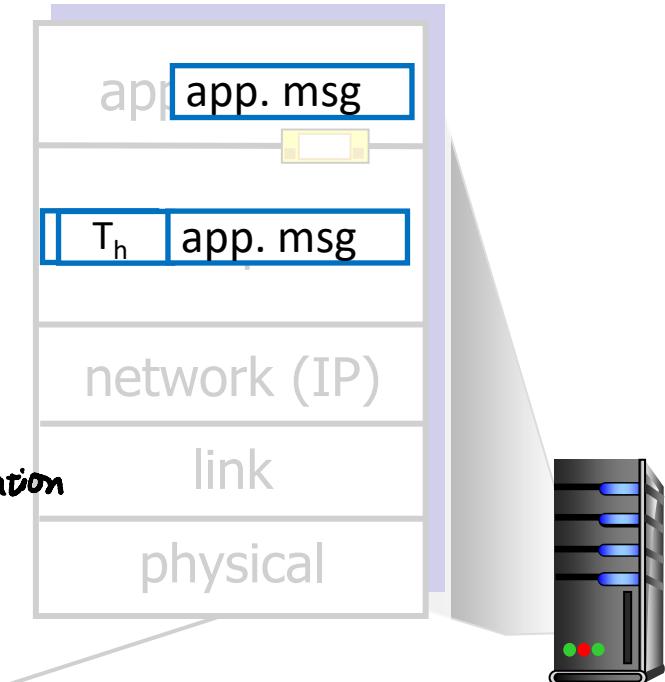
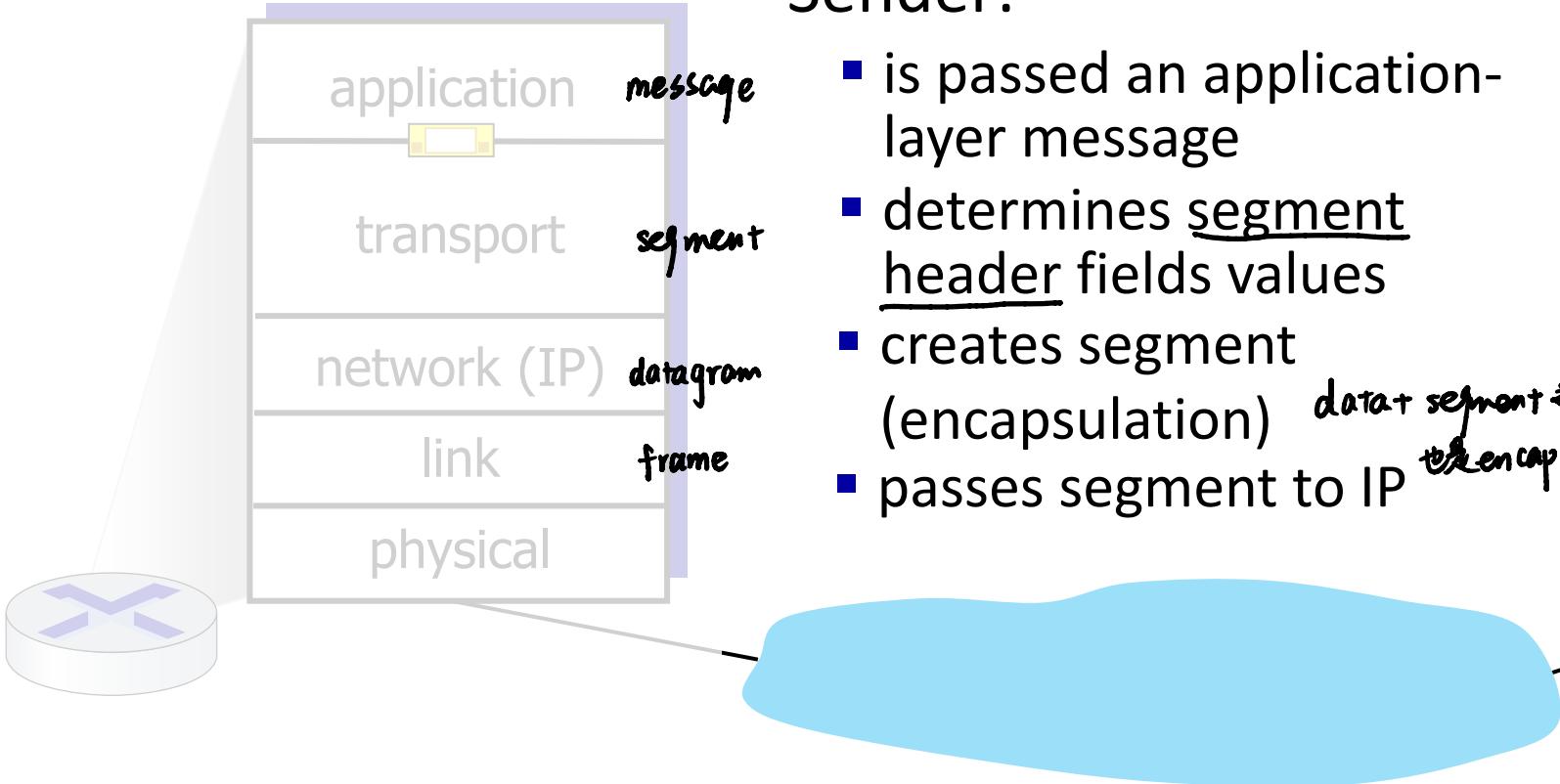
*household analogy:*

- 12 kids in Ann's house sending letters to 12 kids in Bill's house:*
- hosts = houses
  - processes = kids
  - app messages = letters in envelopes
  - transport protocol = Ann and Bill who demux to in-house siblings
  - network-layer protocol = postal service

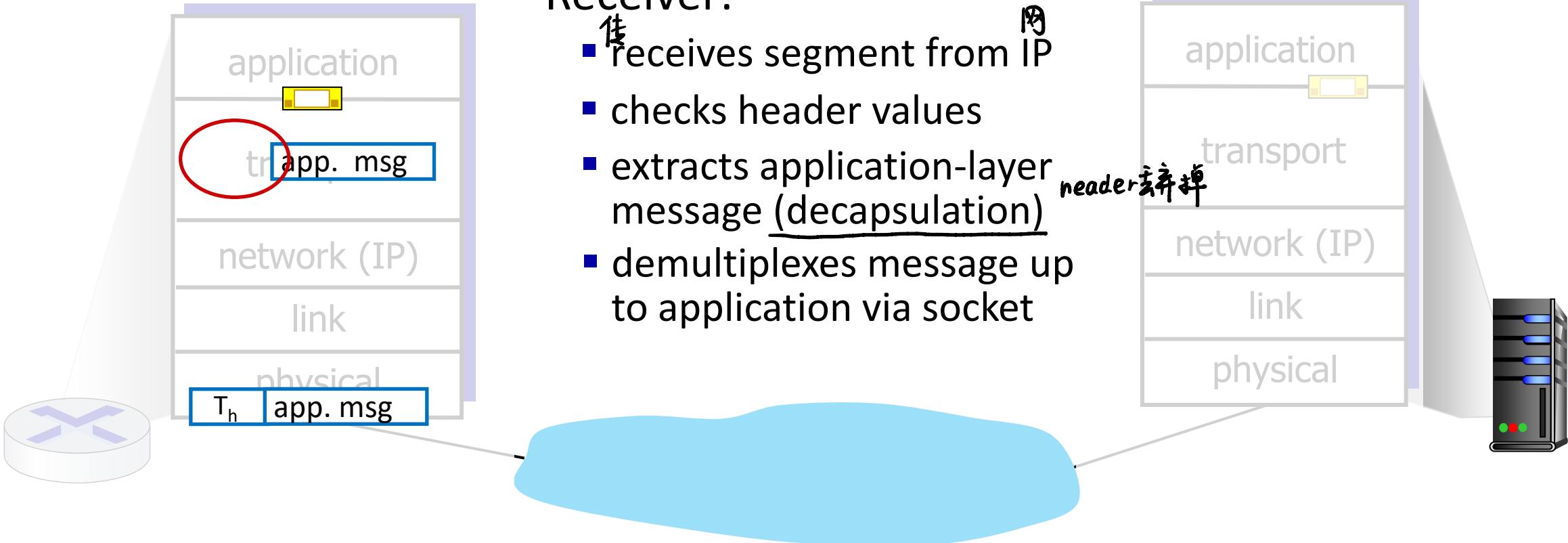
# Transport Layer Actions

Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment (encapsulation) *data + segment info*
- passes segment to IP *to encapsulation*



# Transport Layer Actions



# Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol

- reliable, in-order delivery
- congestion control
- flow control
- connection setup

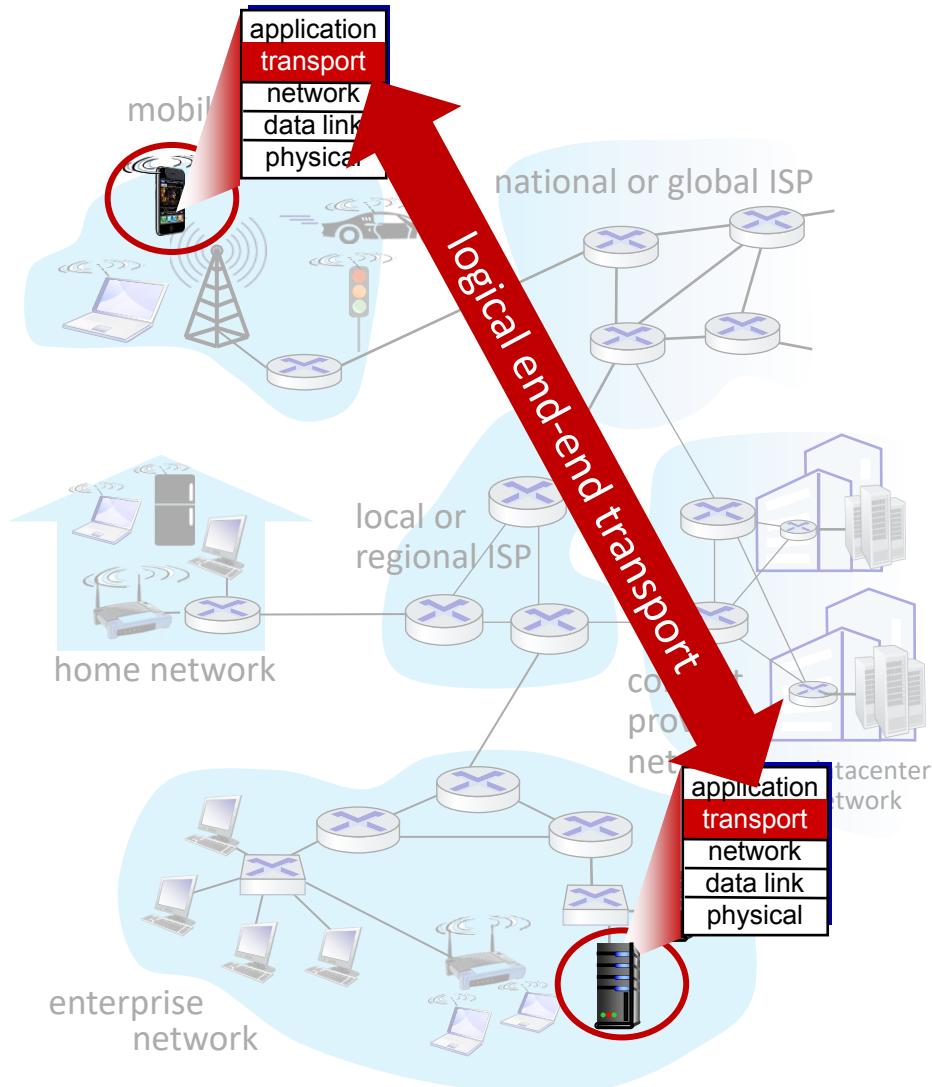
- **UDP:** User Datagram Protocol

- unreliable, unordered delivery
- no-frills extension of “best-effort” IP

- services not available:

- delay guarantees
- bandwidth guarantees

整个网络层决定



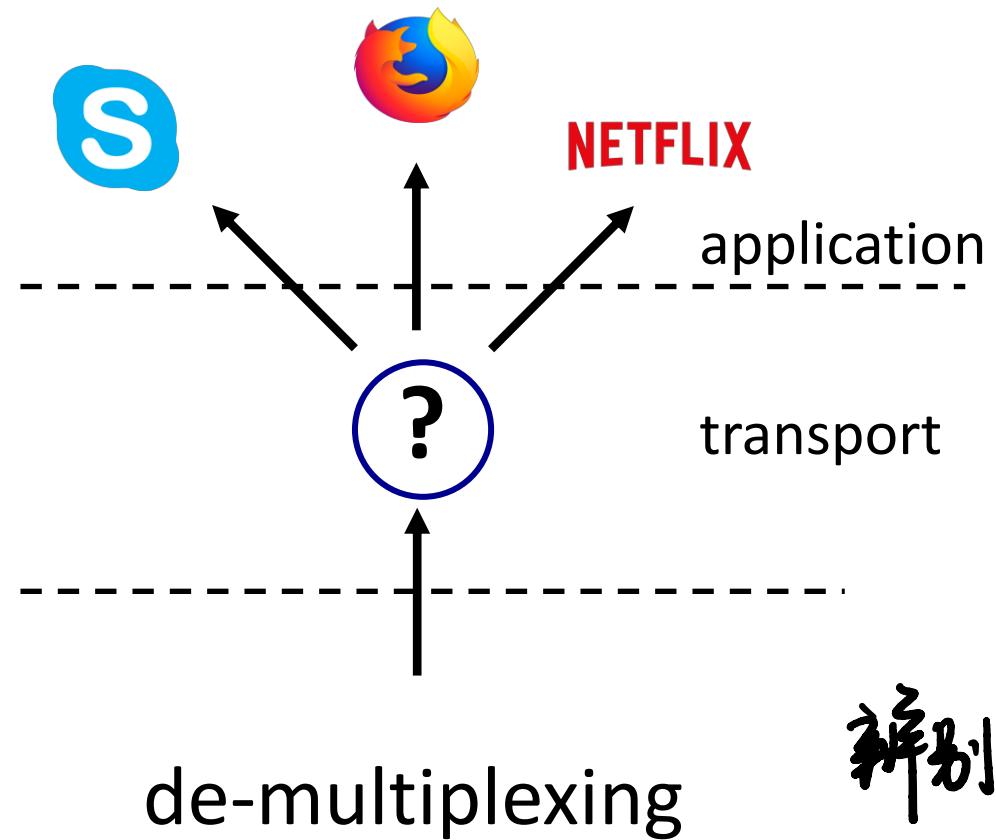
# Chapter 3: roadmap

transfer layer 对应 socket → application  
消息发给哪个应用

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

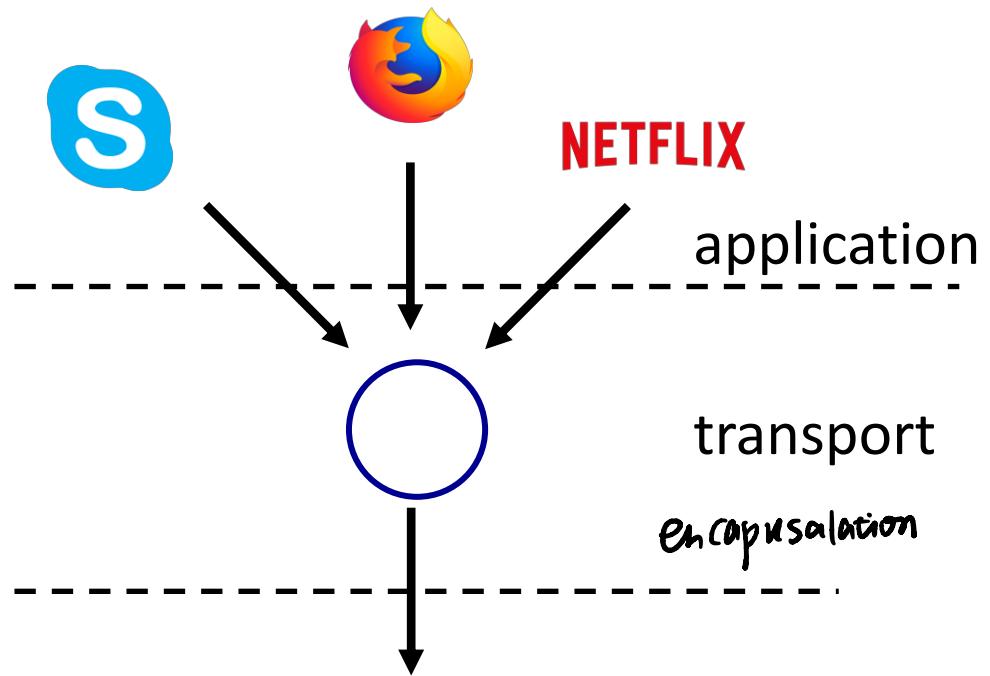
transfer layer 负责建立连接





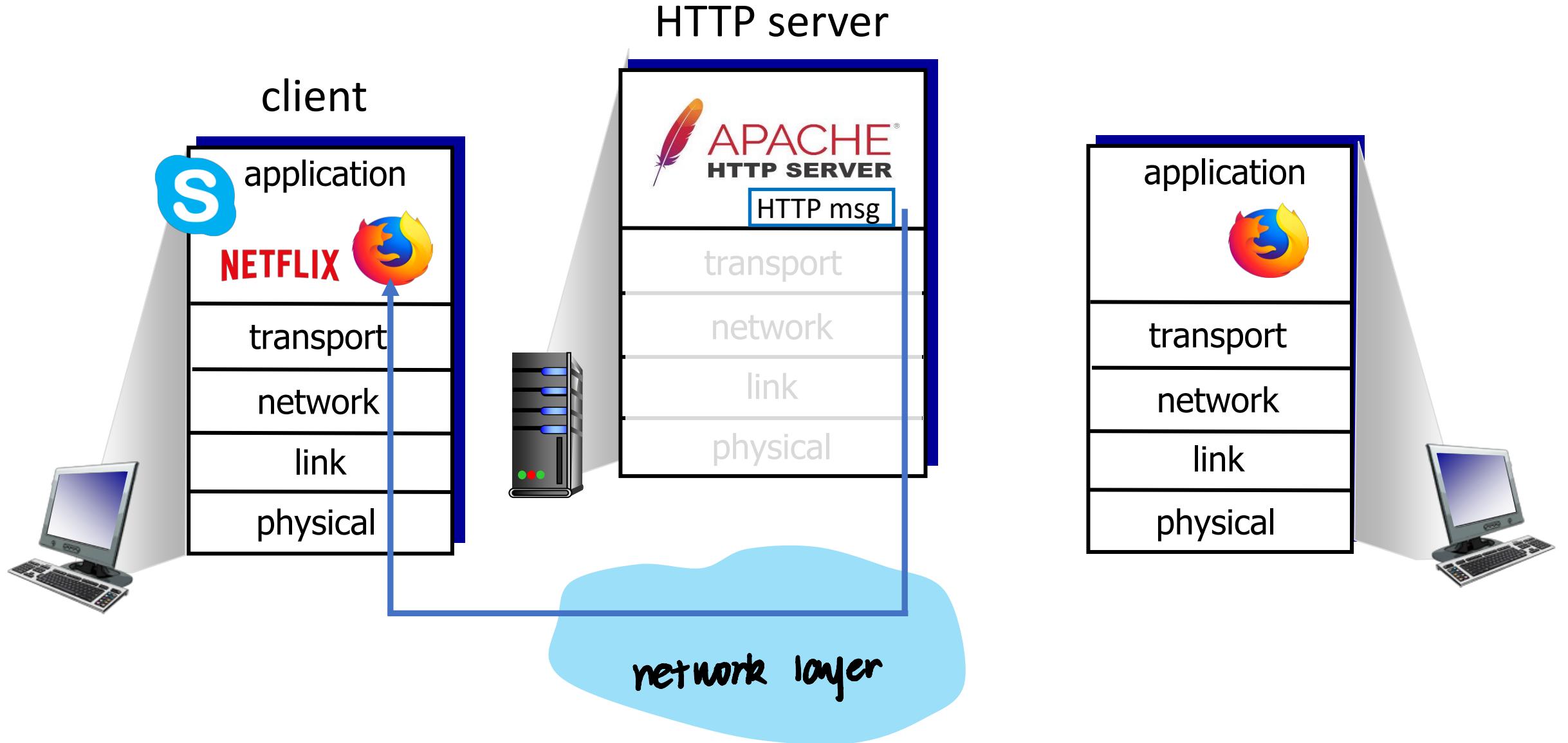
仅端口号 → 区分 socket ?

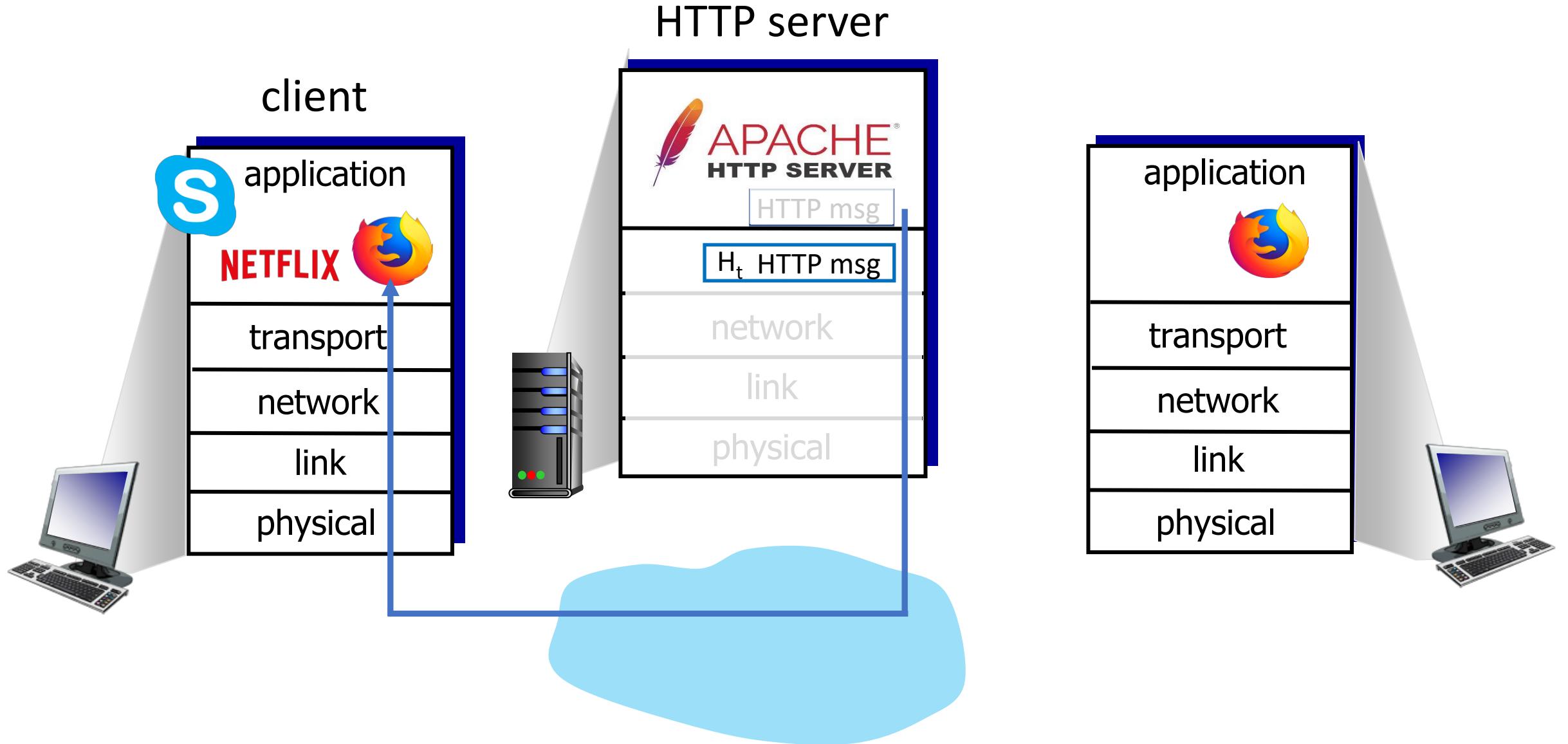
辨别给哪个应用

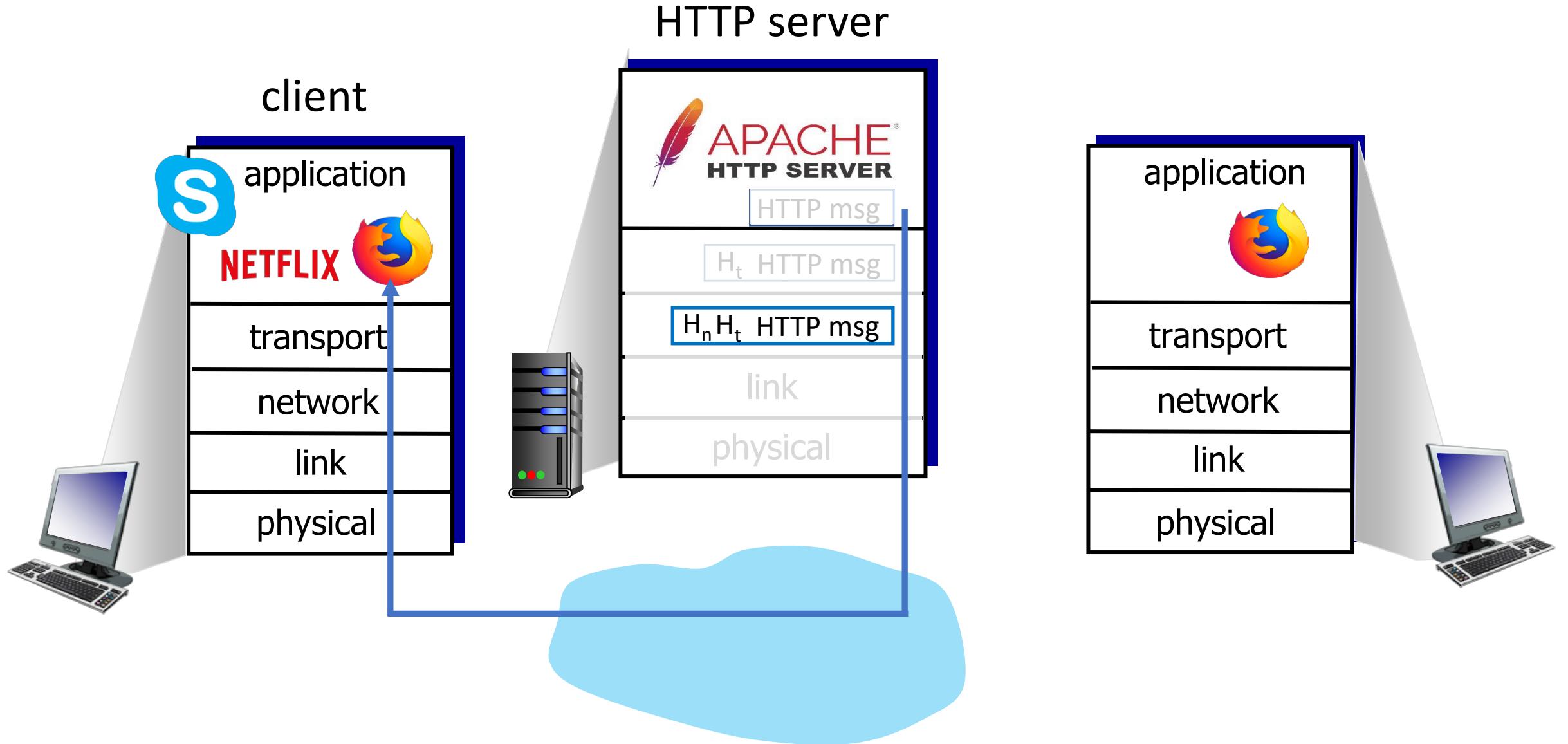


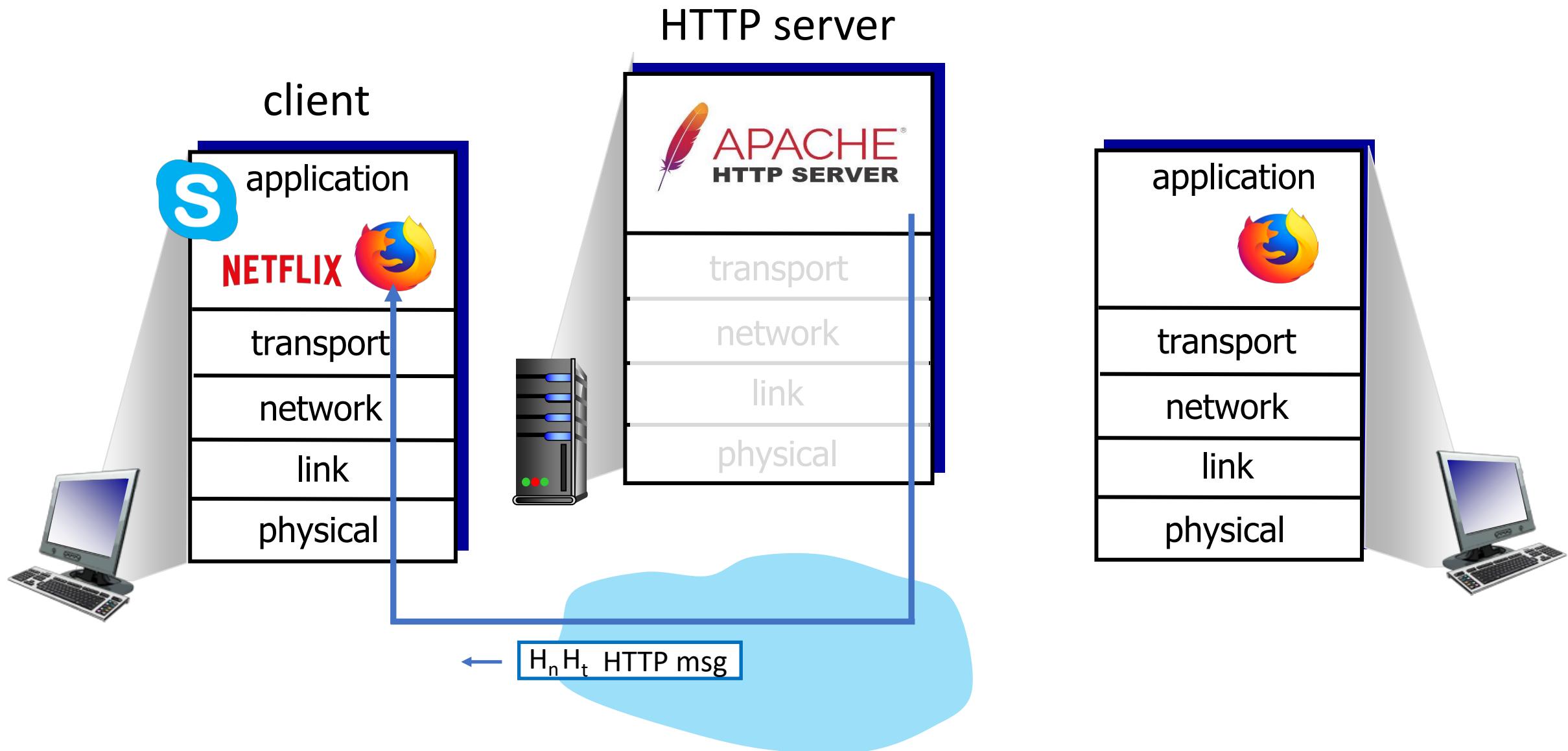
multiplexing 多路复用

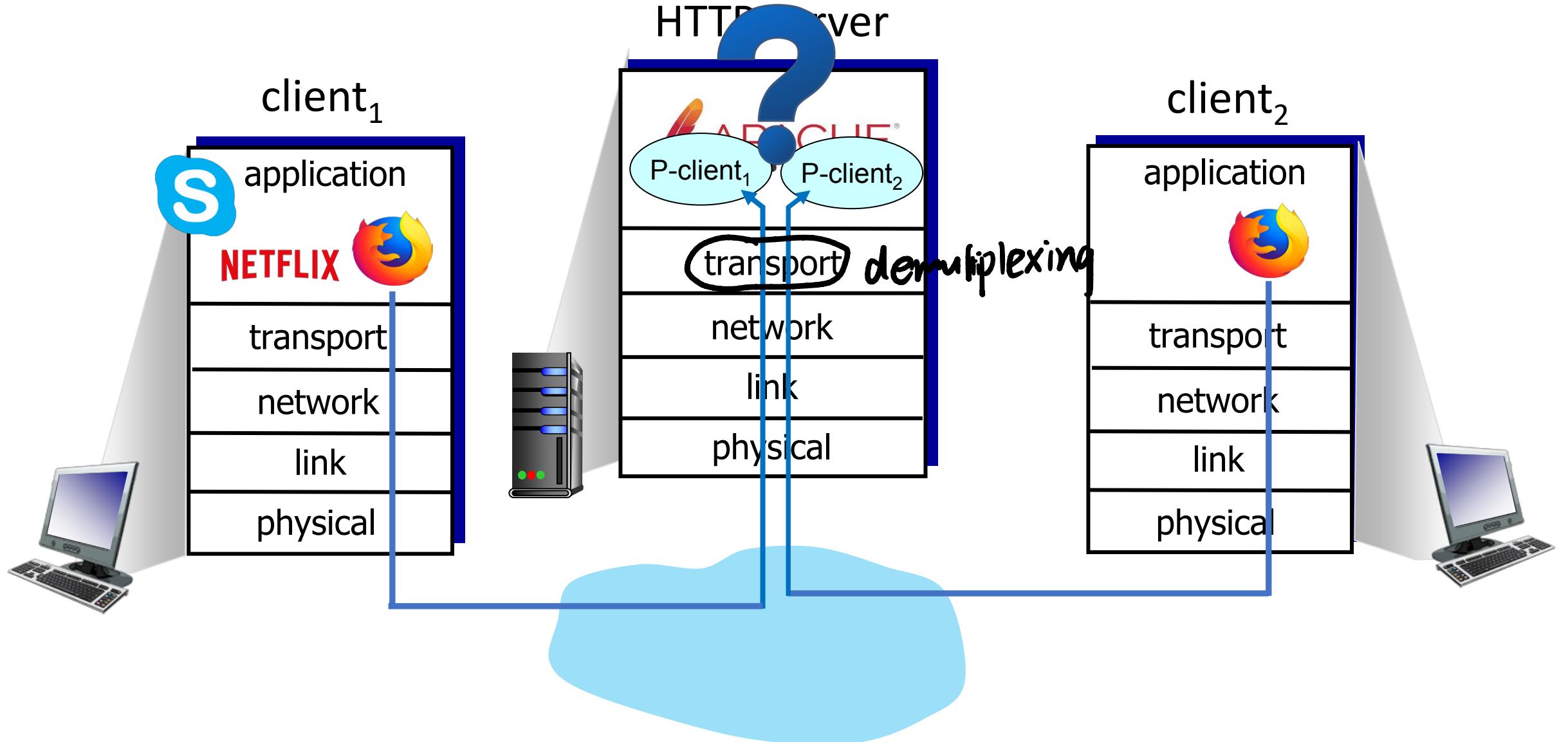
加header之后知道如何分配.

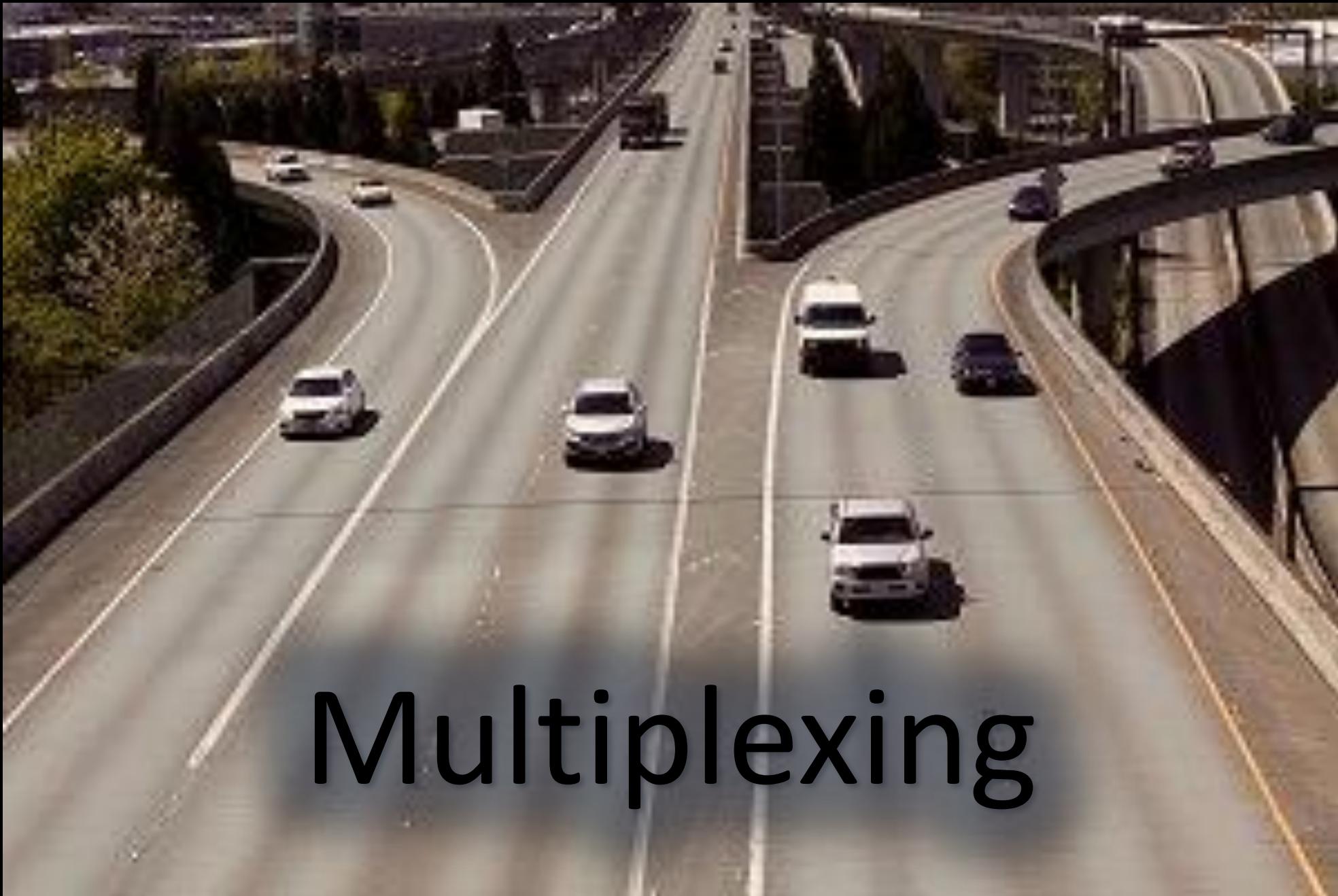












# Multiplexing



# Demultiplexing

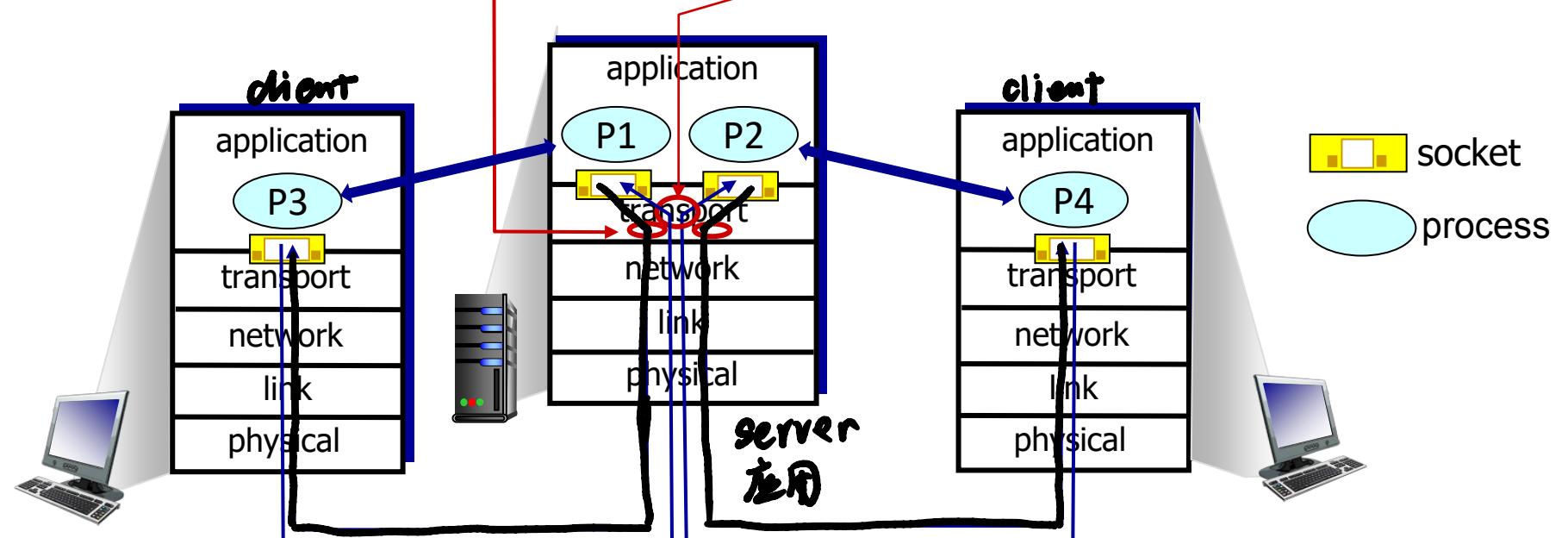
# Multiplexing/demultiplexing

*multiplexing at sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

*demultiplexing at receiver:*

use header info to deliver received segments to correct socket



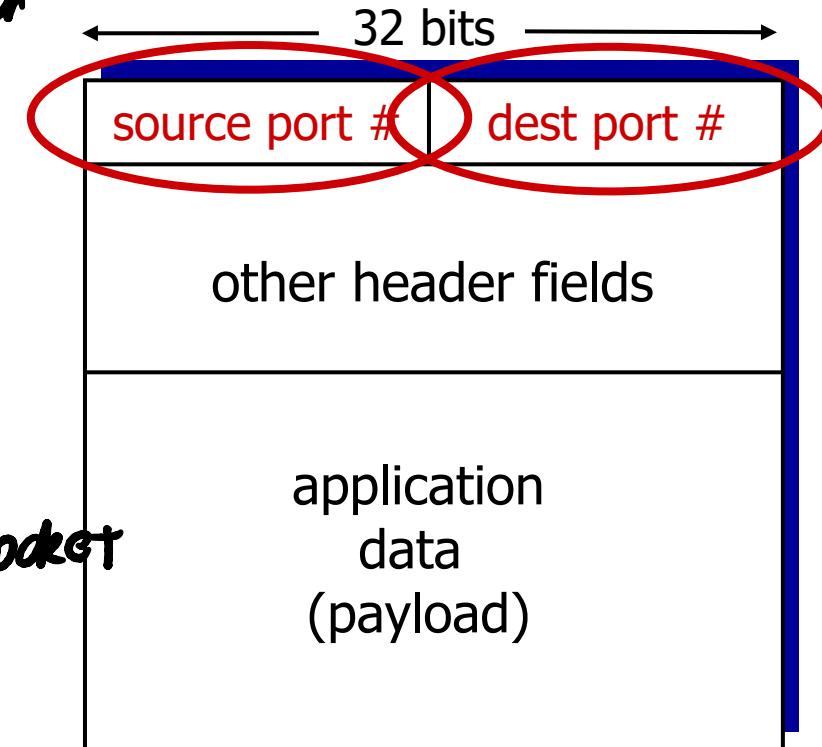
# How demultiplexing works

- host receives IP datagrams *network layer*

- each datagram has source IP address, destination IP address
- each datagram carries one transport-layer segment
- each segment has source, destination port number

- host uses IP addresses & port numbers to direct segment to appropriate socket

組合進行的 socket



TCP/UDP segment format

# Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

```
mySocket =  
    socket (AF_INET, SOCK_DGRAM)  
mySocket.bind ("", 9157);  
    空字符串表示端口。
```

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

*send-to (ip, port)*

when receiving host receives  
**UDP segment**:

- checks destination port # in segment
- directs UDP segment to socket with that port #

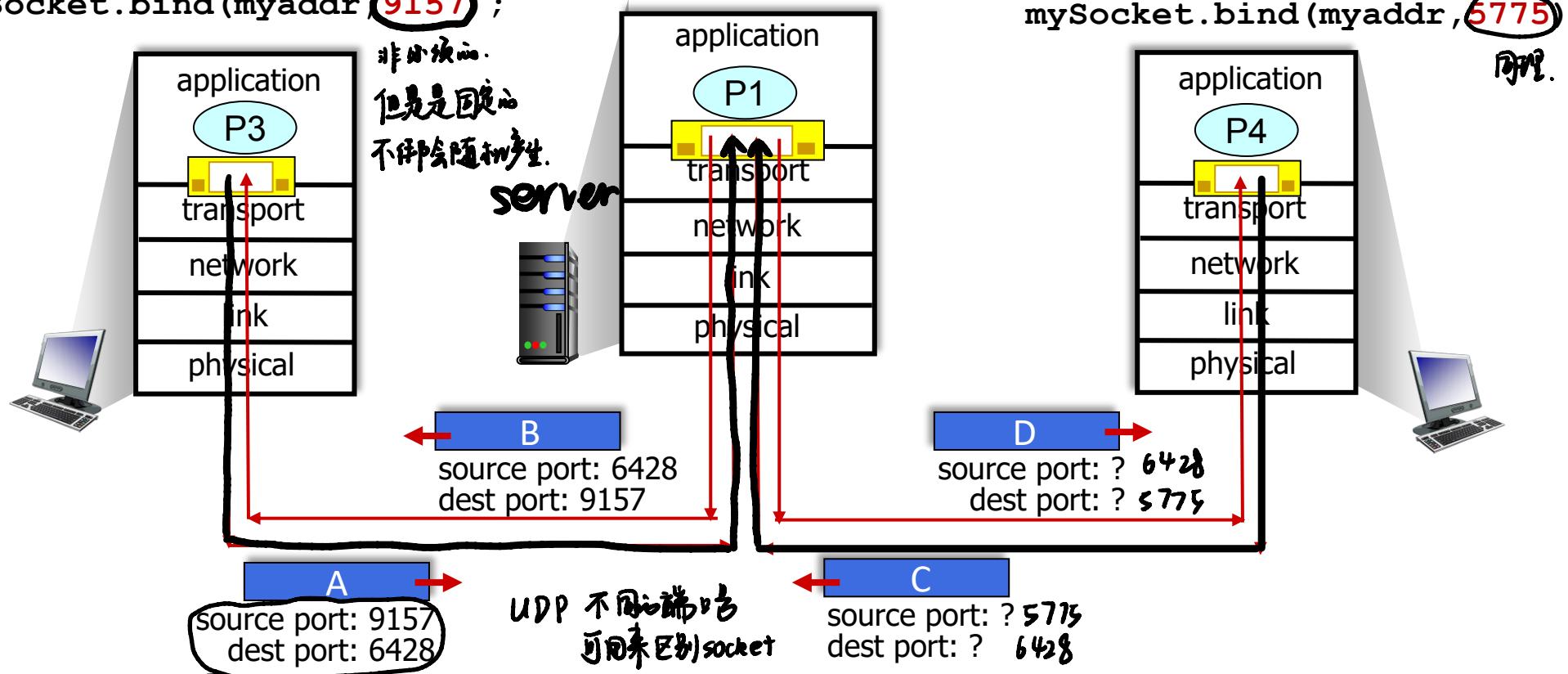


IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

# Connectionless demultiplexing: an example

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 9157);
```

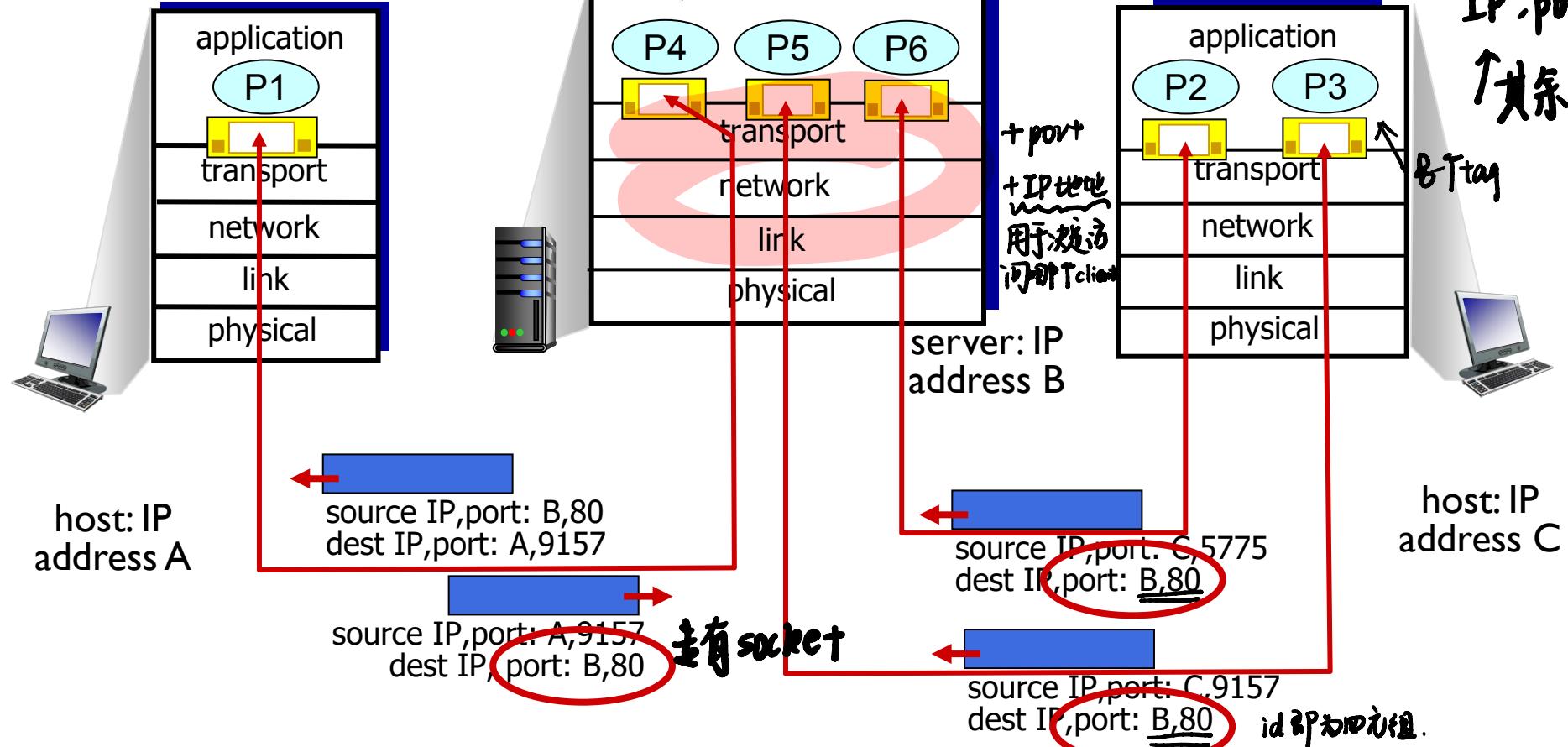


# Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket
- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

# Connection-oriented demultiplexing: example

多路复用, 多路分离



Three segments, all destined to IP address: B,  
dest port: 80 are demultiplexed to *different* sockets

# Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers

# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- **Connectionless transport: UDP**
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



# UDP: User Datagram Protocol

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:  
  - lost
  - delivered out-of-order to app
- *connectionless*: 直接去过来, 不管是否会回去或  
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

## Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

# UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP 网管.
  - HTTP/3 UDP 作用
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

# UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD  
J. Postel  
ISI  
28 August 1980

**User Datagram Protocol**

**Introduction**

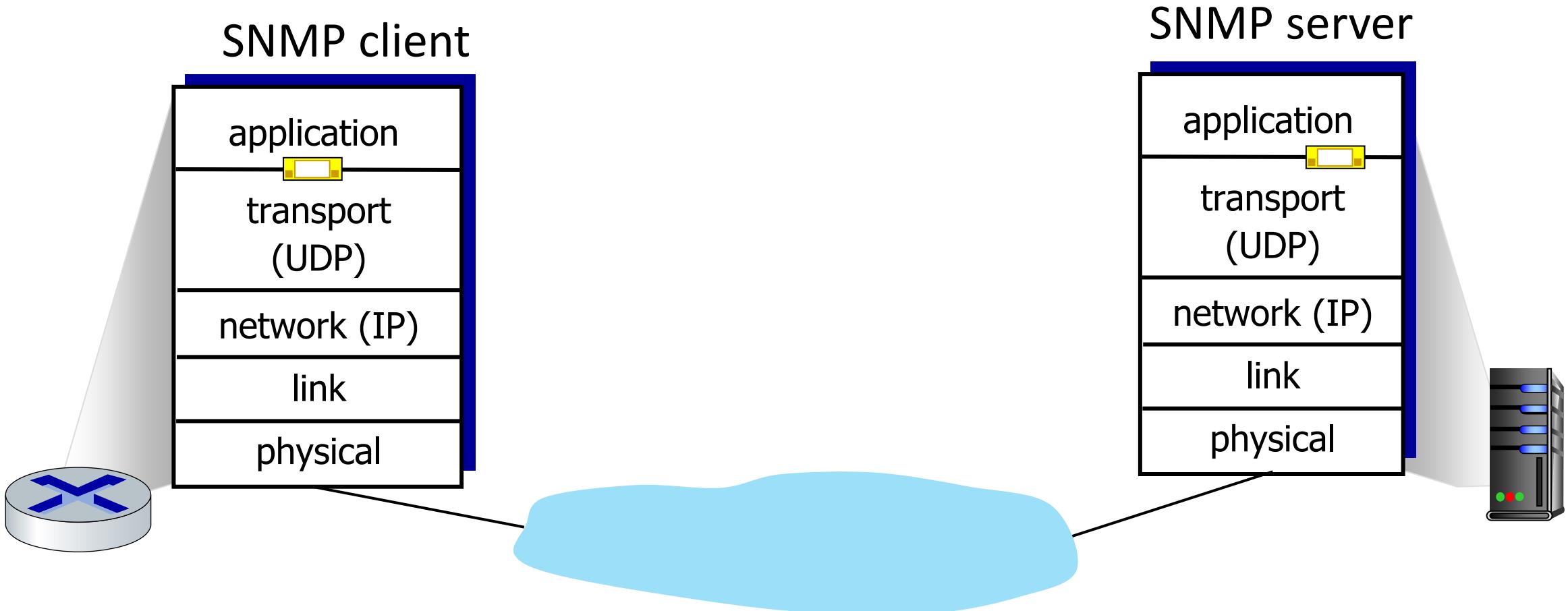
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

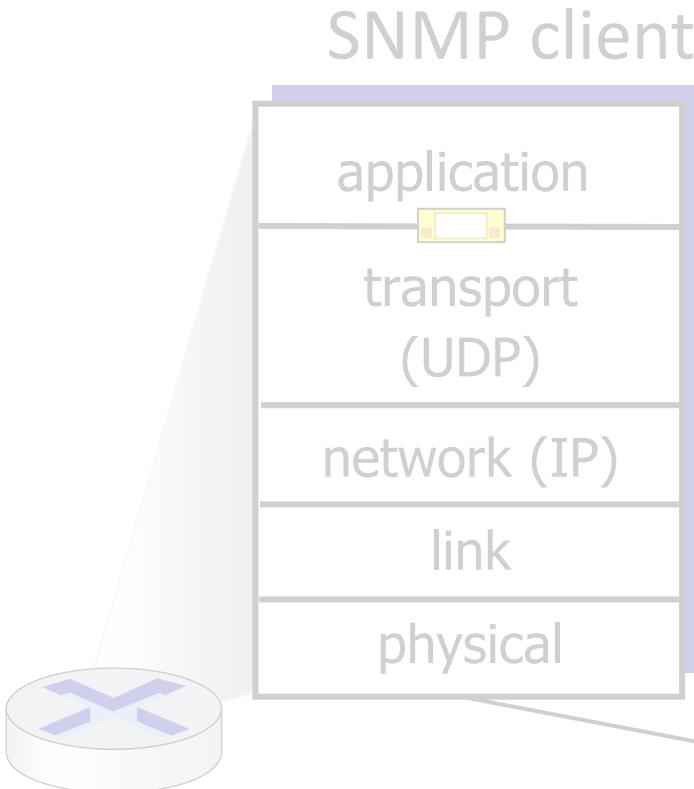
**Format**

0	7 8	15 16	23 24	31
Source Port		Destination Port		
Length		Checksum		
data octets ...				

# UDP: Transport Layer Actions



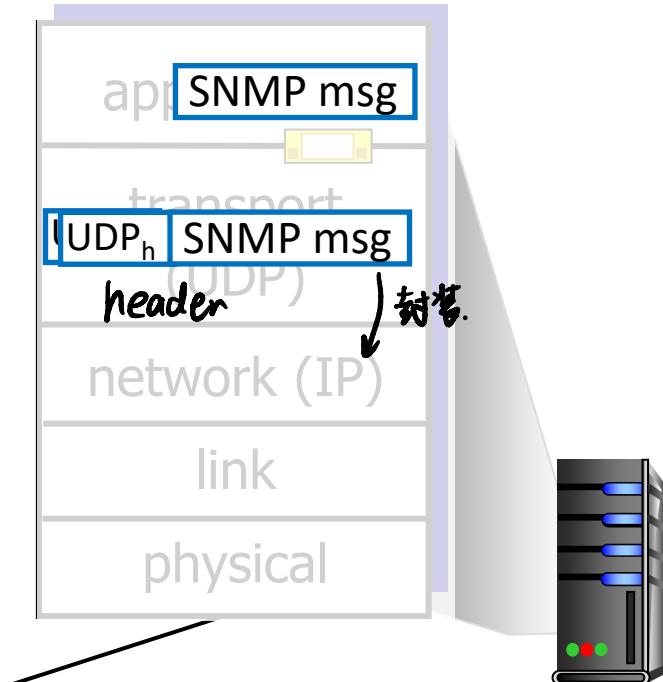
# UDP: Transport Layer Actions



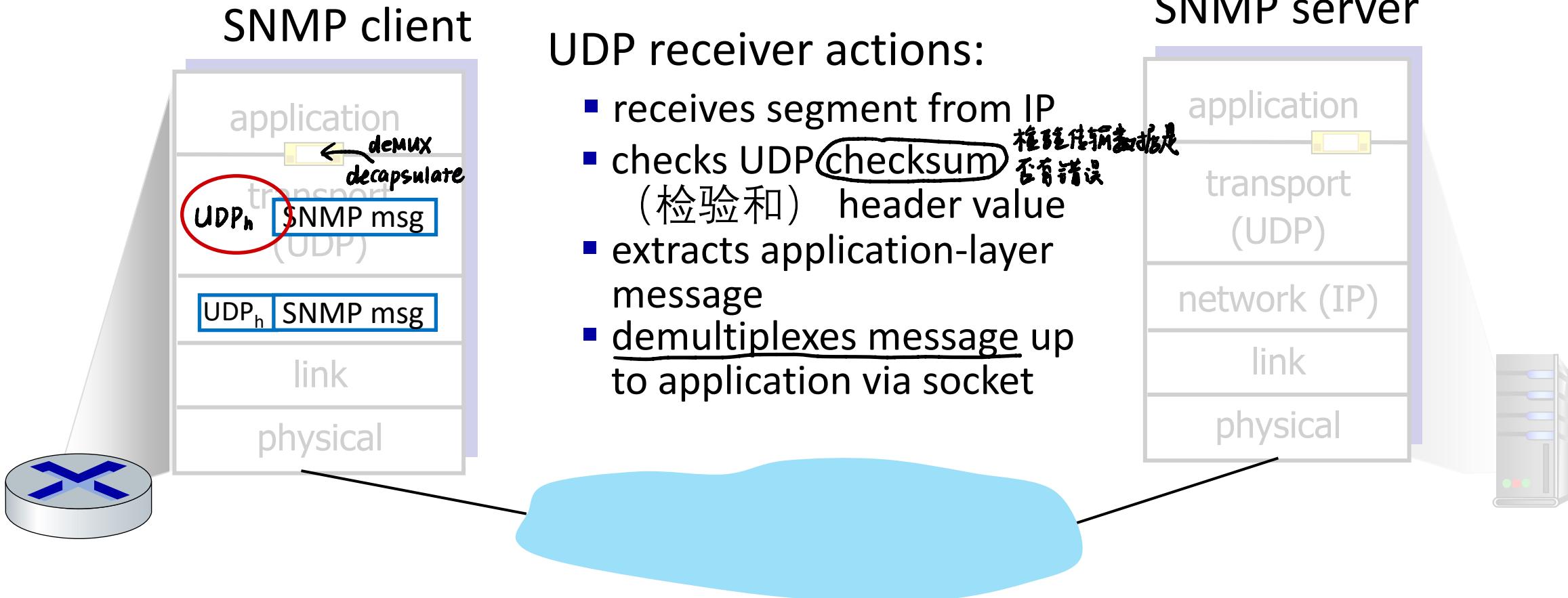
## UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

## SNMP server

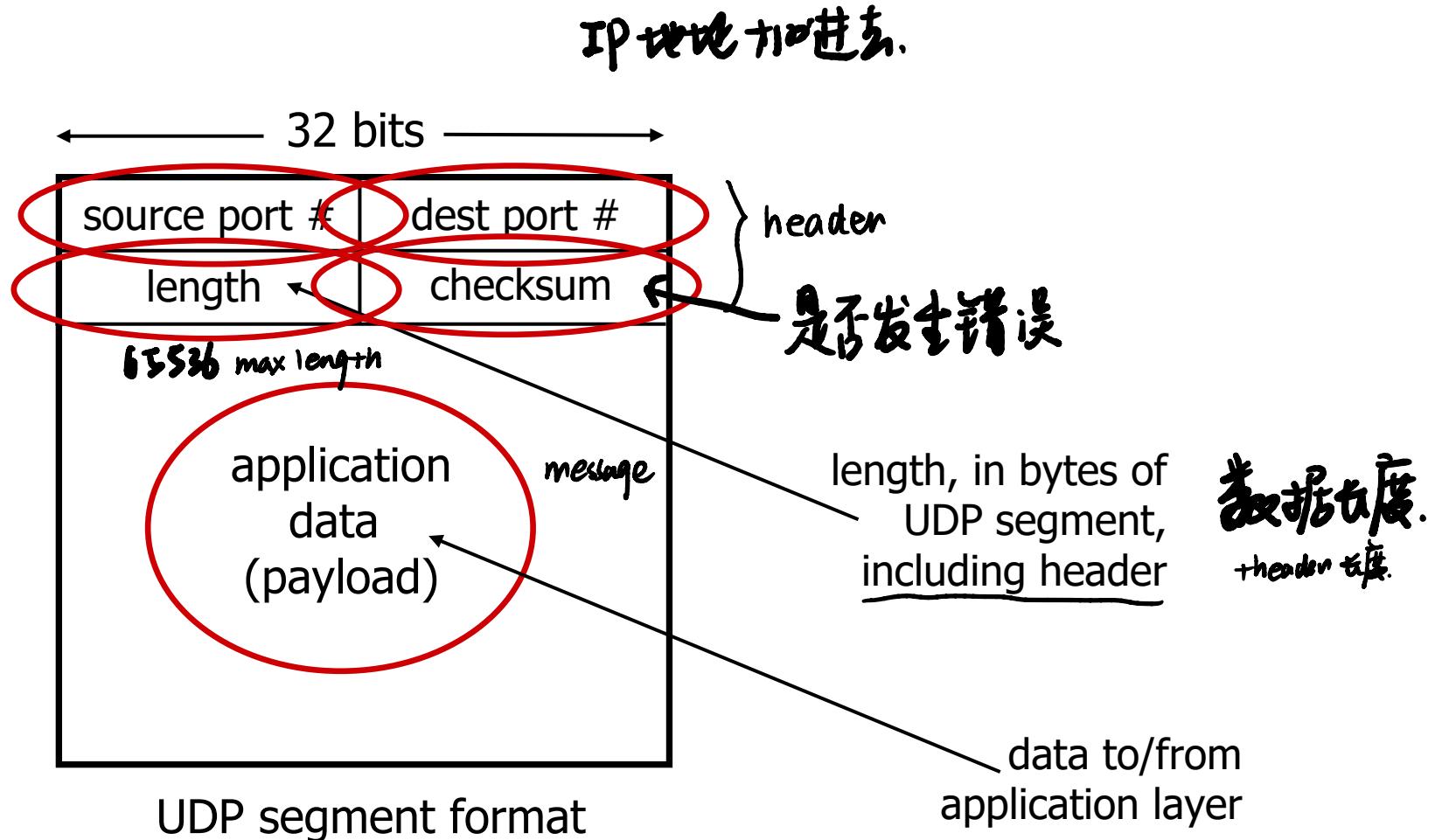


# UDP: Transport Layer Actions



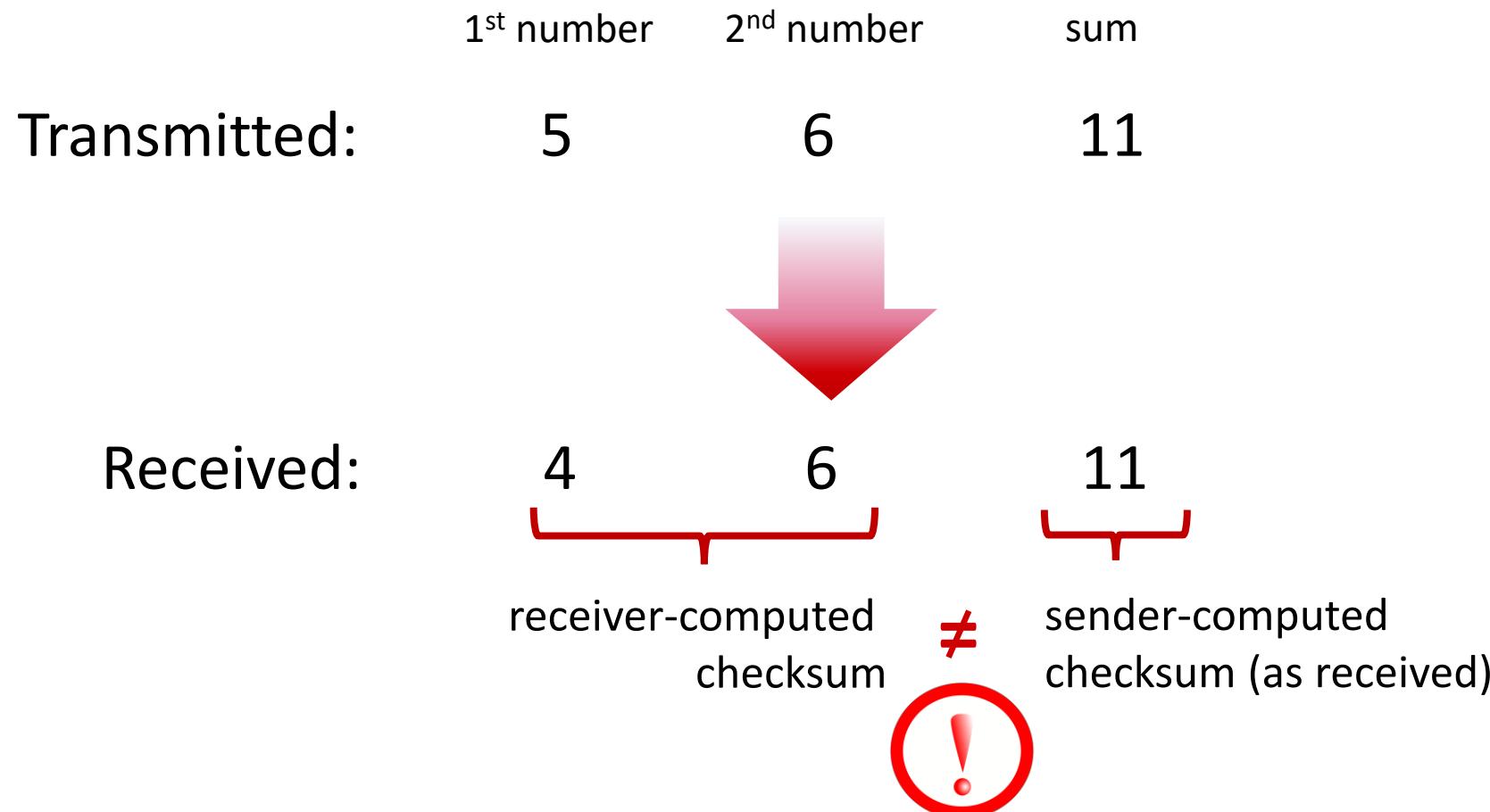
# UDP segment header

UDP header is an 8-bytes fixed and simple header



# UDP checksum (检验和)

**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment



# Internet checksum

**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment

sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless?* More later ....

# Internet checksum: an example

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1
<hr/>																

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

# Internet checksum: weak protection!

example: add two 16-bit integers

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0	0 1
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	1 0
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1	
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0	
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1	

Even though numbers have changed (bit flips), **no** change in checksum!

与其它所有 no sum 相加之后为全 1 (16位)

compare it 效率. (每位进行比较)  
全 1 很快

# Summary: UDP

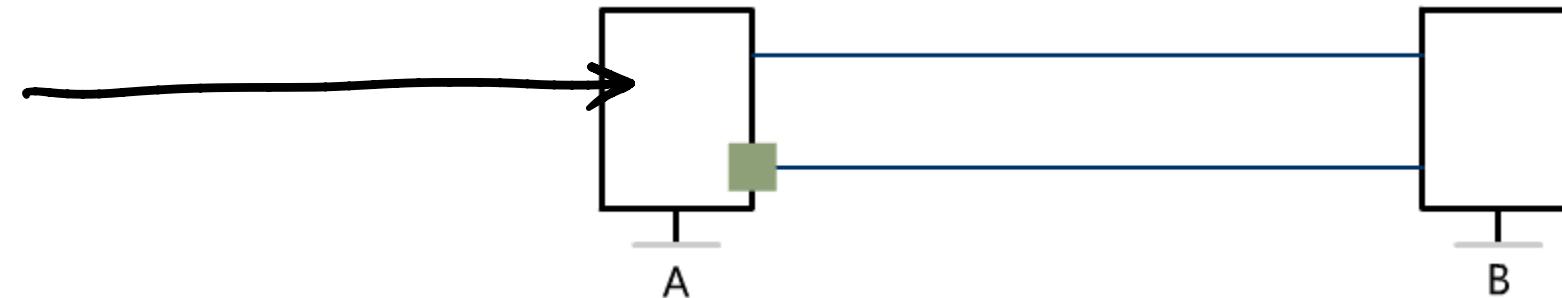
- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

TCP ★

# What if we want reliability?

- The network layer provides only **best effort** service with no guarantee

- Data loss



- Packets out of order



- Flip bits *check sum*

package corruption 其中数据发生问题

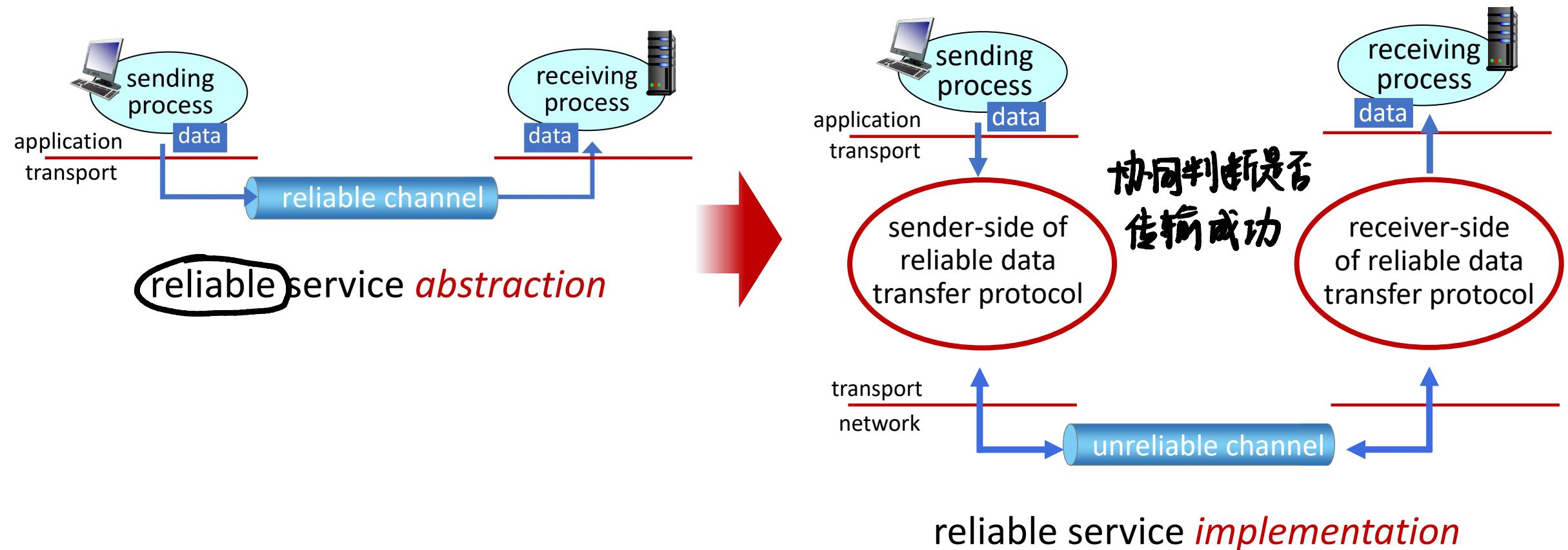
# Chapter 3: roadmap



- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**  
*most important in this class*
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

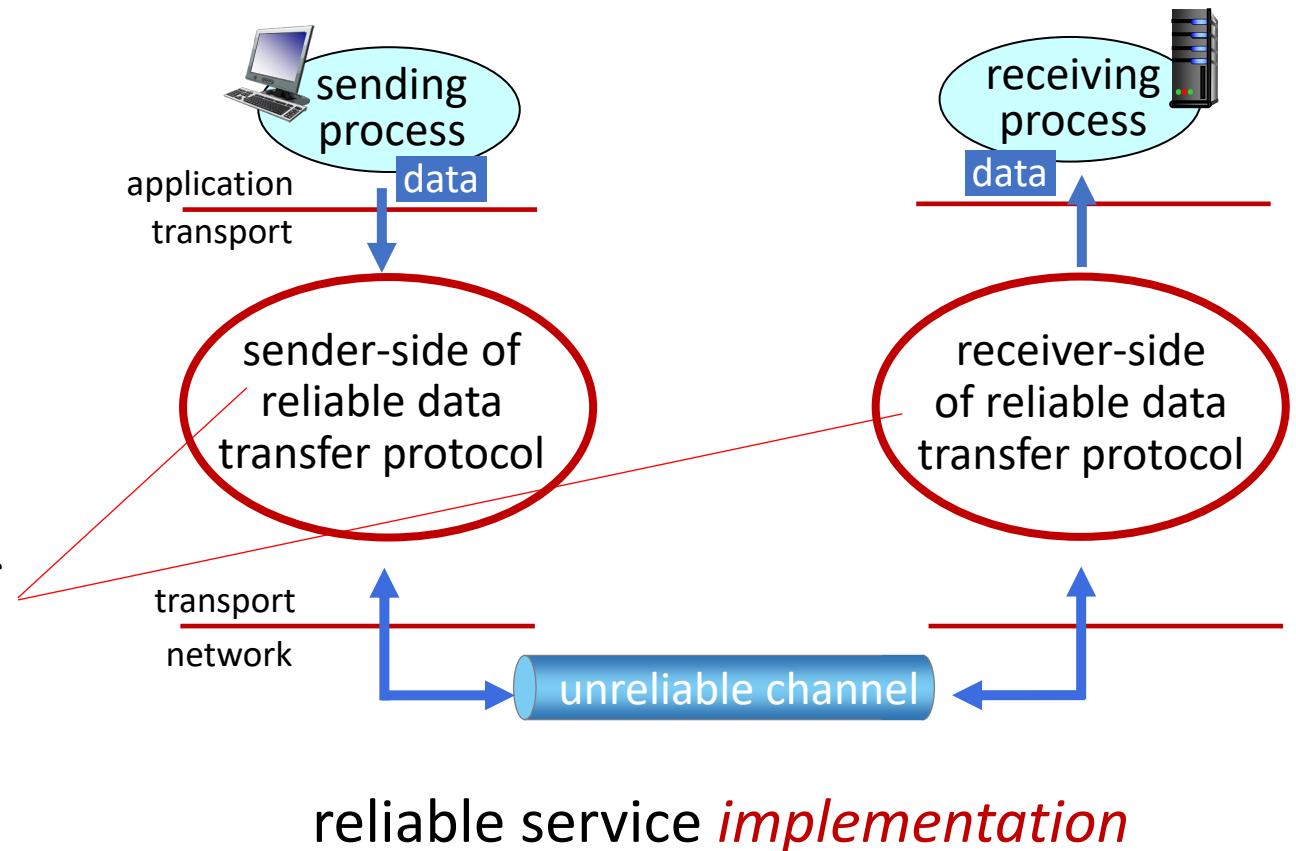


# Principles of reliable data transfer



# Principles of reliable data transfer

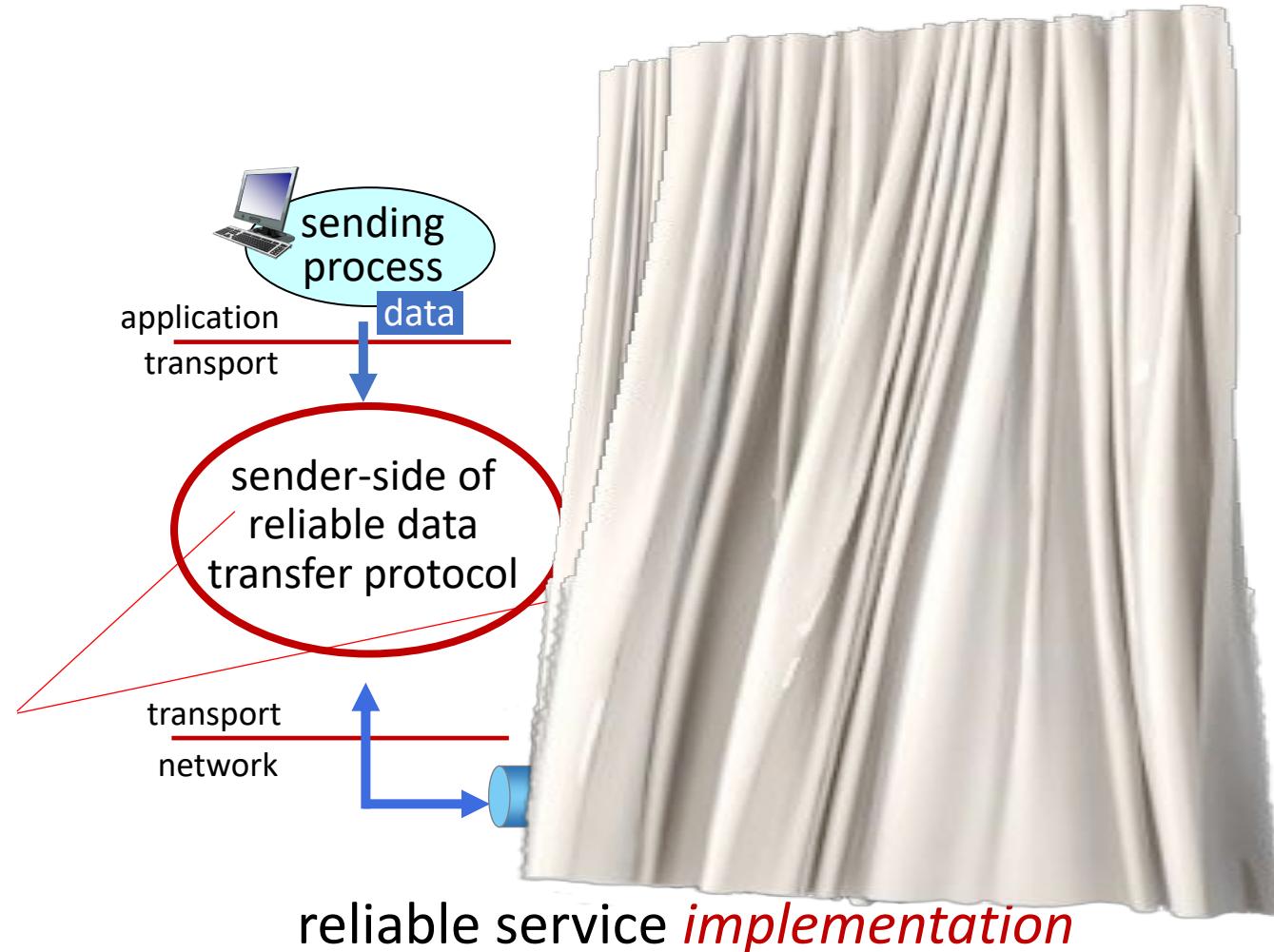
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



# Principles of reliable data transfer

Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

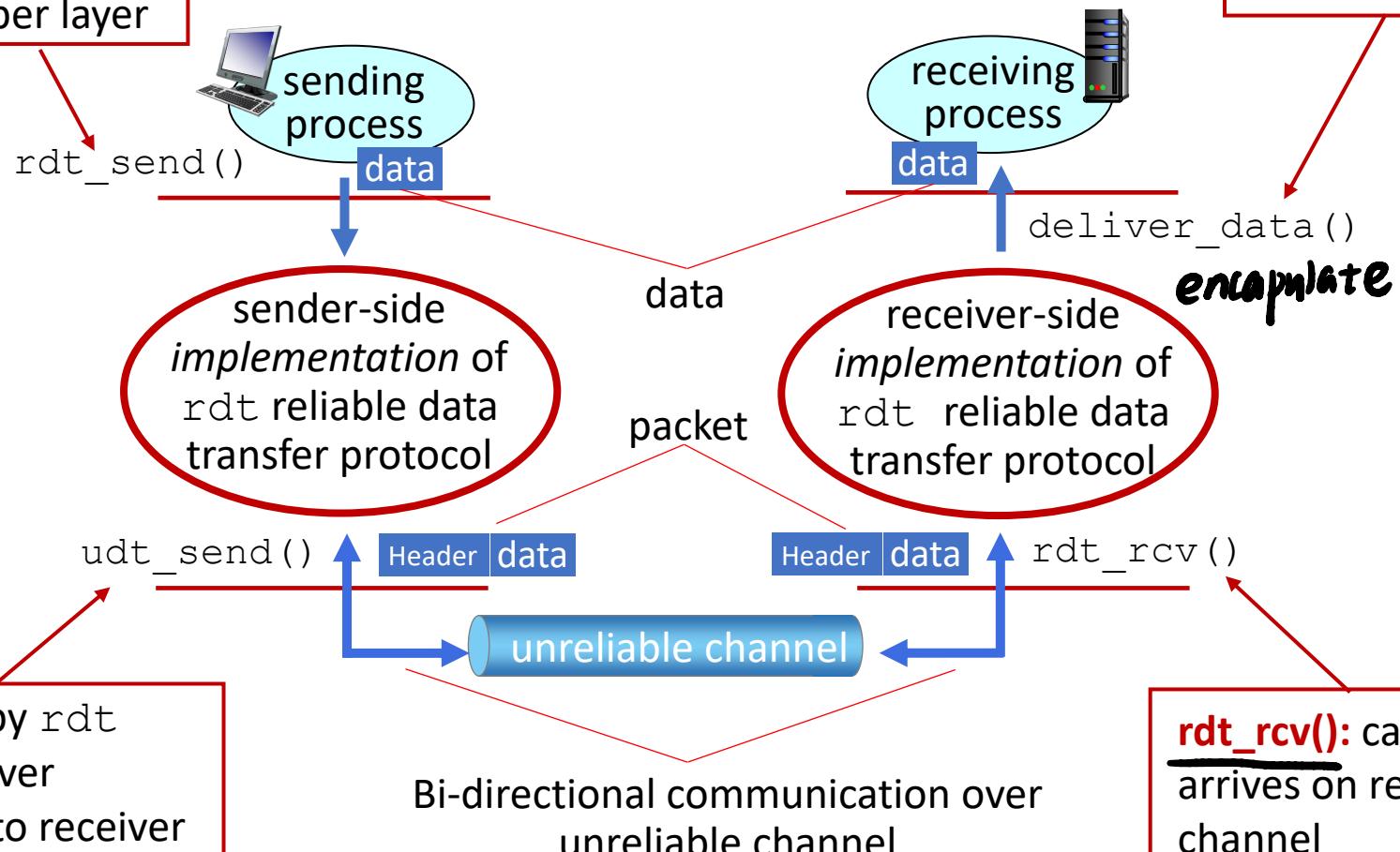
- unless communicated via a message



# Reliable data transfer protocol (rdt): interfaces

reliable data transfer

**rdt\_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



**udt\_send()**: called by rdt to transfer packet over unreliable channel to receiver

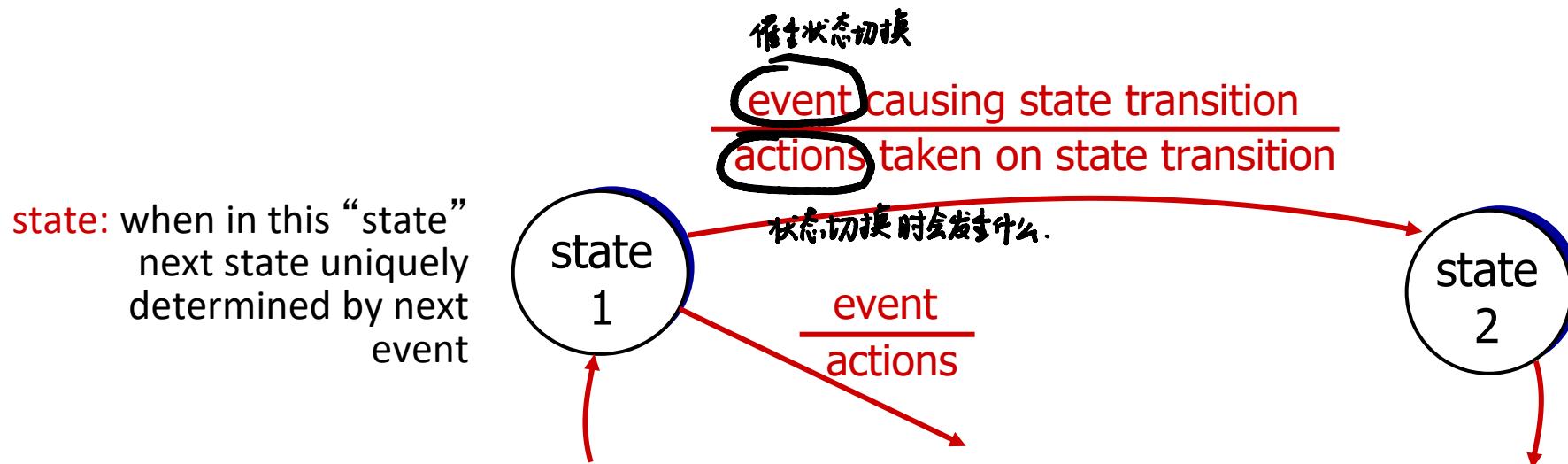
**deliver\_data()**: called by rdt to deliver data to upper layer

**rdt\_rcv()**: called when packet arrives on receiver side of channel

# Reliable data transfer: getting started

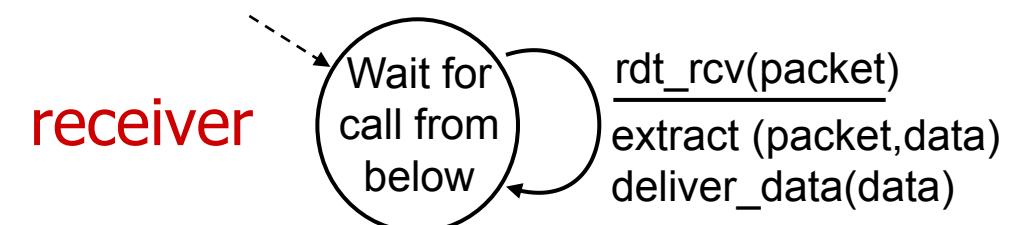
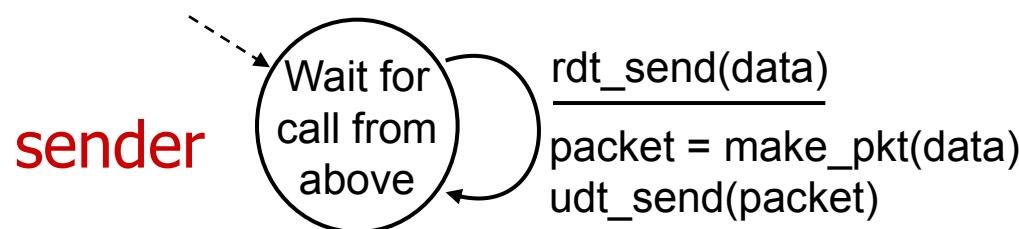
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



# rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



信道会发错

# rdt2.0: channel with bit errors Flip Bit

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the question: how to recover from errors?*

*How do humans recover from “errors” during conversation?*

# rdt2.0: channel with bit errors

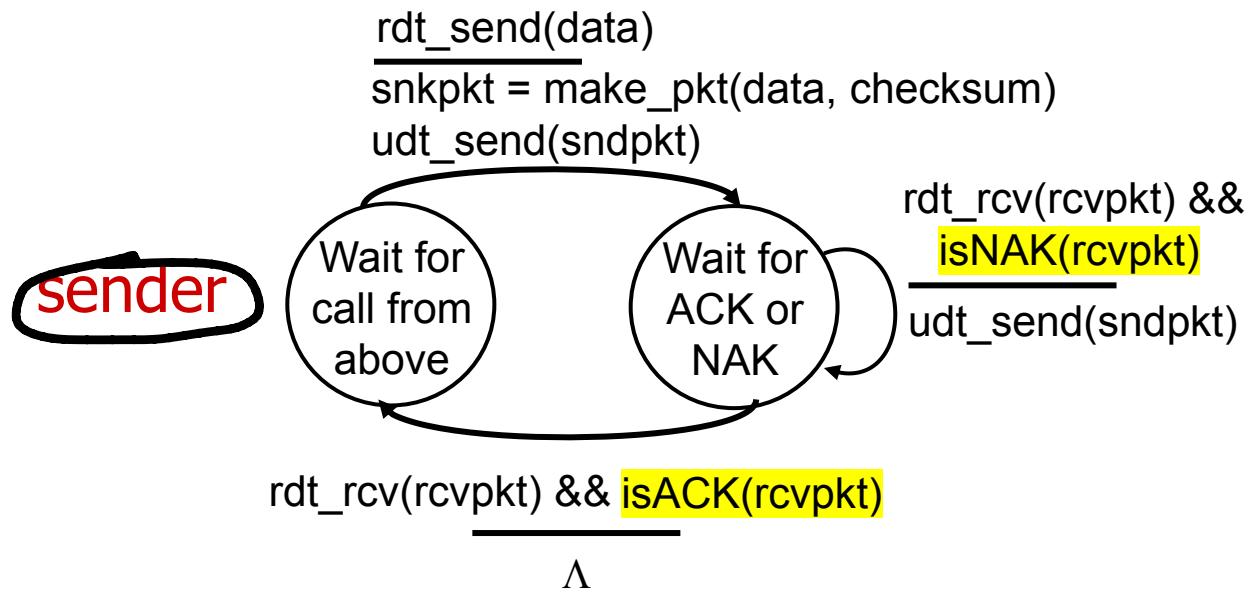
只发

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question:* how to recover from errors?
  - acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

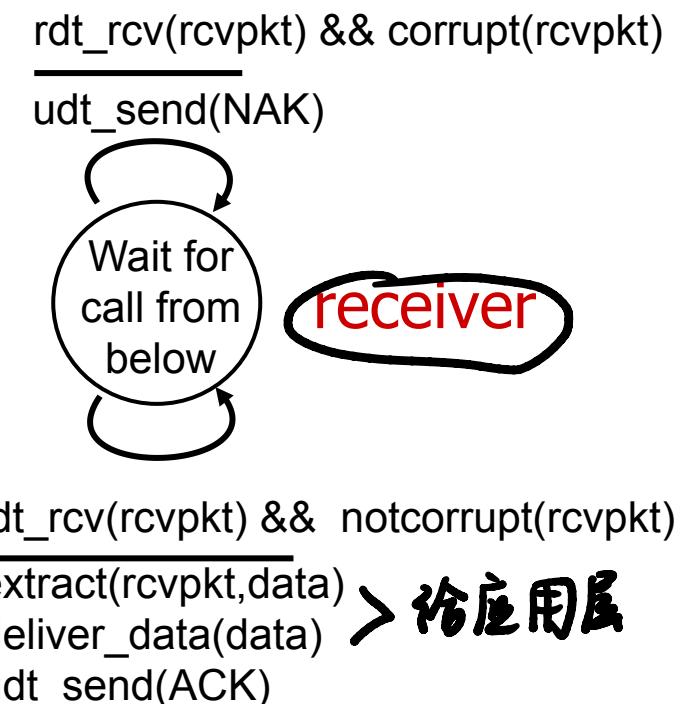
stop and wait

sender sends one packet, then waits for receiver response

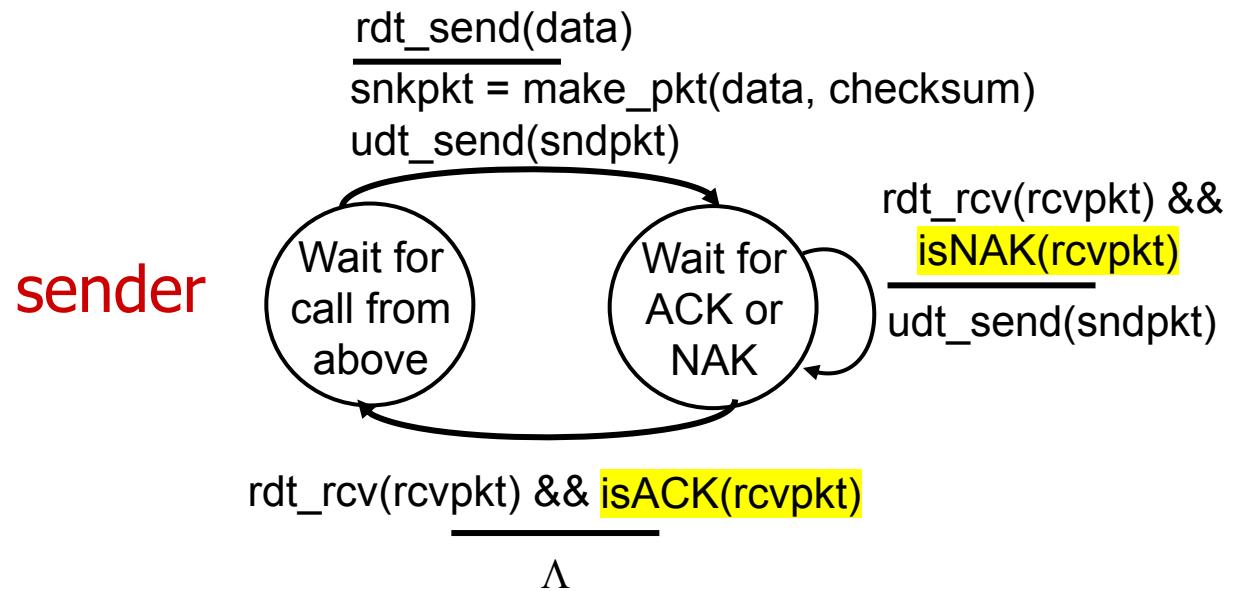
# rdt2.0: FSM specification



**Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender  
■ that’s why we need a protocol!



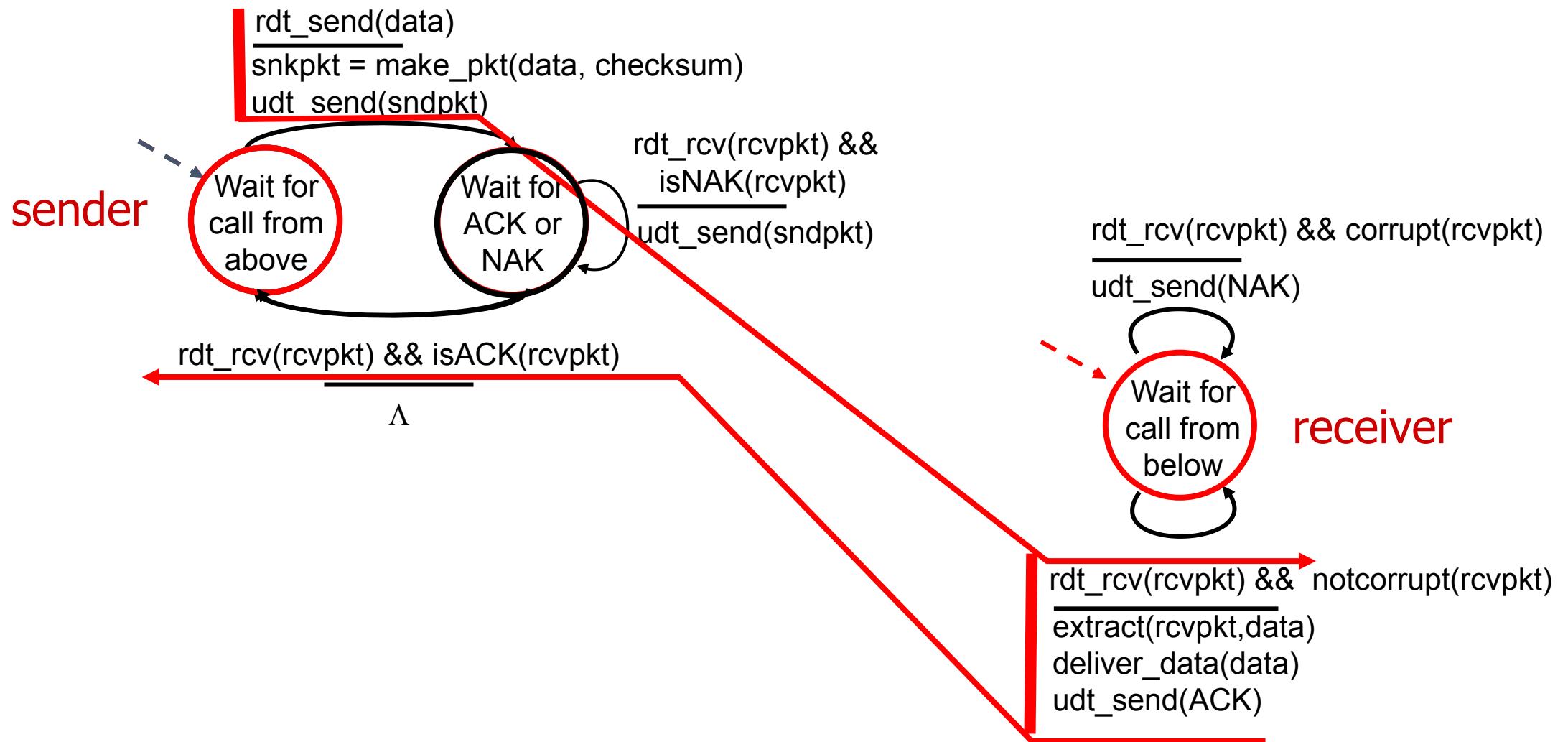
# rdt2.0: FSM specification



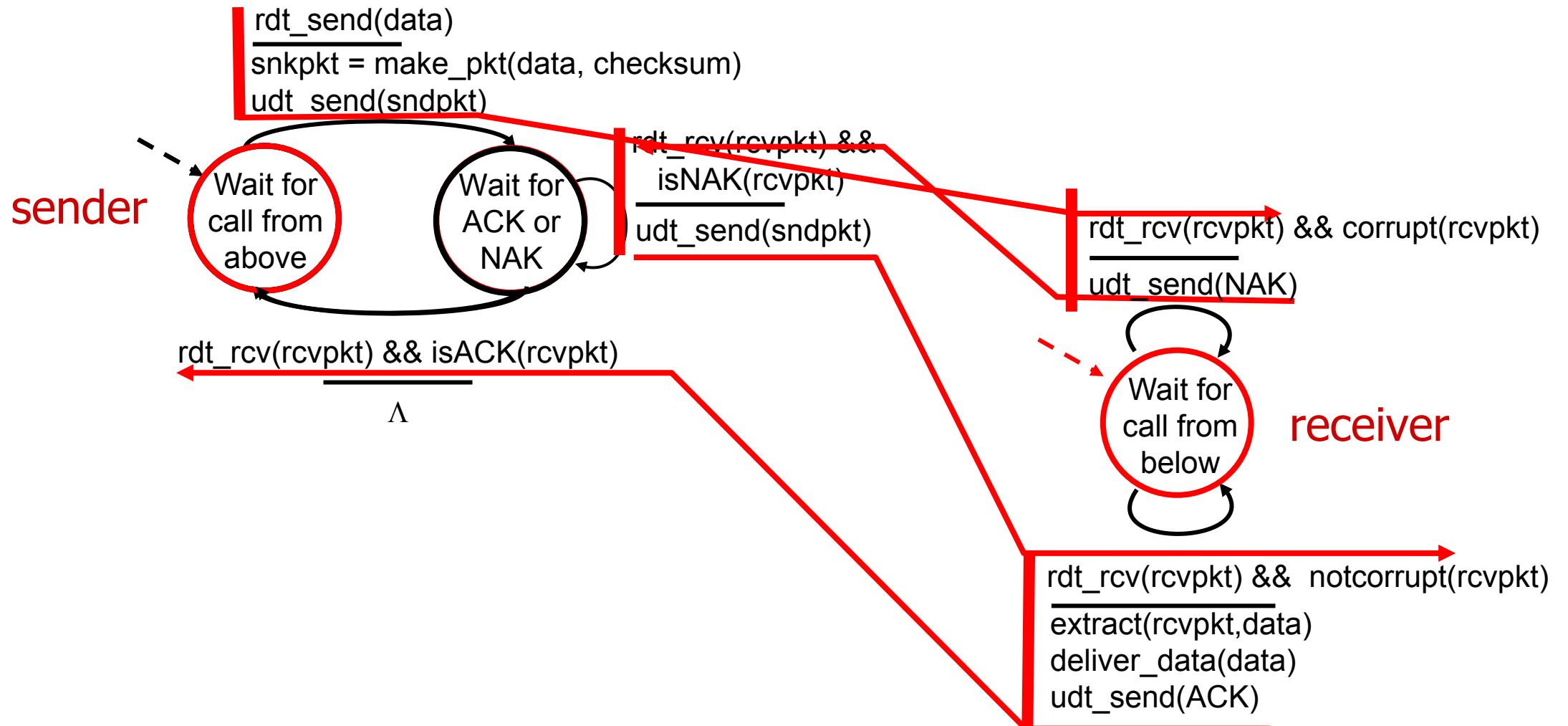
**Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender  
▪ that’s why we need a protocol!



# rdt2.0: operation with no errors



# rdt2.0: corrupted packet scenario



# rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

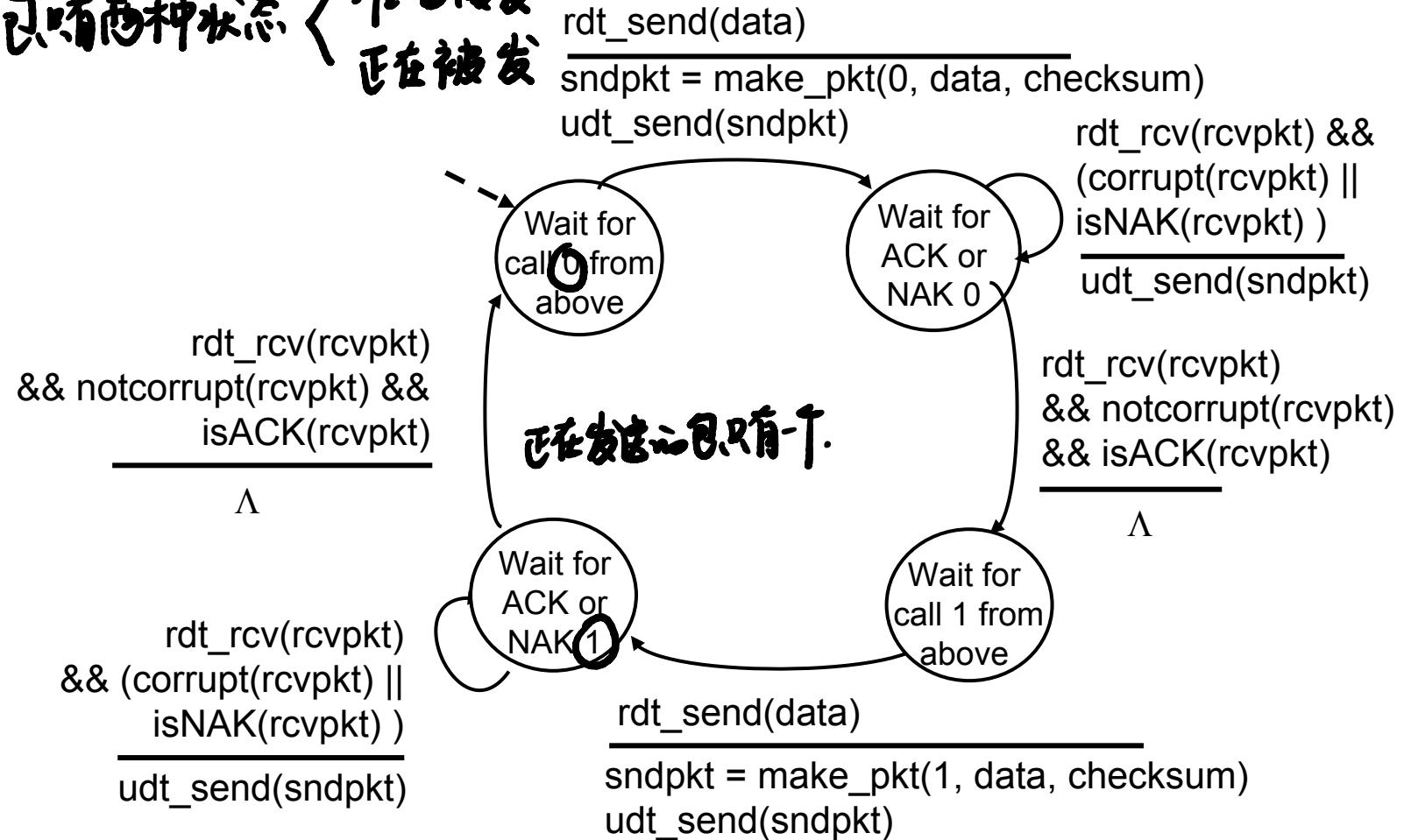
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt  
重复可辨别, 可丢弃.
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

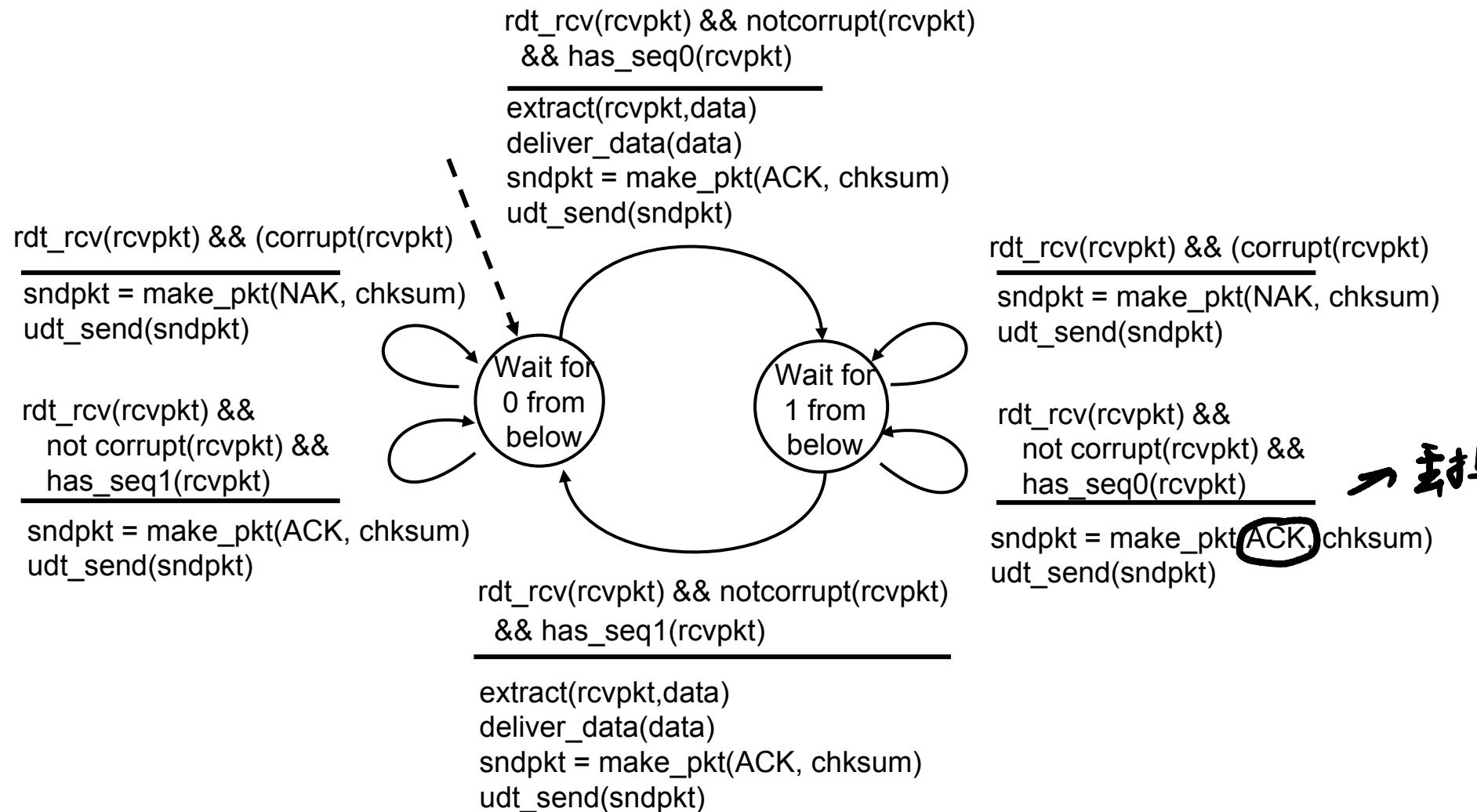
sender sends one packet, then waits for receiver response

# rdt2.1: sender, handling garbled ACK/NAKs

一个包有两种状态 <准备被发 正在被发>



# rdt2.1: receiver, handling garbled ACK/NAKs



# rdt2.1: discussion

## sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice.  
Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

## receiver:

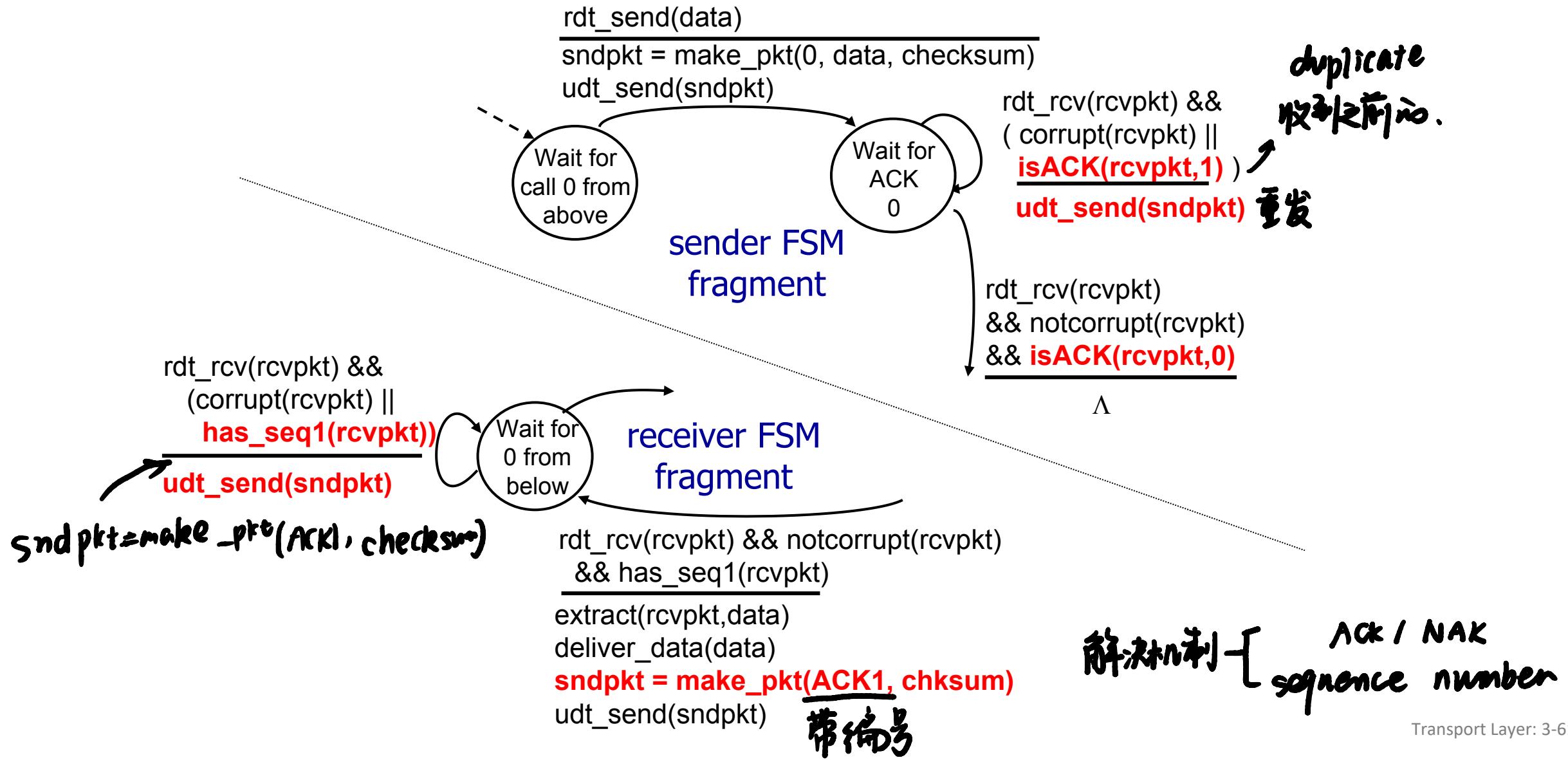
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

# rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:  
*retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors and loss

*New channel assumption:* underlying channel can also *lose* packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help ...  
but not quite enough



*Q:* How do *humans* handle lost sender-to-receiver words in conversation?

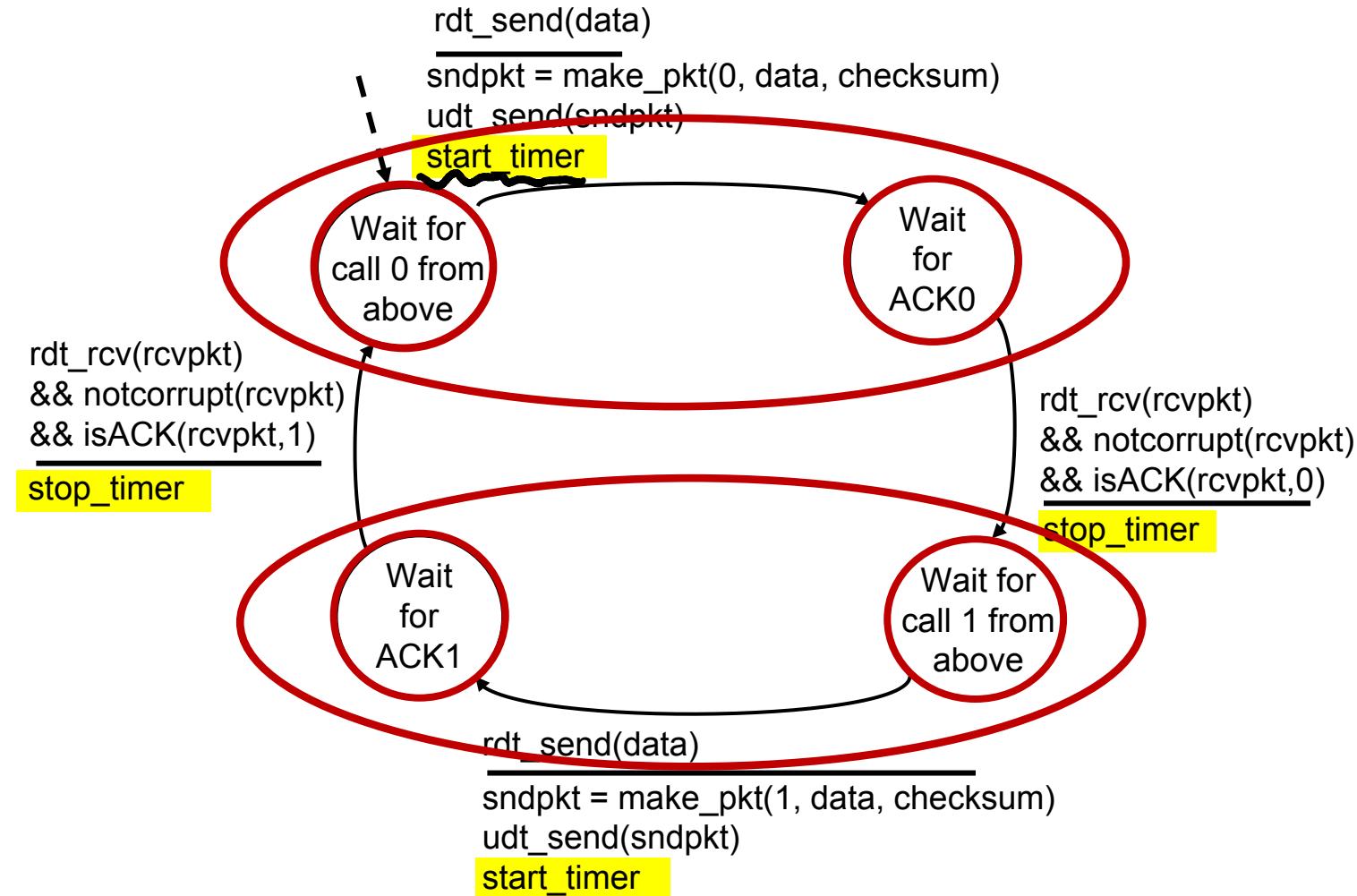
# rdt3.0: channels with errors *and* loss

**Approach:** sender waits “reasonable” amount of time for ACK

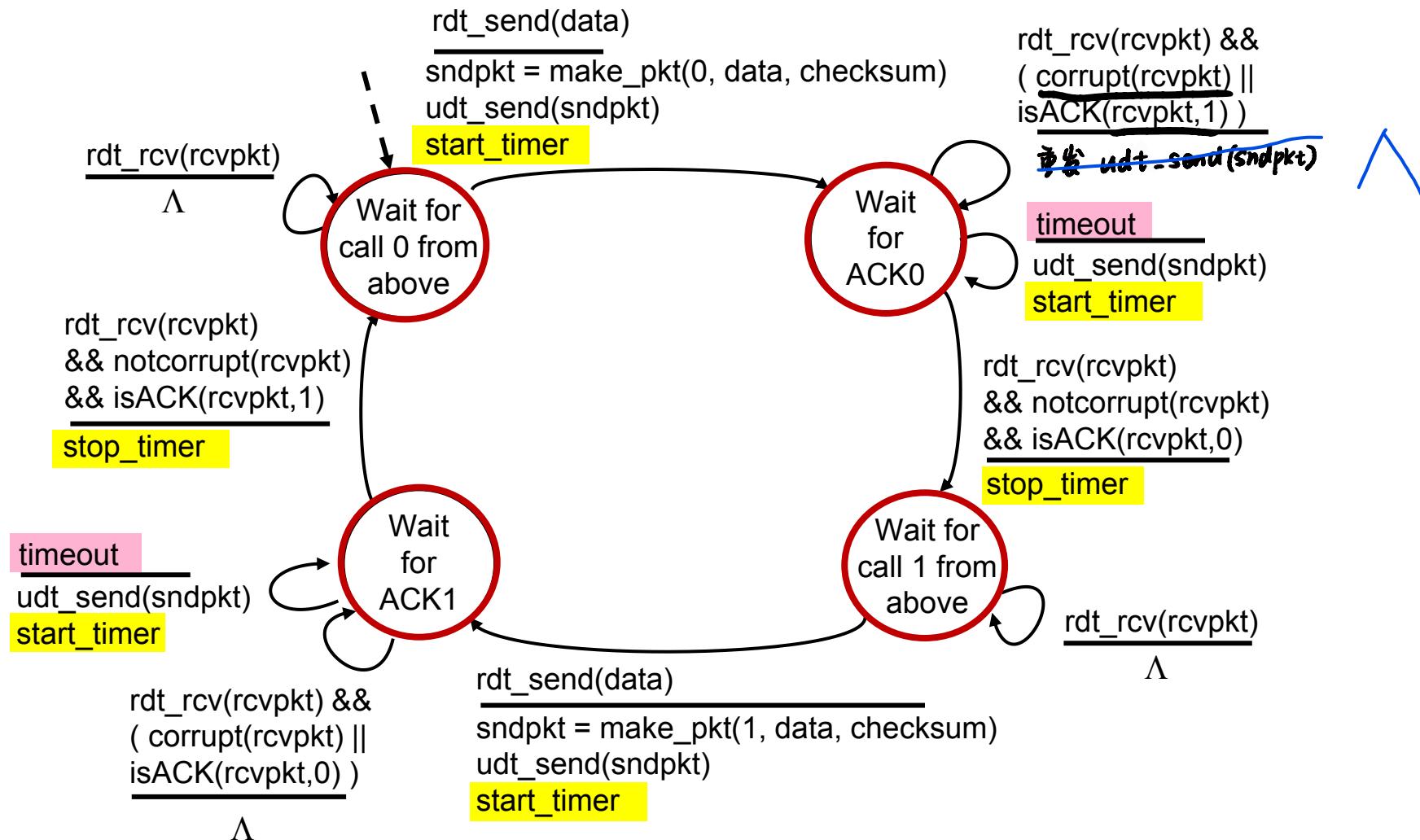
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time



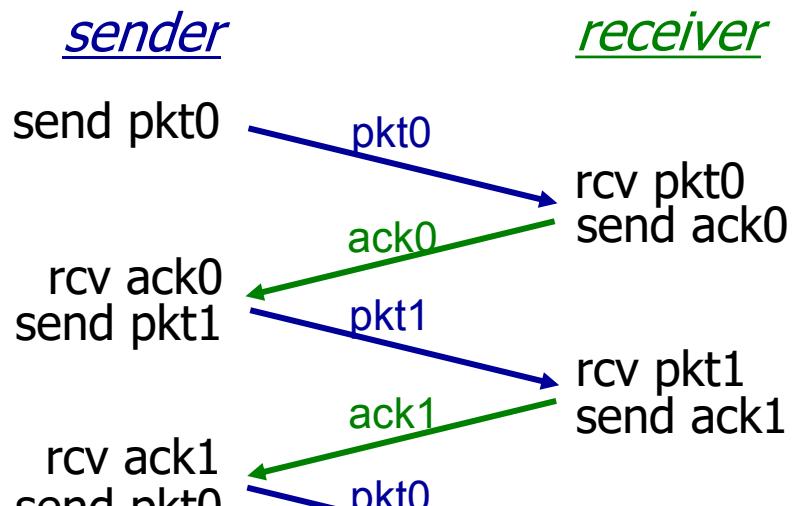
# rdt3.0 sender



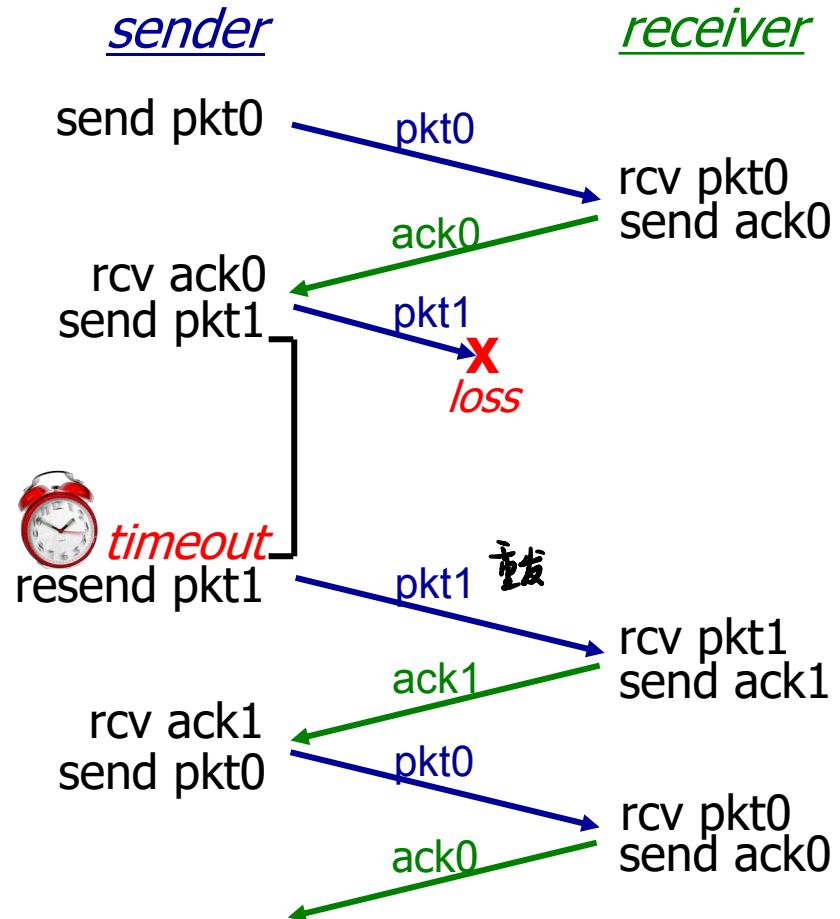
# rdt3.0 sender



# rdt3.0 in action

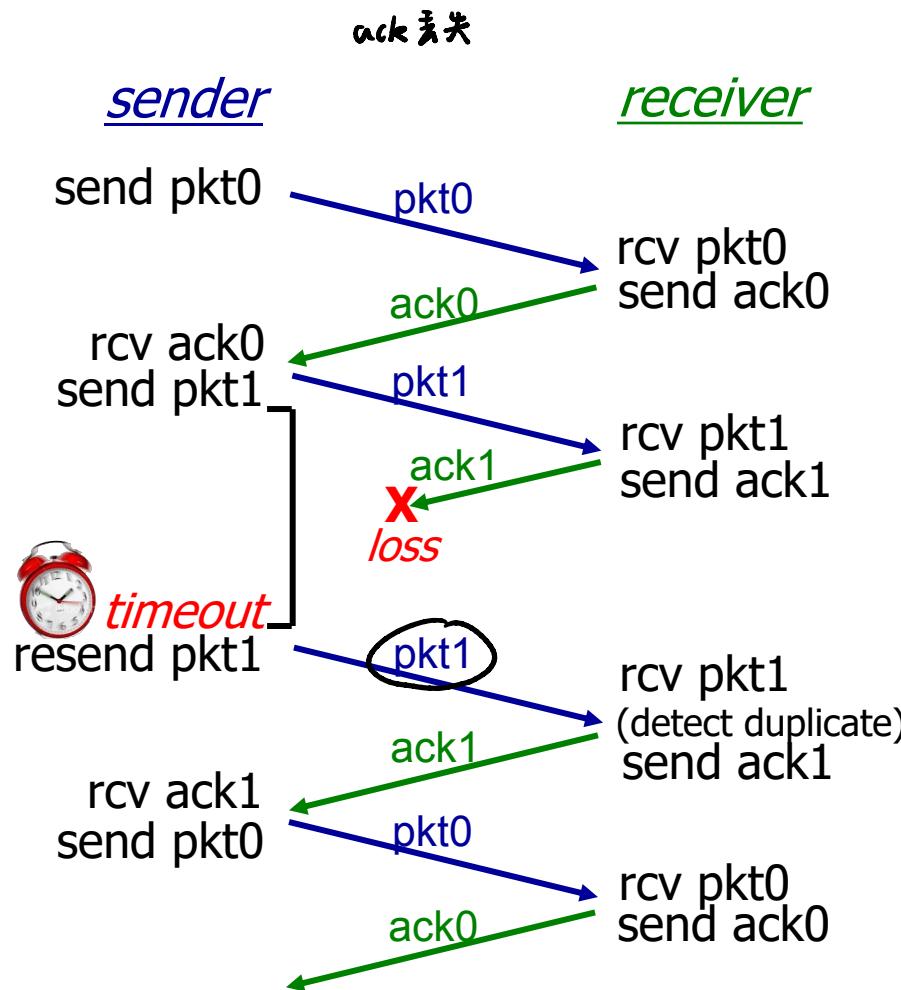


(a) no loss

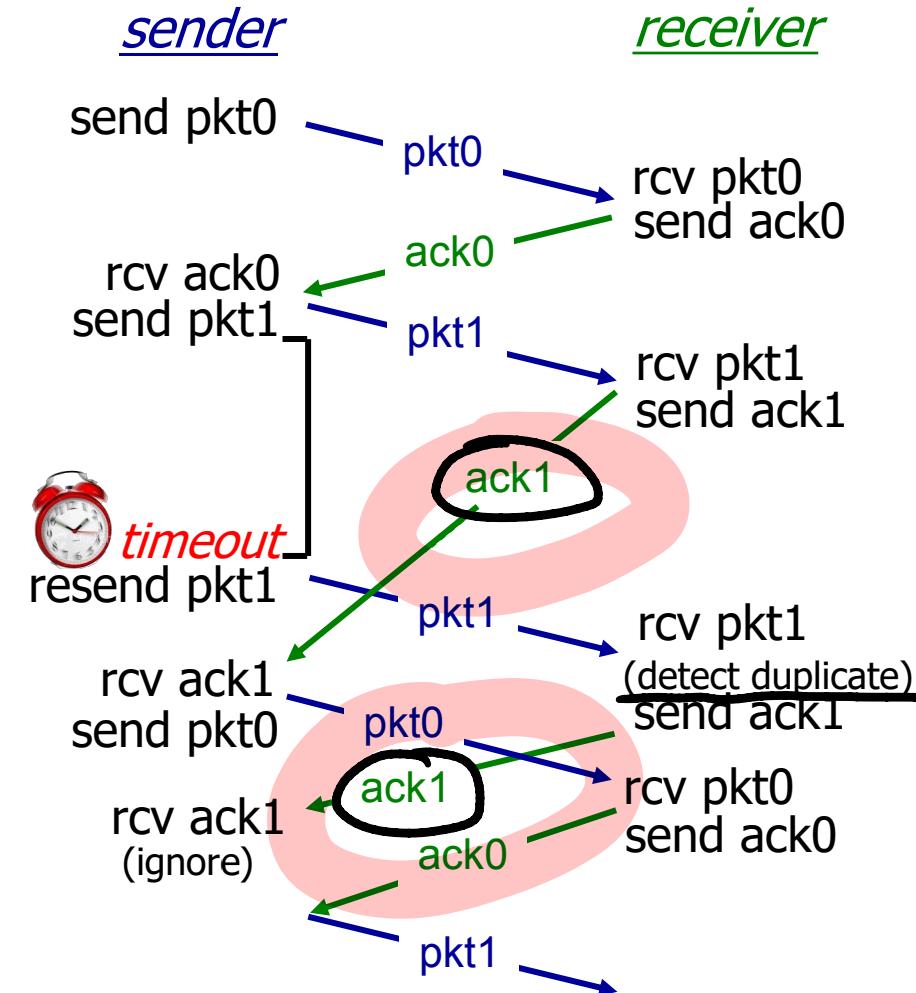


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Performance of rdt3.0 (stop-and-wait)

发送后等回信才继续下一步



# Performance of rdt3.0 (stop-and-wait)

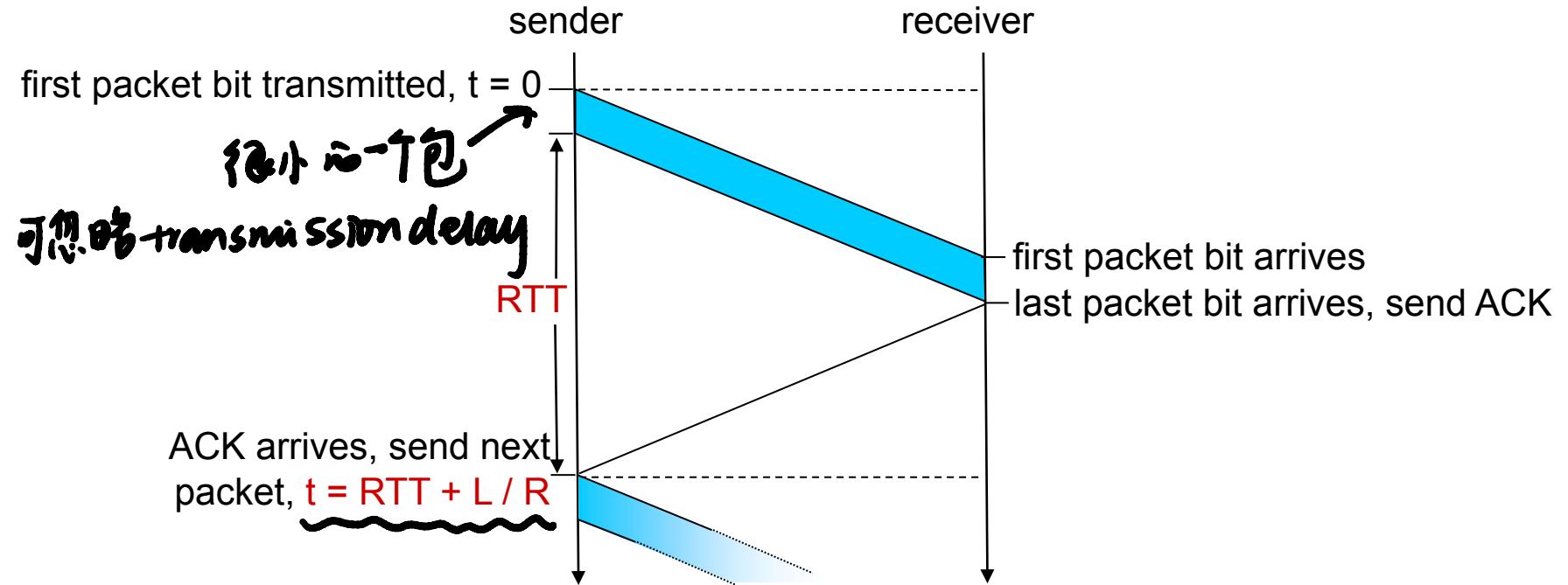
- $U_{\text{sender}}$ : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet

- time to transmit packet into channel:

$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

*1 Gbps*

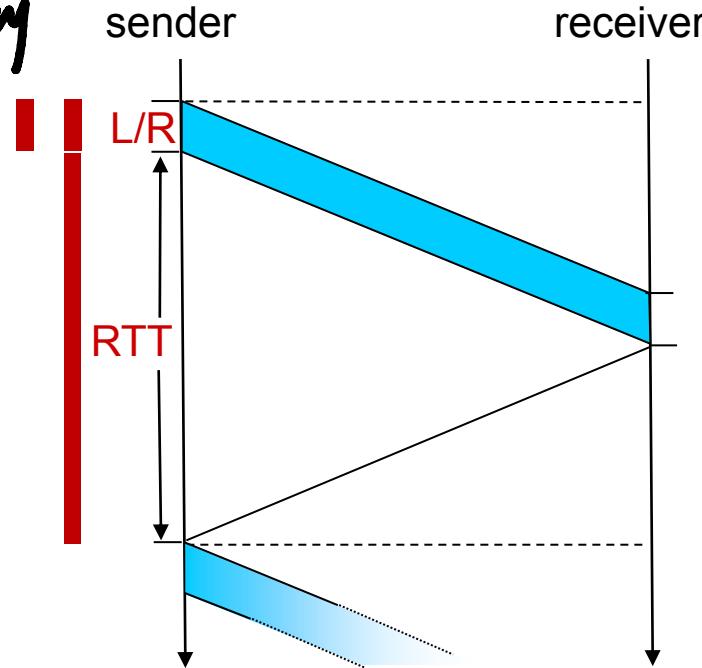
# rdt3.0: stop-and-wait operation



# rdt3.0: stop-and-wait operation

*trasmission delay*

$$U_{\text{sender}} = \frac{L / R}{RTT + L / R}$$
$$= \frac{.008}{30.008}$$
$$= 0.00027$$



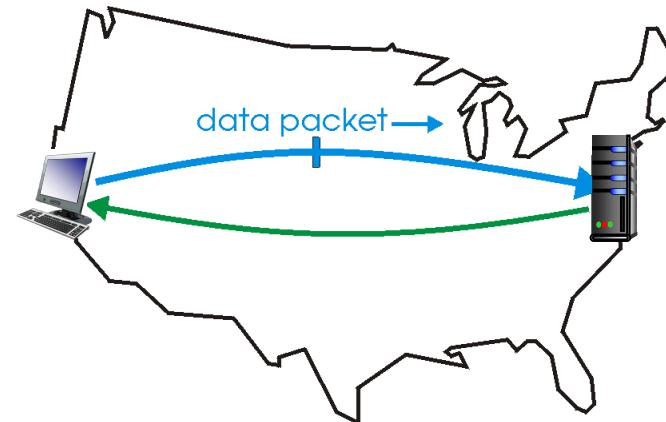
- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

# rdt3.0: pipelined protocols operation

更少 stop & wait

**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver

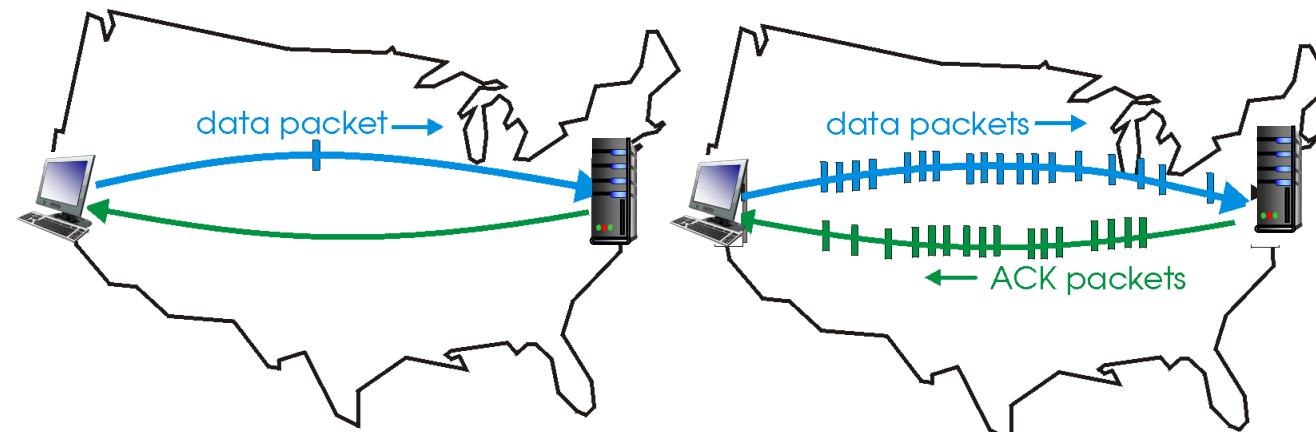


(a) a stop-and-wait protocol in operation

# rdt3.0: pipelined protocols operation

**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

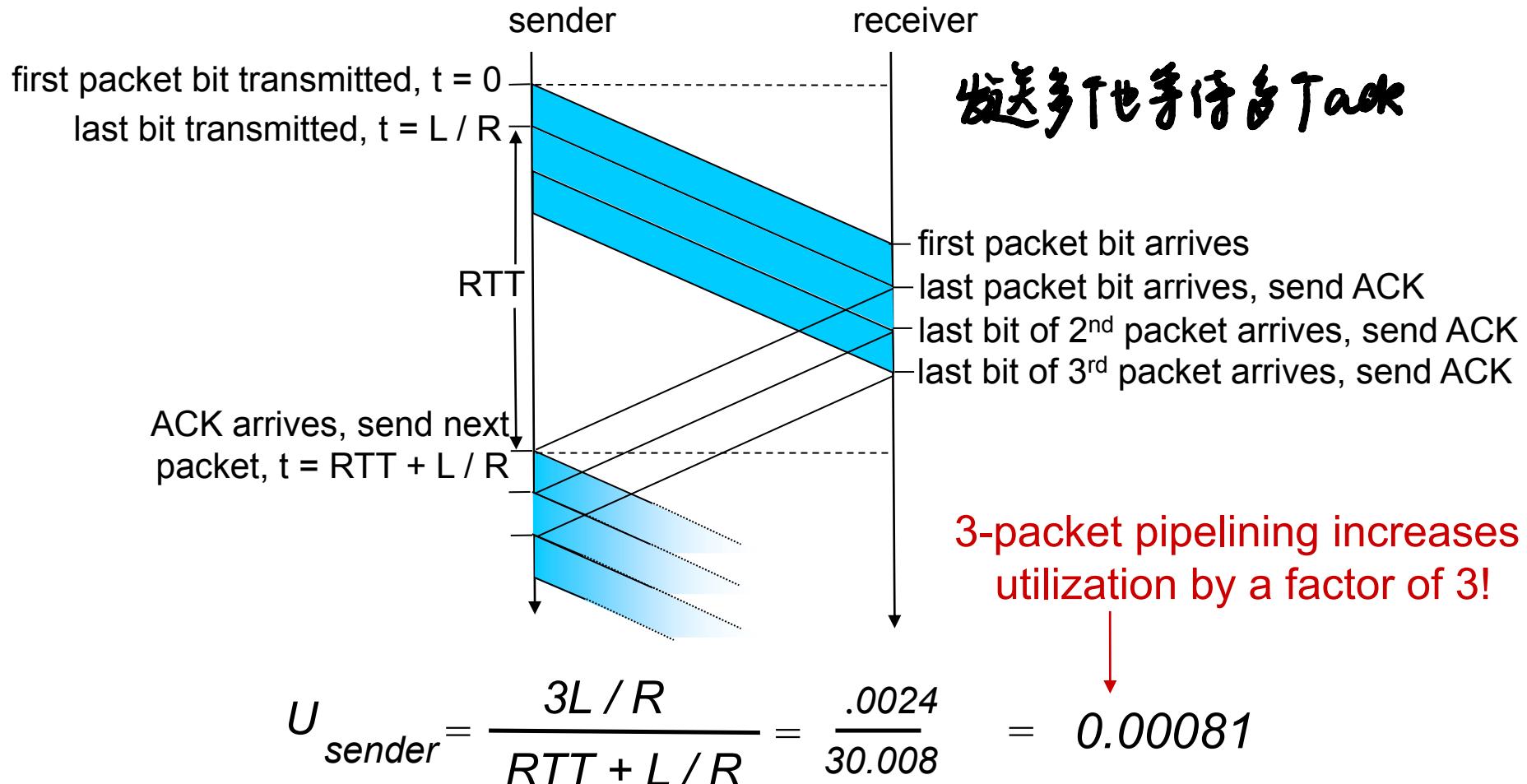
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

# Pipelining: increased utilization

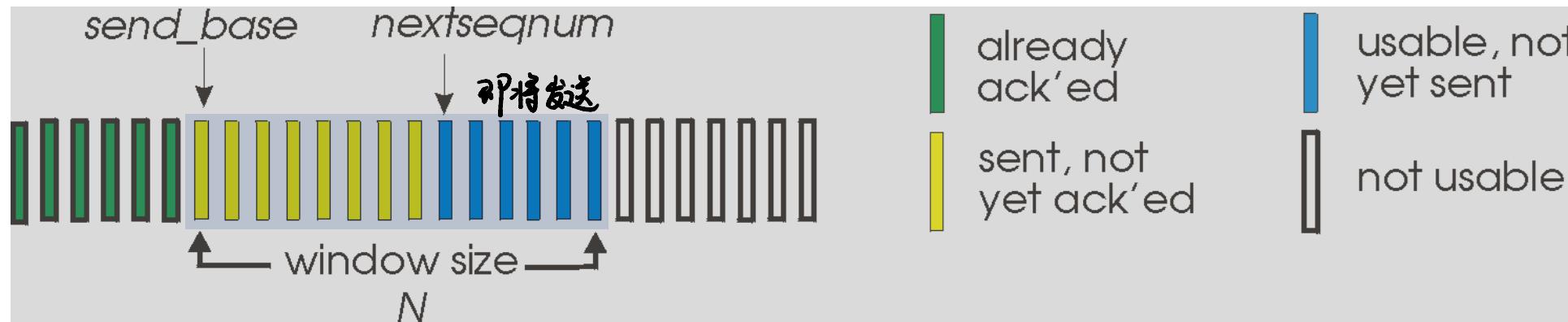


# Go-Back-N (回退N步) : sender

GBN

sliding window

- sender: “window” of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

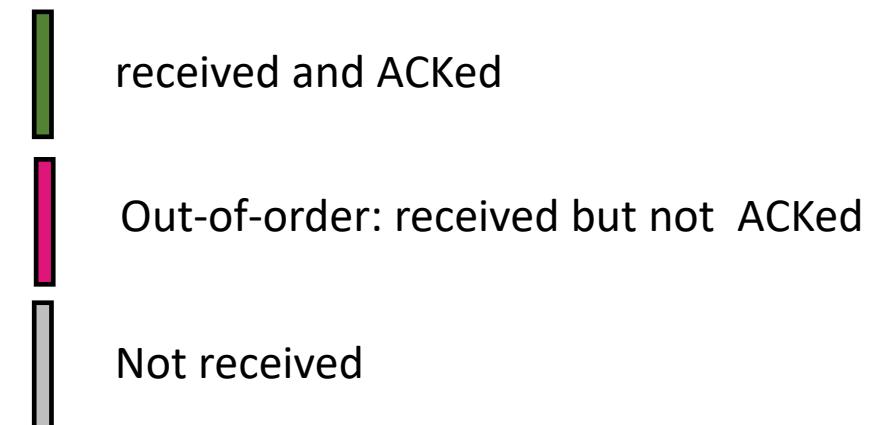
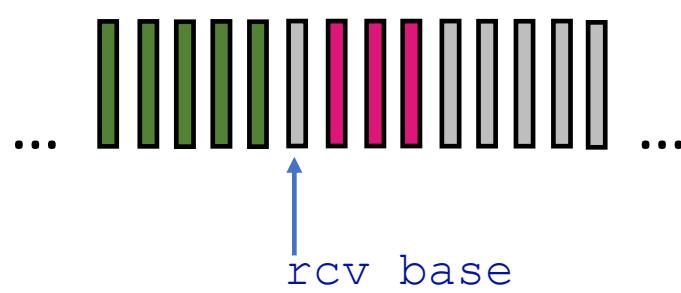


- cumulative ACK: ACK( $n$ ): ACKs all packets up to, including seq#  $n$ 
  - on receiving ACK( $n$ ): move window forward to begin at  $\underline{n+1}$   
向前滑动窗口。
- timer for oldest in-flight packet
- $timeout(n)$ : retransmit packet  $n$  and all higher seq # packets in window  
发生timeout → 窗口内消息全部重发

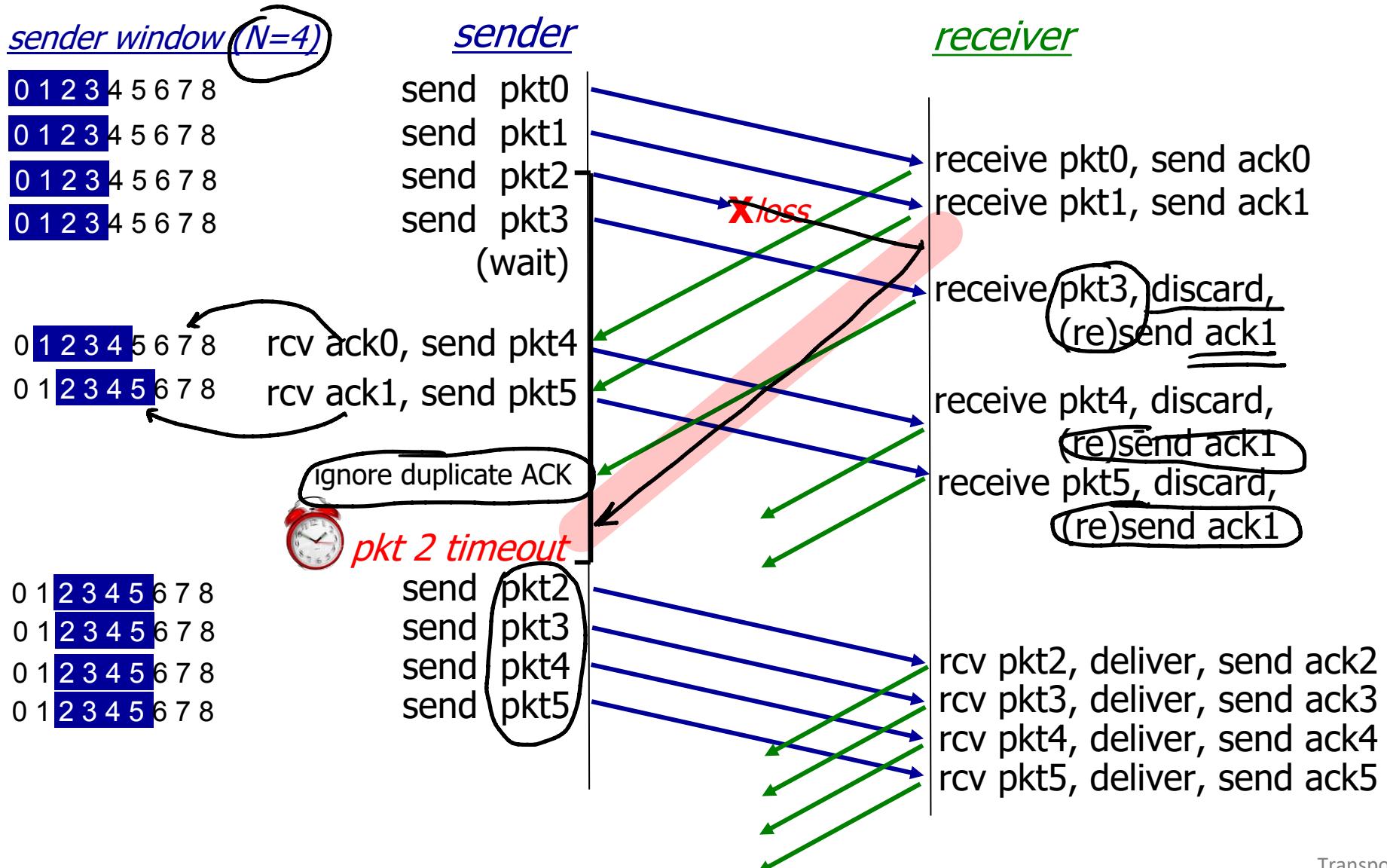
# Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



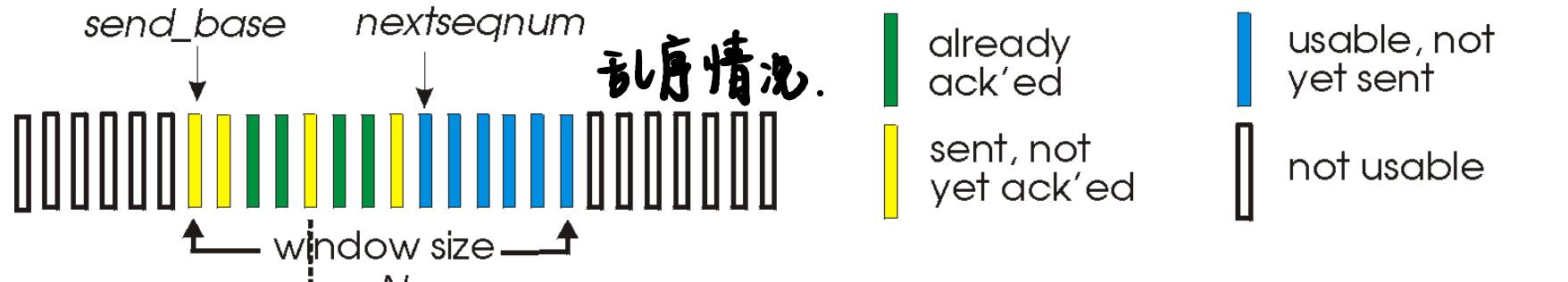
# Go-Back-N in action



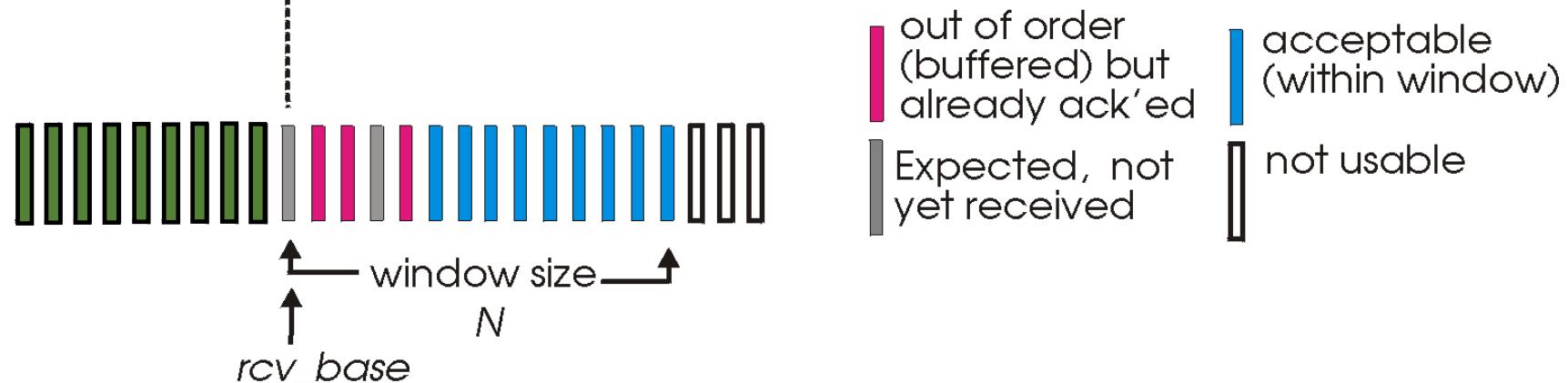
# Selective repeat (选择重传)

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - $N$  consecutive seq #s
  - limits seq #s of sent, unACKed packets

# Selective repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

# Selective repeat: sender and receiver

## sender

data from above:

- if next available seq # in window, send packet

$\text{timeout}(n)$ :

- resend packet  $n$ , restart timer

$\text{ACK}(n)$  in  $[\text{sendbase}, \text{sendbase}+N]$ :

- mark packet  $n$  as received 
- if  $n$  is smallest unACKed packet, advance window base to next unACKed seq #

## receiver

packet  $n$  in  $[\text{rcvbase}, \text{rcvbase}+N-1]$

- send  $\text{ACK}(n)$
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet  $n$  in  $[\text{rcvbase}-N, \text{rcvbase}-1]$

- $\text{ACK}(n)$

otherwise:

- ignore

# Selective Repeat in action

sender window ( $N=4$ )

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0  
send pkt1  
send pkt2  
send pkt3  
(wait)

rcv ack0, send pkt4  
rcv ack1, send pkt5

record ack3 arrived



*pkt 2 timeout*  
send pkt2  
(but not 3,4,5)

*Q: what happens when ack2 arrives?*

buffer 保有到ack

receiver

receive pkt0, send ack0  
receive pkt1, send ack1

receive pkt3, buffer,  
send ack3

receive pkt4, buffer,  
send ack4  
receive pkt5, buffer,  
send ack5

rcv pkt2; deliver pkt2,  
pkt3, pkt4, pkt5; send ack2

buffer 保有到pkt.

# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

- point-to-point:

- one sender, one receiver

- reliable, in-order byte steam:

- no “message boundaries”

- full duplex (全双工) data:

- bi-directional data flow in same connection

- MSS: maximum segment size

1460, 1440, 1420. 链路层以太网 frame 1500字节 - 网络层 header.

我爱你，爱着你，就像老鼠爱大米，巴拉巴拉... (省略一万字)



# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam*:
  - no “message boundaries”
- full duplex (全双工) data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

# TCP segment structure

ACK: seq # of next expected byte; A bit: this is an ACK

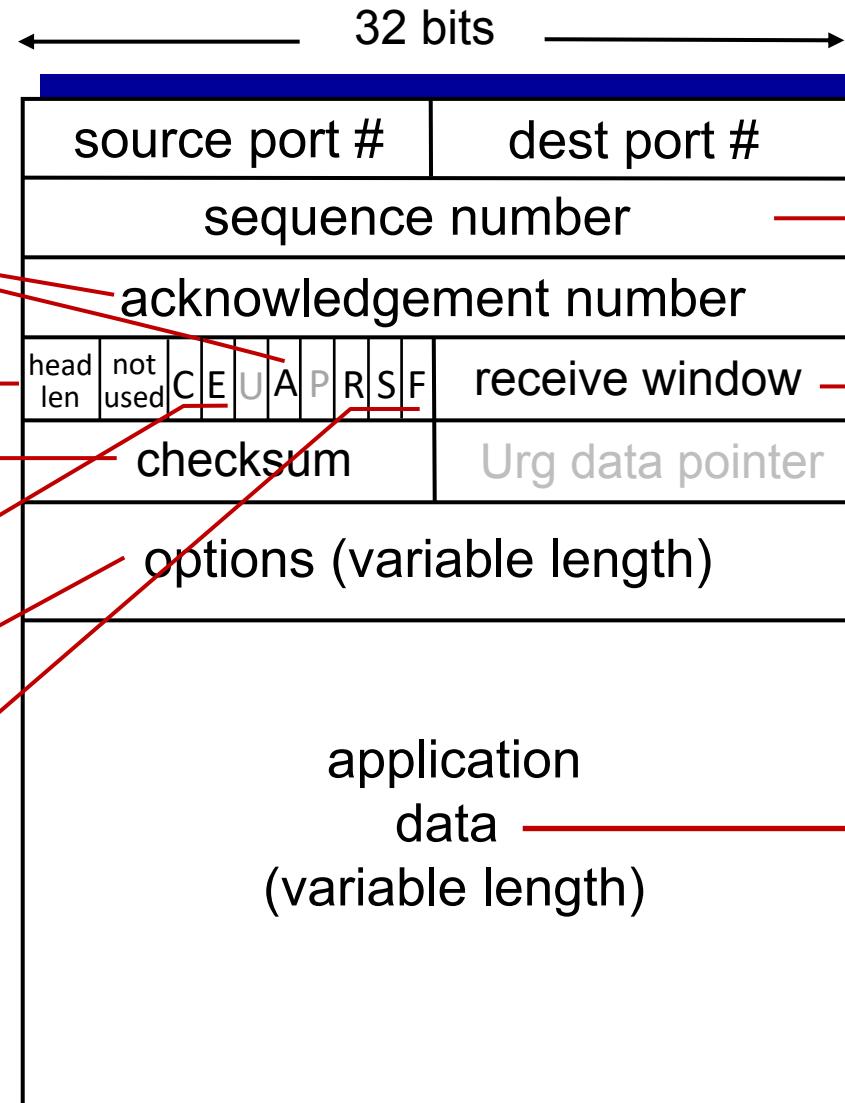
length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

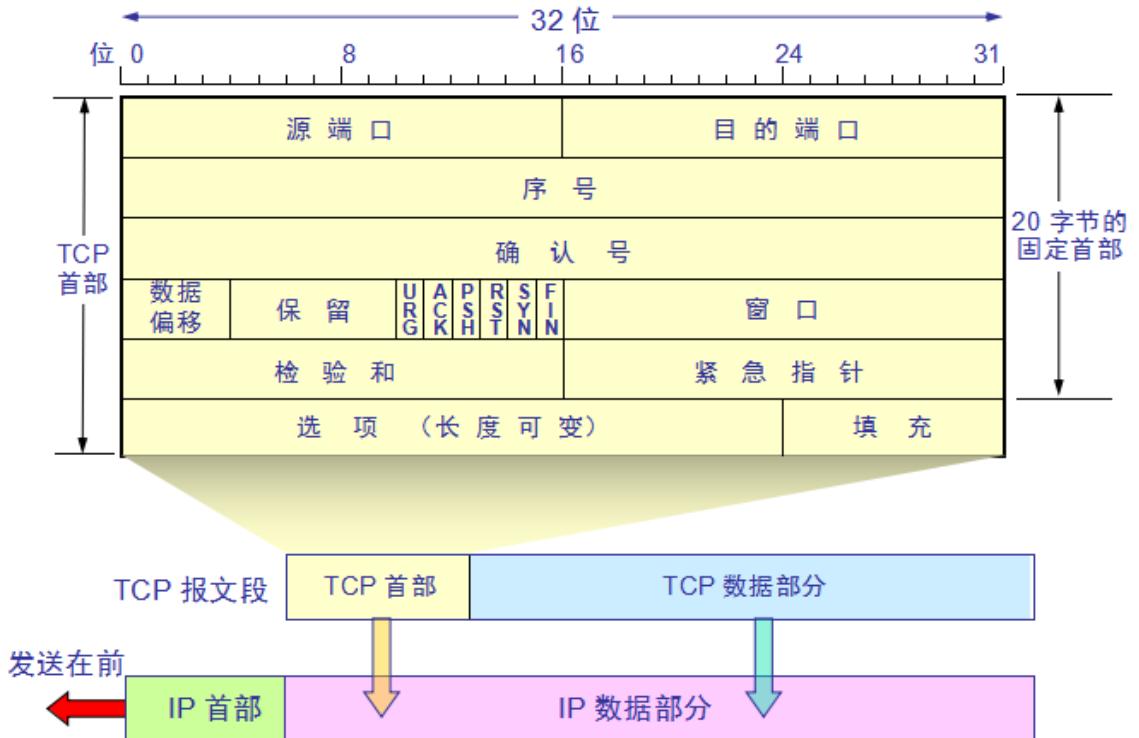


segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

# TCP segment structure



- The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so called **maximum transmission unit, MTU**)
- Ethernet and PPP link-layer protocols  
MTU: 1,500 bytes.
- A typical value of MSS: 1460 bytes

# TCP sequence numbers, ACKs

## *Sequence numbers:*

- byte stream “number” of first byte in segment’s data

## *Acknowledgements:*

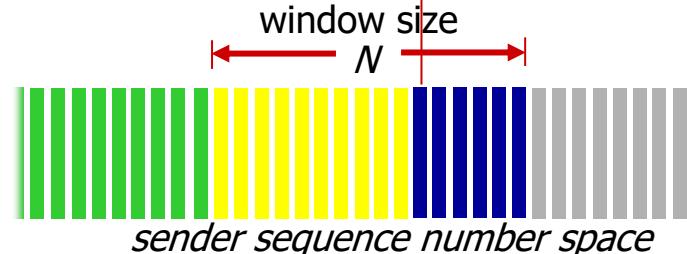
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn’t say, - up to implementor

outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

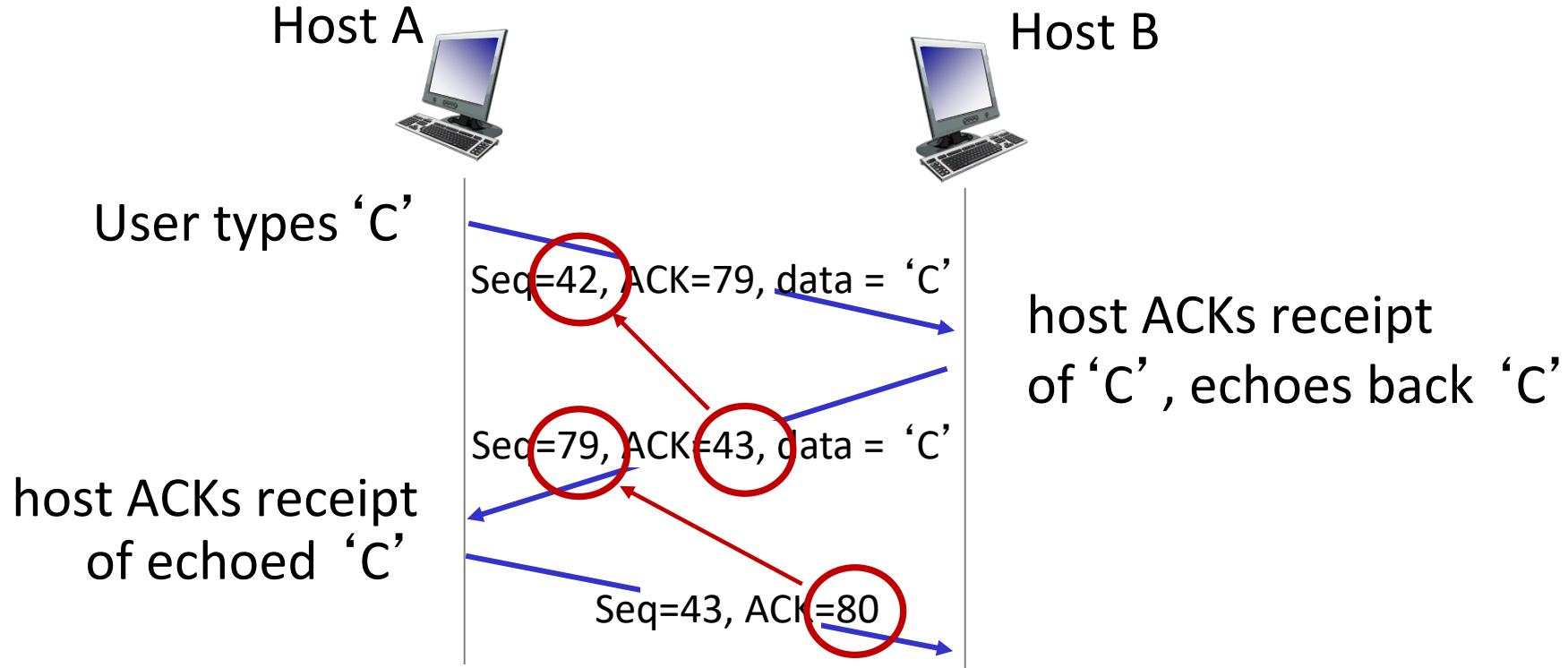


sent  
ACKed      sent, not-  
yet ACKed ("in-flight")      usable  
but not yet sent      not  
usable

outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

# TCP sequence numbers, ACKs



simple telnet scenario

# TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short*: premature timeout, unnecessary retransmissions
- *too long*: slow reaction to segment loss

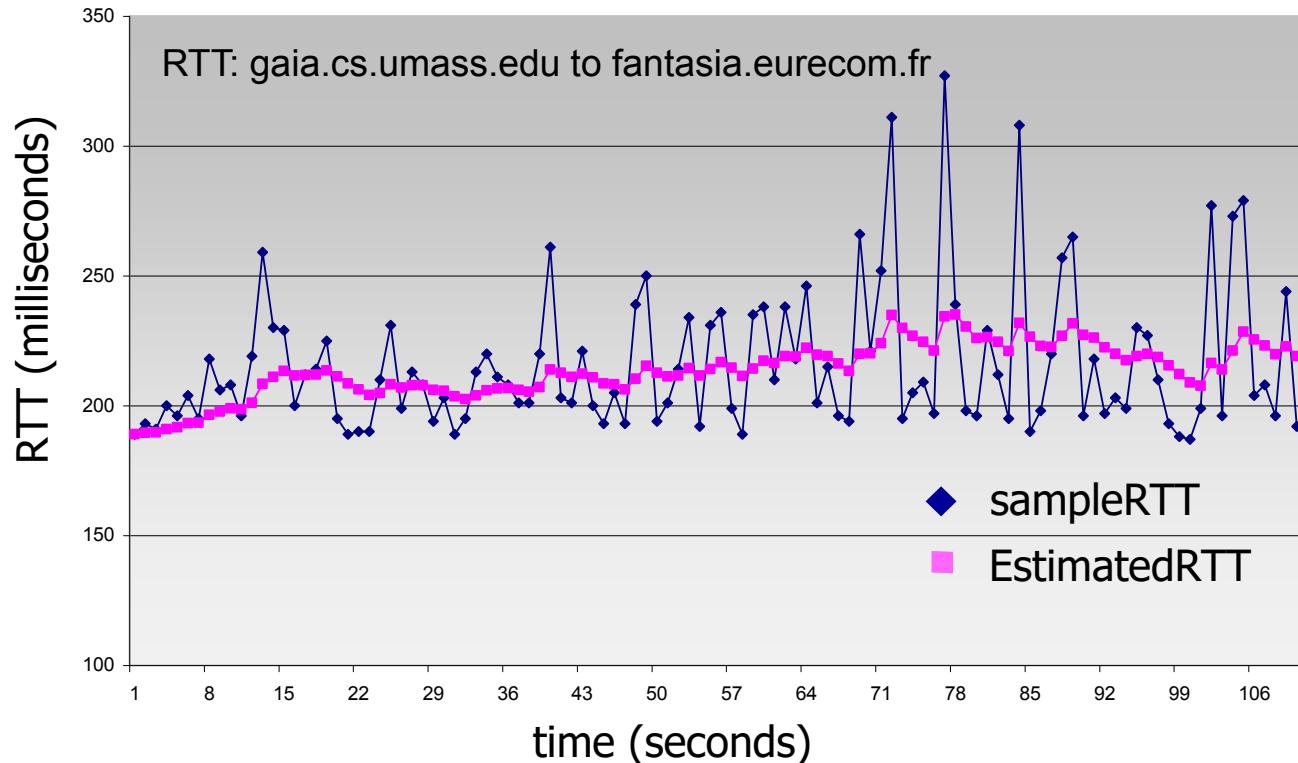
Q: how to estimate RTT?

- *SampleRTT*: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- *SampleRTT* will vary, want estimated RTT “smoother”
  - average several *recent* measurements, not just current *SampleRTT*

# TCP round trip time, timeout

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value:  $\alpha = 0.125$



# TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus “safety margin”
    - large variation in **EstimatedRTT**: want a larger safety margin

**TimeoutInterval = EstimatedRTT + 4\*DevRTT**



estimated RTT

“safety margin”

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

\* Check out the online interactive exercises for more examples: [http://qaia.cs.umass.edu/kurose\\_ross/interactive/](http://qaia.cs.umass.edu/kurose_ross/interactive/)

# TCP Sender (simplified)

*event: data received from application*

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval:  
**TimeOutInterval**

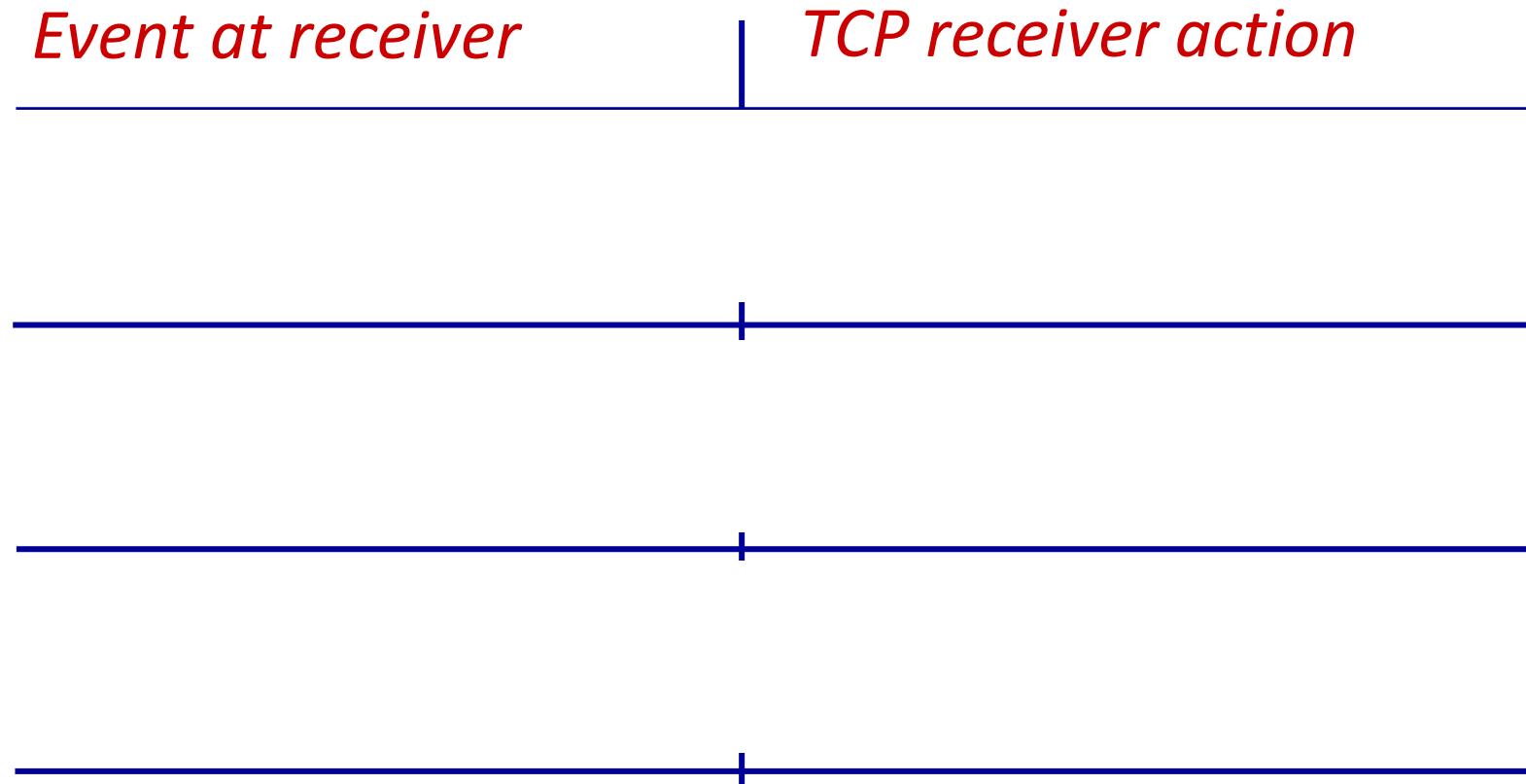
*event: timeout*

- retransmit segment that caused timeout
- restart timer

*event: ACK received*

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

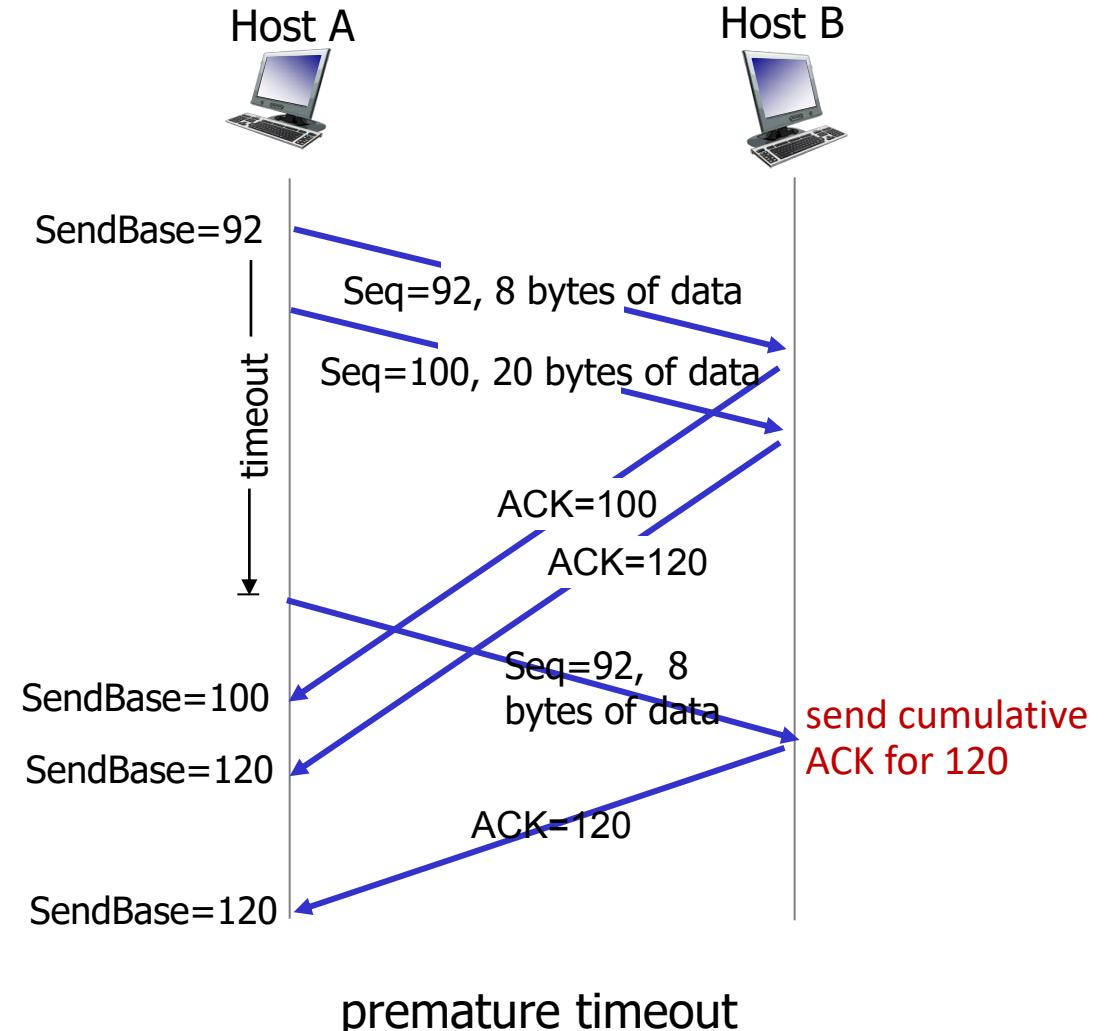
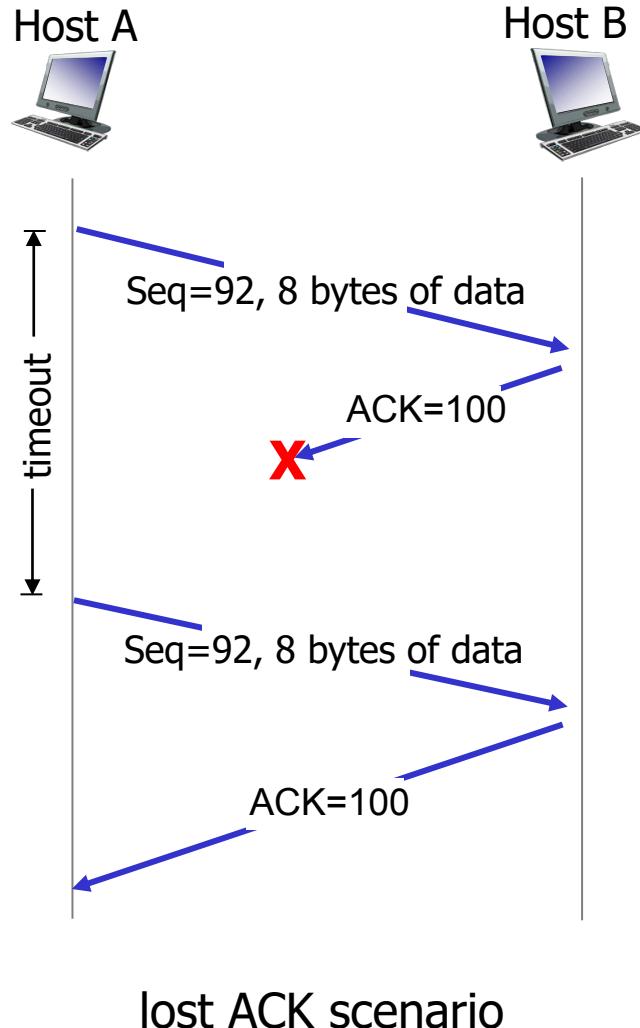
# TCP Receiver: ACK generation [RFC 5681]



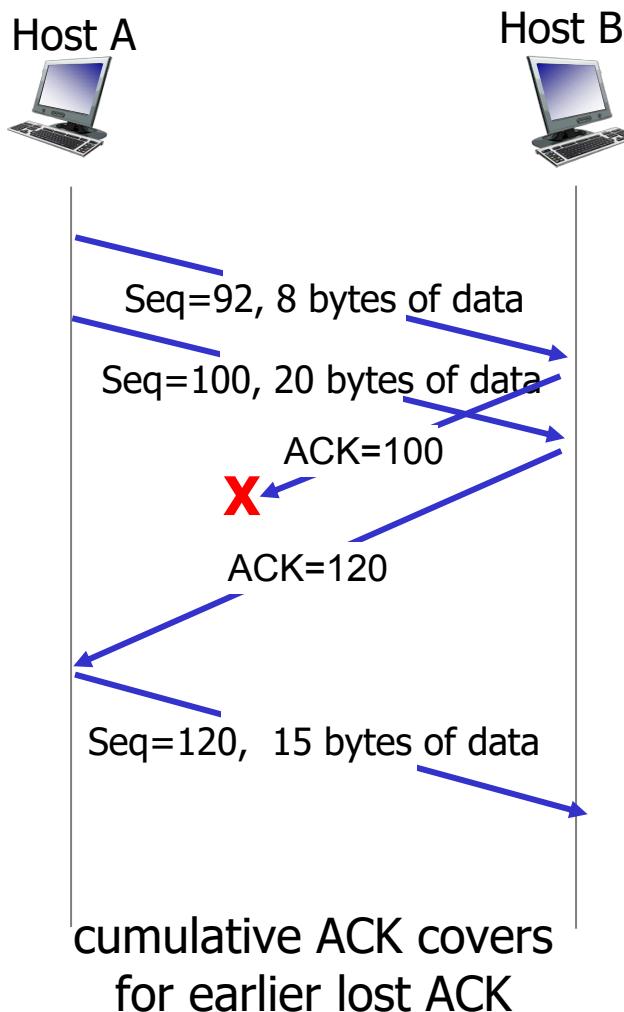
# TCP Receiver: ACK generation [RFC 5681]

<i>Event at receiver</i>	<i>TCP receiver action</i>
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send <i>duplicate ACK</i> , indicating seq. # of next expected byte

# TCP: retransmission scenarios



# TCP: retransmission scenarios



# TCP fast retransmit

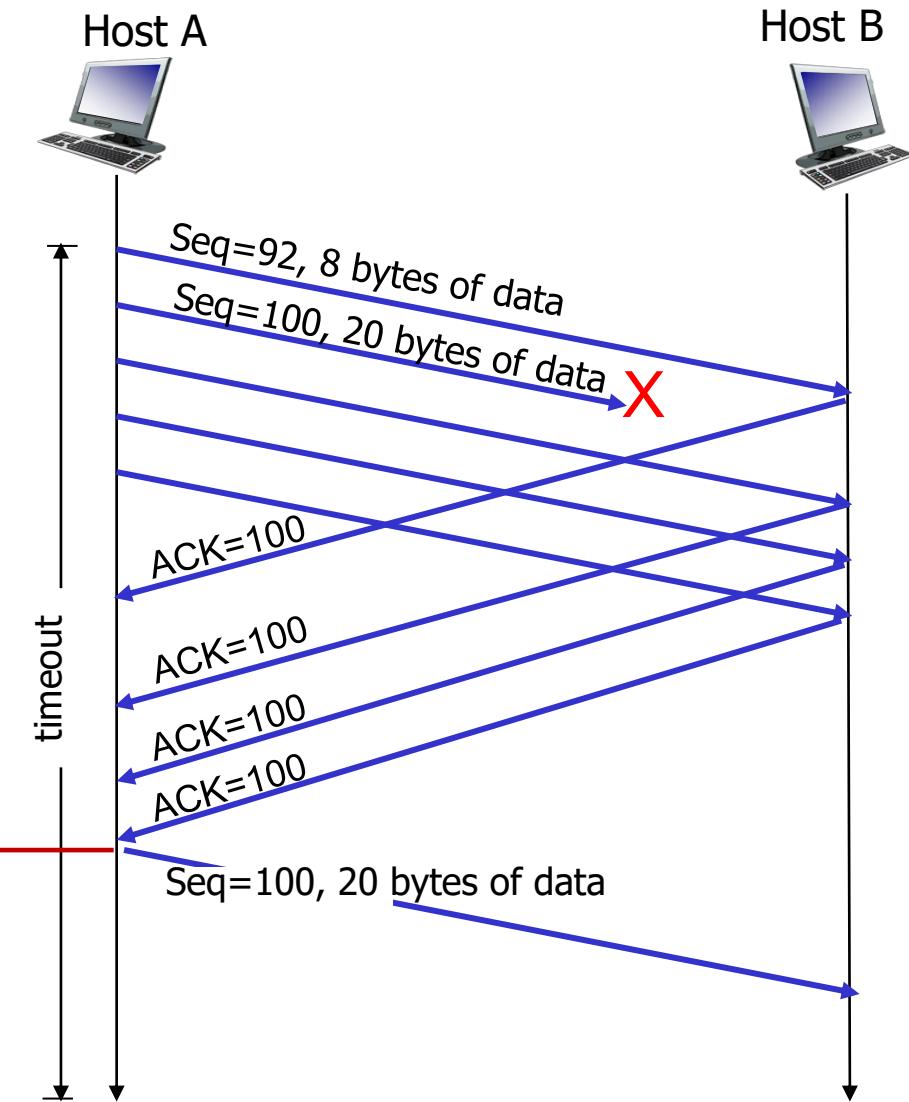
## *TCP fast retransmit*

if sender receives 3 additional ACKs for same data (“triple duplicate ACKs”), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don’t wait for timeout



Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



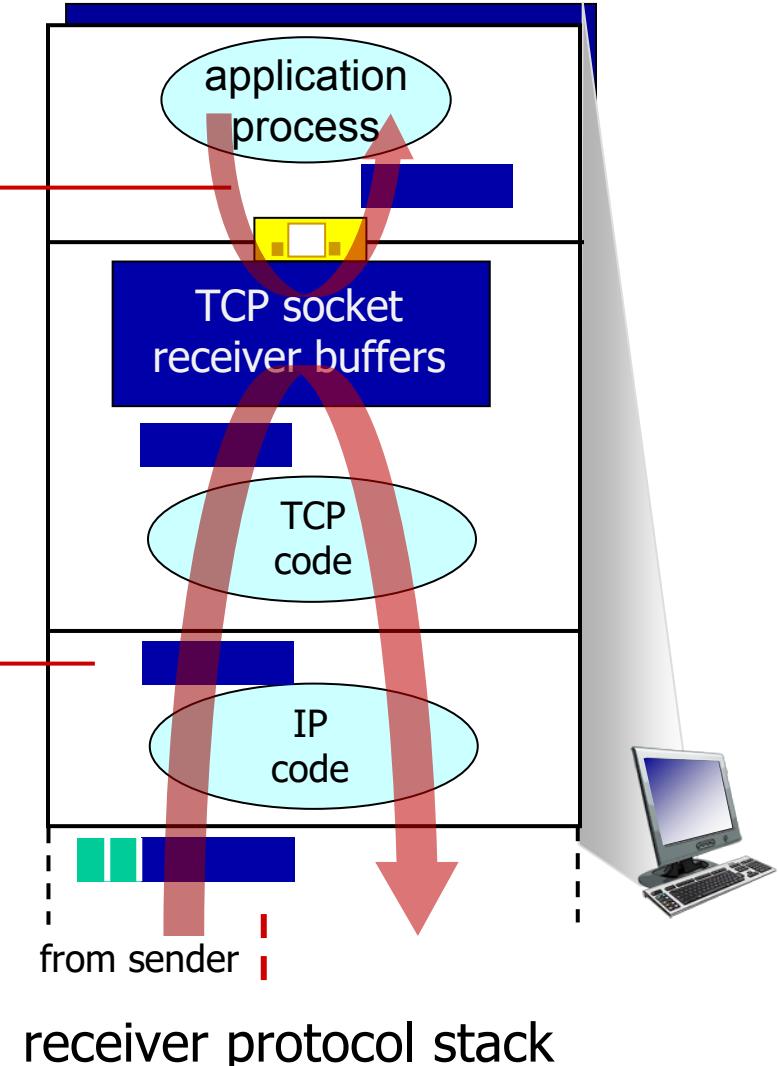
# TCP flow control

# (流量控制)

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

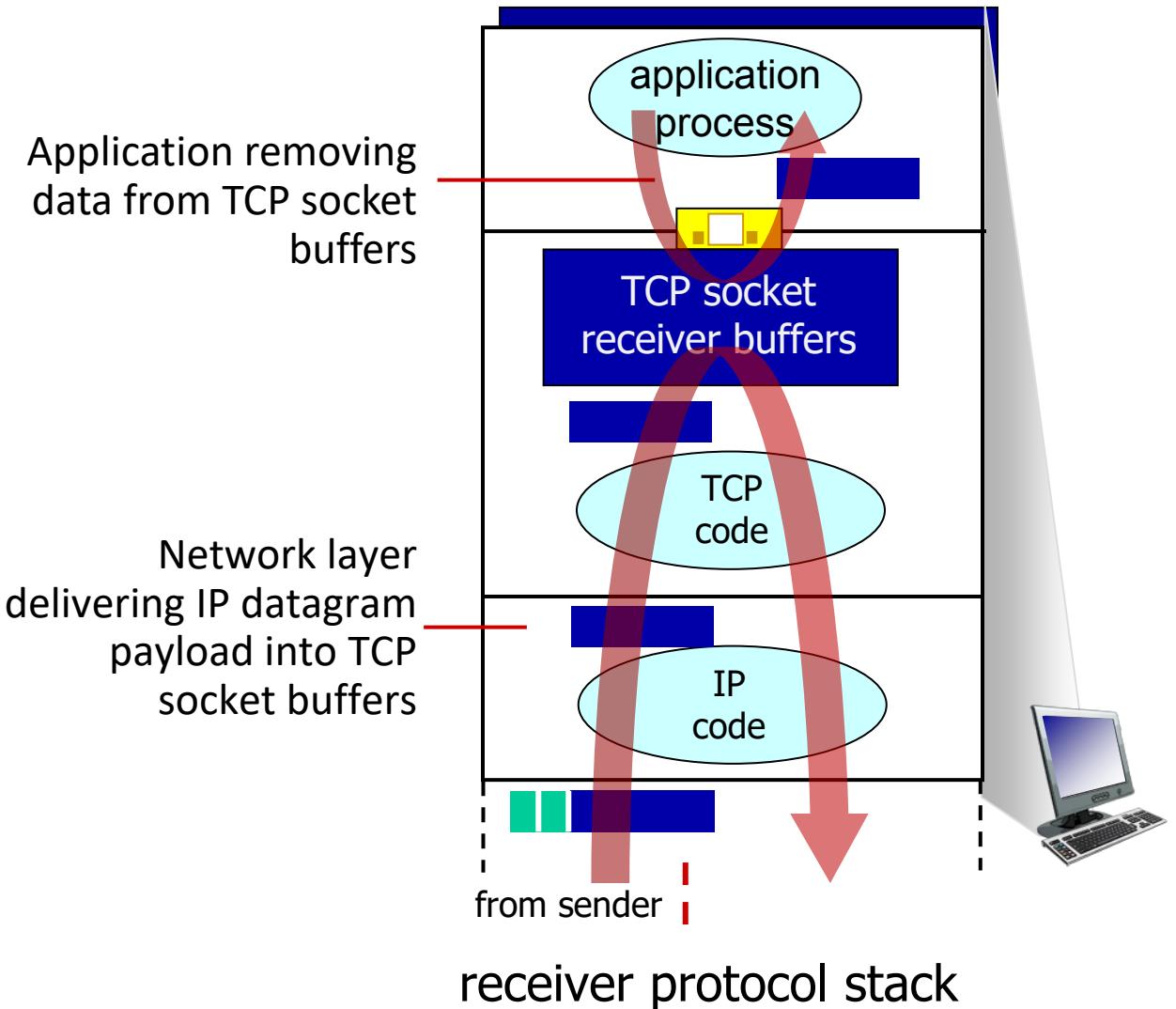
Application removing data from TCP socket buffers

Network layer delivering IP datagram payload into TCP socket buffers



# TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



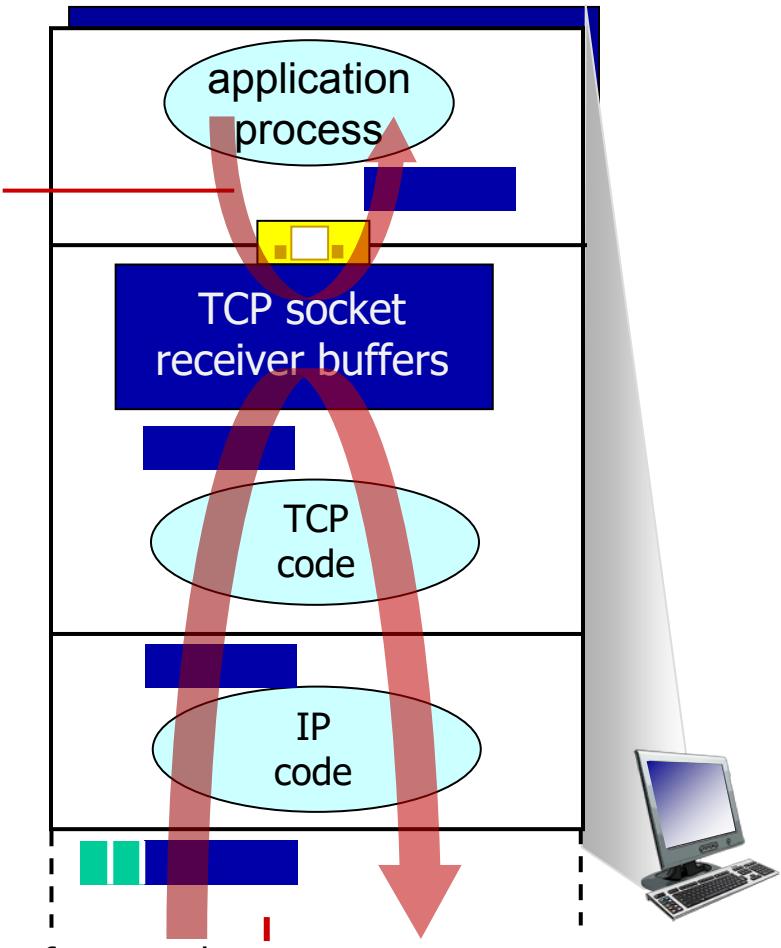
# TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

## flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

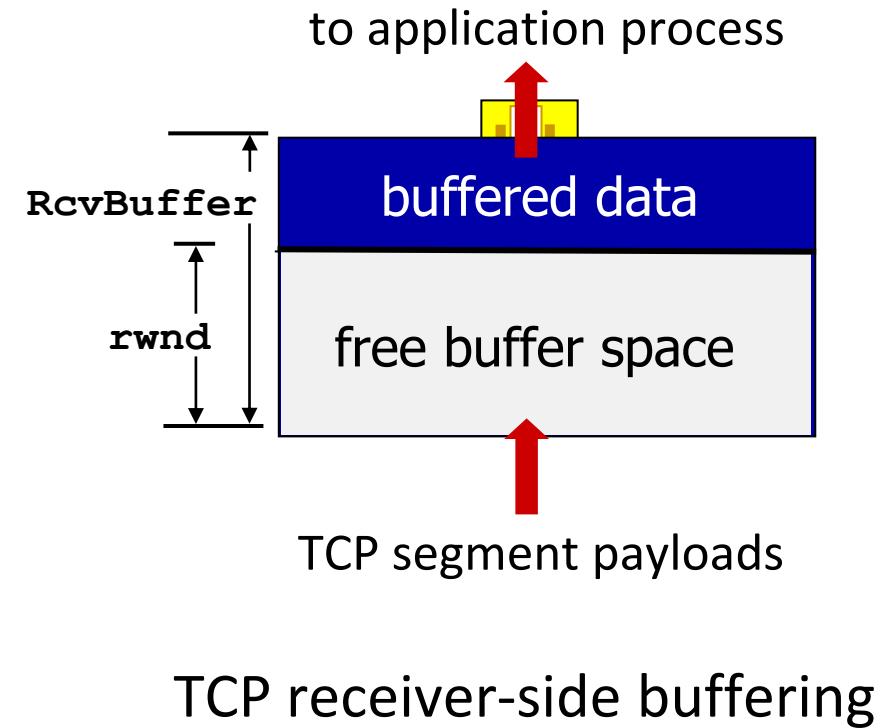
Application removing data from TCP socket buffers



receiver protocol stack

# TCP flow control

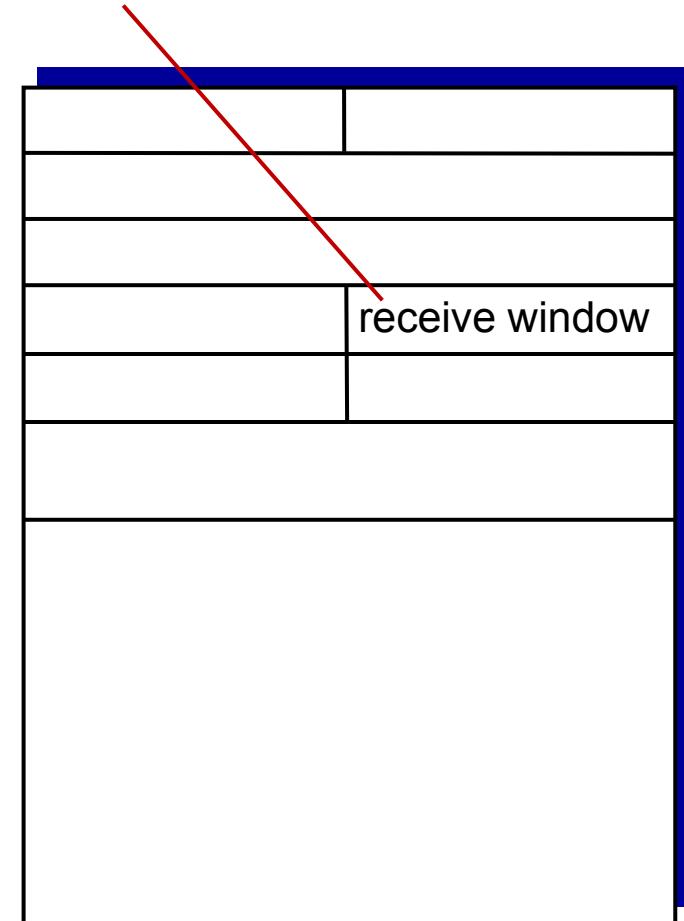
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow



# TCP flow control

- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept

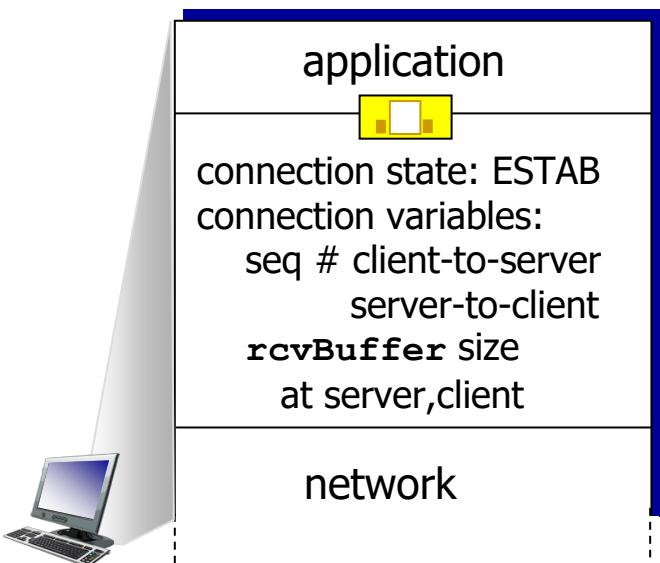


TCP segment format

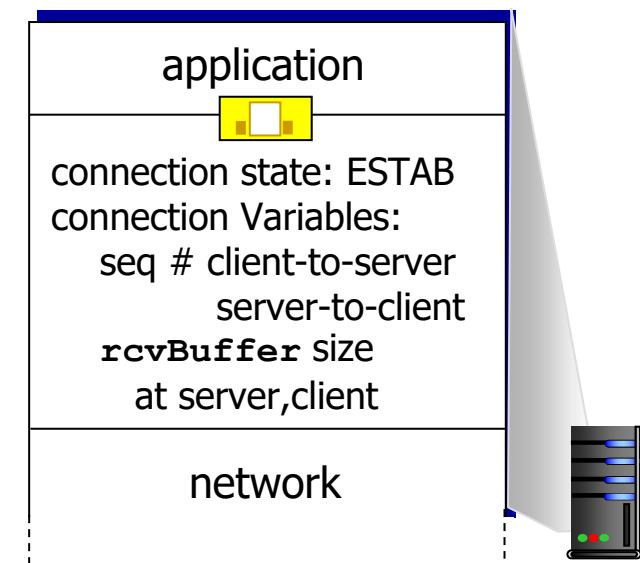
# TCP connection management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)



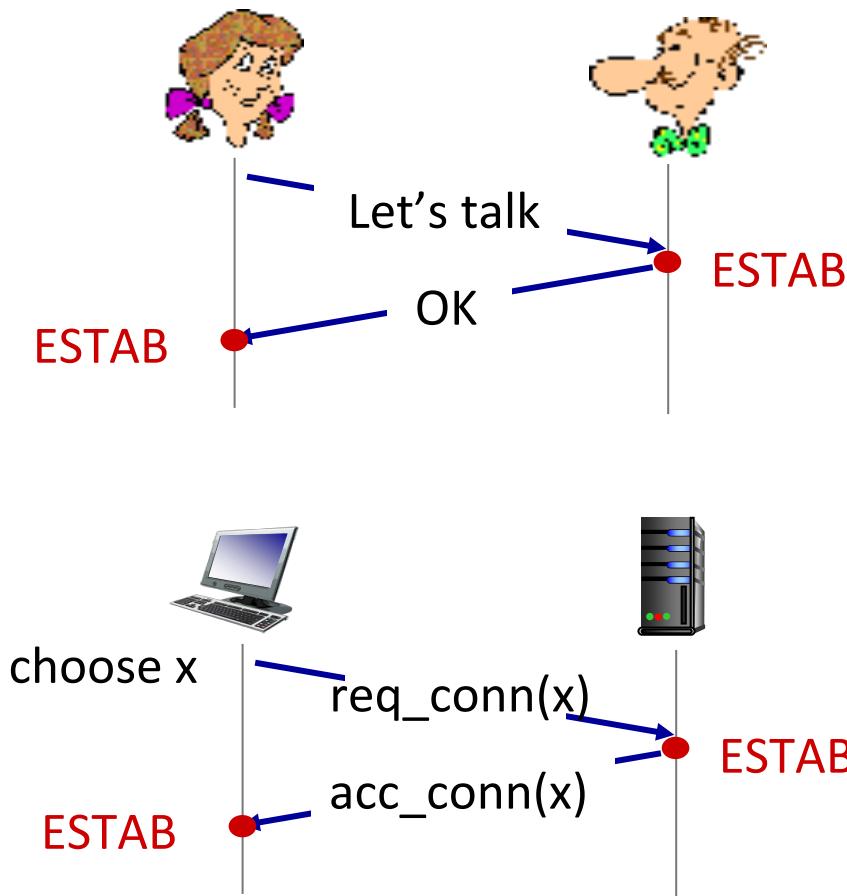
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



```
Socket connectionSocket =  
    welcomeSocket.accept();
```

# Agreeing to establish a connection

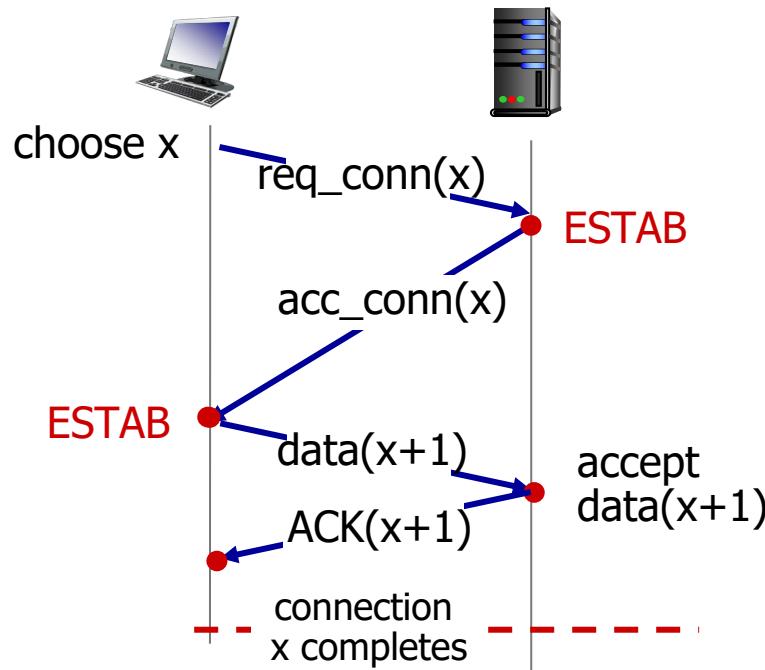
2-way handshake:



*Q:* will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g.  $\text{req\_conn}(x)$ ) due to message loss
- message reordering
- can't “see” other side

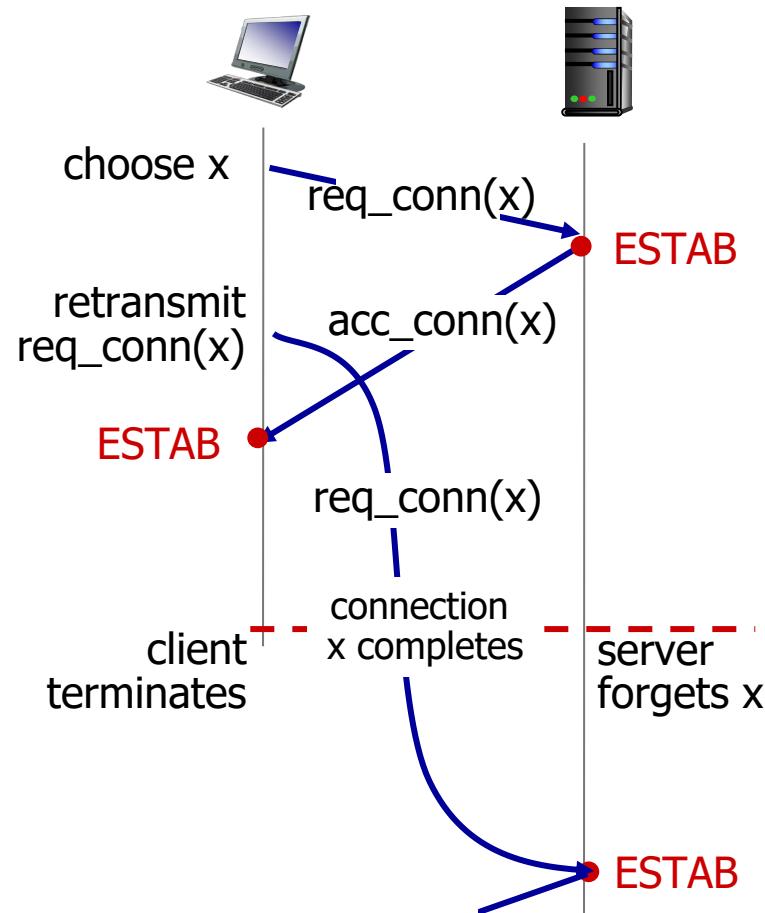
# 2-way handshake scenarios



No problem!

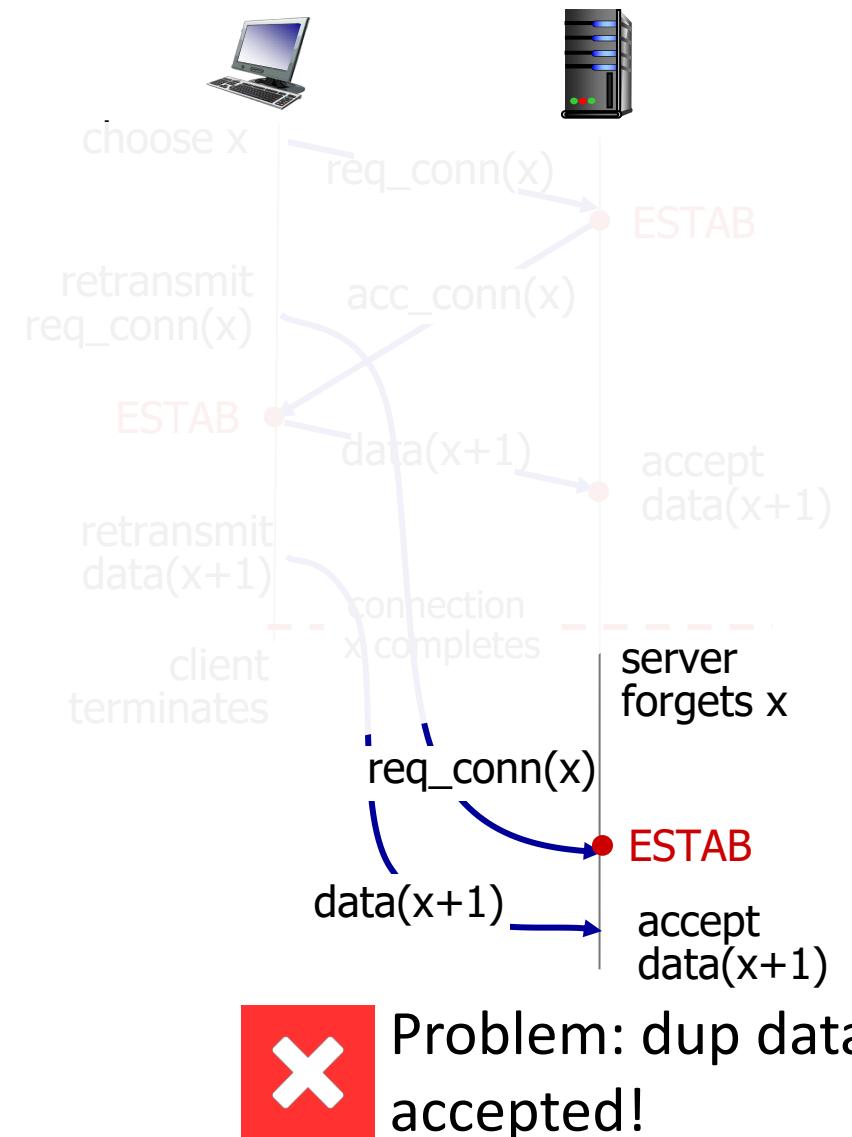


# 2-way handshake scenarios

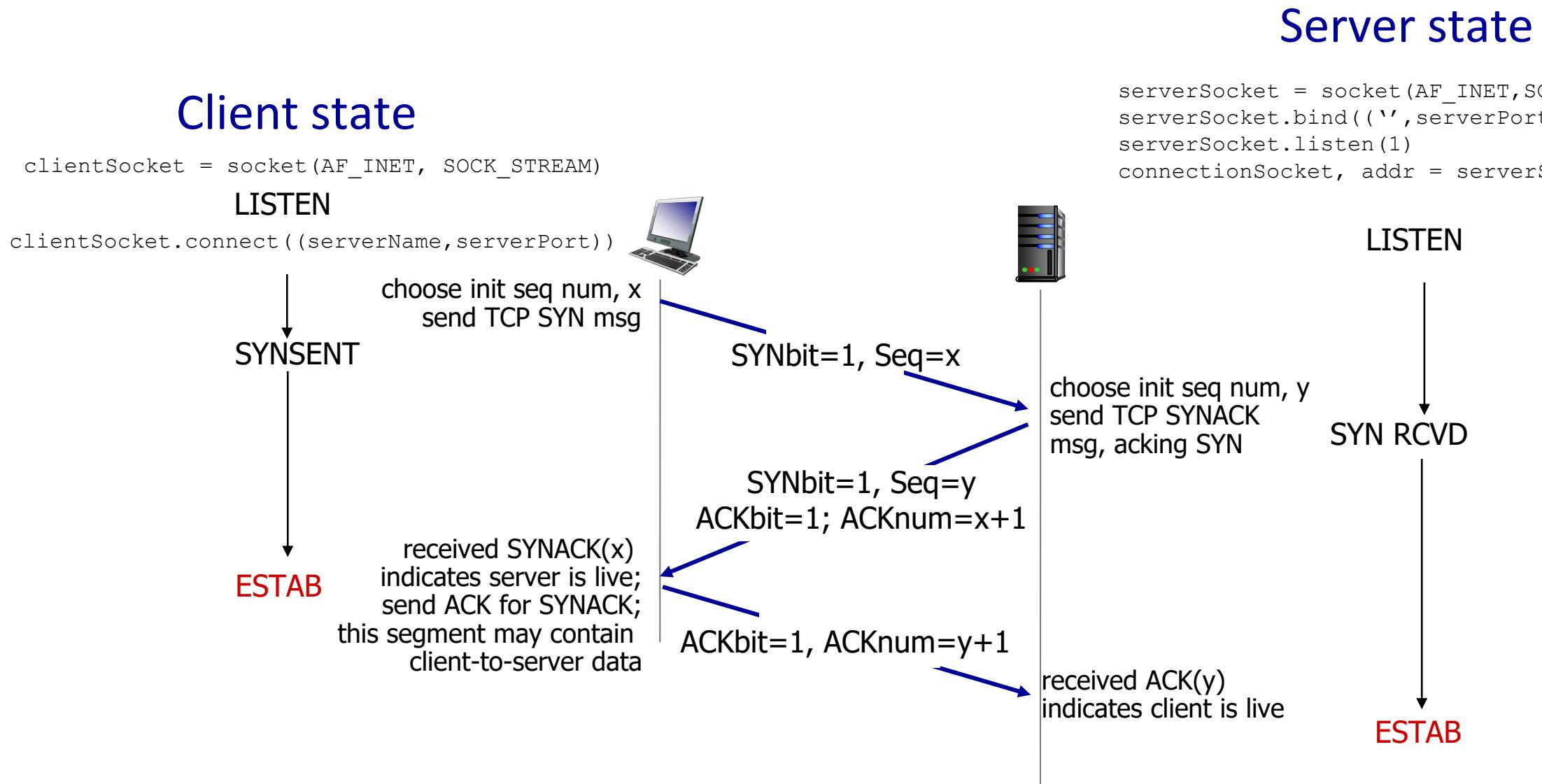


Problem: half open  
connection! (no client)

# 2-way handshake scenarios



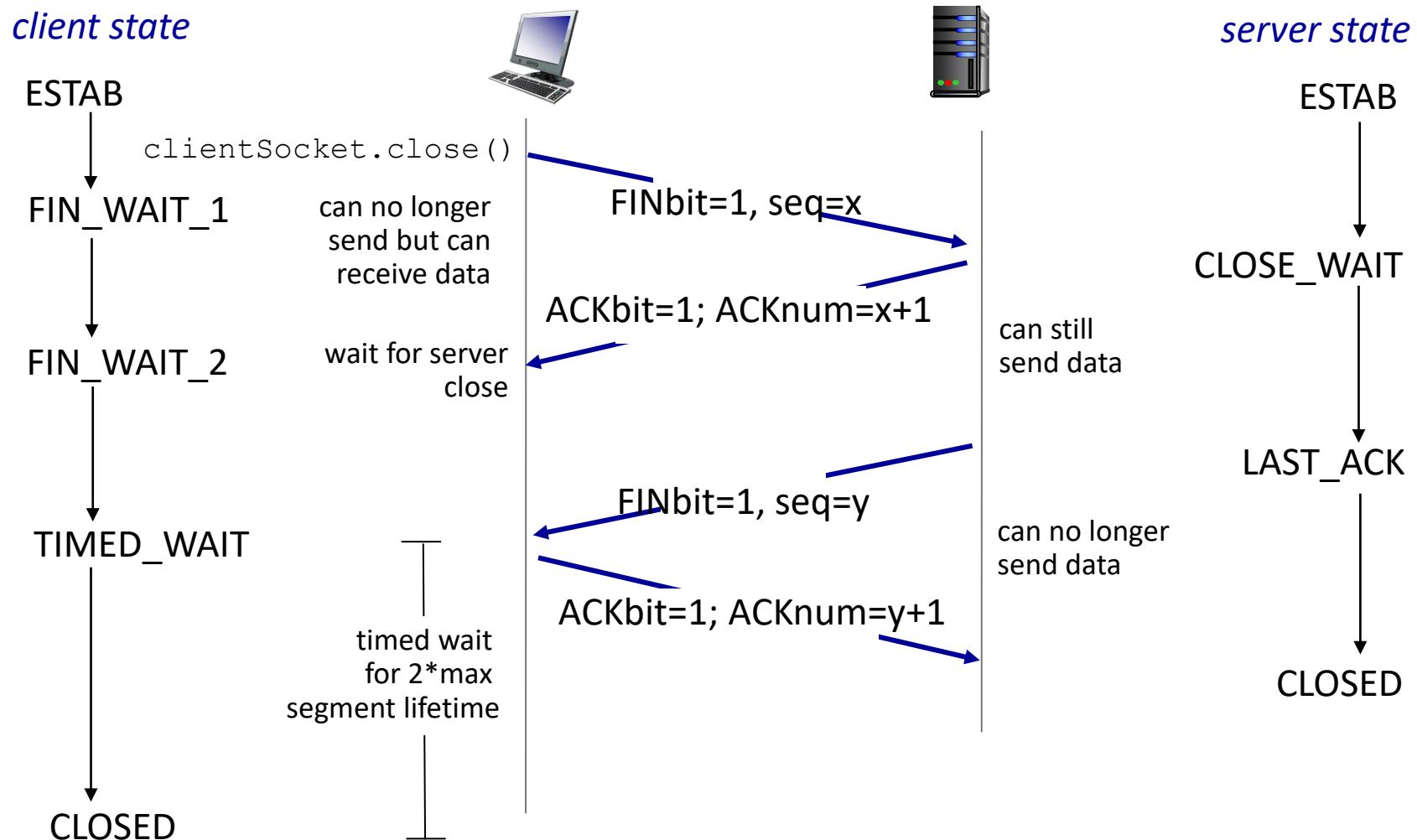
# TCP 3-way handshake (三次握手)



# A human 3-way handshake protocol



# TCP: closing a connection



# Closing a TCP connection

- 4-way handshake (四次挥手)
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality



# Principles of congestion control

## Congestion:

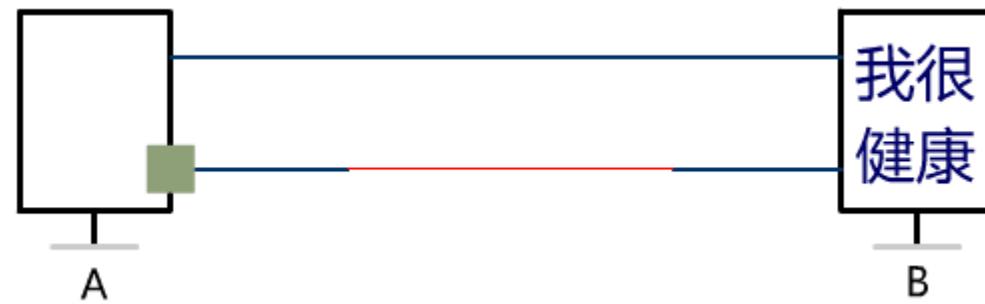
- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!



**congestion control:**  
too many senders,  
sending too fast

**flow control:** one sender  
too fast for one receiver

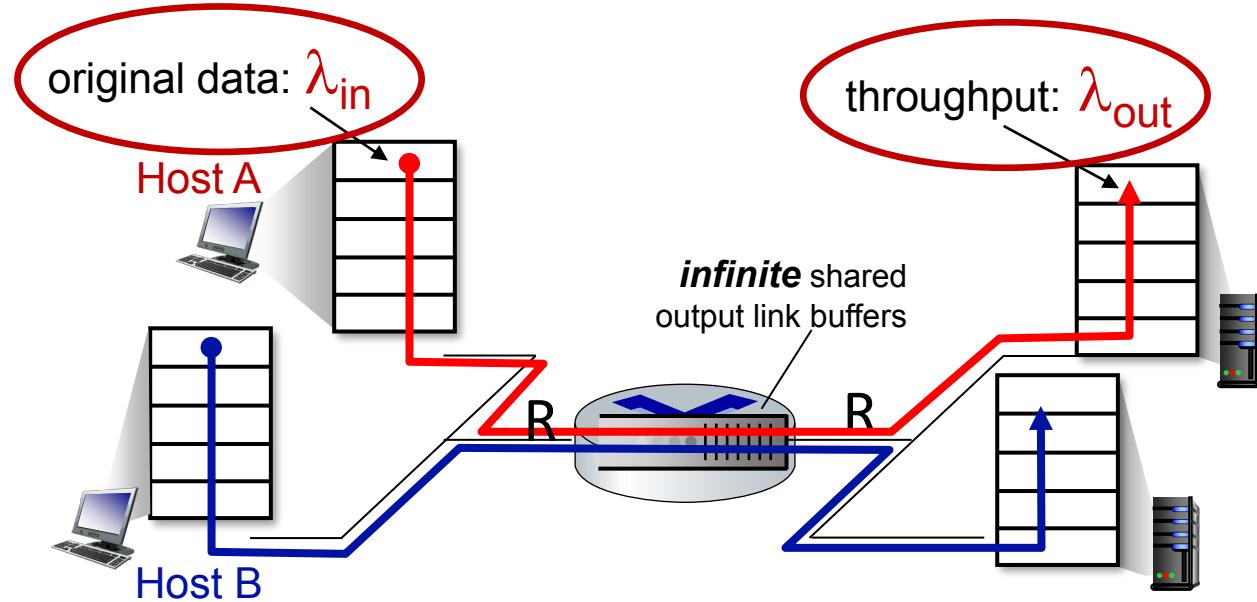
# Principles of congestion control



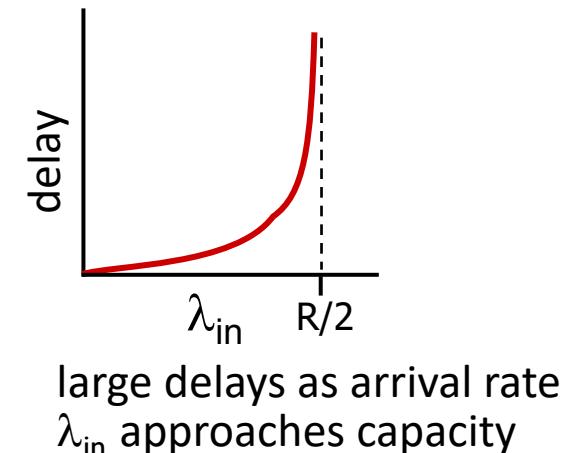
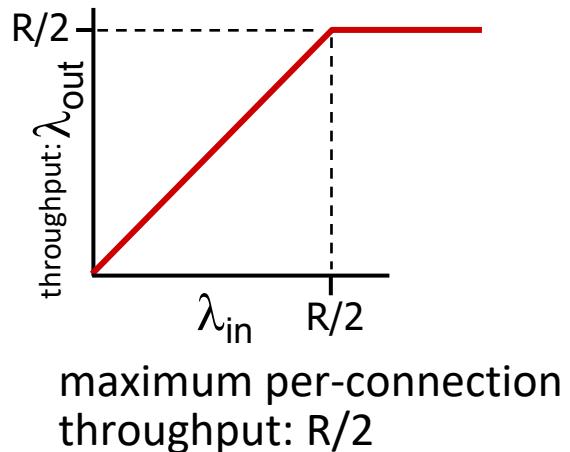
# Causes/costs of congestion: scenario 1

Simplest scenario:

- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed

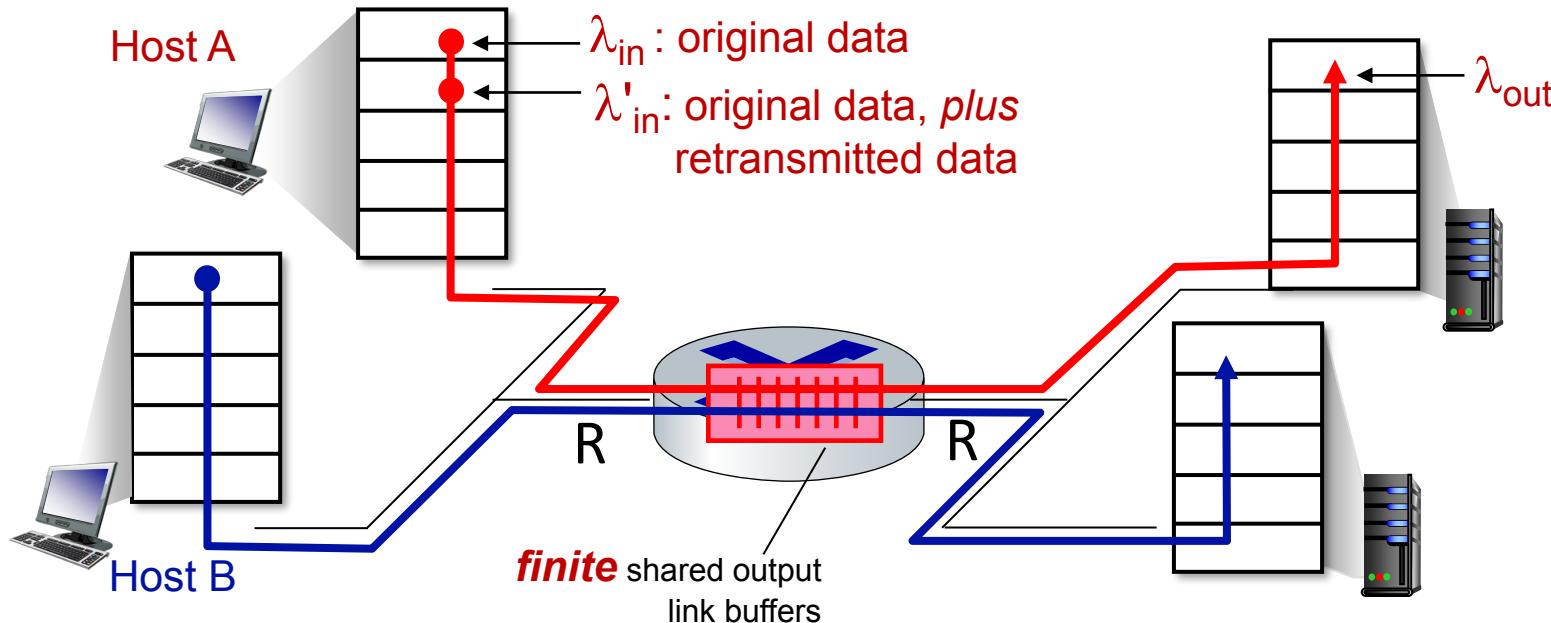


**Q:** What happens as arrival rate  $\lambda_{in}$  approaches  $R/2$ ?



# Causes/costs of congestion: scenario 2

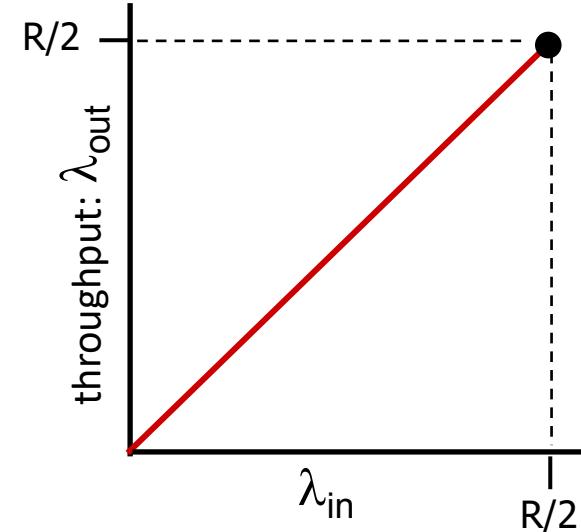
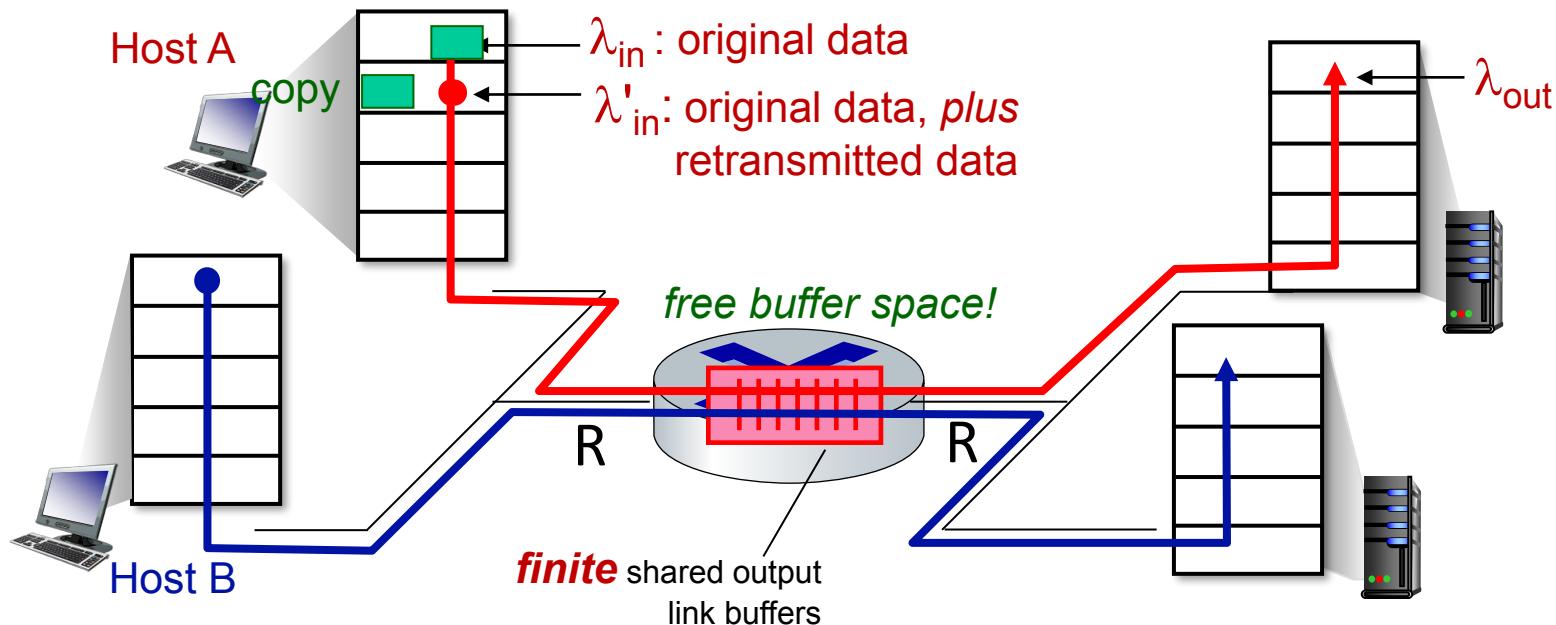
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

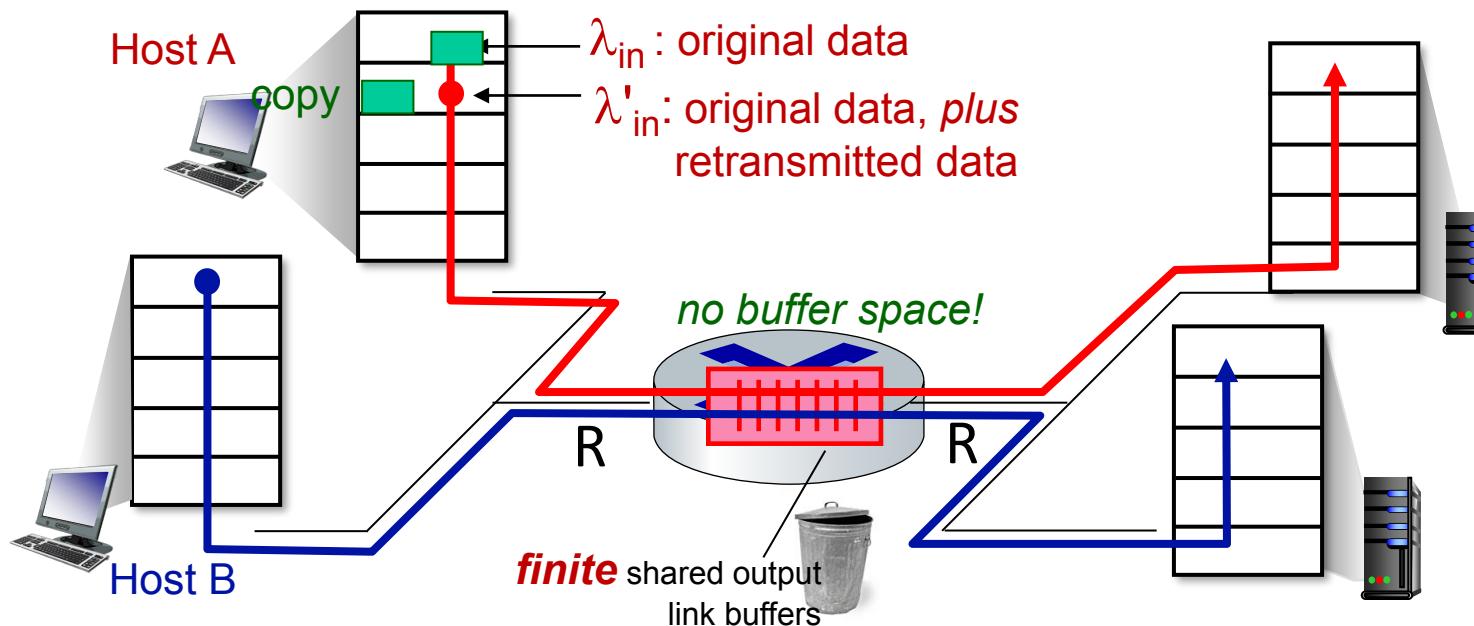
- sender sends only when router buffers available



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

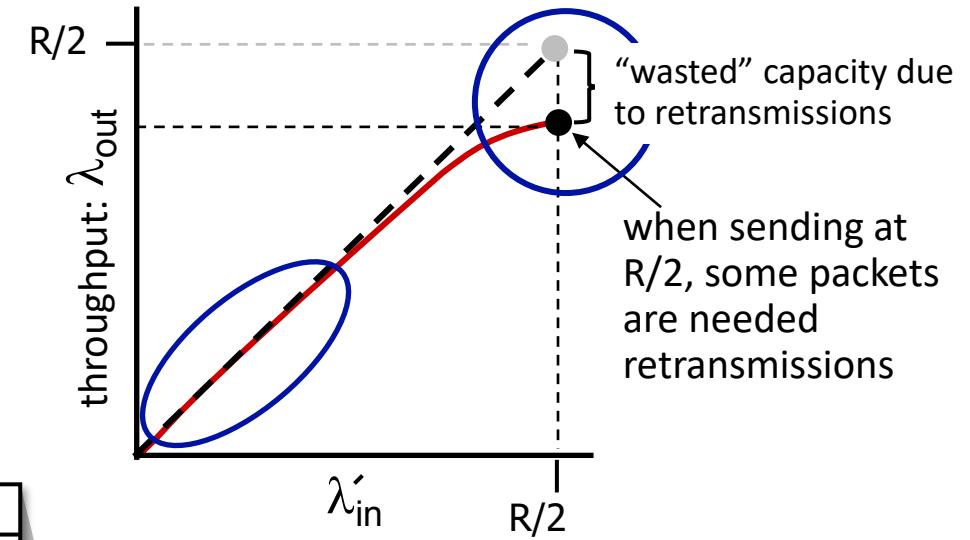
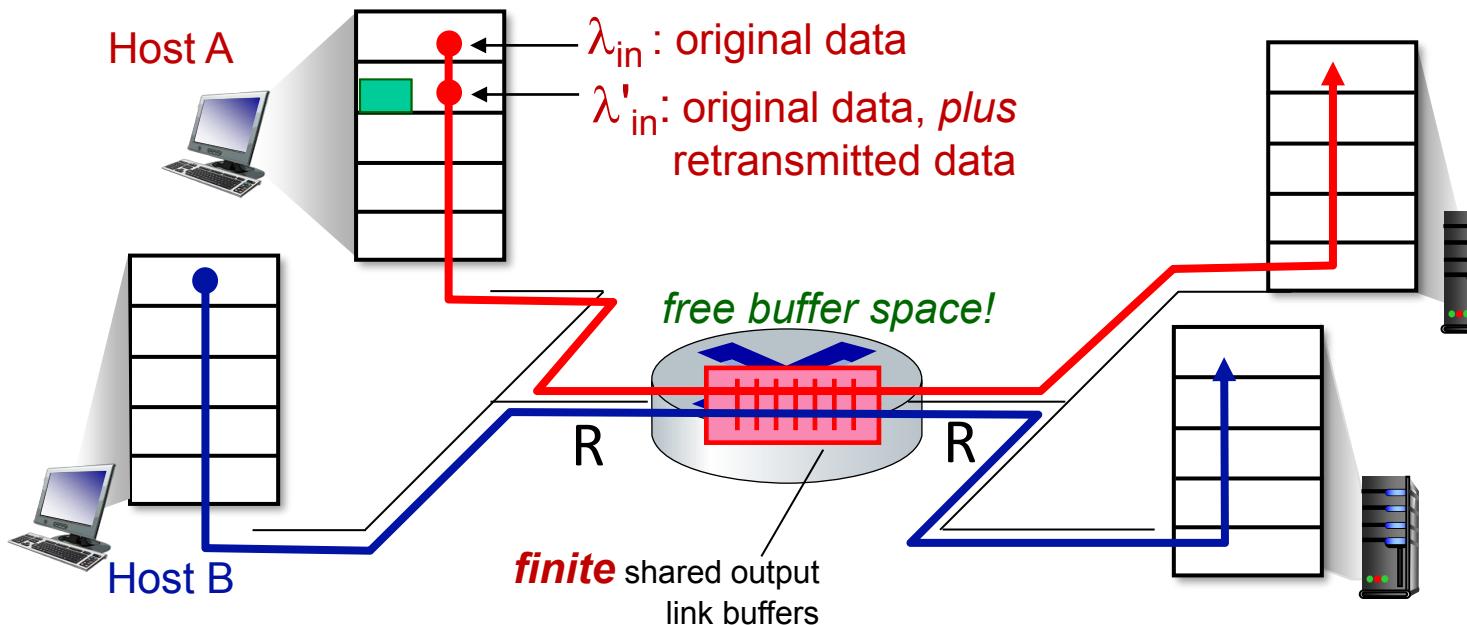
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

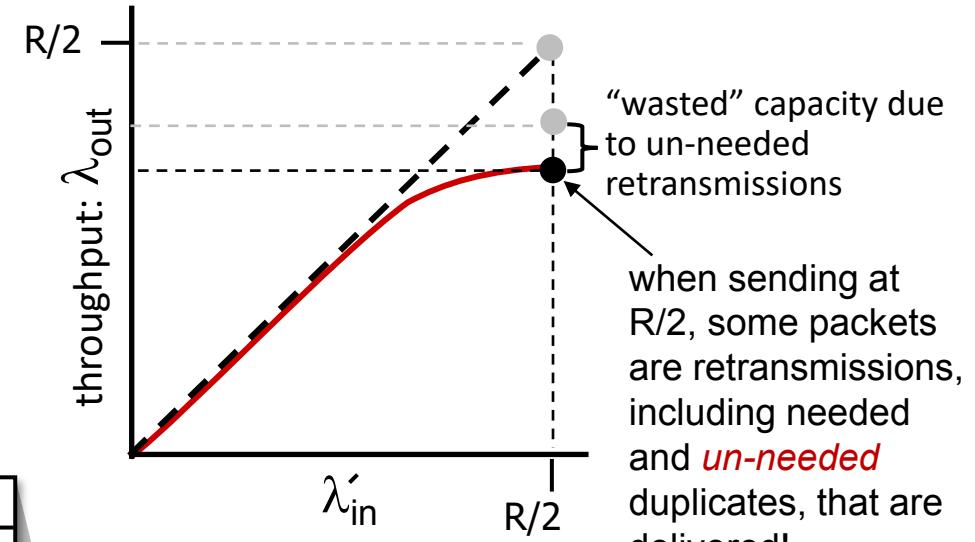
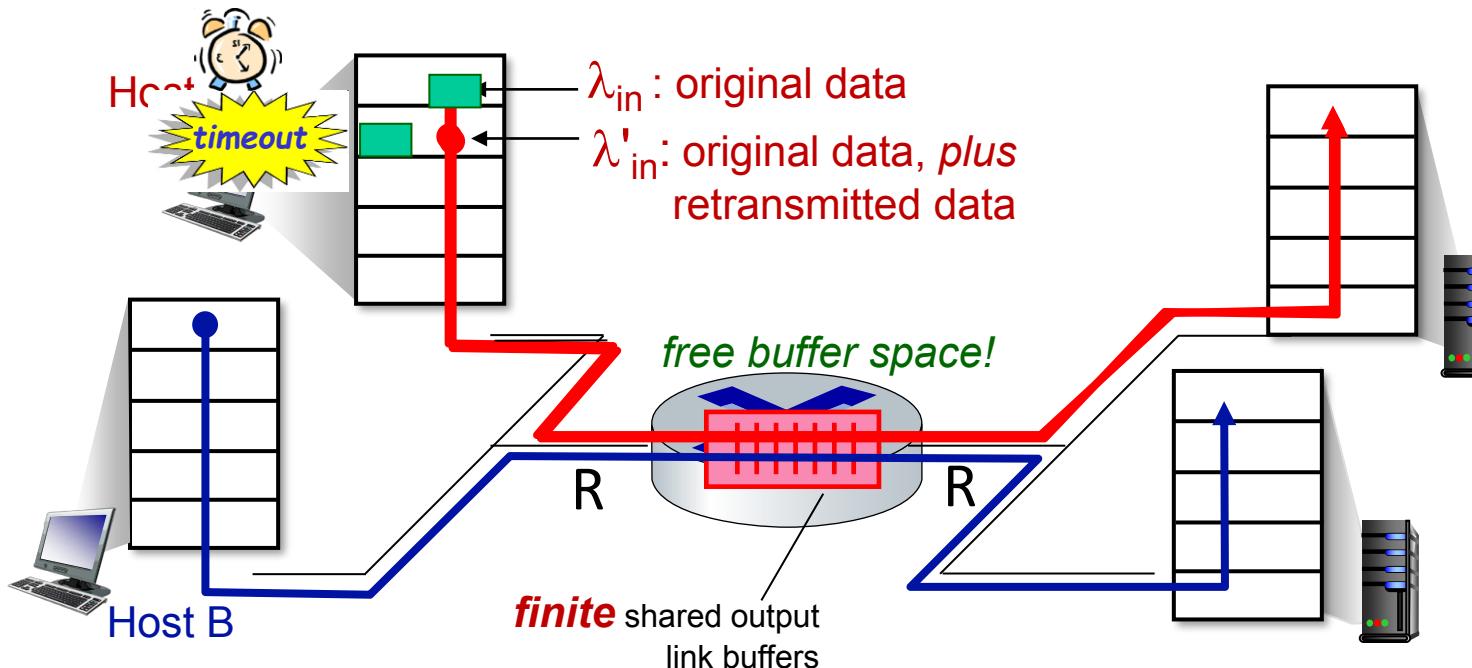
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



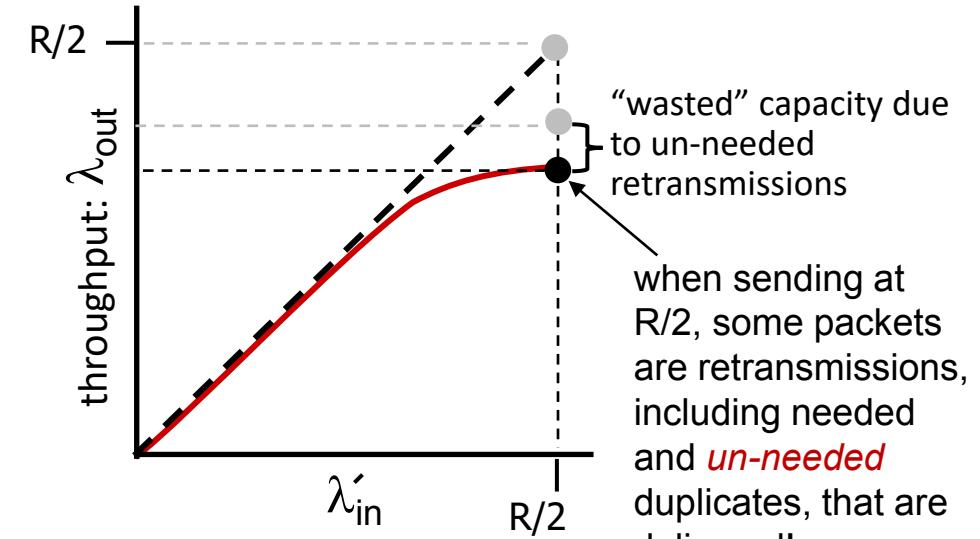
# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered

## “costs” of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

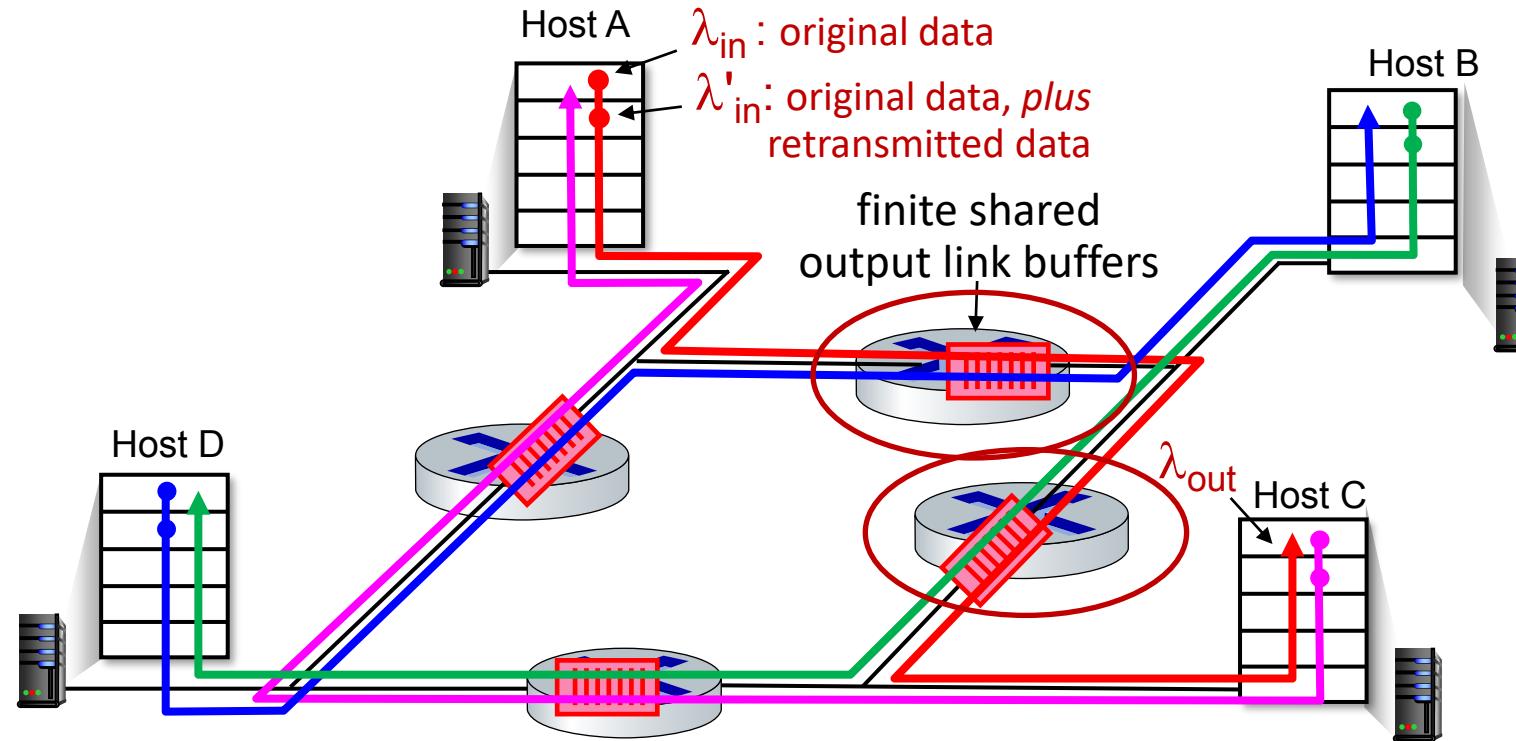


# Causes/costs of congestion: scenario 3

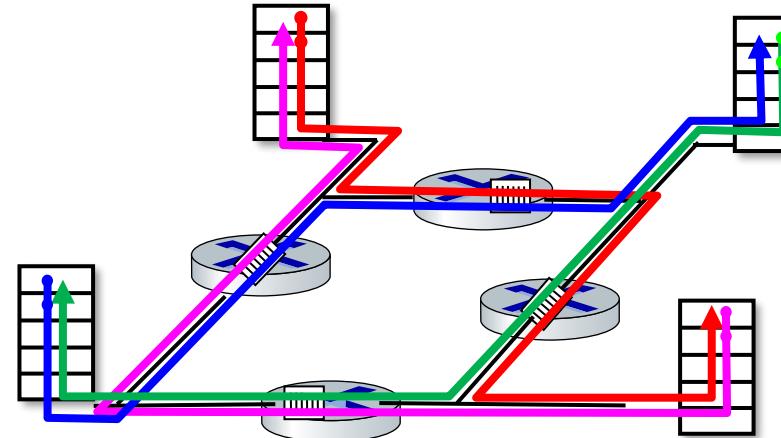
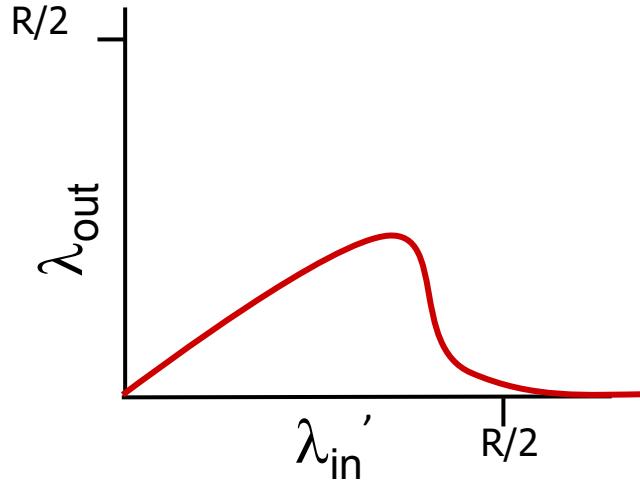
- four senders
- multi-hop paths
- timeout/retransmit

**Q:** what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

**A:** as red  $\lambda'_{in}$  increases, all arriving blue pkts at upper queue are dropped, blue throughput  $\rightarrow 0$



# Causes/costs of congestion: scenario 3

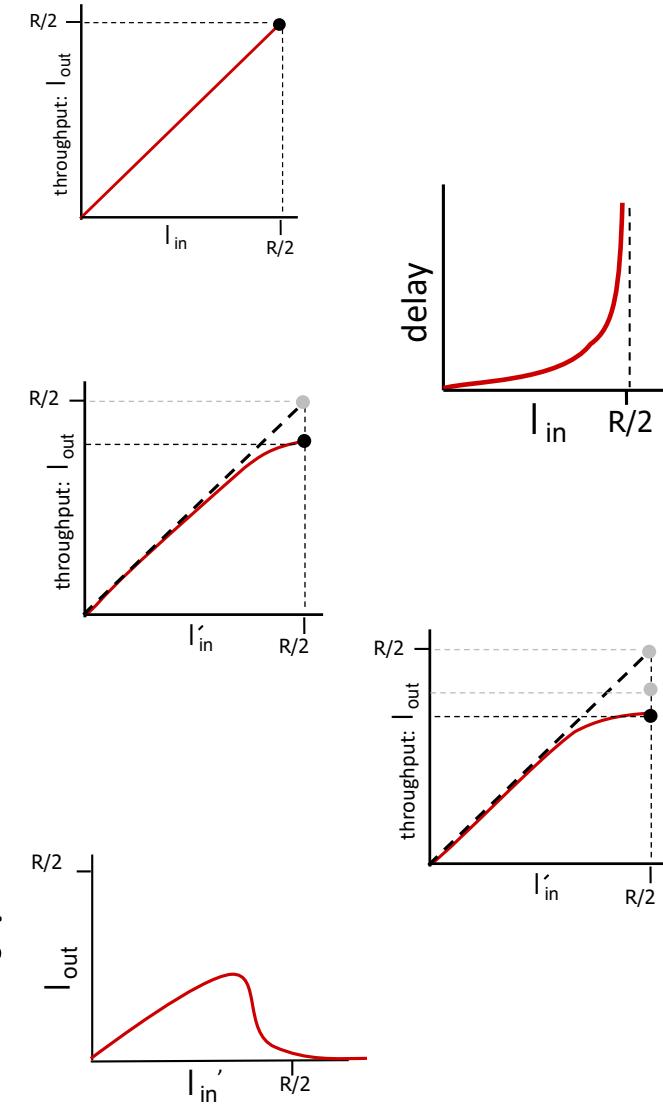


another “cost” of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

# Causes/costs of congestion: insights

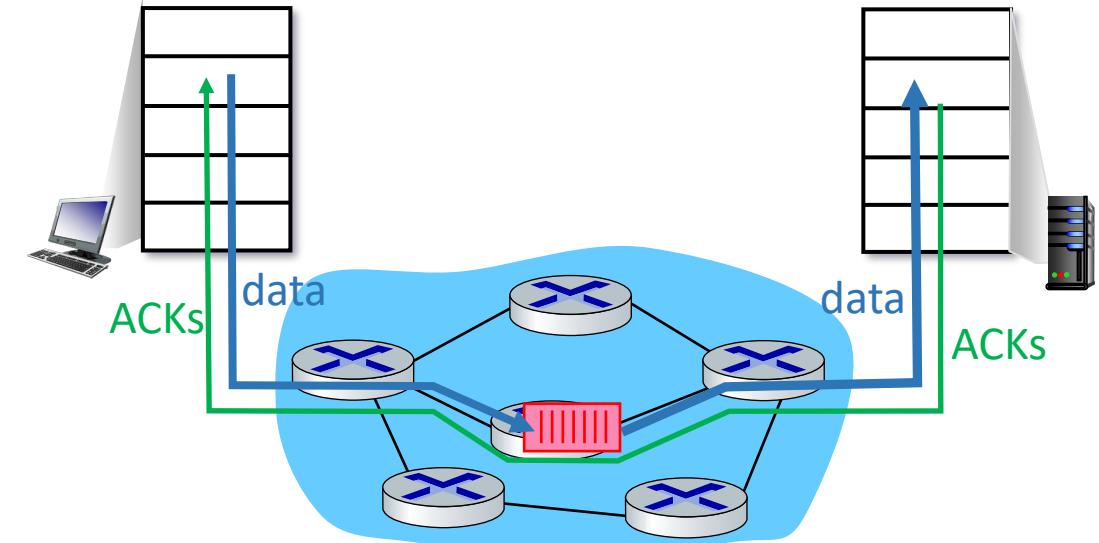
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



# Approaches towards congestion control

## End-end congestion control:

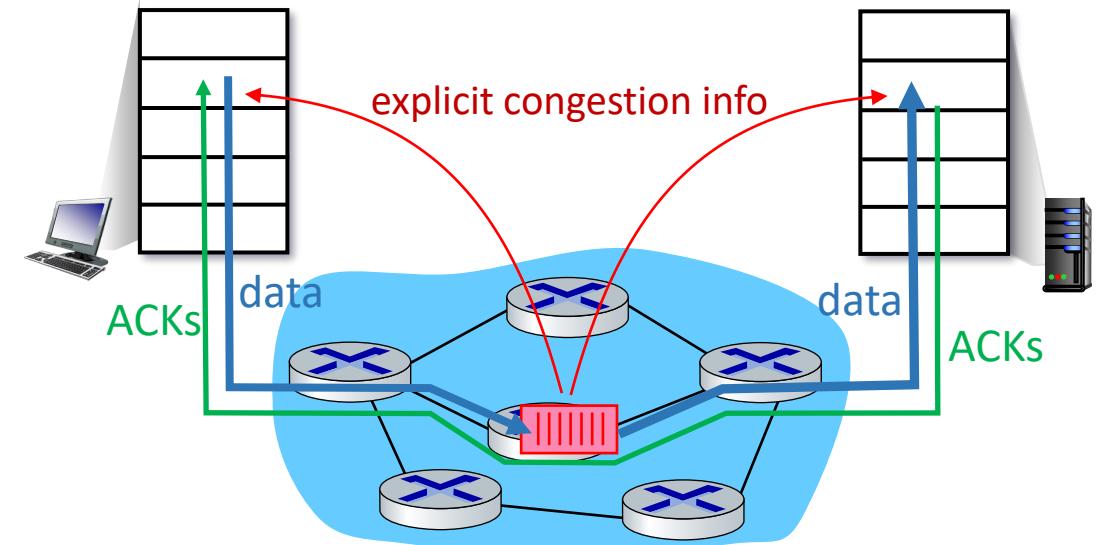
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



# Approaches towards congestion control

## Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- TCP ECN, ATM, DECbit protocols



# Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- **TCP congestion control**
- Evolution of transport-layer functionality



# TCP congestion control: AIMD

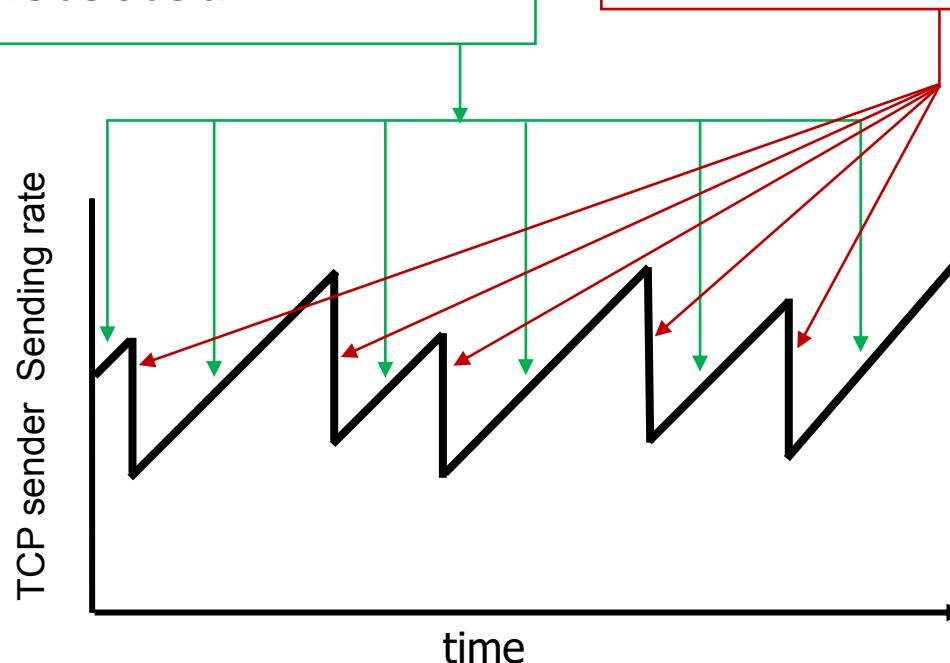
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

## Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

## Multiplicative Decrease

cut sending rate in half at each loss event



**AIMD** sawtooth behavior: *probing* for bandwidth

# TCP AIMD: more

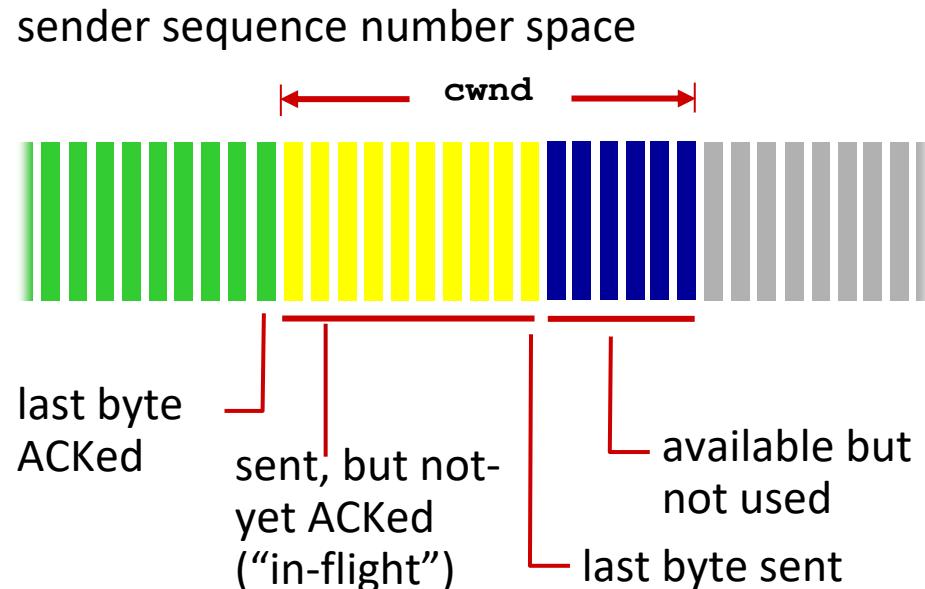
*Multiplicative decrease* detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

# TCP congestion control: details



TCP sending behavior:

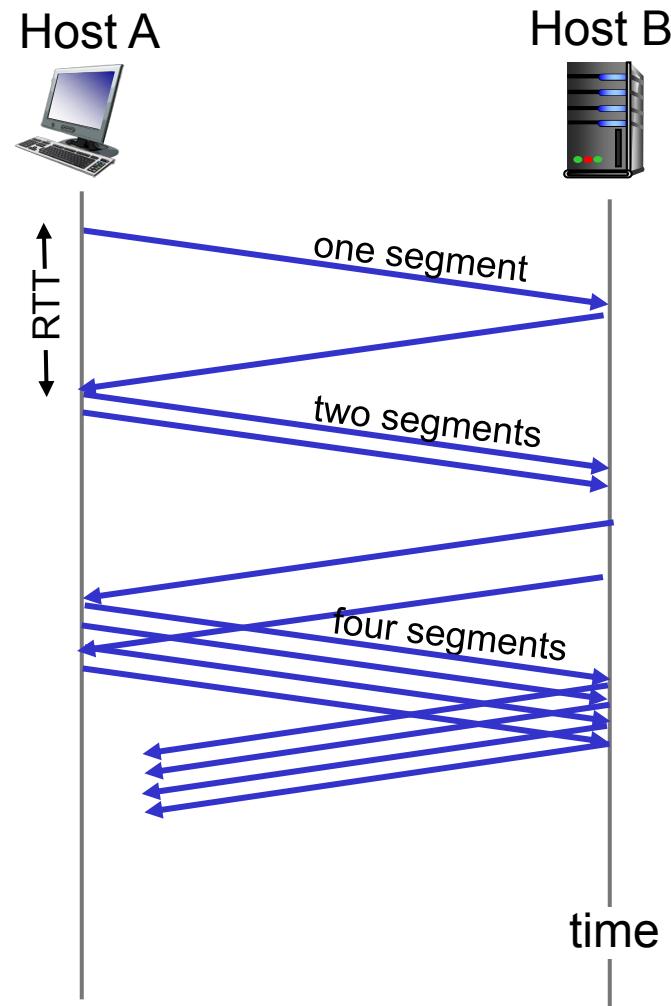
- *roughly*: send  $cwnd$  bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{cwnd}{RTT} \text{ bytes/sec}$$

- TCP sender limits transmission:  $\text{LastByteSent} - \text{LastByteAcked} \leq cwnd$
- $cwnd$  is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

# TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



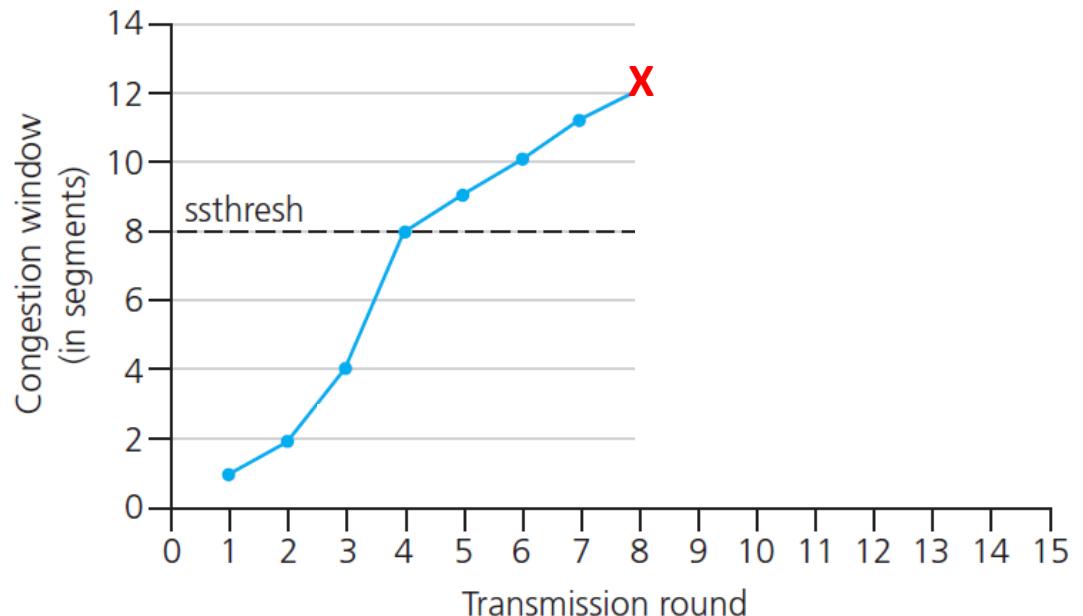
# TCP: from slow start to congestion avoidance

**Q:** when should the exponential increase switch to linear?

**A:** when **cwnd** gets to 1/2 of its value before timeout.

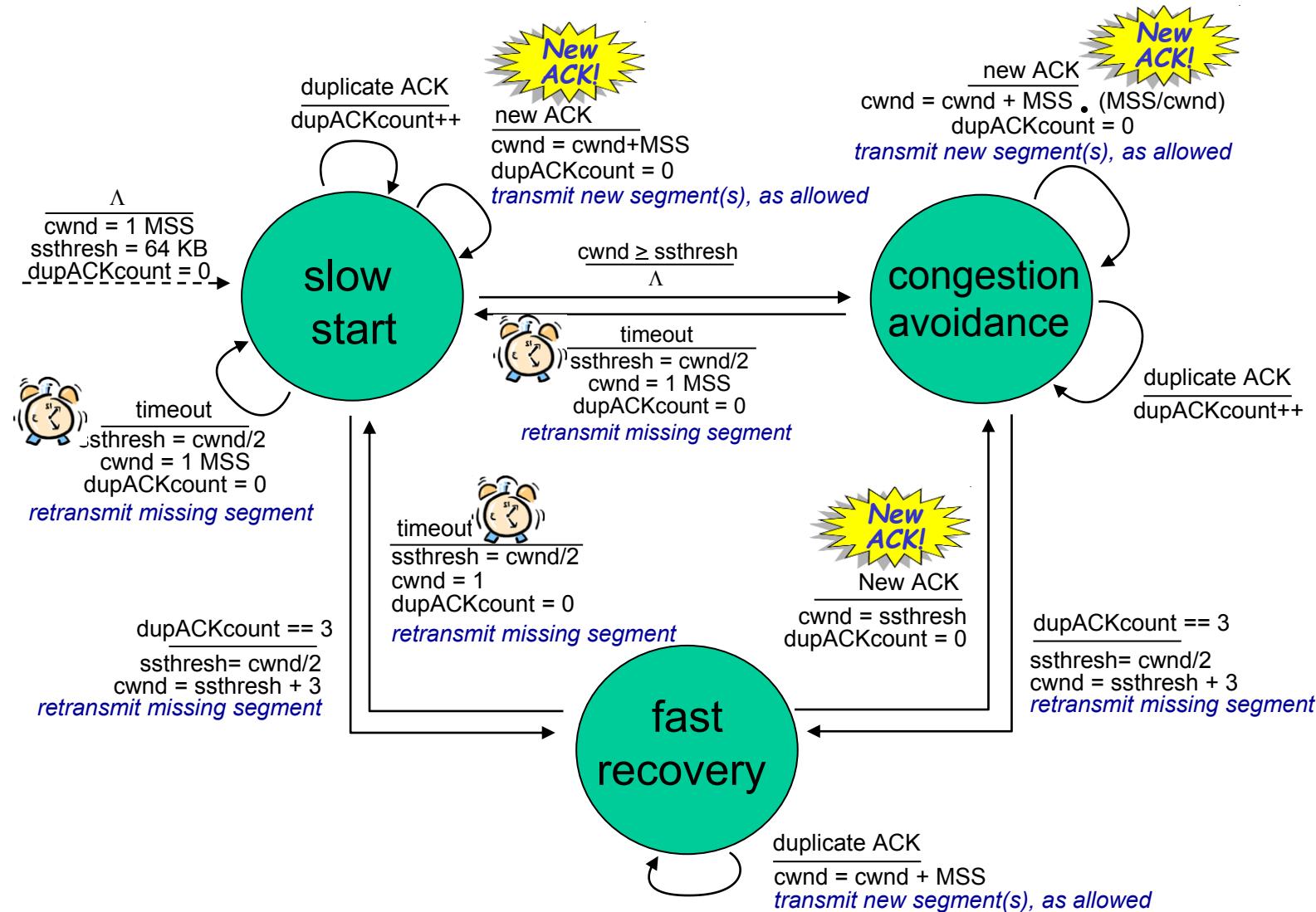
## Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



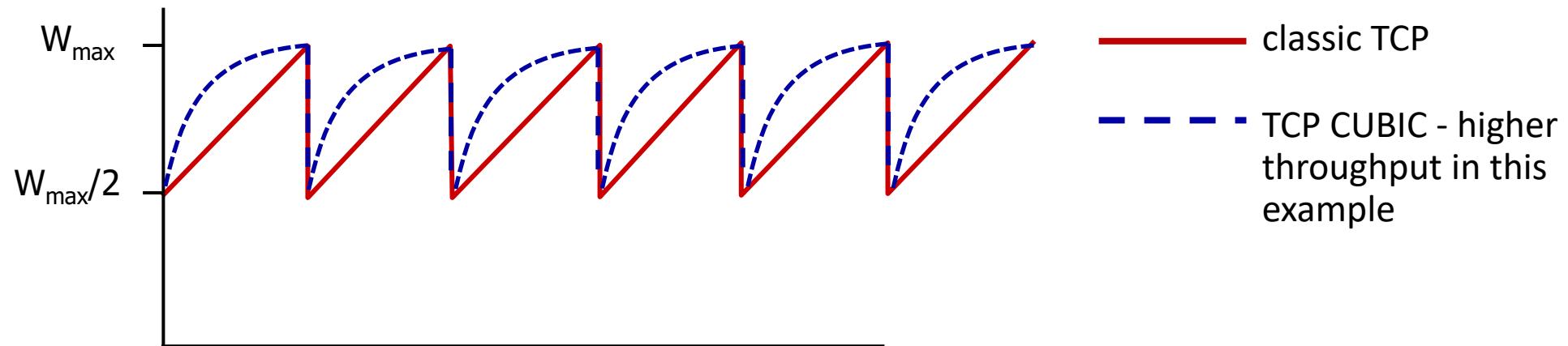
\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# Summary: TCP congestion control



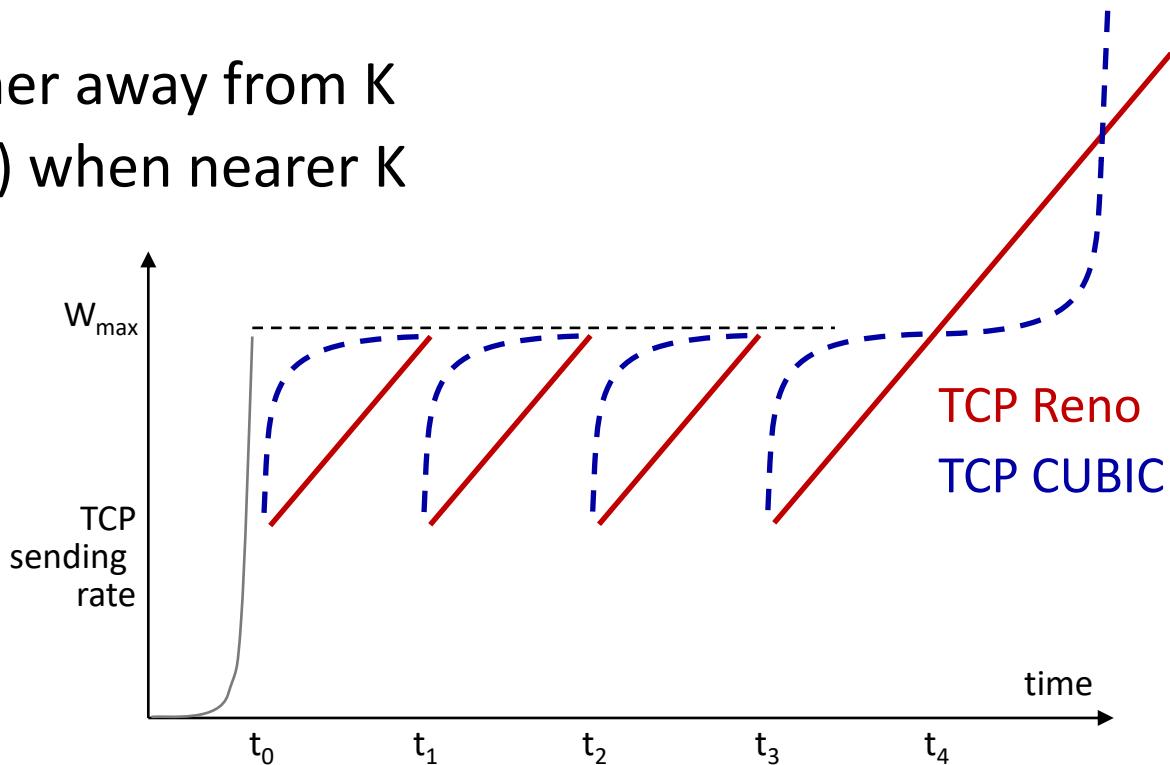
# TCP CUBIC

- Is there a better way than AIMD to “probe” for usable bandwidth?
- Insight/intuition:
  - $W_{\max}$ : sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn’t changed much
  - after cutting rate/window in half on loss, initially ramp to  $W_{\max}$  *faster*, but then approach  $W_{\max}$  more *slowly*



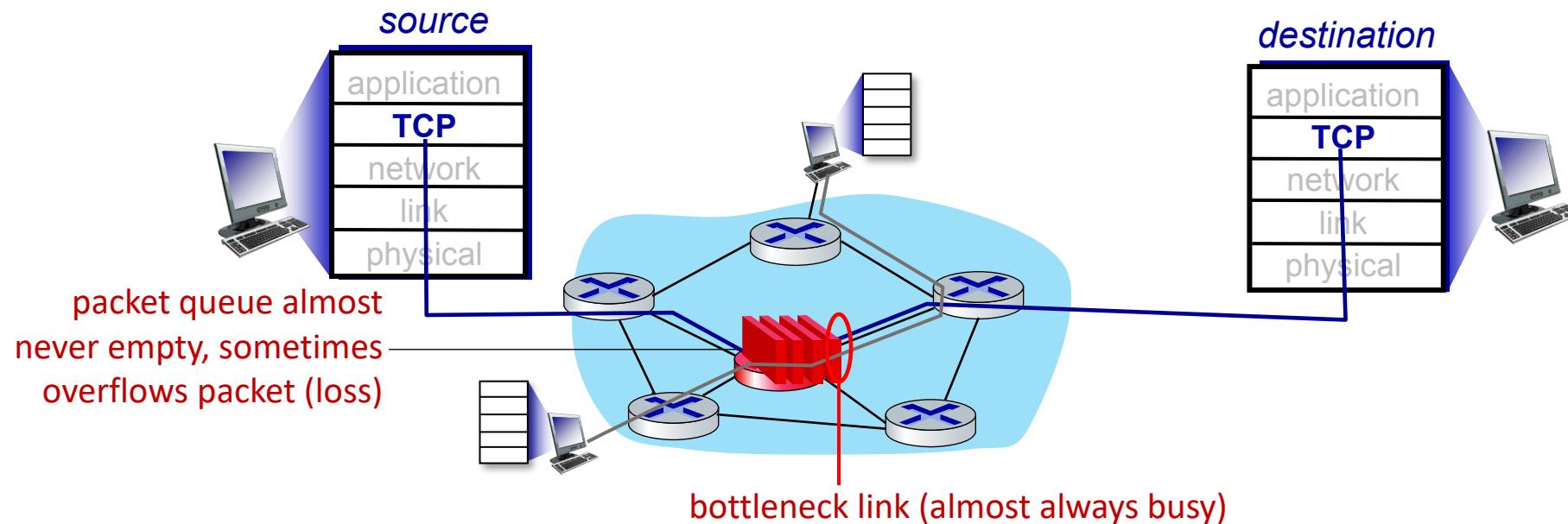
# TCP CUBIC

- K: point in time when TCP window size will reach  $W_{\max}$ 
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



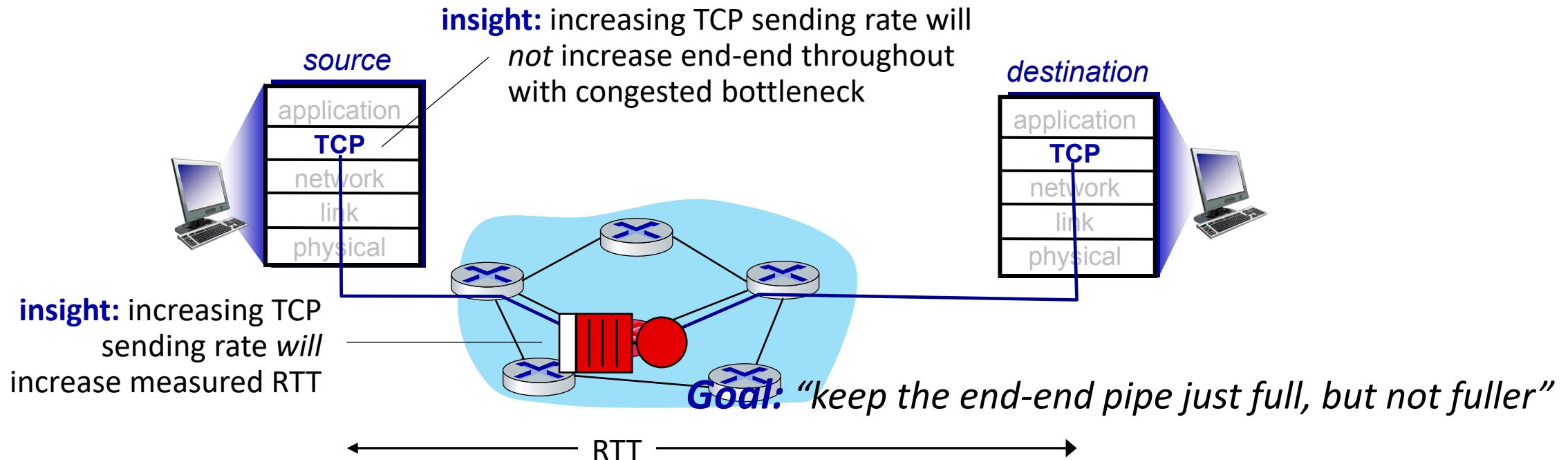
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



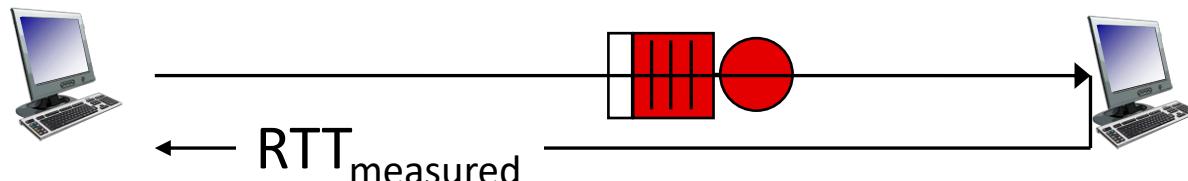
# TCP and the congested “bottleneck link”

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



# Delay-based TCP congestion control

Keeping sender-to-receiver pipe “just full enough, but no fuller”: keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{\text{RTT}_{\text{measured}}}$$

## Delay-based approach:

- $\text{RTT}_{\min}$  - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window  $cwnd$  is  $cwnd/\text{RTT}_{\min}$ 
  - if measured throughput “very close” to uncongested throughput  
increase  $cwnd$  linearly /\* since path not congested \*/
  - else if measured throughput “far below” uncongested throughput  
decrease  $cwnd$  linearly /\* since path is congested \*/

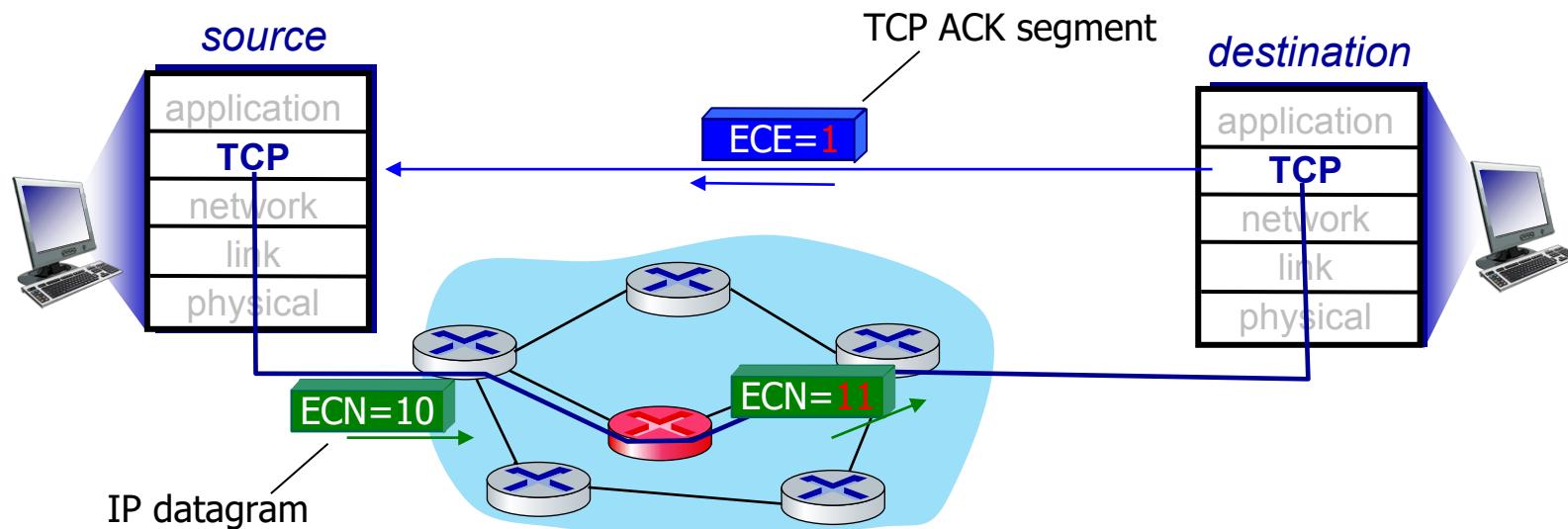
# Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughout (“keeping the just pipe full... ”) while keeping delay low (“...but not fuller”)
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google’s (internal) backbone network

# Explicit congestion notification (ECN)

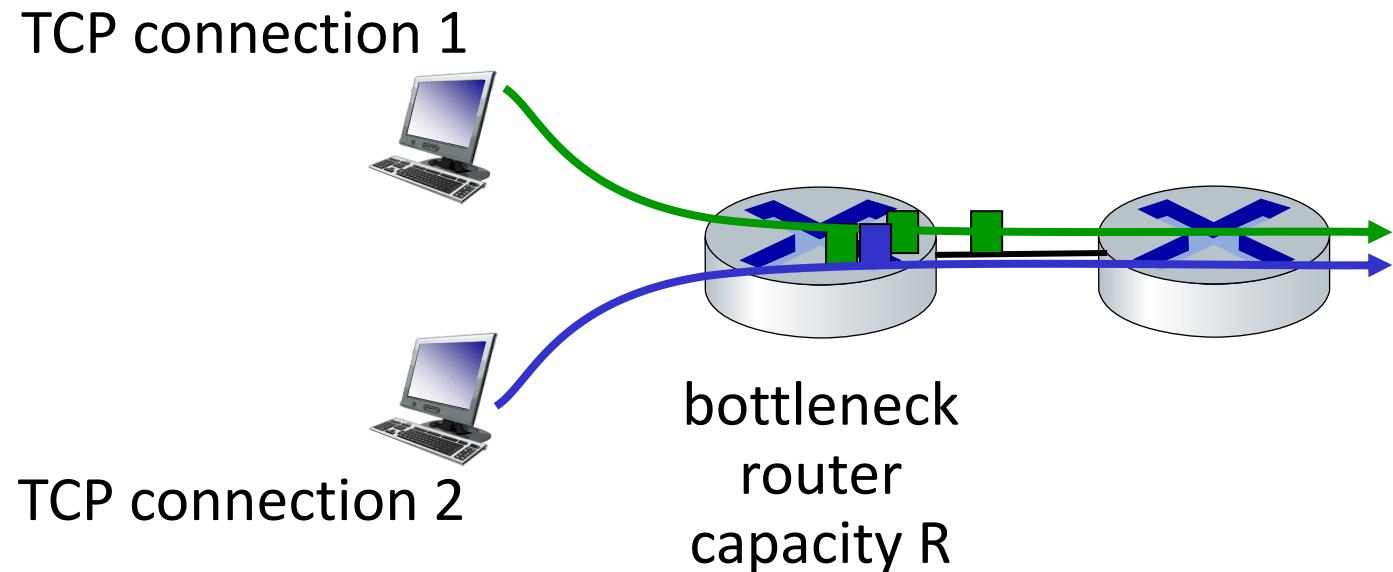
TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



# TCP fairness

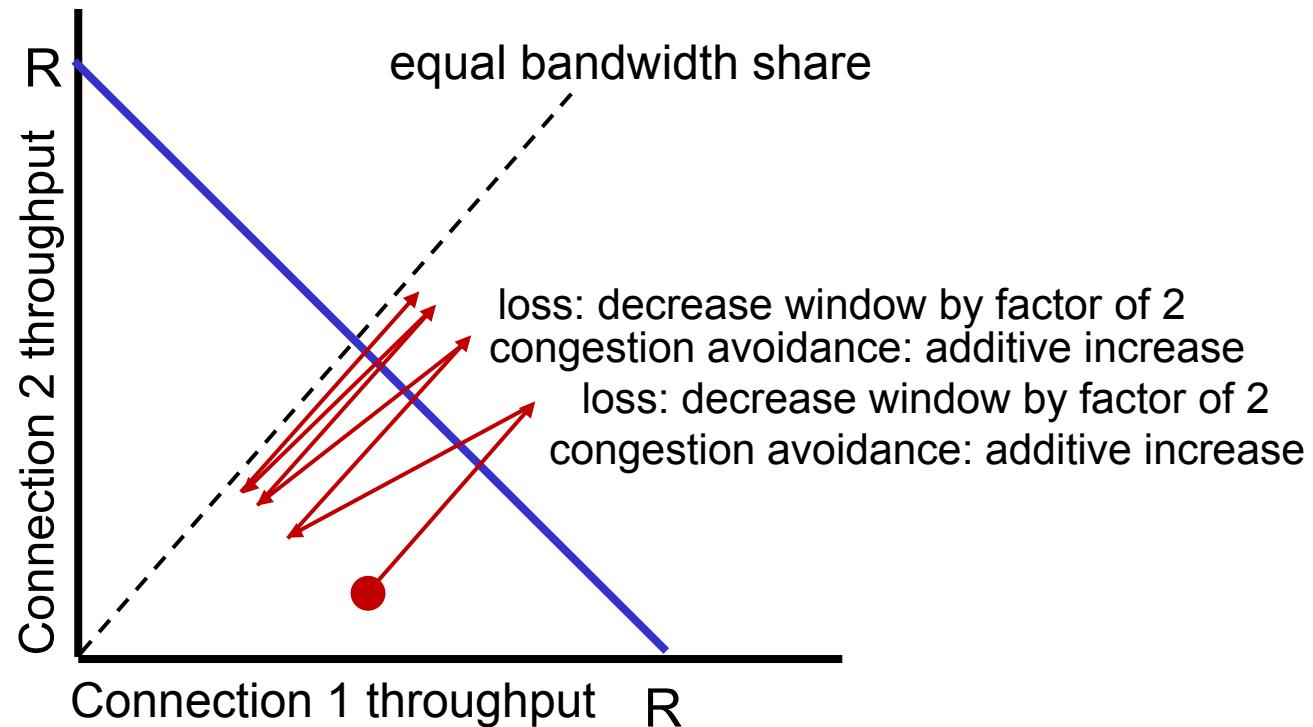
**Fairness goal:** if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



*Is TCP fair?*

**A:** Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

# Fairness: must all network apps be “fair”?

## Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no “Internet police” policing use of congestion control

## Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2