



Informatics

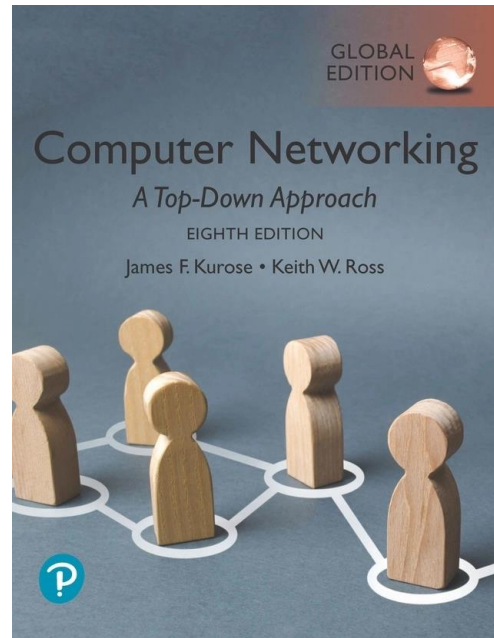


Computer Systems

Networks – Introduction and Application Layer

Gernot Steindl

03.06.2024



- Literature: „Computer Networking – A Top-Down Approach“, written by James F. Kurose and Keith W. Ross
 - <https://www.pearson.de/computer-networking-global-edition>
 - https://gaia.cs.umass.edu/kurose_ross/index.php (Includes resources for students!)
 - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU’s library: https://catalogplus.tuwien.at/permalink/f/8j3js/UTW_alma21140332460003336

- Introduction to the Internet
 - What is the Internet and a protocol?
 - Network edge: hosts, access network
 - Network core: packet/circuit switching, internet structure
 - Protocol layers, service models
- Application Layer
 - Process Communication (Sockets)
 - HTTP
 - DNS

An Introduction to the Internet

Introduction: roadmap

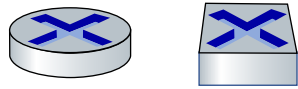
- What is the Internet?
- What is a protocol?
- Network edge: hosts, access network
- Network core: packet/circuit switching, internet structure
- Protocol layers, service models



The Internet: a “nuts and bolts” view



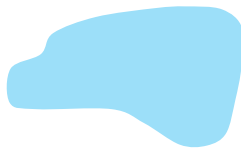
Billions of **devices** are connected to the Internet



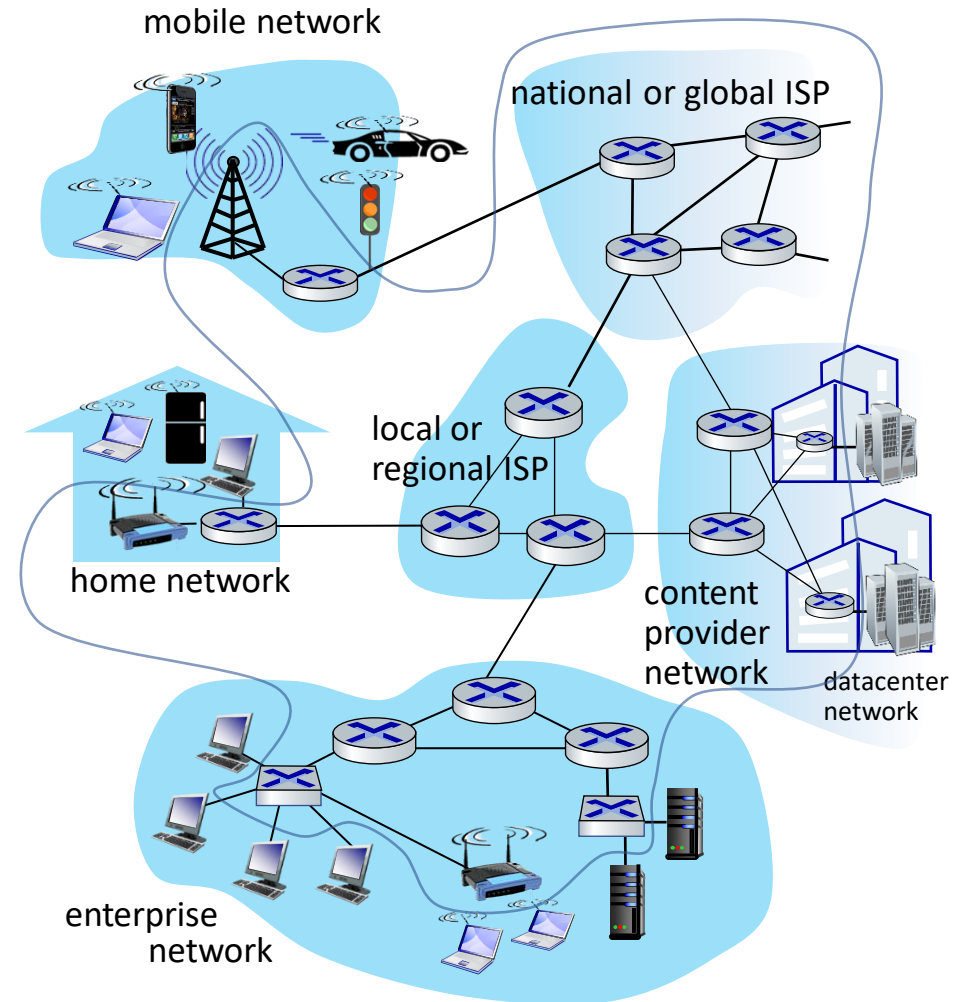
Packet switches forward packets (chunks of data)



Communication Links allow transmitting data

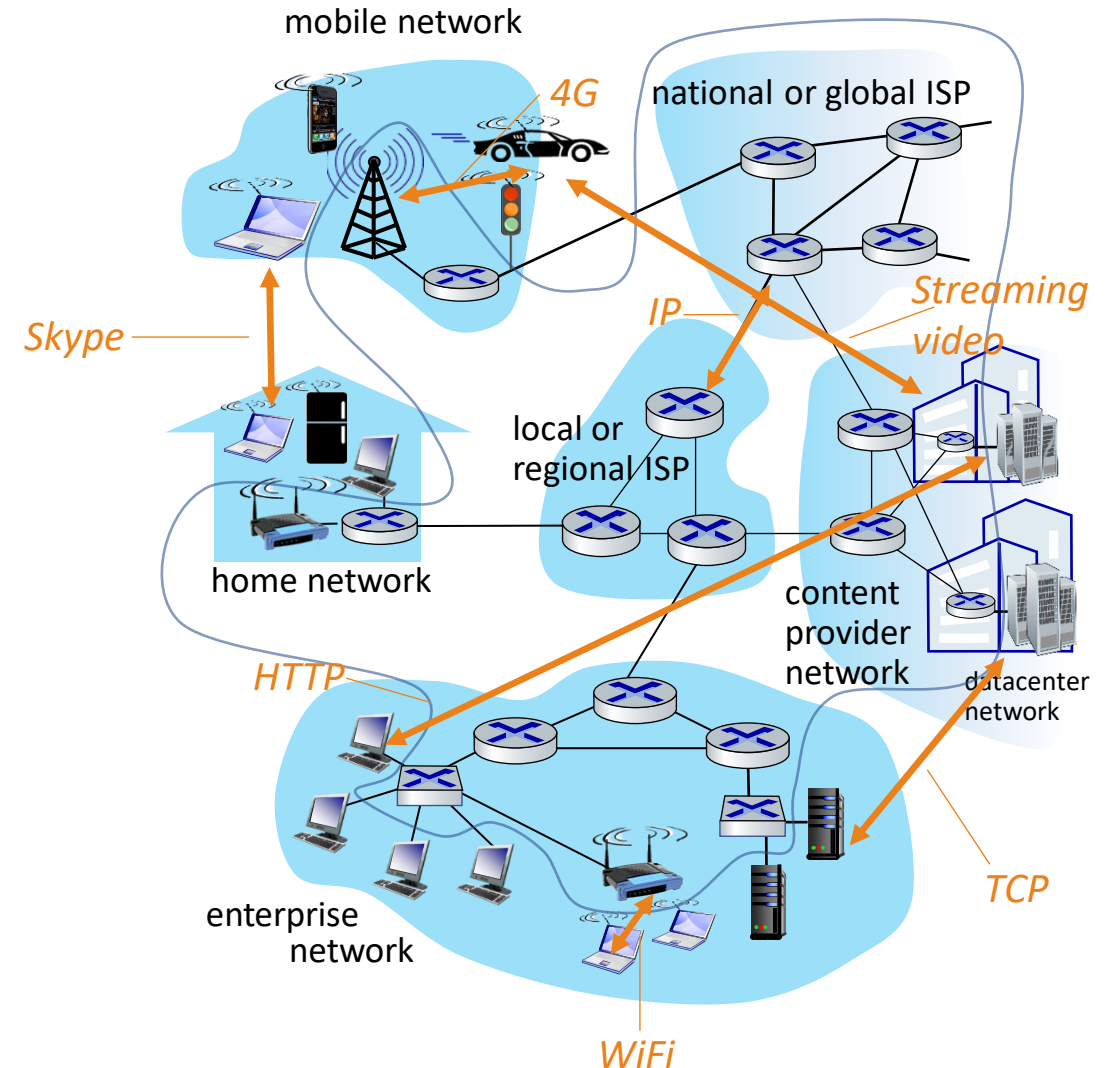


Networks are a collection of devices, packet switches and links (managed by an organization)



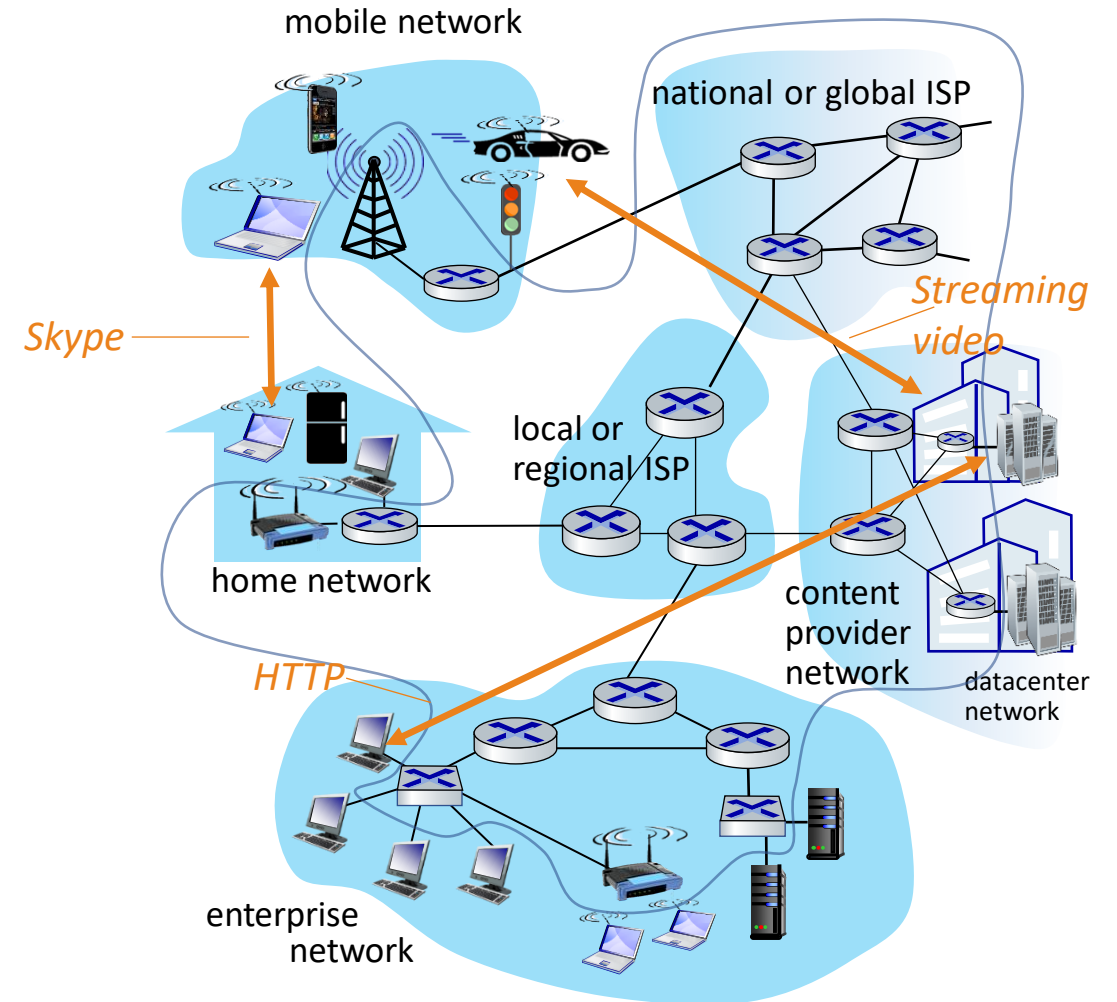
The Internet: a “nuts and bolts” view

- The **Internet** is a “**network of networks**”
 - Interconnected ISPs
- **Protocols** are everywhere
 - Sending and receiving messages
 - For example: HTTP, video streaming, TCP, IP, Ethernet, ...
- **Internet Standards**
 - RFC: Request for Comments
 - IETF: Internet Engineering Task Force



The Internet: a “services” view

- **Infrastructure** that provides services to applications
 - Web, streaming video, multimedia teleconferencing, email, games, e-commerce, social media, inter-connected 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

Network protocols

- “What’s the time?”
- “I have a question”
- Introductions

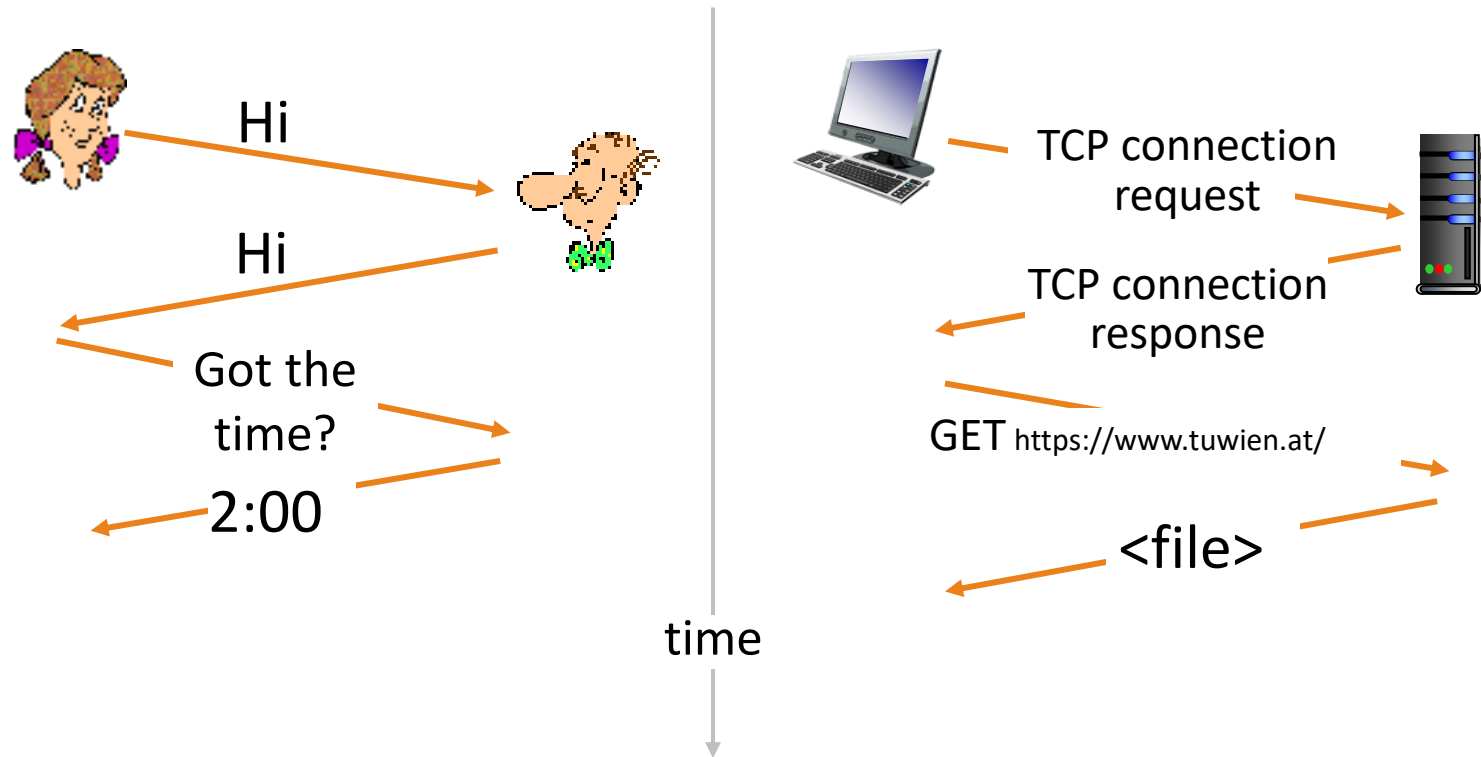
Rules for:

- ... specific messages sent
- ... specific actions taken when message received
- ... specific actions taken on other events

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

What's a protocol?

A human protocol and a computer network protocol:



Introduction : roadmap

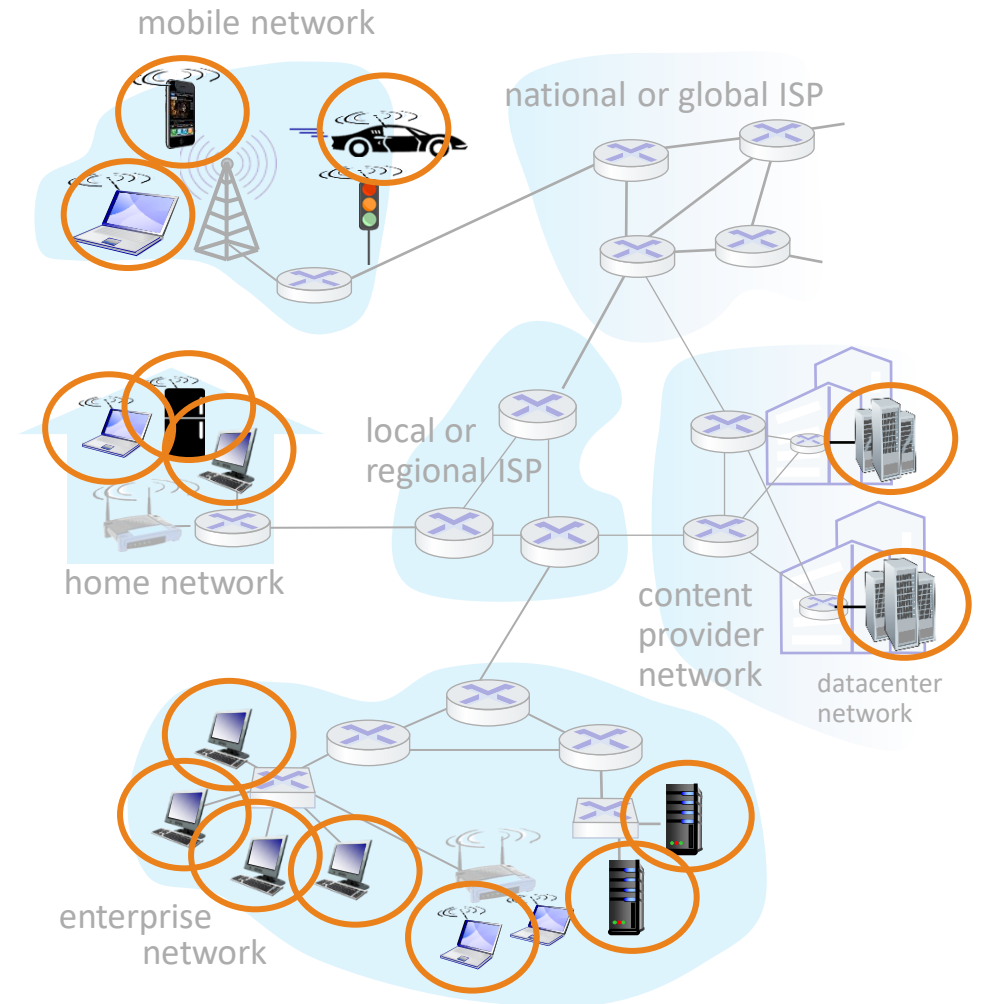
- What *is* the Internet?
- What *is* a protocol?
- **Network edge:** hosts, access network
- Network core: packet/circuit switching
- Protocol layers, service models



A closer look at Internet structure

Network Edge

- Hosts: clients and servers
- Servers often in data centers



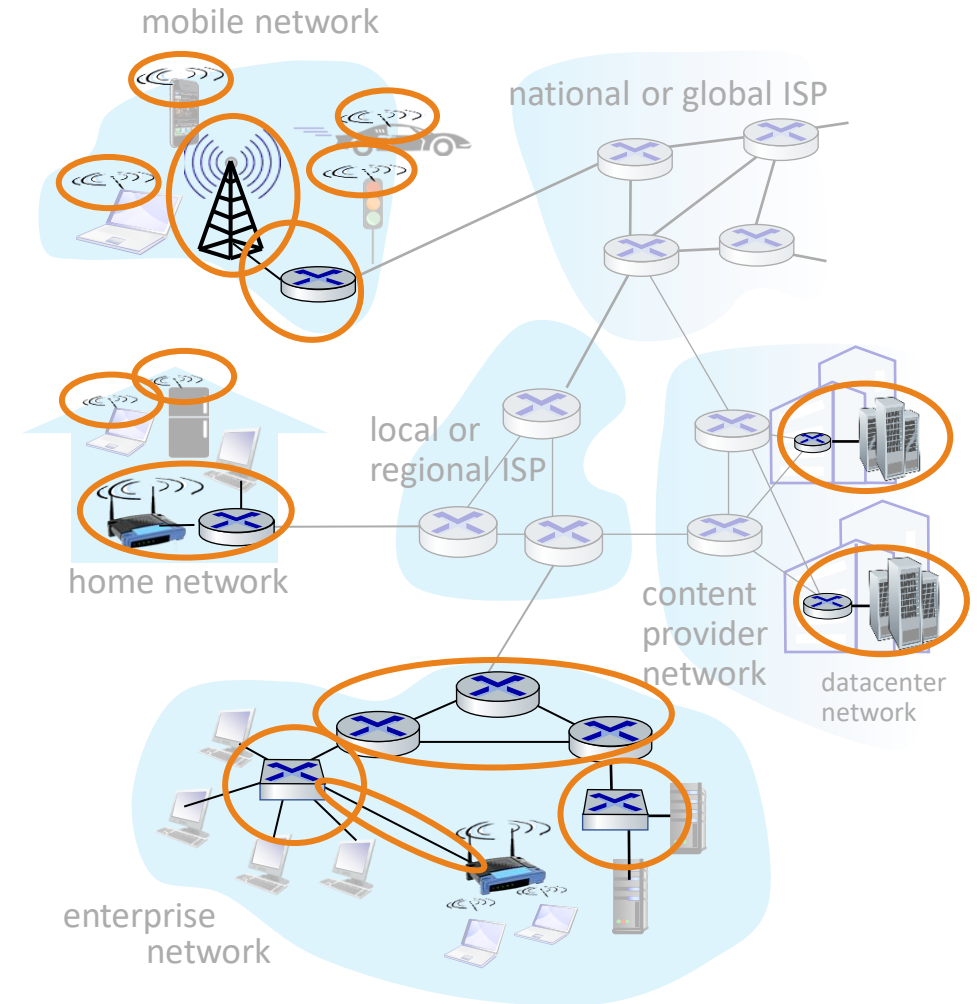
A closer look at Internet structure

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

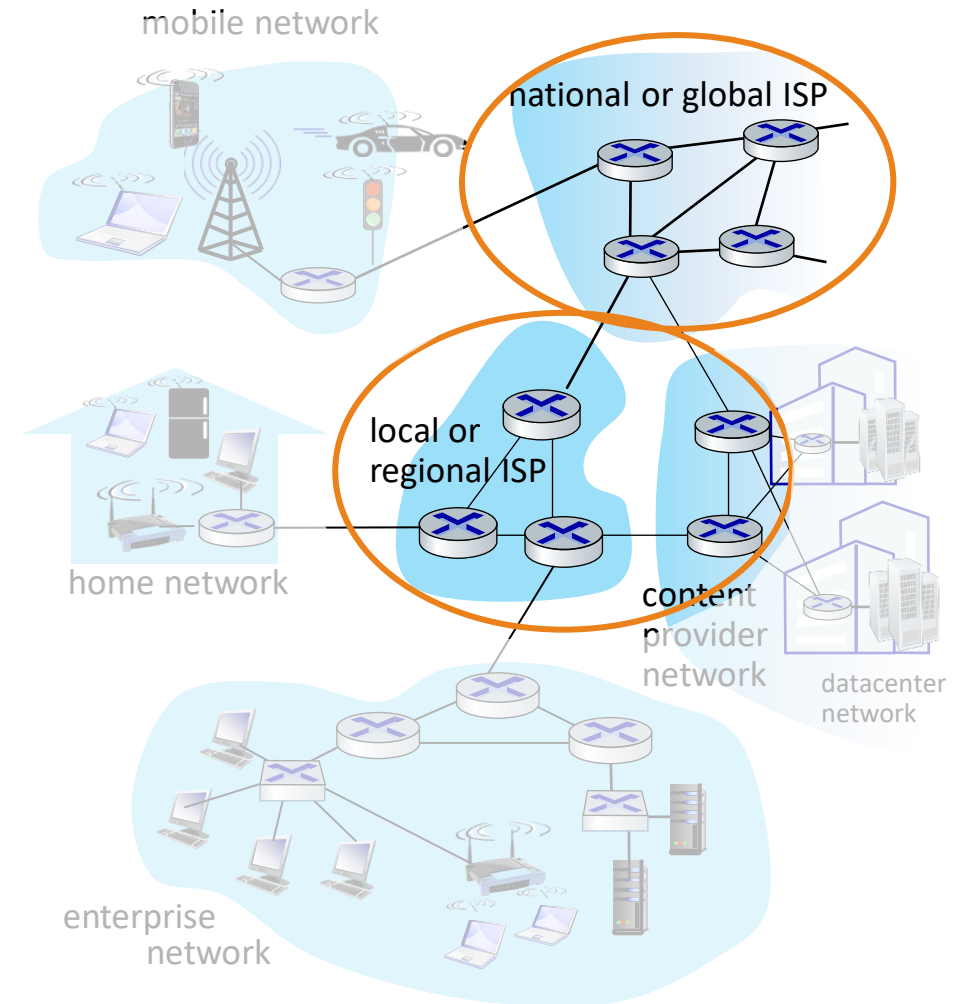
- Hosts: clients and servers
- Servers often in data centers

Access networks, physical media

- Wired, wireless communication links

Network core

- Interconnected routers
- Network of networks



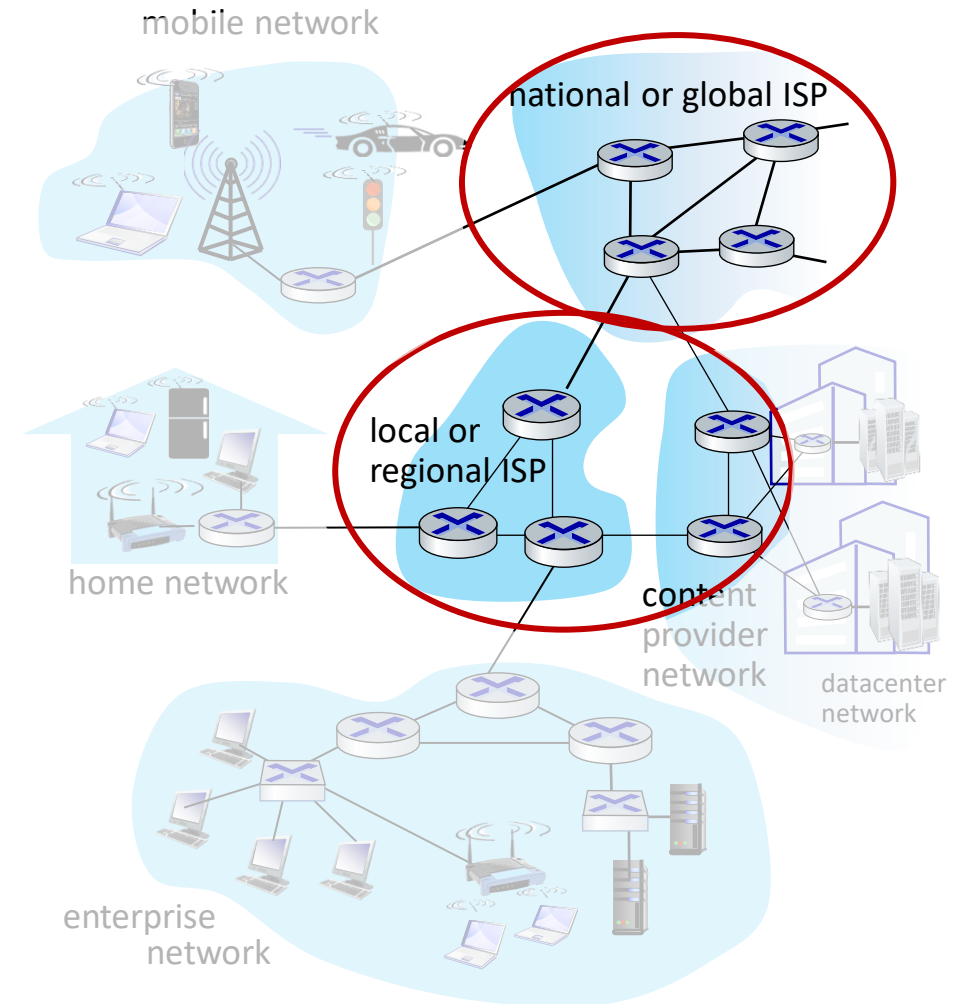
Introduction : roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network
- **Network core:** packet/circuit switching
- Protocol layers, service models



The network core

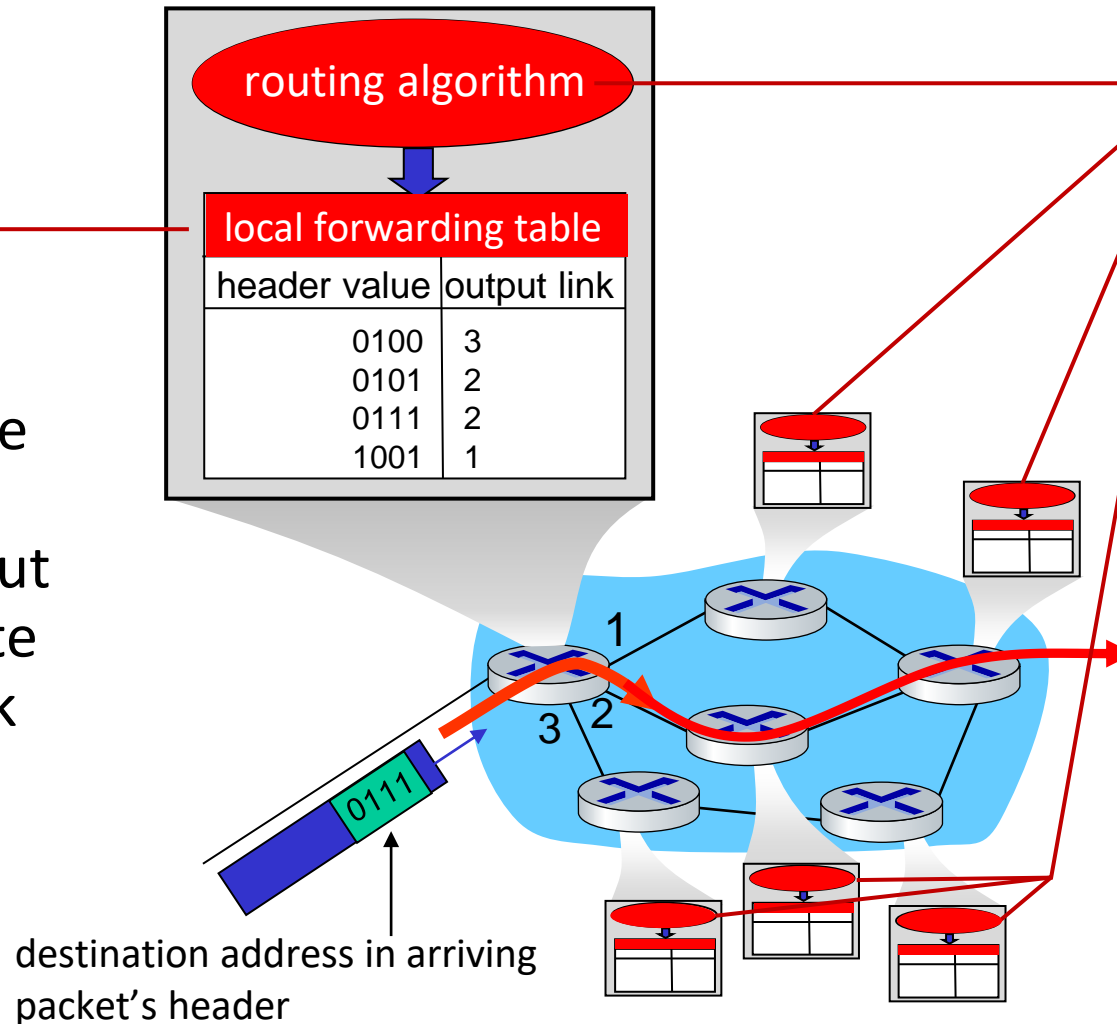
- mesh of interconnected routers
- **packet-switching**: hosts break application-layer messages into *packets*
- network **forwards** packets from one router to the next, across links on path from **source to destination**



Two key network-core functions

Forwarding:

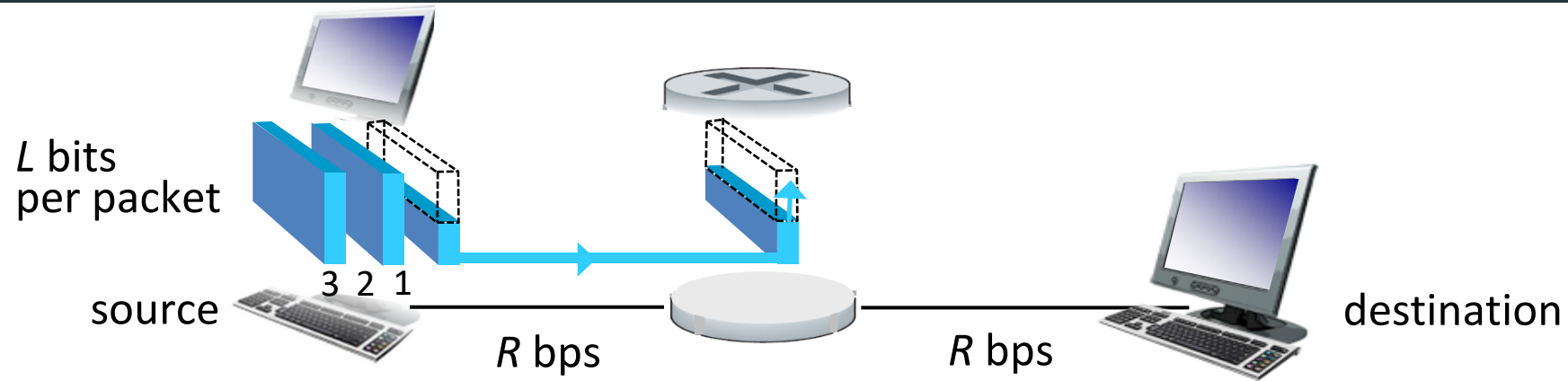
- aka “switching”
- *local* action: move arriving packets from router’s input link to appropriate router output link



Routing:

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

Packet-switching: store-and-forward

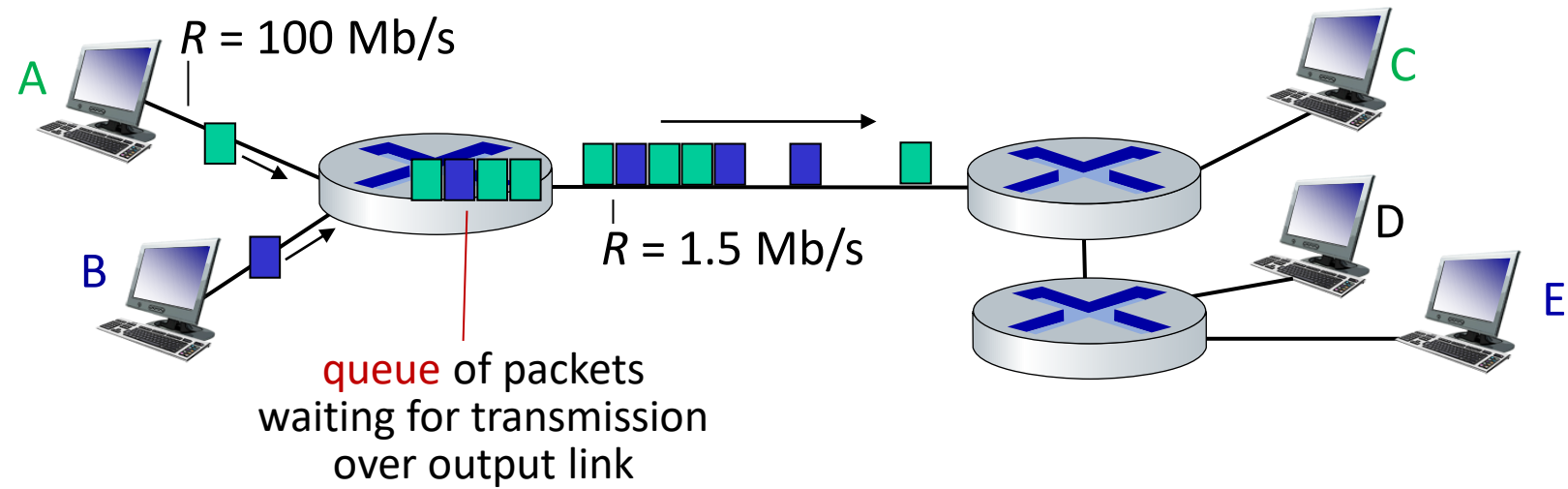


- **packet transmission delay:** takes L/R seconds to transmit (push out) L -bit packet into link at R bps
- **store and forward:** entire packet must arrive at router before it can be transmitted on next link

One-hop numerical example:

- $L = 10$ Kbits
- $R = 100$ Mbps
- one-hop transmission delay = 0.1 msec

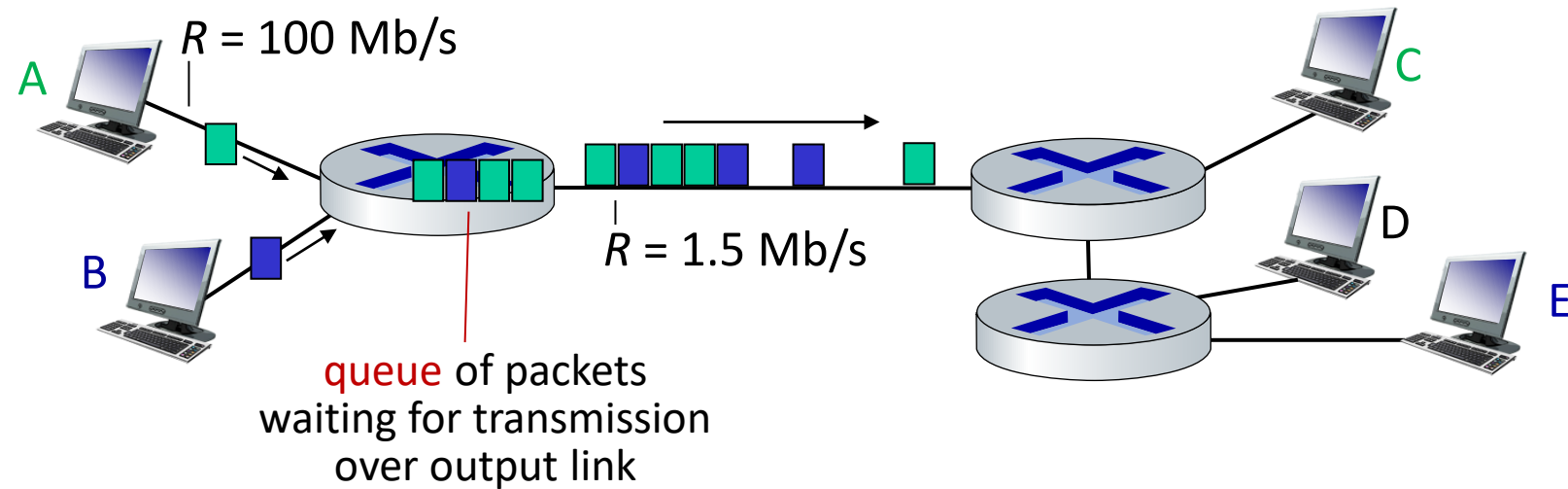
Packet-switching: queueing



Queueing occurs when work arrives faster than it can be serviced:



Packet-switching: queueing



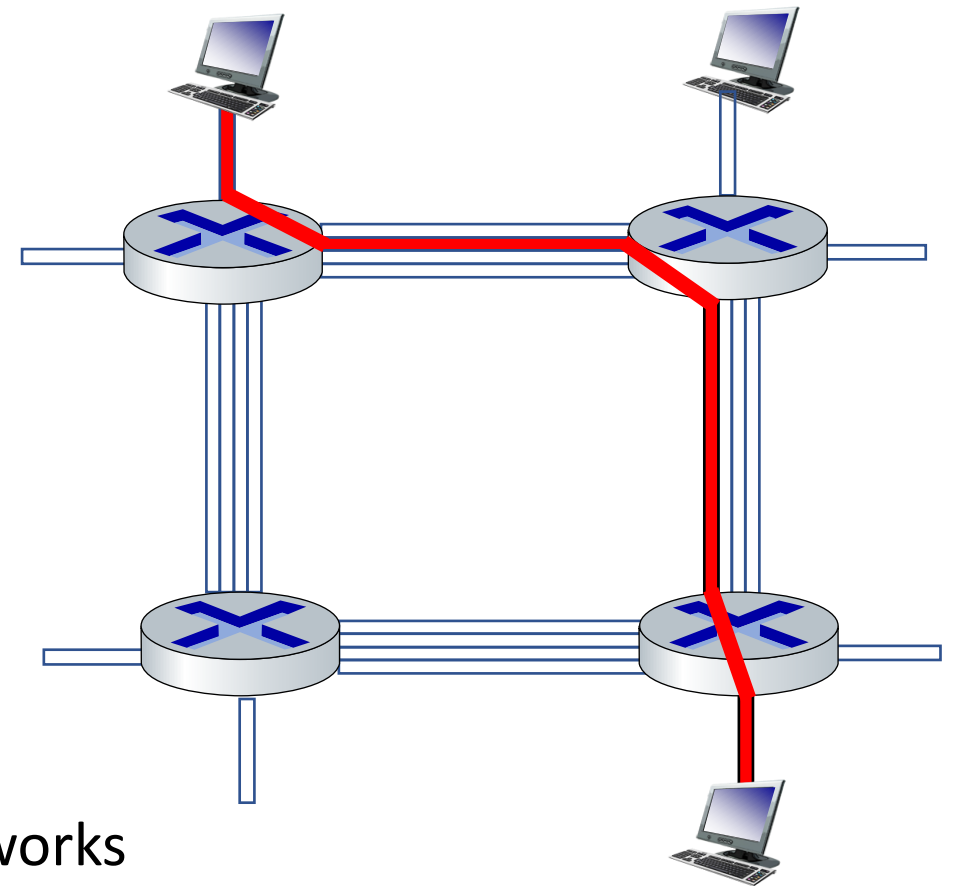
Packet queuing and loss: if arrival rate (in bps) to link exceeds transmission rate (bps) of link for some period of time:

- 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 2nd circuit in top link and 1st 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



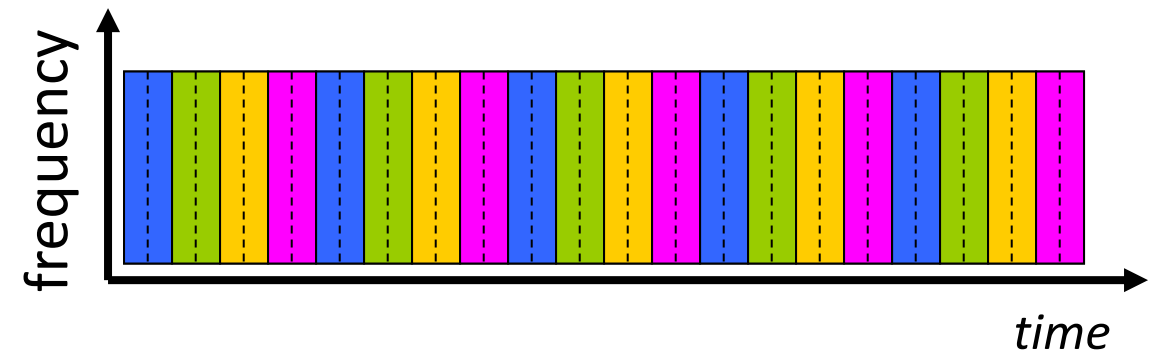
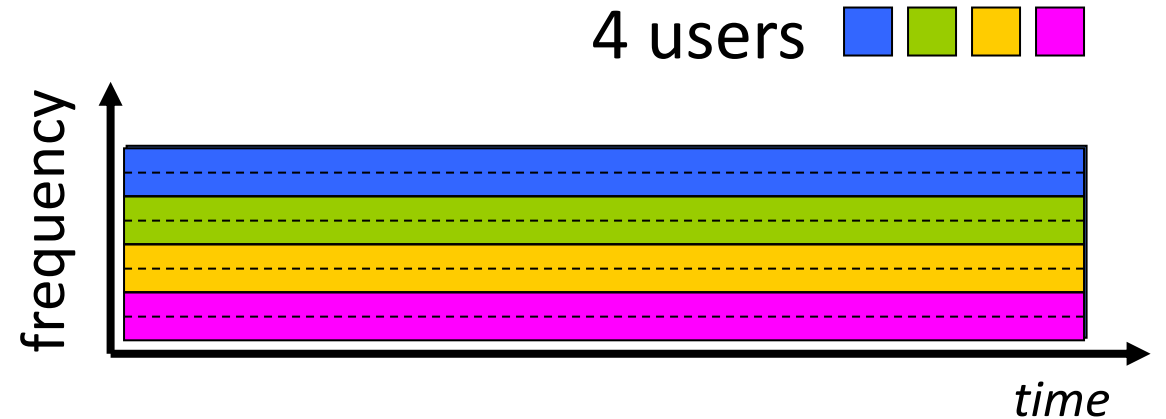
Circuit switching: FDM and TDM

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)

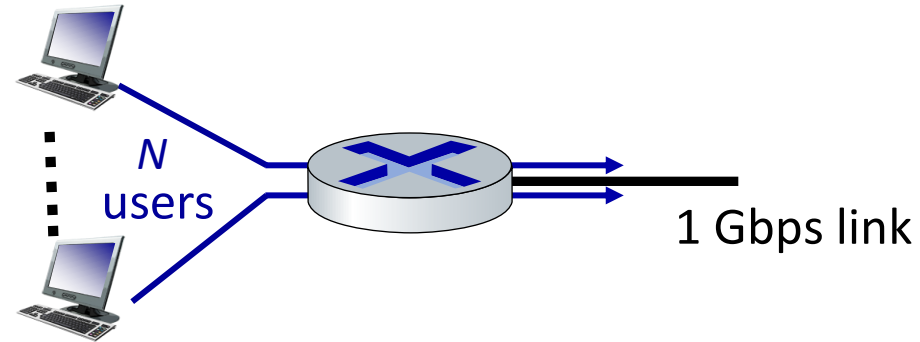
- time divided into slots
- each call allocated periodic slot(s), can transmit at maximum rate of (wider) frequency band (only) during its time slot(s)



Packet switching versus circuit switching

example:

- 1 Gb/s link
- each user:
 - 100 Mb/s when “active”
 - active 10% of time



Q: how many users can use this network under circuit-switching and packet switching?

- **circuit-switching:** 10 users
- **packet switching:** with 35 users, probability > 10 active at same time is less than .0004

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: How to provide circuit-like behavior with packet-switching?**
 - “It’s complicated.” We’ll study various techniques that try to make packet switching as “circuit-like” as possible.

Introduction: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network
- Network core: packet/circuit switching
- Protocol layers, service models



Networks are complex with many “pieces”

- Hosts
- Routers
- Links over various media (e.g., Wi-Fi)
- Applications
- Protocols
- Hardware, software

Question: Is there any hope of organizing the structure of the network?

- And/or our discussions on networks?

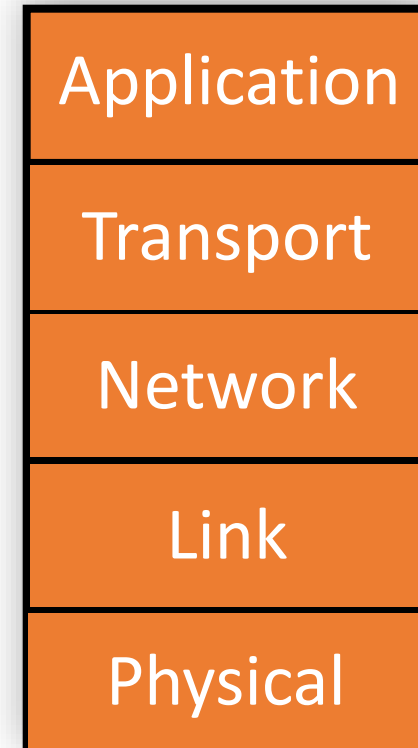
Why Layering?

Approach to designing/discussing complex systems:

- Explicit structure allows identification, relationship of system's pieces
 - Layered **reference model** for discussions
- Modularization eases maintenance, updating of system
 - Change in layer's service implementation is transparent to the rest of the system
 - For example, change in gate procedure doesn't affect rest of the system

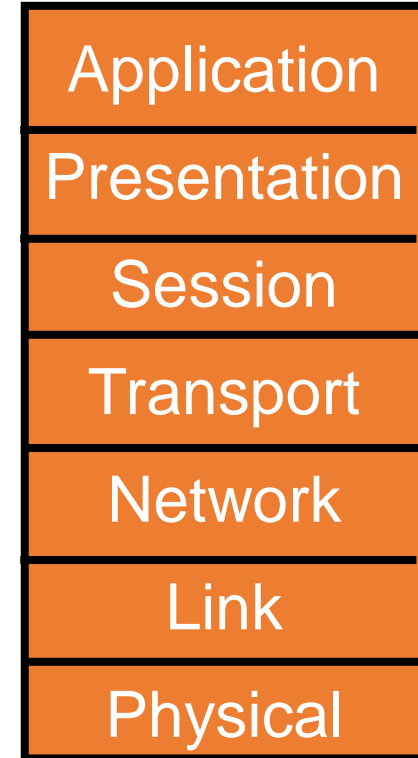
Layered Internet Protocol Stack

- **Application**: supporting network applications
 - HTTP, IMAP, SMTP, DNS
- **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 (Wi-Fi), PPP
- **Physical**: bits “on the wire”



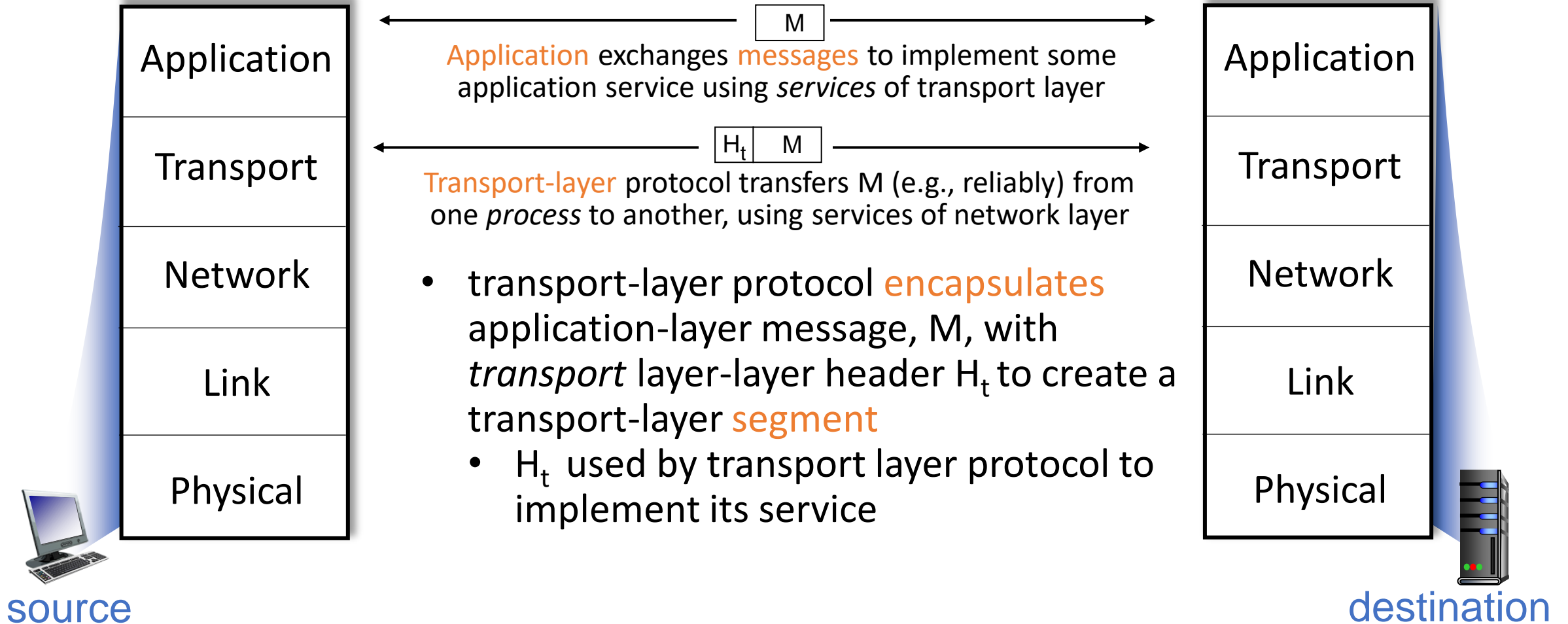
ISO/OSI Reference Model

- **Presentation**: allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions
- **Session**: synchronization, checkpointing, recovery of data exchange
- Internet stack “missing” these layers!
 - These services, *if needed*, must be implemented in application

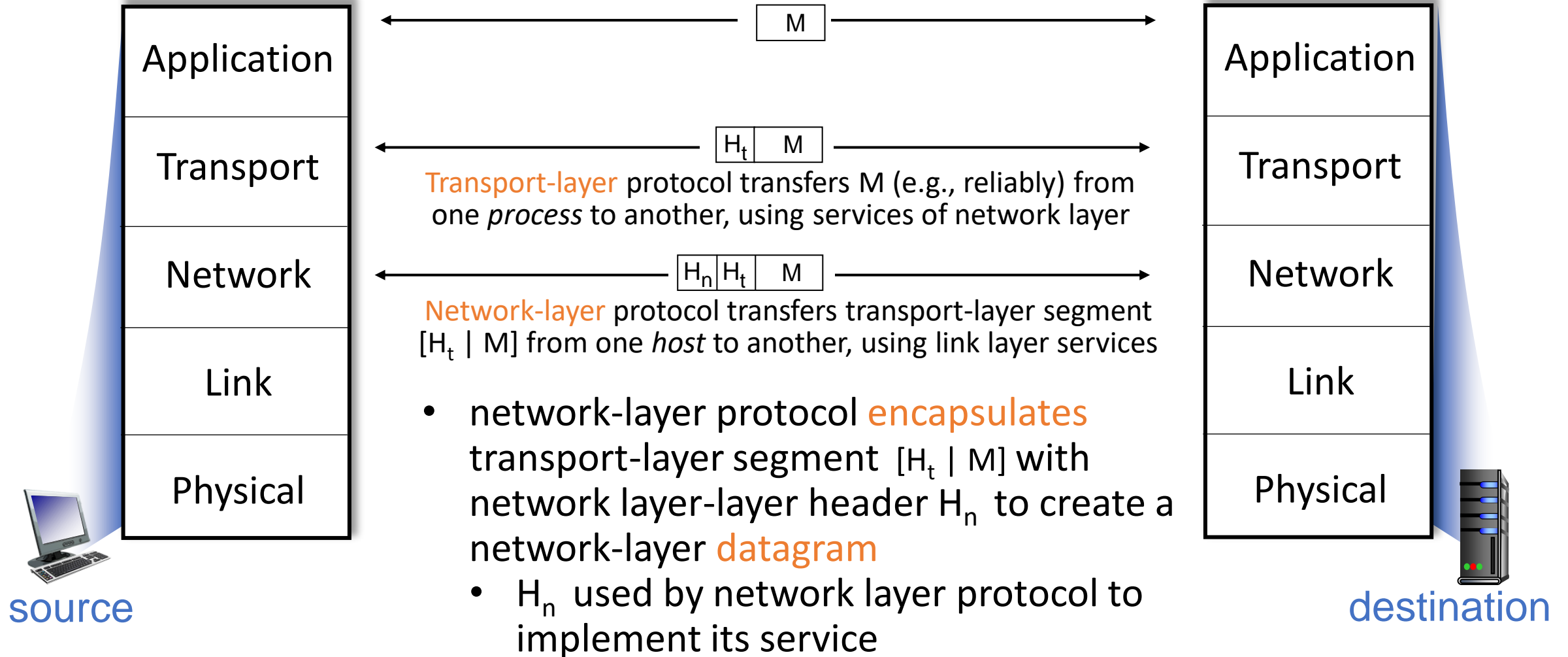


The seven-layer ISO/OSI reference model

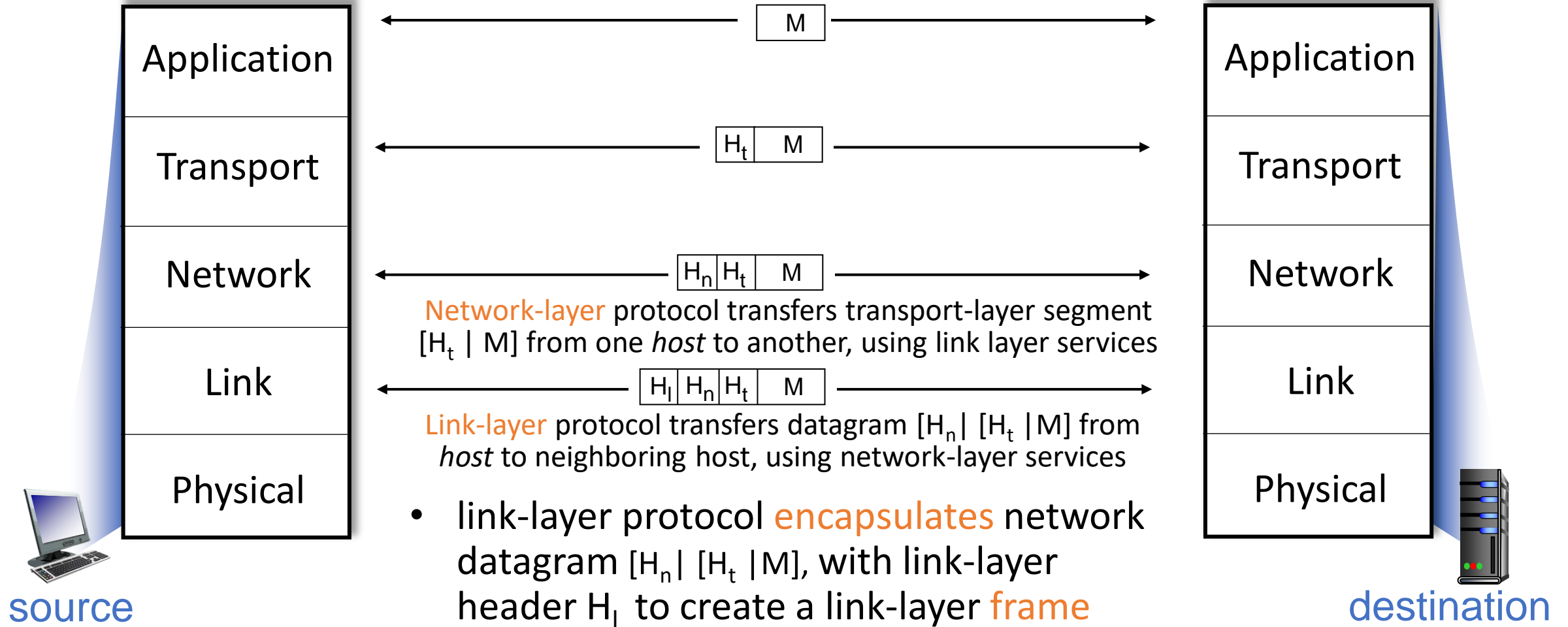
Services, Layering, and Encapsulation



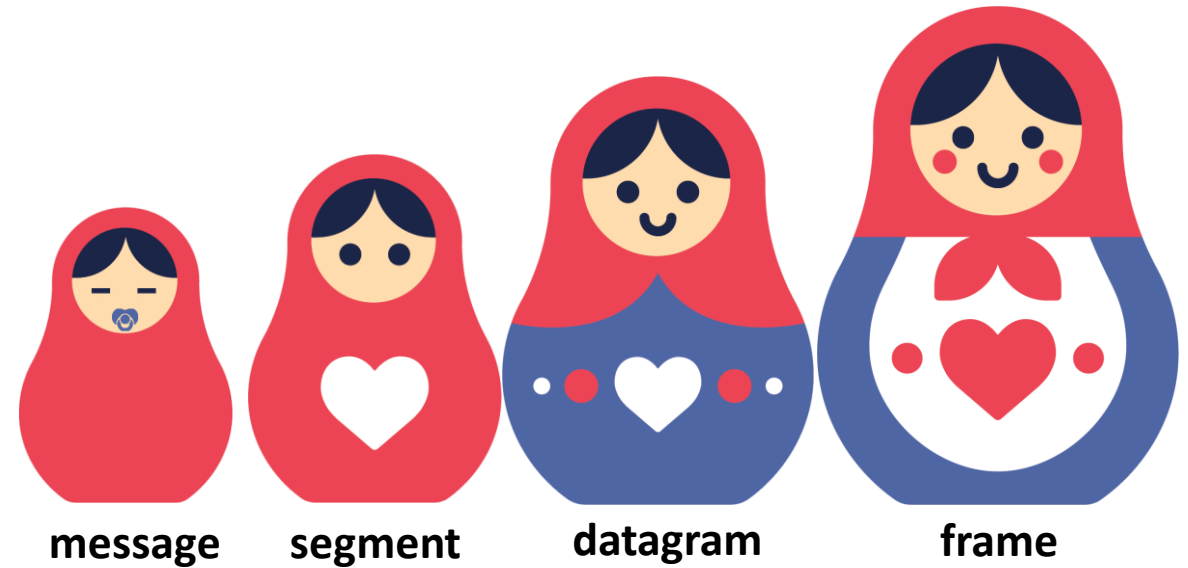
Services, Layering, and Encapsulation



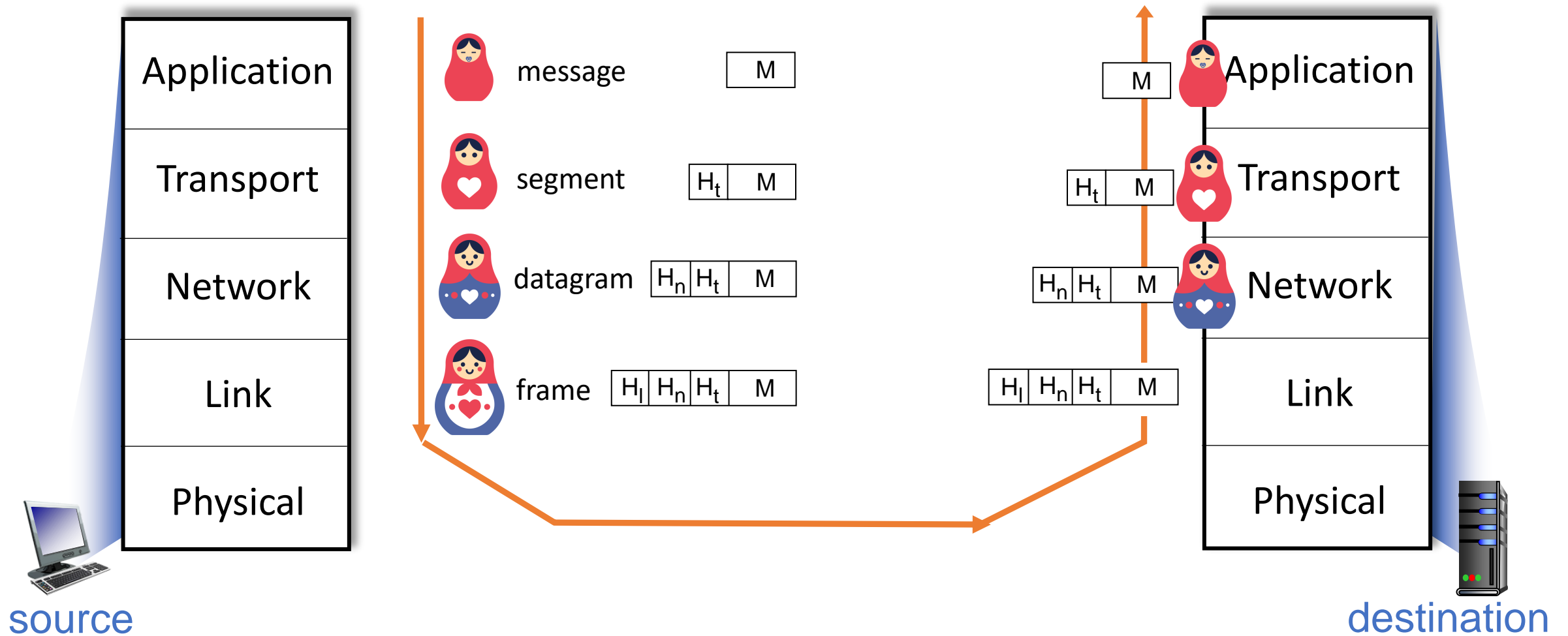
Services, Layering, and Encapsulation



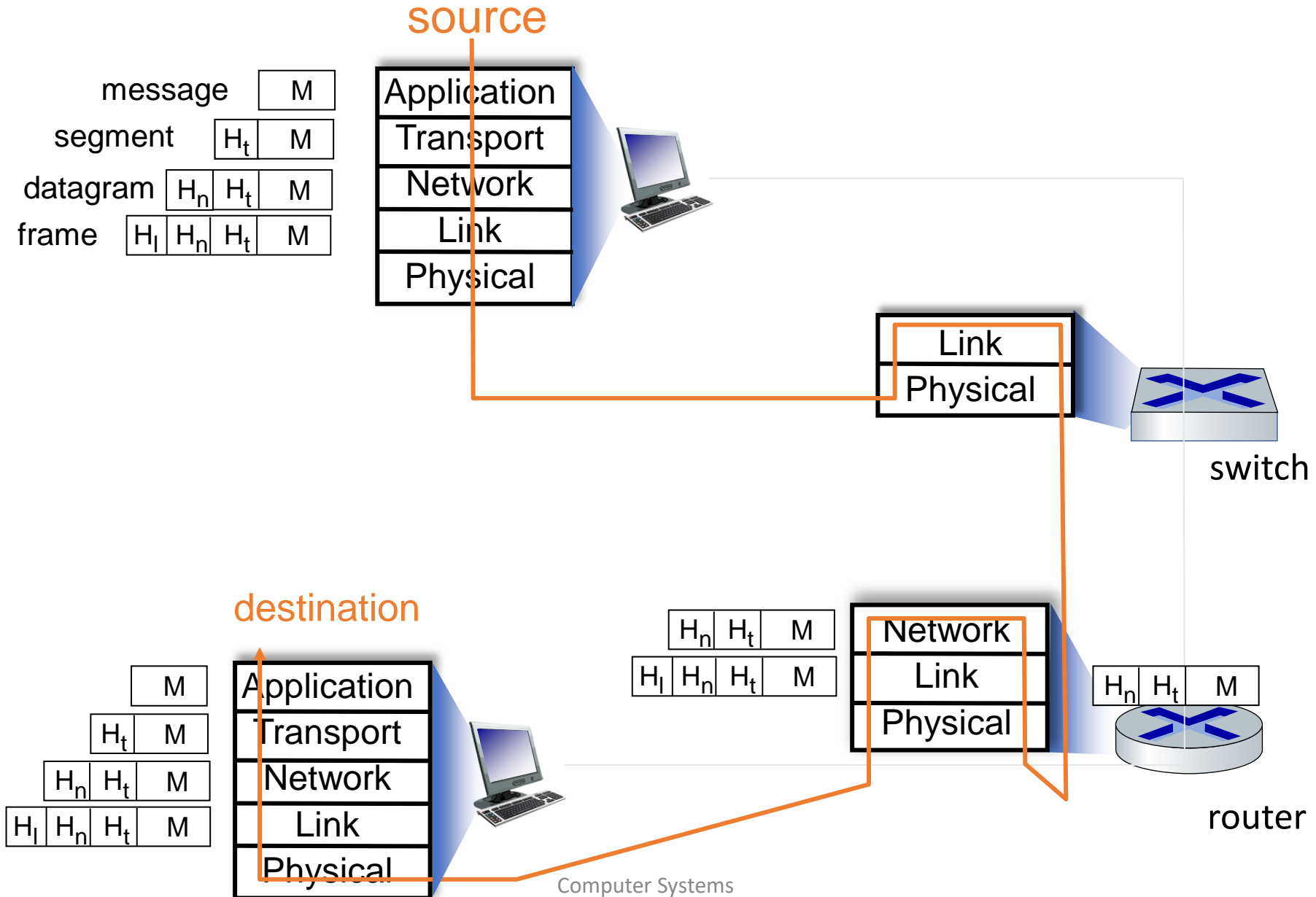
Matryoshka dolls (stacking dolls)



Services, Layering, and Encapsulation



Services, Layering, and Encapsulation



Application Layer

Some Network Apps

- Social networks
- Web
- Text messaging
- E-mail
- Online games
- Streaming stored video
- ...

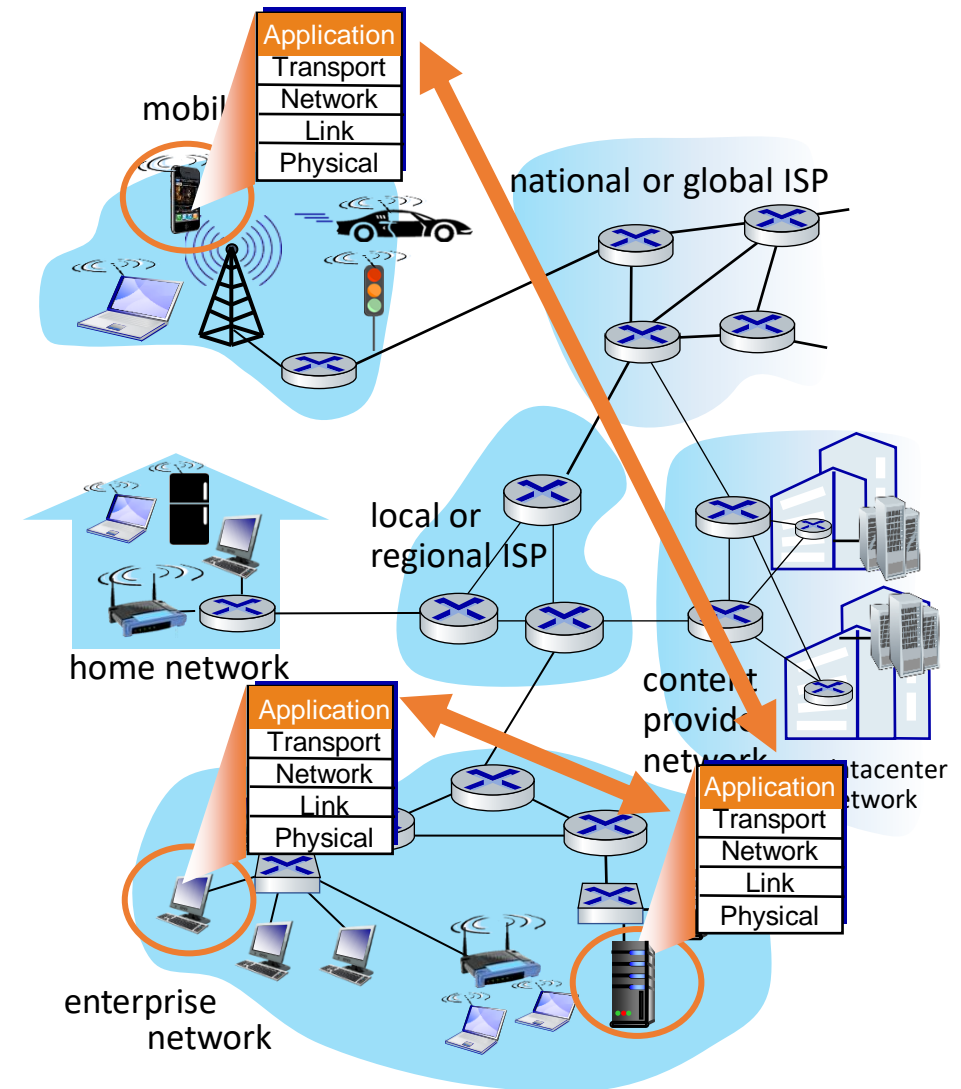
Creating a Network App

Write programs that:

- Run on different end systems
- Communicate over the network
- For example, web server software communicates with your 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

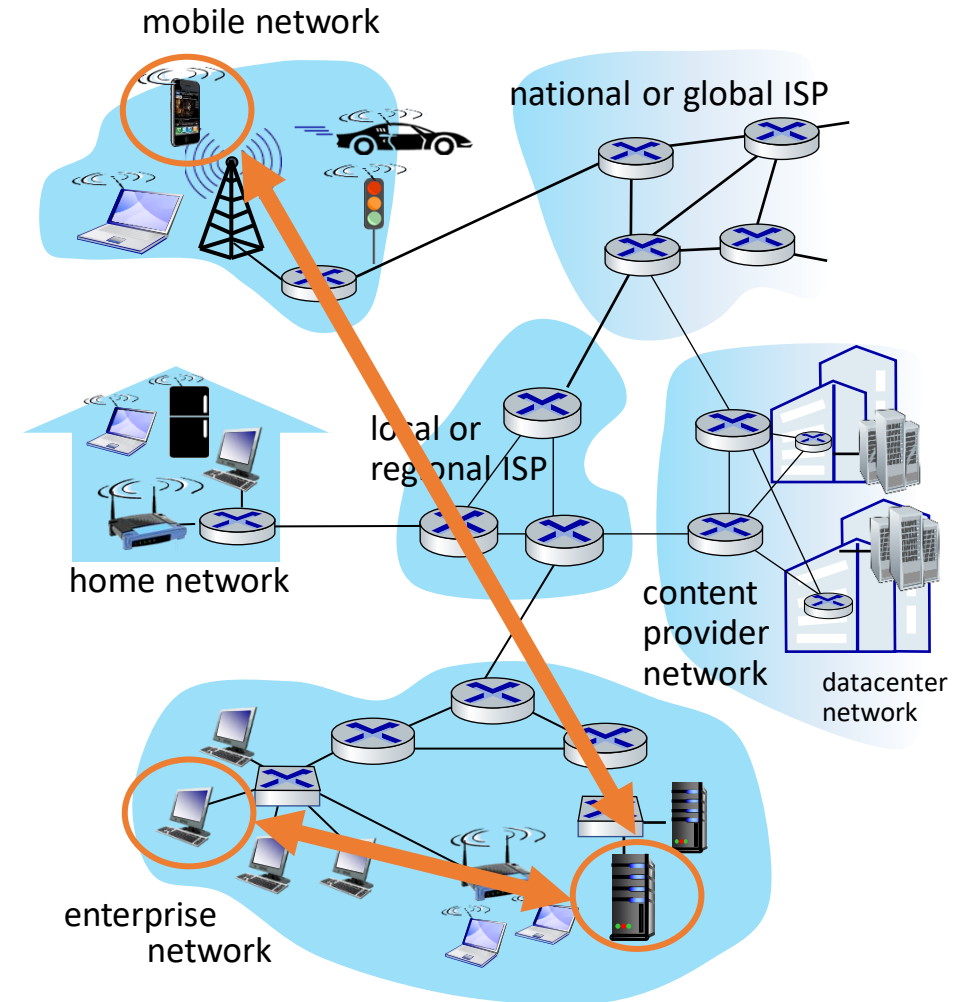
Server

- Always-on host
- 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

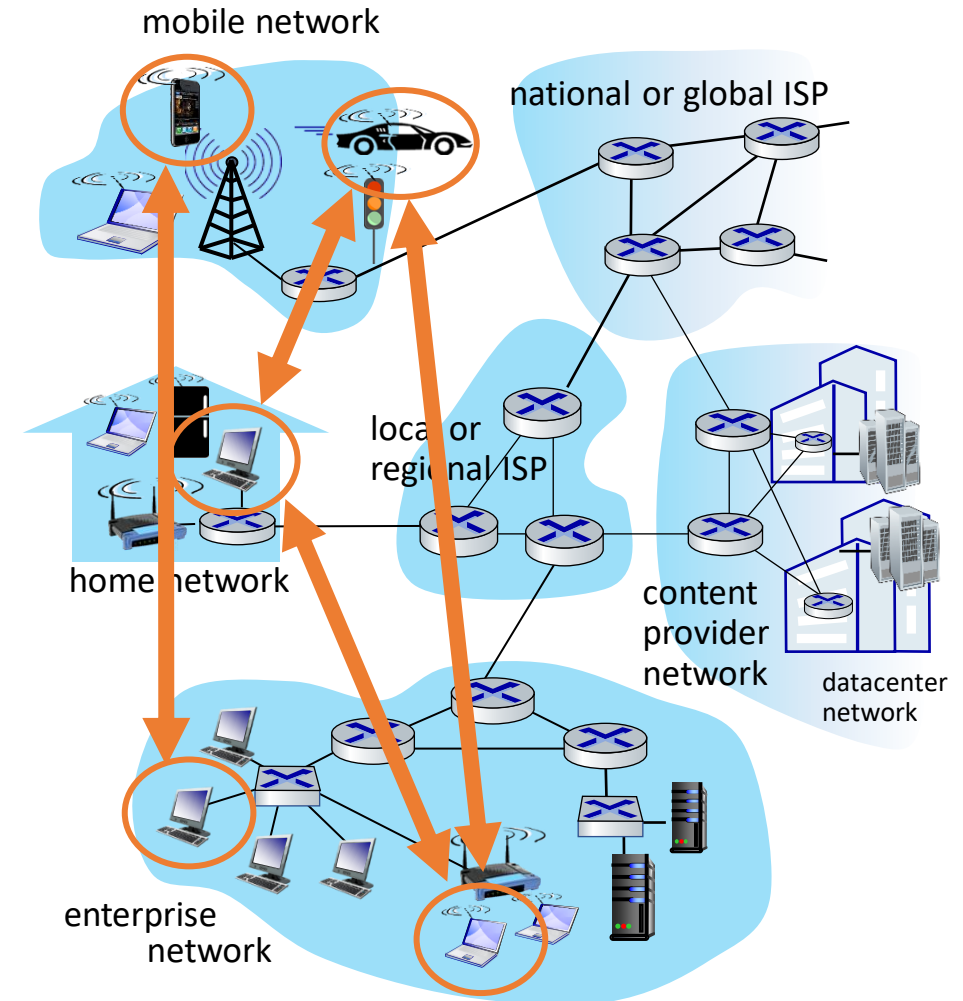
Examples: HTTP, IMAP, FTP



Peer-Peer Architecture (not further discussed in this lecture)

- 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 [BitTorrent]



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**

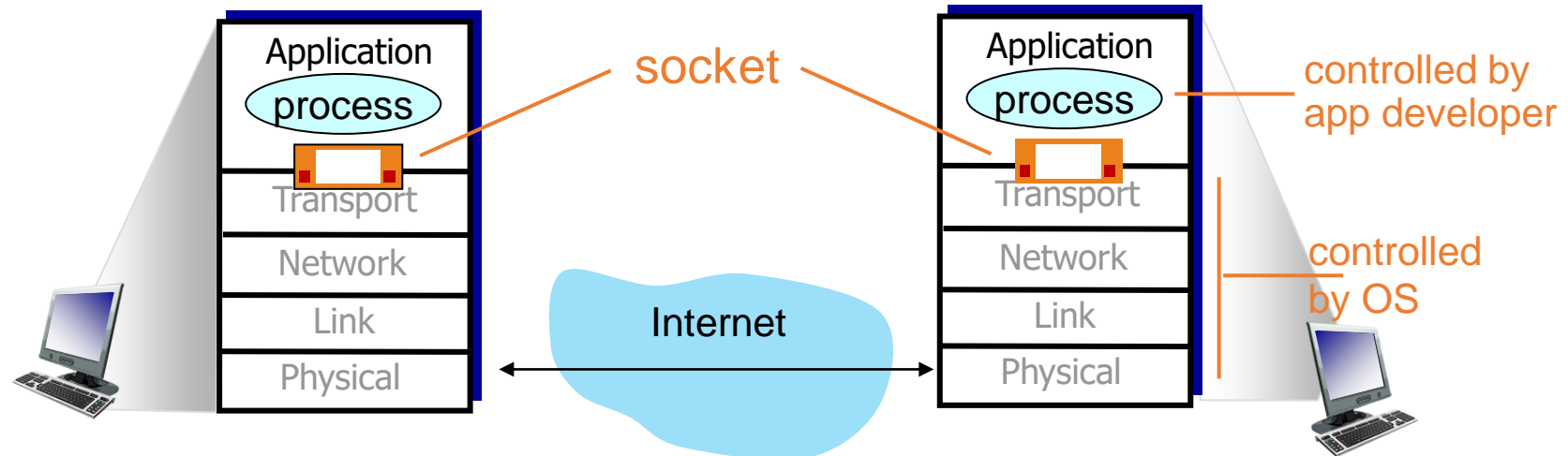
clients, servers

Client process: process that initiates communication

Server process: process that waits to be contacted

Sockets

- Process sends/receives messages to/from its socket
- Socket analogous to door
 - sending process shoves message out of the door
 - 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 *www.tuwien.at* web server:
 - **IP address**: 128.130.35.76
 - **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

Timing

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

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

Security

- Confidentially, integrity, availability, ...

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: 5 Kbps – 1 Mbps Video: 10 Kbps – 5 Mbps	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

TCP service

- **Reliable transport** between sending and receiving process
- **Flow control**: sender won't overwhelm receiver
- **Congestion control**: throttle sender when network overloaded
- **Connection-oriented**: setup required between client and server processes
- **Does not provide** timing, minimum throughput guarantee, security

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.

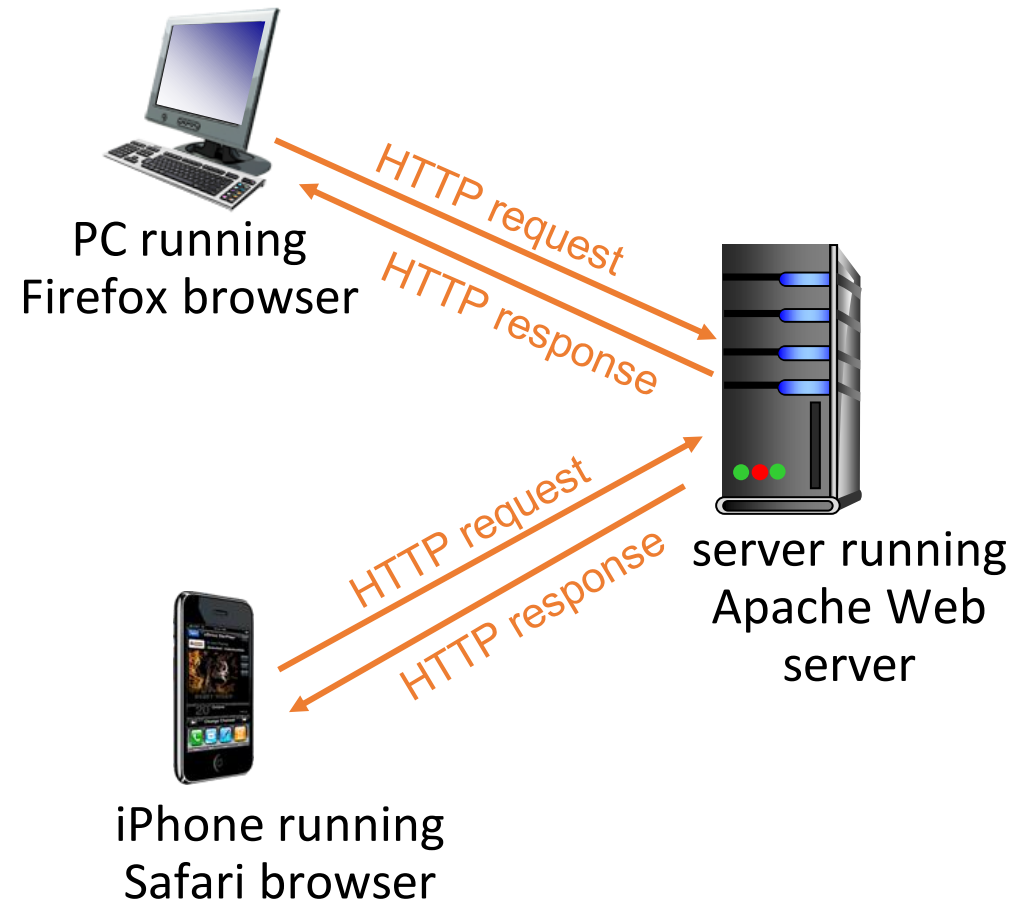
Internet Applications, and Transport Protocols

Application	Application Layer Protocol	Transport Protocol
File transfer / download	FTP [RFC 959]	TCP
E-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
Streaming audio / video	HTTP [RFC 7230], DASH	TCP
Interactive games	WOW, FPS (proprietary)	UDP or TCP

The Web and HTTP

Hyper Text Transfer Protocol

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



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

Protocols that maintain “state” are complex!

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

User enters URL: `https://informatics.tuwien.ac.at/orgs/e191-03`

The screenshot shows the website for the Automation Systems research unit at TU Wien. The header includes the TU Wien logo, the text 'Informatics 20 YEARS', and navigation links for 'MENU', 'A-Z', and 'SEARCH'. A dropdown menu labeled 'ORG UNITS' is open, showing the selected unit 'Automation Systems E191-03'. Below the title, it states 'A RESEARCH UNIT OF THE INSTITUTE OF COMPUTER ENGINEERING'. The main text describes the unit's focus on IoT applications in manufacturing, power grids, buildings, rescue, and traffic management. A sidebar on the right contains contact information for Johann Blieberger, including his title, website (www.auto.tuwien.ac.at), and location (Treitlstrasse 3). Below the contact info, there is a section 'ON THIS PAGE' with links for '# About', '# People', and '# Activities'. At the bottom of the page, there is a decorative graphic with the text 'INIK FUER MENSCHEN' overlaid on a building outline.

TU Informatics **20 YEARS**
WIEN

MENU A-Z SEARCH

ORG UNITS ▾

Automation Systems E191-03

A RESEARCH UNIT OF THE INSTITUTE OF COMPUTER ENGINEERING

The “Internet of Things” (IoT) is commonly anticipated with numerous applications including but not limited to manufacturing, power grids, buildings, rescue and relief operations, environmental monitoring, and traffic management.

CONTACT

Head: **Johann Blieberger**

Web: www.auto.tuwien.ac.at

Location: Treitlstrasse 3

ON THIS PAGE

- # About
- # People
- # Activities

INIK FUER MENSCHEN

HTTP Example

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`

- Use your browser to examine, for example, requests for linked images
- We will explore some of these aspects shortly
- Note that the actual server redirects you to the `https://` URL for security reasons, we will ignore this for now and assume that the `http://` URL directly hosts the Web page

Response Status Codes

HTTP Method

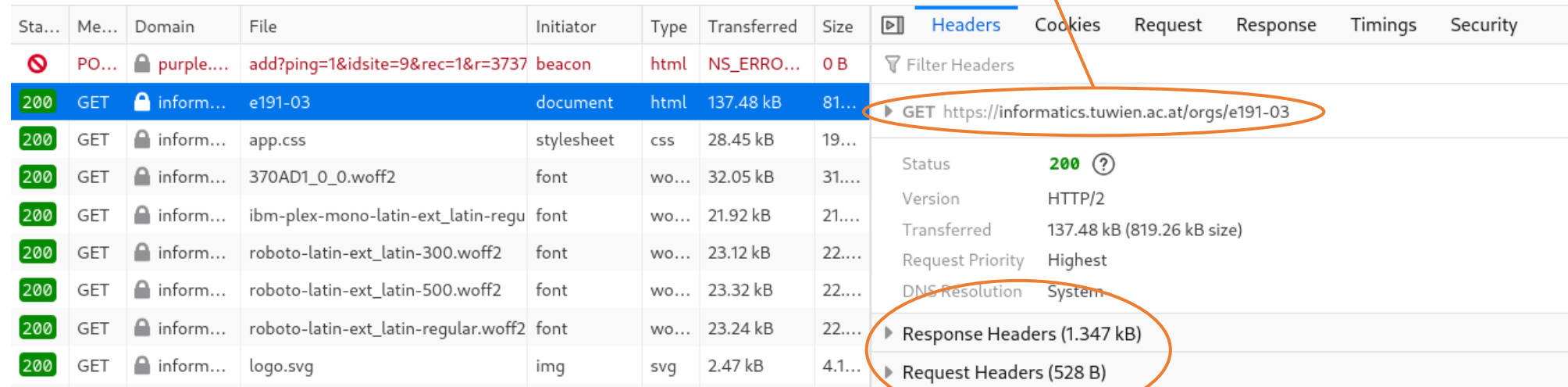
Requested Object

Status	Method	Domain	File	Initiator	Type	Transferred	Size	0 ms	40.9
200	GET	informatics.tuwien.ac.at	logo.svg	img	svg	2.47 kB	4.13 kB	15 ms	
200	GET	informatics.tuwien.ac.at	20years.svg	img	svg	4.32 kB	17.46 kB	15 ms	
200	GET	informatics.tuwien.ac.at	pixel-770x550.png	img	png	829 B	159 B	14 ms	
200	GET	informatics.tuwien.ac.at	silhouette.svg	img	svg	1.18 kB	896 B	14 ms	
200	GET	informatics.tuwien.ac.at	sprite.svg	img	svg	1.70 kB	5.80 kB	2 ms	
200	GET	informatics.tuwien.ac.at	apple-touch-icon.png	FaviconLoader.sys.mjs:17...	png	4.89 kB	4.24 kB	2 ms	
200	GET	informatics.tuwien.ac.at	favicon-16x16.png	FaviconLoader.sys.mjs:17...	png	948 B	298 B	1 ms	
200	GET	informatics.tuwien.ac.at	hero-1x.webp	imageset	webp	30.15 kB	29.44 kB	5 ms	

HTTP Example

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`

- You can also view the details for requests ...



The screenshot shows the Chrome DevTools Network tab. The top part is a table of network requests. The second row is selected, showing a GET request to `https://informatics.tuwien.ac.at/orgs/e191-03`. The details pane on the right shows the request status (200), version (HTTP/2), transferred size (137.48 kB), and priority (Highest). The response headers and request headers are also visible.

Sta...	Me...	Domain	File	Initiator	Type	Transferred	Size	
🚫	PO...	purple...	add?ping=1&idsite=9&rec=1&r=3737	beacon	html	NS_ERRO...	0 B	
200	GET	inform...	e191-03	document	html	137.48 kB	81...	▶ GET https://informatics.tuwien.ac.at/orgs/e191-03
200	GET	inform...	app.css	stylesheet	css	28.45 kB	19...	
200	GET	inform...	370AD1_0_0.woff2	font	wo...	32.05 kB	31...	
200	GET	inform...	ibm-plex-mono-latin-ext_latin-regu	font	wo...	21.92 kB	21...	
200	GET	inform...	roboto-latin-ext_latin-300.woff2	font	wo...	23.12 kB	22...	
200	GET	inform...	roboto-latin-ext_latin-500.woff2	font	wo...	23.32 kB	22...	
200	GET	inform...	roboto-latin-ext_latin-regular.woff2	font	wo...	23.24 kB	22...	
200	GET	inform...	logo.svg	img	svg	2.47 kB	4.1...	

Status	200
Version	HTTP/2
Transferred	137.48 kB (819.26 kB size)
Request Priority	Highest
DNS Resolution	System

Response Headers (1.347 kB)
Request Headers (528 B)

HTTP Example

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`

- ... and the corresponding responses
- If this piques your interest: 188.951 Web Engineering

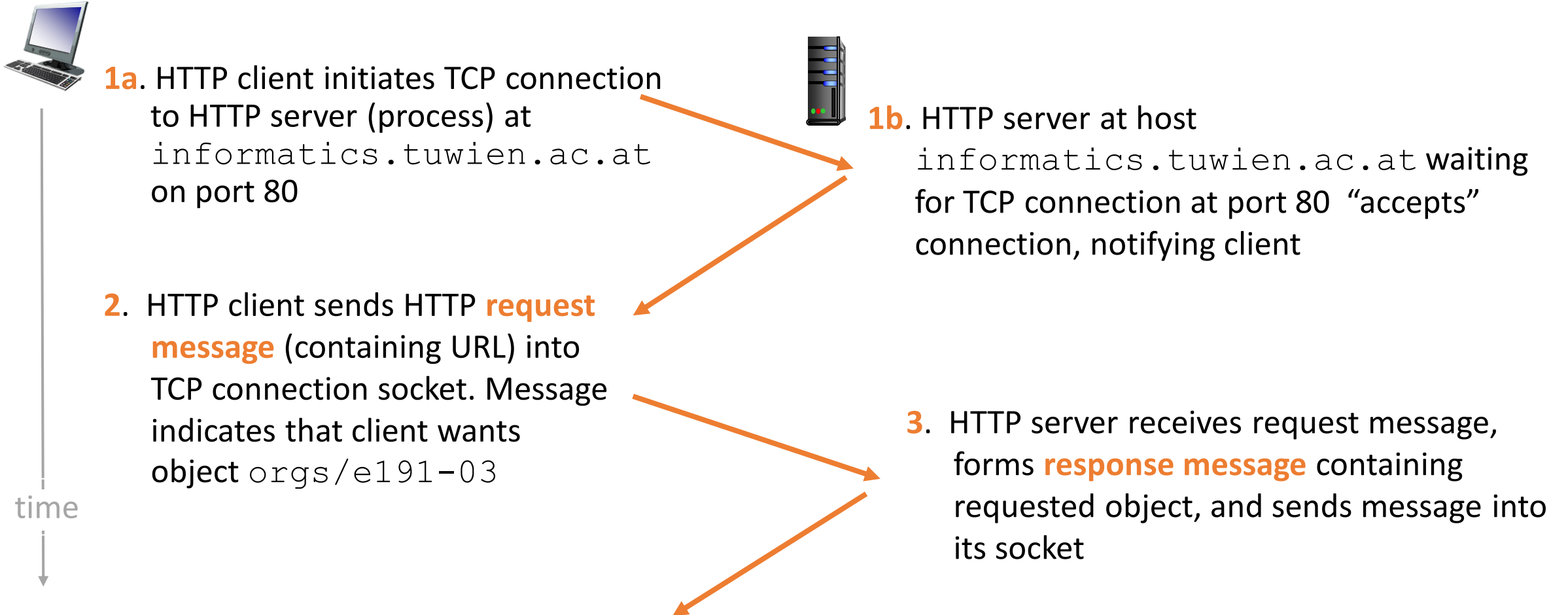
The screenshot shows the Network tab of a browser's developer tools. The 'Response' tab is selected for the 'e191-03' request. The response content is HTML code, with the following snippet visible:

```
<link rel="stylesheet" media="all" href="/assets/w18/1.8.10/css/app.css" />  
<style class="anchorjs">.anchorjs-xc8vj0f { top: 0 }</style>  
<link rel="preload" crossorigin="anonymous" as="font" type="font/woff2" href=  
<link rel="preload" crossorigin="anonymous" as="font" type="font/woff2" href=  
<link rel="preload" crossorigin="anonymous" as="font" type="font/woff2" href=  
<link rel="preload" crossorigin="anonymous" as="font" type="font/woff2" href=  
<link rel="preload" crossorigin="anonymous" as="font" type="font/woff2" href=  
<link rel="preconnect" href="https://purple.zkk.tuwien.ac.at:8080/">
```

The 'app.css' request in the list is highlighted with a blue background, and an orange arrow points from the circled link in the response to this request.

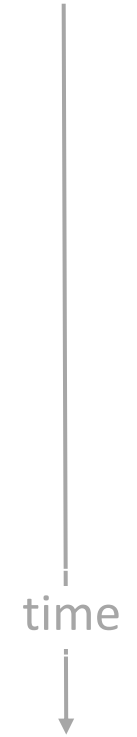
HTTP Example

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`



HTTP Example

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`



5. HTTP client receives response message containing html file, displays html. Parsing html file, finds referenced objects (e.g., images)
6. Repeat steps 1-5 for referenced objects

4. HTTP server closes TCP connection



HTTP Example – Some Additional Notes (not relevant for the exam)

User enters URL: `http://informatics.tuwien.ac.at/orgs/e191-03`

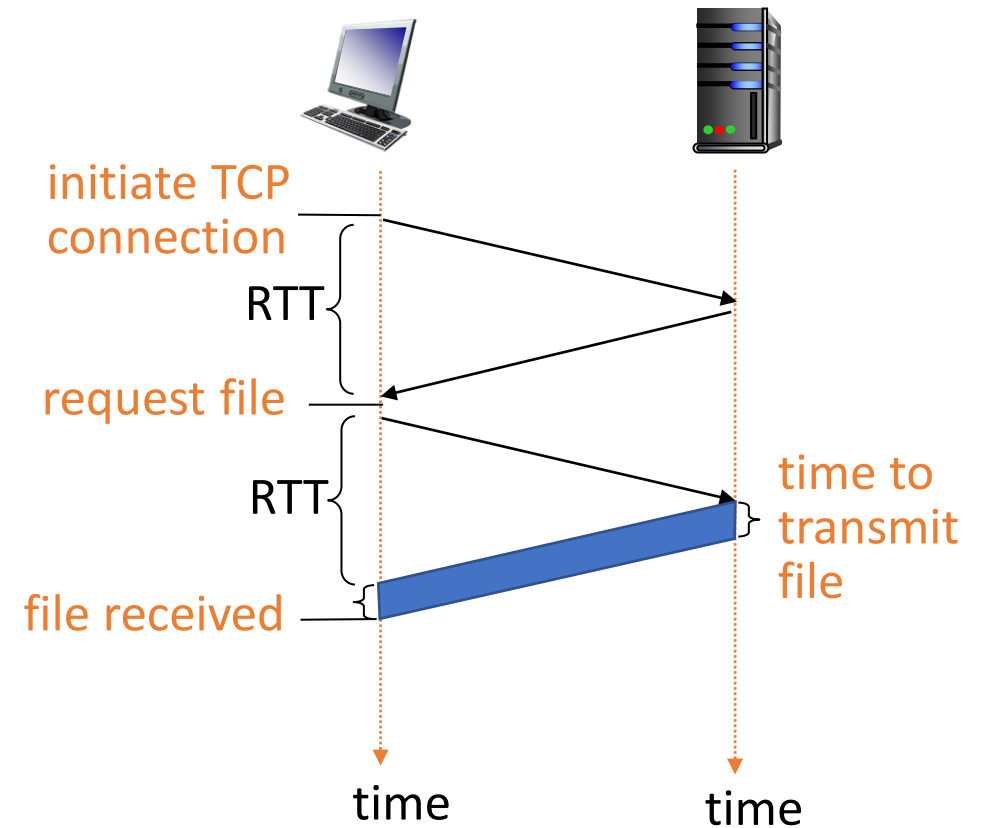
- The HTTP URL often redirects you to an HTTPS (HTTP Secure) URL on modern Web pages

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



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

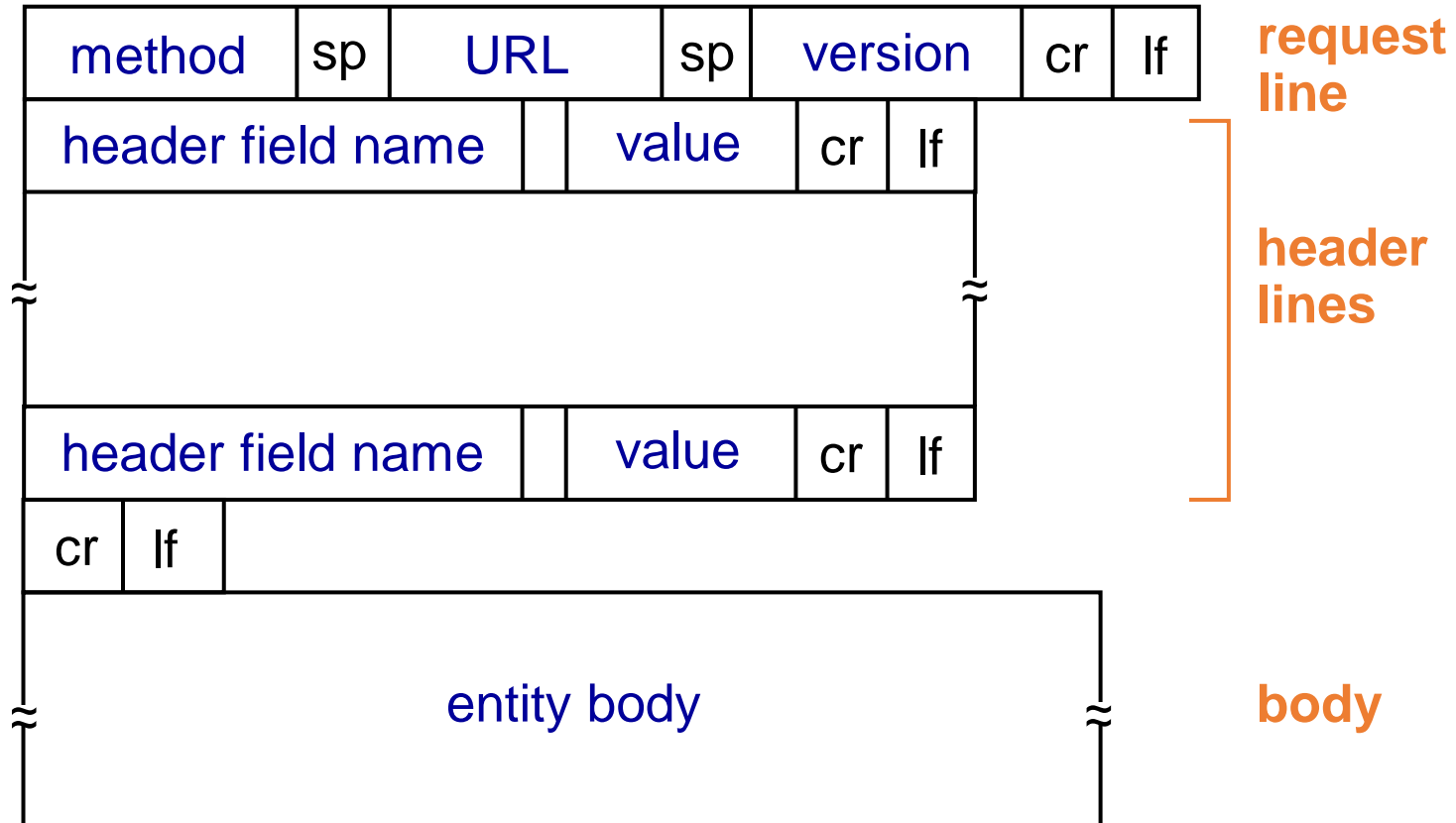
Non-persistent HTTP – Response Time

- Browsers may use multiple TCP connections to issue parallel requests
- Example loading a web page:
 - Maximum number of parallel connections: 2
 - *RTT* is 100 *ms*
 - Web page references 3 images
 - File transmission time and Web page parsing time can be ignored
- Issued requests:
 - Loading initial page (200 *ms*)
 - Loading the first two images (200 *ms*)
 - Loading the third image (200 *ms*)
- 600 *ms* are required to load the entire web page
- Again, Persistent HTTP and later HTTP versions can improve this performance

- Two types of HTTP messages: requests and responses
- HTTP request message:
 - ASCII (human-readable format)
 - Each line is terminated with `\r\n` (carriage return, line feed)
 - An empty line with only `\r\n` marks the end of HTTP headers
 - Simple example from a request to the informatics Web page:

```
GET /orgs/e191-03 HTTP/1.1
Host: informatics.tuwien.ac.at
User-Agent: Mozilla/5.0 Firefox/121.0
Accept: text/html
Accept-Language: de-AT
```

HTTP Request Message – General Format



Other HTTP Request Methods

POST Method

- Web page often includes inputs
- 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 '?'):
`https://tuw1.tuwien.ac.at/search/index.php?q=computersysteme`

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 the HTTP request message

See <https://www.rfc-editor.org/rfc/rfc9110.html> for a full list of HTTP methods.

HTTP Response Message

- Similar structure to request message
- Instead of path the response specifies a status code
- Simplified example of a response from the informatics Web server:

```
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 22 Jan 2024 16:55:43 GMT
Content-Type: text/html; charset=utf-8
```

```
<!DOCTYPE html>
```

```
...
```

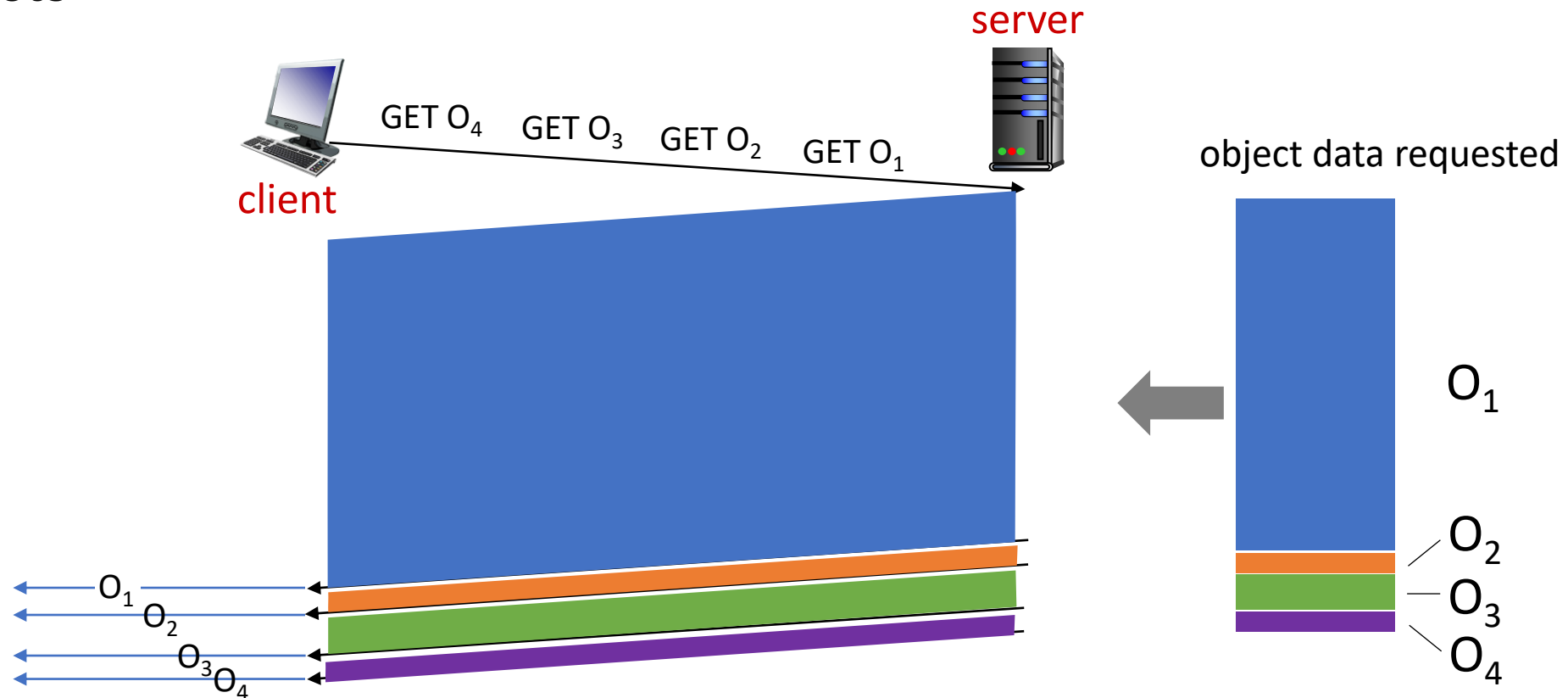
HTTP Response Status Codes

- Status code appears in 1st line in server-to-client response message
- Some examples:
 - **200 OK**
 - Request succeeded, requested object later in this message
 - **301 Moved Permanently**
 - Requested object moved, new location specified later in this message (in Location: field)
 - **400 Bad Request**
 - Request msg not understood by server
 - **404 Not Found**
 - Requested document not found on this server
 - **505 HTTP Version Not Supported**
 - Fun status codes in the Hyper Text Coffee Pot Control Protocol (HTCPCP, not relevant for tests)
 - 418 I'm a teapot
 - <https://www.rfc-editor.org/rfc/rfc2324> 😊

- 1991 – HTTP/0.9: only GET and HTML
- 1996 – HTTP/1.0: Non-persistent
- 1999 – HTTP/1.1: persistent
(keep alive)
- 2015 – HTTP/2: multiplexing avoid head-of-line blocking
(supported by most browsers)
- 2021 – HTTP/3: Using UDP; adds security and pipelining -error and congestion control
per object
(implementing reliable transfer within the application layer)

HTTP/2: mitigating HOL blocking

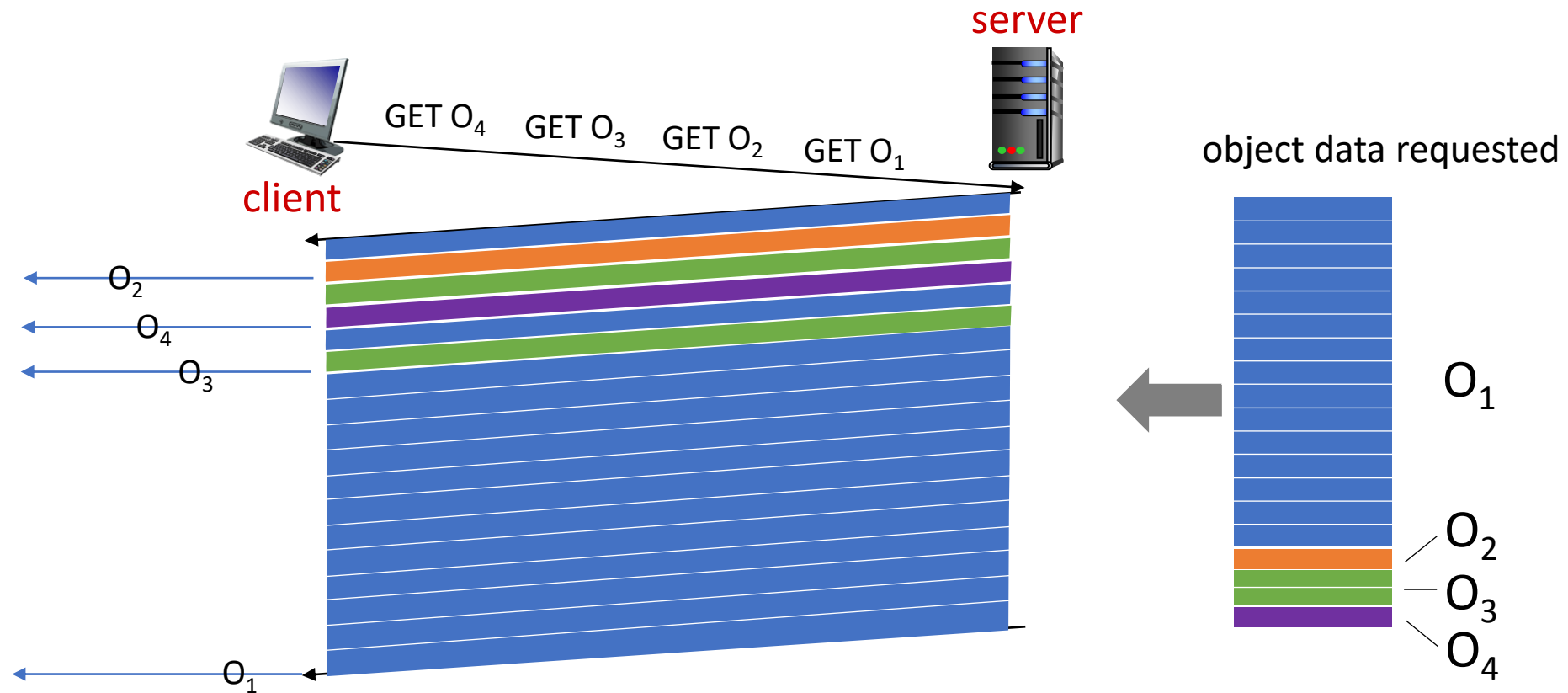
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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



O₂, O₃, O₄ delivered quickly, O₁ slightly delayed

Domain Name System

People have many identifiers

- SSN, name, passport number, ...

Internet hosts and routers

- IP address (32 bit) – used for addressing datagrams
- “name”, e.g., “informatics.tuwien.ac.at” used by humans

How do we map between the names and the IP addresses?

Domain Name System (DNS)

- Distributed database implemented in hierarchy of many **name servers**
- Application-layer protocol: hosts, DNS servers communicate to **resolve** names (address/name translation)
 - Note: core Internet function, **implemented as application-layer protocol**
 - Complexity at network’s “edge”

DNS services

- Hostname-to-IP-address translation
- Host aliasing
 - Canonical, alias names
- Mail server aliasing
- Load distribution
 - Replicated Web servers: many IP addresses correspond to one name

Why not centralize DNS?

- Single point of failure
- Traffic volume
- Distant centralized database

Because it doesn't scale!

- Comcast DNS server: 600 billion DNS queries per day
- Akamai DNS server: 2.2 trillion DNS queries per day

Thinking About the DNS

Humongous distributed database

- ~ billions of records, each simple

Handles many trillions of queries per day

- Many more reads than writes
- Performance matters: almost every Internet transaction interacts with DNS - msec count!

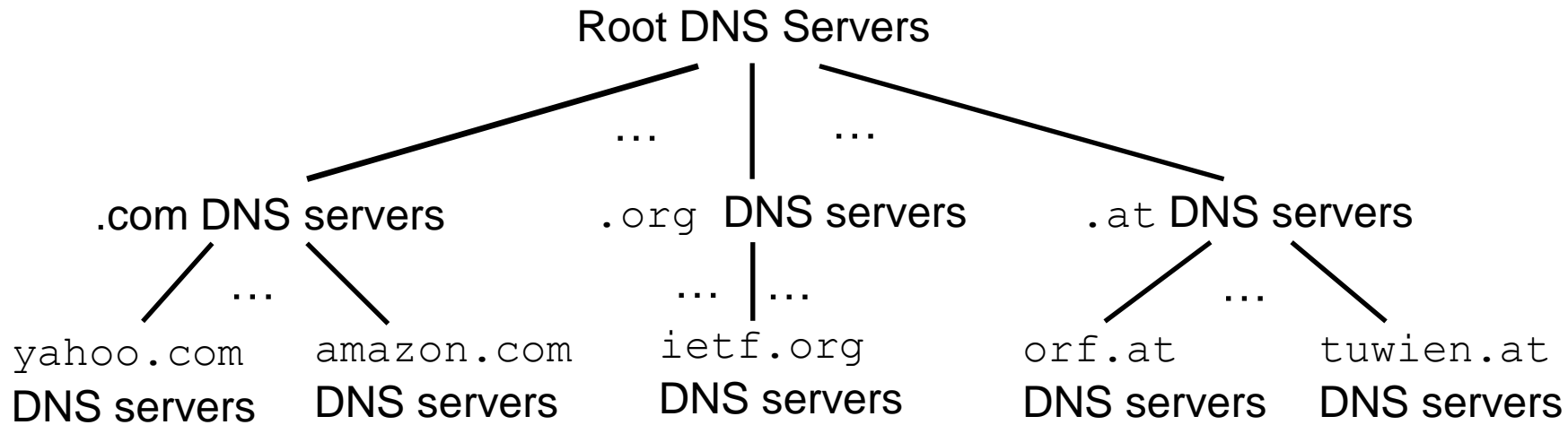
Organizationally, physically decentralized

- Millions of different organizations responsible for their records

Must be “bulletproof”: reliability and security



DNS: A Distributed, Hierarchical Database



Root

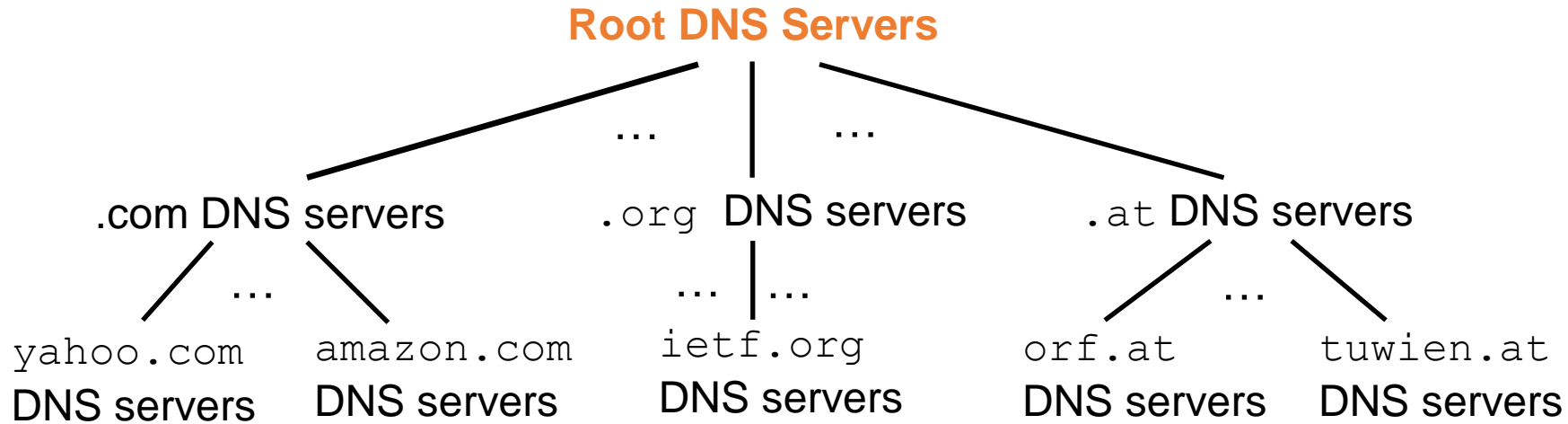
Top-Level Domain

Authoritative

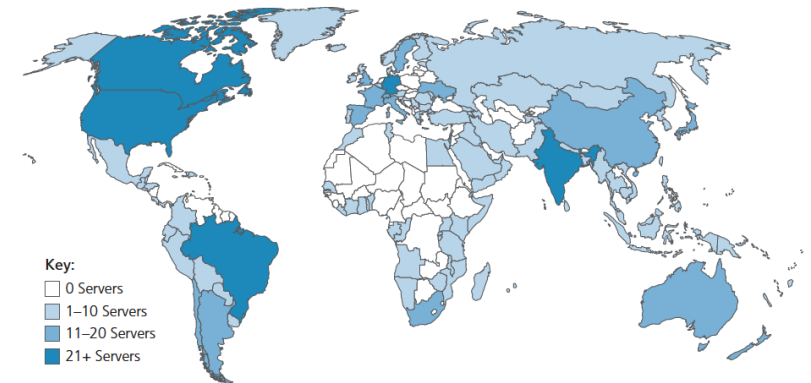
Client wants IP address for `www.tuwien.at`; 1st approximation:

- client queries root server to find `.at` DNS server
- client queries `.at` DNS server to get `tuwien.at` DNS server
- client queries `tuwien.at` DNS server to get IP address for `www.tuwien.at`

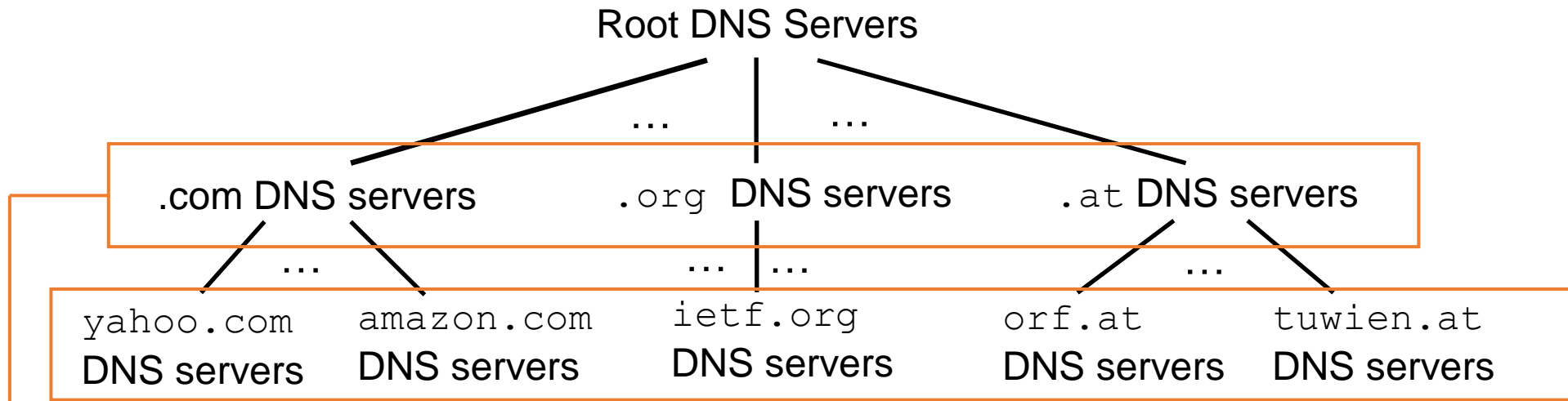
DNS Root Name Servers



- Official, contact-of-last-resort by name servers that can not resolve name
- Incredibly important Internet function
 - Internet couldn't function without it!
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain



Top-Level Domain and Authoritative Servers



Top-Level Domain (TLD) Servers

- Responsible for .at, .com, .net, .org, ...
- nic.at GmbH manages the .at domain

Authoritative 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

- When host makes DNS query, it is sent to its **local DNS server**
 - Local DNS server returns reply, answering:
 - From its **local cache** of recent name-to-address translation pairs (possibly out of date!)
 - Forwarding request into DNS hierarchy for resolution
 - Each ISP has local DNS name server; to find yours:
 - MacOS: `% scutil --dns`
 - Windows: `>ipconfig /all`
 - Linux: e.g., `nmcli device show <interfacename> | grep IP4.DNS`
- Local DNS server doesn't strictly belong to hierarchy

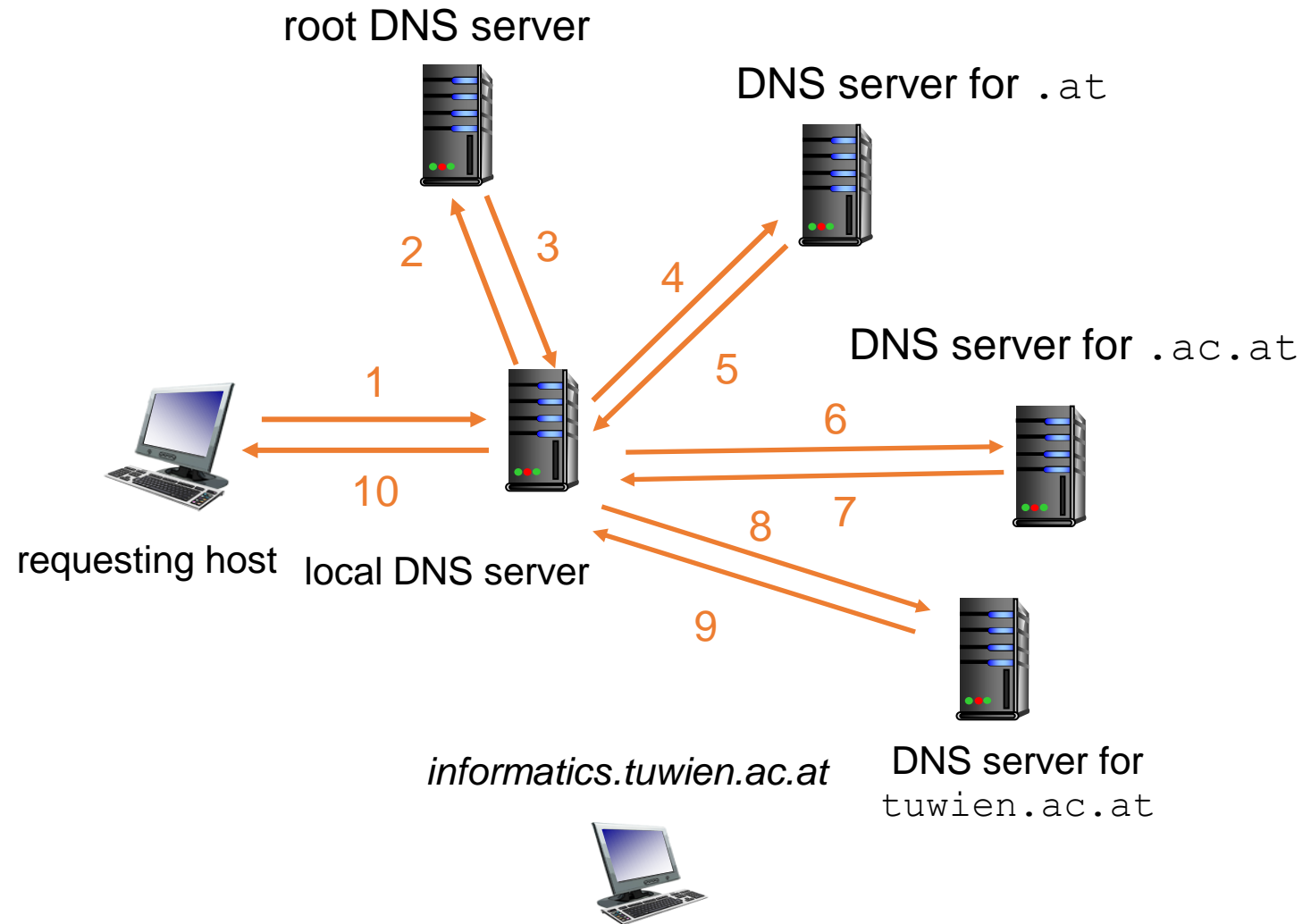
DNS Name Resolution: Iterated Query

- Example: host wants IP address of `informatics.tuwien.ac.at`

Iterated query

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

There is also a recursive query approach that we will not discuss further



Caching DNS Information

- Once (any) name server learns mapping, it **caches** mapping, and **immediately** returns a cached mapping in response to a query
 - Caching improves response time
 - Cache entries timeout (disappear) after some time (TTL)
 - TLD servers typically cached in local name servers
- Cached entries may be **out-of-date**
 - If named host changes IP address, may not be known Internet-wide until all TTLs expire!
 - **Best-effort name-to-address translation!**

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

RR format: (name, value, type, ttl)

type=A

- name is hostname
- value is IP address

type=NS

- name is domain (e.g., `tuwien.ac.at`)
- value is IP address is hostname of authoritative name server for this domain (e.g., `tunamed.tuwien.ac.at`.)

type=CNAME

- name is alias name for some “canonical” (the real) name
- value is the canonical name
- For example, `www.tuwien.at` is an alias for `www.tuwien.ac.at`

type=MX

- value is name of SMTP mail server associated with name

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

← 2 bytes → ← 2 bytes →

identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

DNS protocol messages

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

← 2 bytes → ← 2 bytes →

identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

name, type fields for a query

RRs in response to query

records for authoritative servers

additional “helpful” info that may be used

Getting your info into the DNS

example: new startup “Network Utopia”

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

- Introduction to the Internet
 - What is the Internet and a protocol?
 - Network edge: hosts, access network
 - Network core: packet/circuit switching, internet structure
 - Protocol layers, service models
- Application Layer
 - Process Communication (Sockets)
 - HTTP
 - DNS



Informatics

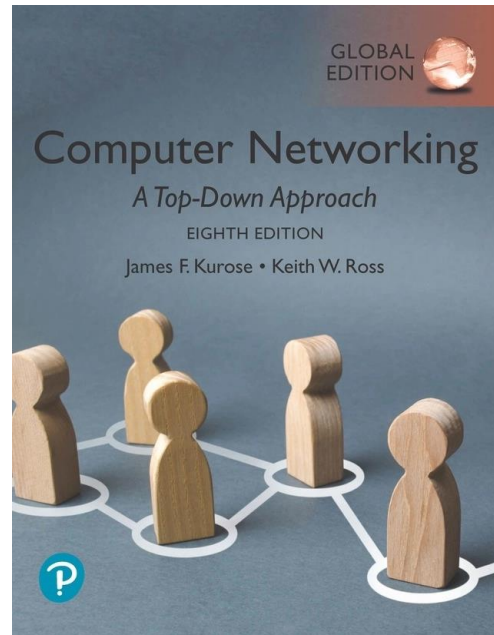


Computer Systems

Networks – Transport Layer

Gernot Steindl

03.06.2024



- Literature: „Computer Networking – A Top-Down Approach“, written by James F. Kurose and Keith W. Ross
 - <https://www.pearson.de/computer-networking-global-edition>
 - https://gaia.cs.umass.edu/kurose_ross/index.php (Includes resources for students!)
 - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU’s library: https://catalogplus.tuwien.at/permalink/f/8j3js/UTW_alma21140332460003336

- In this lecture we want to ...
 - understand principles behind transport layer services
 - Multiplexing, demultiplexing
 - Reliable Data Transfer
 - Flow control
 - Congestion Control
 - learn about the 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



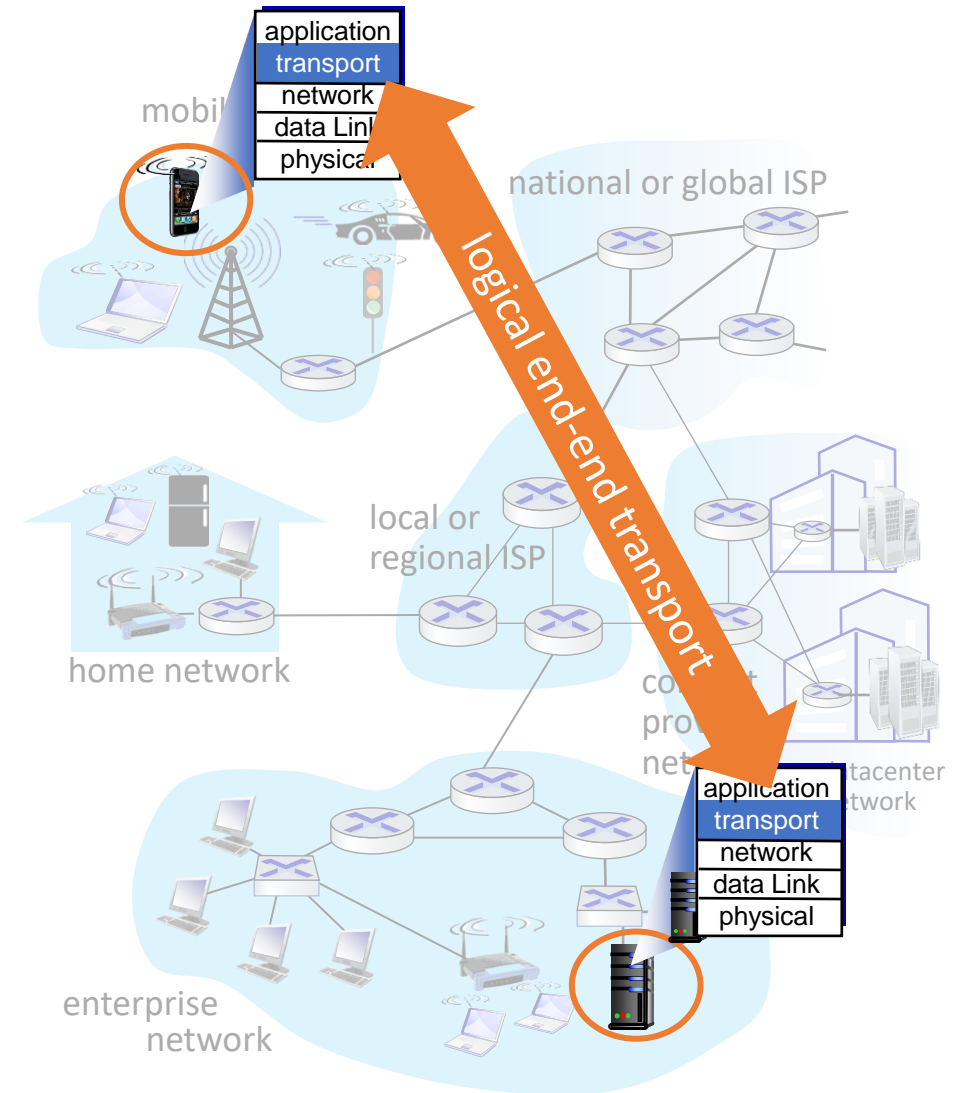
Transport layer: roadmap

- **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



Transport Services and Protocols

- 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 are available to Internet applications
 - UDP and TCP



Transport vs. Network Layer Services and Protocols

Analogy in a household:

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

- Hosts = houses
- Processes = kids
- App messages = letters in envelopes
- Transport protocol = Ann and Bill who demultiplex to in-house siblings
- Network-layer protocol = postal service



Transport vs. Network Layer Services and Protocols

Analogy in a household:

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

- Hosts = houses
- Processes = kids
- App messages = letters in envelopes
- Transport protocol = Ann and Bill who demultiplex to in-house siblings
- Network-layer protocol = postal service

- **Transport Layer:**

- Communication between **processes**
- Can provide some additional services to the application layer
- Relies on network layer services

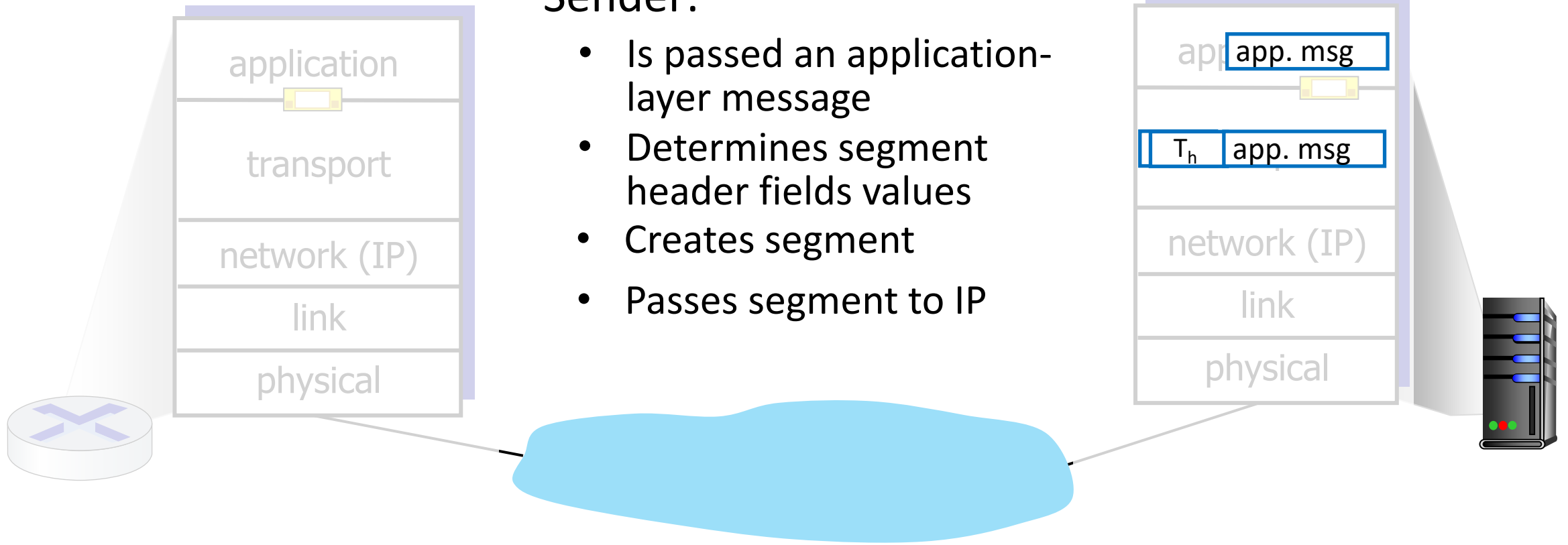
- **Network Layer:**

- Communication between **hosts**

Transport Layer Actions

Sender:

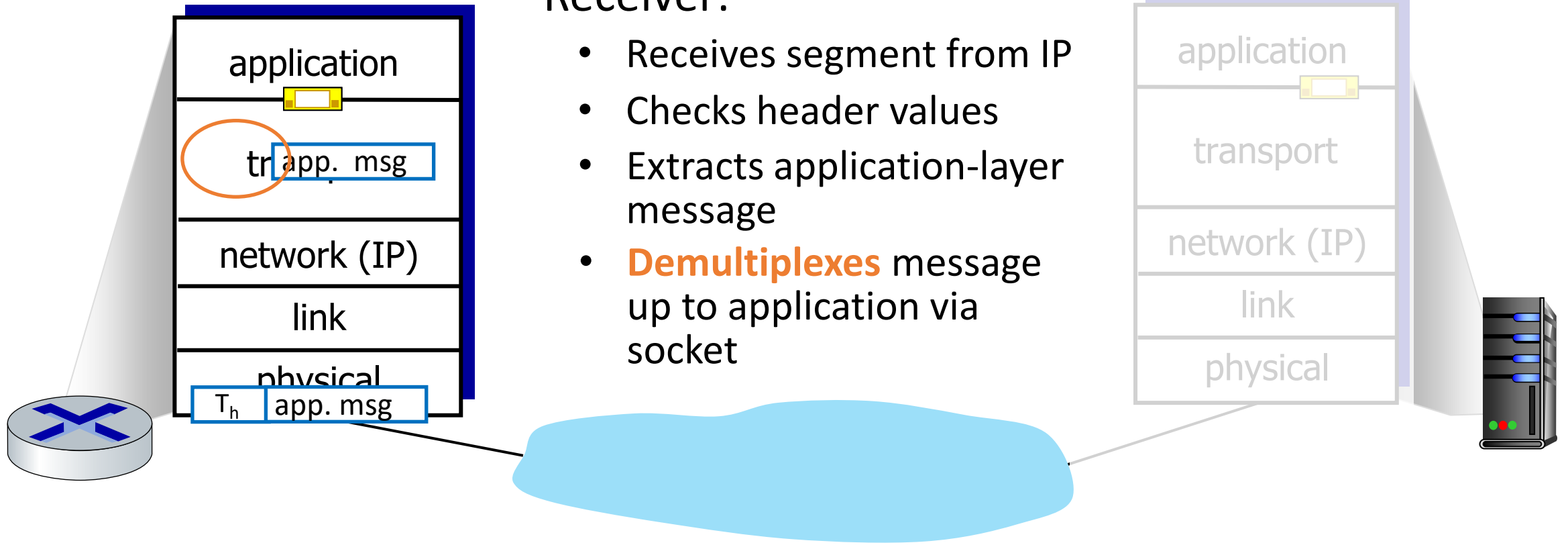
- Is passed an application-layer message
- Determines segment header fields values
- Creates segment
- Passes segment to IP



Transport Layer Actions

Receiver:

- Receives segment from IP
- Checks header values
- Extracts application-layer message
- **Demultiplexes** message up to application via socket



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 in both protocols:

- Delay guarantees
- Bandwidth guarantees

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



Transport vs. network layer services and protocols

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

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

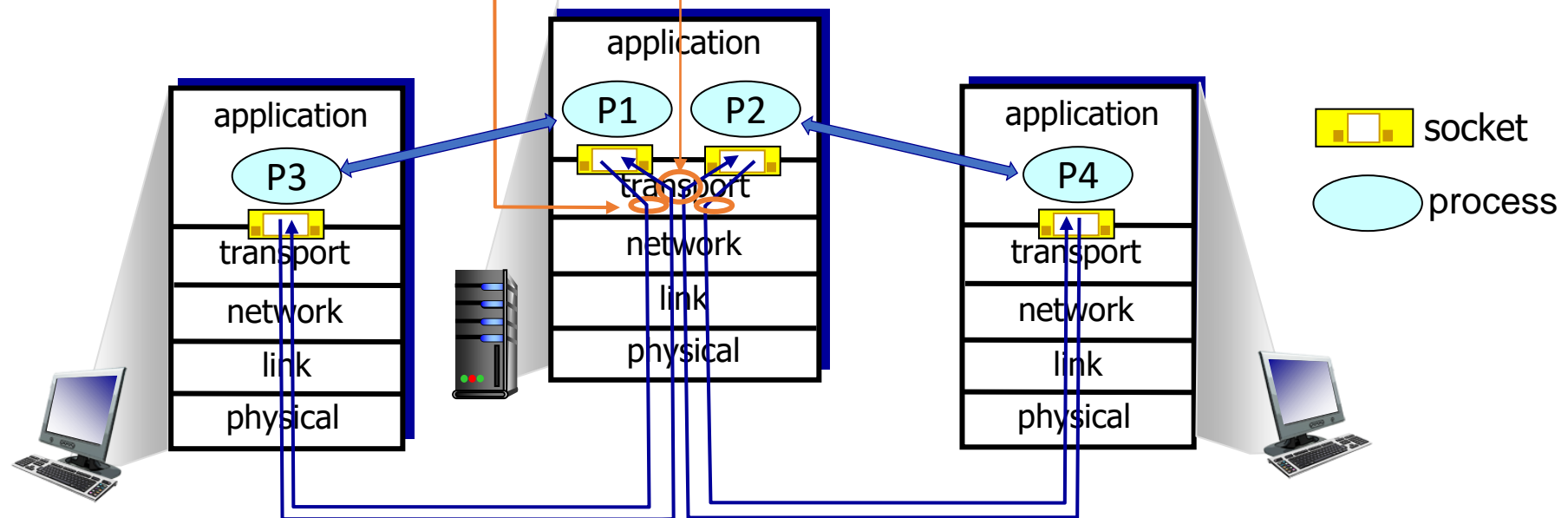
Multiplexing and Demultiplexing

Multiplexing as sender

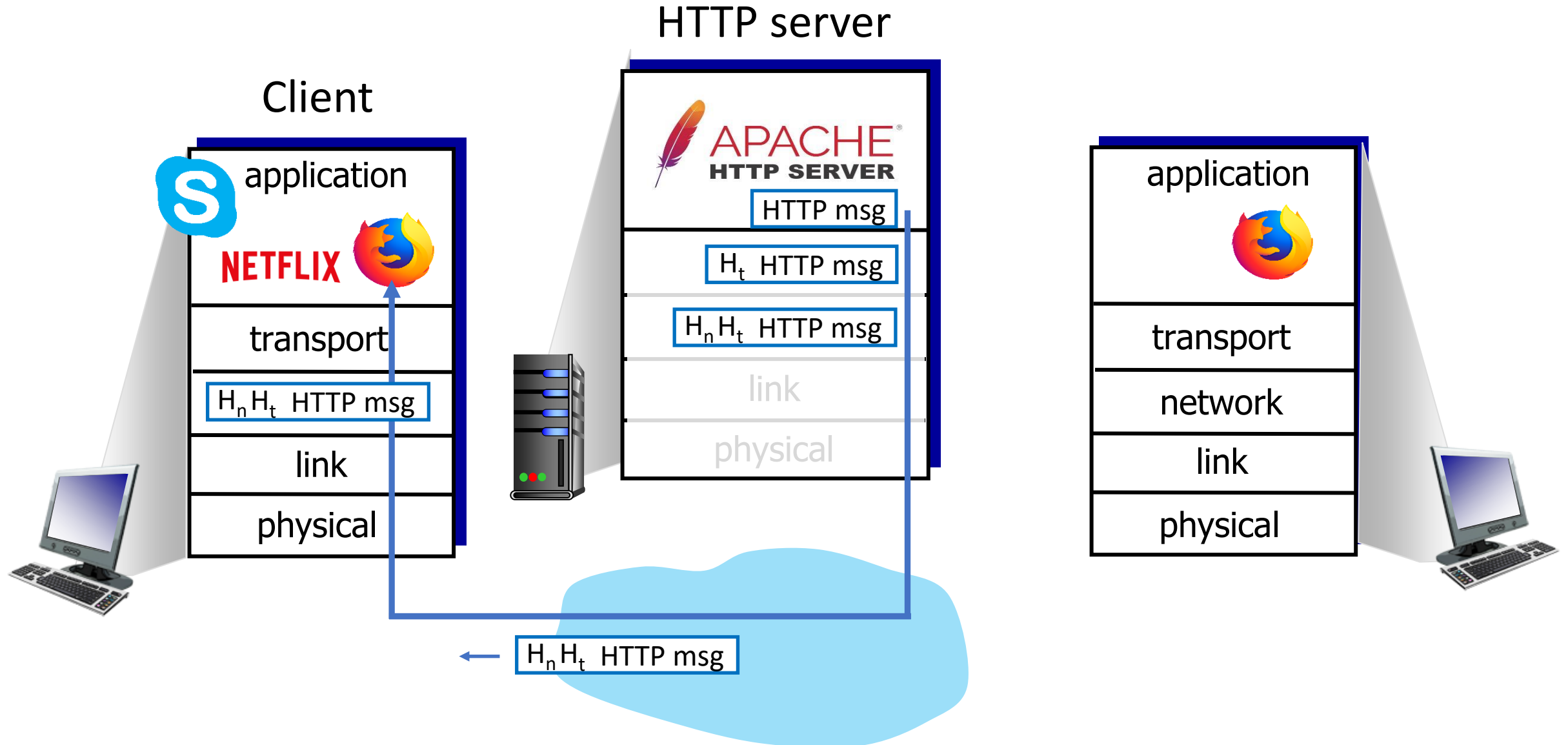
handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing as receiver

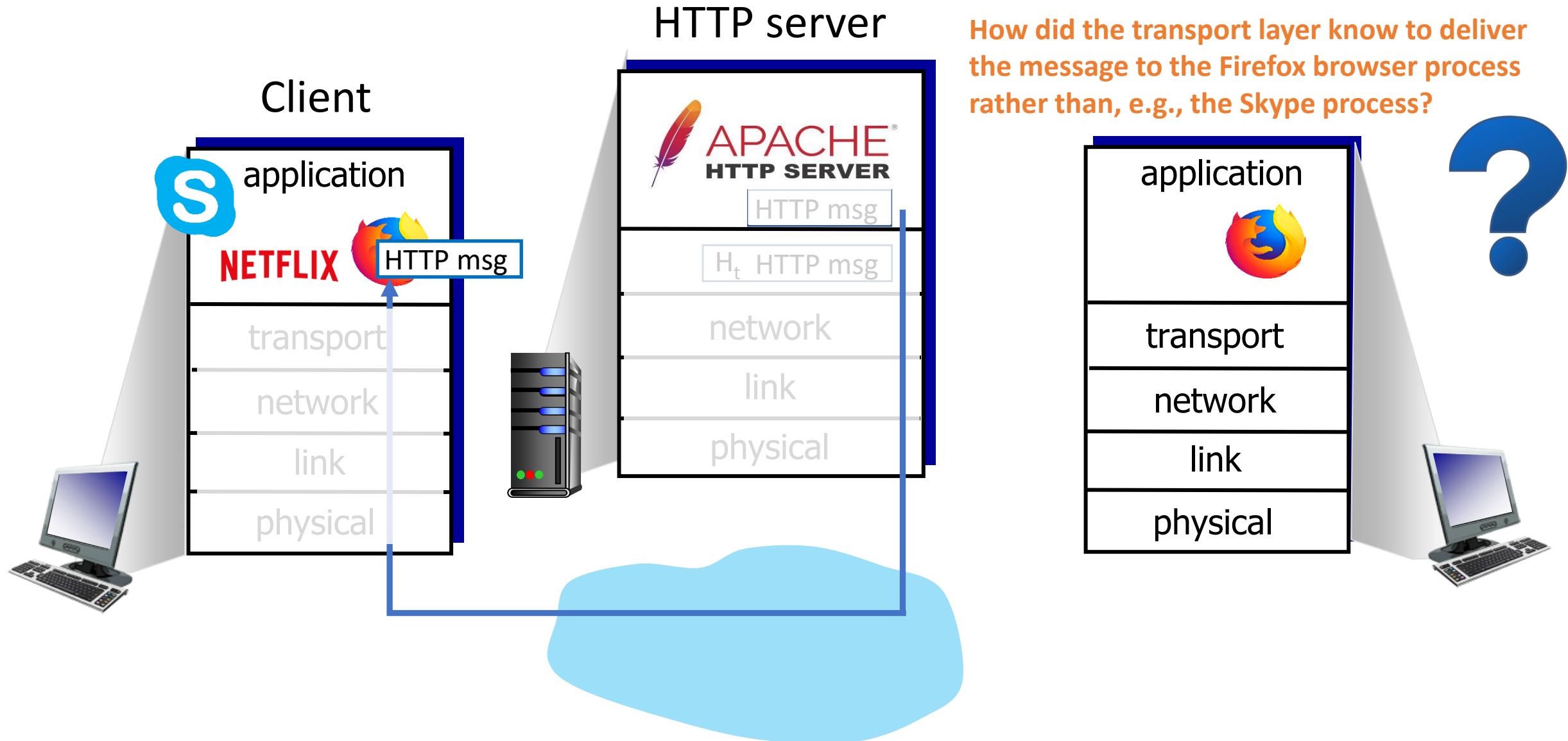
use header info to deliver received segments to correct socket



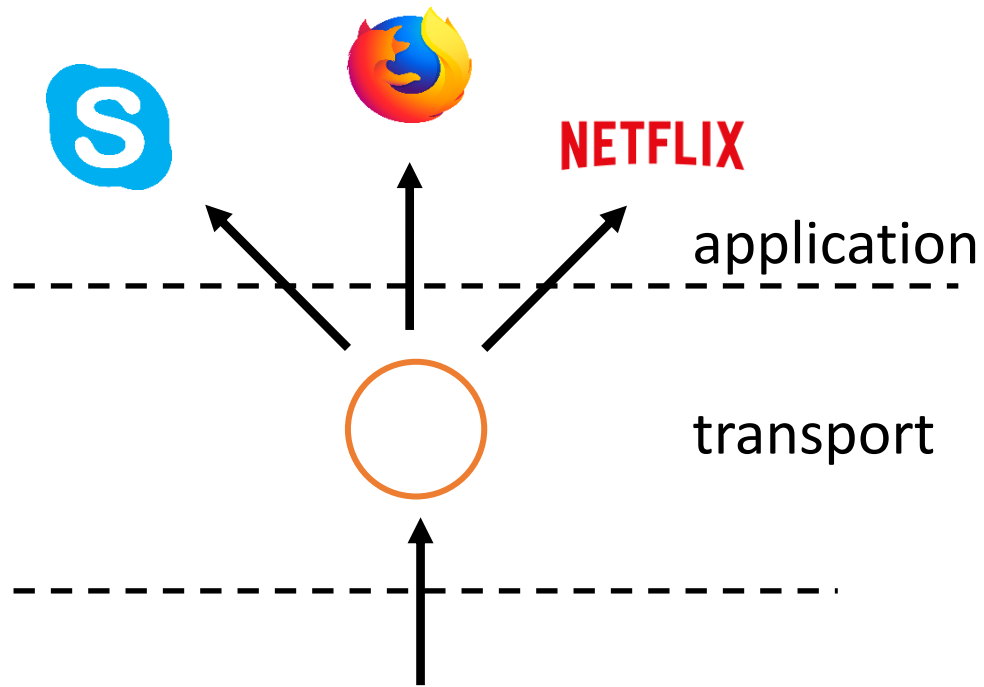
Multiplexing and Demultiplexing



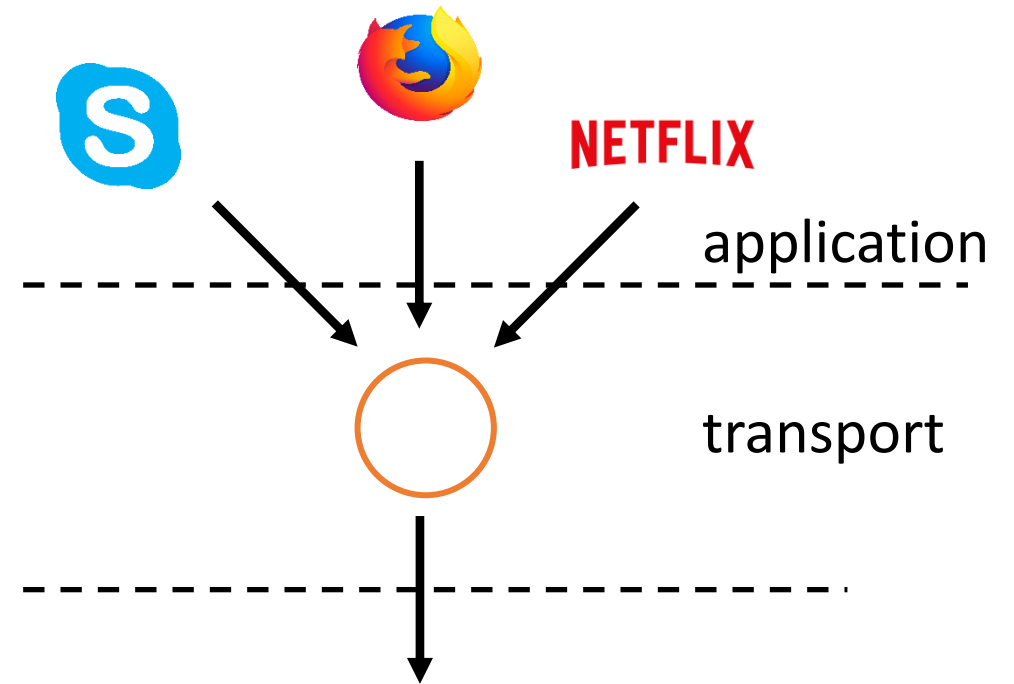
Multiplexing and Demultiplexing



Multiplexing and Demultiplexing



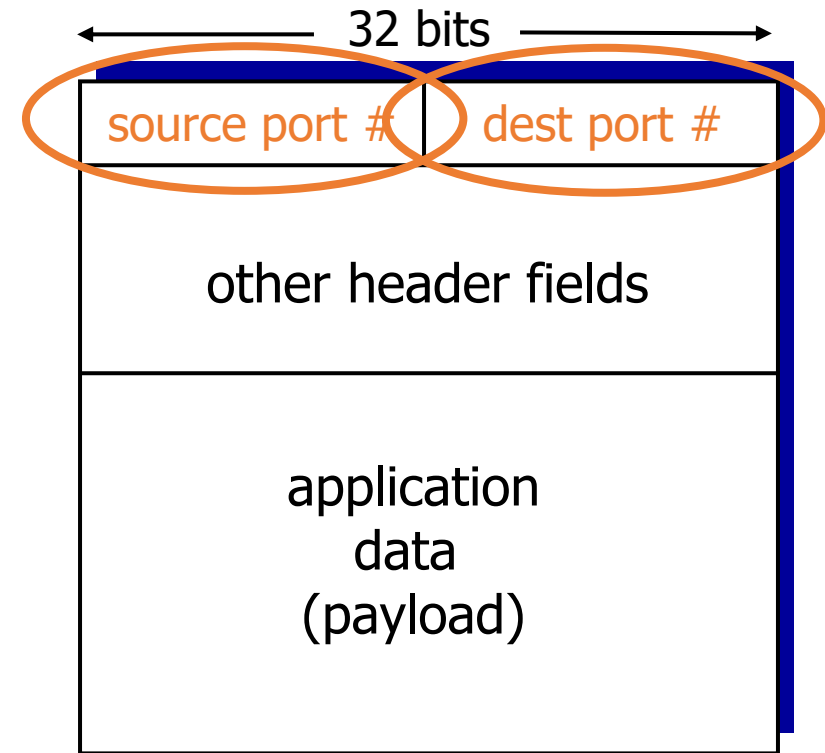
Demultiplexing



Multiplexing

How Demultiplexing Works

- Host receives IP datagrams
 - 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
 - Which information is used depends on the protocol



TCP/UDP segment format

Connectionless Demultiplexing

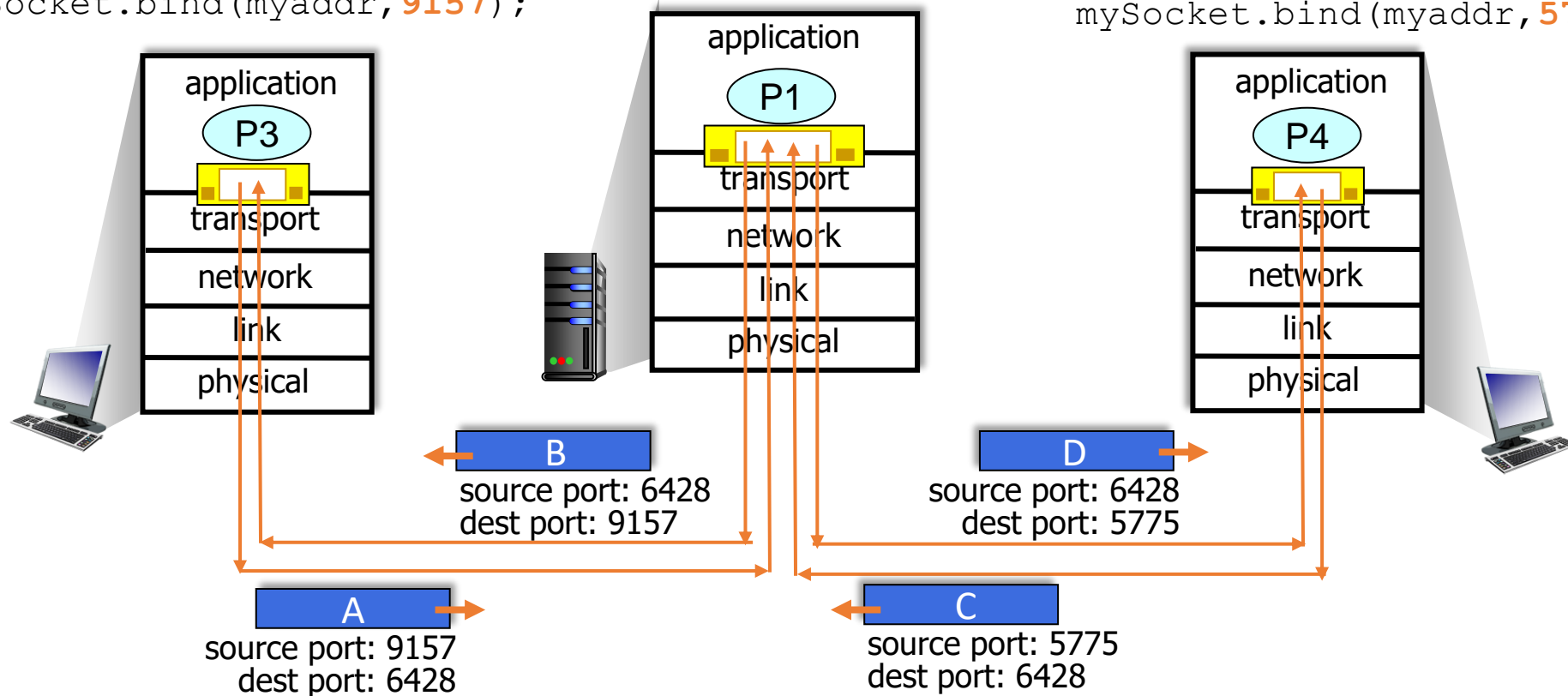
- When creating socket, must specify **host-local** port number:
 - `DatagramSocket s =
new DatagramSocket(12534);`
- When a sender creates a datagram to send into its UDP socket, it must specify:
 - Destination IP address
 - Destination port #
- When receiving host receives UDP segment:
 - Checks destination port number in segment
 - Directs UDP segment to socket with that port number
- IP/UDP datagrams with the **same destination port number**, 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_STREAM)  
mySocket.bind(myaddr, 9157);
```

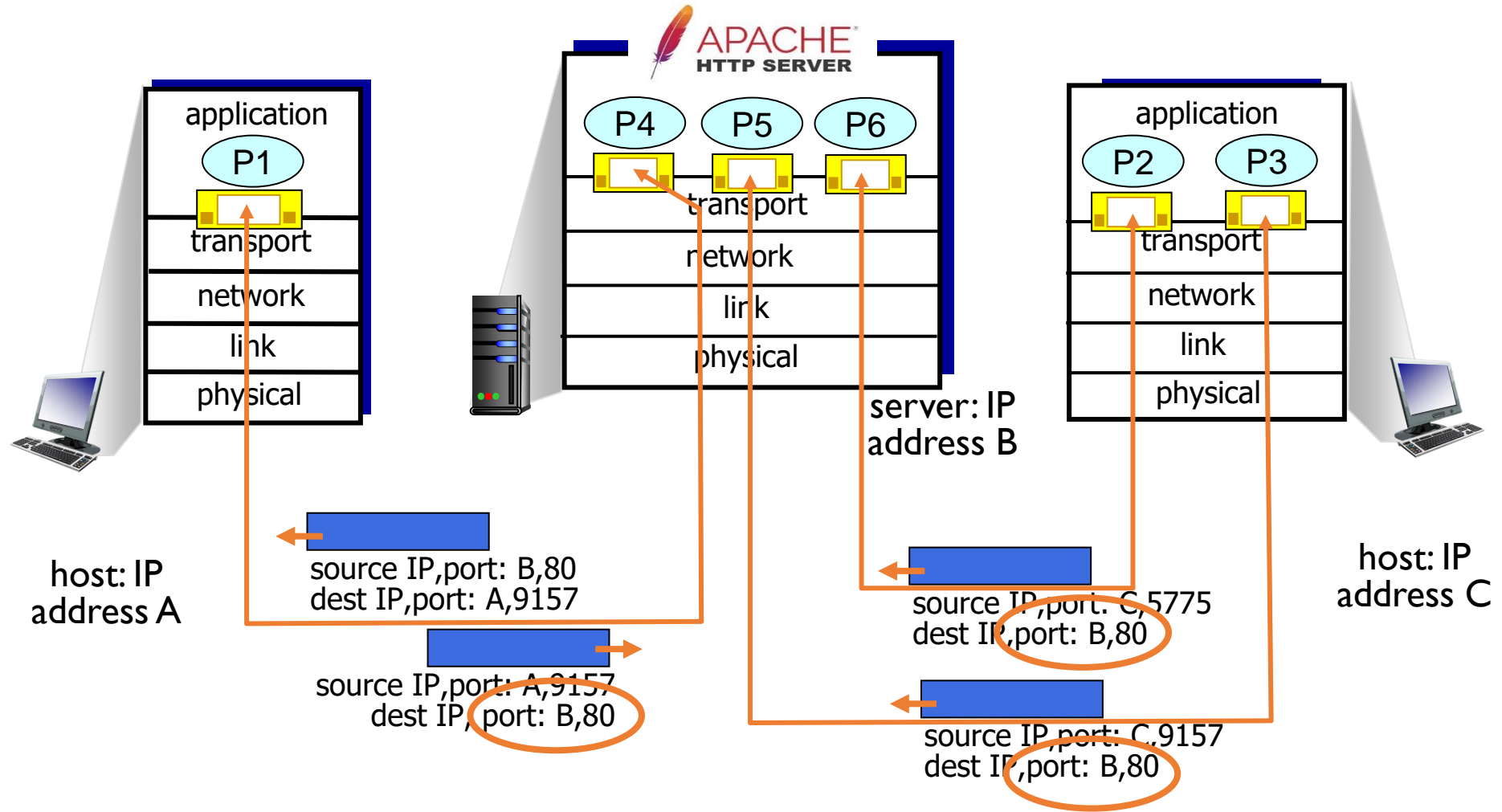
```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



Connection-Oriented Demultiplexing

- TCP socket identified by 4-tuple:
 - Source IP address
 - Source port number
 - Destination IP address
 - Destination port number
- Demultiplexer: receiver uses **all four values** 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: An Example



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

- 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
- Multiplexing/demultiplexing happen at all layers

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



UDP: User Datagram Protocol

- “Bare bone” Internet transport protocol
- “Best effort” service, segments may be:
 - Lost
 - Delivered out-of-order to the app
- **Connectionless**
 - No handshaking between UDP sender, receiver
 - Each UDP segment handled independently of others

Why is there a UDP?

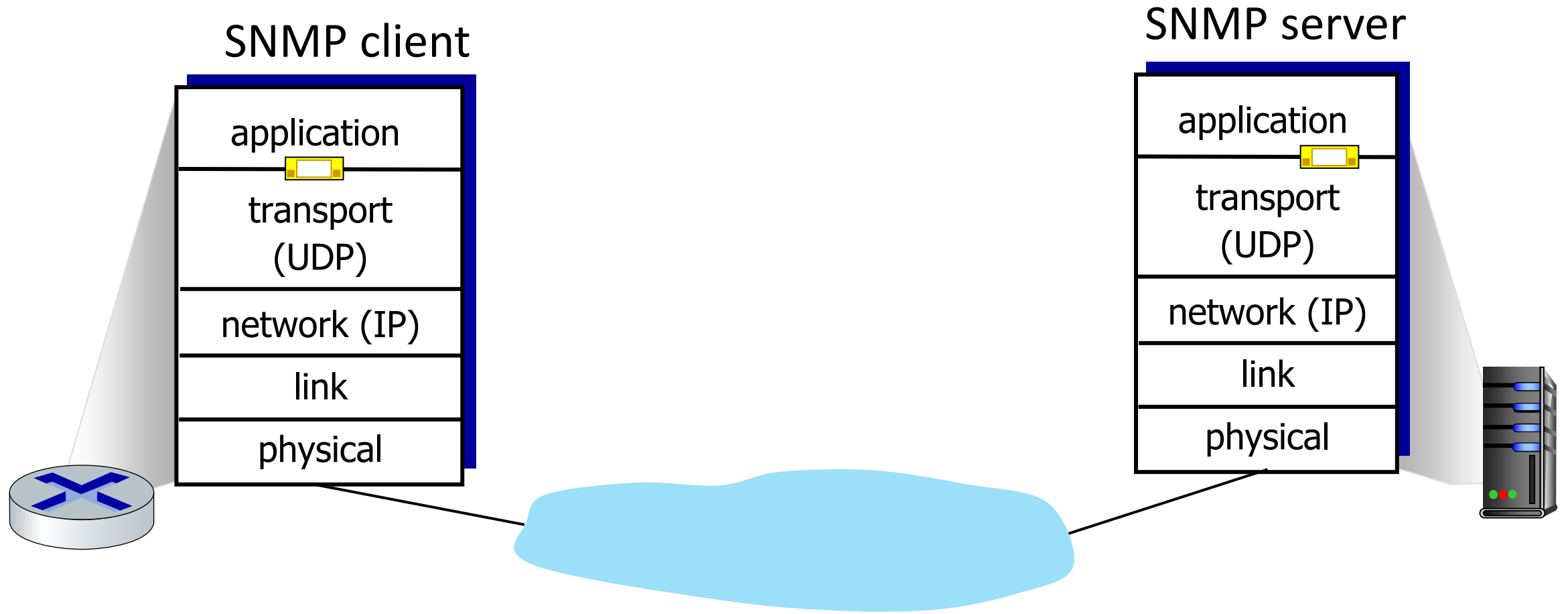
- No connection establishment (which can add RTT delay)
- Simple: no connection state at sender, receiver
- Small header size
- No congestion control
 - UDP can blast away as fast as desired!
 - Can function in the face of congestion

UDP: User Datagram Protocol

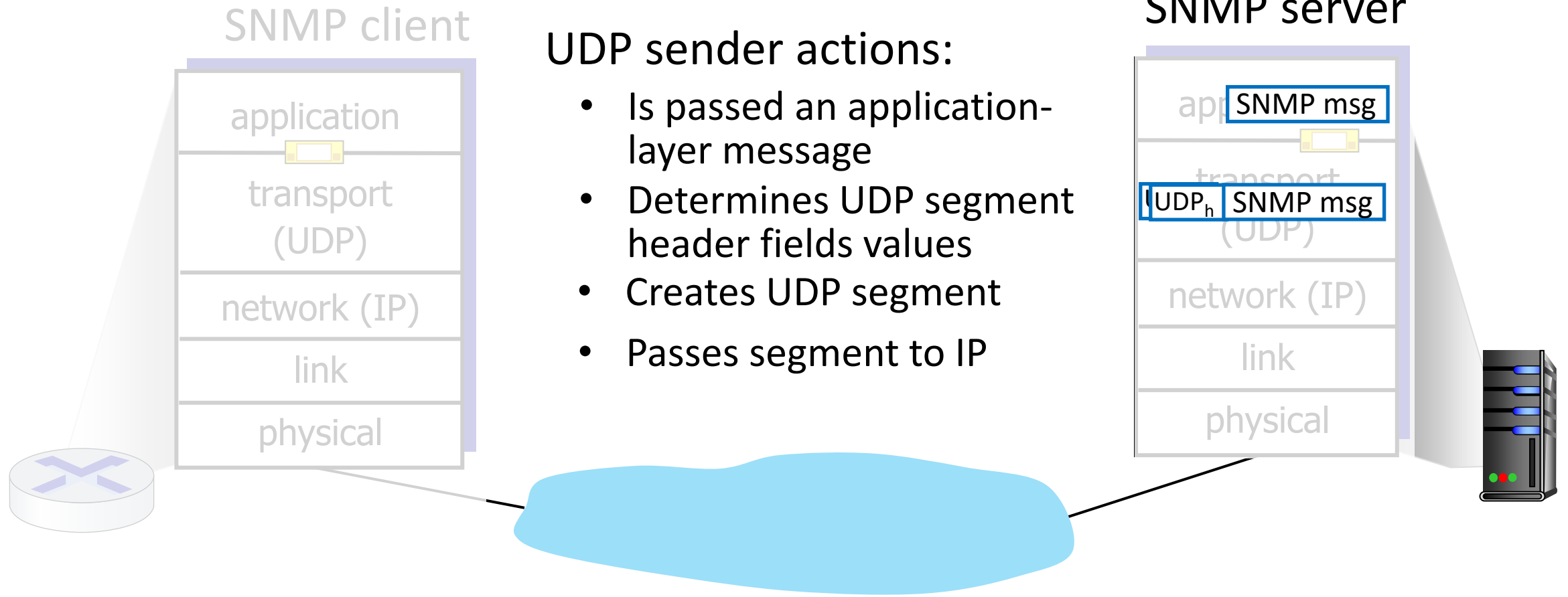
- UDP use:
 - Streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - HTTP/3
- If reliable transfer needed over UDP (e.g., HTTP/3):
 - Add needed reliability at application layer
 - Add congestion control at application layer

<https://datatracker.ietf.org/doc/html/rfc768>

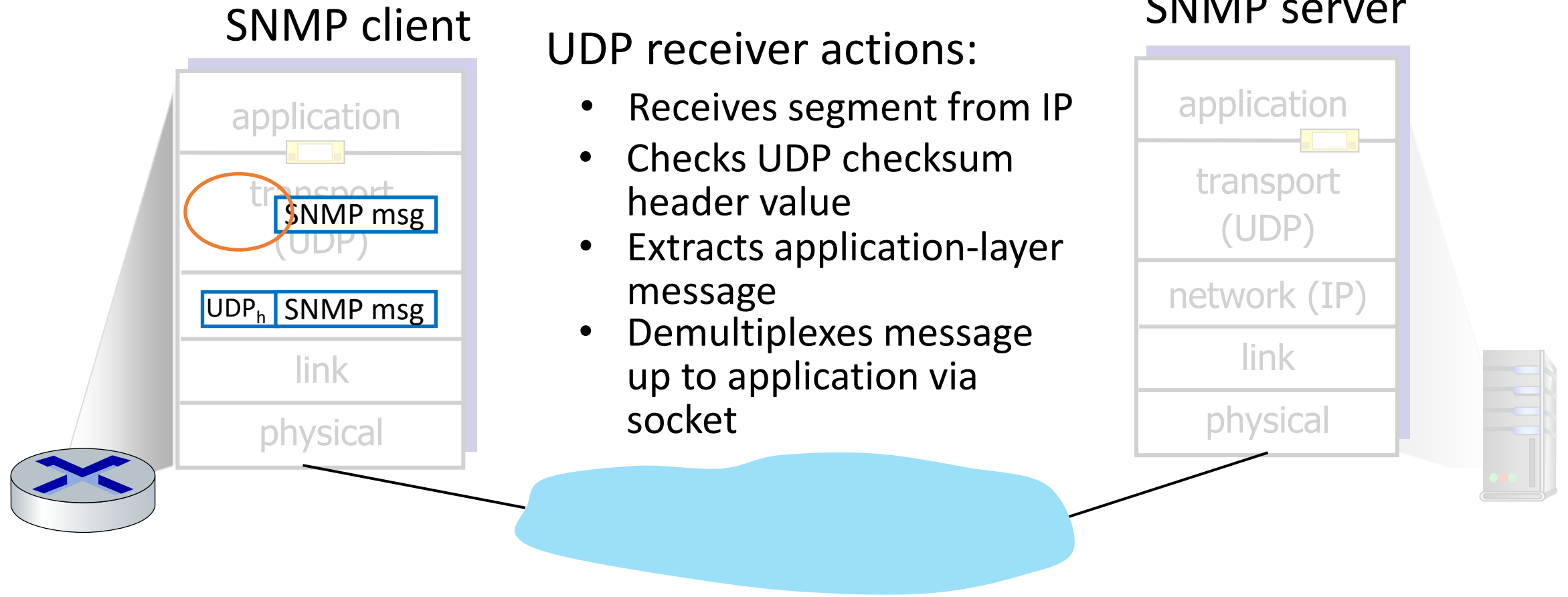
UDP: Transport Layer Actions



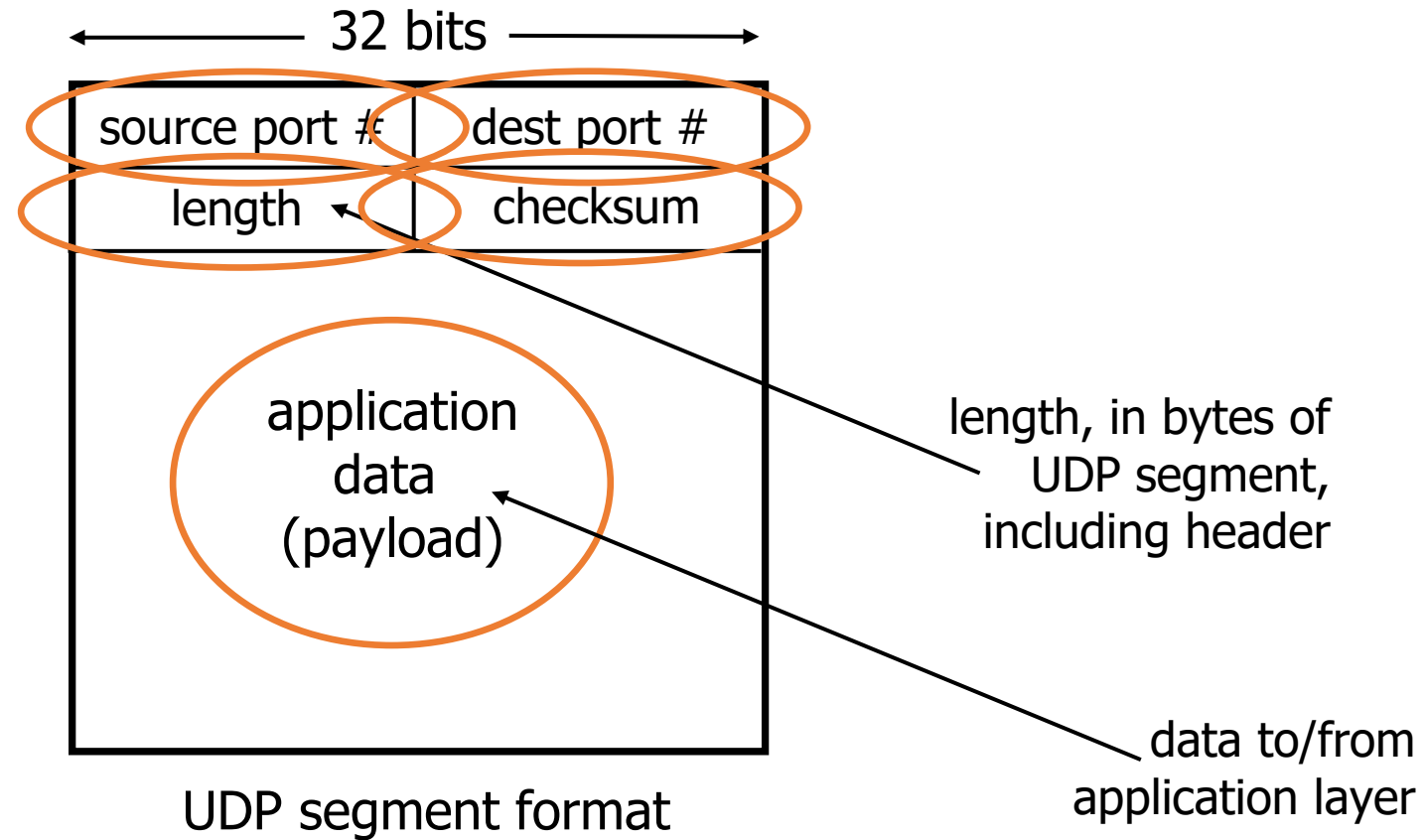
UDP: Transport Layer Actions



UDP: Transport Layer Actions

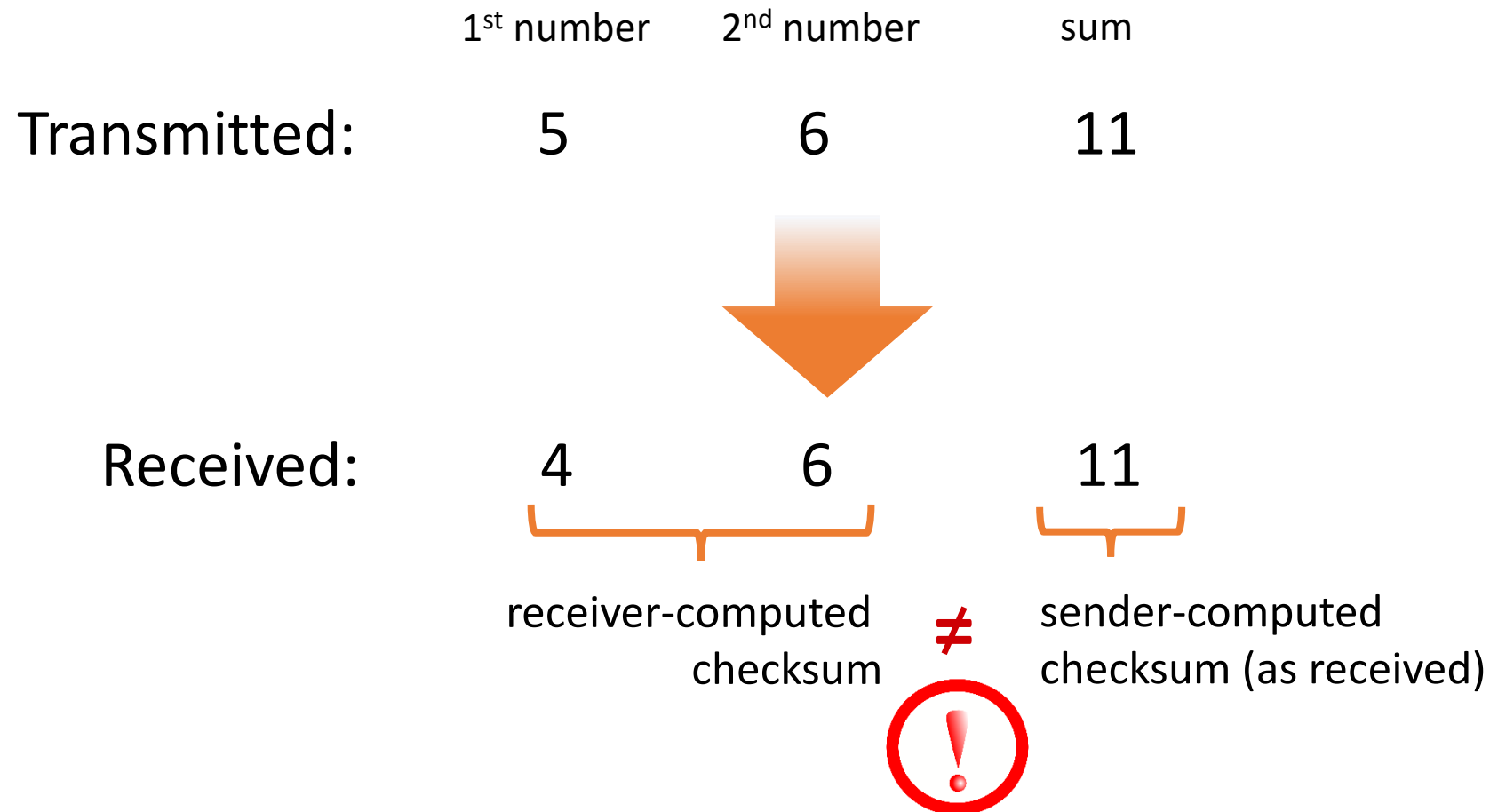


UDP Segment Header



UDP Checksum

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



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

Sender

- Treat content of 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 occurred? More later

Internet Checksum: An Example

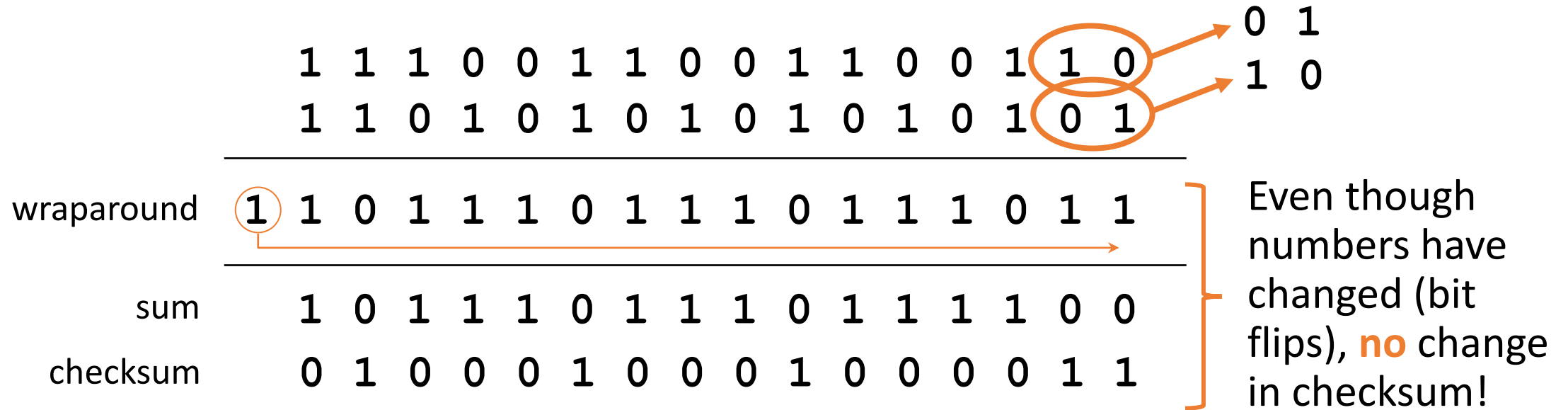
Example: Add two 16-bit integers

	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	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	

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



Summary: UDP

- “No frills” protocol:
 - Segments may be lost
 - Segments may be 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)

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

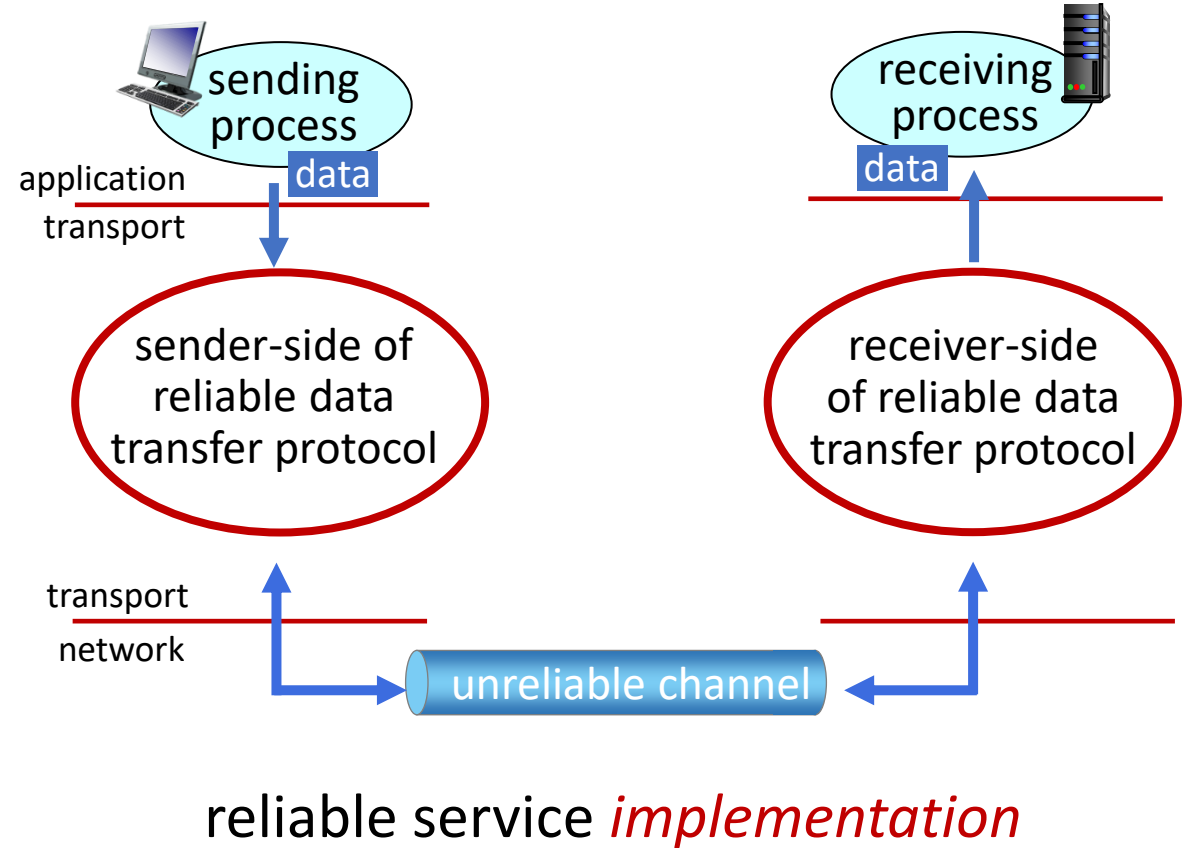
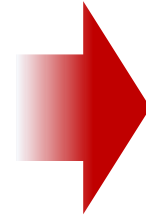
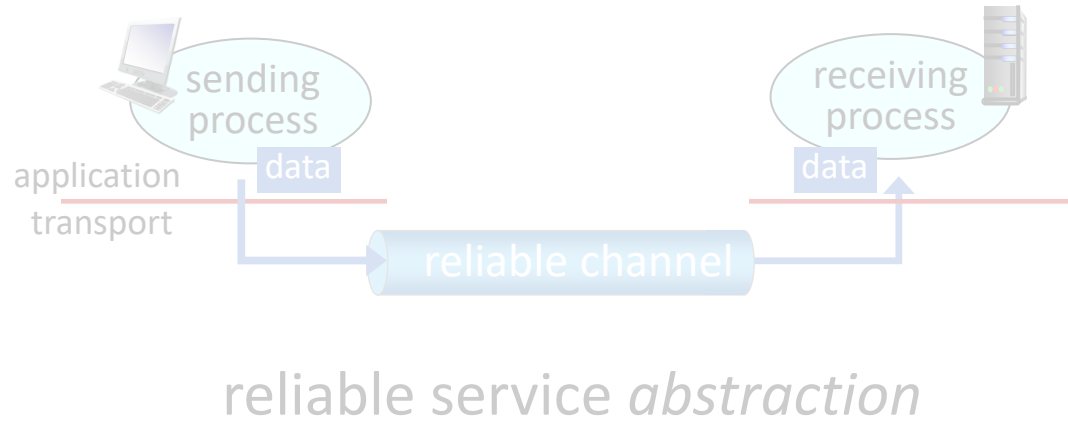


Principles of reliable data transfer



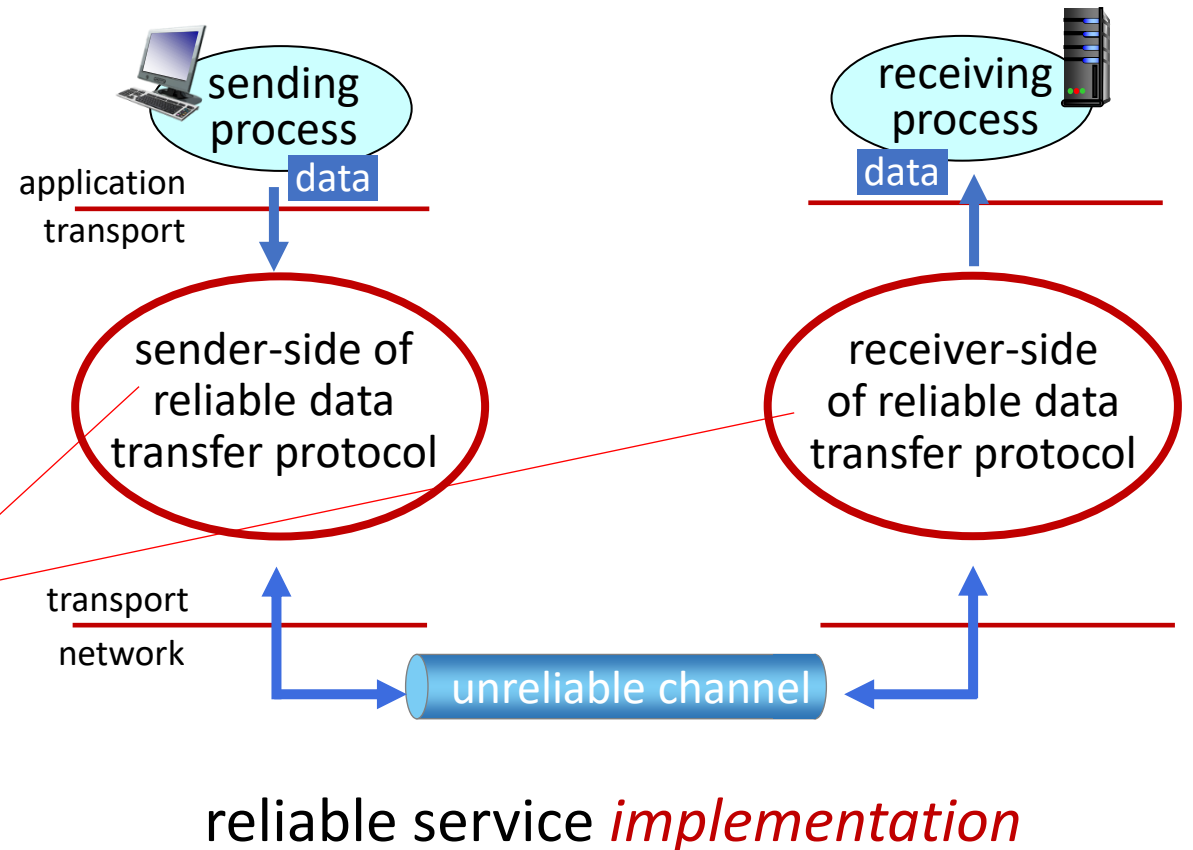
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

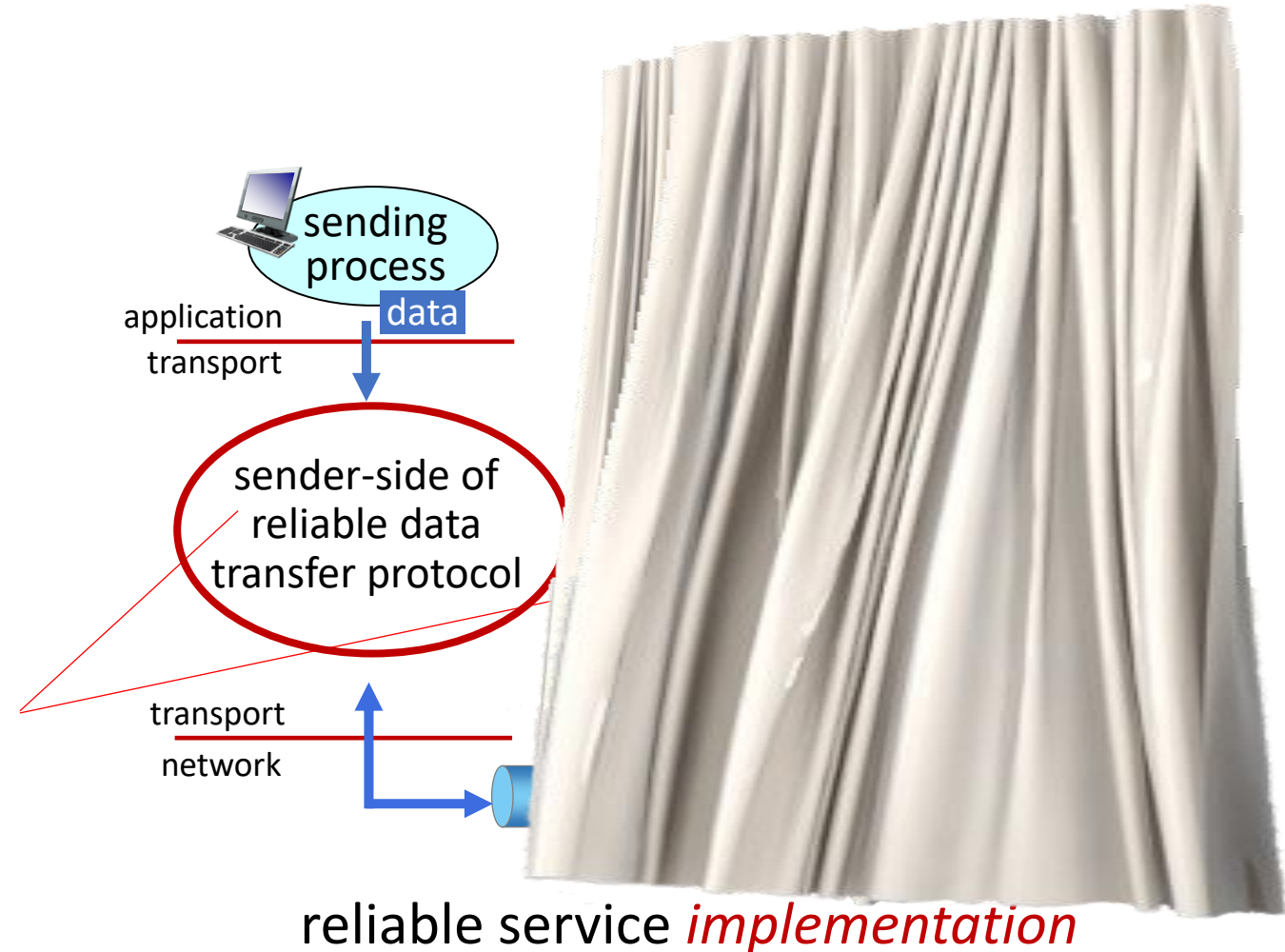
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



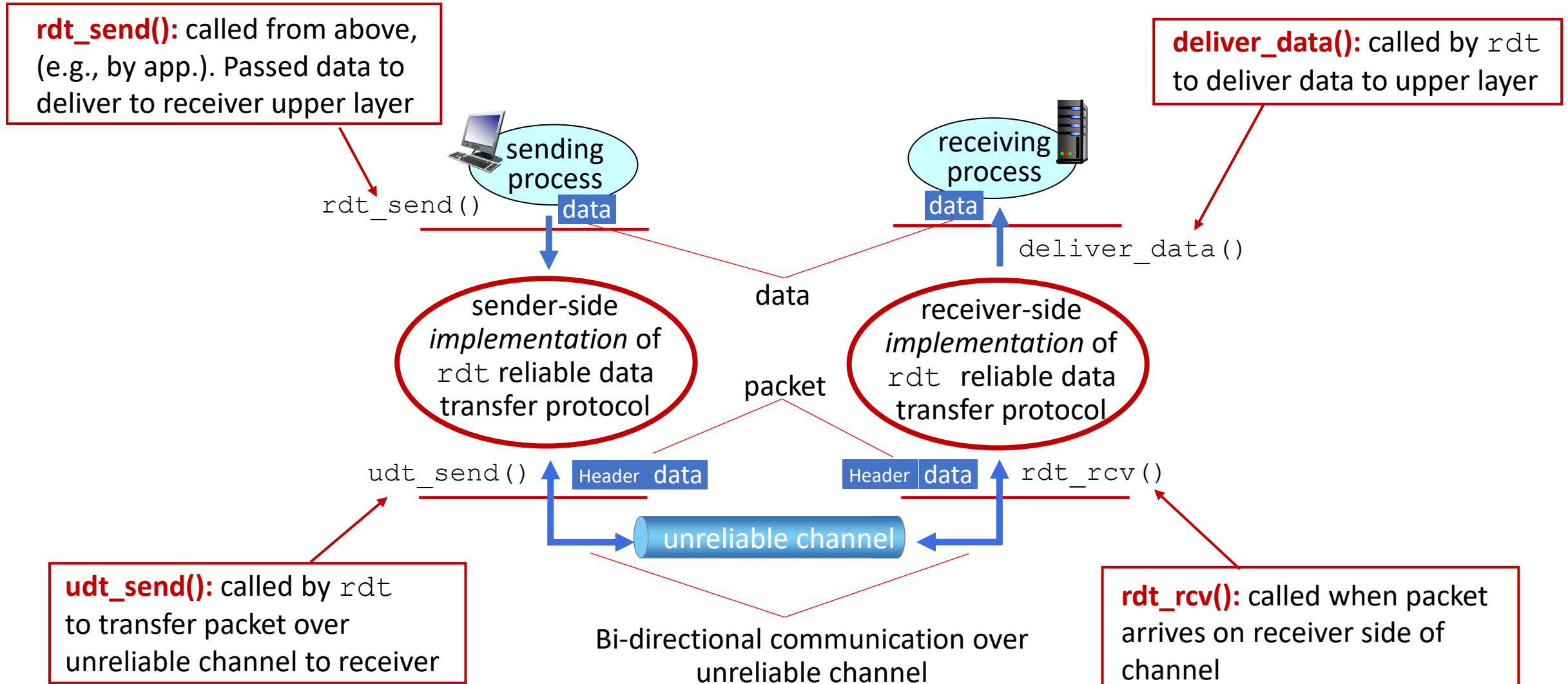
Principles of reliable data transfer

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

- unless communicated via a message



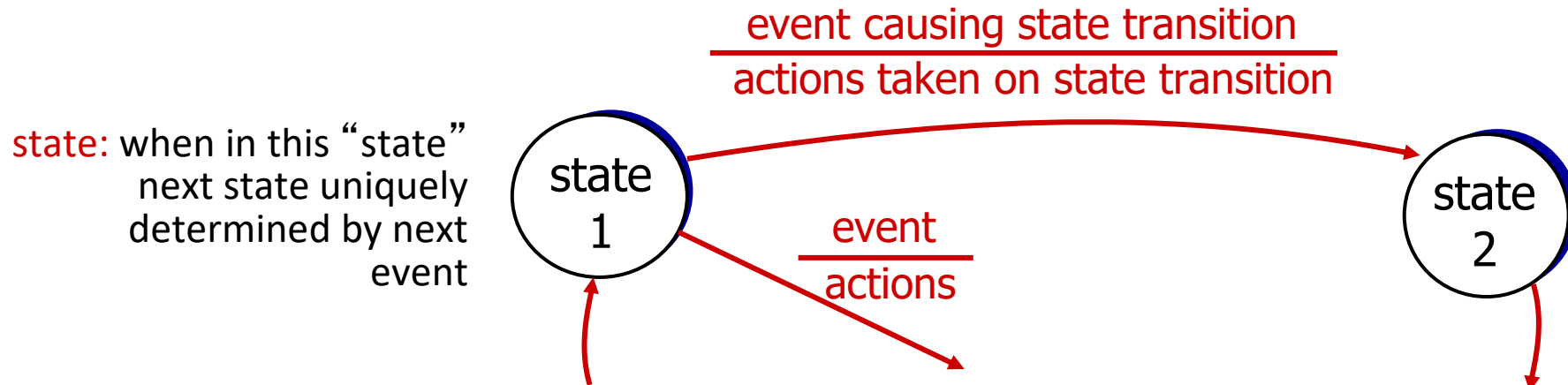
Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

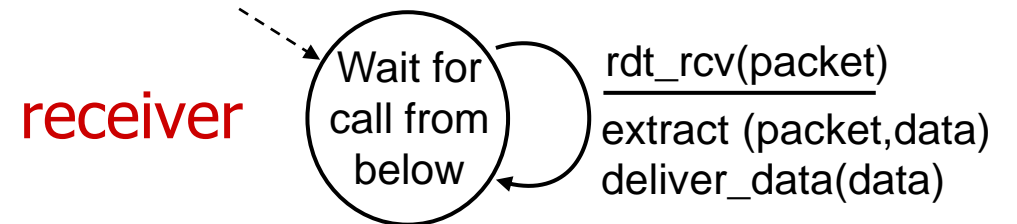
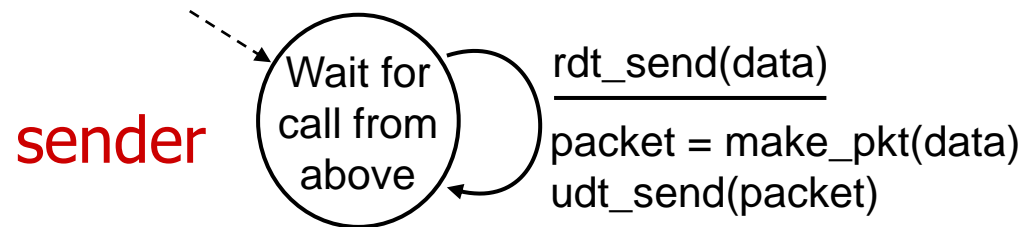
We will:

- 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



rdt2.0: channel with bit errors

- 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?

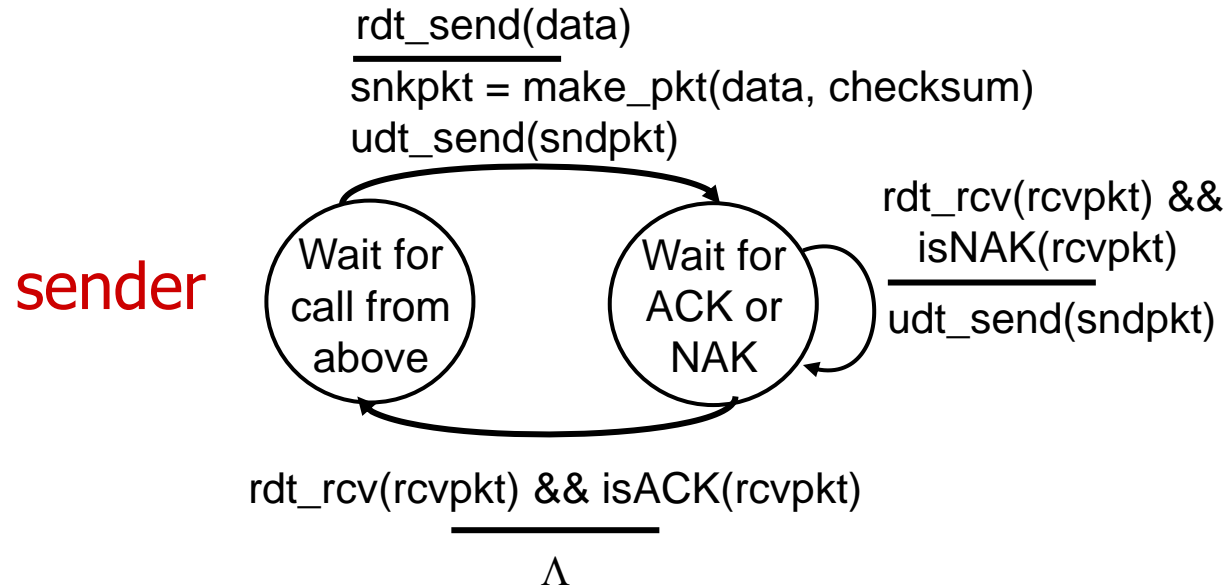
rdt2.0: channel with bit errors

- 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

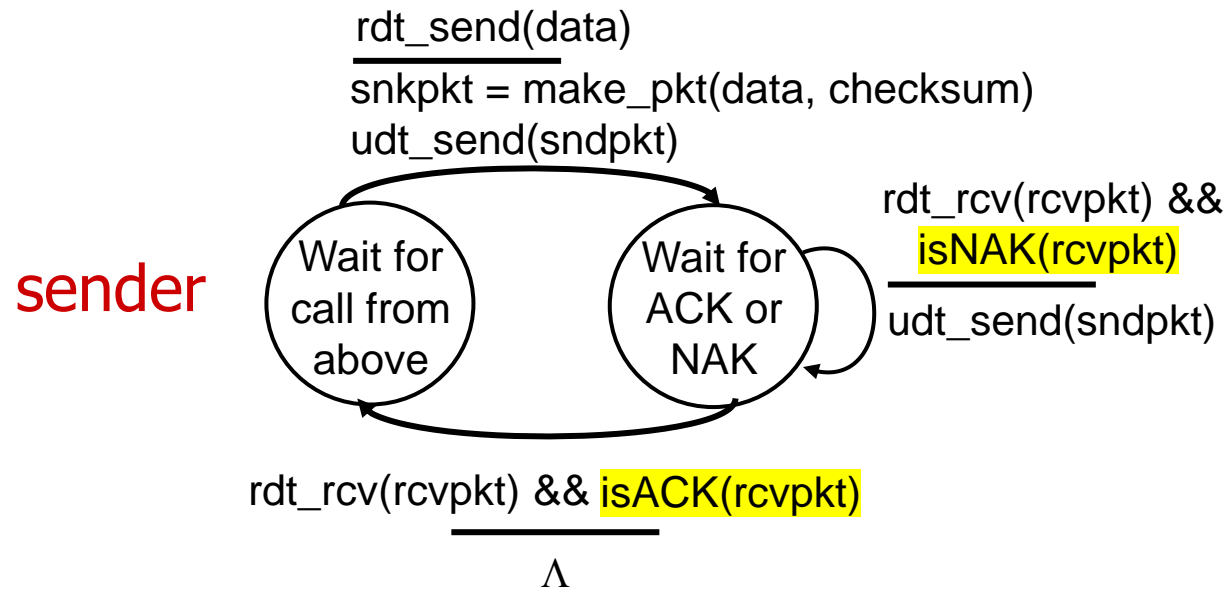
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications



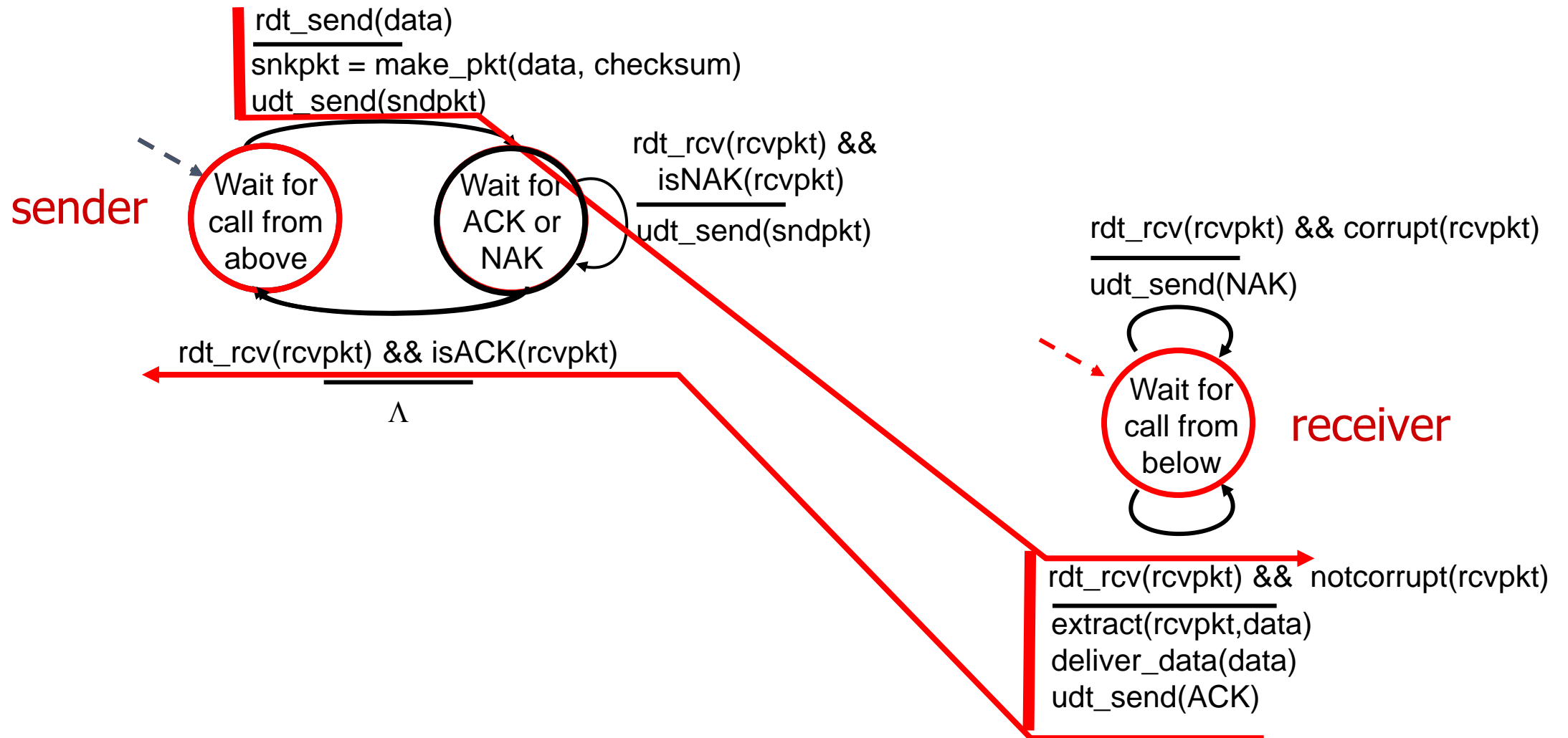
rdt2.0: FSM specification



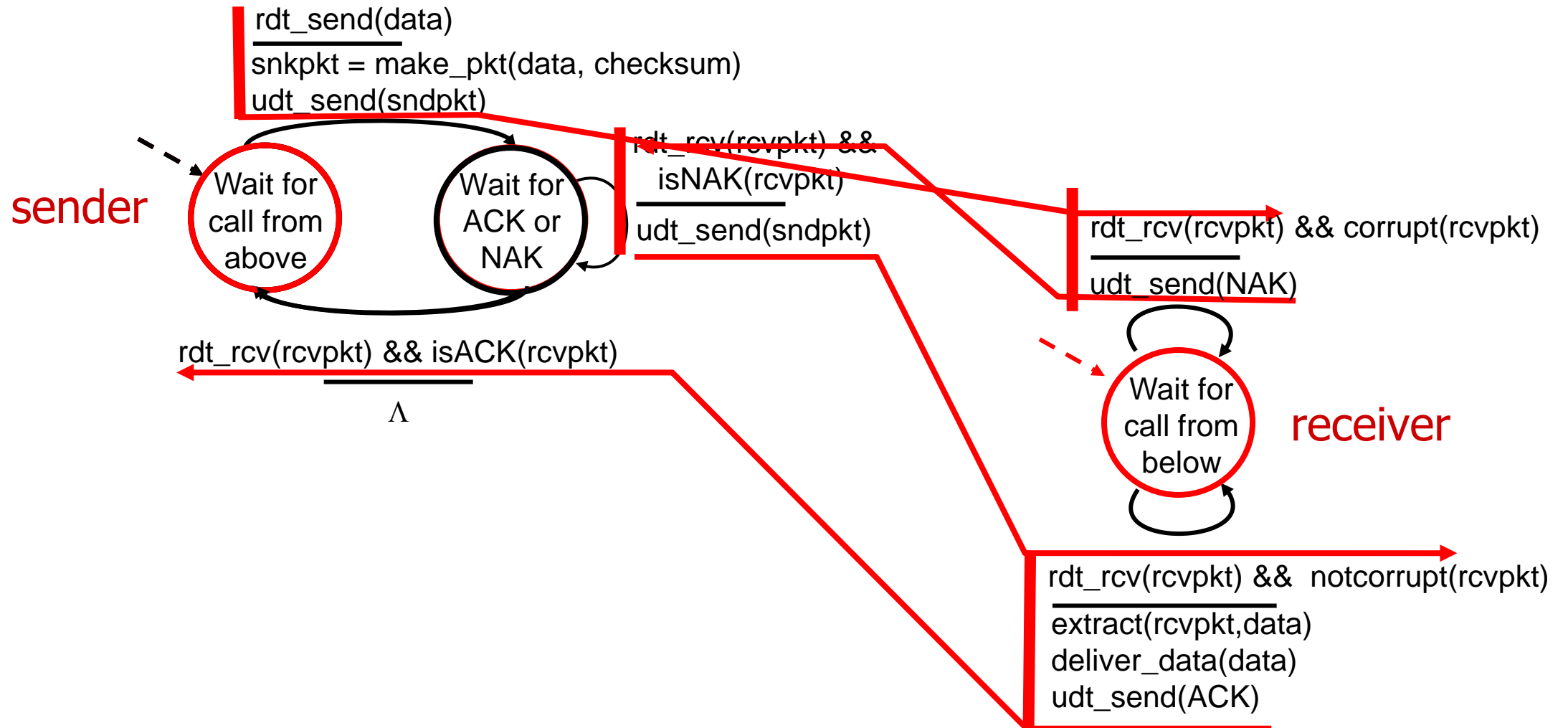
- Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender
- that’s why we need a protocol!



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

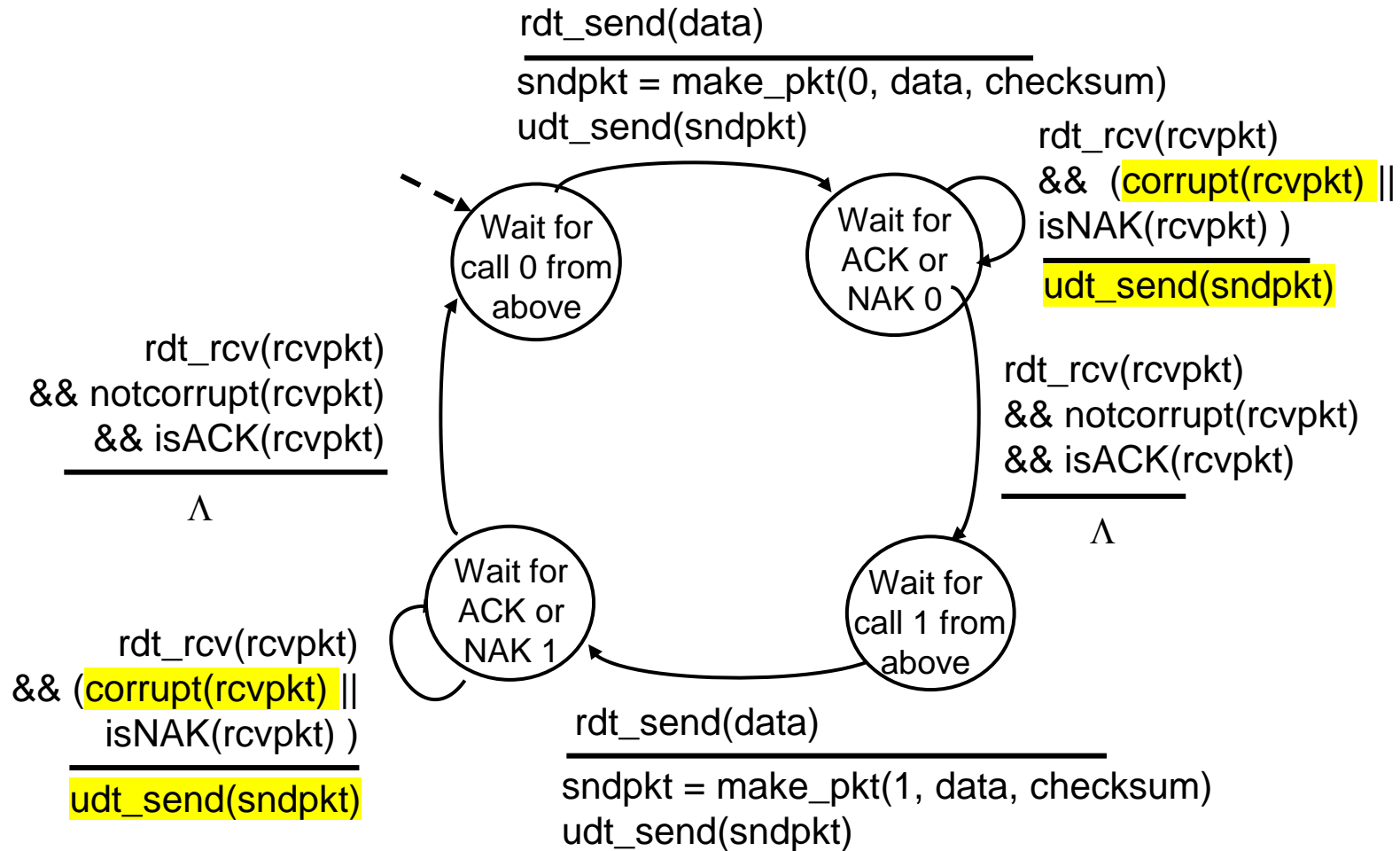
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

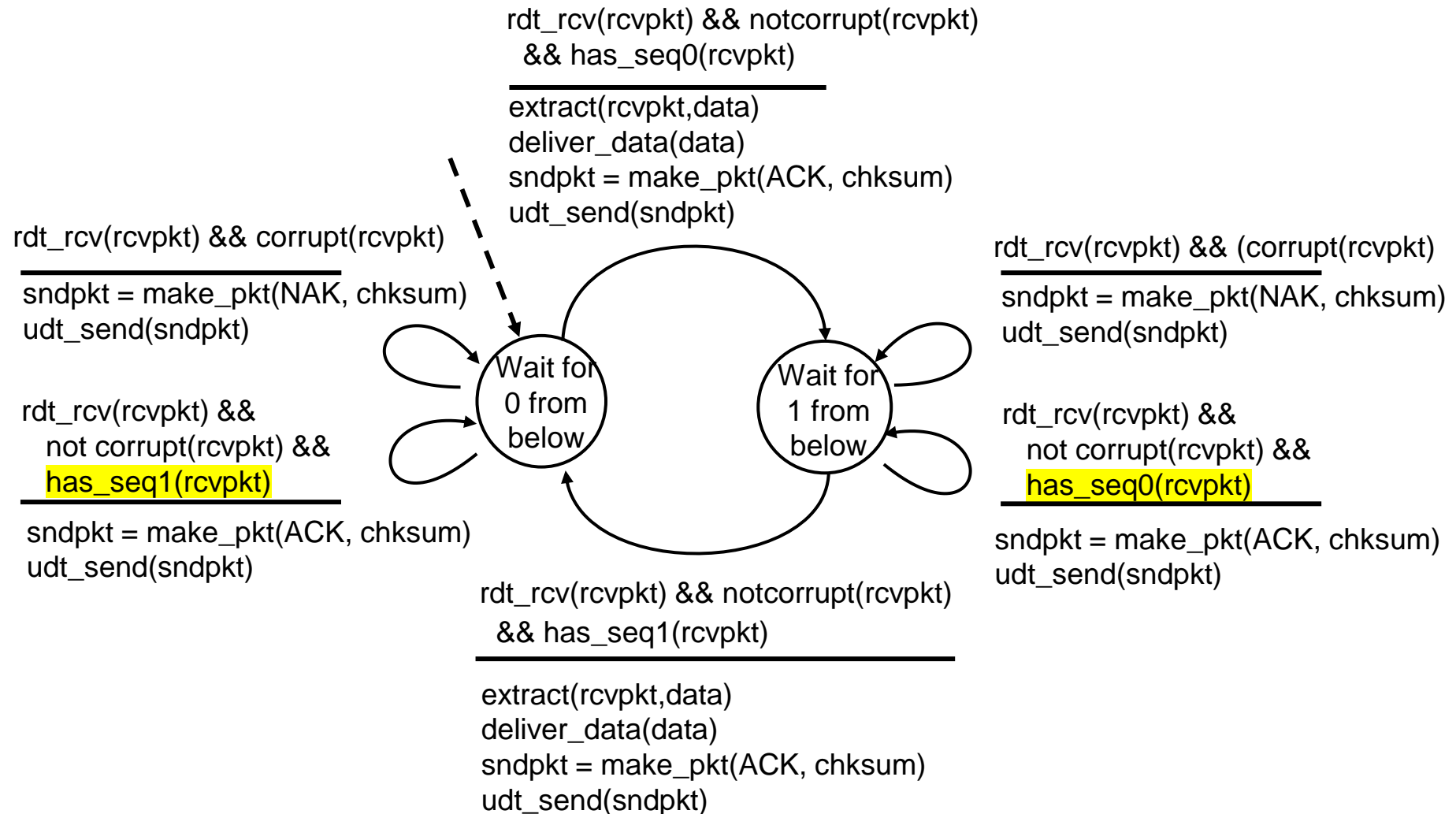
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs



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:

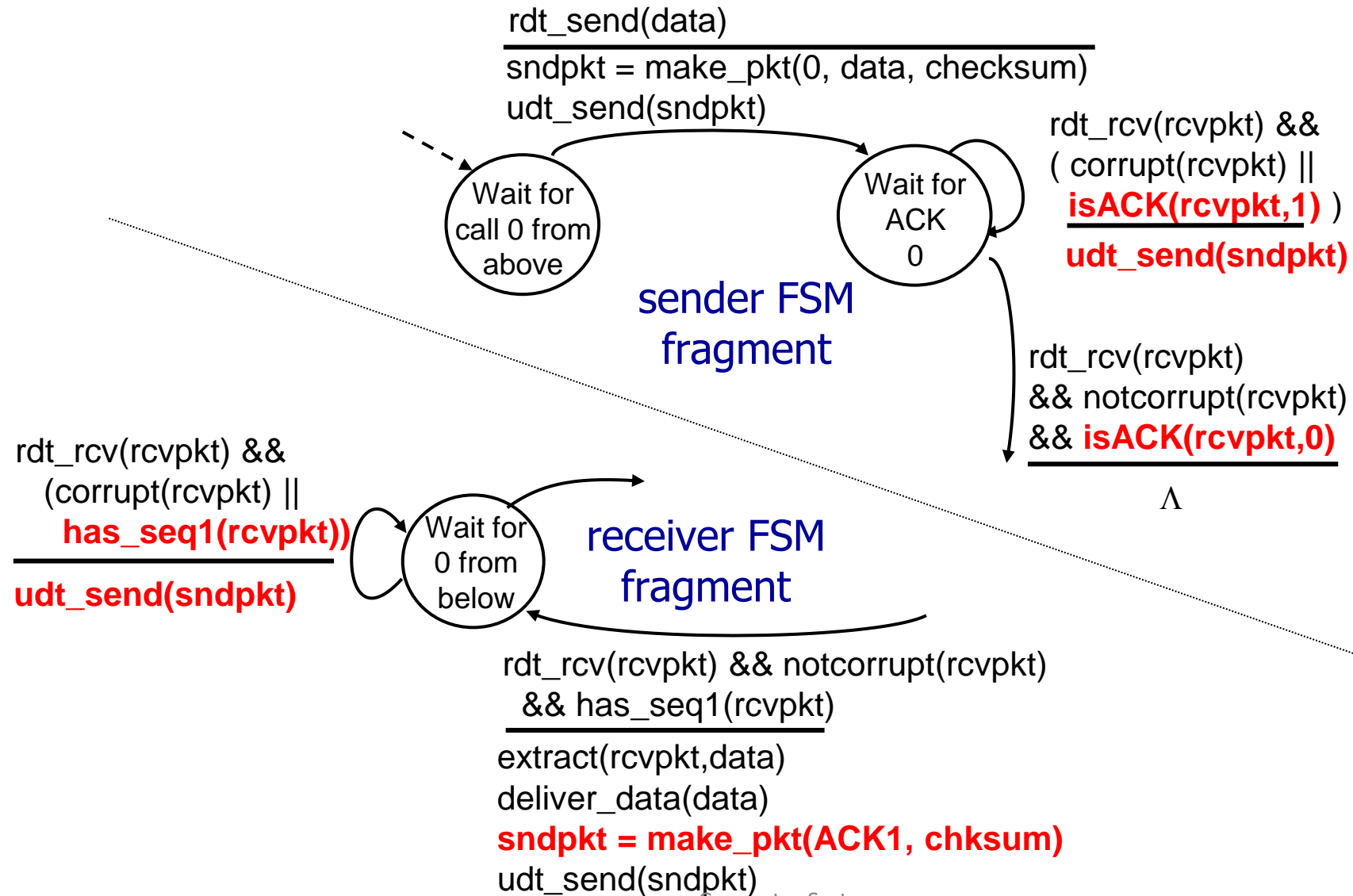
- must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

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



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

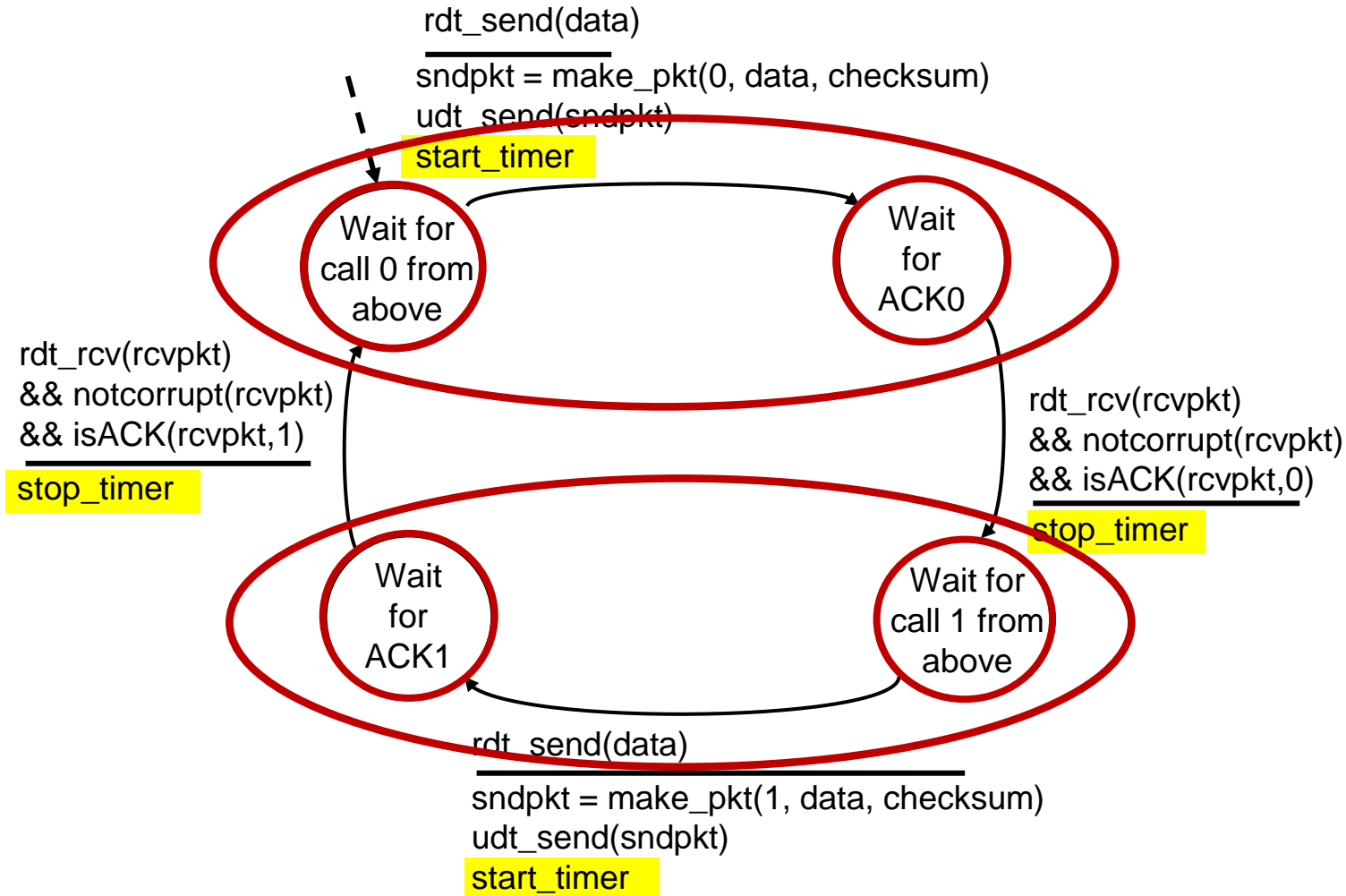
- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

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

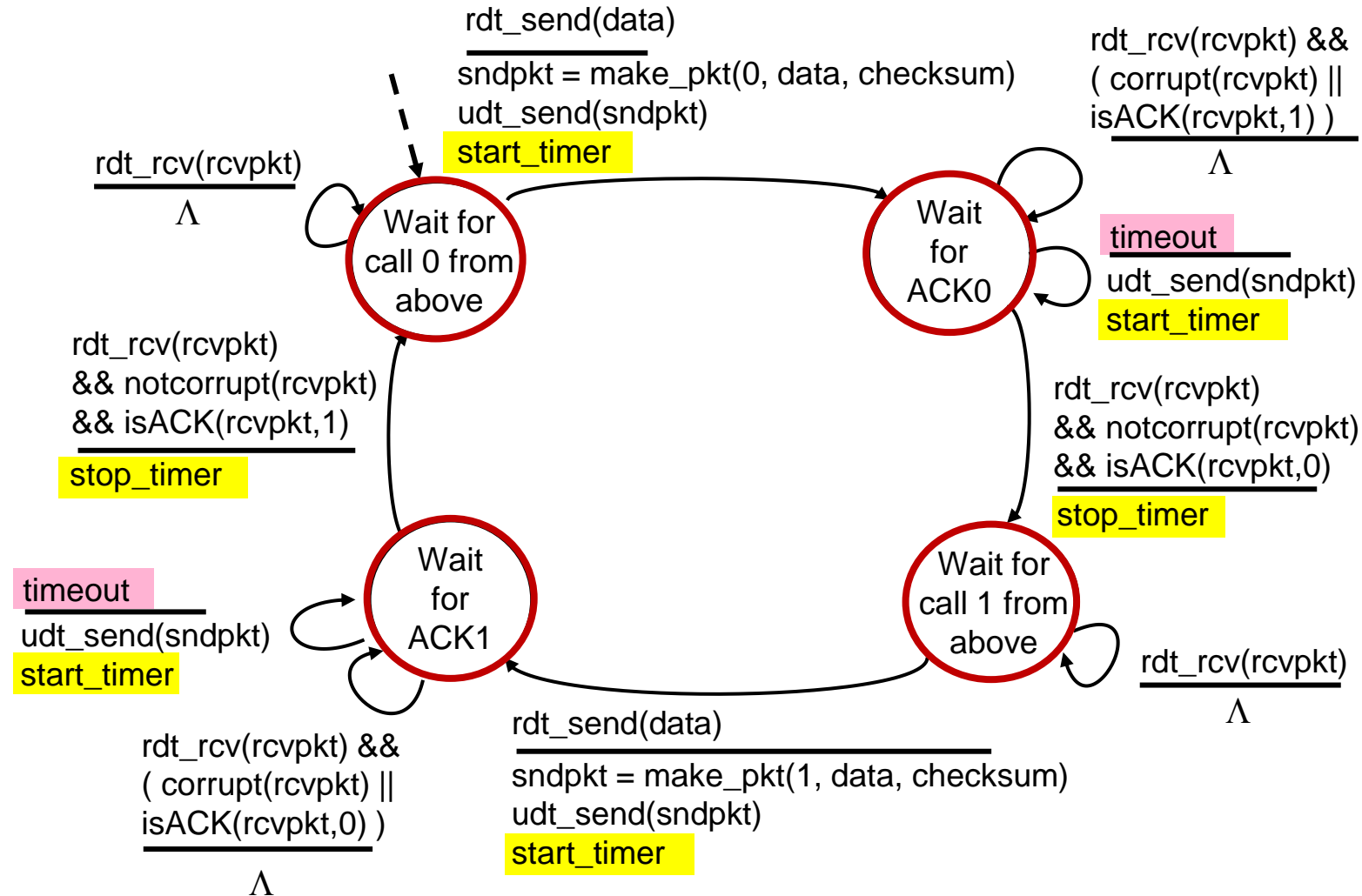
- Approach:* sender waits “reasonable” amount of time for ACK
- retransmits if no ACK received in this time
 - if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but seq #s already handles this!
 - receiver must specify seq # of packet being ACKed
 - use countdown timer to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



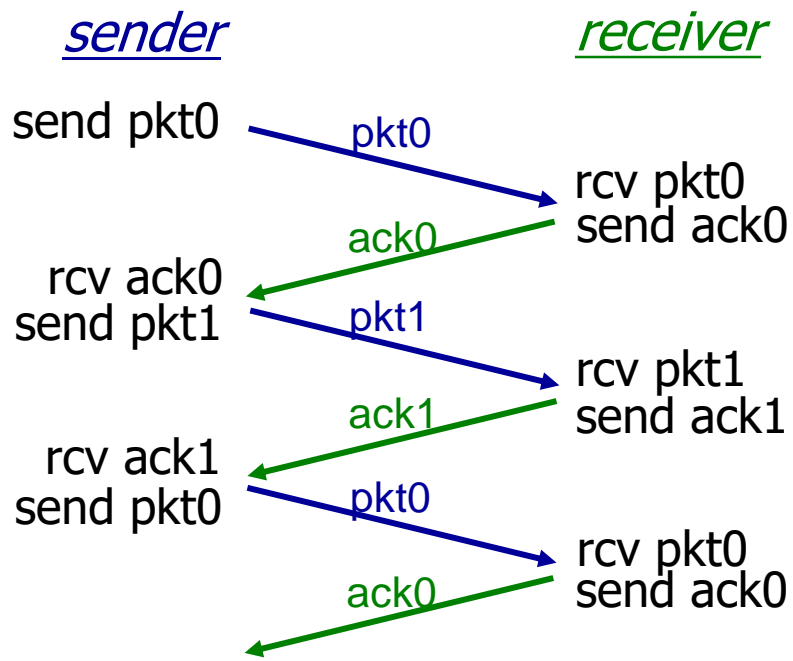
rdt3.0 sender



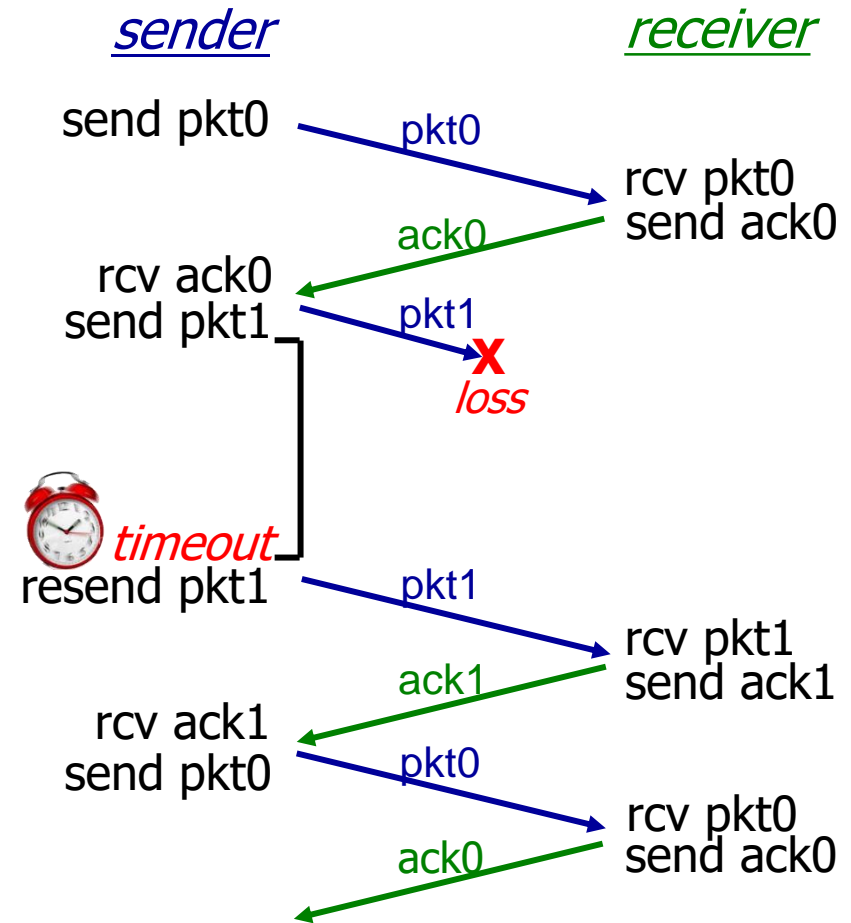
rdt3.0 sender



rdt3.0 in action

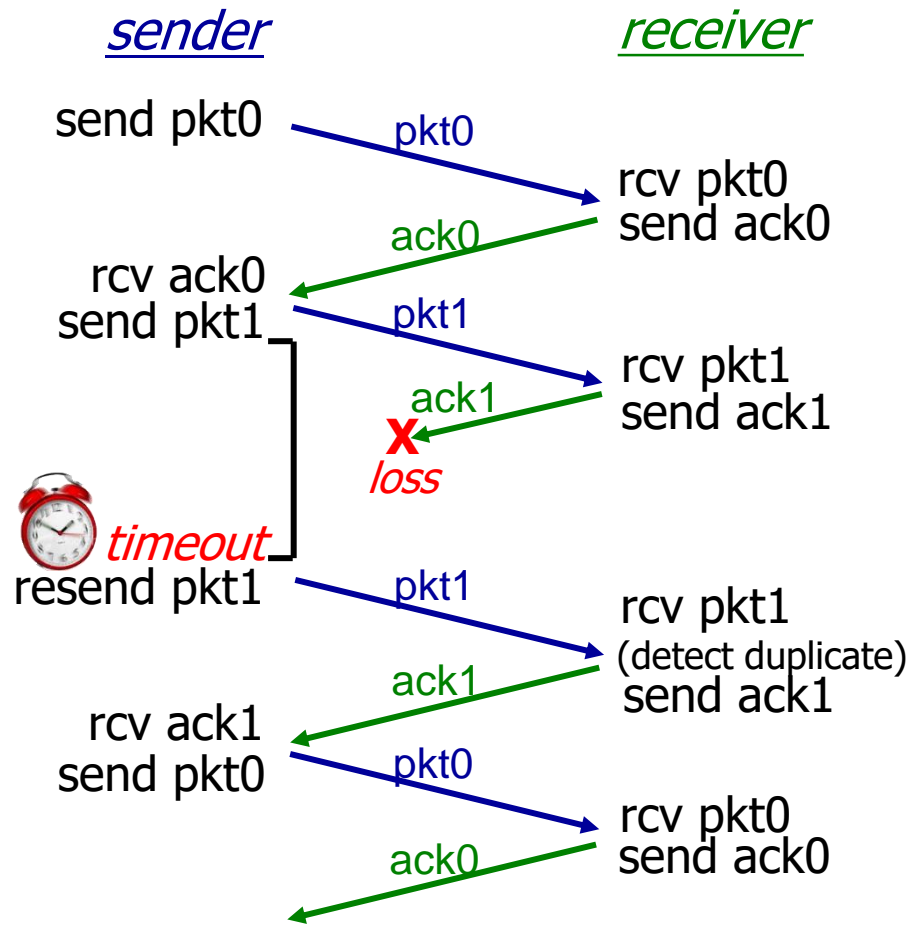


(a) no loss

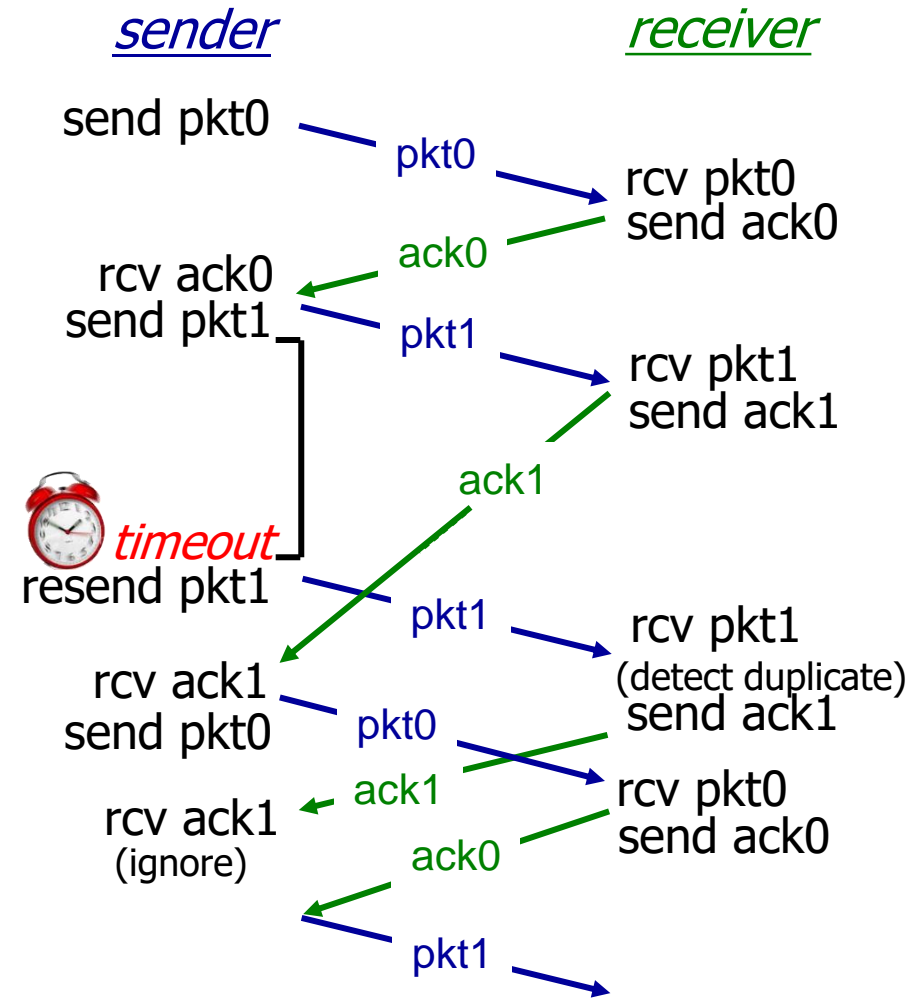


(b) packet loss

rdt3.0 in action



(c) ACK loss



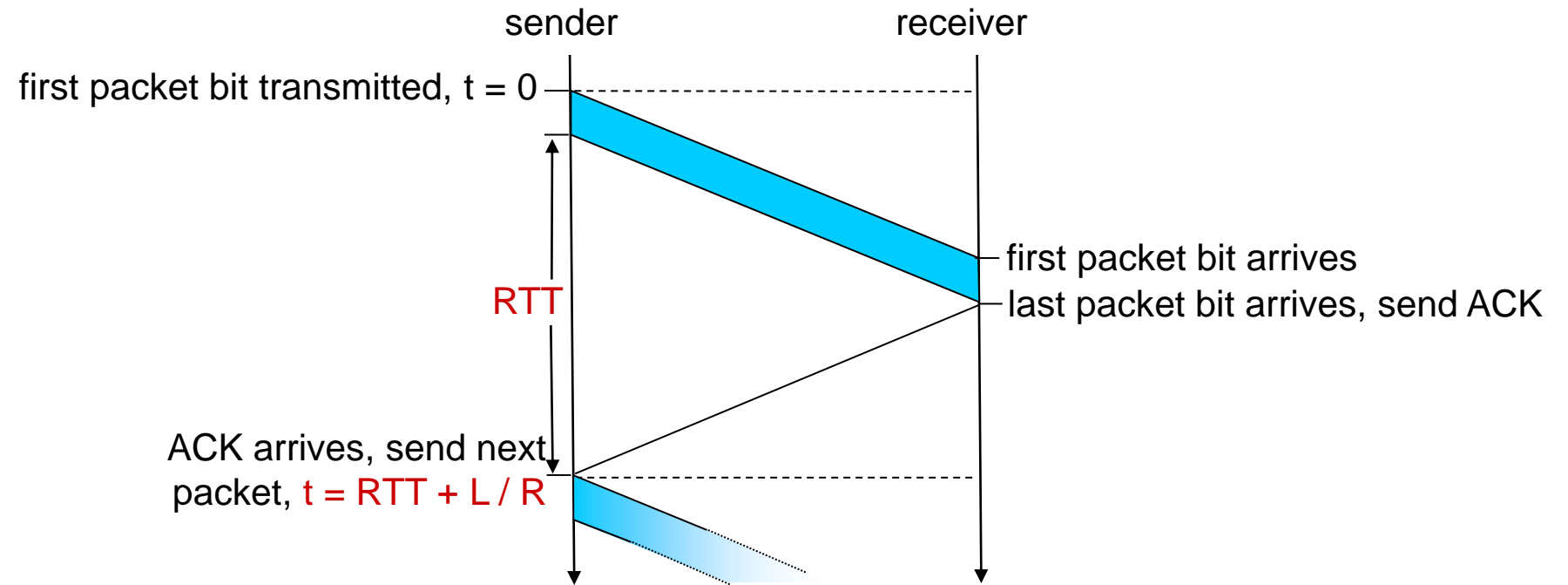
(d) premature timeout/ delayed ACK

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

- U_{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:

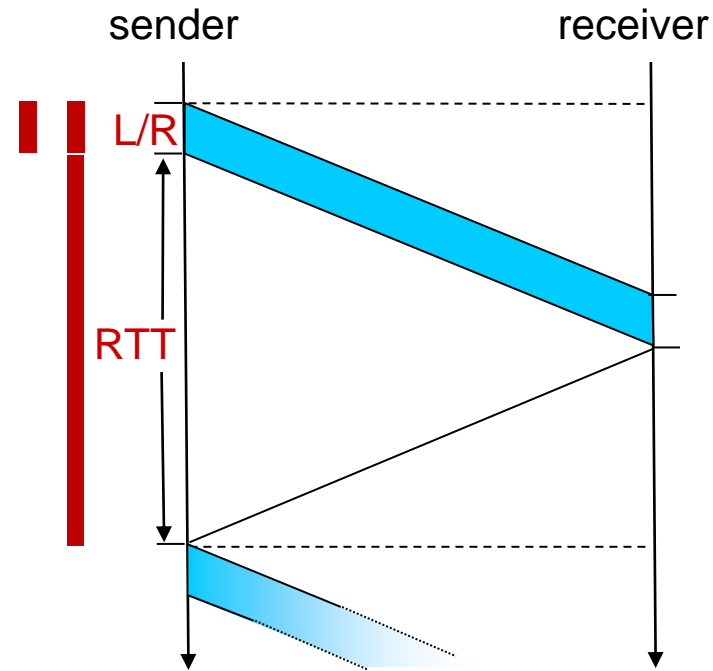
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation



rdt3.0: stop-and-wait operation

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

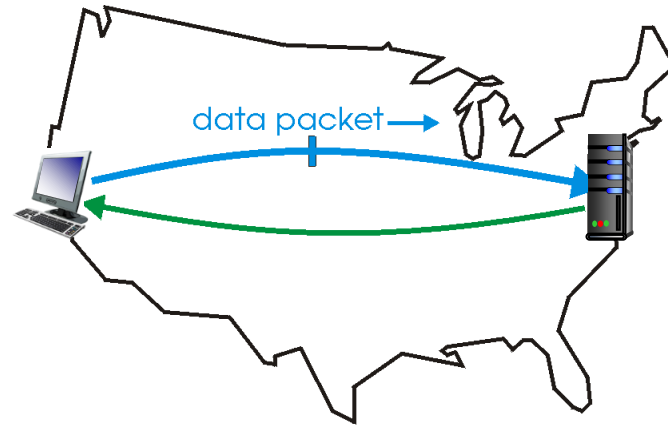


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

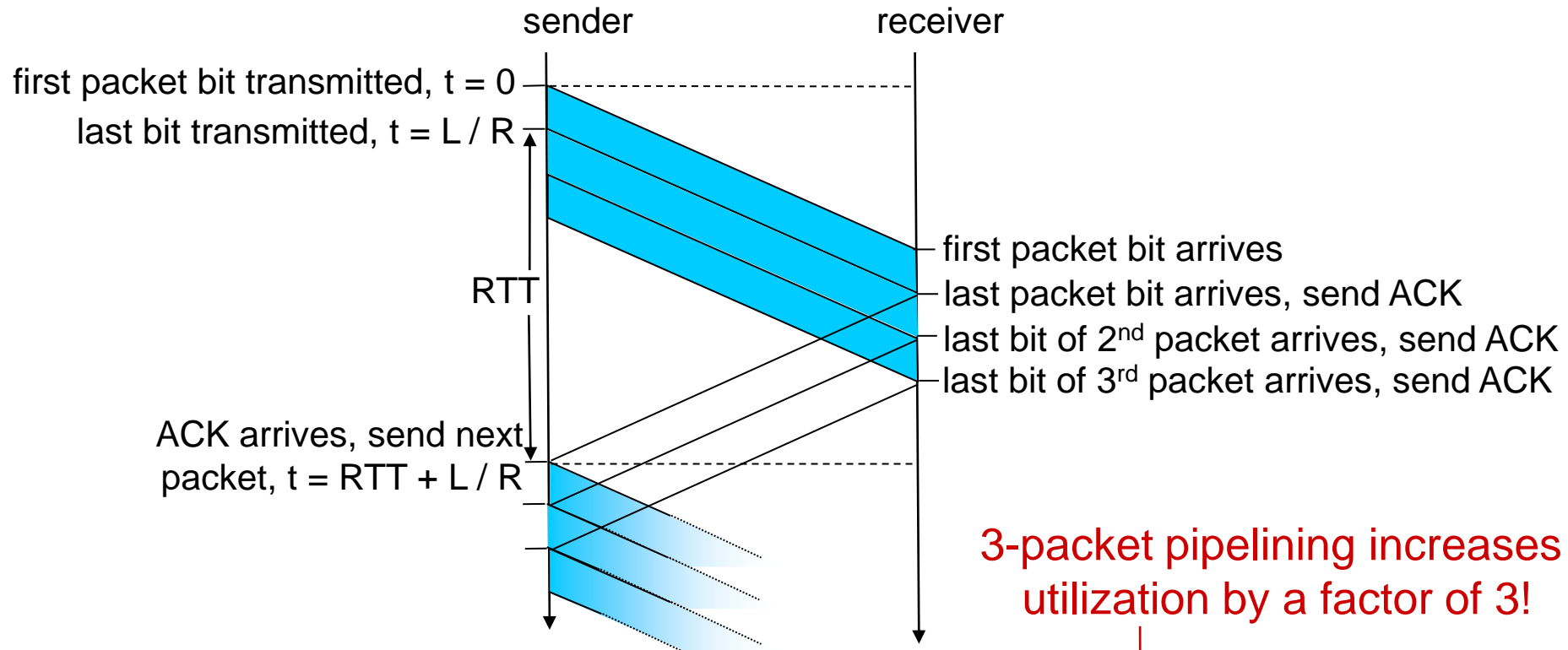
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

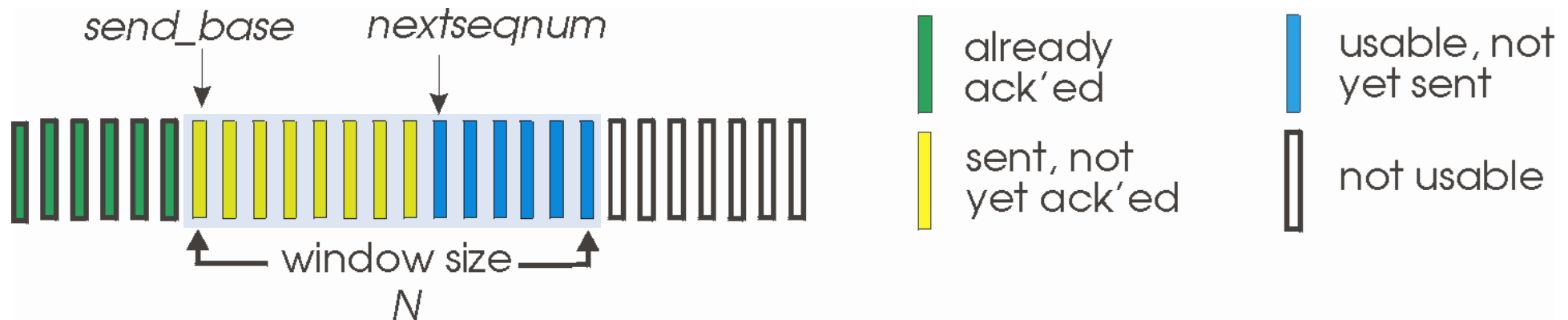
Pipelining: increased utilization



$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: sender

- 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 $n+1$
- timer for oldest in-flight packet
- ***timeout(n)***: retransmit packet n and all higher seq # packets in window

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

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

ignore duplicate ACK



pkt 2 timeout

send pkt2
 send pkt3
 send pkt4
 send pkt5

receiver

receive pkt0, send ack0
 receive pkt1, send ack1

receive pkt3, discard,
 (re)send ack1

receive pkt4, discard,
 (re)send ack1

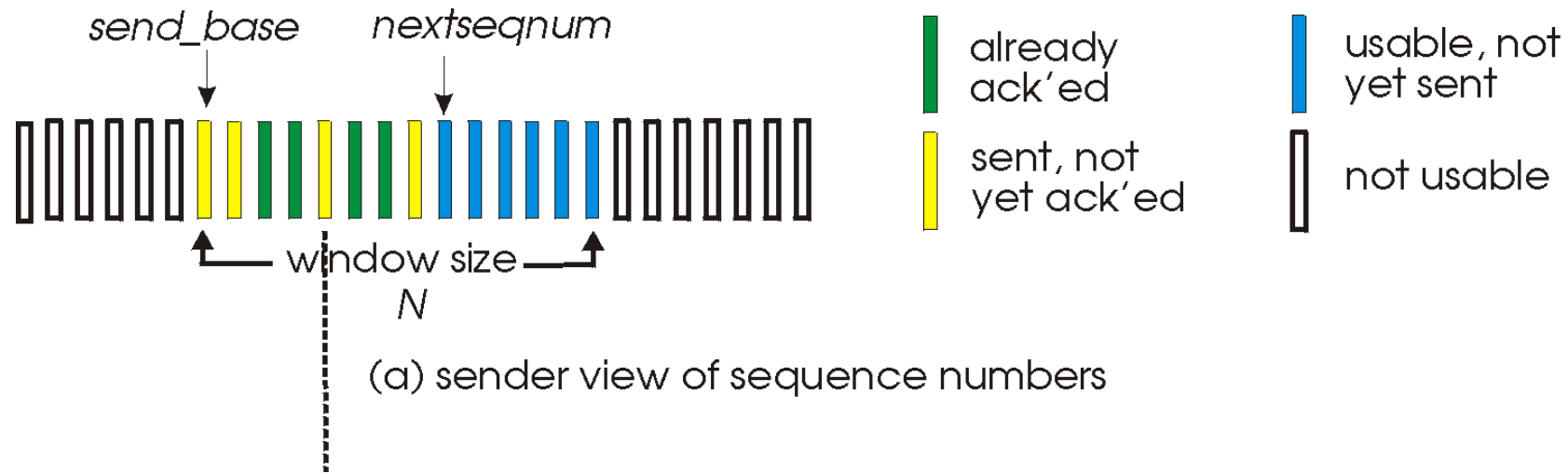
receive pkt5, discard,
 (re)send ack1

rcv pkt2, deliver, send ack2
 rcv pkt3, deliver, send ack3
 rcv pkt4, deliver, send ack4
 rcv pkt5, deliver, send ack5

Selective repeat: the approach

- *pipelining*: multiple packets in flight
- *receiver individually ACKs* all correctly received packets
 - buffers packets, as needed, for in-order delivery to upper layer
- sender:
 - maintains (conceptually) a timer for each unACKed pkt
 - timeout: retransmits single unACKed packet associated with timeout
 - maintains (conceptually) “window” over N consecutive seq #s
 - limits pipelined, “in flight” packets to be within this window

Selective repeat: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(n):

- resend packet n , restart timer

ACK(n) in $[\text{sendbase}, \text{sendbase}+N-1]$:

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

receiver

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

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

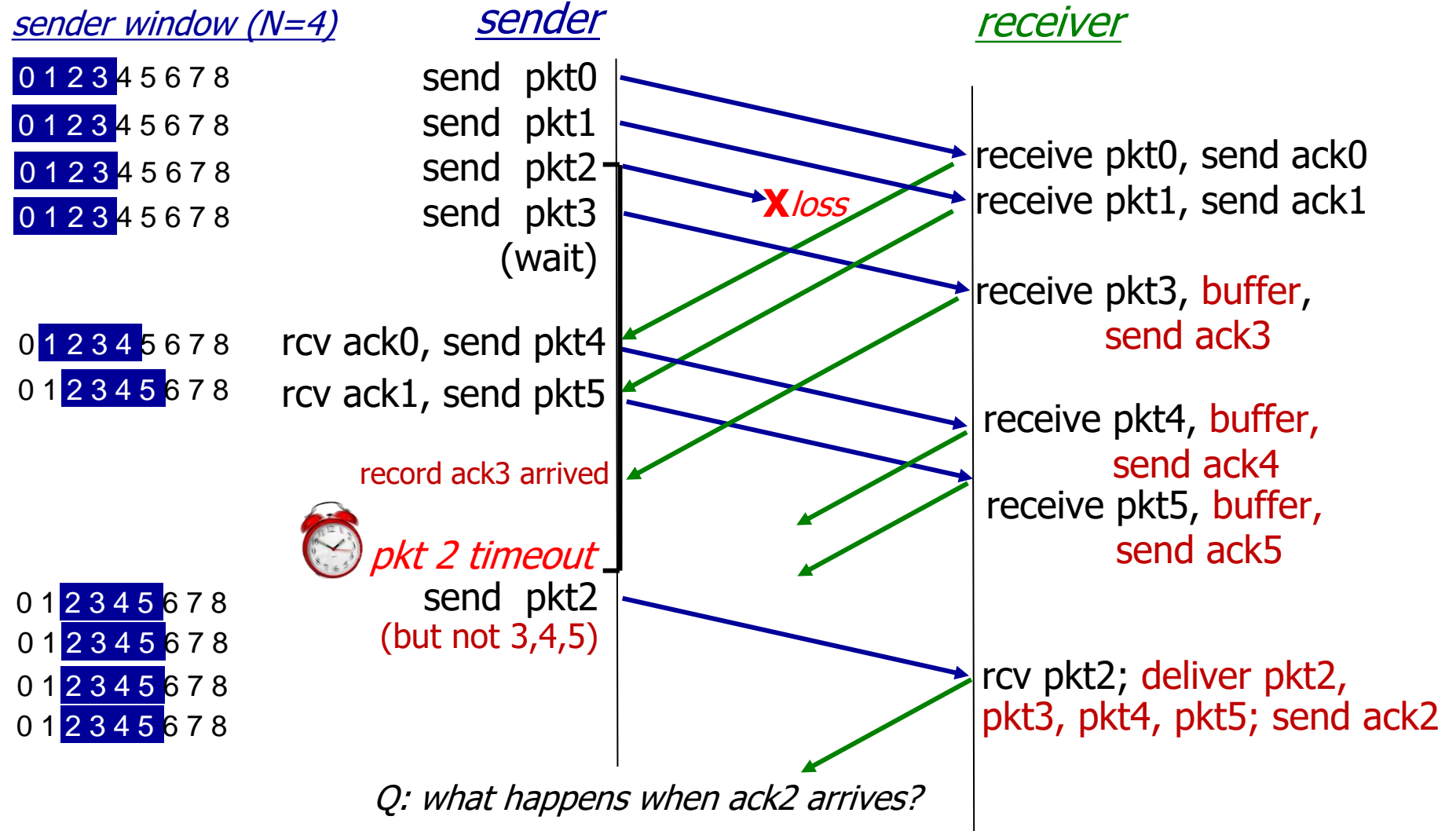
packet n in $[\text{rcvbase}-N, \text{rcvbase}-1]$

- ACK(n)

otherwise:

- ignore

Selective Repeat in action



Transport Layer: Roadmap

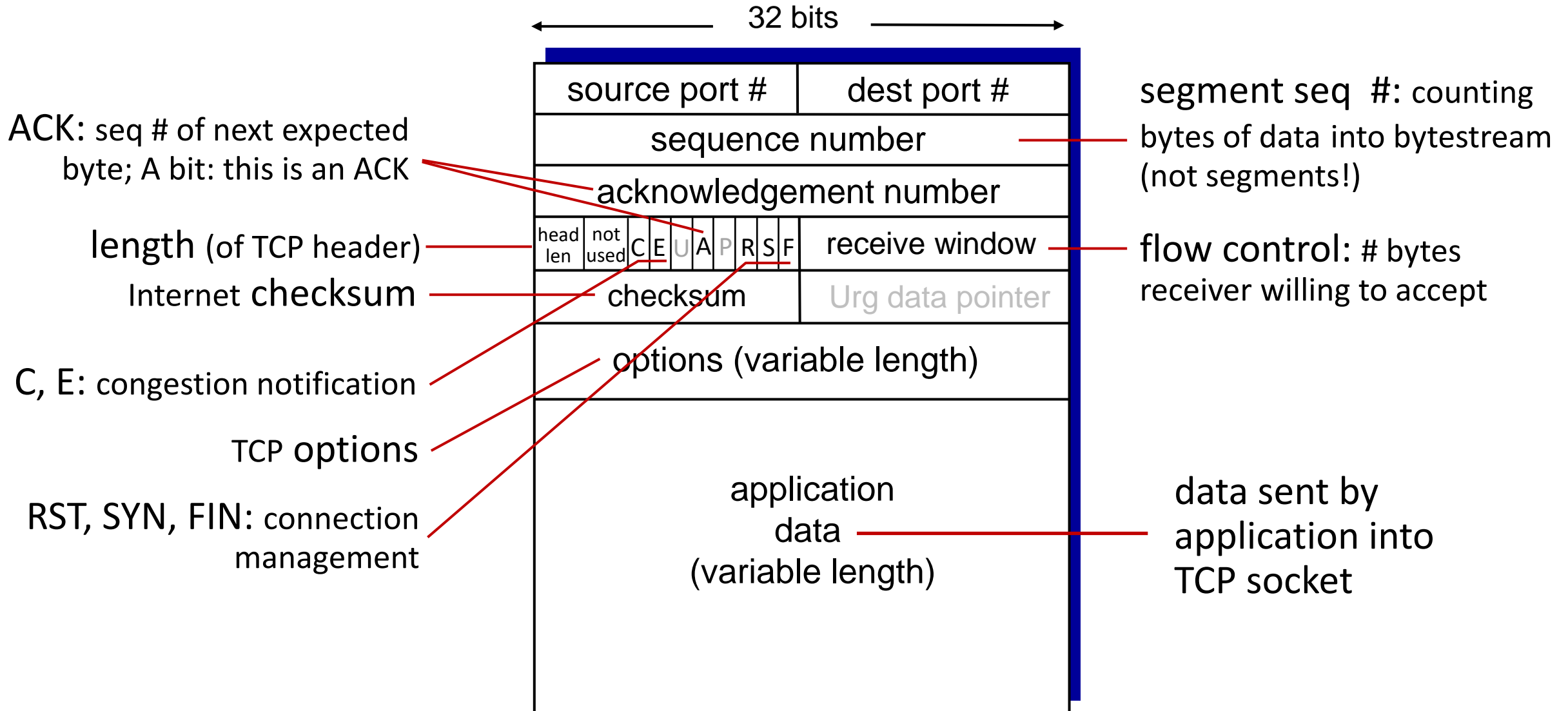
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- Principles of congestion control
- TCP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



TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- **point-to-point:**
 - one sender, one receiver
- **reliable, in-order *byte stream*:**
 - no “message boundaries”
- **full duplex 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



TCP sequence numbers, ACKs

Sequence numbers:

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

Acknowledgements:

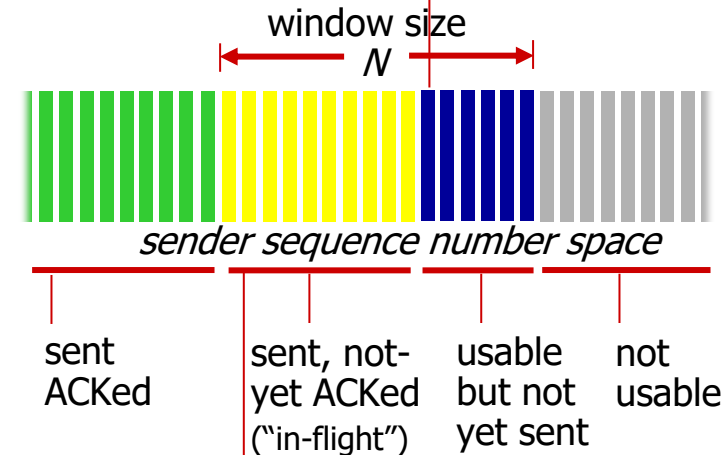
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

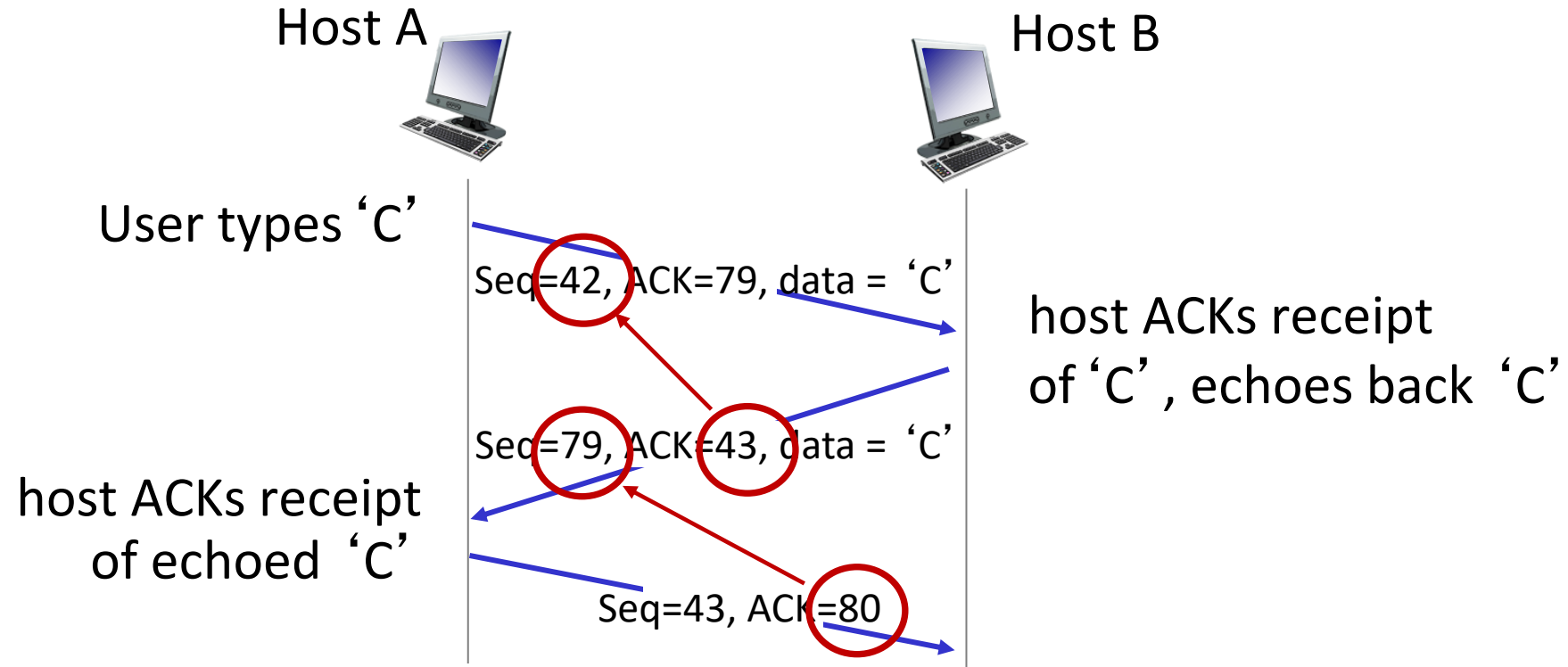
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
A	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

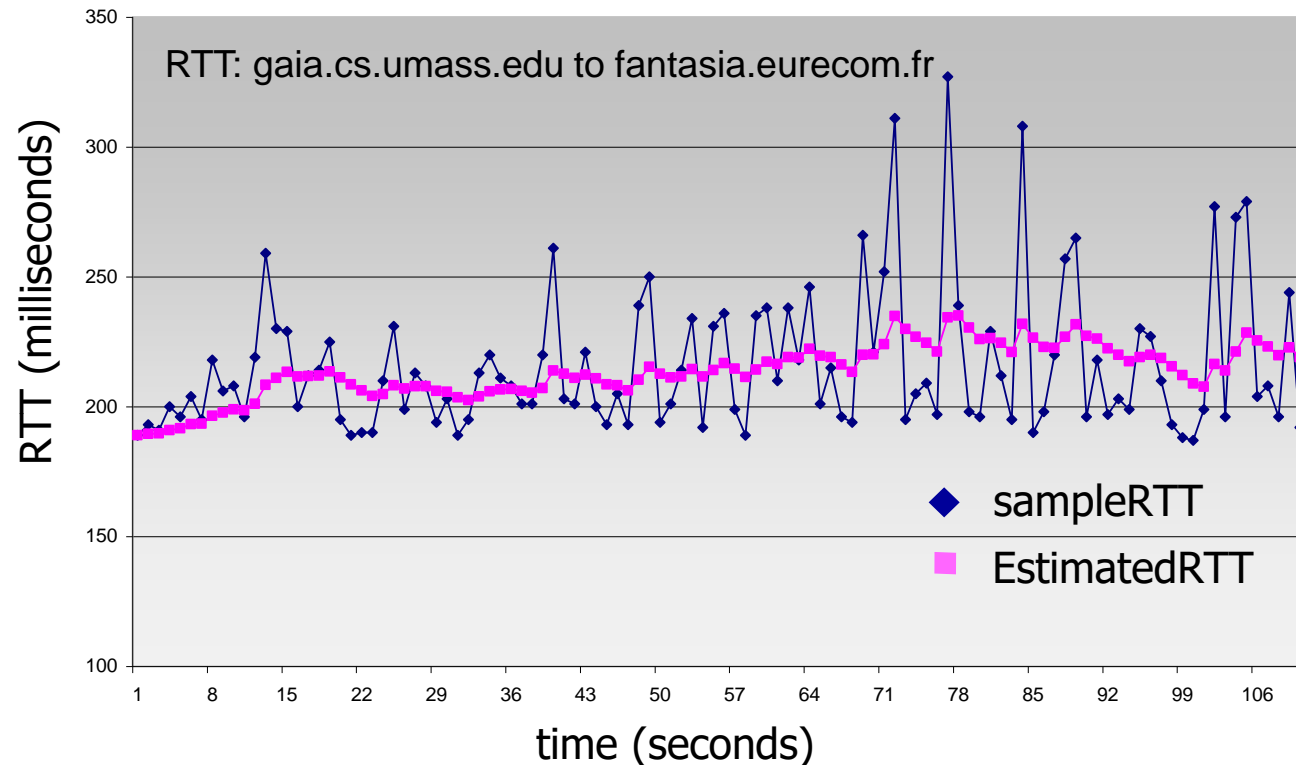
Q: how to estimate RTT?

- `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

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑
estimated RTT

↑
“safety margin”

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

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

(typically, $\beta = 0.25$)

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**

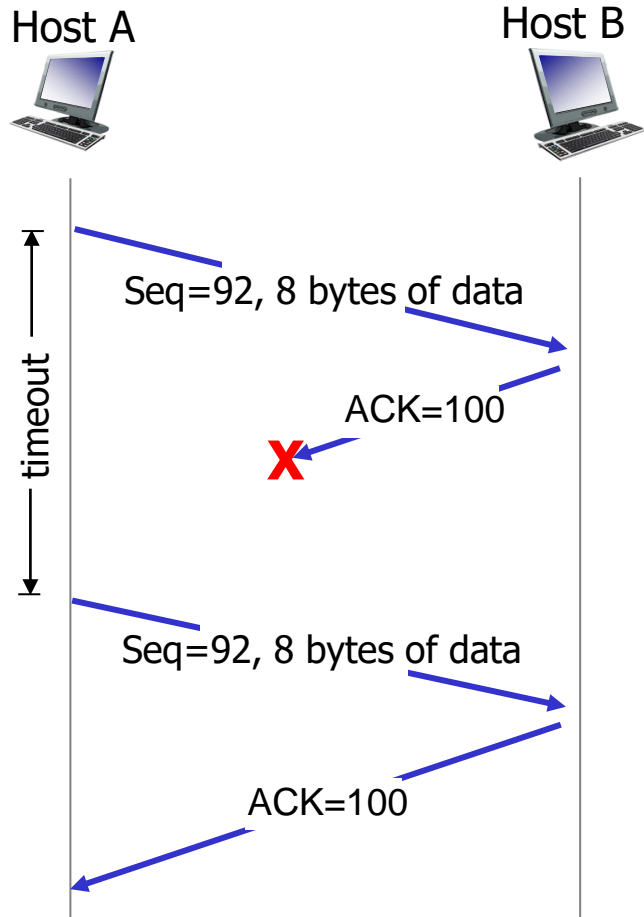
event: timeout

- retransmit segment that caused timeout
- restart timer

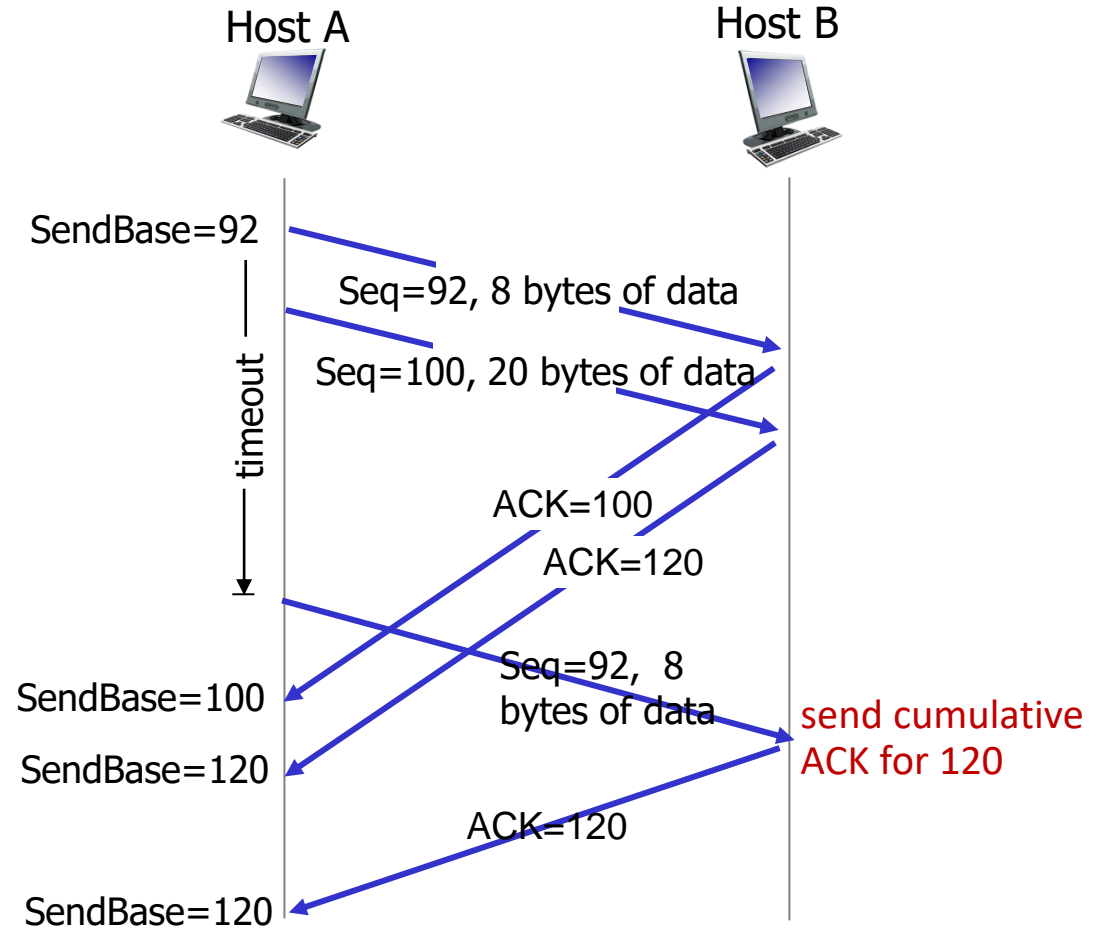
event: ACK received

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

TCP: retransmission scenarios

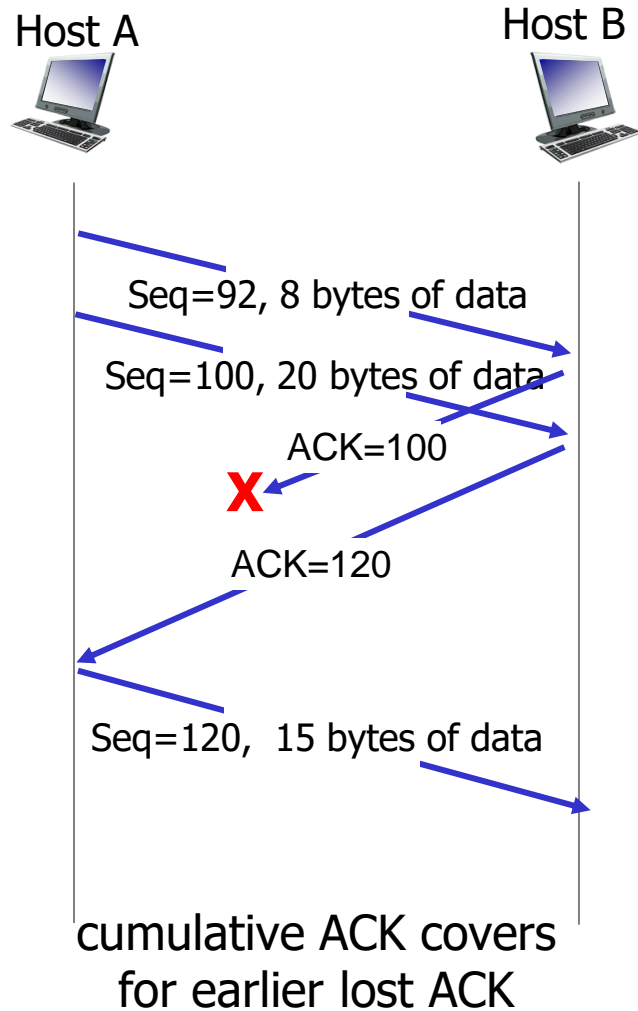


lost ACK scenario



premature timeout

TCP: retransmission scenarios



TCP fast retransmit

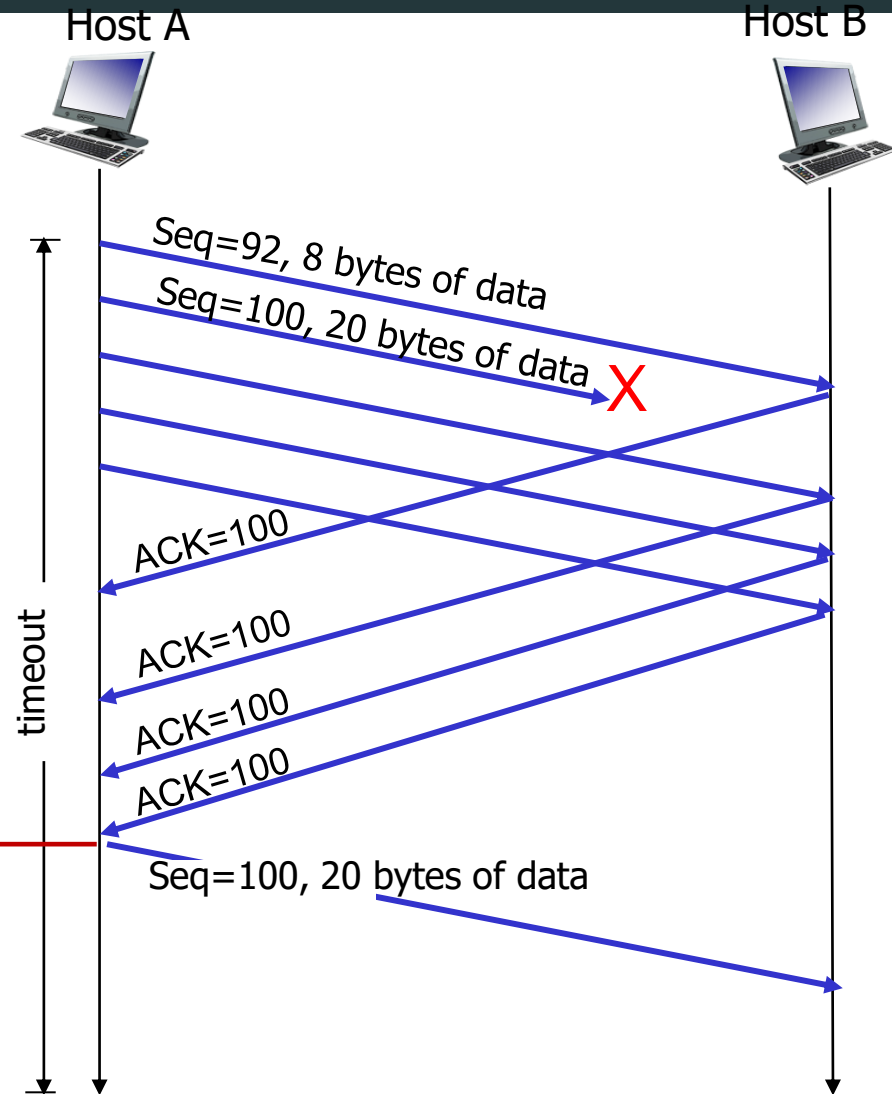
TCP fast retransmit

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!



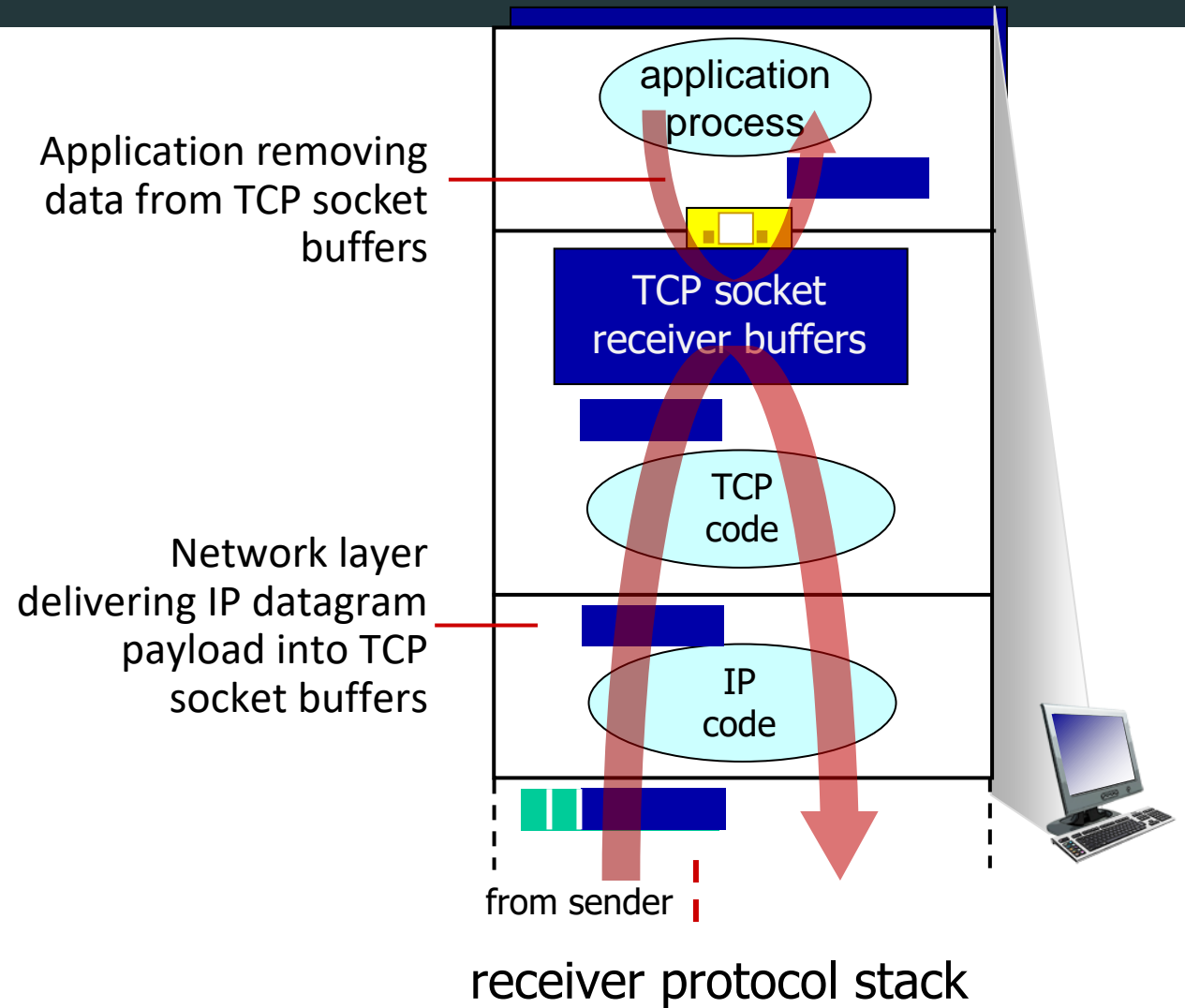
Transport Layer: Roadmap

- 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



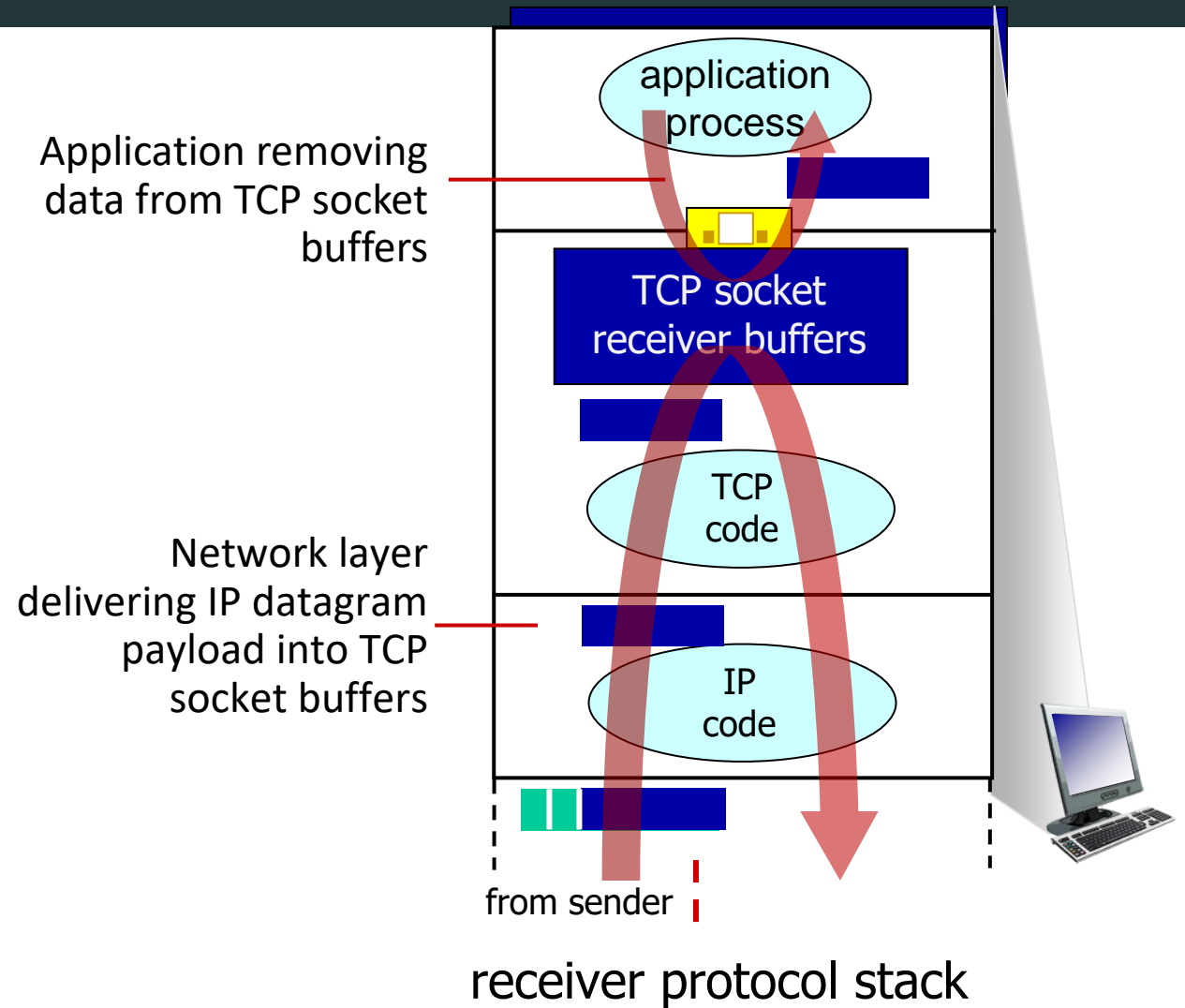
TCP flow control

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



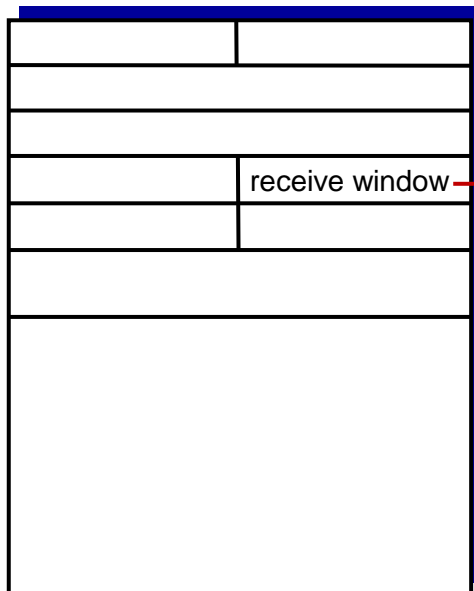
TCP flow control

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



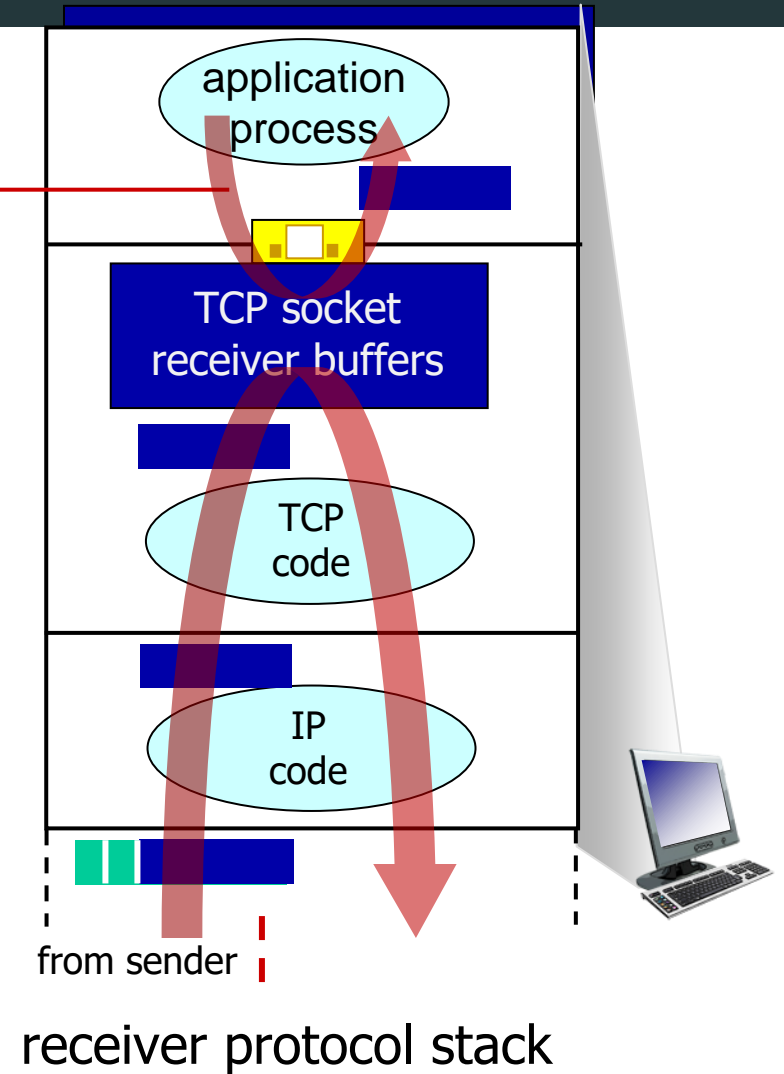
TCP flow control

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



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



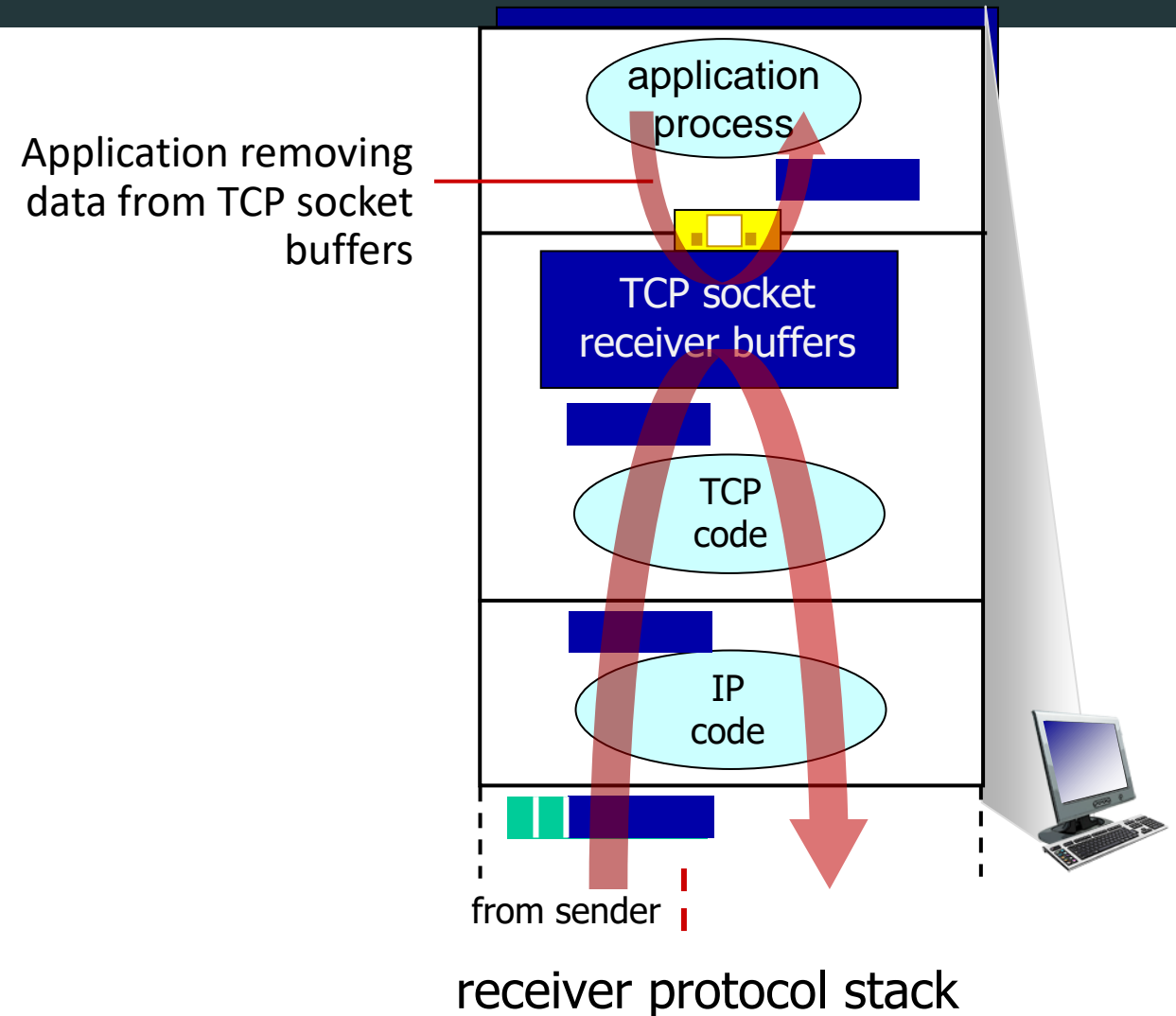
receiver protocol stack

TCP flow control

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

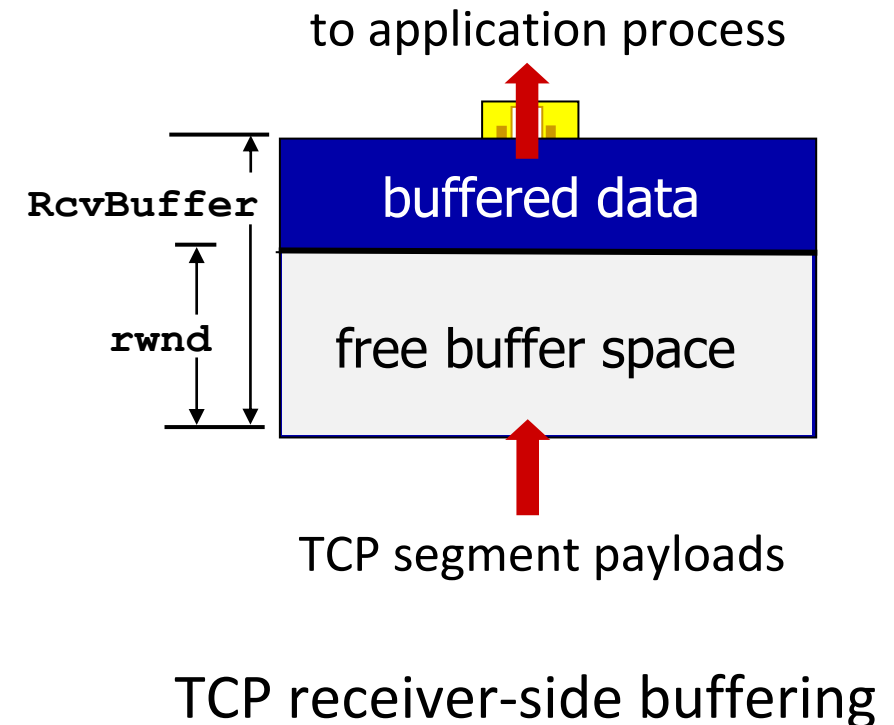
flow control

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



TCP flow control

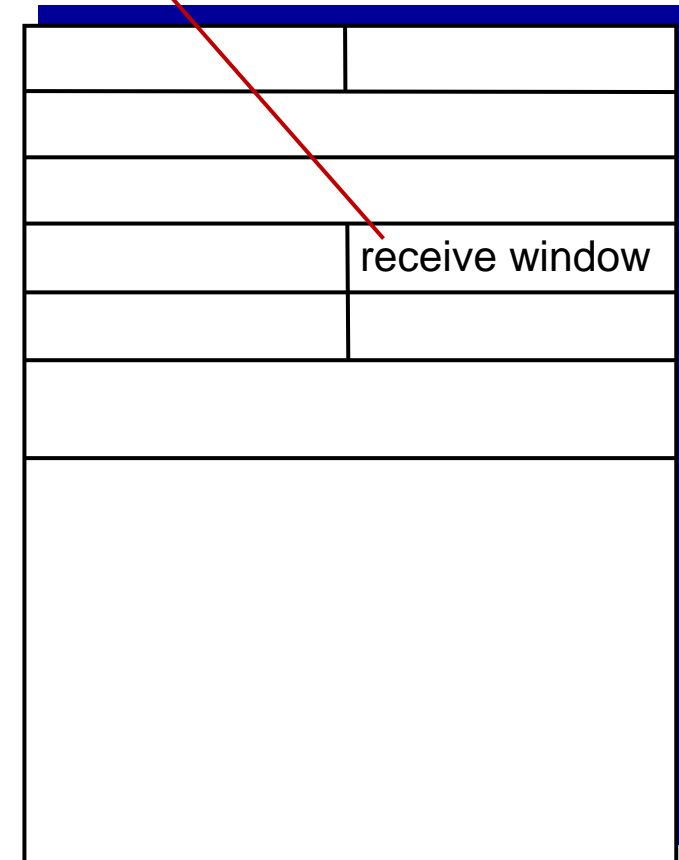
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
 - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
 - many operating systems auto-adjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive 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 auto-adjust **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

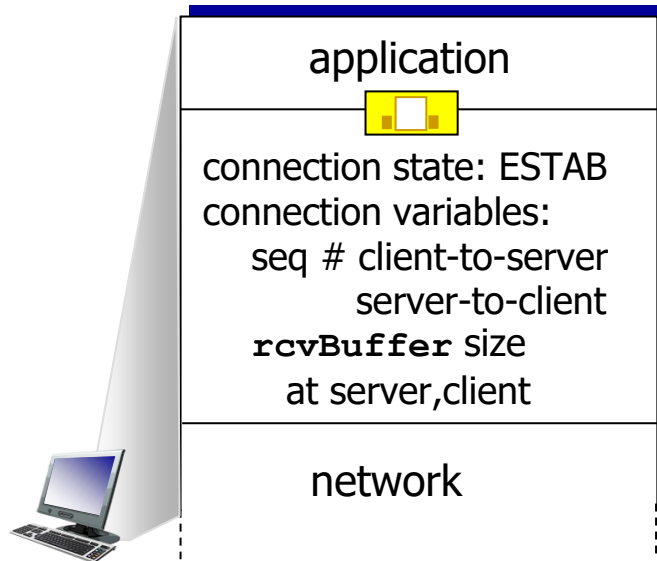


TCP segment format

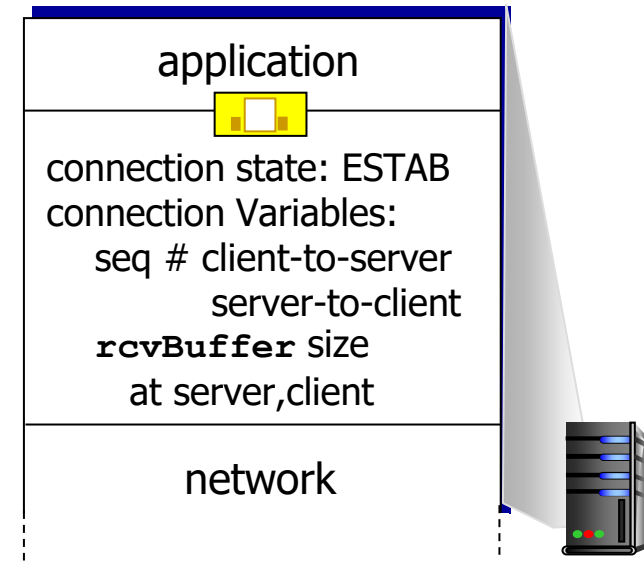
TCP connection management

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)



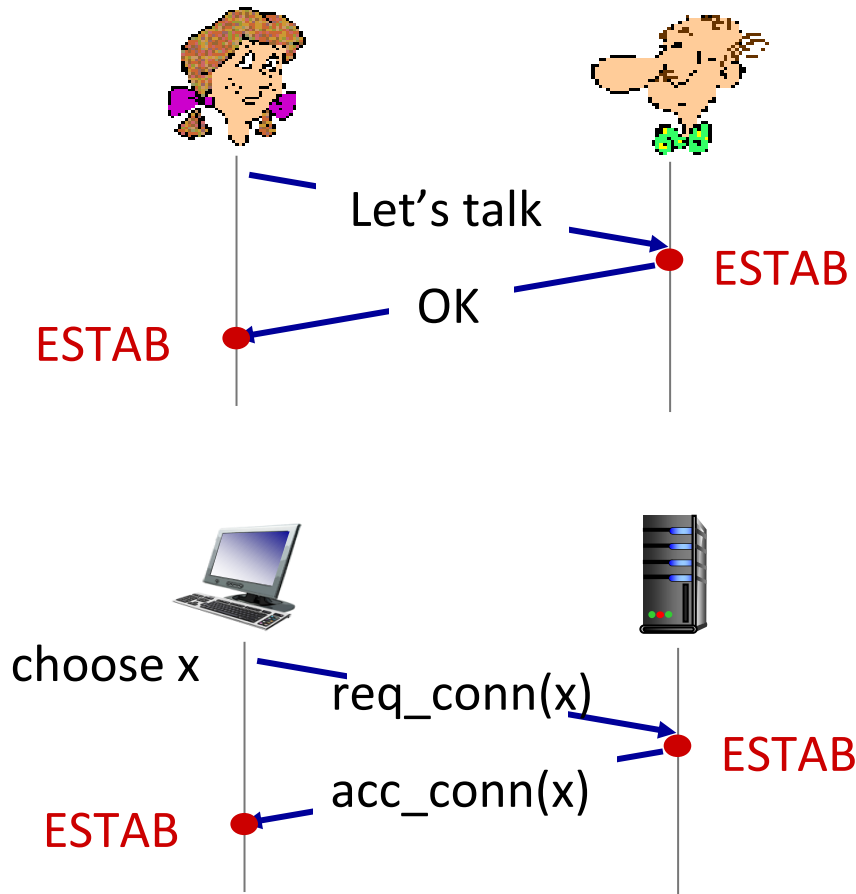
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

Agreeing to establish a connection

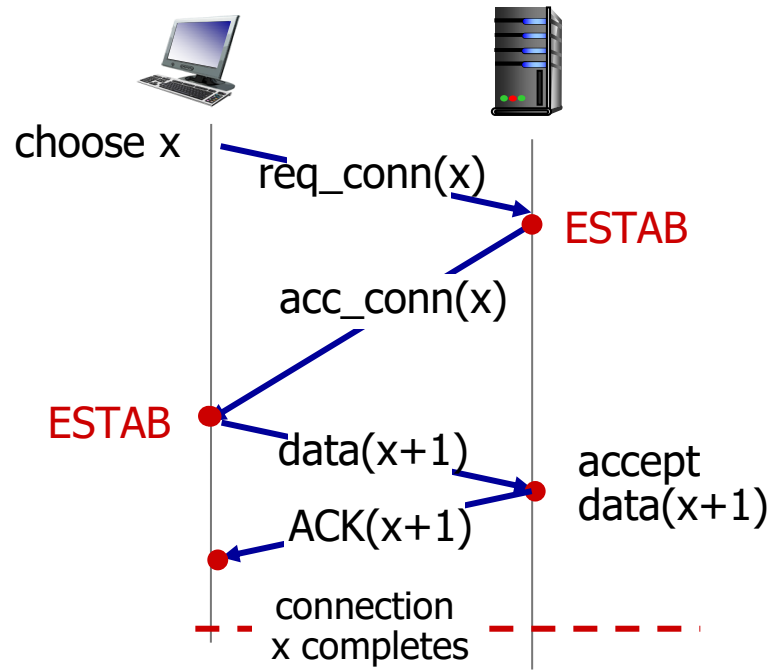
2-way handshake:



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

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

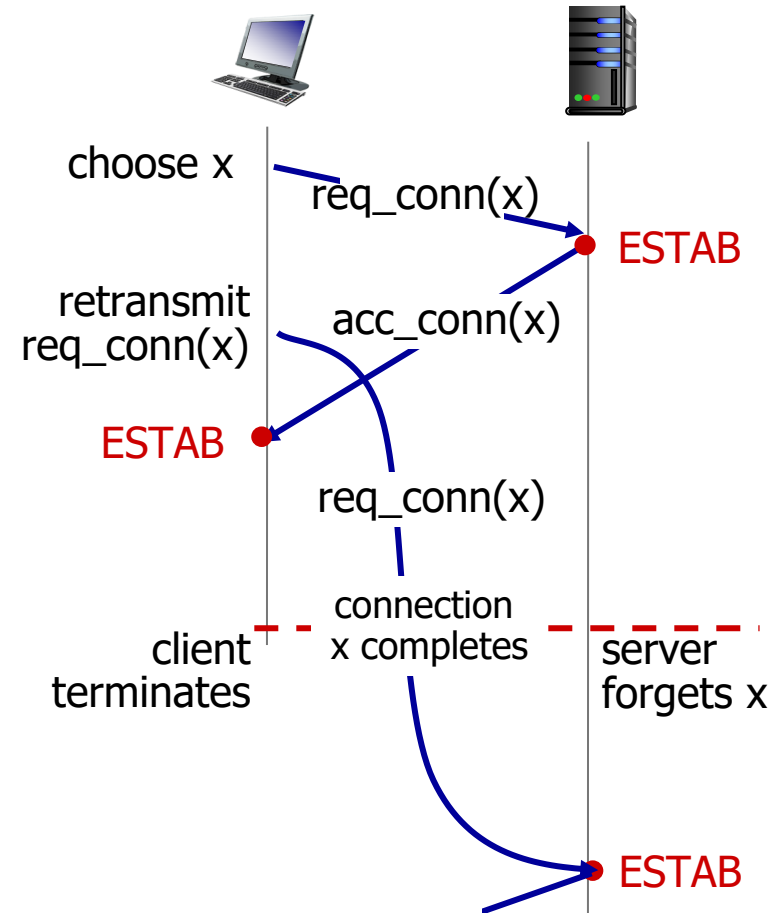
2-way handshake scenarios




No problem!

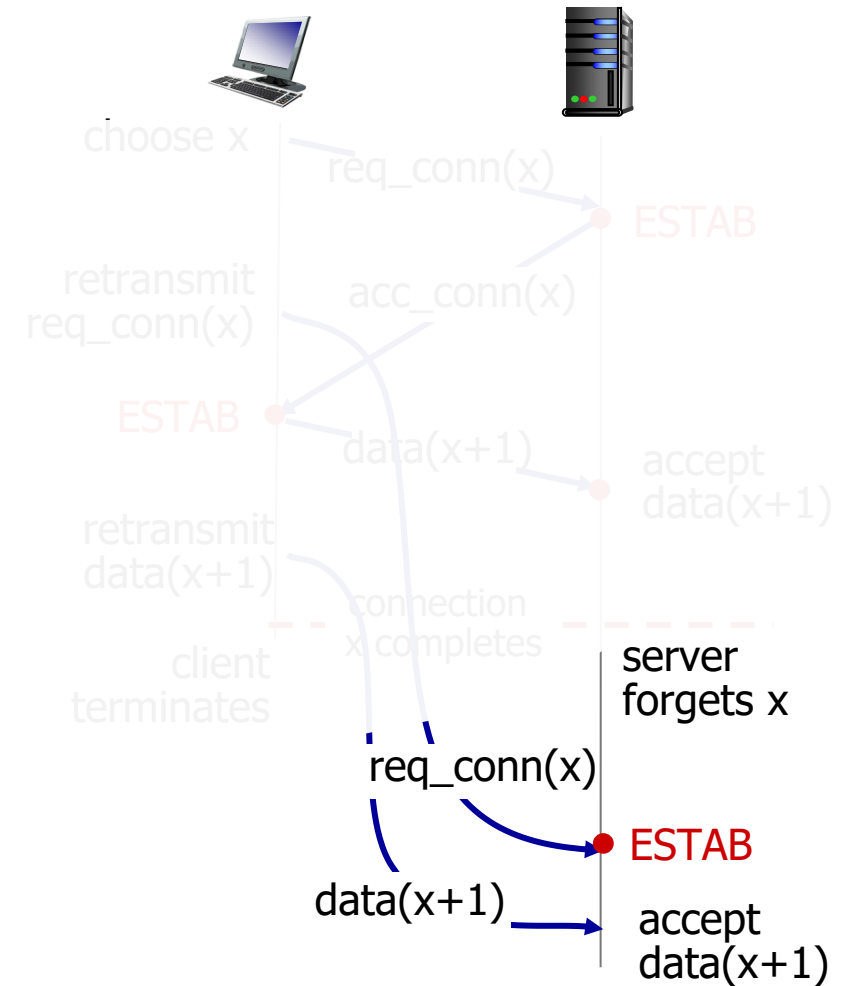


2-way handshake scenarios



 Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!

TCP 3-way handshake

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data



SYNbit=1, Seq=x

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

Server state

```
serverSocket = socket(AF_INET, SOCK_STREAM)  
serverSocket.bind(('', serverPort))  
serverSocket.listen(1)  
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

Closing a TCP connection

- 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

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



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!



congestion control:

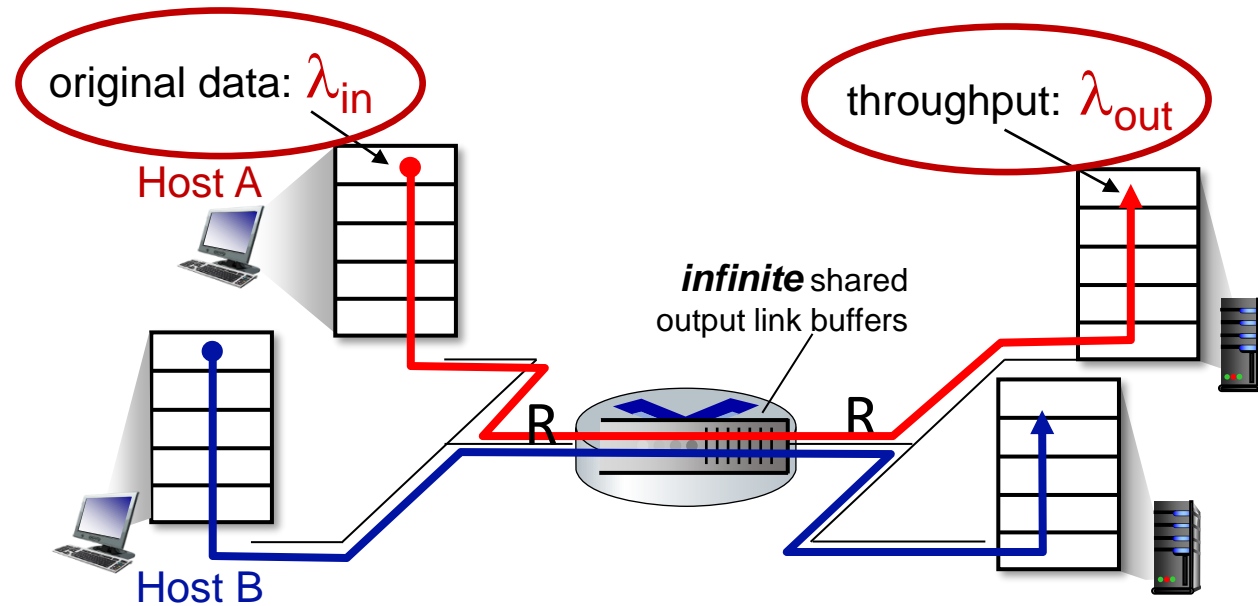
too many senders,
sending too fast

flow control: one sender
too fast for one receiver

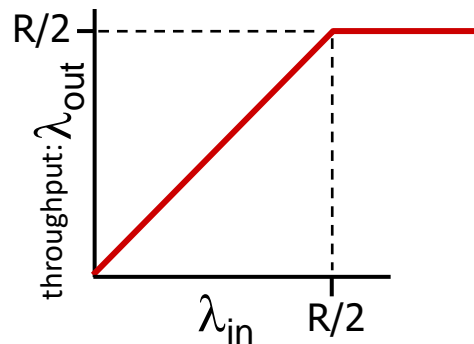
Causes/costs of congestion: scenario 1

Simplest scenario:

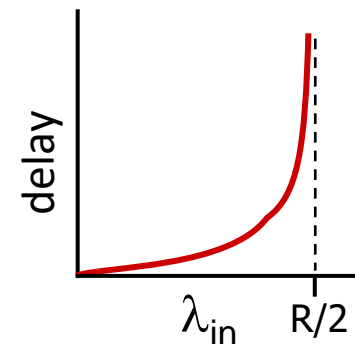
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



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



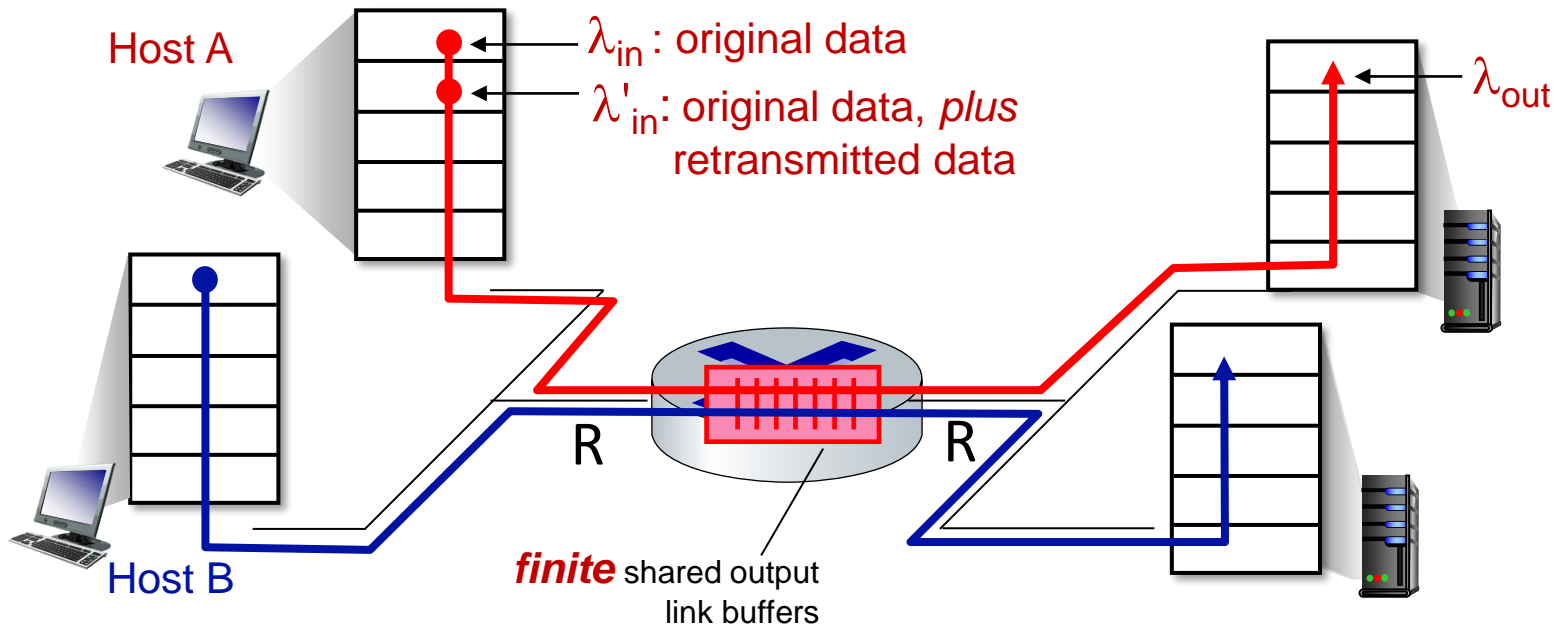
maximum per-connection throughput: $R/2$



large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

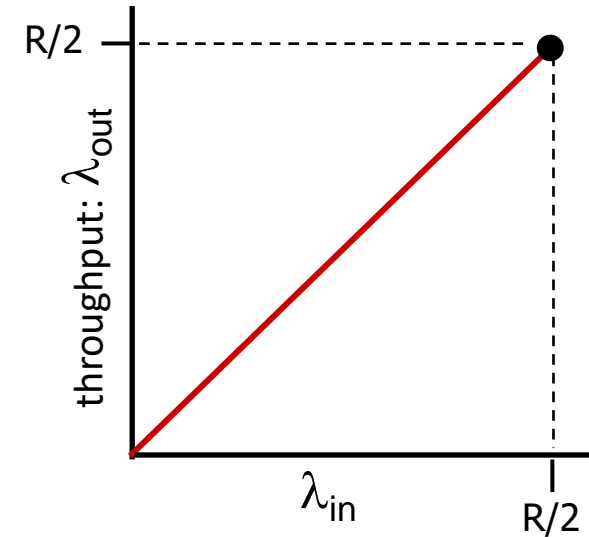
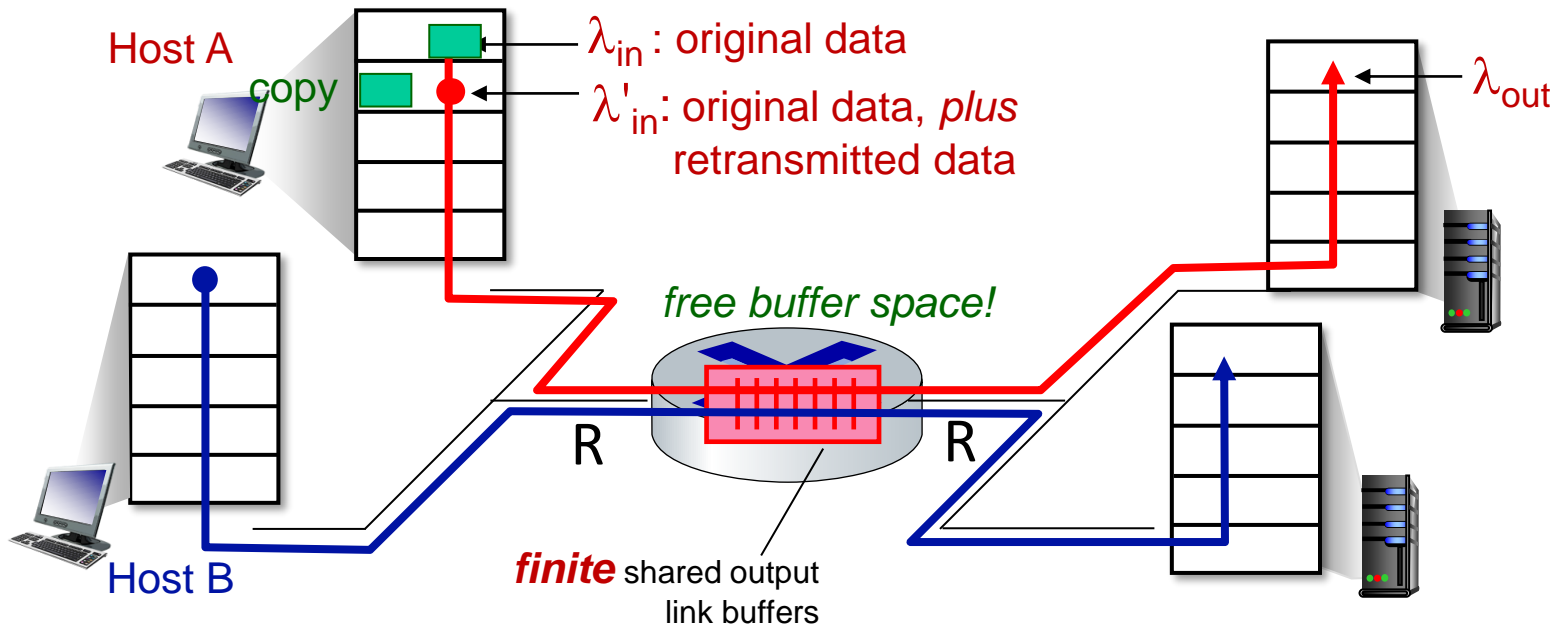
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

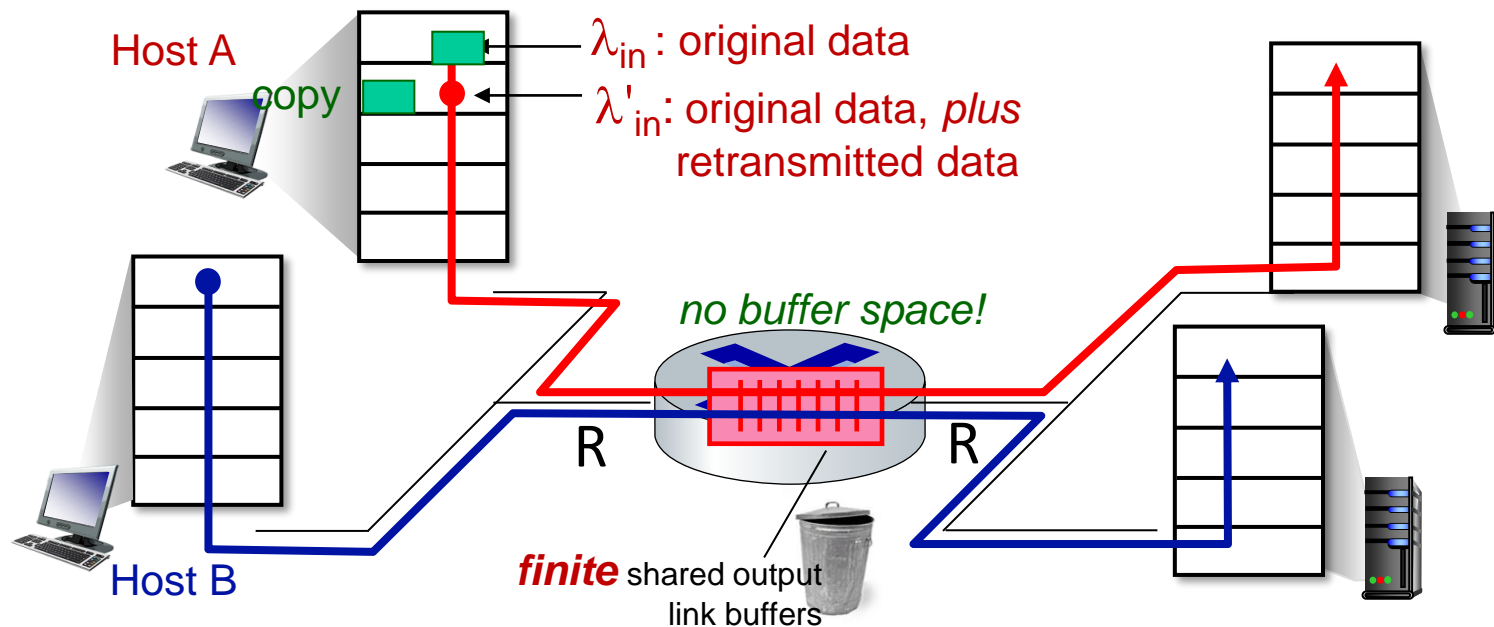
- sender sends only when router buffers available



Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

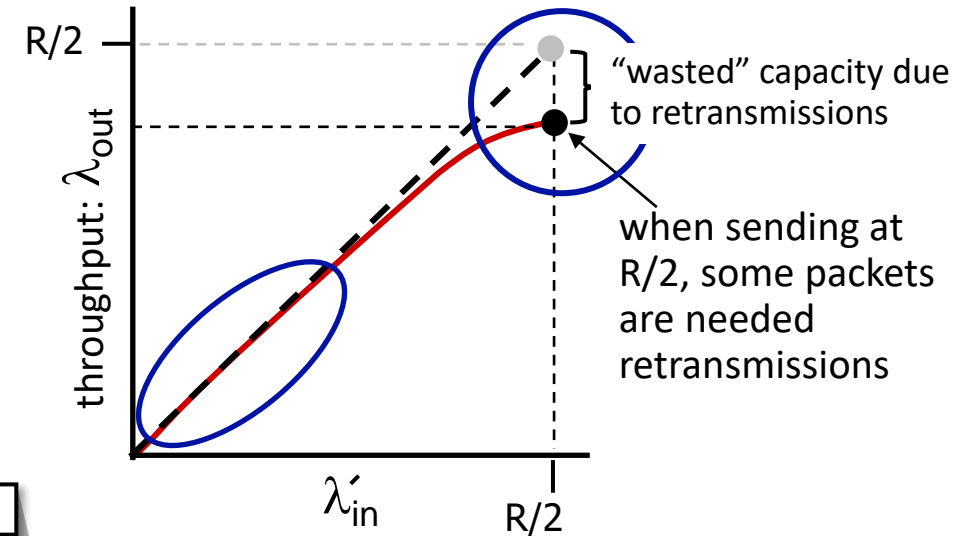
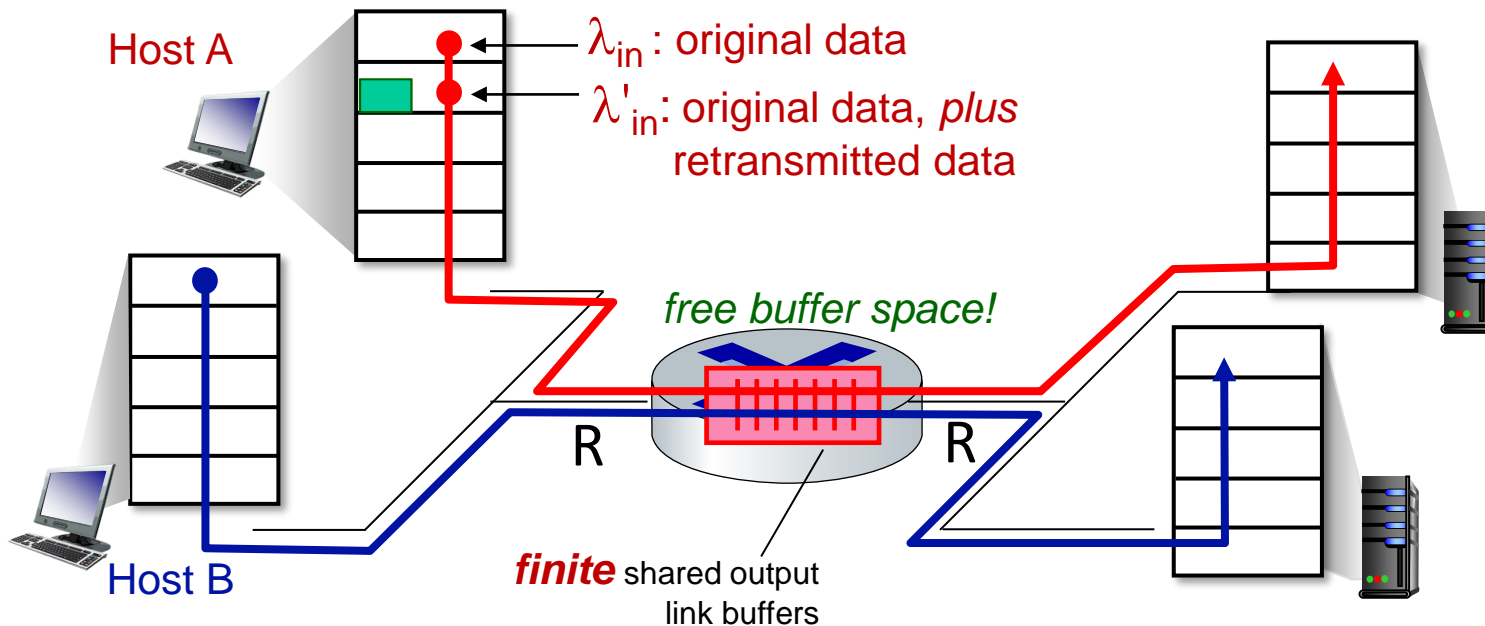
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

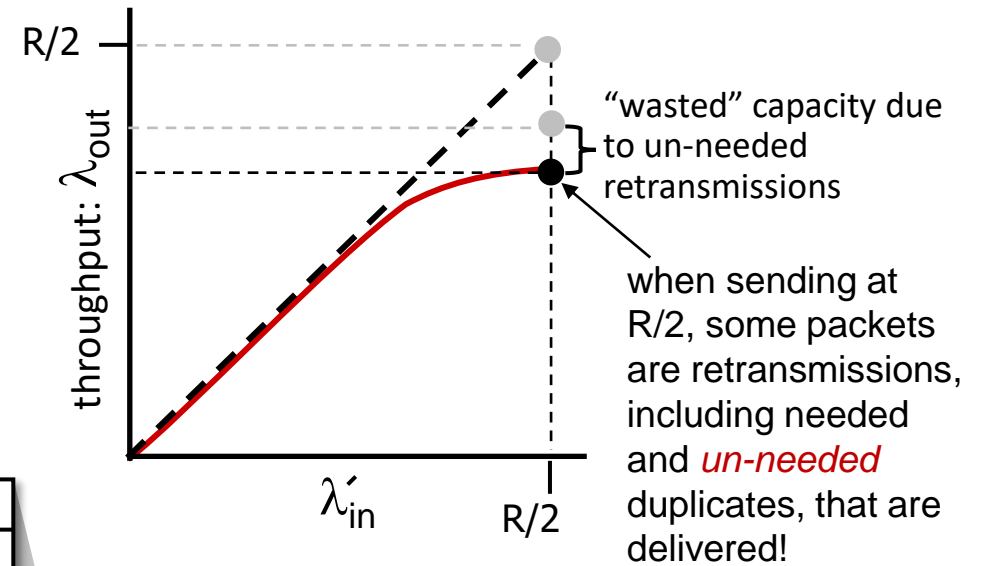
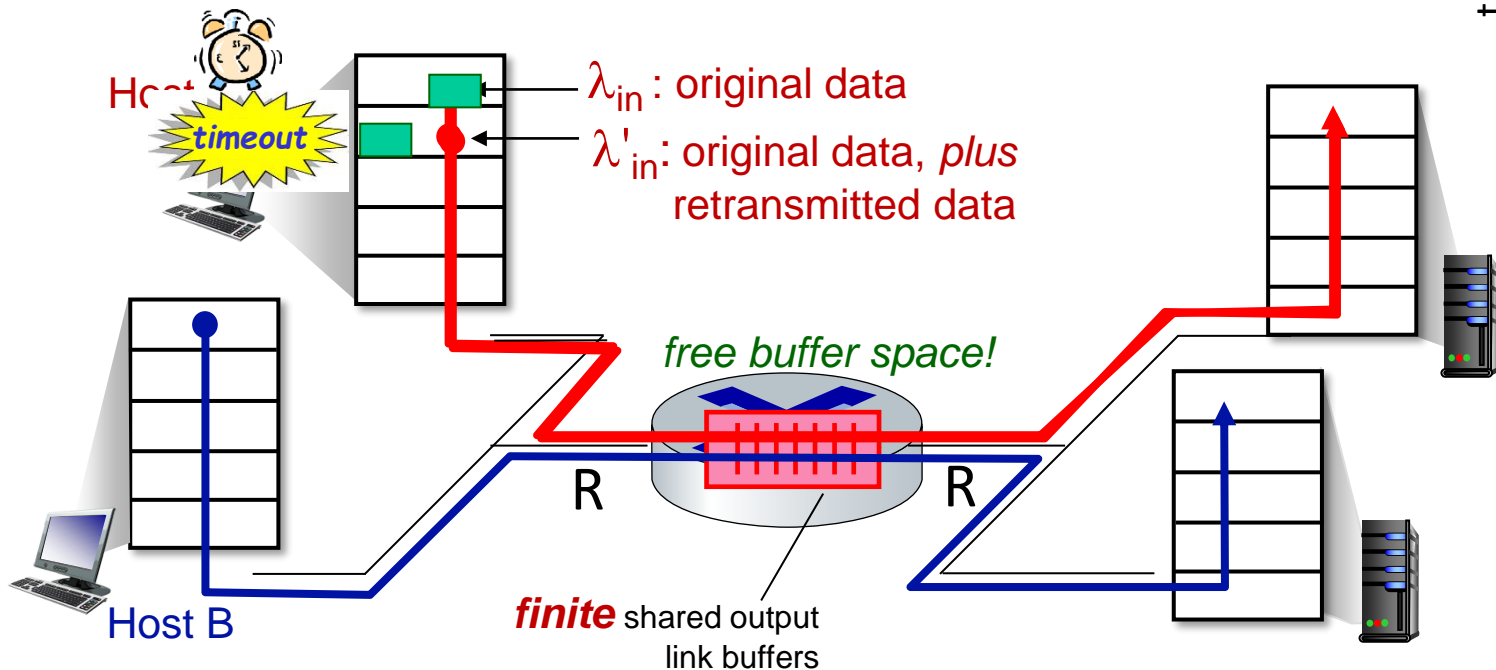
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



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



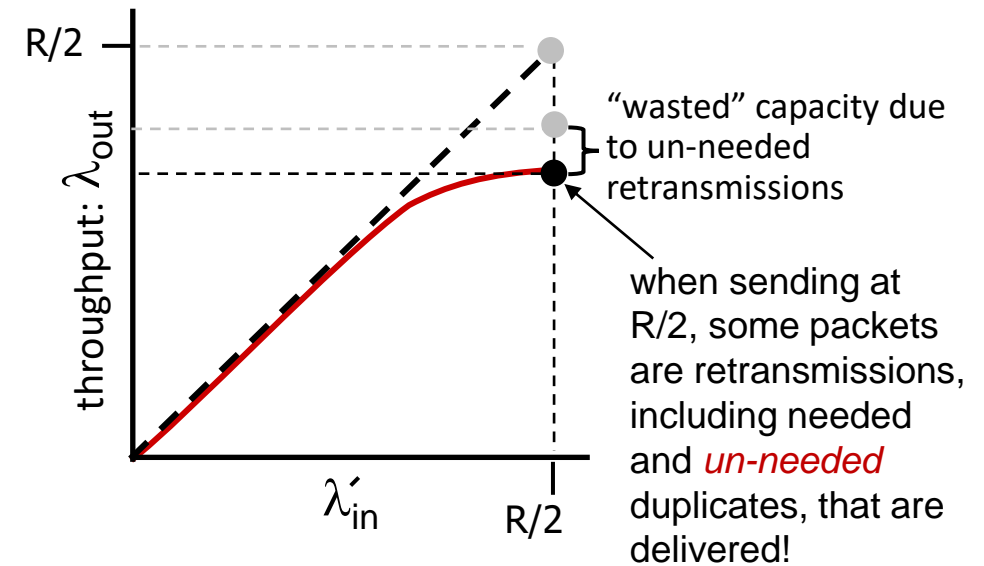
Causes/costs of congestion: scenario 2

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

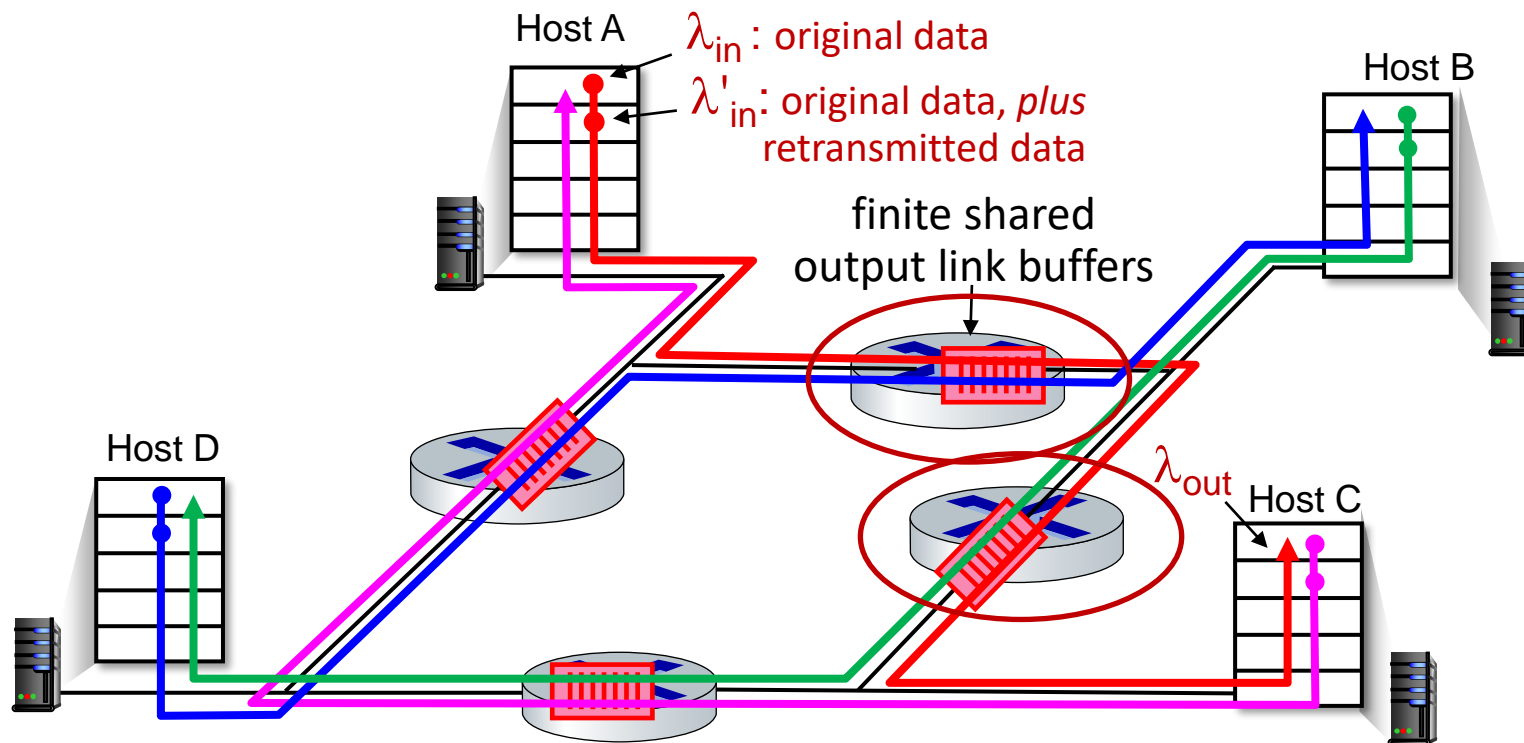


Causes/costs of congestion: scenario 3

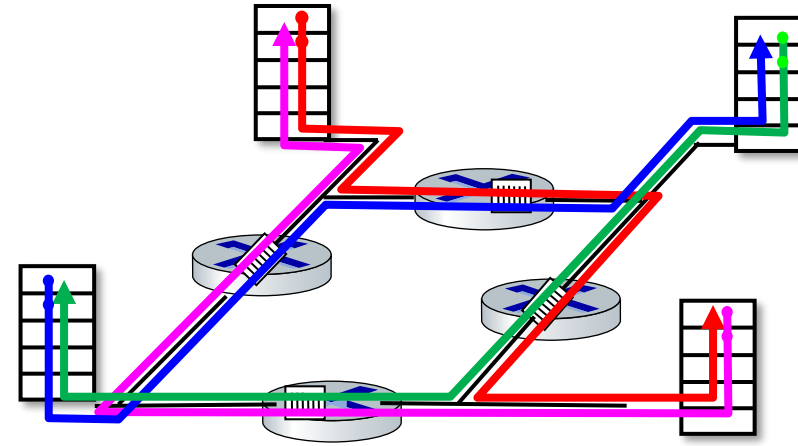
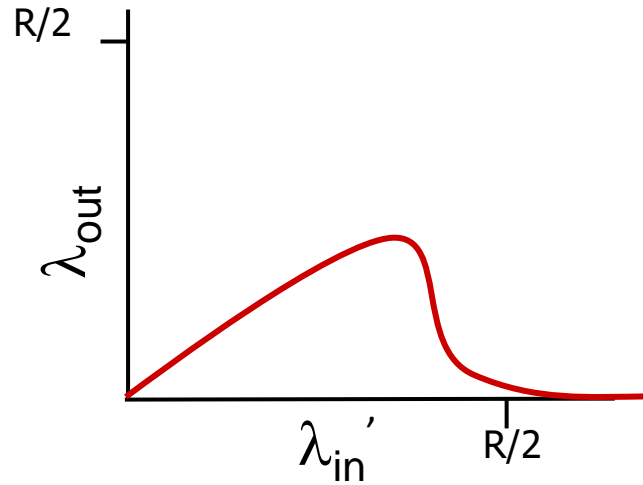
- *four* senders
- *multi-hop* paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?

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



Causes/costs of congestion: scenario 3

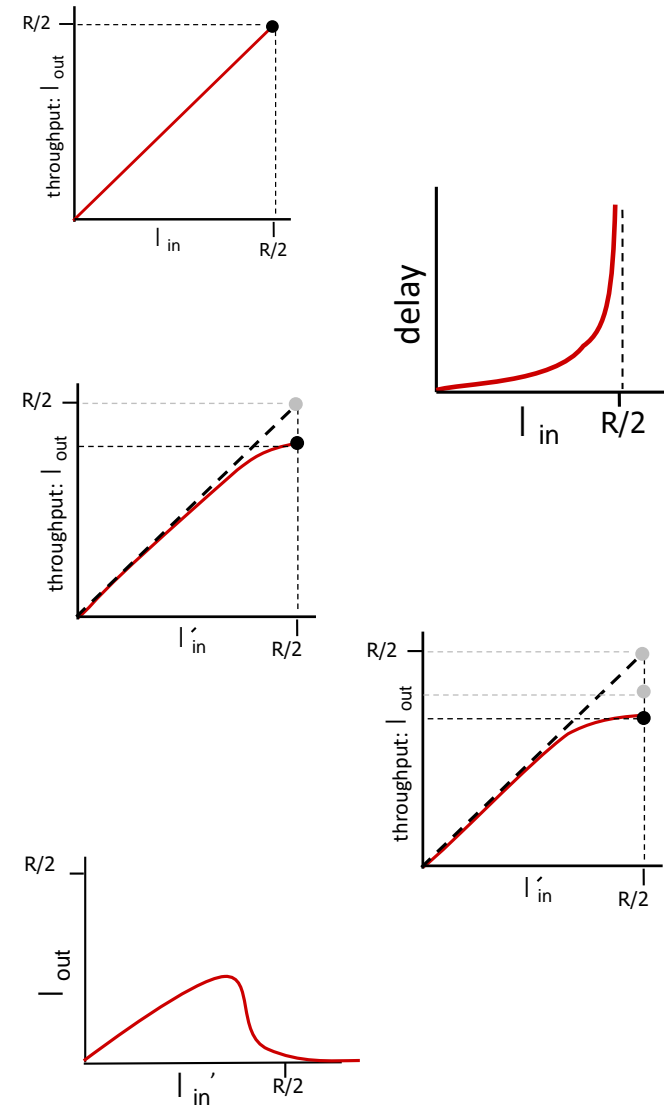


another “cost” of congestion:

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

Causes/costs of congestion: insights

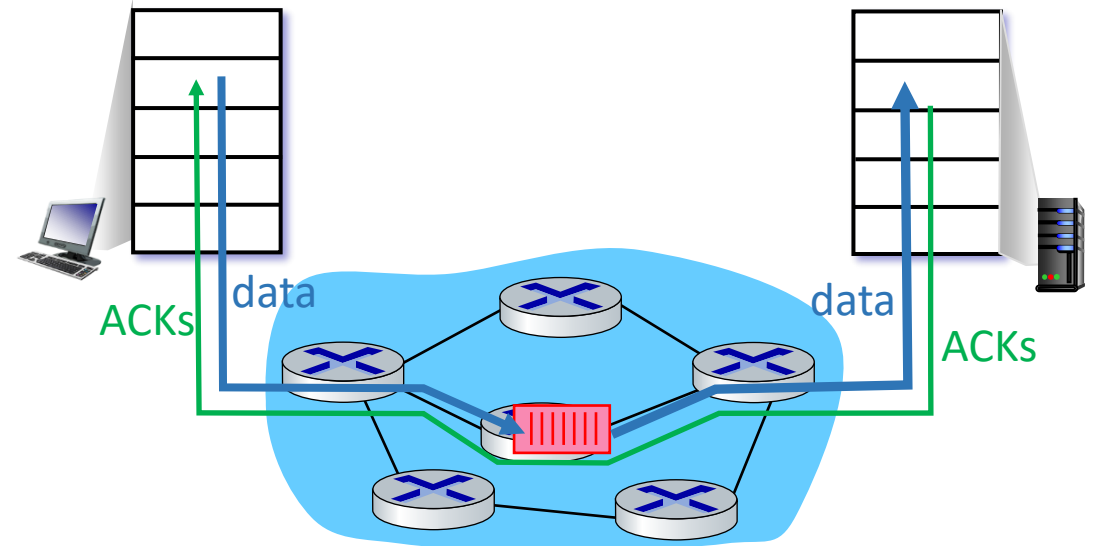
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

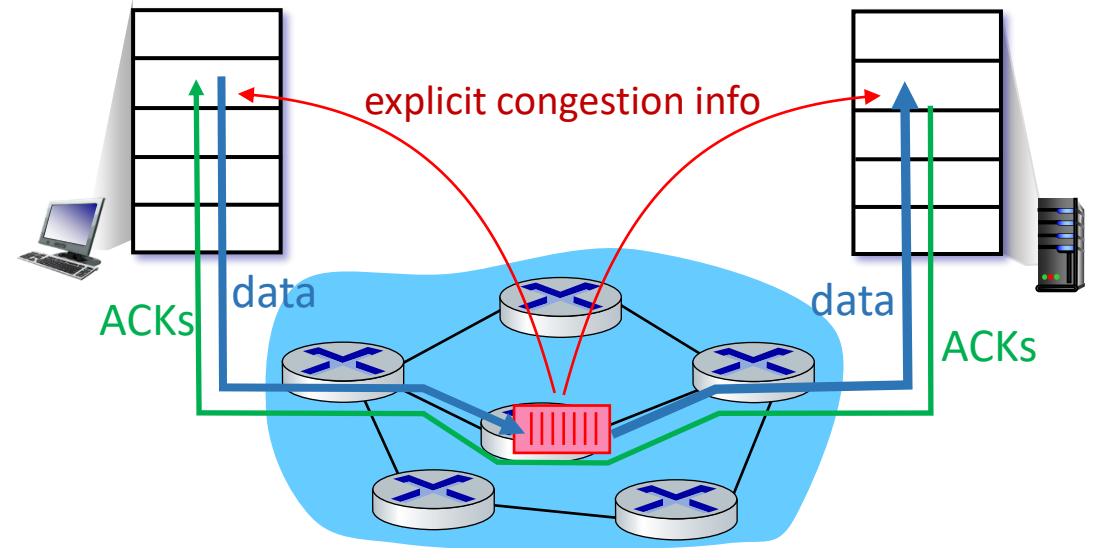
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
 - approach taken by TCP



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



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**
- Evolution of transport-layer functionality



TCP congestion control: AIMD

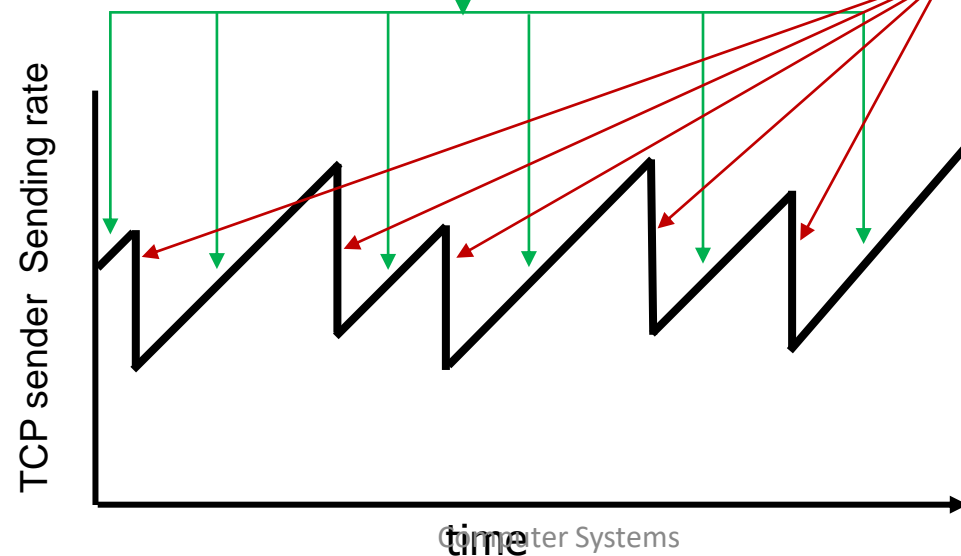
- *approach*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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

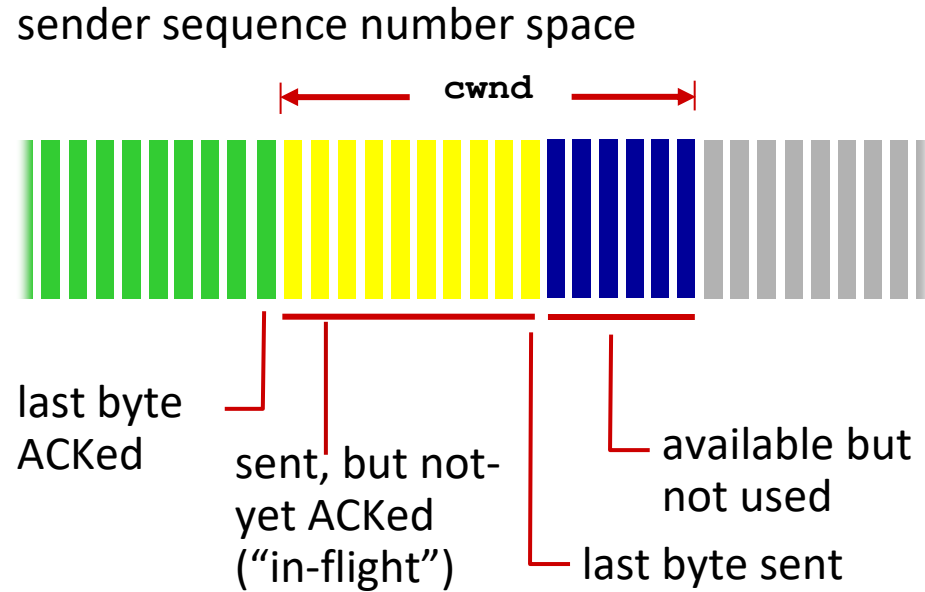
Multiplicative decrease detail: sending rate is

- 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:

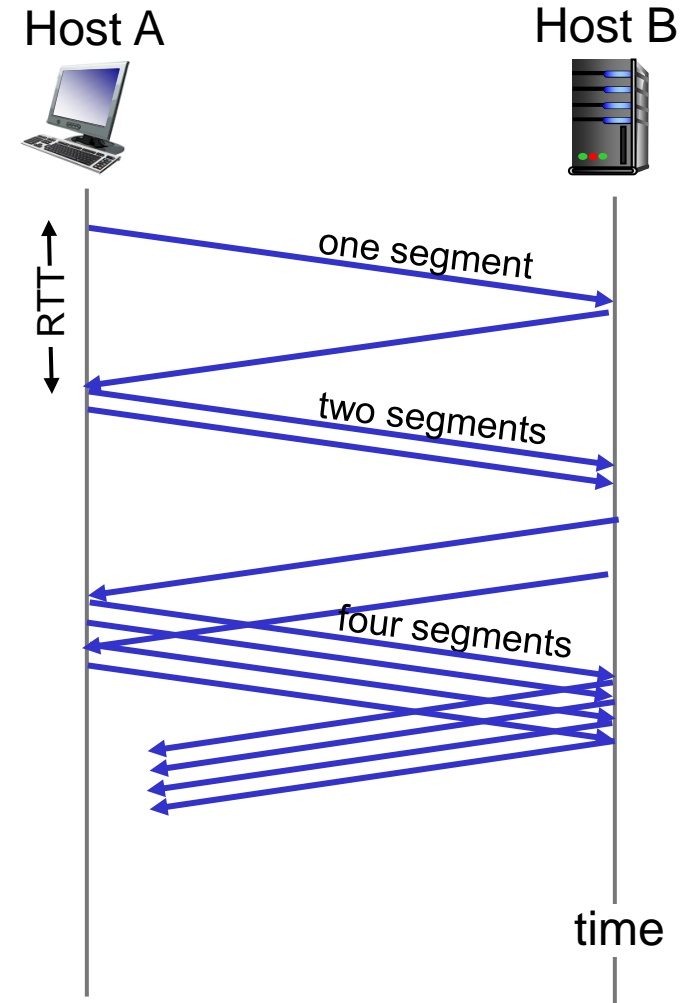
- *roughly*: send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: $\text{LastByteSent} - \text{LastByteAked} \leq \text{cwnd}$
- `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



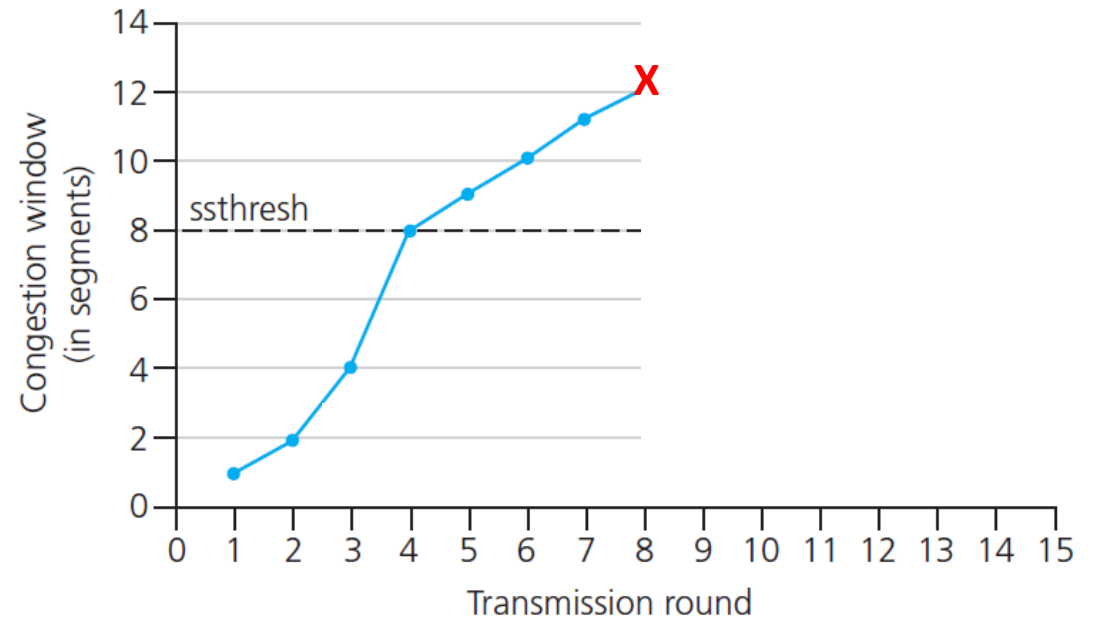
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

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

Implementation:

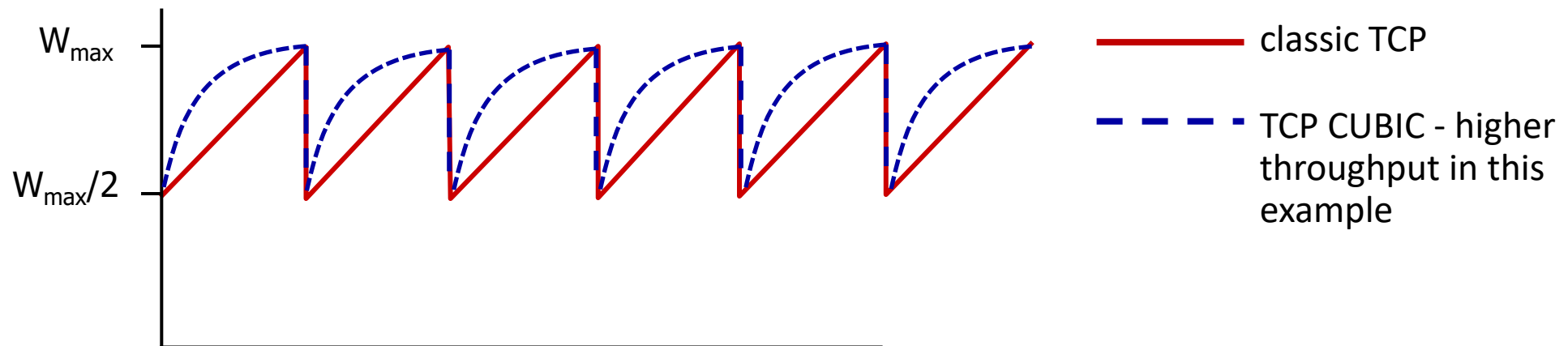
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



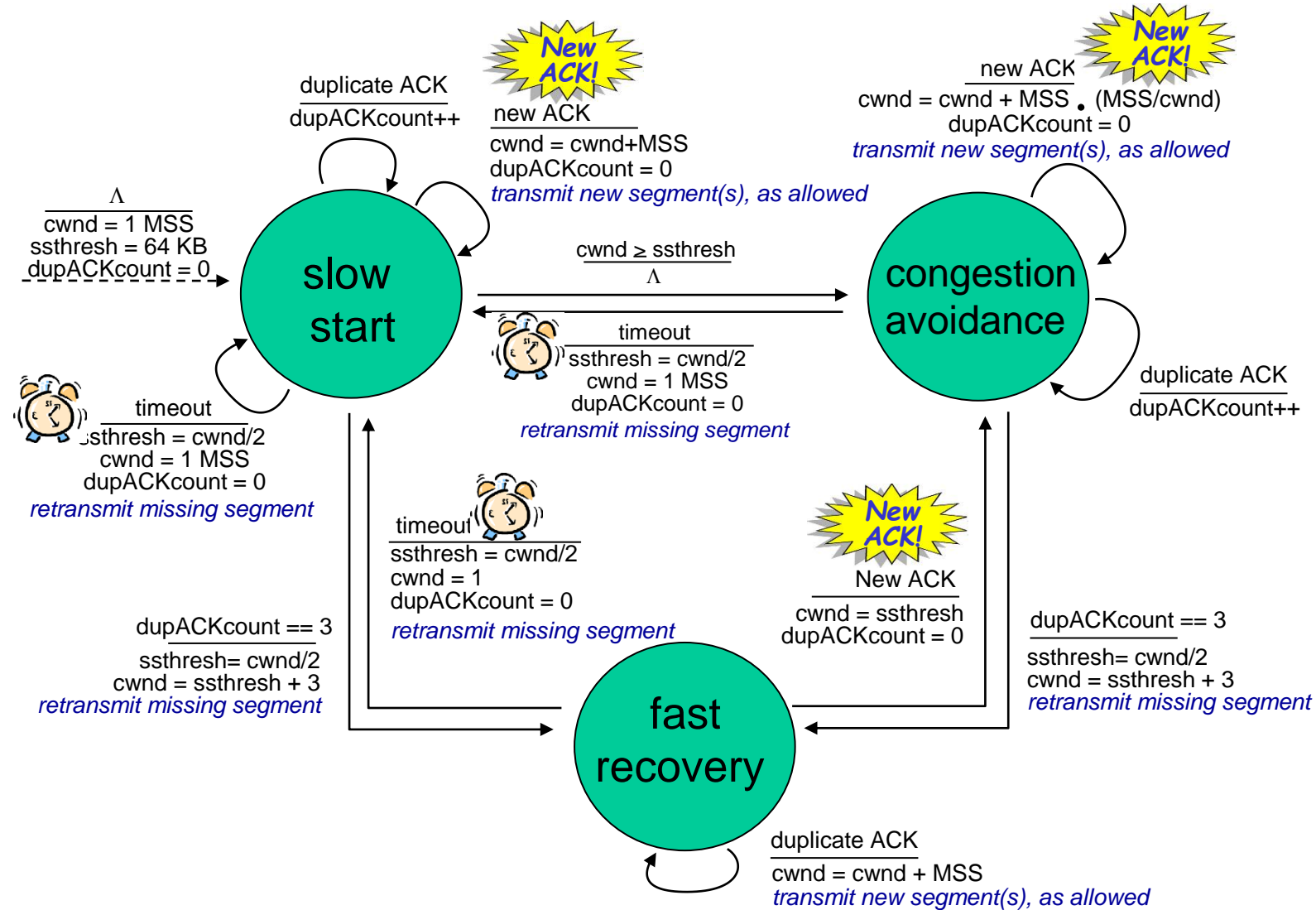
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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 to W_{\max} *faster*, but then approach W_{\max} more *slowly*



Summary: TCP congestion control



- 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



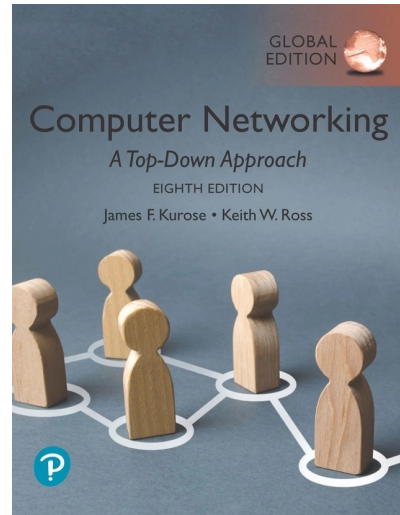
Computer Systems

Networking: Network Layer

Amirali AMIRI

10.06.2024

Sources



- Literature: “Computer Networking: A Top-Down Approach” Written by James F. Kurose and Keith W. Ross
 - https://gaia.cs.umass.edu/kurose_ross/index.php (includes resources for students).
 - They also provide slideshows – the basis for ours! You can investigate the extended version at their website.
 - Also available at the TU library!

Network layer: Roadmap

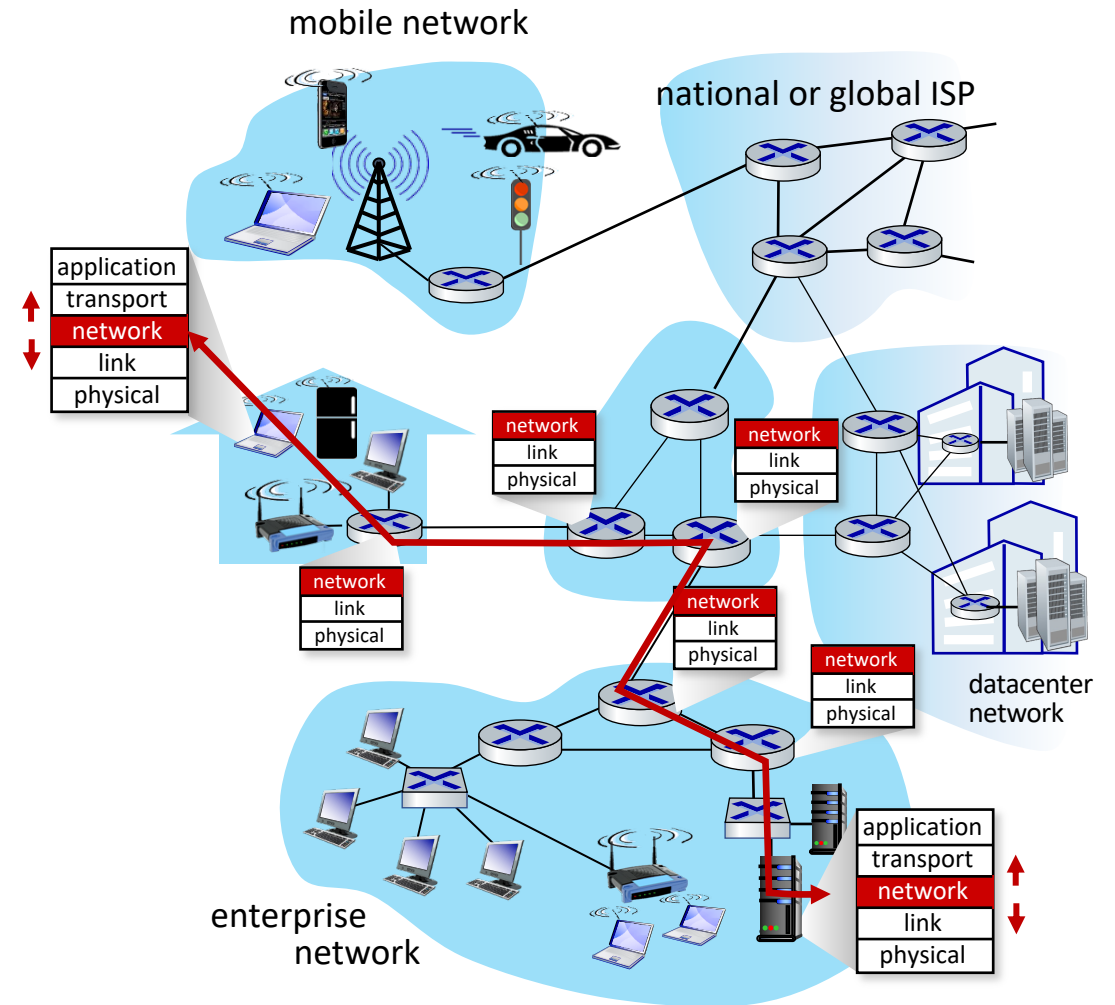
- Network layer: overview
 - data plane
 - control plane
- What's inside a router
 - input ports, switching,
 - output ports, scheduling
- IP: the Internet Protocol
 - datagram format
 - addressing
 - network address translation
 - IPv6



- Control Plane
 - introduction
 - routing algorithm: link state
 - SDN, ICMP

Network-layer services and protocols

- transport segment from sending to receiving host
 - **sender:** encapsulates segments into datagrams, passes to link layer
 - **receiver:** delivers segments to transport layer protocol
- network layer protocols in *every Internet device*: hosts, routers
- **routers:**
 - examines header fields in all IP datagrams passing through it
 - moves datagrams from input ports to output ports to transfer datagrams along end-end path



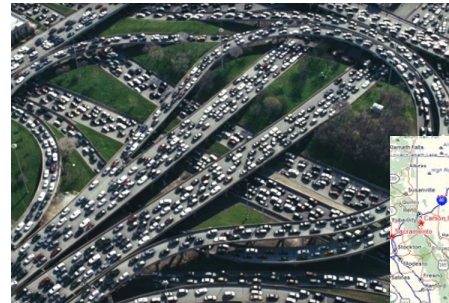
Two key network-layer functions

network-layer functions:

- *forwarding*: move packets from a router's input link to appropriate router output link
- *routing*: determine route taken by packets from source to destination
 - *routing algorithms*

analogy: taking a trip

- *forwarding*: process of getting through single interchange
- *routing*: process of planning trip from source to destination



forwarding

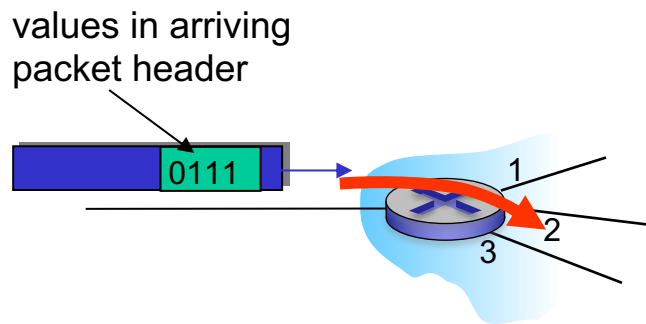


routing

Network layer: data plane, control plane

Data plane:

- *local*, per-router function
- determines how datagram arriving on router input port is forwarded to router output port



Control plane

- *network-wide* logic
- determines how datagram is routed among routers along end-end path from source host to destination host
- two control-plane approaches:
 - *traditional routing algorithms*: implemented in routers
 - *software-defined networking (SDN)*: implemented in (remote) servers

Network layer: Roadmap

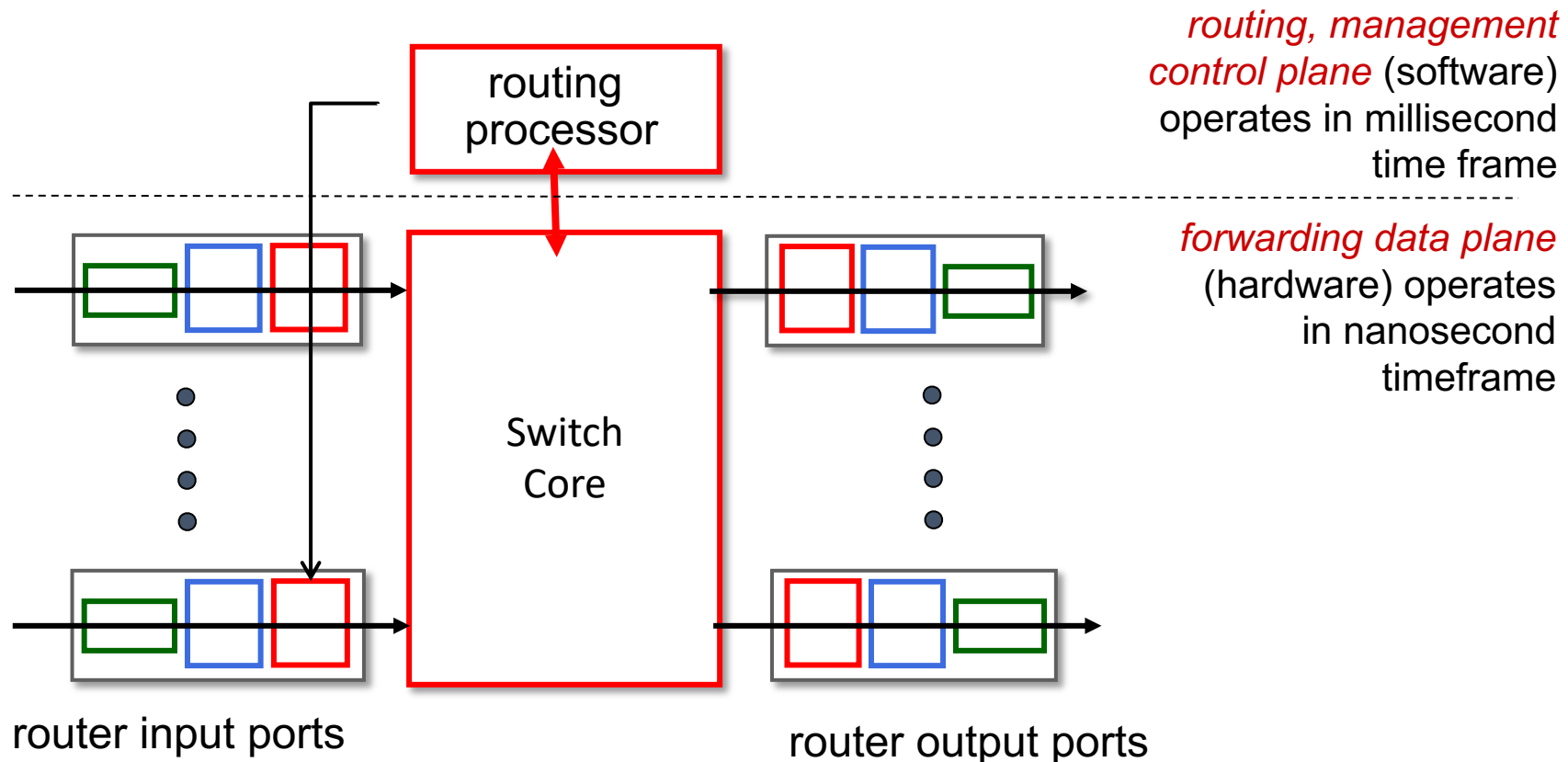
- Network layer: overview
 - data plane
 - control plane
- What's inside a router
 - input ports, switching,
 - output ports, scheduling
- IP: the Internet Protocol
 - datagram format
 - addressing
 - network address translation
 - IPv6



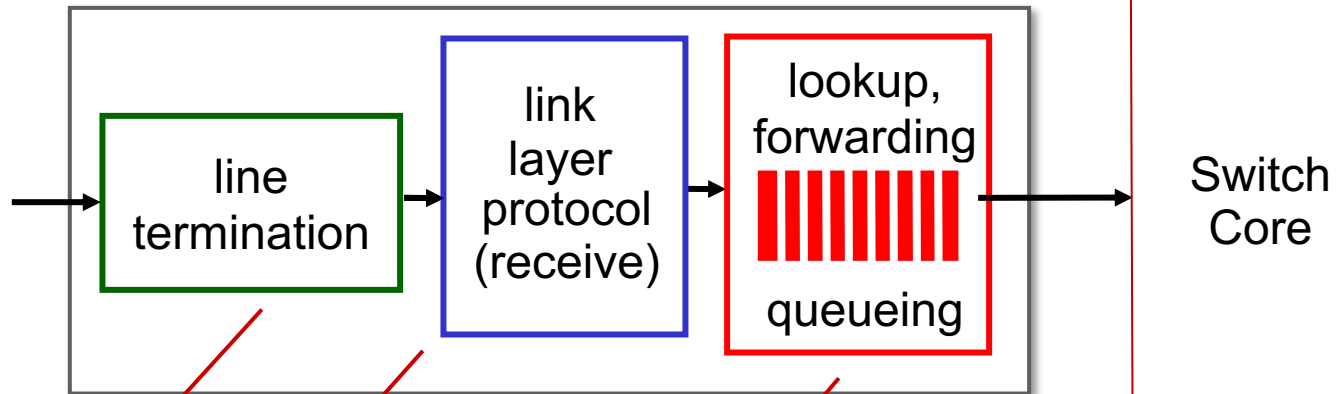
- Control Plane
 - introduction
 - routing algorithm: link state
 - SDN, ICMP

Router architecture overview

high-level view of generic router architecture:



Input port functions



physical layer:
bit-level reception

link layer:
e.g., Ethernet

Switching:

- using header field values, lookup output port, etc.
- **destination-based forwarding**: forward based only on destination IP address (traditional)
- other methods exist, e.g., **generalized forwarding**, but we do not discuss in this lecture.

Destination-based forwarding

forwarding table

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

Q: but what happens if ranges don't divide up so nicely?

Destination-based forwarding

forwarding table

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00010000 00000100 through 11001000 00010111 00010000 00000111	2
11001000 00010111 00011000 11111111	
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

Q: but what happens if ranges don't divide up so nicely?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 00010111 00011*** *****	2
otherwise	3

examples:

11001000 00010111 00010110 10100001 which interface?

11001000 00010111 00011000 10101010 which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 match! 1 00011*** *****	2
otherwise	3

examples:

11001000 00010111 00010110 10100001	which interface?
11001000 00010111 00011000 10101010	which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 00010111 00011*** *****	2
otherwise	3

↑
match!

examples:

11001000 00010111 00010110 10100001	which interface?
11001000 00010111 00011000 10101010	which interface?

Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 00010111 00011*** *****	2
otherwise	3

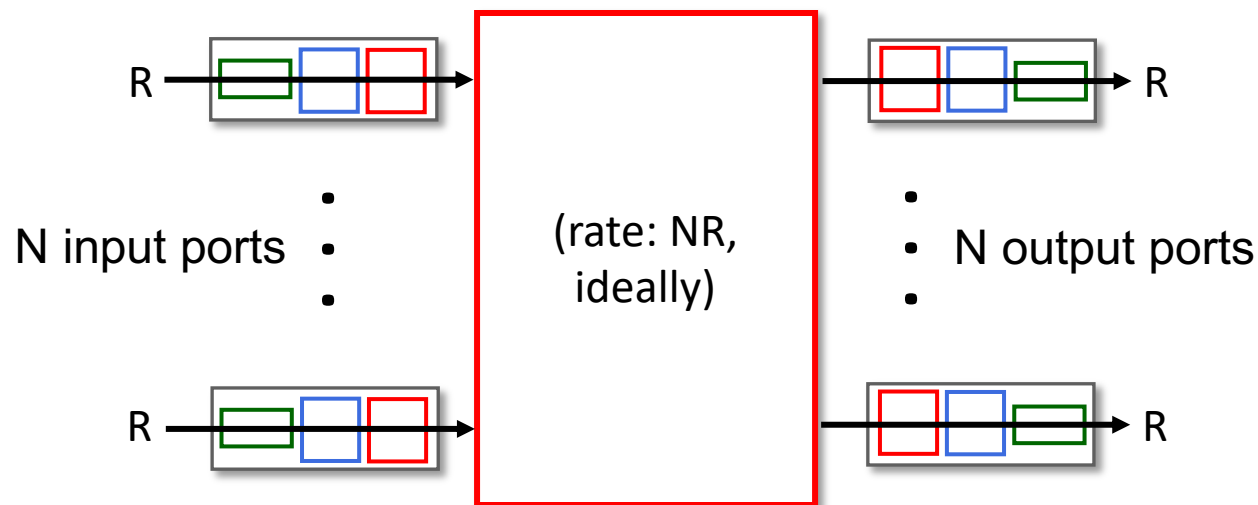
match!

examples:

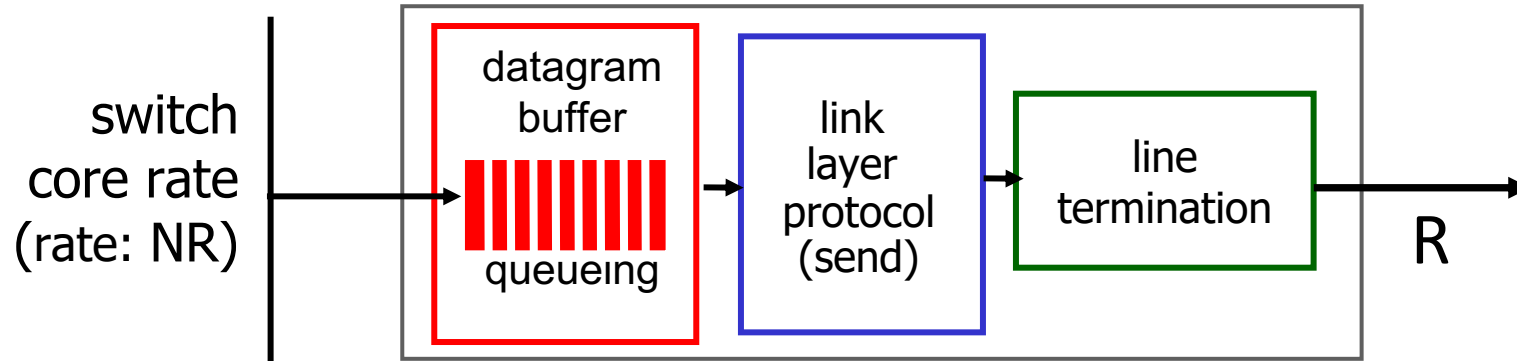
11001000 00010111 00010110	10100001	which interface?
11001000 00010111 00011000	10101010	which interface?

Switching Rate

- transfer packet from input link to appropriate output link
- **switching rate**: rate at which packets can be transferred from inputs to outputs
 - often measured as multiple of input/output line rate
 - N inputs: switching rate N times line rate desirable



Output port queuing



This is a really important slide

- **Buffering** required when datagrams arrive from switch core faster than link transmission rate.
- **Scheduling discipline** chooses among queued datagrams for transmission

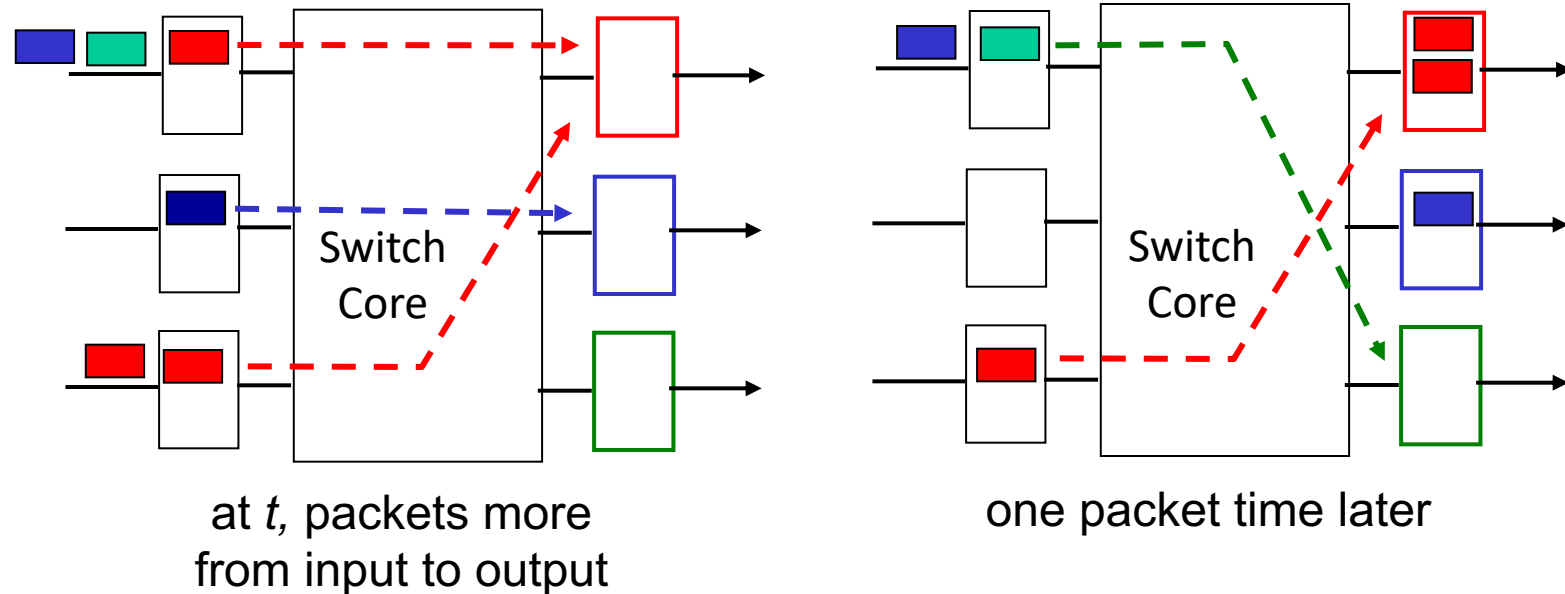


Datagrams can be lost due to congestion, lack of buffers



Priority scheduling – who gets best performance

Output port queuing



- buffering when arrival rate via switch exceeds output line speed
- *queueing (delay) and loss due to output port buffer overflow!*

Packet Scheduling: FCFS

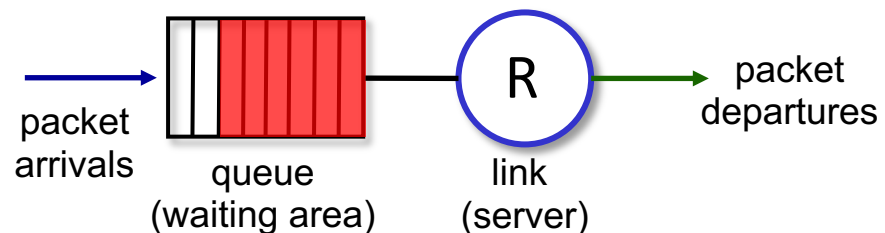
packet scheduling: deciding which packet to send next on link

- first come, first served
- priority
- round robin

FCFS: packets transmitted in order of arrival to output port

- also known as: First-in-first-out (FIFO)
- **Real-world examples?**

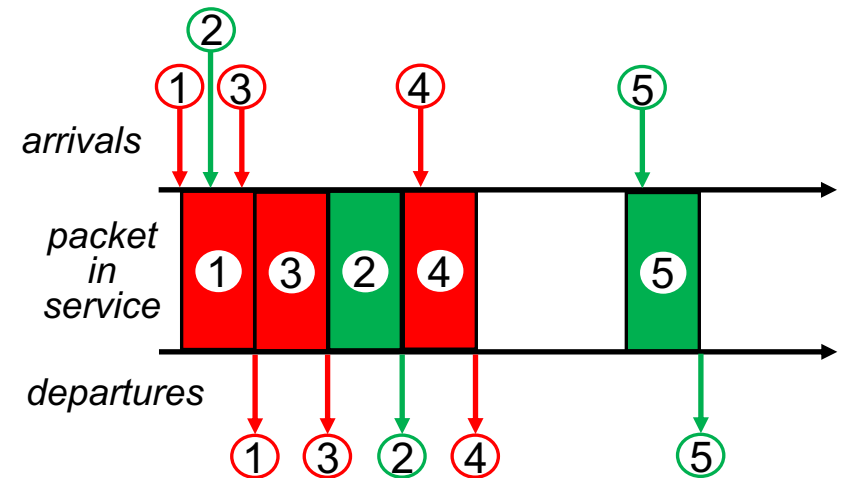
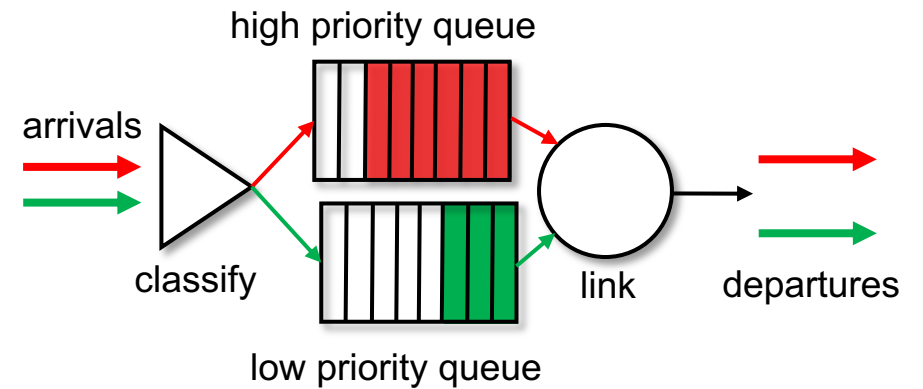
Abstraction: queue



Scheduling policies: priority

Priority scheduling:

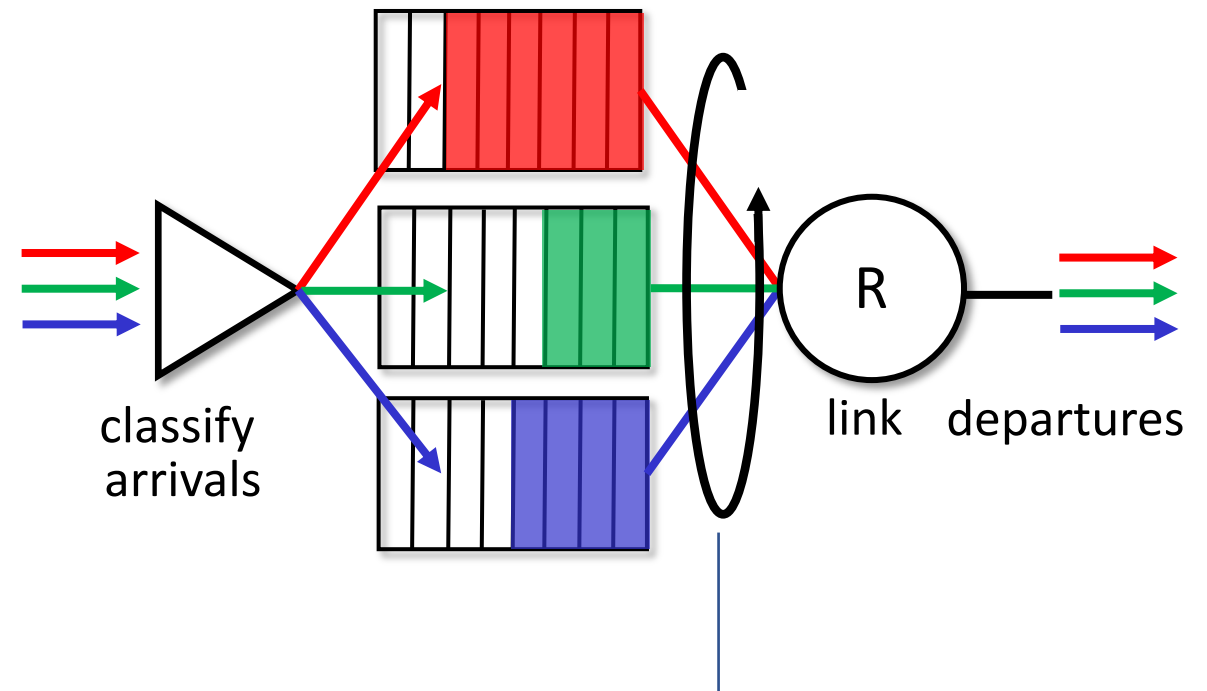
- arriving traffic classified, queued by class
 - any header fields can be used for classification
- send packet from highest priority queue that has buffered packets
 - FCFS within priority class



Scheduling policies: round robin

Round Robin (RR) scheduling:

- arriving traffic classified, queued by class
 - any header fields can be used for classification
- server cyclically, repeatedly scans class queues, sending one complete packet from each class (if available) in turn



Network layer: Roadmap

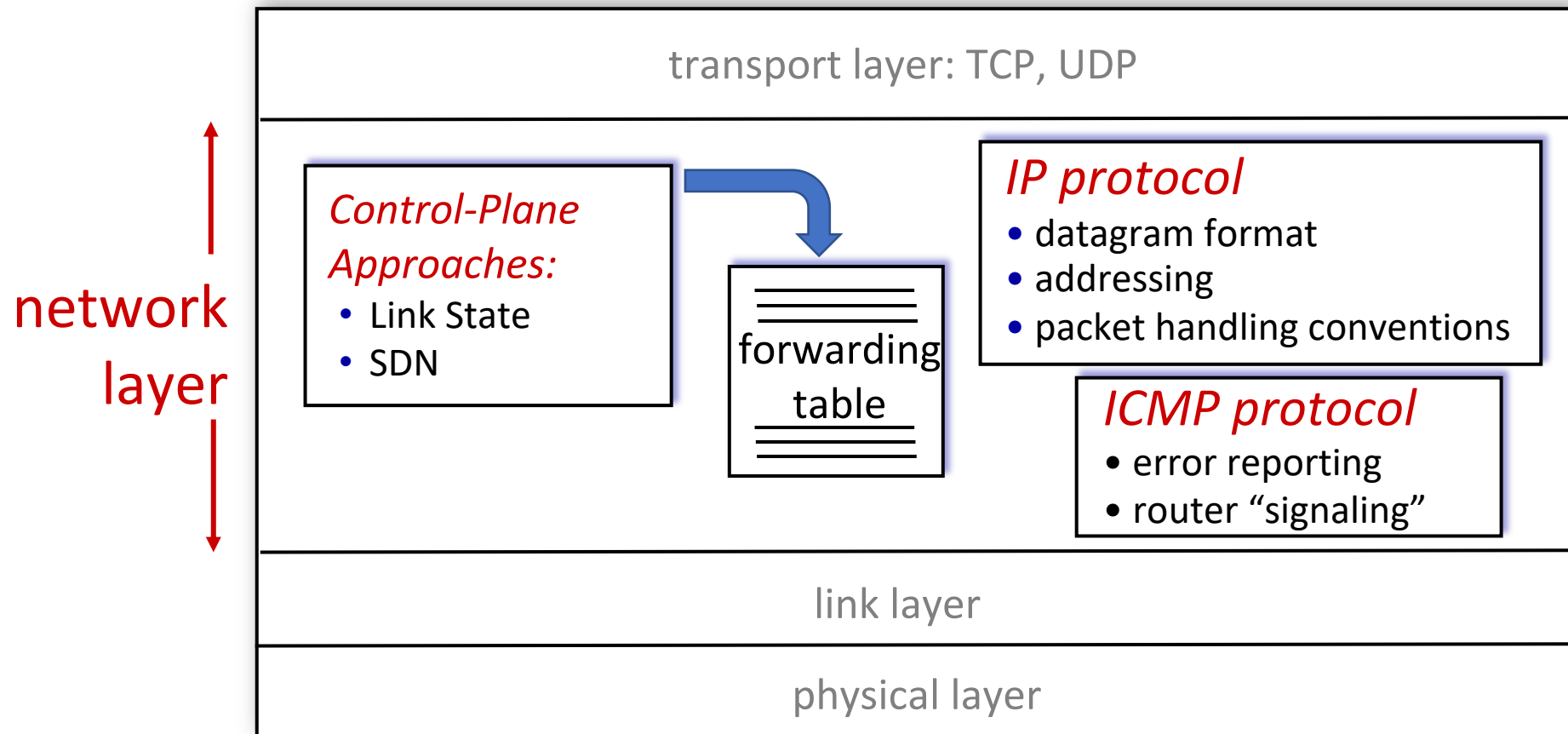
- Network layer: overview
 - data plane
 - control plane
- What's inside a router
 - input ports, switching,
 - output ports, scheduling
- **IP: the Internet Protocol**
 - datagram format
 - addressing
 - network address translation
 - IPv6



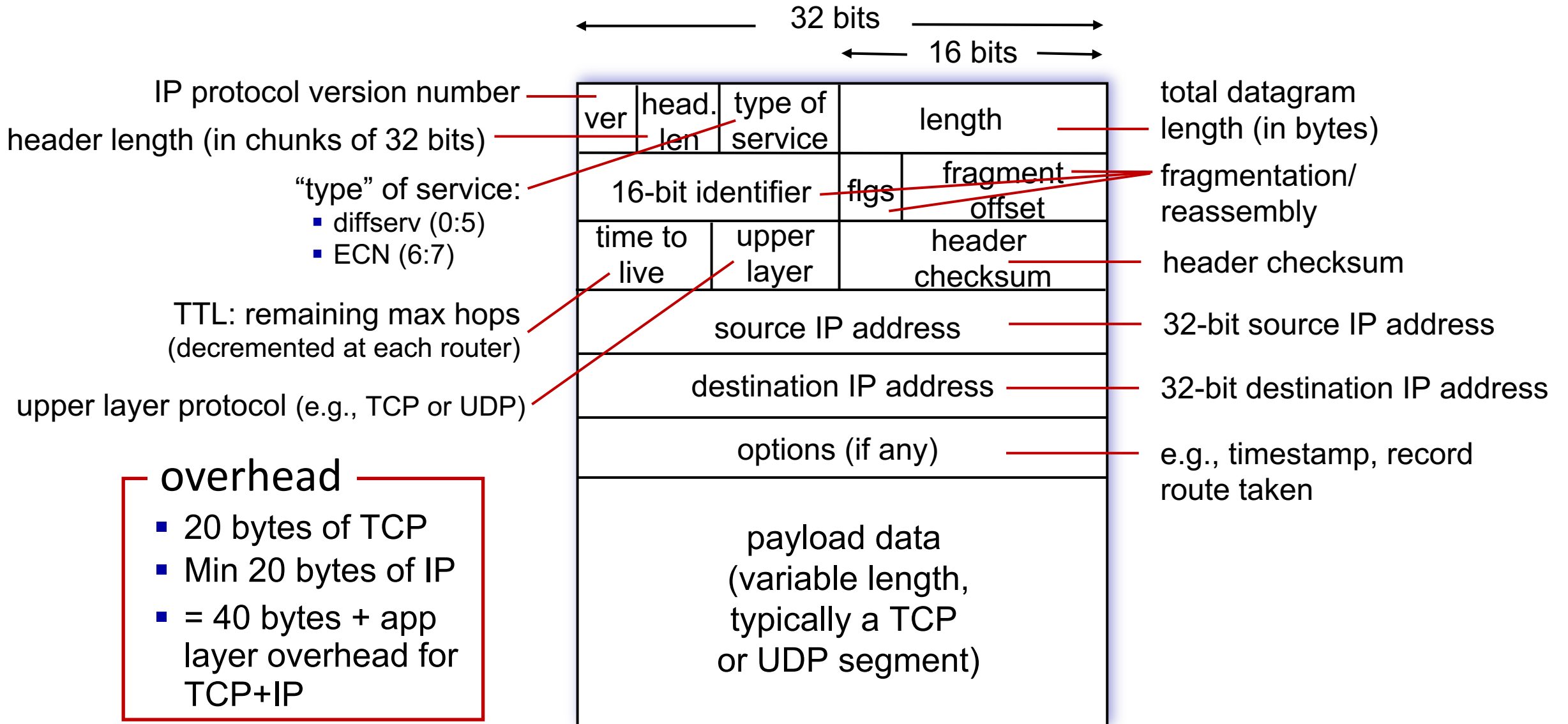
- Control Plane
 - introduction
 - routing algorithm: link state
 - SDN, ICMP

Network Layer: Internet

host, router network layer functions:

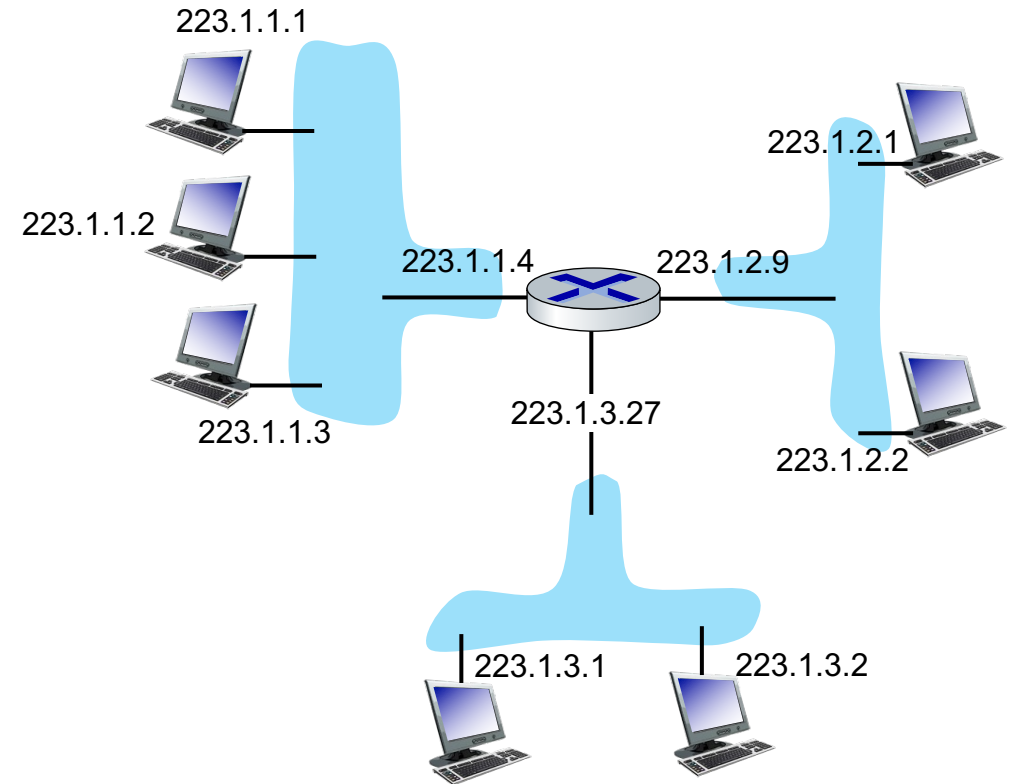


IP Datagram format



IP addressing: introduction

- **IP address:** 32-bit identifier associated with each host or router *interface*
- **interface:** connection between host/router and physical link
 - router's typically have multiple interfaces
 - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)



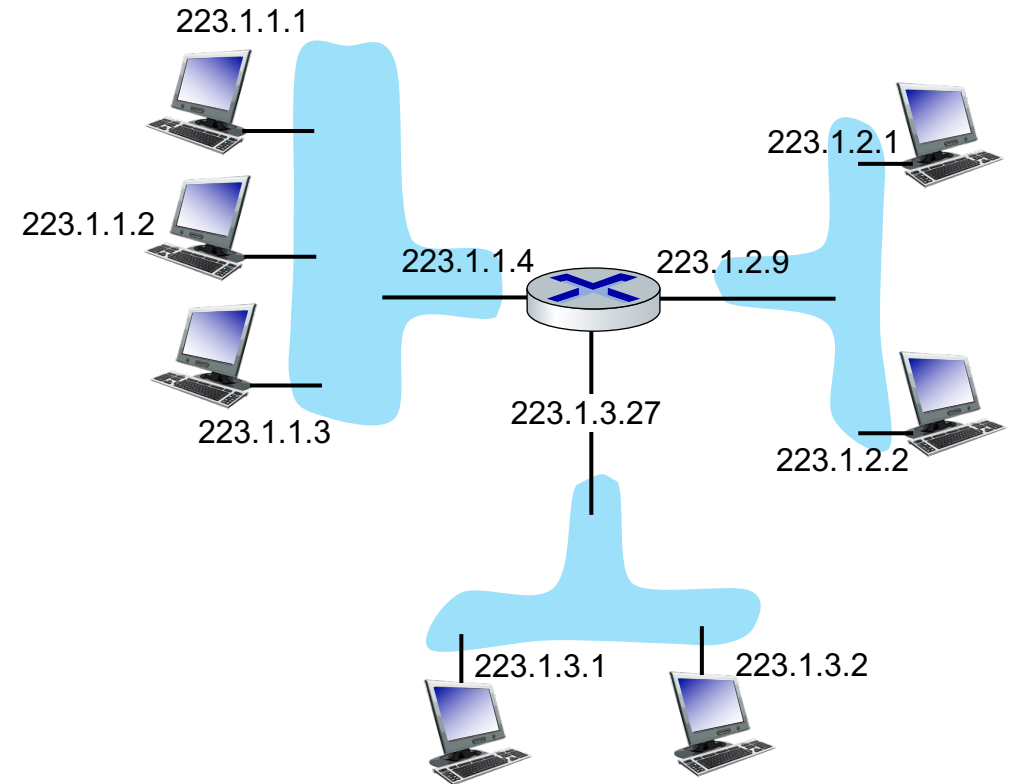
dotted-decimal IP address notation:

223.1.1.1 = 11011111 00000001 00000001 00000001

223 1 1 1

IP addressing: introduction

- **IP address:** 32-bit identifier associated with each host or router *interface*
- **interface:** connection between host/router and physical link
 - router's typically have multiple interfaces
 - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)



dotted-decimal IP address notation:

223.1.1.1 = 11011111 00000001 00000001 00000001

223 1 1 1

Network Layer: 4-26

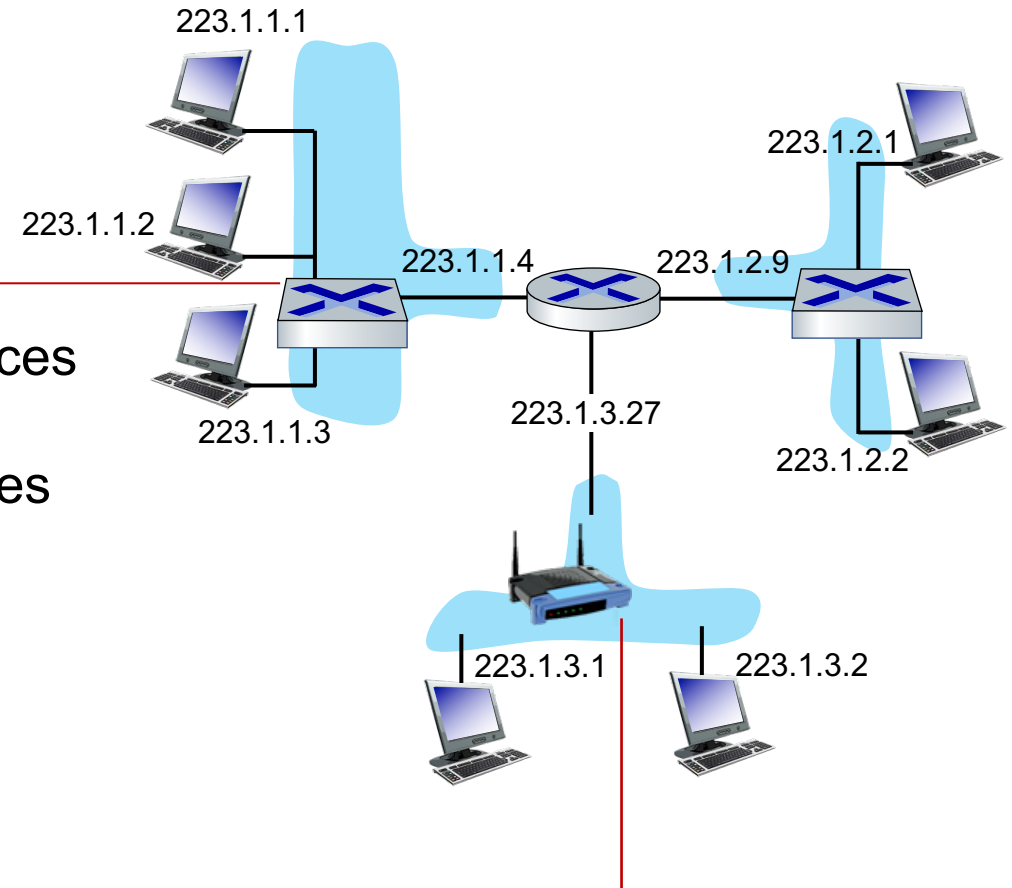
IP addressing: introduction

Q: how are interfaces actually connected?

A: we'll learn about that in the next session.

For now: don't need to worry about how one interface is connected to another (with no intervening router)

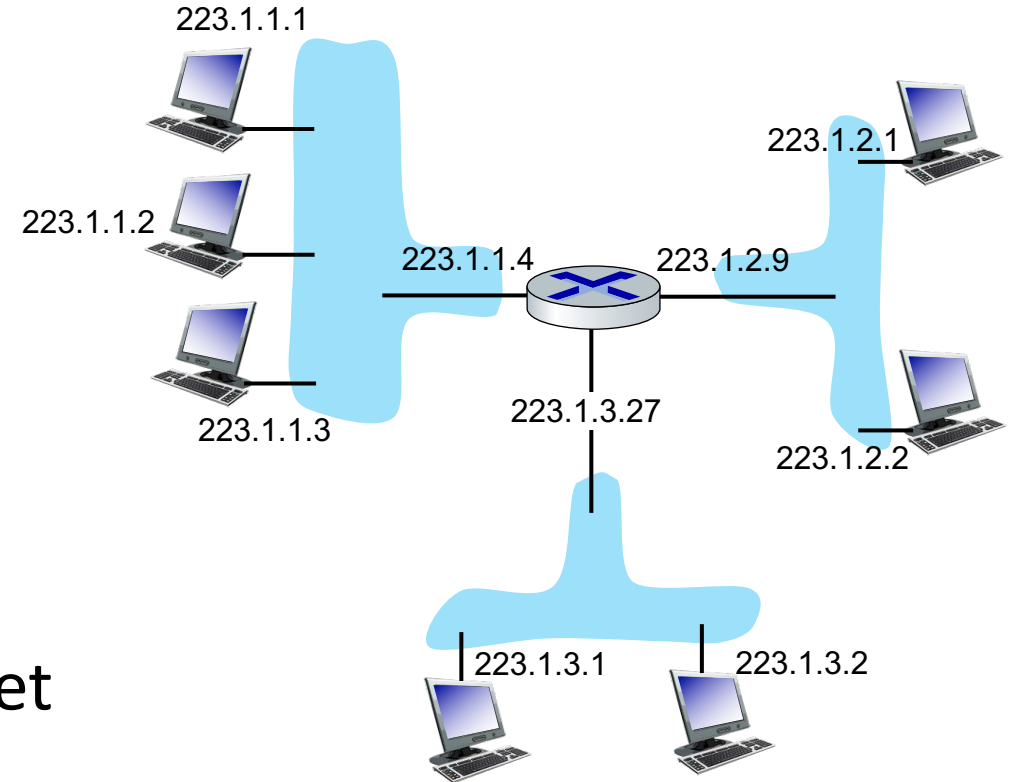
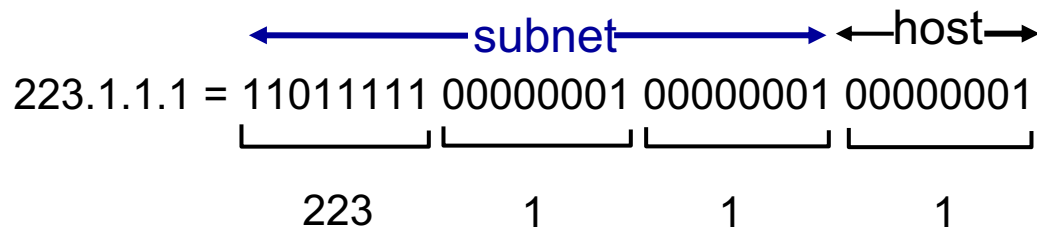
A: wired Ethernet interfaces connected by Ethernet switches



A: wireless WiFi interfaces connected by WiFi base station

Subnets

- *What's a subnet?*
 - device interfaces that can physically reach each other **without passing through an intervening router**
- IP addresses have structure:
 - **subnet part:** devices in same subnet have common high order bits
 - **host part:** **remaining** low order bits



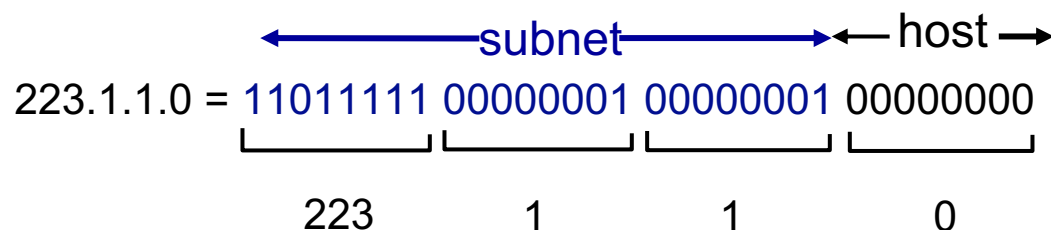
network consisting of 3 subnets

IP addressing: CIDR

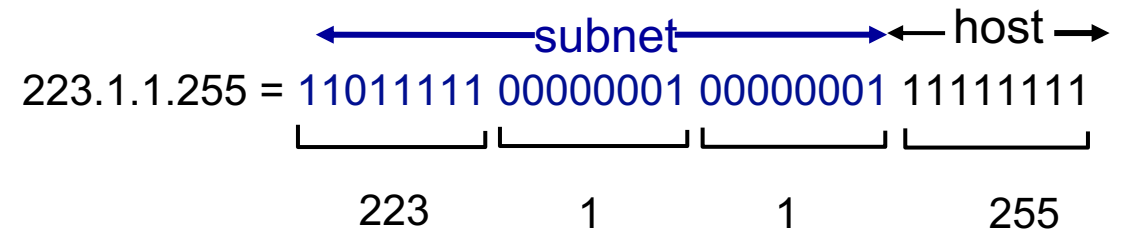
CIDR: Classless InterDomain Routing (pronounced “cider”)

- subnet portion of address of arbitrary length
- address format: **a.b.c.d/x**, where x is # bits in subnet portion
- All-zeros and the all-ones host values are reserved for the **subnet address** and its **broadcast address**.

subnet 223.1.1.0 /24



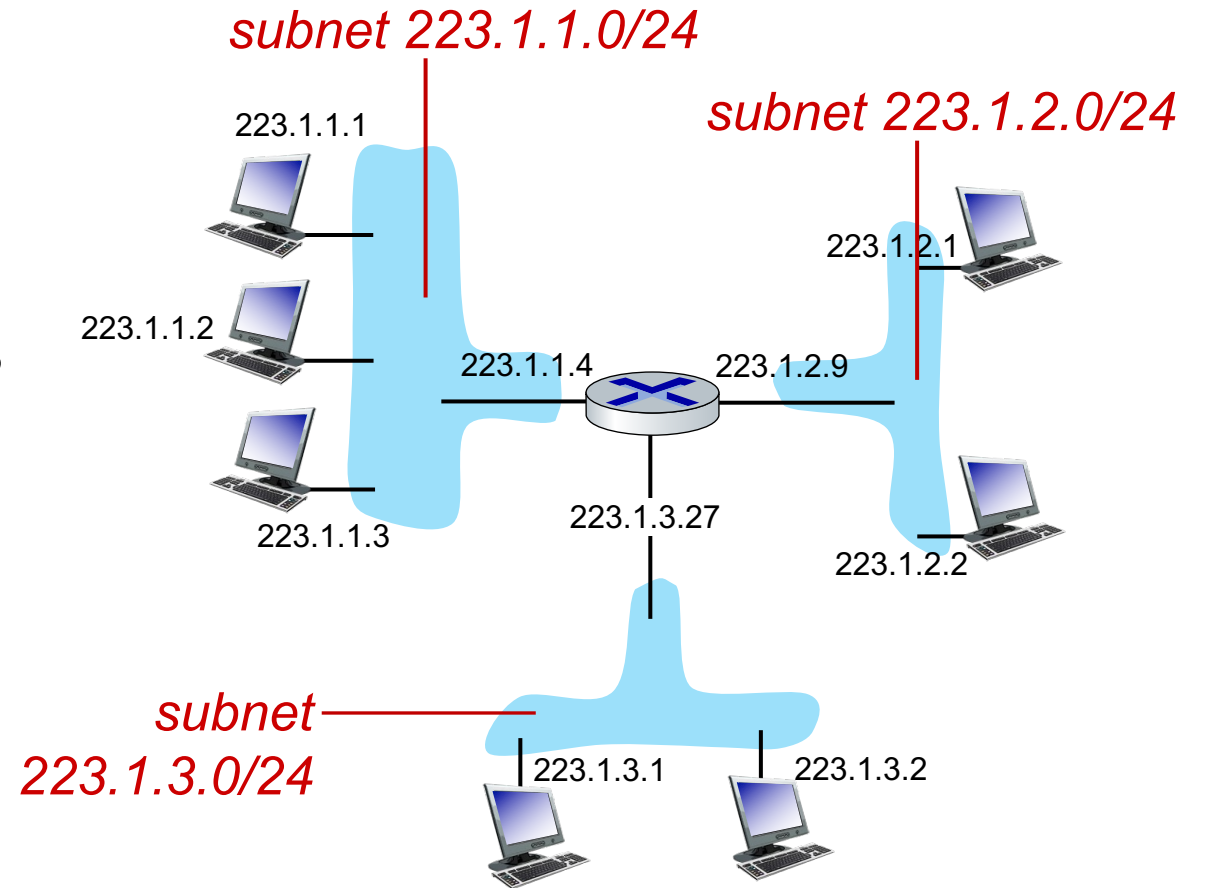
Broadcast Address



Subnets

Recipe for defining subnets:

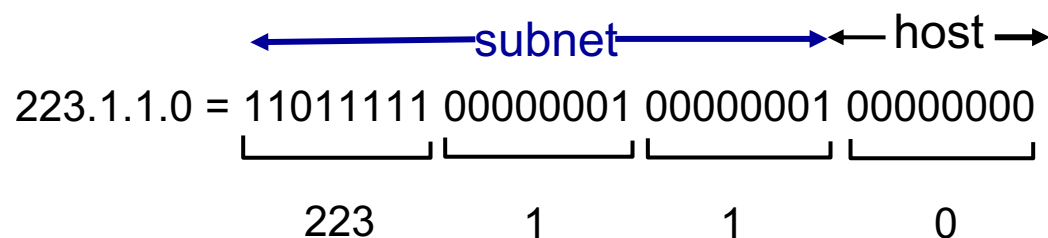
- detach each interface from its host or router, creating “islands” of isolated networks
- each isolated network is called a *subnet*



subnet mask: /24

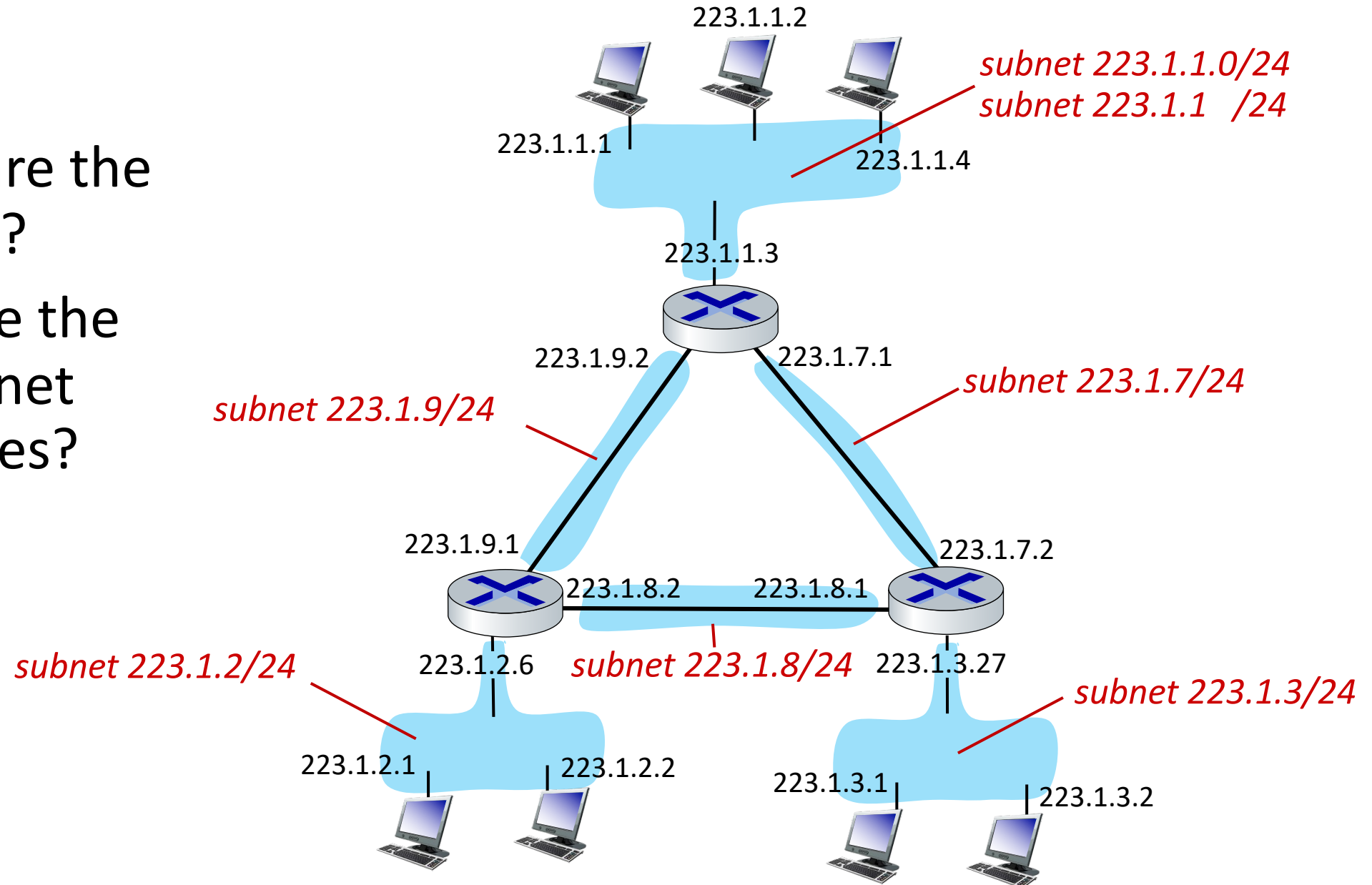
(high-order 24 bits: subnet part of IP address)

subnet 223.1.1.0 /24



Subnets

- where are the subnets?
- what are the /24 subnet addresses?



IP addresses: how does a host get one?

- hard-coded by sysadmin in config file (e.g., /etc/rc.config in UNIX)
- **DHCP: Dynamic Host Configuration Protocol** “plug-and-play”

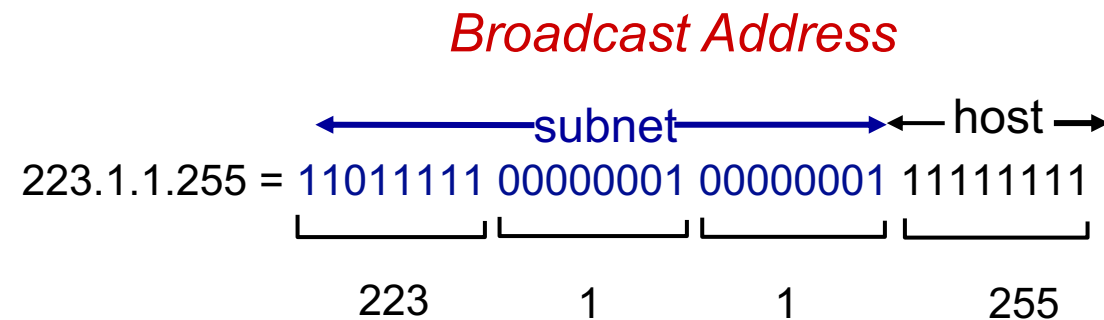
goal: A host obtains IP address *dynamically* from the network server when it “joins” a network

- can renew its lease on address in use
- allows reuse of addresses (only hold address while connected/on)
- support for mobile users who join/leave network

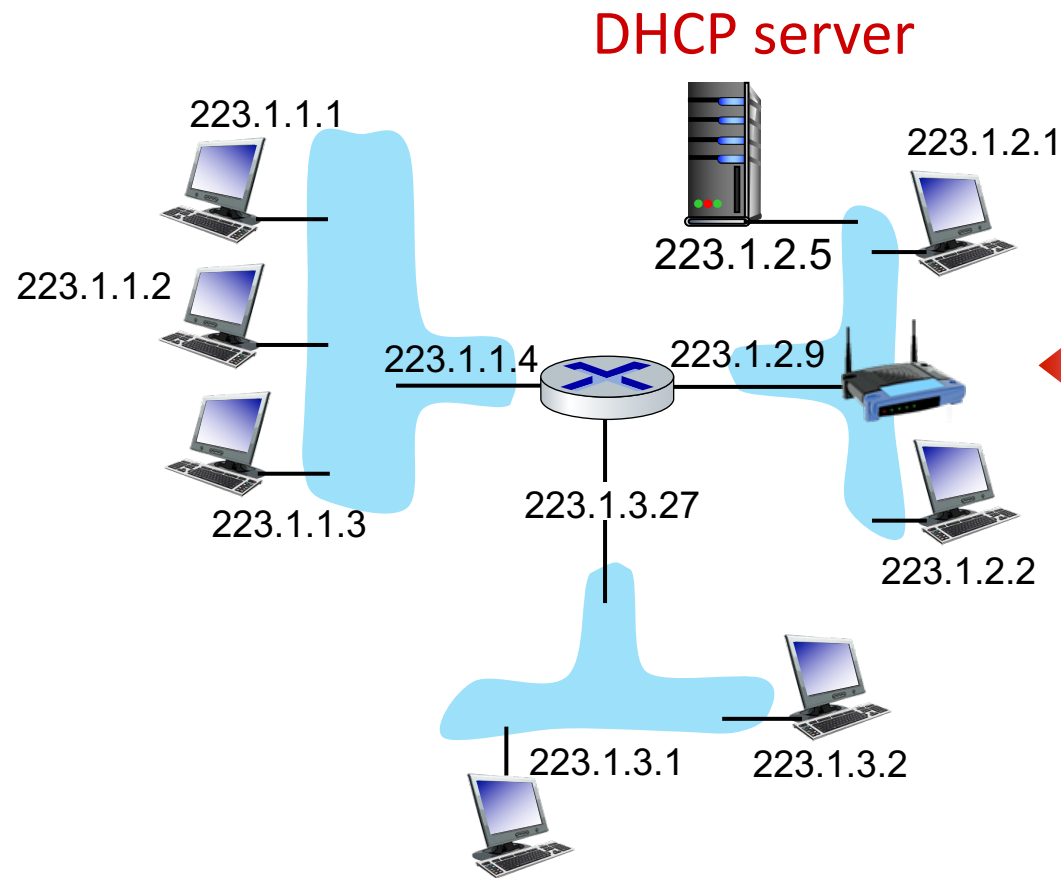
DHCP: Dynamic Host Configuration Protocol

DHCP overview:

- host broadcasts **DHCP discover** msg [optional]
- DHCP server responds with **DHCP offer** msg [optional]
- host requests IP address: **DHCP request** msg
- DHCP server sends address: **DHCP ack** msg



DHCP client-server scenario



Typically, DHCP server will be co-located in router, serving all subnets to which router is attached



arriving **DHCP client** needs address in this network

DHCP client-server scenario

DHCP server: 223.1.2.5



DHCP discover

Broadcast: is there a DHCP server out there?

Arriving client



DHCP offer

Broadcast: I'm a DHCP server! Here's an IP address you can use

DHCP request

Broadcast: OK. I would like to use this IP address!

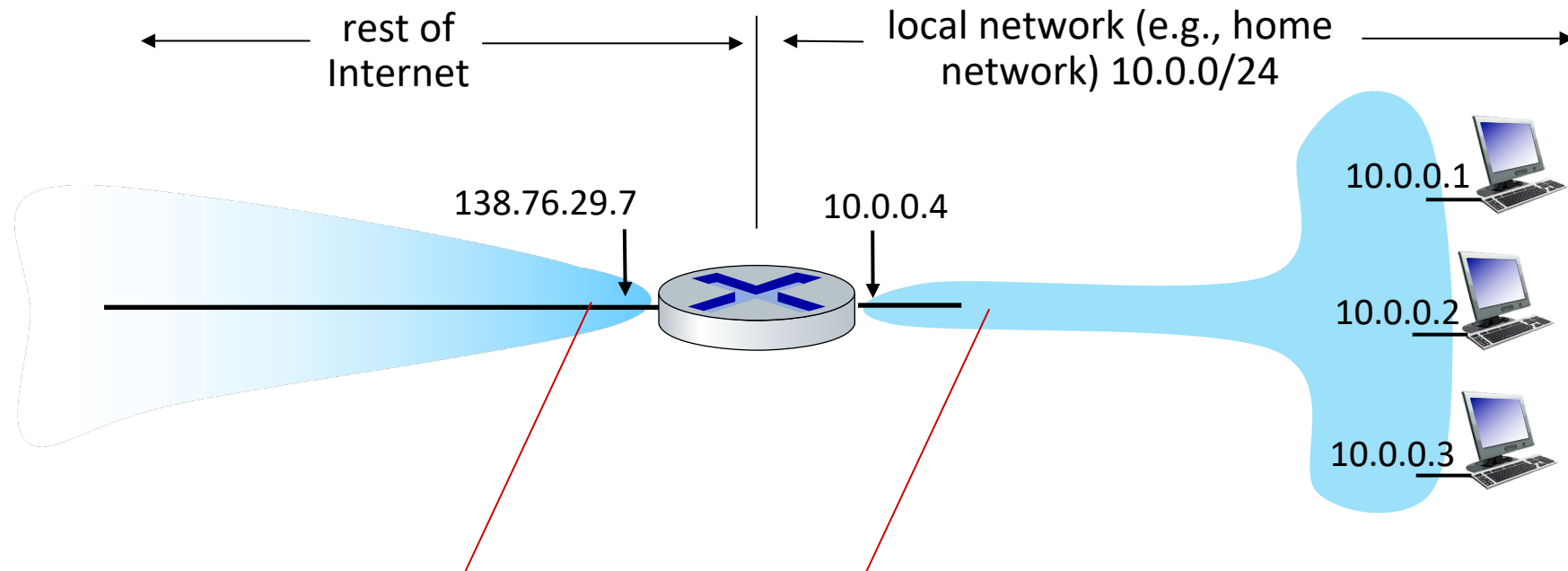
DHCP ACK

Broadcast: OK. You've got that IP address!

The two steps above can be skipped "if a client remembers and wishes to reuse a previously allocated network address" [RFC 2131]

NAT: network address translation

NAT: all devices in local network share just **one** IPv4 address as far as outside world is concerned



all datagrams *leaving* local network have *same* source NAT IP address: 138.76.29.7, but *different* source port numbers

datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)

NAT: network address translation

- all devices in local network have 32-bit addresses in a “private” IP address space that can only be used in local network:
 - 10/8
 - 172.16/12
 - 192.168/16
- advantages:
 - just **one** IP address needed from provider for *all* devices
 - can change addresses of host in local network without notifying outside world
 - security: devices inside local net not directly addressable, visible by outside world
 - ...

NAT: network address translation

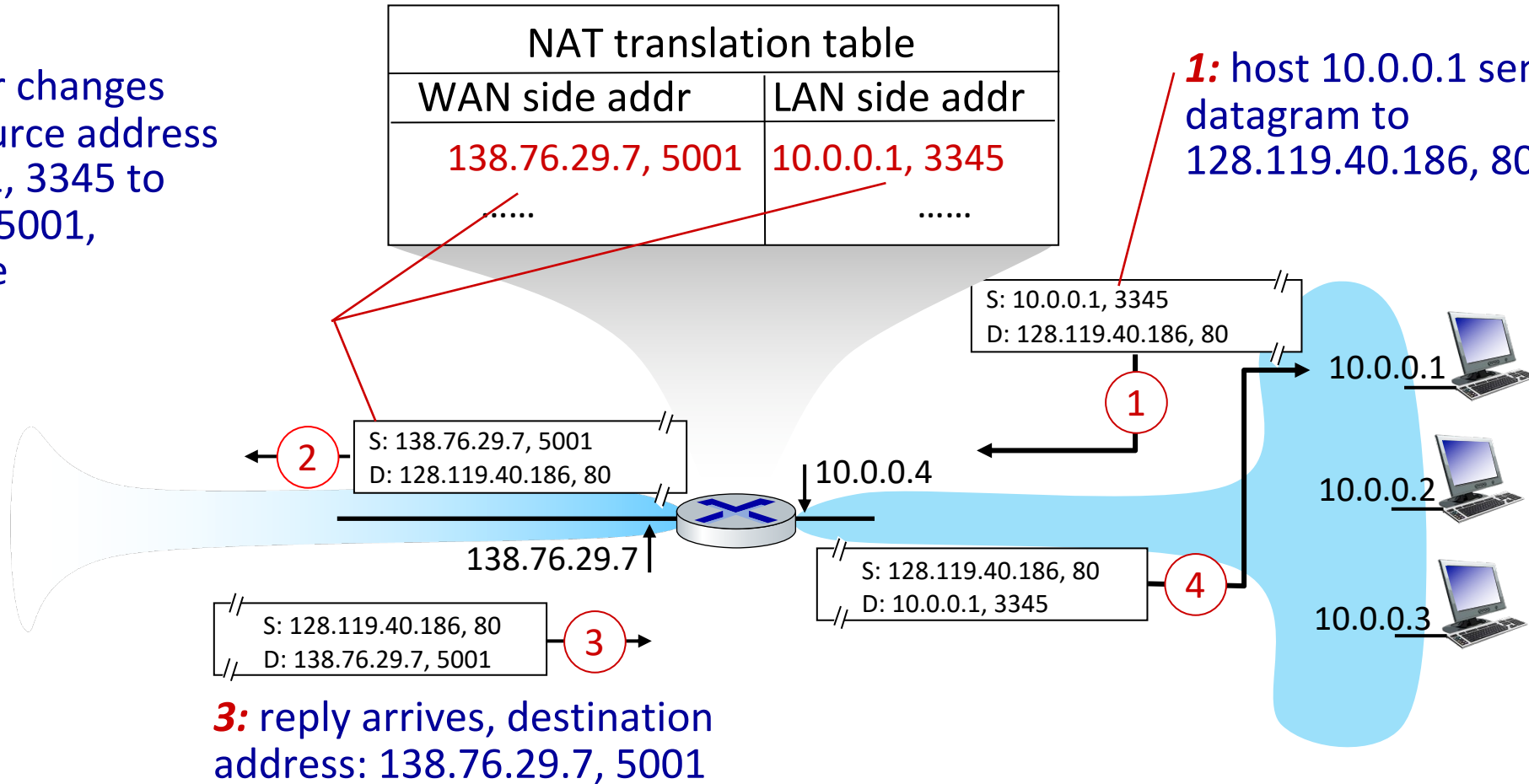
implementation: NAT router must (transparently):

- **outgoing datagrams:** replace (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
- **remember (in NAT translation table)** every (source IP address, port #) to (NAT IP address, new port #) translation pair
- **incoming datagrams:** replace (NAT IP address, new port #) in destination fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table

NAT: network address translation

2: NAT router changes datagram source address from 10.0.0.1, 3345 to 138.76.29.7, 5001, updates table

1: host 10.0.0.1 sends datagram to 128.119.40.186, 80



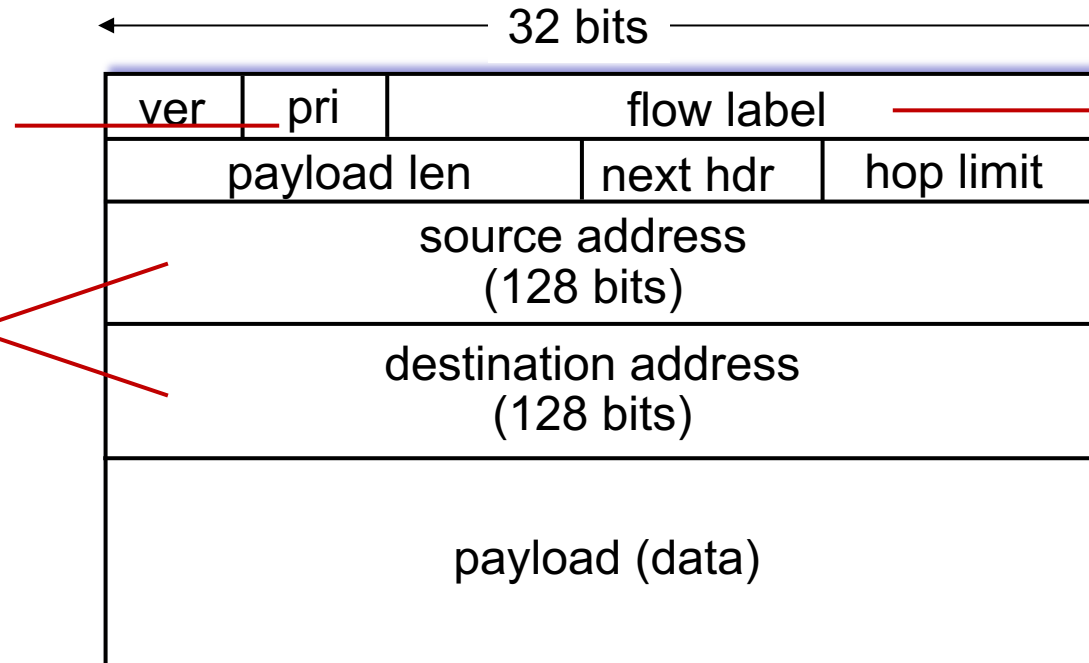
IPv6: motivation

- **initial motivation:** 32-bit IPv4 address space would be completely allocated
- additional motivation:
 - speed processing/forwarding: 40-byte fixed length header
 - enable different network-layer treatment of “flows”

IPv6 datagram format

priority: identify priority among datagrams in flow

128-bit IPv6 addresses



flow label: identify datagrams in same "flow." (concept of "flow" not well-defined).

What's missing (compared with IPv4):

- no checksum (to speed processing at routers)
- no fragmentation/reassembly
- no options (available as upper-layer, next-header protocol at router)

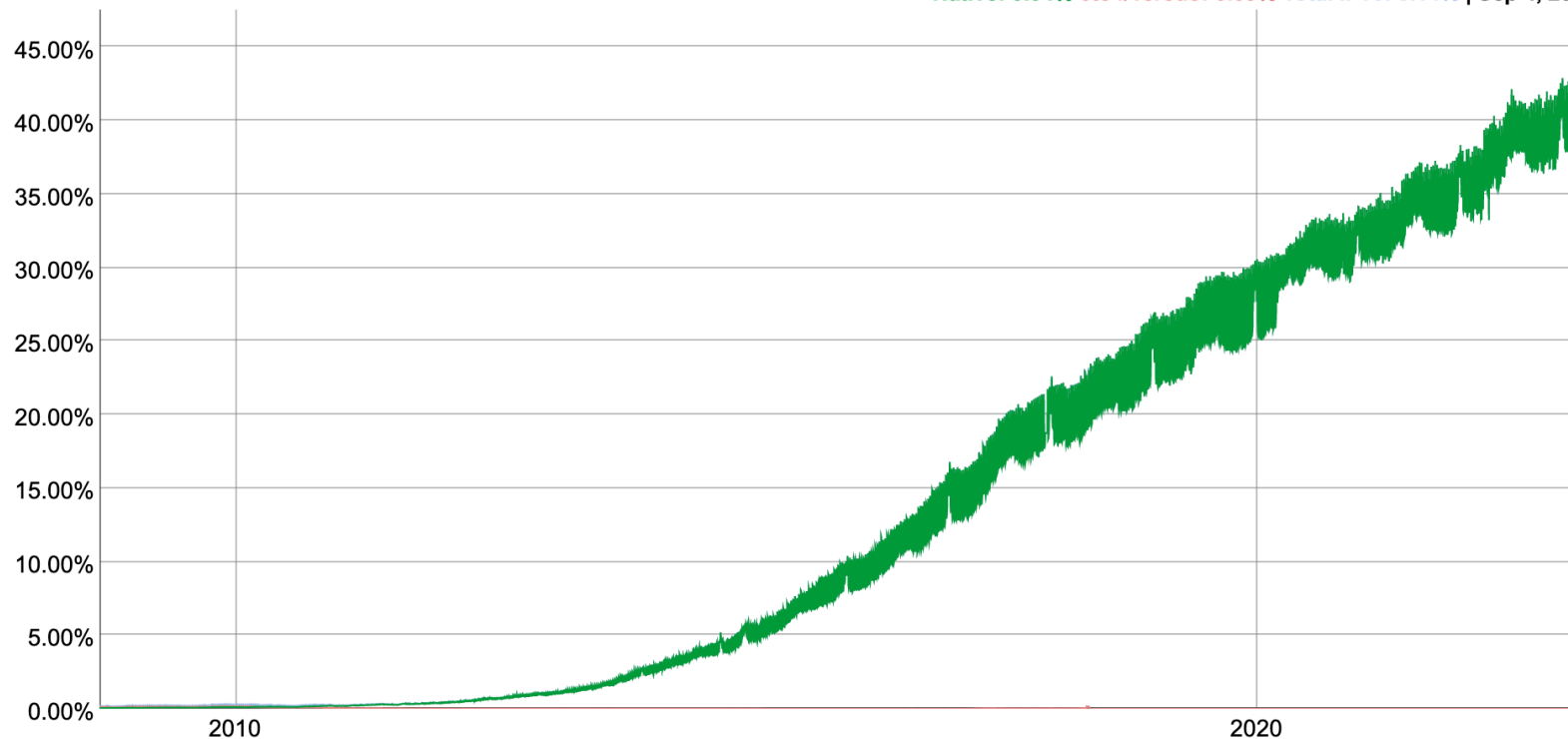
IPv6: adoption

- Google¹: ~ 40% of clients access services via IPv6 (2023)
- NIST: 1/3 of all US government domains are IPv6 capable

IPv6 Adoption

We are continuously measuring the availability of IPv6 connectivity among Google users. The graph shows the percentage of users that access Google over IPv6.

Native: 0.04% 6to4/Teredo: 0.09% Total IPv6: 0.14% | Sep 4, 2008



Network layer: Roadmap

- Network layer: overview
 - data plane
 - control plane
- What's inside a router
 - input ports, switching,
 - output ports, scheduling
- IP: the Internet Protocol
 - datagram format
 - addressing
 - network address translation
 - IPv6



- Control Plane
 - introduction
 - routing algorithm: link state
 - SDN, ICMP

Network-layer functions

- **forwarding**: move packets from router's input to appropriate router output
- **routing**: determine route taken by packets from source to destination

data plane

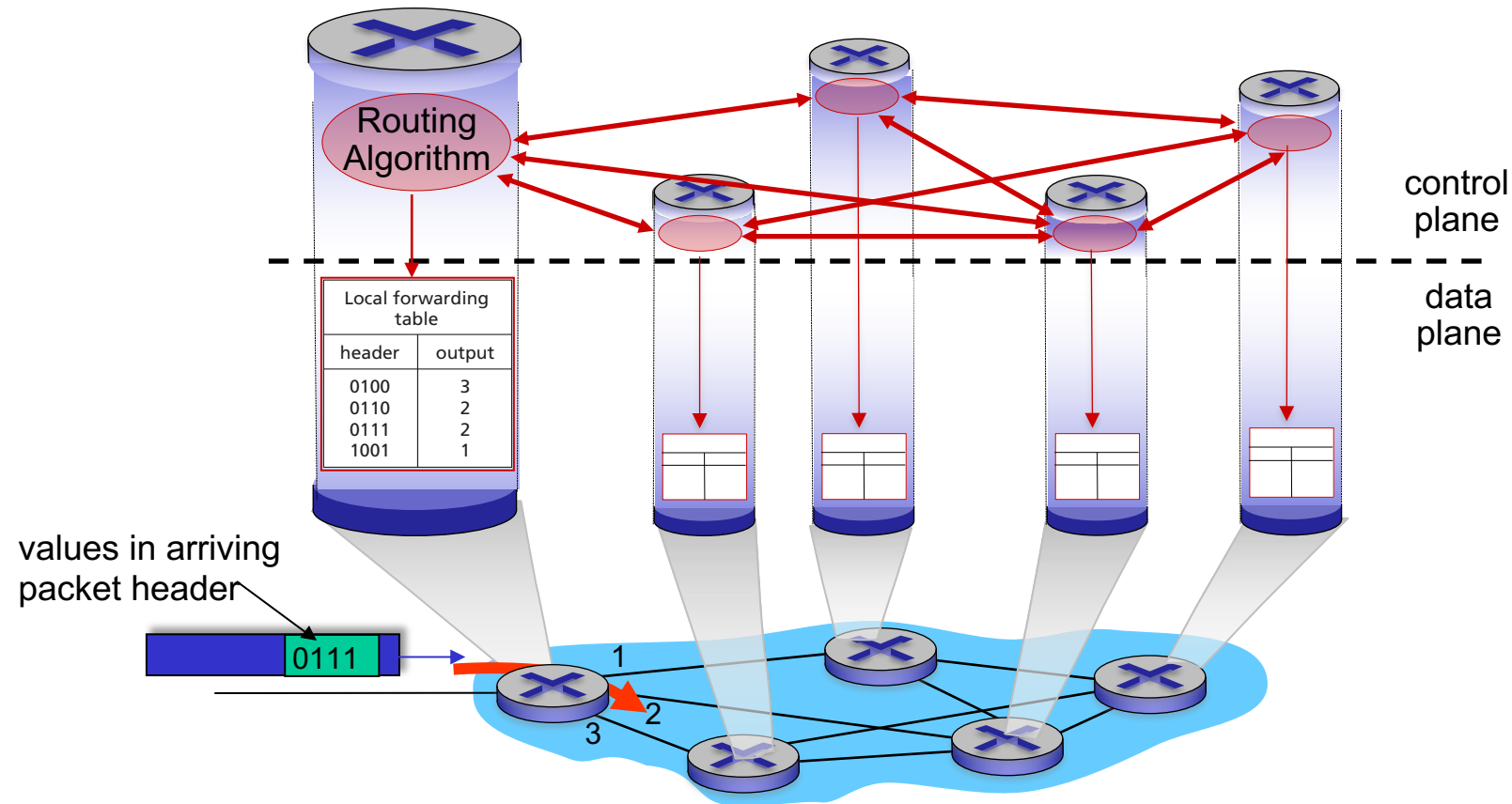
control plane

Two approaches to structuring network control plane:

- per-router control (traditional)
- logically centralized control (software defined networking)

Per-router control plane

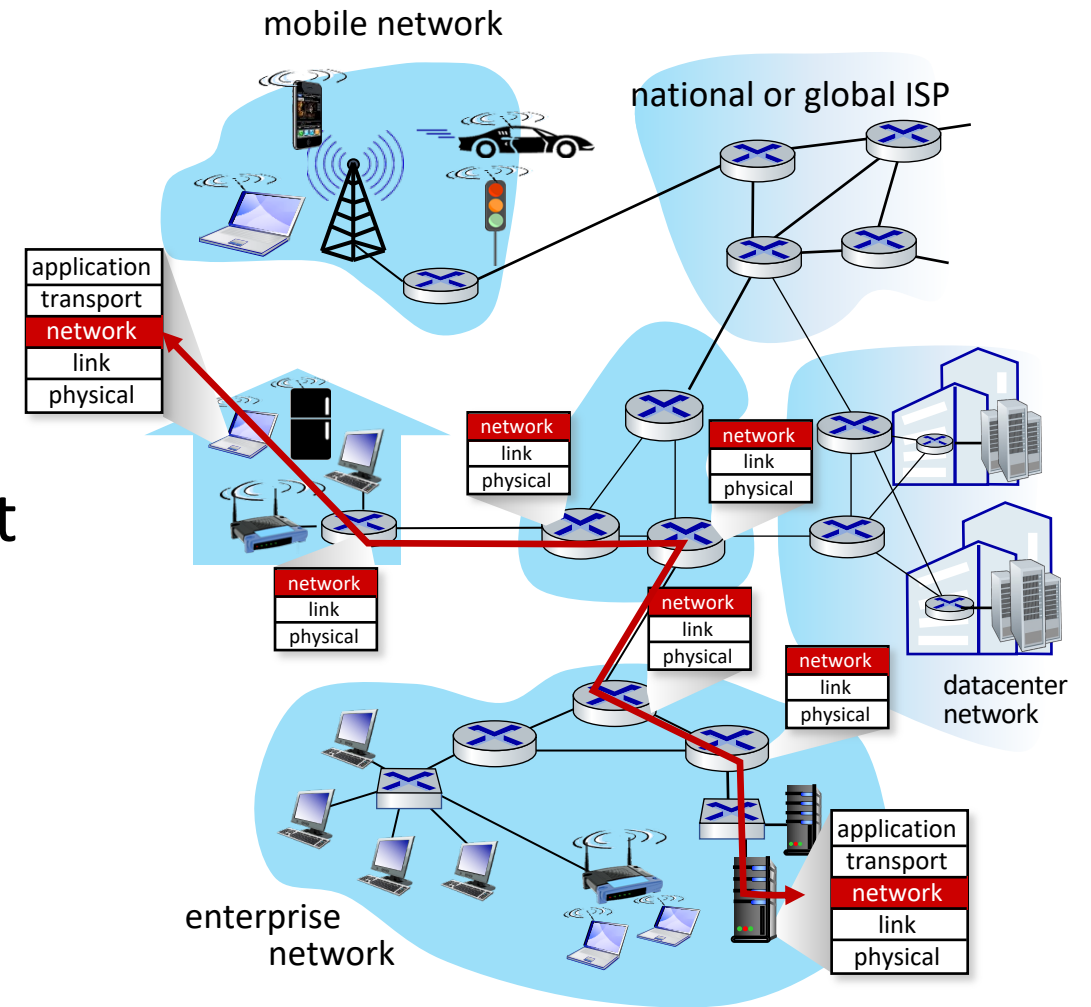
Individual routing algorithm components *in each and every router* interact in the control plane



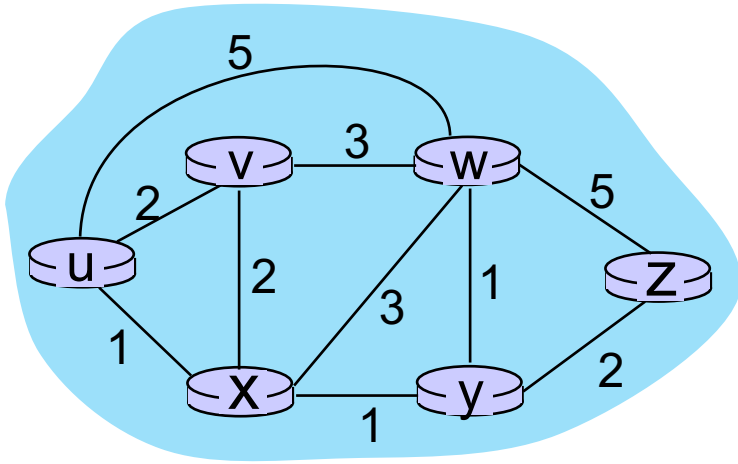
Routing protocols

Routing protocol goal: determine “good” paths (equivalently, routes), from sending hosts to receiving host, through network of routers

- **path:** sequence of routers packets traverse from given initial source host to final destination host
- **“good”:** least “cost”, “fastest”, “least congested”
- routing: a “top-10” networking challenge!



Graph abstraction: link costs



$c_{a,b}$: cost of *direct* link connecting a and b

e.g., $c_{w,z} = 5$, $c_{u,z} = \infty$

cost defined by network operator:
could always be 1, or inversely related
to bandwidth, or inversely related to
congestion

graph: $G = (N, E)$

N : set of routers = $\{ u, v, w, x, y, z \}$

E : set of links = $\{ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) \}$

Dijkstra's link-state routing algorithm

- **centralized:** network topology, link costs known to *all* nodes
 - accomplished via “link state broadcast”
 - all nodes have same info
- computes least cost paths from one node (“source”) to all other nodes
 - gives *forwarding table* for that node
- **iterative:** after k iterations, know least cost path to k destinations

notation

- $C_{x,y}$: direct link cost from node x to y ; $= \infty$ if not direct neighbors
- $D(v)$: *current* estimate of cost of least-cost-path from source to destination v
- $p(v)$: predecessor node along path from source to v
- N' : set of nodes whose least-cost-path *definitively* known

Dijkstra's link-state routing algorithm

1 *Initialization:*

2 $N' = \{u\}$ /* compute least cost path from u to all other nodes */

3 for all nodes v

4 if v adjacent to u /* u initially knows direct-path-cost only to direct neighbors */

5 then $D(v) = c_{u,v}$ /* but may not be *minimum* cost! */

6 else $D(v) = \infty$

7

8 *Loop*

9 find w not in N' such that $D(w)$ is a minimum

10 add w to N'

11 update $D(v)$ for all v adjacent to w and not in N' :

12 $D(v) = \min (D(v), D(w) + c_{w,v})$

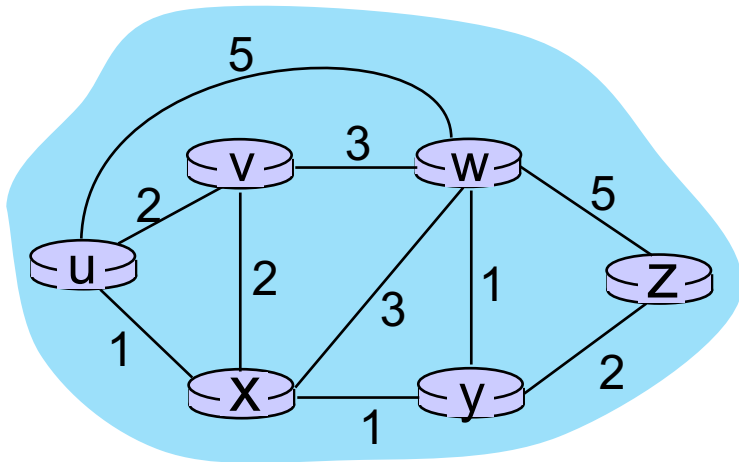
13 /* new least-path-cost to v is either old least-cost-path to v or known

14 least-cost-path to w plus direct-cost from w to v */

15 *until all nodes in N'*

Dijkstra's algorithm: an example

Step	N'	v D(v),p(v)	w D(w),p(w)	x D(x),p(x)	y D(y),p(y)	z D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1						
2						
3						
4						
5						

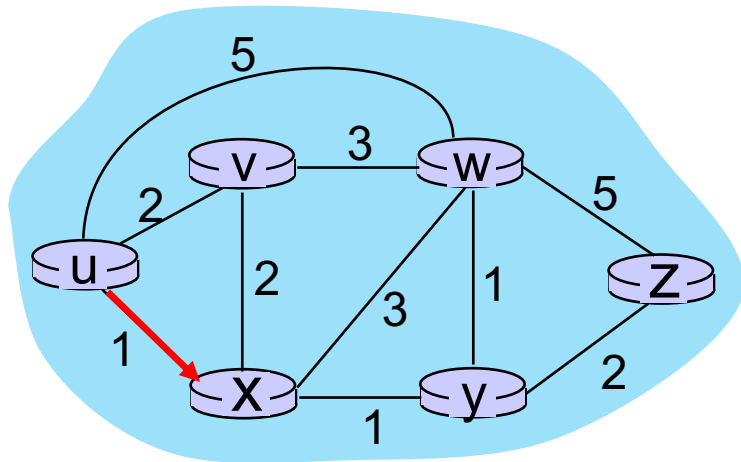


Initialization (step 0):

For all a : if a adjacent to u then $D(a) = c_{u,a}$

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux					
2						
3						
4						
5						



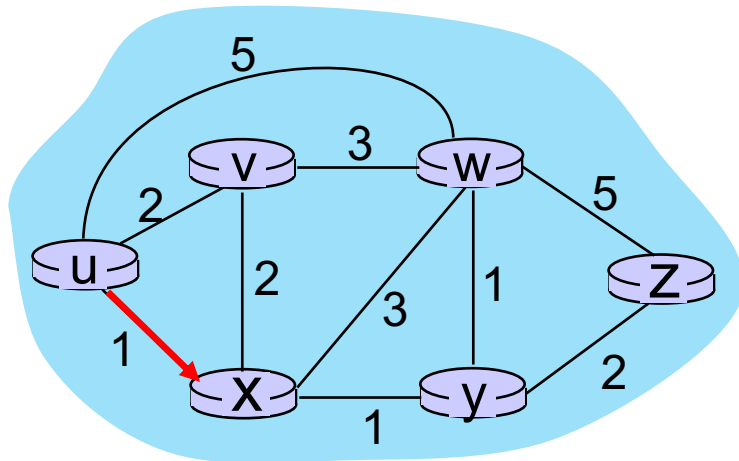
8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2						
3						
4						
5						



8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

11 update $D(b)$ for all b adjacent to a and not in N' :

$$D(b) = \min (D(b), D(a) + c_{a,b})$$

$$D(v) = \min (D(v), D(x) + c_{x,v}) = \min(2, 1+2) = 2$$

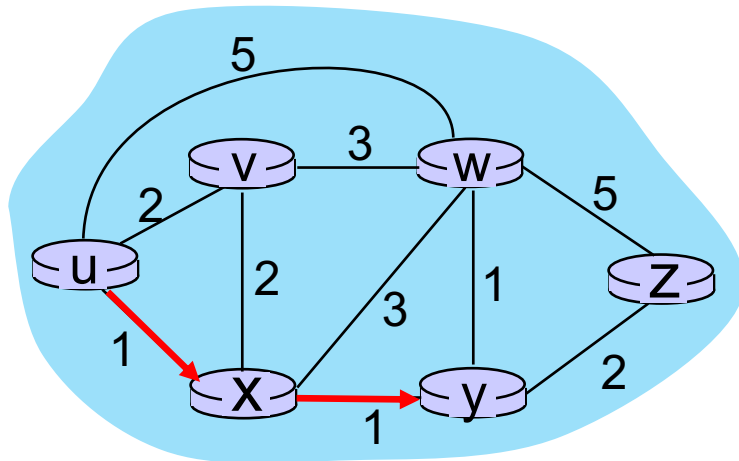
$$D(w) = \min (D(w), D(x) + c_{x,w}) = \min(5, 1+3) = 4$$

$$D(y) = \min (D(y), D(x) + c_{x,y}) = \min(\infty, 1+1) = 2$$



Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy					
3						
4						
5						



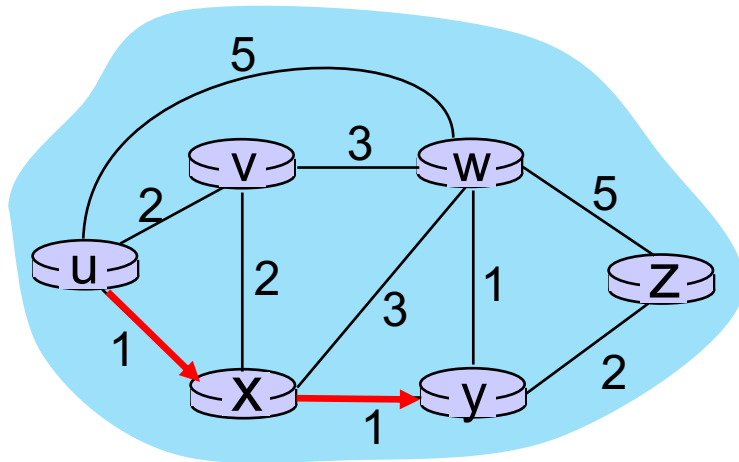
8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3						
4						
5						



8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

11 update $D(b)$ for all b adjacent to a and not in N' :

$$D(b) = \min (D(b), D(a) + c_{a,b})$$

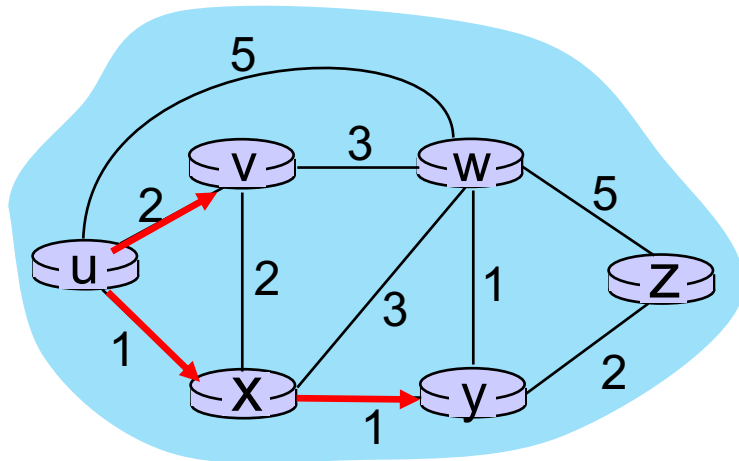
$$D(w) = \min (D(w), D(y) + c_{y,w}) = \min (4, 2+1) = 3$$

$$D(z) = \min (D(z), D(y) + c_{y,z}) = \min (\infty, 2+2) = 4$$



Dijkstra's algorithm: an example

Step	N'	$D(v), p(v)$	$D(w), p(w)$	$D(x), p(x)$	$D(y), p(y)$	$D(z), p(z)$
0	u	2, u	5, u	1, u	∞	∞
1	ux	2, u	4, x	2, x	∞	∞
2	uxy	2, u	3, y		4, y	
3	uxyv					
4						
5						



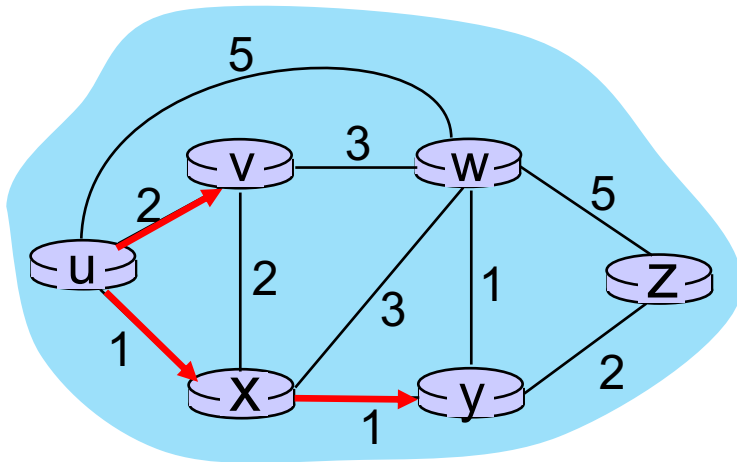
8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x	2,x	∞	∞
2	uxy	2,u	3,y		4,y	
3	uxyv		3,y		4,y	
4						
5						



8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

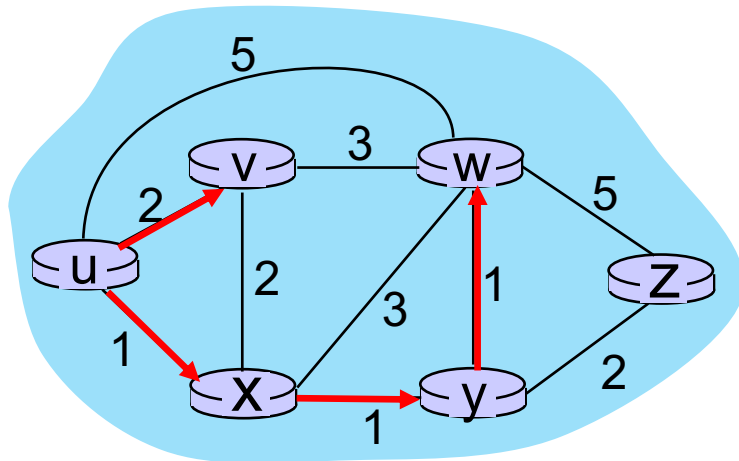
11 update $D(b)$ for all b adjacent to a and not in N' :

$$D(b) = \min (D(b), D(a) + c_{a,b})$$

$$D(w) = \min (D(w), D(v) + c_{v,w}) = \min (3, 2+3) = 3$$

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x	2,x	∞	∞
2	uxy	2,u	3,y		4,y	
3	uxyv		3,y		4,y	
4	uxyvw					
5						



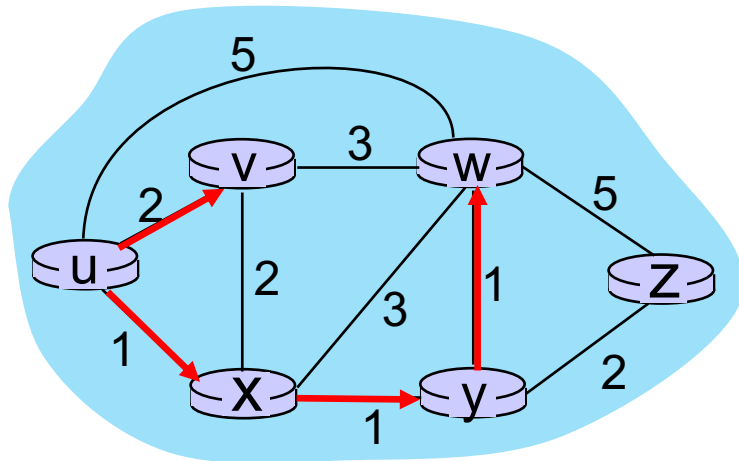
8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x	2,x	∞	∞
2	uxy	2,u	3,y		4,y	
3	uxyv		3,y		4,y	
4	uxyvw				4,y	
5						



8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

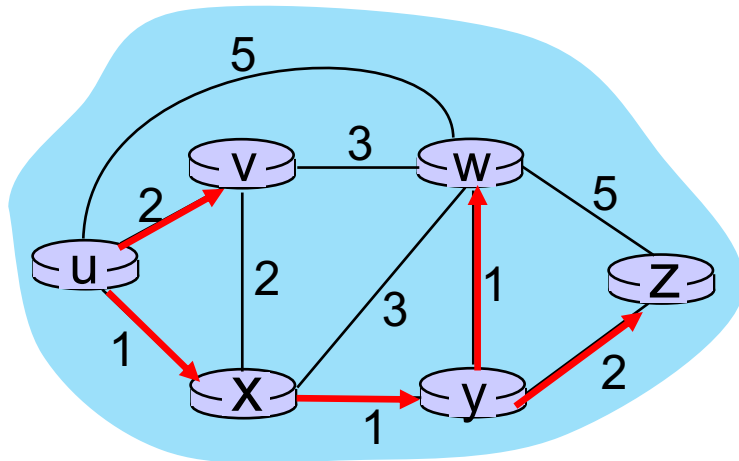
11 update $D(b)$ for all b adjacent to a and not in N' :

$$D(b) = \min (D(b), D(a) + c_{a,b})$$

$$D(z) = \min (D(z), D(w) + c_{w,z}) = \min (4, 3+5) = 4$$

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x	2,x	∞	∞
2	uxy	2,u	3,y	3,y	4,y	∞
3	uxyv	2,u	3,y	3,y	4,y	4,y
4	uxyvw	2,u	3,y	3,y	4,y	4,y
5	uxyvwz	2,u	3,y	3,y	4,y	4,y



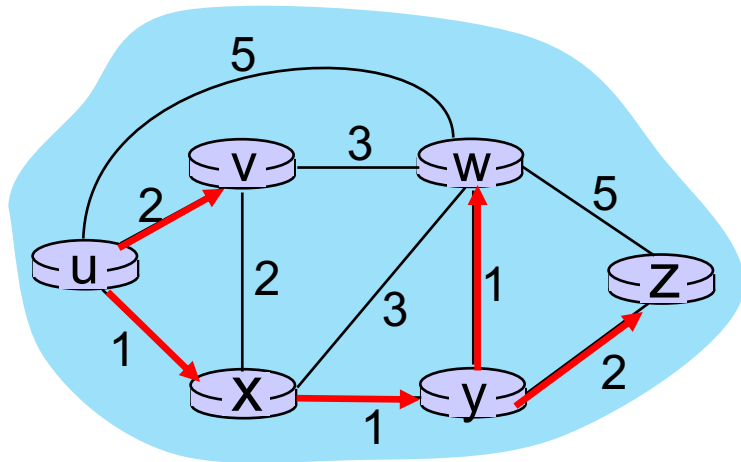
8 Loop

9 find a not in N' such that $D(a)$ is a minimum

10 add a to N'

Dijkstra's algorithm: an example

Step	N'	v $D(v),p(v)$	w $D(w),p(w)$	x $D(x),p(x)$	y $D(y),p(y)$	z $D(z),p(z)$
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x	2,x	∞	∞
2	uxy	2,u	3,y	3,y	4,y	∞
3	uxyv	2,u	3,y	3,y	4,y	4,y
4	uxyvw	2,u	3,y	3,y	4,y	4,y
5	uxyvwz	2,u	3,y	3,y	4,y	4,y

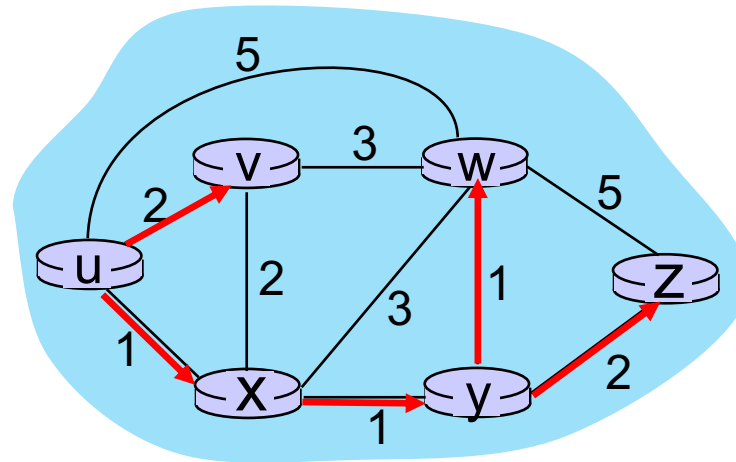


8 Loop

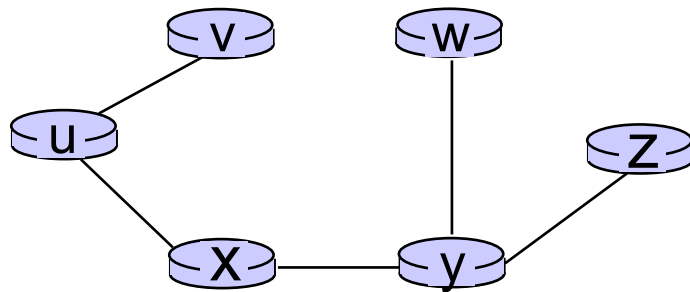
- 9 find a not in N' such that $D(a)$ is a minimum
- 10 add a to N'
- 11 update $D(b)$ for all b adjacent to a and not in N' :

$$D(b) = \min (D(b), D(a) + c_{a,b})$$

Dijkstra's algorithm: an example



resulting least-cost-path tree from u:



resulting forwarding table in u:

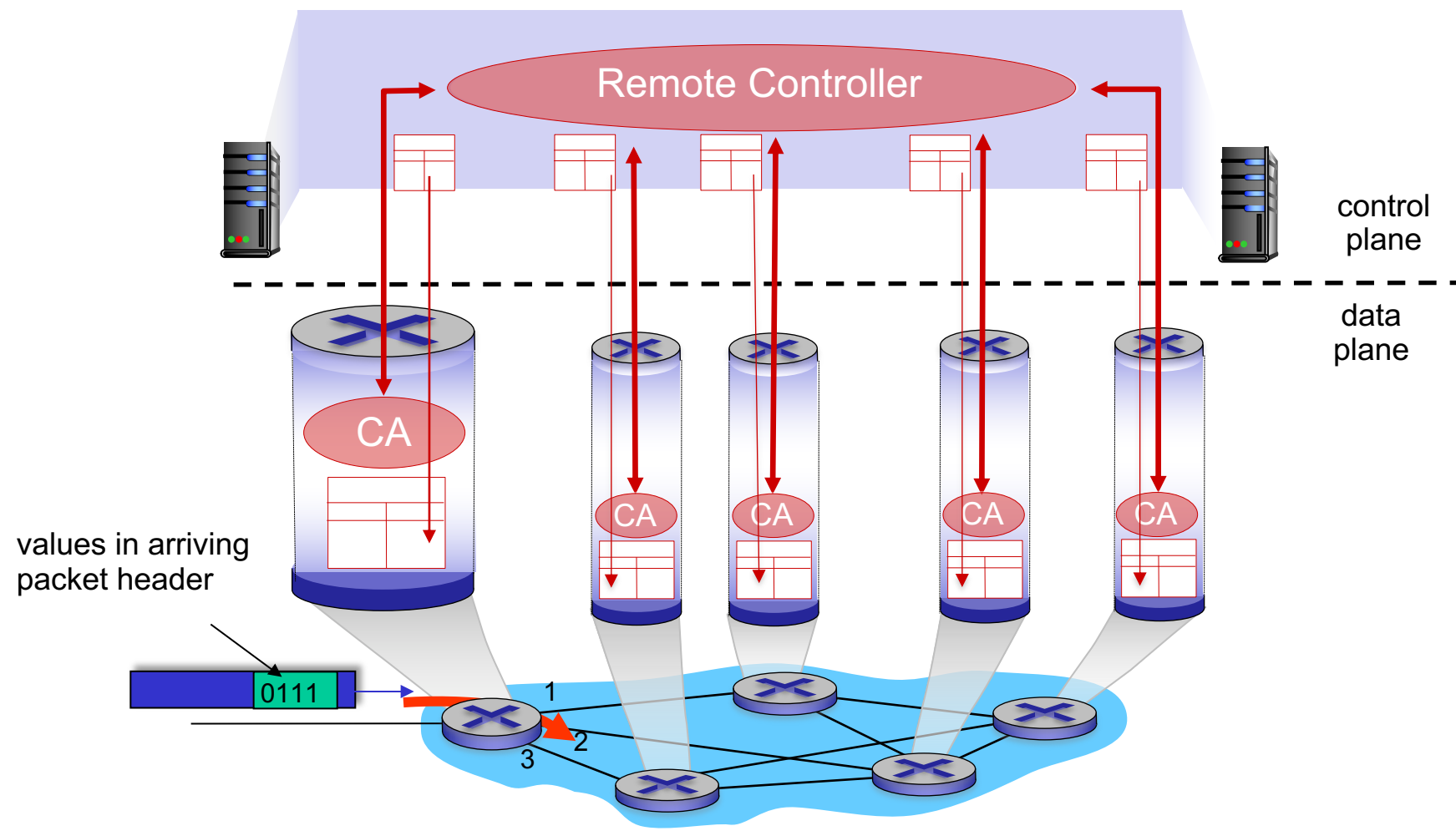
destination	outgoing link
v	(u,v)
x	(u,x)
y	(u,x)
w	(u,x)
x	(u,x)

route from u to v directly

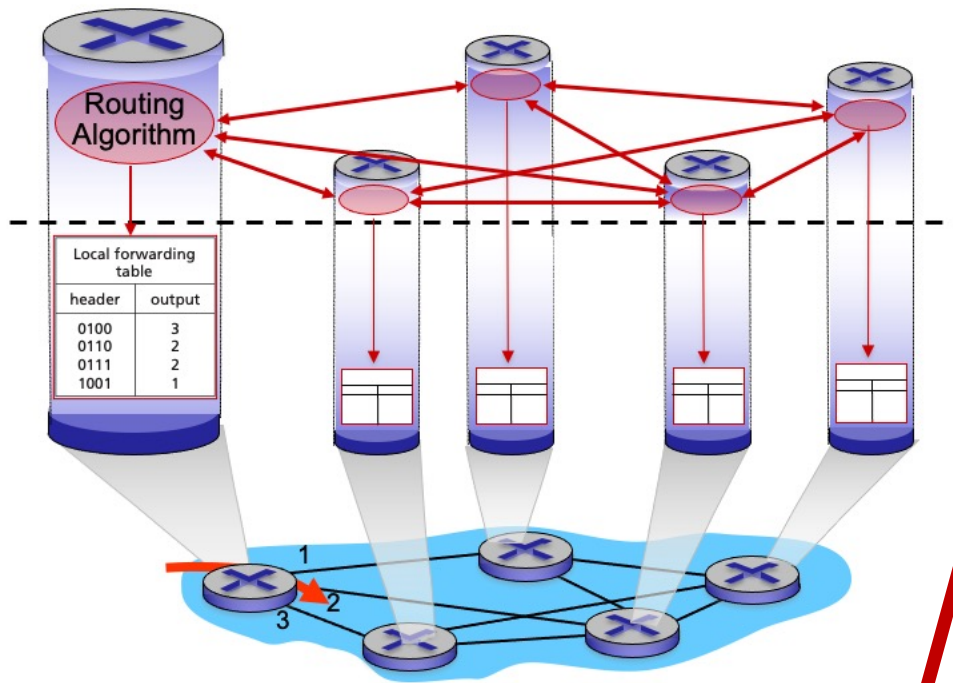
route from u to all other destinations via x

Software-Defined Networking (SDN)

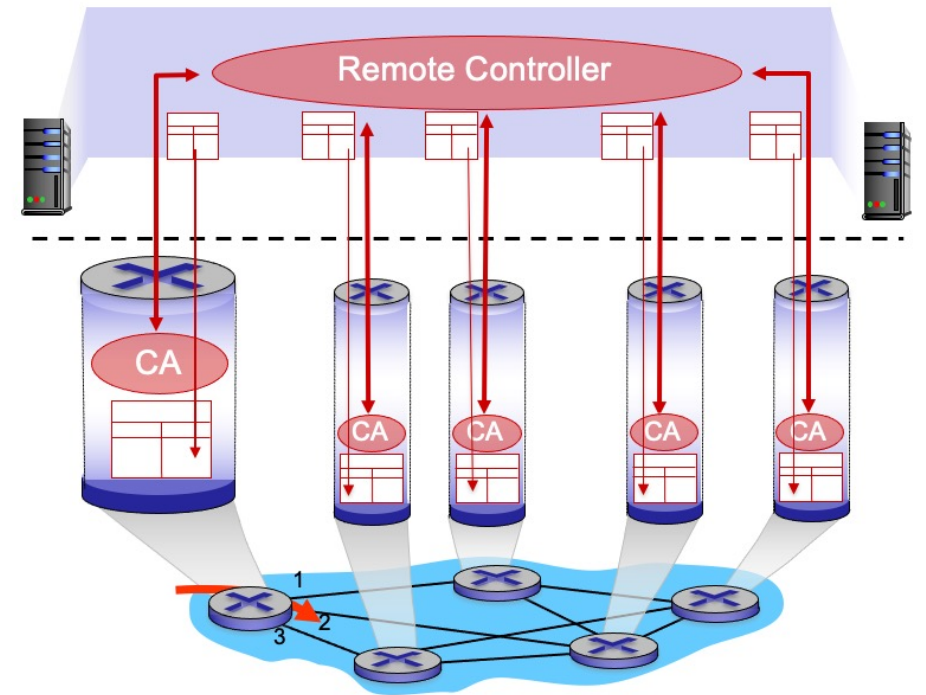
Remote controller computes, installs forwarding tables in routers



Per-router control plane



SDN control plane



SDN and the future of traditional network protocols

- SDN-computed versus router-computer forwarding tables:
 - just one example of logically-centralized-computed versus protocol computed
- one could imagine SDN-computed congestion control:
 - controller sets sender rates based on router-reported (to controller) congestion levels



How will implementation of network functionality (SDN versus protocols) evolve?



ICMP: internet control message protocol

- used by hosts and routers to communicate network-level information
 - error reporting: unreachable host, network, port, protocol
 - echo request/reply (used by ping)
- network-layer “above” IP:
 - ICMP messages carried in IP datagrams
- *ICMP message*: type, code plus first 8 bytes of IP datagram causing error

<u>Type</u>	<u>Code</u>	<u>description</u>
0	0	echo reply (ping)
3	0	dest. network unreachable
3	1	dest host unreachable
3	2	dest protocol unreachable
3	3	dest port unreachable
3	6	dest network unknown
3	7	dest host unknown
4	0	source quench (congestion control - not used)
8	0	echo request (ping)
9	0	route advertisement
10	0	router discovery
11	0	TTL expired
12	0	bad IP header

Network layer: Summary

- Overview of principles behind the **data plane**:
 - forwarding versus routing
 - how a router works
 - Addressing
 - DHCP, NAT, IPv6
- Overview of principles behind the **control plane**:
 - routing algorithm: link state
 - software-defined networking
 - ICMP

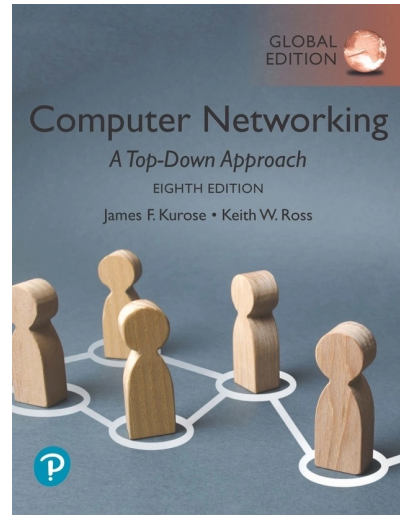
Computer Systems

Networking: Link Layer

Amirali AMIRI

13.06.2024

Sources



- Literature: “Computer Networking: A Top-Down Approach” Written by James F. Kurose and Keith W. Ross
 - https://gaia.cs.umass.edu/kurose_ross/index.php (includes resources for students).
 - They also provide slideshows – the basis for ours! You can investigate the extended version at their website.
 - Also available at the TU library!

Link layer and LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
 - addressing, ARP
 - Ethernet
 - Switches
- Summary

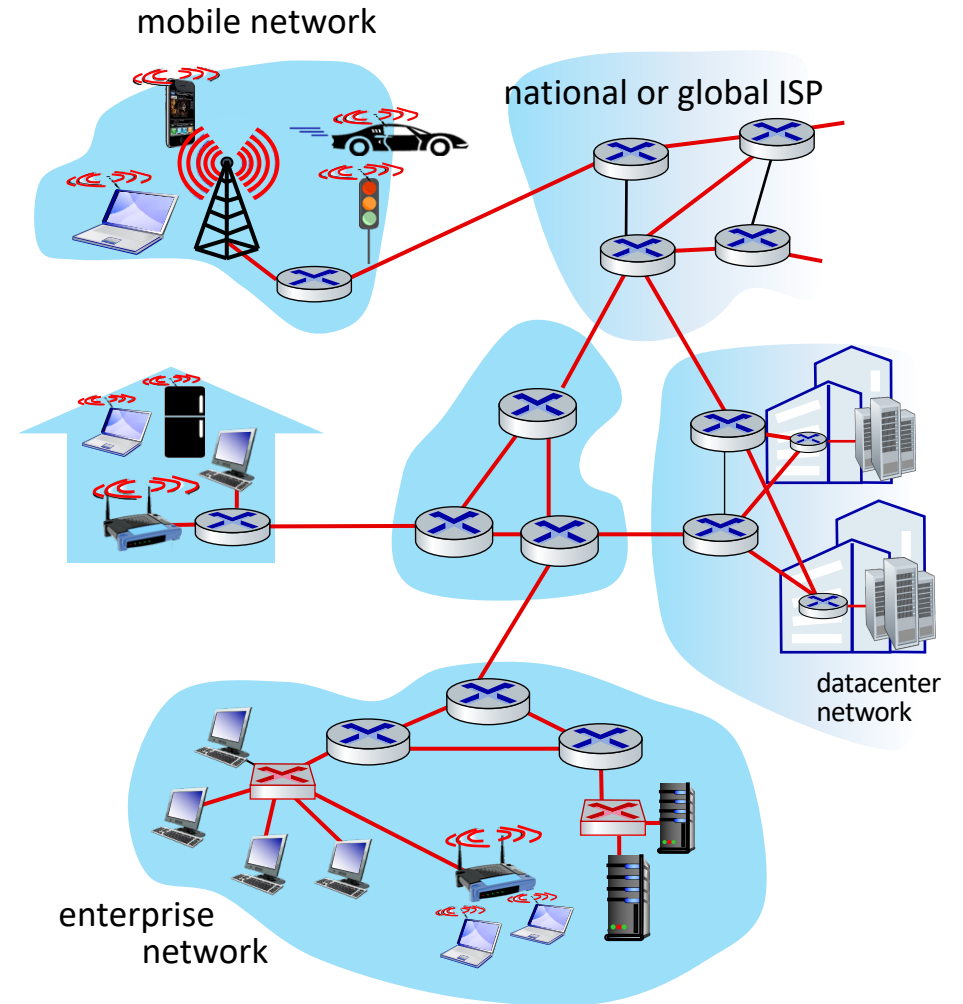


Link layer: introduction

terminology:

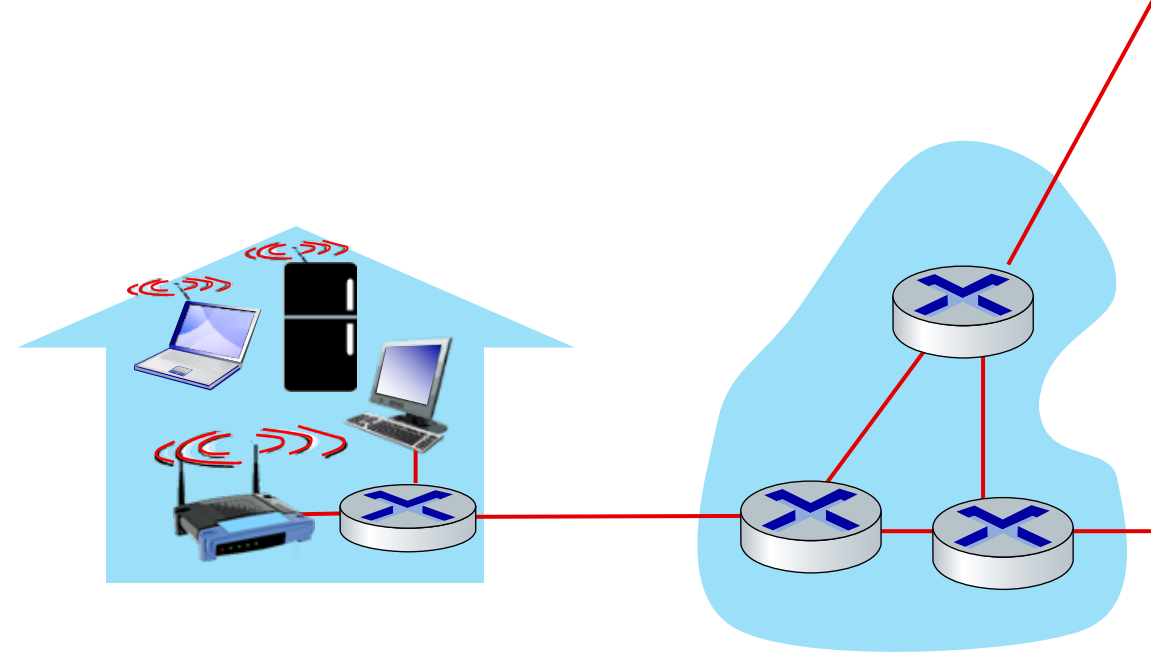
- hosts, routers: **nodes**
- communication channels that connect **adjacent** nodes along communication path: **links**
 - wired , wireless
 - LANs
- layer-2 packet: **frame**, encapsulates datagram

link layer has responsibility of transferring datagram from one node to **physically adjacent** node over a link

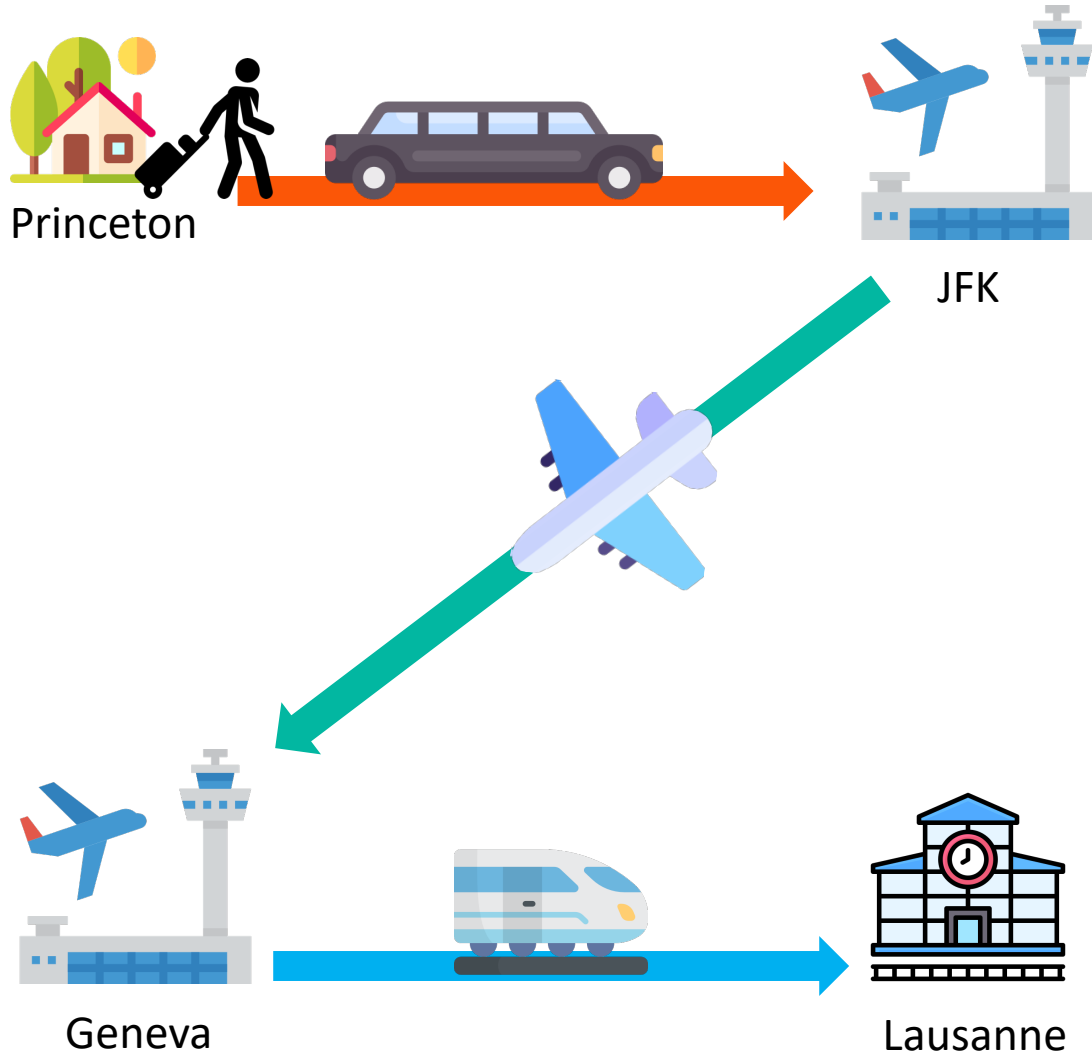


Link layer: context

- datagram transferred by **different link protocols** over different links:
 - e.g., WiFi on first link, Ethernet on next link
- each link protocol provides different services
 - e.g., **may or may not** provide reliable data transfer over link



Transportation analogy



transportation analogy:

- trip from Princeton to Lausanne
 - limo: Princeton to JFK
 - plane: JFK to Geneva
 - train: Geneva to Lausanne
- tourist = datagram
- transport segment = communication link
- transportation mode = link-layer protocol
- travel agent = routing algorithm

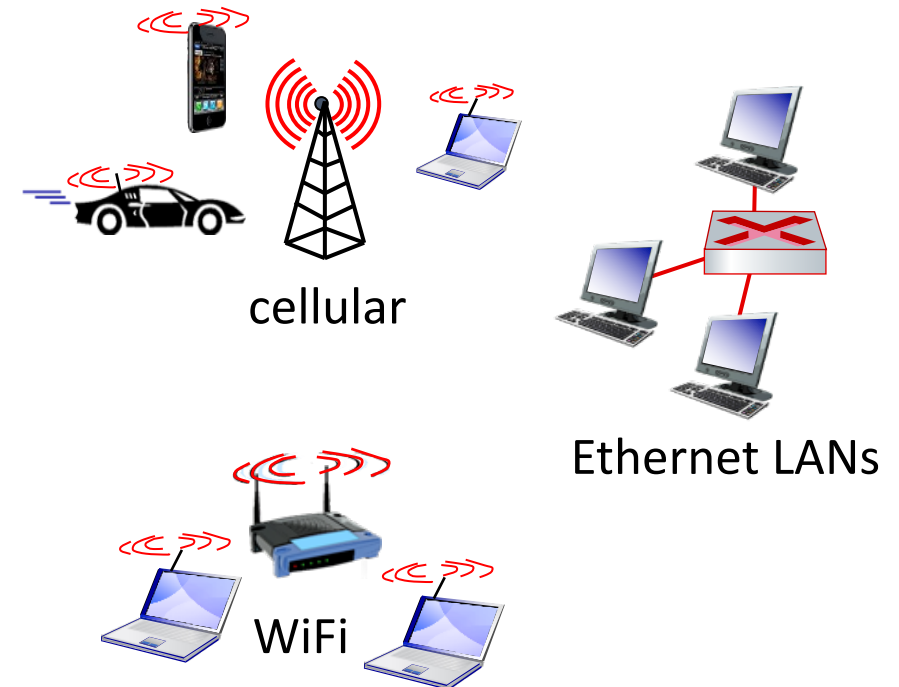
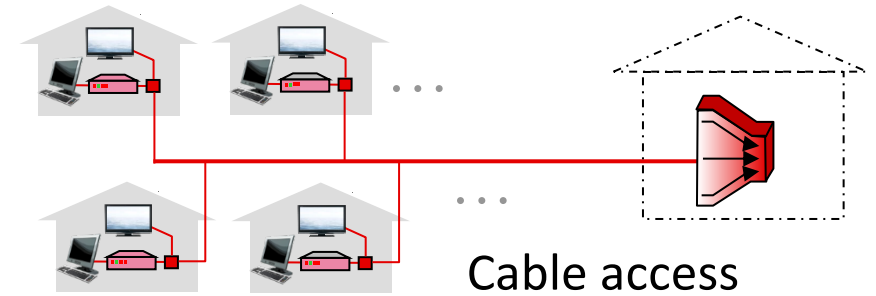
Link layer: services

■ framing, link access:

- encapsulate datagram into frame, adding header, trailer
- channel access if shared medium
- “MAC” addresses in frame headers identify source, destination (different from IP address!)

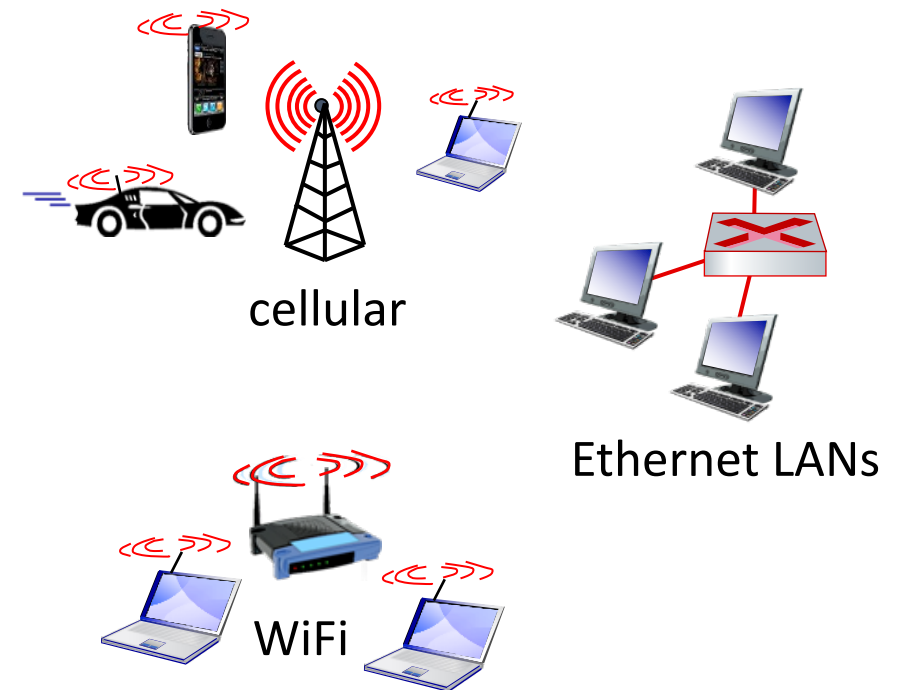
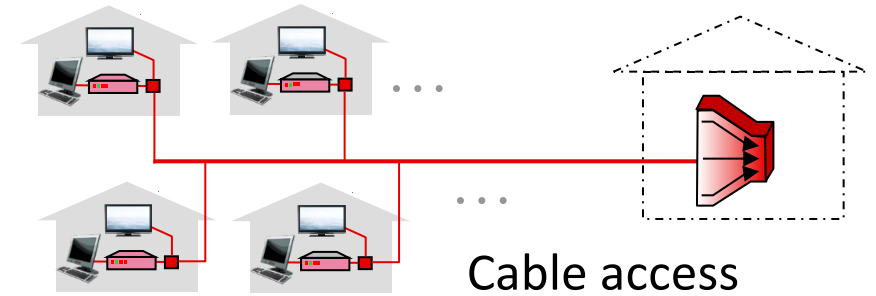
■ reliable delivery between adjacent nodes

- we already know how to do this!
- seldom used on low bit-error links
- wireless links: high error rates



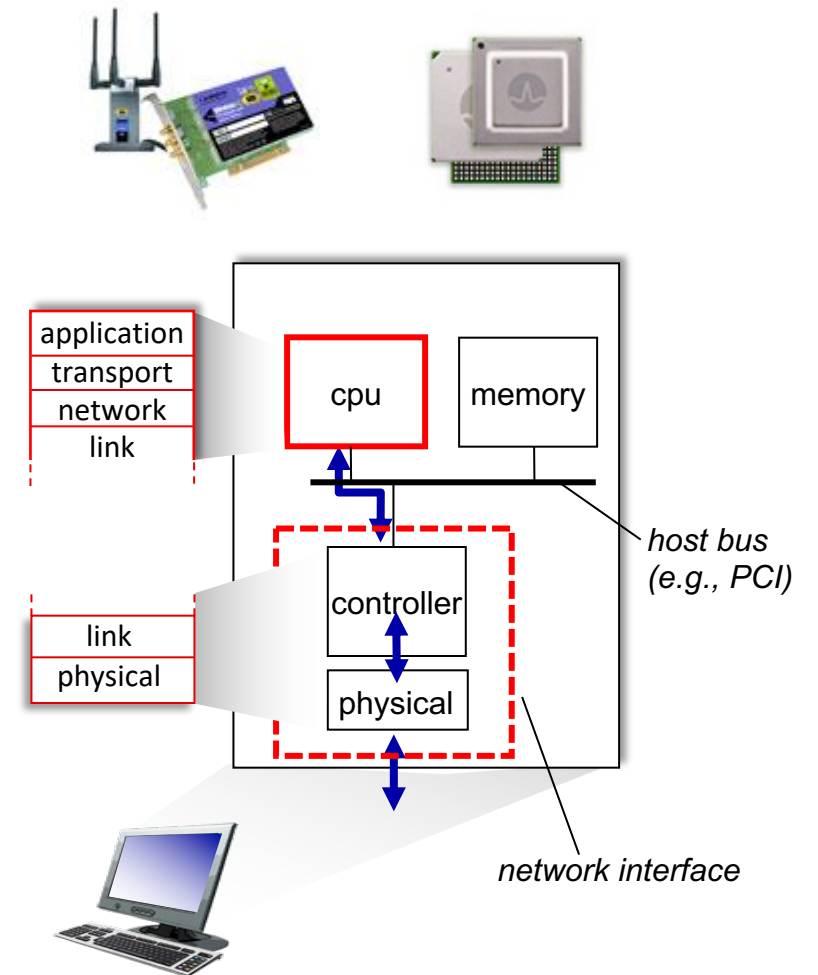
Link layer: services

- **flow control:**
 - pacing between adjacent sending and receiving nodes
- **error detection:**
 - errors caused by signal attenuation, noise.
 - receiver detects errors, signals retransmission, or drops frame
- **error correction:**
 - receiver identifies *and corrects* bit error(s) without retransmission
- **half-duplex and full-duplex:**
 - with half duplex, nodes at both ends of link can transmit, but not at same time



Host link-layer implementation

- in each-and-every host
- link layer implemented on-chip or in network interface card (NIC)
 - implements link, physical layer
- attaches into host's system buses
- combination of hardware, software, firmware



Link layer and LANs: roadmap

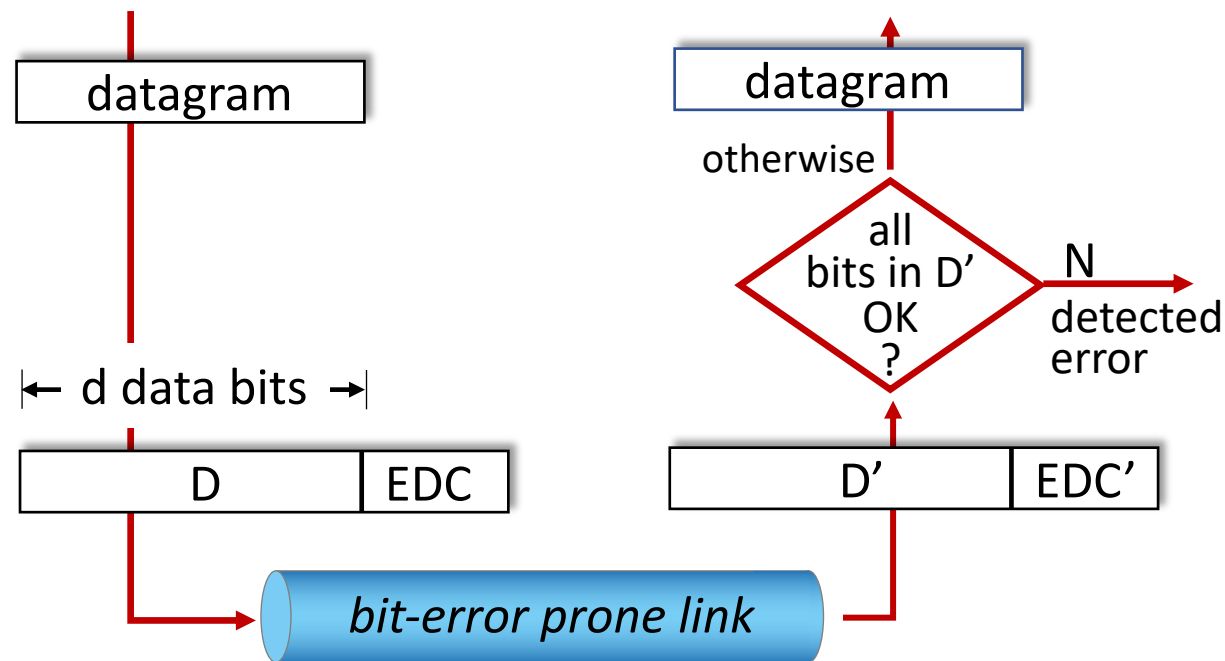
- introduction
- error detection, correction
- multiple access protocols
- LANs
 - addressing, ARP
 - Ethernet
 - switches
- Summary



Error detection

EDC: error detection and correction bits (e.g., redundancy)

D: data protected by error checking, may include header fields



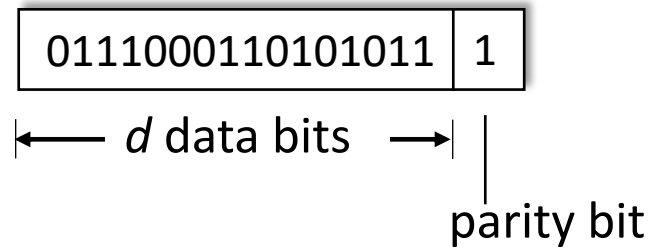
Error detection not 100% reliable!

- protocol may miss some errors, but rarely
- larger EDC field yields better detection and correction

Parity checking

single bit parity:

- detect single bit errors



Even/odd parity: set parity bit so there is an even/odd number of 1's

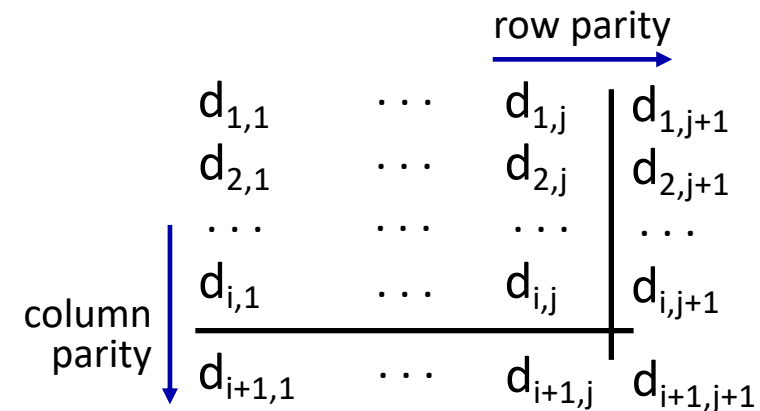
At receiver:

- compute parity of d received bits
- compare with received parity bit – if different than error detected



Can detect *and* correct errors (without retransmission!)

- two-dimensional parity: detect *and correct* single bit errors



no errors:

1	0	1	0	1	1
1	1	1	1	0	0
0	1	1	1	0	1
0	0	1	0	1	0

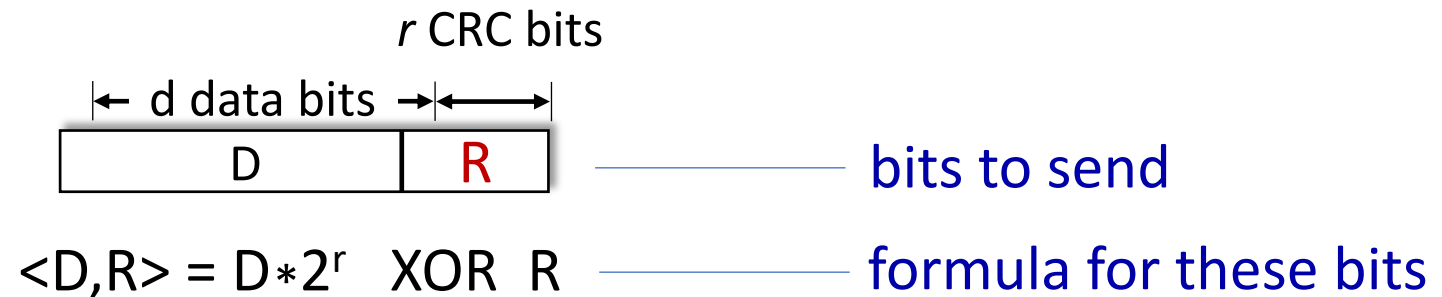
detected and correctable single-bit error:

1	0	1	0	1	1
1	0	1	1	0	0
0	1	1	1	0	1
0	0	1	0	1	0

\rightarrow parity error
 \downarrow parity error

Cyclic Redundancy Check (CRC)

- more powerful error-detection coding
- **D**: data bits (given, think of these as a binary number)
- **G**: bit pattern (generator), of $r+1$ bits (given, specified in CRC standard)

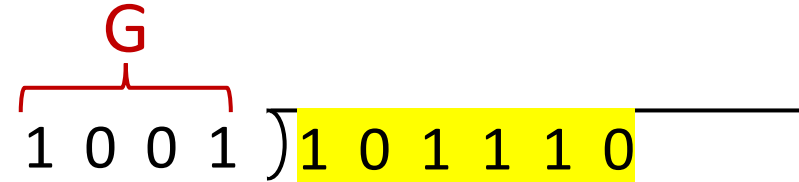


- sender*: compute r CRC bits, **R**, such that $\langle D, R \rangle$ *exactly* divisible by $G \pmod{2}$
- receiver knows G , divides $\langle D, R \rangle$ by G . If non-zero remainder: error detected!
 - can detect all burst errors less than $r+1$ bits
 - widely used in practice (Ethernet, 802.11 WiFi)

Cyclic Redundancy Check (CRC): example

Sender wants to compute R
such that:

$$D \cdot 2^r \text{ XOR } R = nG$$



... or equivalently (XOR R both sides):

$$D \cdot 2^r = nG \text{ XOR } R$$

... which says:

if we divide $D \cdot 2^r$ by G , we
want remainder R to satisfy:

$$R = \text{remainder} \left[\frac{D \cdot 2^r}{G} \right] \text{ algorithm for computing } R$$

Cyclic Redundancy Check (CRC): example

Sender wants to compute R
such that:

$$D \cdot 2^r \text{ XOR } R = nG$$

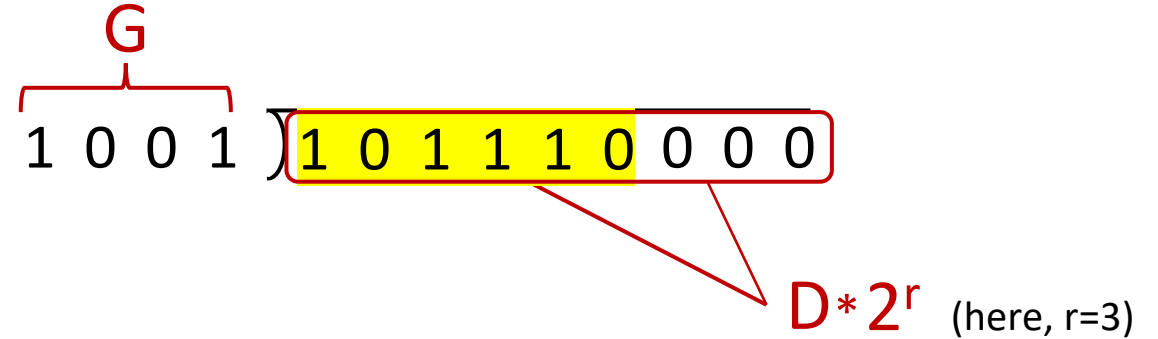
... or equivalently (XOR R both sides):

$$D \cdot 2^r = nG \text{ XOR } R$$

... which says:

if we divide $D \cdot 2^r$ by G, we
want remainder R to satisfy:

$$R = \text{remainder} \left[\frac{D \cdot 2^r}{G} \right] \text{ algorithm for computing } R$$



Cyclic Redundancy Check (CRC): example

Sender wants to compute R
such that:

$$D \cdot 2^r \text{ XOR } R = nG$$

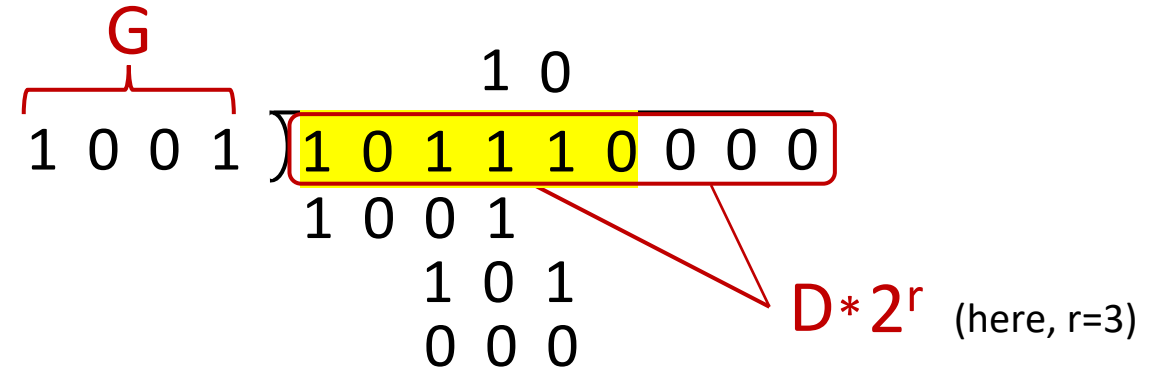
... or equivalently (XOR R both sides):

$$D \cdot 2^r = nG \text{ XOR } R$$

... which says:

if we divide $D \cdot 2^r$ by G, we
want remainder R to satisfy:

$$R = \text{remainder} \left[\frac{D \cdot 2^r}{G} \right] \text{ algorithm for computing } R$$



Cyclic Redundancy Check (CRC): example

Sender wants to compute R
such that:

$$D \cdot 2^r \text{ XOR } R = nG$$

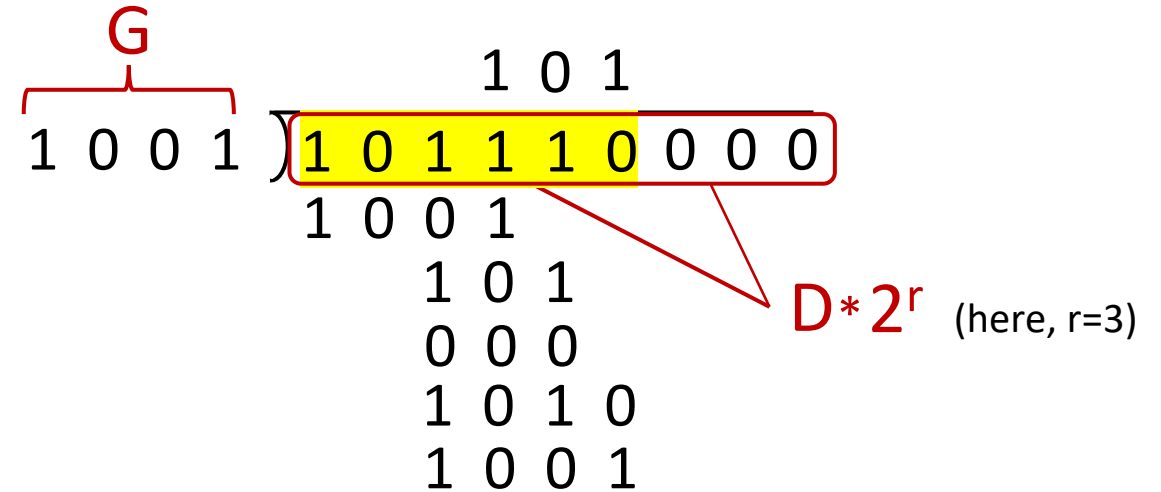
... or equivalently (XOR R both sides):

$$D \cdot 2^r = nG \text{ XOR } R$$

... which says:

if we divide $D \cdot 2^r$ by G, we
want remainder R to satisfy:

$$R = \text{remainder} \left[\frac{D \cdot 2^r}{G} \right] \text{ algorithm for computing } R$$



Cyclic Redundancy Check (CRC): example

Sender wants to compute R
such that:

$$D \cdot 2^r \text{ XOR } R = nG$$

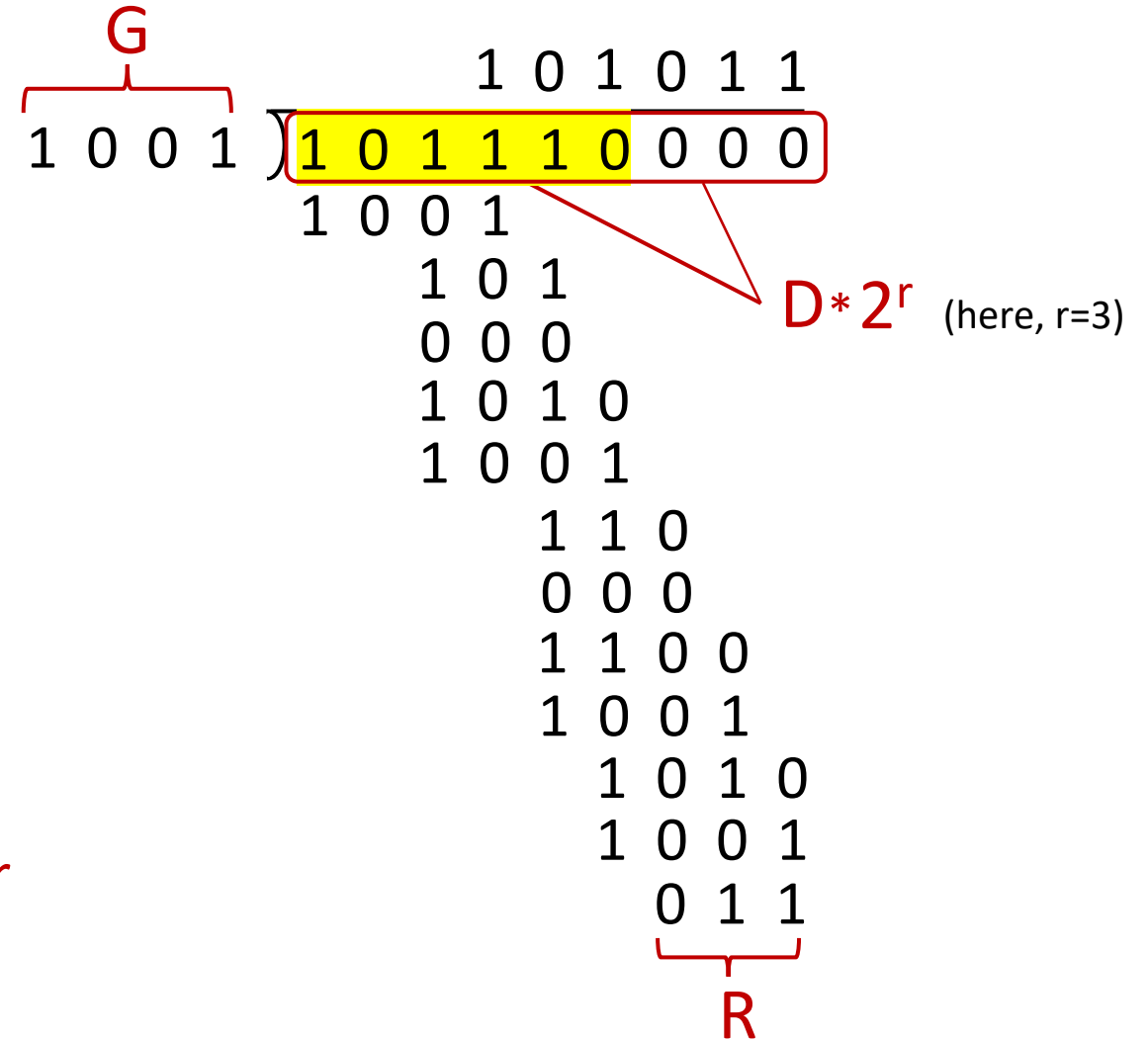
... or equivalently (XOR R both sides):

$$D \cdot 2^r = nG \text{ XOR } R$$

... which says:

if we divide $D \cdot 2^r$ by G, we
want remainder R to satisfy:

$$R = \text{remainder} \left[\frac{D \cdot 2^r}{G} \right] \text{ algorithm for computing } R$$



Link layer and LANs: roadmap

- introduction
- error detection, correction
- **multiple access protocols**
- LANs
 - addressing, ARP
 - Ethernet
 - switches
- Summary



Multiple access links, protocols

two types of “links”:

- point-to-point
 - point-to-point link between Ethernet switch, host
 - PPP for dial-up access
- broadcast (shared wire or medium)
 - old-school Ethernet
 - 802.11 wireless LAN, 4G/4G. satellite



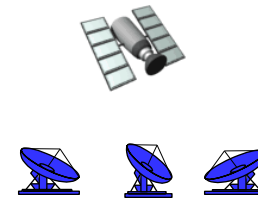
shared wire (e.g.,
cabled Ethernet)



shared radio: 4G/5G



shared radio: WiFi



shared radio: satellite



humans at a cocktail party
(shared air, acoustical)

Multiple access protocols

- single shared broadcast channel
- two or more simultaneous transmissions by nodes: interference
 - *collision* if node receives two or more signals at the same time

multiple access protocol

- distributed algorithm that determines how nodes share channel, i.e., determine when node can transmit
- communication about channel sharing must use channel itself!
 - no out-of-band channel for coordination

An ideal multiple access protocol

given: multiple access channel (MAC) of rate R bps

desiderata:

1. when one node wants to transmit, it can send at rate R .
2. when M nodes want to transmit, each can send at average rate R/M
3. fully decentralized:
 - no special node to coordinate transmissions
 - no synchronization of clocks, slots
4. simple

MAC protocols: taxonomy

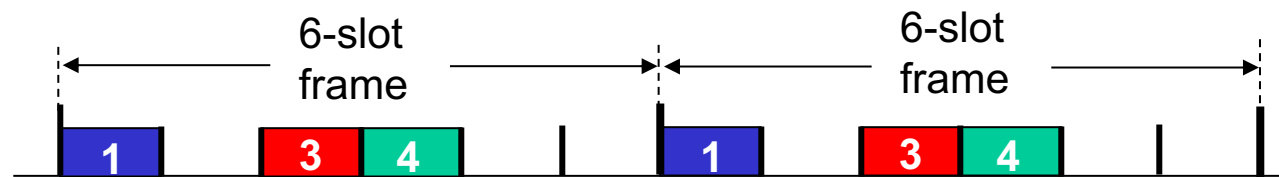
three broad classes:

- **channel partitioning**
 - divide channel into smaller “pieces” (time slots, frequency, etc.)
 - allocate piece to node for exclusive use
- **random access**
 - channel not divided, allow collisions
 - “recover” from collisions
- **“taking turns”**
 - nodes take turns, but nodes with more to send can take longer turns

Channel partitioning MAC protocols: TDMA

TDMA: time division multiple access

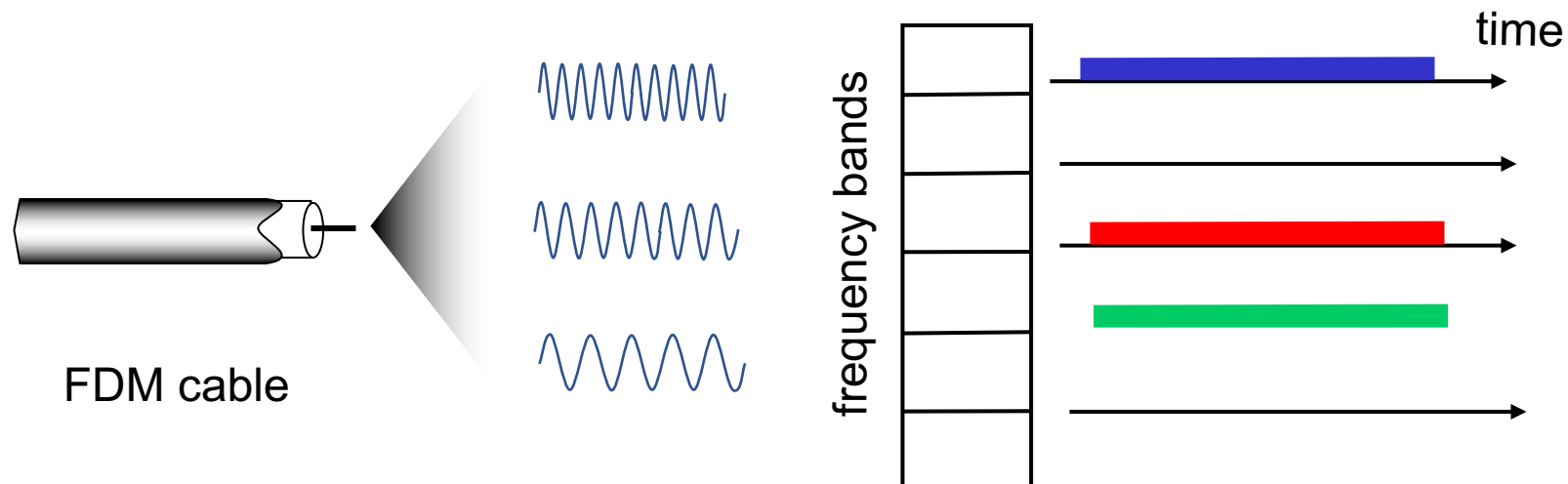
- access to channel in “rounds”
- each station gets fixed length slot (length = packet transmission time) in each round
- unused slots go idle
- example: 6-station LAN, 1,3,4 have packets to send, slots 2,5,6 idle



Channel partitioning MAC protocols: FDMA

FDMA: frequency division multiple access

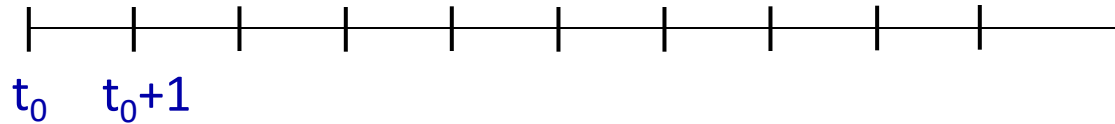
- channel spectrum divided into frequency bands
- each station assigned fixed frequency band
- unused transmission time in frequency bands go idle
- example: 6-station LAN, 1,3,4 have packet to send, frequency bands 2,5,6 idle



Random access protocols

- when node has packet to send
 - transmit at full channel data rate R
 - no *a priori* coordination among nodes
- two or more transmitting nodes:
“collision”
- **random access protocol** specifies:
 - how to detect collisions
 - how to recover from collisions (e.g., via delayed retransmissions)
- examples of random-access MAC protocols:
 - slotted ALOHA
 - CSMA, CSMA/CD

Slotted ALOHA



assumptions:

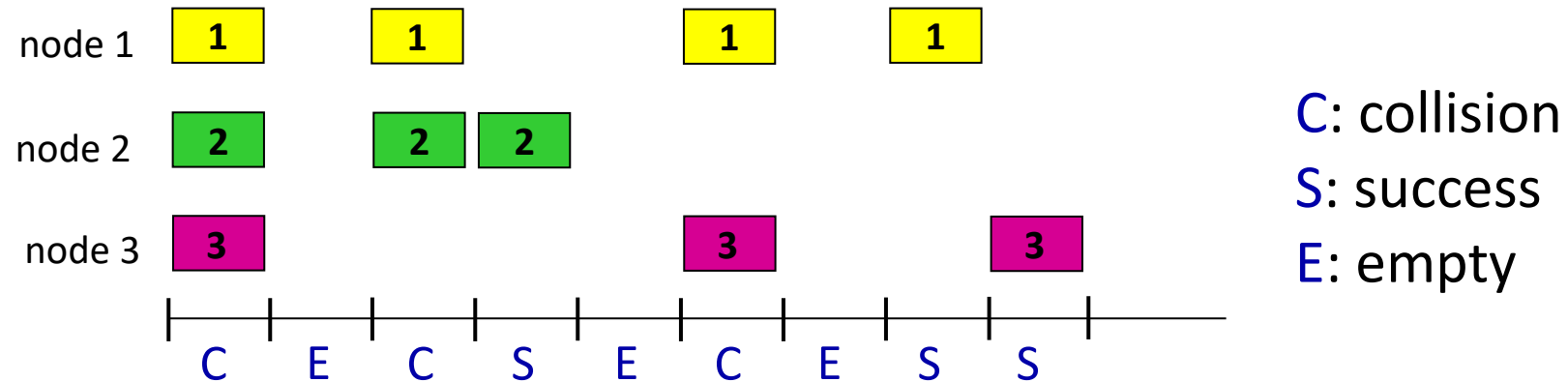
- all frames same size
- time divided into equal size slots (time to transmit 1 frame)
- nodes start to transmit only slot beginning
- nodes are synchronized
- if 2 or more nodes transmit in slot, all nodes detect collision

operation:

- when node obtains fresh frame, transmits in next slot
 - *if no collision*: node can send new frame in next slot
 - *if collision*: node retransmits frame in each subsequent slot with probability p until success

randomization – why?

Slotted ALOHA



Pros:

- single active node can continuously transmit at full rate of channel
- highly decentralized: only slots in nodes need to be in sync
- simple

Cons:

- collisions, wasting slots
- idle slots
- nodes may be able to detect collision in less than time to transmit packet
- clock synchronization

CSMA (carrier sense multiple access)

simple **CSMA**: listen before transmit:

- if channel sensed idle: transmit entire frame
- if channel sensed busy: defer transmission
- human analogy: don't interrupt others!

CSMA/CD: CSMA with *collision detection*

- collisions *detected* within short time
- colliding transmissions aborted, reducing channel wastage
- collision detection easy in wired, difficult with wireless
- human analogy: the polite conversationalist

Ethernet CSMA/CD algorithm

1. Ethernet receives datagram from network layer, creates frame
2. If Ethernet senses channel:
 - if **idle**: start frame transmission.
 - if **busy**: wait until channel idle, then transmit
3. If entire frame transmitted without collision - done!
4. If another transmission detected while sending: abort, send jam signal
5. After aborting, enter *binary (exponential) backoff*:
 - after m th collision, chooses K at random from $\{0, 1, 2, \dots, 2^m - 1\}$. Ethernet waits $K \cdot 512$ bit times, returns to Step 2
 - more collisions: longer backoff interval

“Taking turns” MAC protocols

channel partitioning MAC protocols:

- share channel *efficiently* and *fairly* at high load
- inefficient at low load: delay in channel access, $1/N$ bandwidth allocated even if only 1 active node!

random access MAC protocols

- efficient at low load: single node can fully utilize channel
- high load: collision overhead

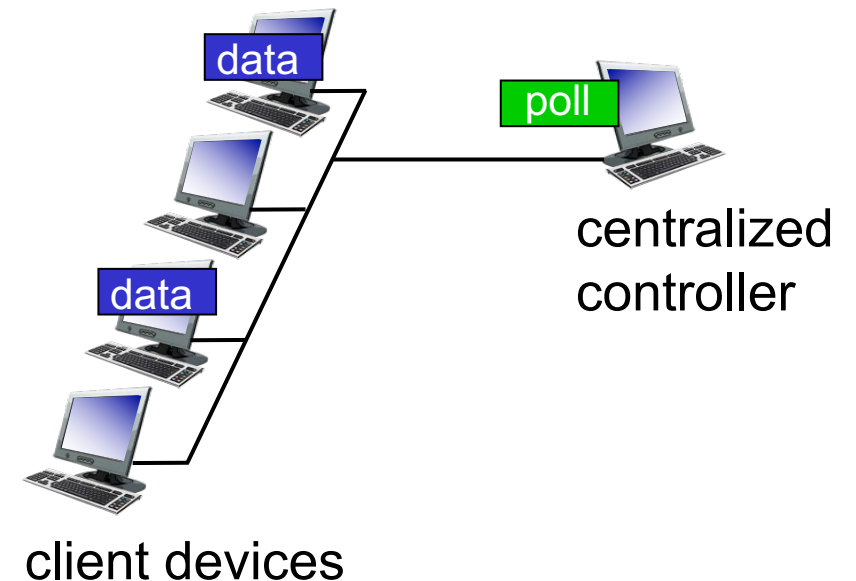
“taking turns” protocols

- look for best of both worlds!

“Taking turns” MAC protocols

polling:

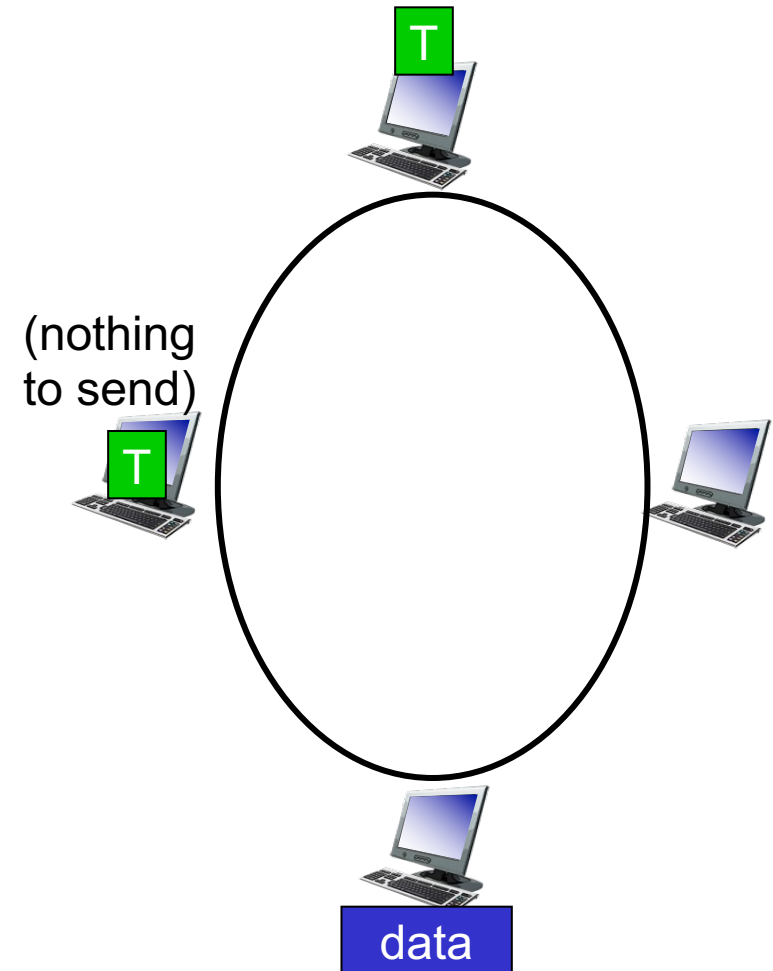
- centralized controller “invites” other nodes to transmit in turn
- concerns:
 - polling overhead
 - latency
 - single point of failure (master)
- Bluetooth uses polling



“Taking turns” MAC protocols

token passing:

- control *token* message explicitly passed from one node to next, sequentially
 - transmit while holding token
- concerns:
 - token overhead
 - latency
 - single point of failure (token)



Summary of MAC protocols

- **channel partitioning**, by time, frequency, etc.
 - Time Division, Frequency Division
- **random access** (dynamic),
 - Slotted ALOHA
 - CSMA carrier sensing: easy in some technologies (wire), hard in others (wireless)
 - CSMA/CD
- **taking turns**
 - polling from central site, token passing

Link layer and LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
 - addressing, ARP
 - Ethernet
 - switches
- Summary



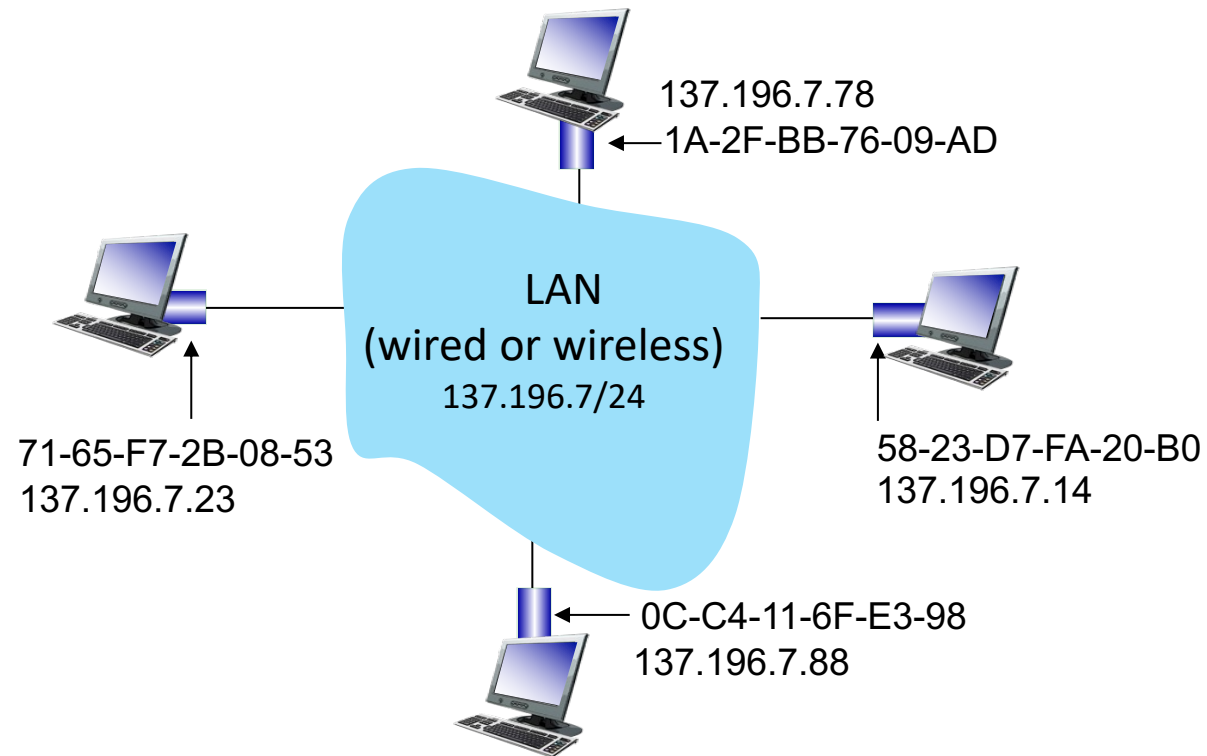
MAC addresses

- 32-bit IP address:
 - *network-layer* address for interface
 - used for layer 3 (network layer) forwarding
 - e.g.: 128.119.40.136
- MAC (or LAN or physical or Ethernet) address:
 - function: used “locally” to get frame from one interface to another physically-connected interface (same subnet, in IP-addressing sense)
 - 48-bit MAC address (for most LANs) burned in NIC ROM, also sometimes software settable
 - e.g.: 1A-2F-BB-76-09-AD
 - *hexadecimal (base 16) notation*
(each “numeral” represents 4 bits)

MAC addresses

each interface on LAN

- has unique 48-bit **MAC** address
- has a locally unique 32-bit IP address (as we've seen)

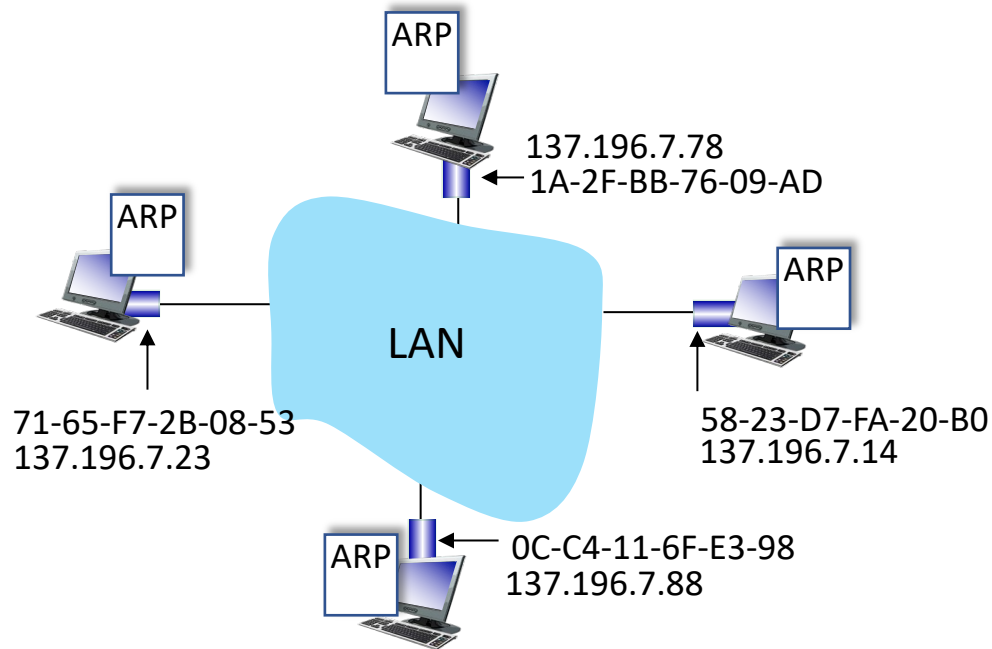


MAC addresses

- MAC address allocation administered by IEEE
- manufacturer buys portion of MAC address space (to assure uniqueness)
- analogy:
 - MAC address: like Social Security Number
 - IP address: like postal address
- MAC flat address: portability
 - can move interface from one LAN to another
 - recall IP address *not* portable: depends on IP subnet to which node is attached

ARP: address resolution protocol

Question: how to determine interface's MAC address, knowing its IP address?



ARP table: each IP node (host, router) on LAN has table

- IP/MAC address mappings for some LAN nodes:
< IP address; MAC address; TTL >
- TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)

ARP protocol in action

example: A wants to send datagram to B

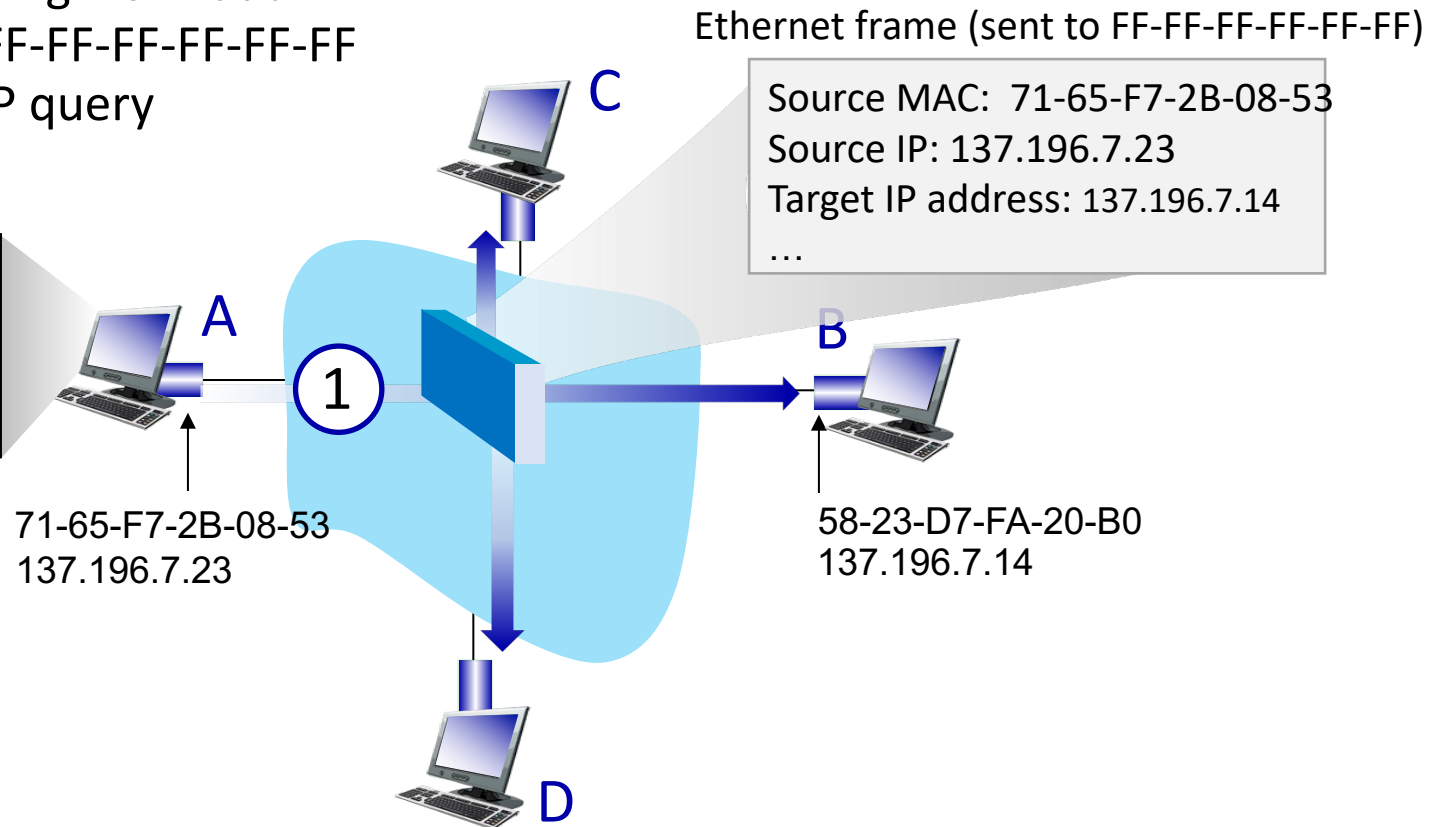
- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address

A broadcasts ARP query, containing B's IP addr

- ①
- destination MAC address = FF-FF-FF-FF-FF-FF
 - all nodes on LAN receive ARP query

ARP table in A

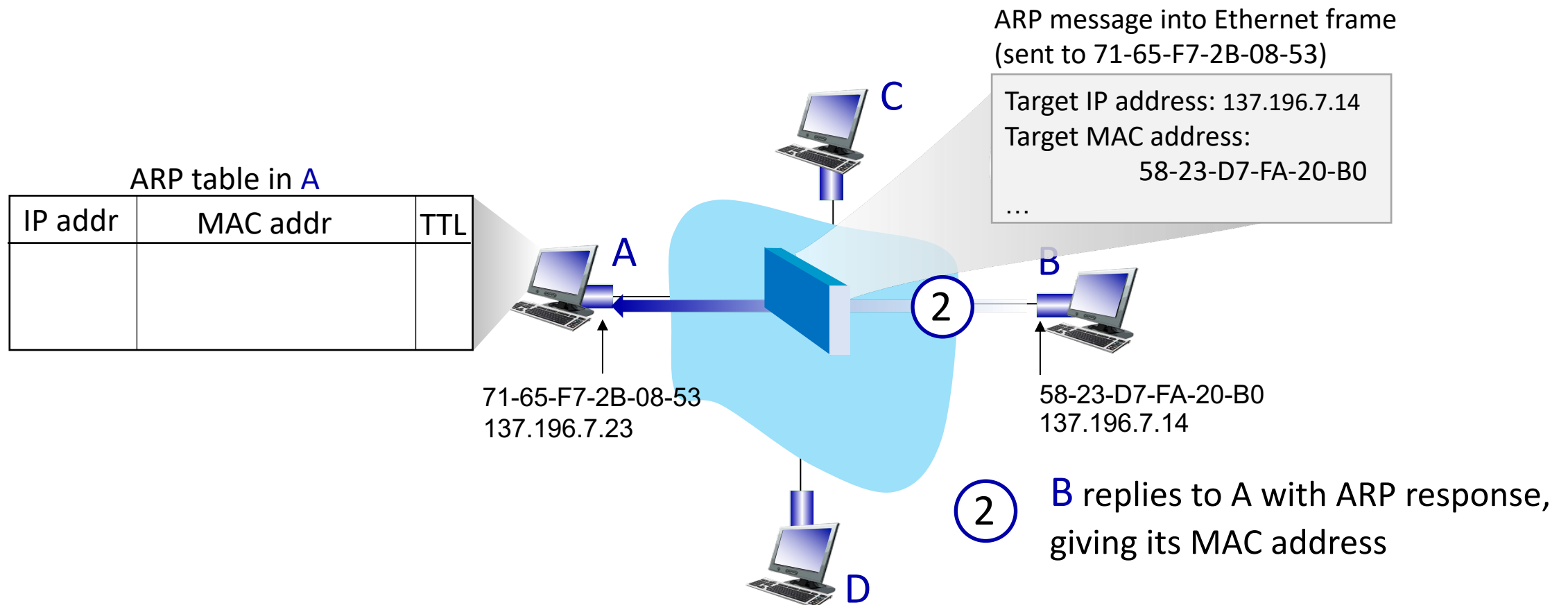
IP addr	MAC addr	TTL



ARP protocol in action

example: A wants to send datagram to B

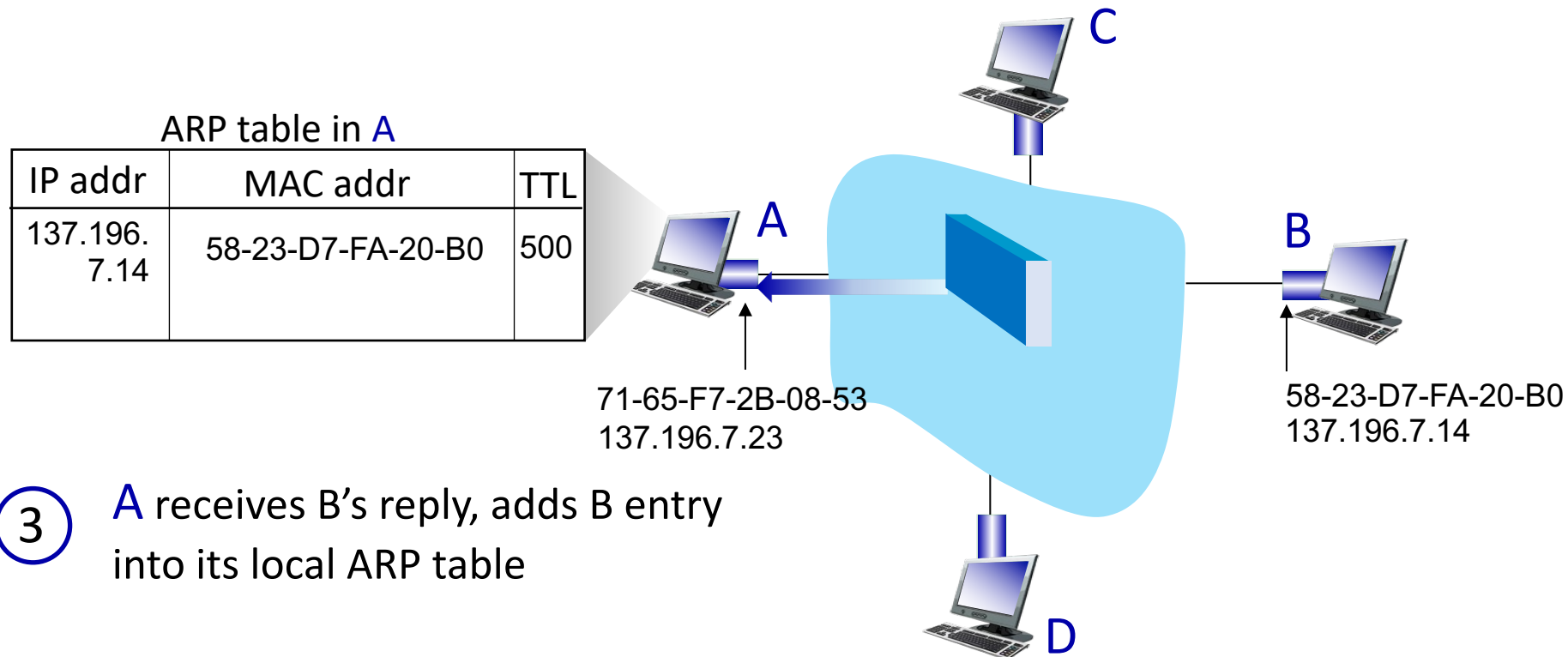
- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address



ARP protocol in action

example: A wants to send datagram to B

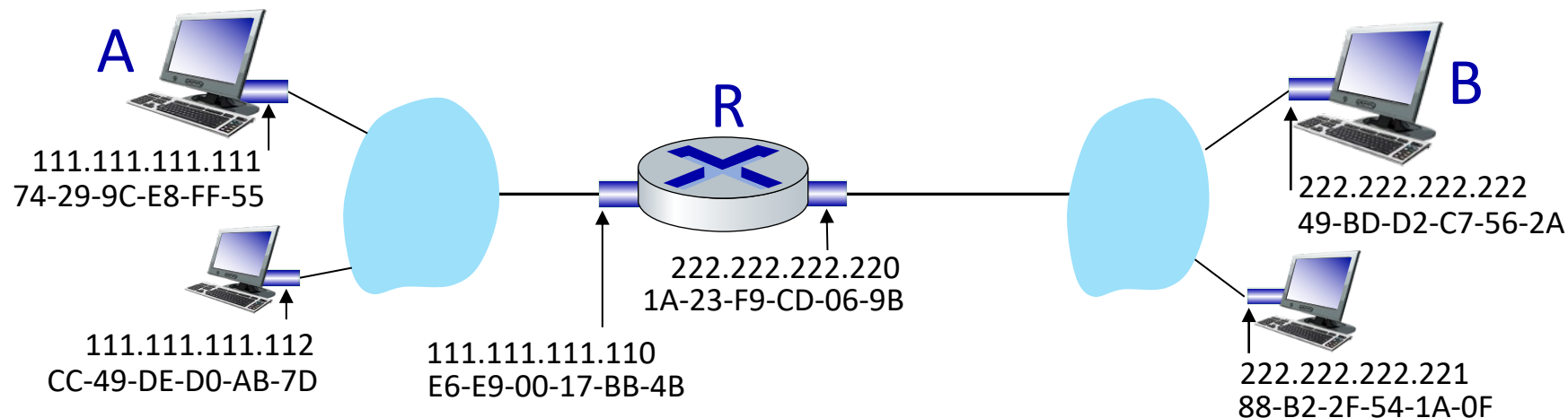
- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address



Routing to another subnet: addressing

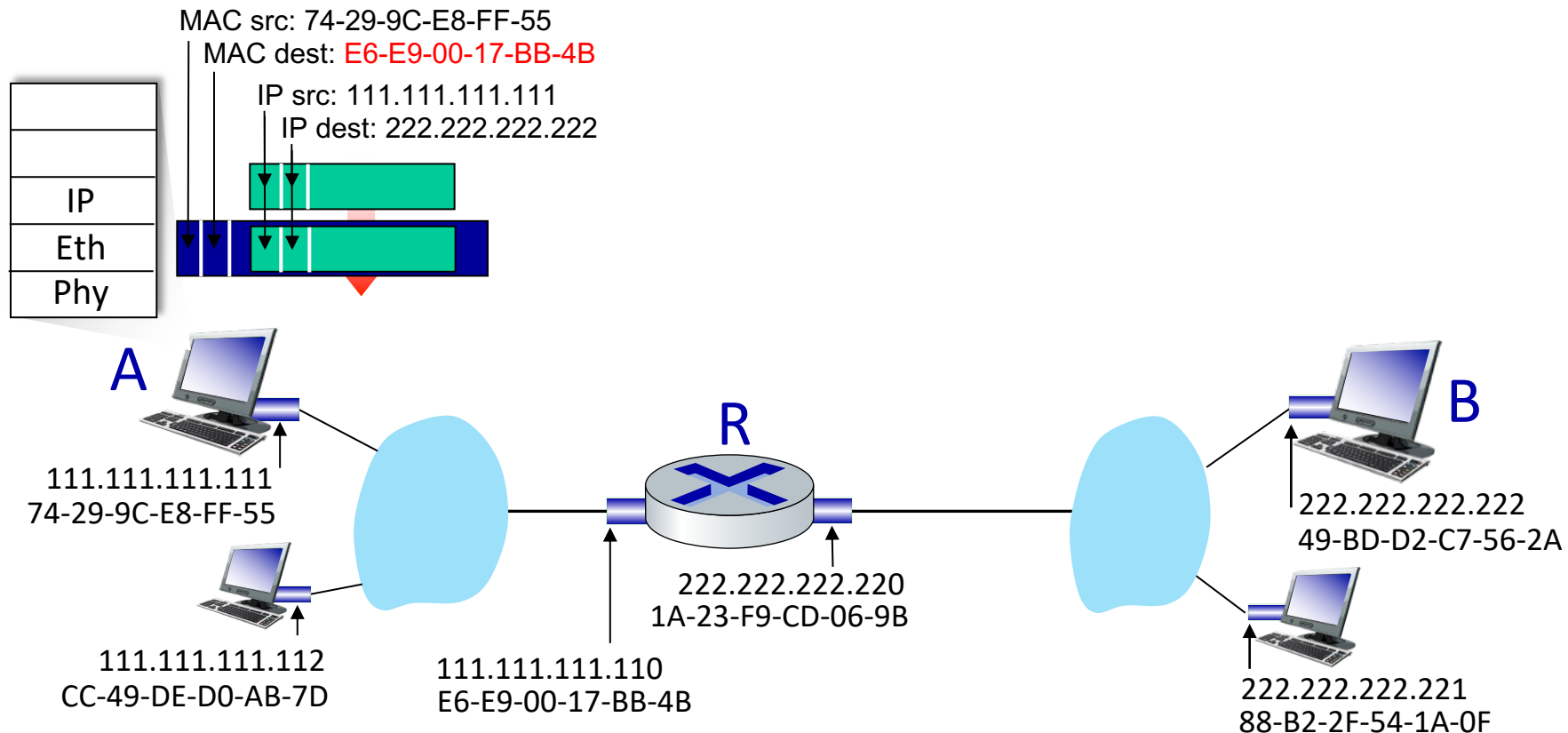
walkthrough: sending a datagram from *A* to *B* via *R*

- focus on addressing – at IP (datagram) and MAC layer (frame) levels
- assume that:
 - A knows B's IP address
 - A knows IP address of first hop router, R (how?)
 - A knows R's MAC address (how?)



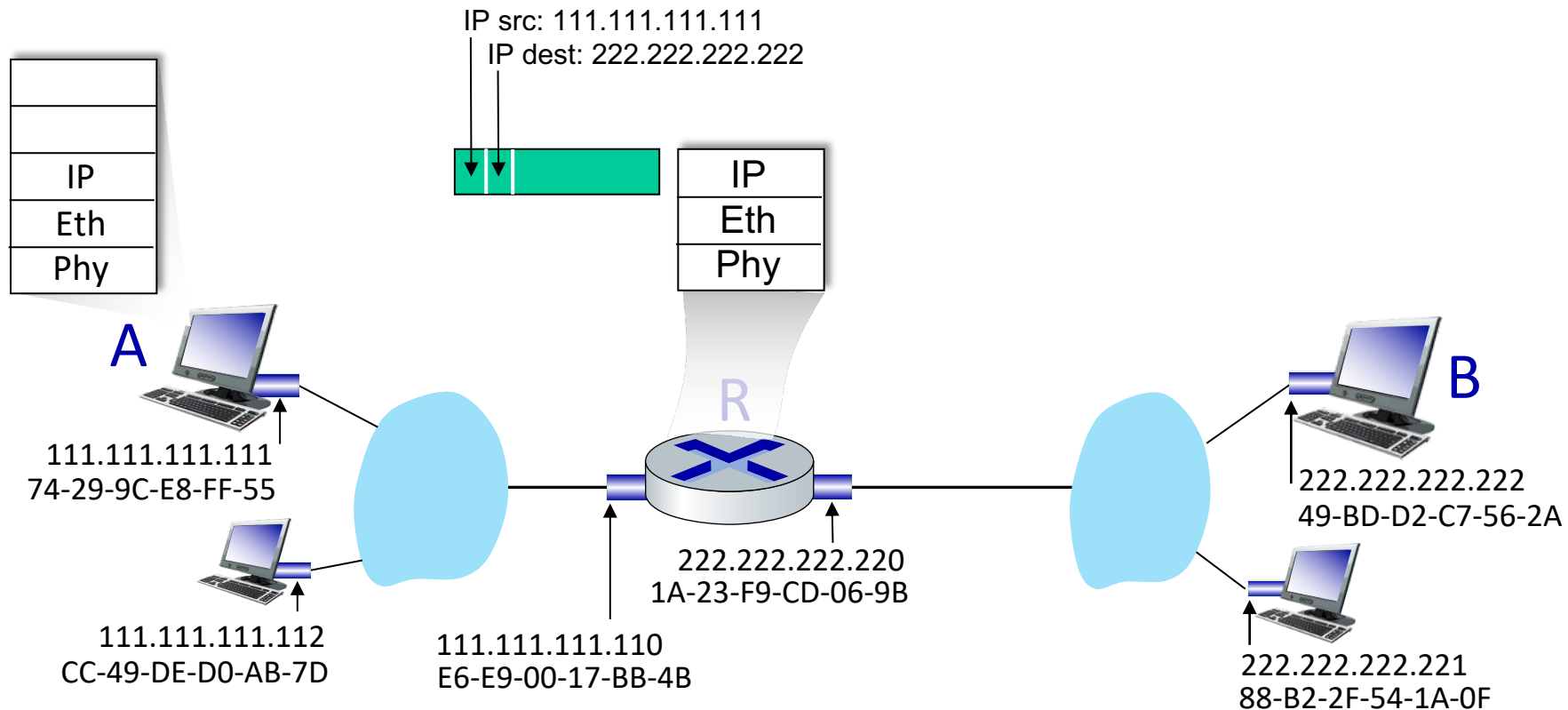
Routing to another subnet: addressing

- A creates IP datagram with IP source A, destination B
- A creates link-layer frame containing A-to-B IP datagram
 - R's MAC address is frame's destination



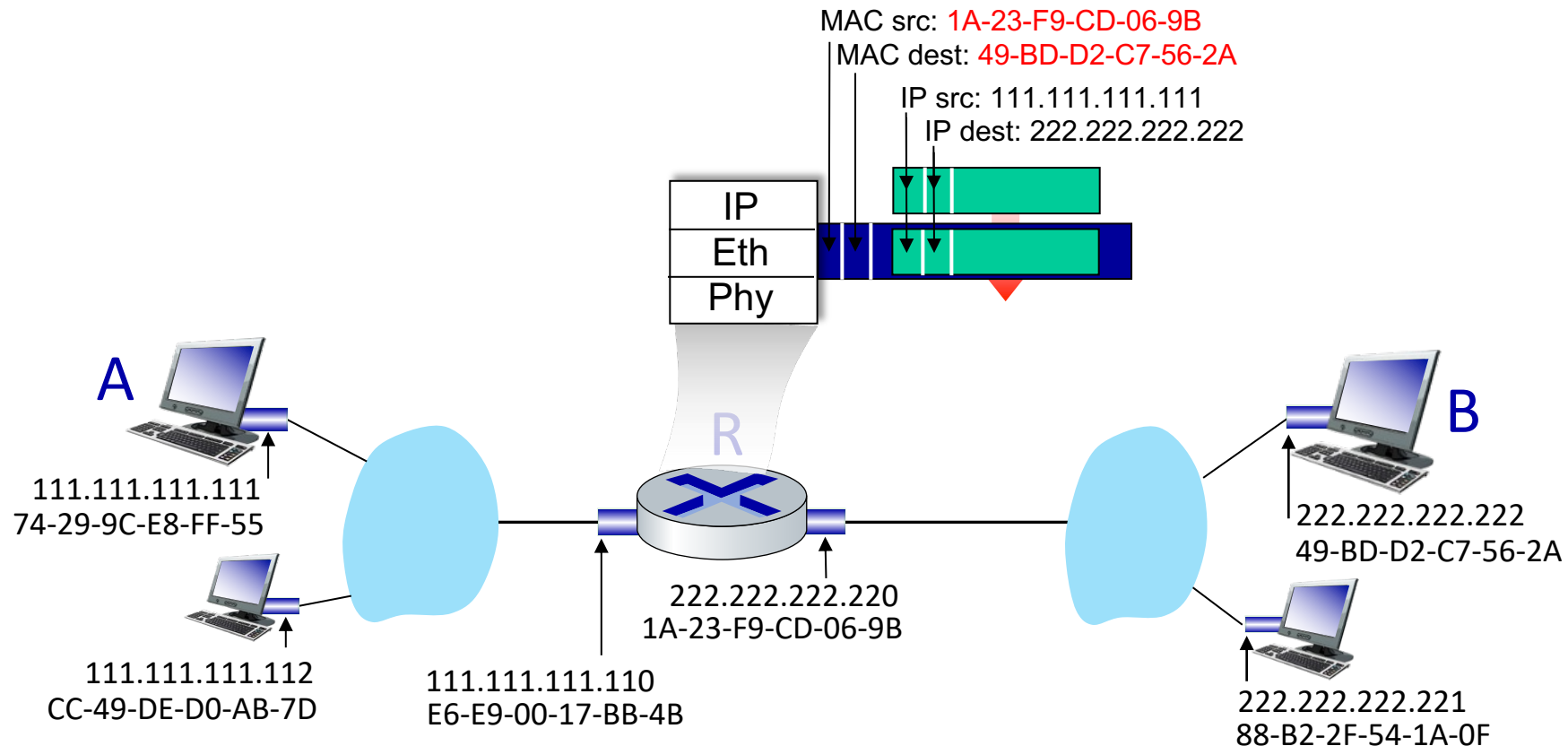
Routing to another subnet: addressing

- frame sent from A to R
- frame received at R, datagram removed, passed up to IP



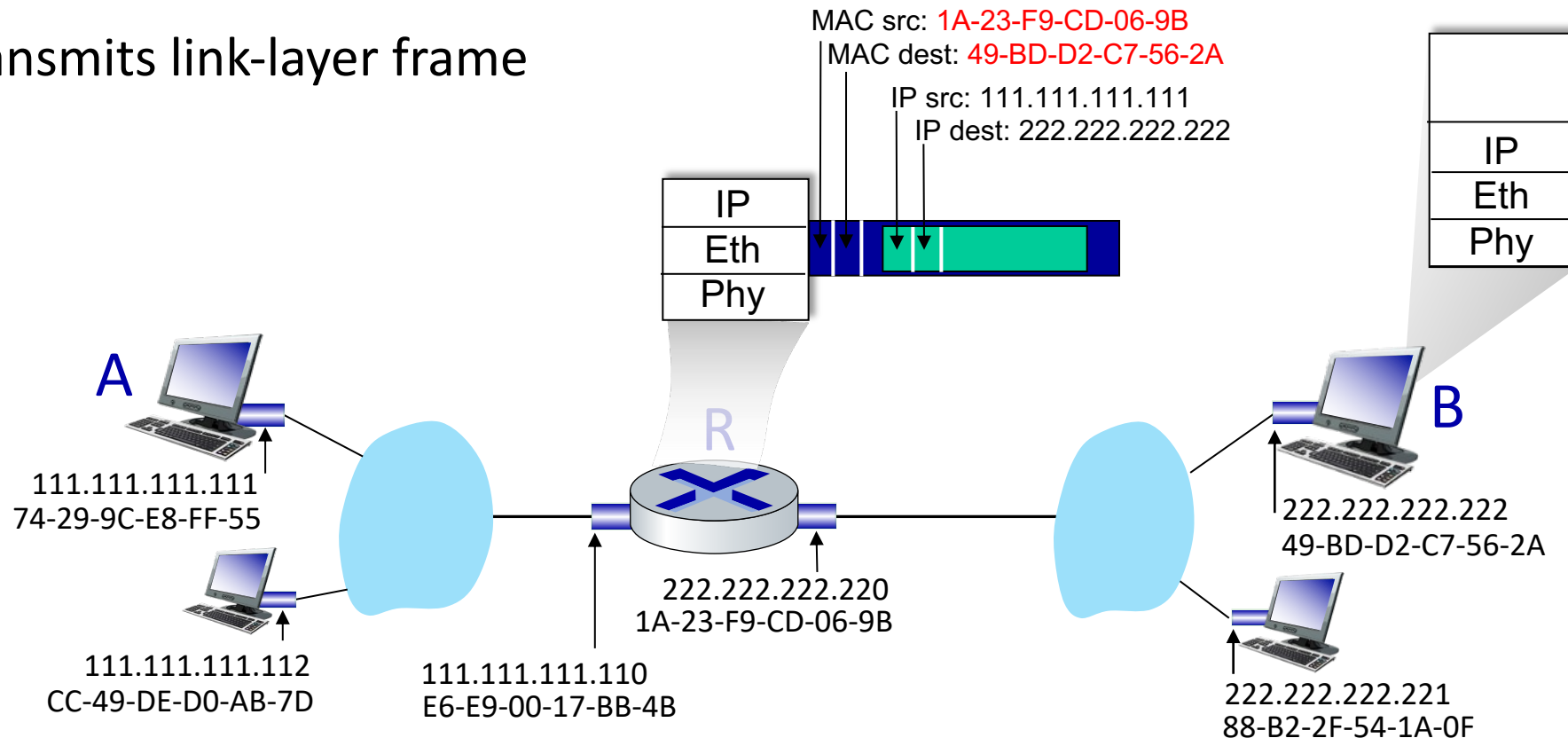
Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address



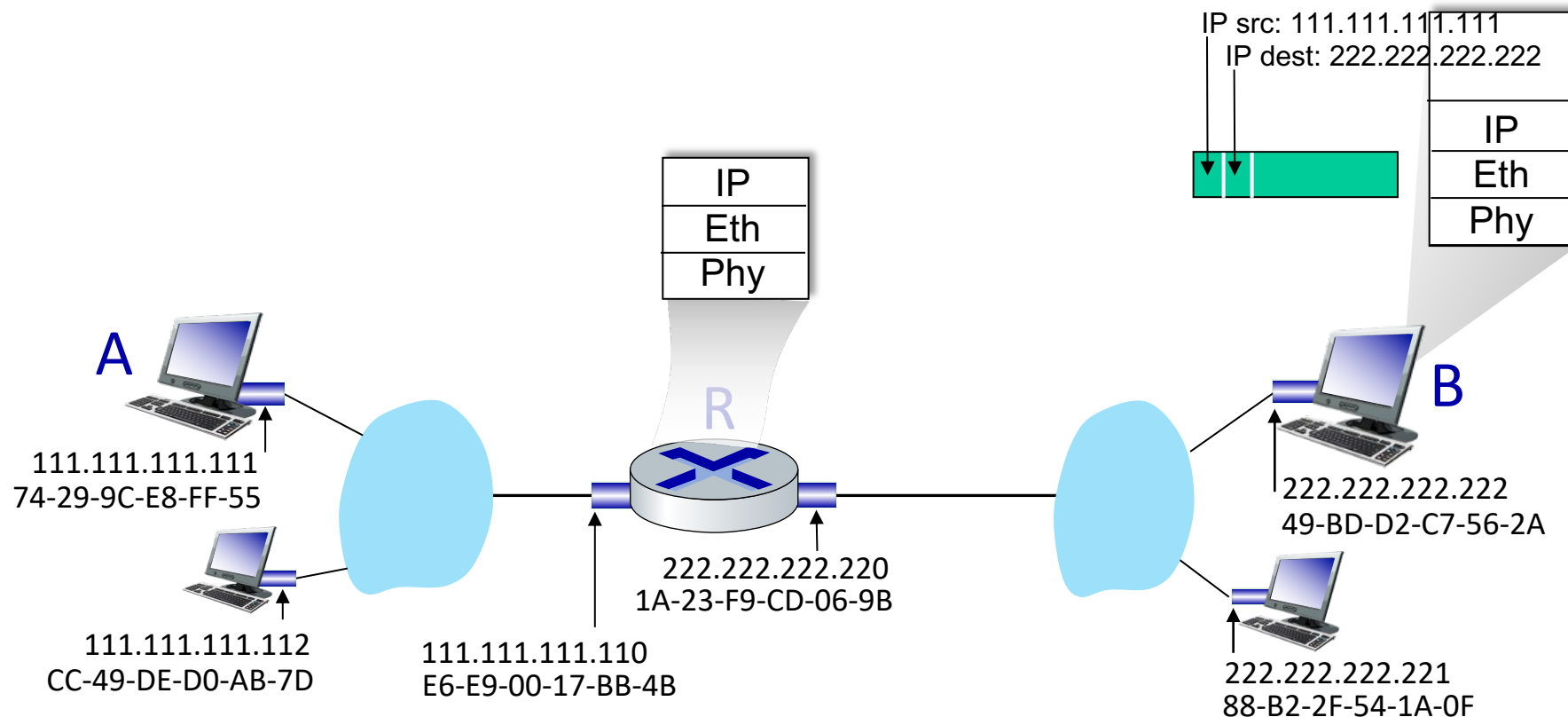
Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address
- transmits link-layer frame



Routing to another subnet: addressing

- B receives frame, extracts IP datagram destination B
- B passes datagram up protocol stack to IP



Link layer and LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
 - addressing, ARP
 - Ethernet
 - switches
- Summary

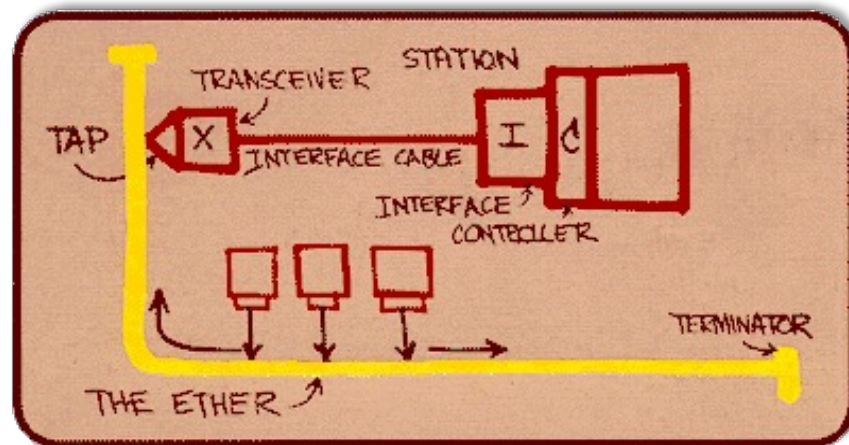


Ethernet

“dominant” wired LAN technology:

- first widely used LAN technology
- simpler, cheap
- kept up with speed race: 10 Mbps – 400 Gbps
- single chip, multiple speeds (e.g., Broadcom BCM5761)

Metcalfe's Ethernet sketch



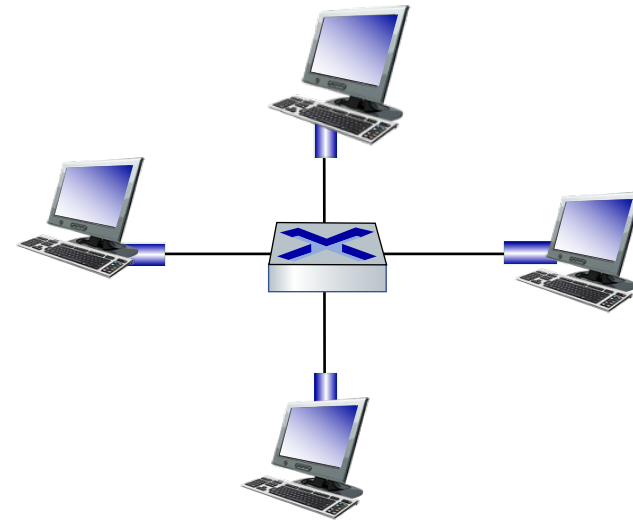
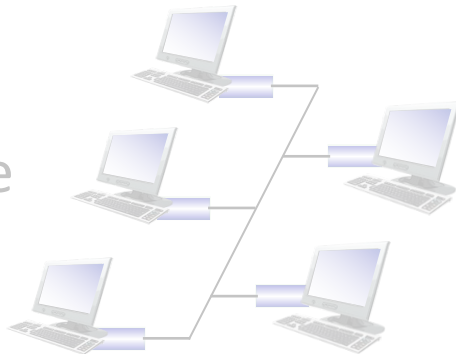
Bob Metcalfe: Ethernet co-inventor,
2022 ACM Turing Award recipient



Ethernet: physical topology

- **bus:** popular through mid 90s
 - all nodes in same collision domain (can collide with each other)
- **switched:** prevails today
 - active link-layer 2 *switch* in center
 - each “spoke” runs a (separate) Ethernet protocol (nodes do not collide with each other)

bus: coaxial cable



switched

Ethernet frame structure

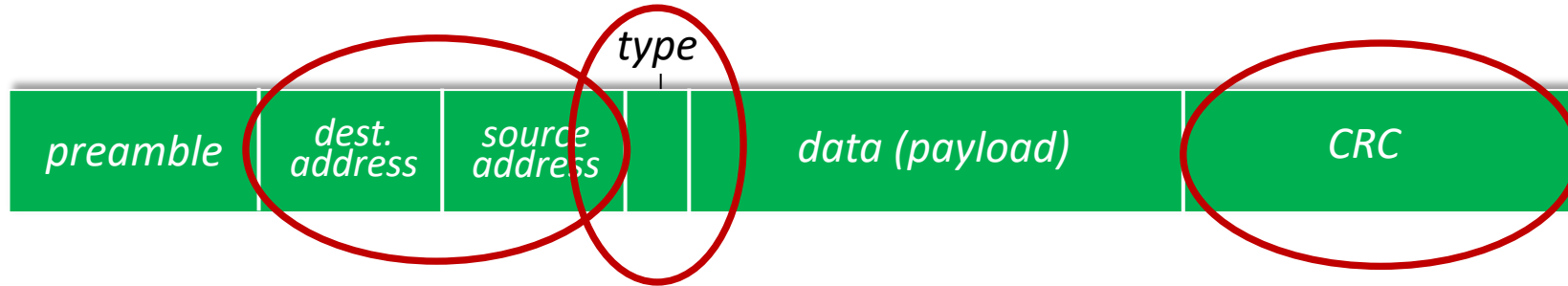
sending interface encapsulates IP datagram (or other network layer protocol packet) in **Ethernet frame**



preamble:

- used to synchronize receiver, sender clock rates
- 7 bytes of 10101010 followed by one byte of 10101011

Ethernet frame structure



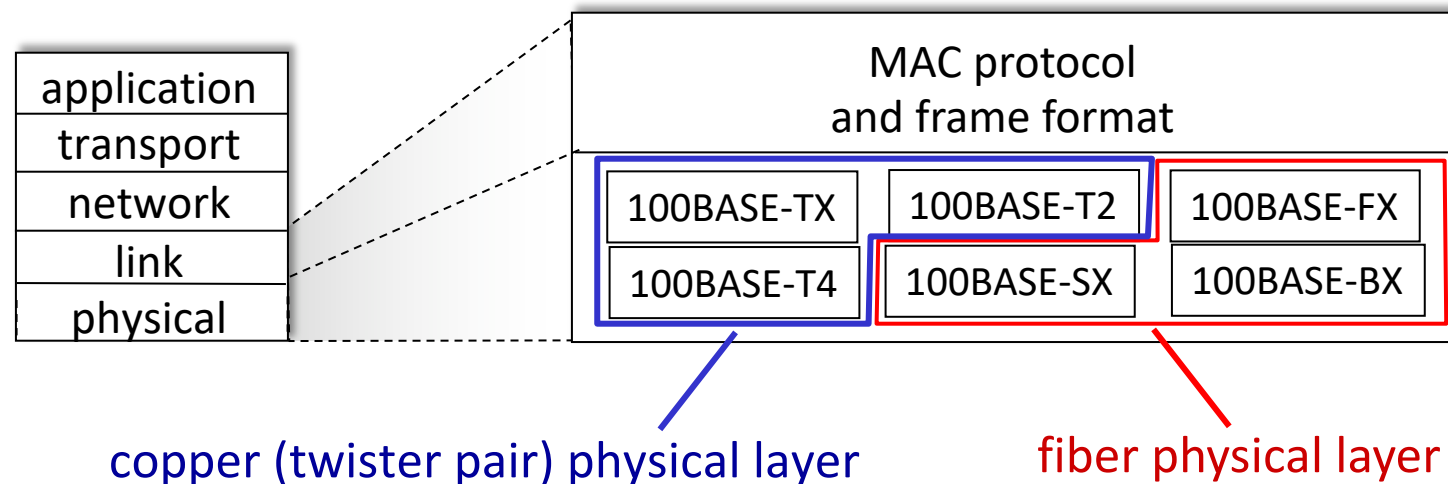
- **addresses:** 6-Byte source, destination MAC addresses
 - if adapter receives frame with matching destination address, or with broadcast address (e.g., ARP packet), it passes data in frame to network layer protocol
 - otherwise, adapter discards frame
- **type:** indicates higher-layer protocol
 - mostly IP but others possible, e.g., Novell IPX, AppleTalk
 - used to demultiplex up at receiver
- **CRC:** cyclic-redundancy check at receiver
 - error detected: frame is dropped

Ethernet: unreliable, connectionless

- **connectionless**: no handshaking between sending and receiving NICs
- **unreliable**: receiving NIC doesn't send ACKs or NAKs to sending NIC
 - data in dropped frames recovered only if initial sender uses higher layer rdt (e.g., TCP), otherwise dropped data lost
- Ethernet's MAC protocol: unslotted **CSMA/CD with binary backoff**

802.3 Ethernet standards: link & physical layers

- *many* different Ethernet standards
 - common MAC protocol and frame format
 - different speeds: 2 Mbps, ... 100 Mbps, 1Gbps, 10 Gbps, 40 Gbps, 80 Gbps
 - different physical layer media: fiber, cable



Link layer and LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
 - addressing, ARP
 - Ethernet
 - switches
- Summary

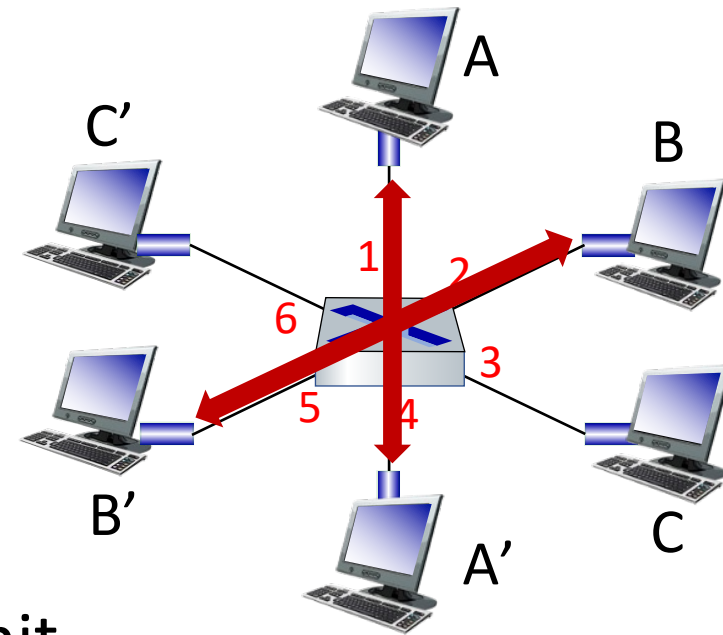


Ethernet switch

- Switch is a **link-layer** device: takes an *active* role
 - store, forward Ethernet (or other type of) frames
 - examine incoming frame's MAC address, *selectively* forward frame to one-or-more outgoing links when frame is to be forwarded on segment, uses CSMA/CD to access segment
- **transparent:** hosts *unaware* of presence of switches
- **plug-and-play, self-learning**
 - switches do not need to be configured

Switch: multiple simultaneous transmissions

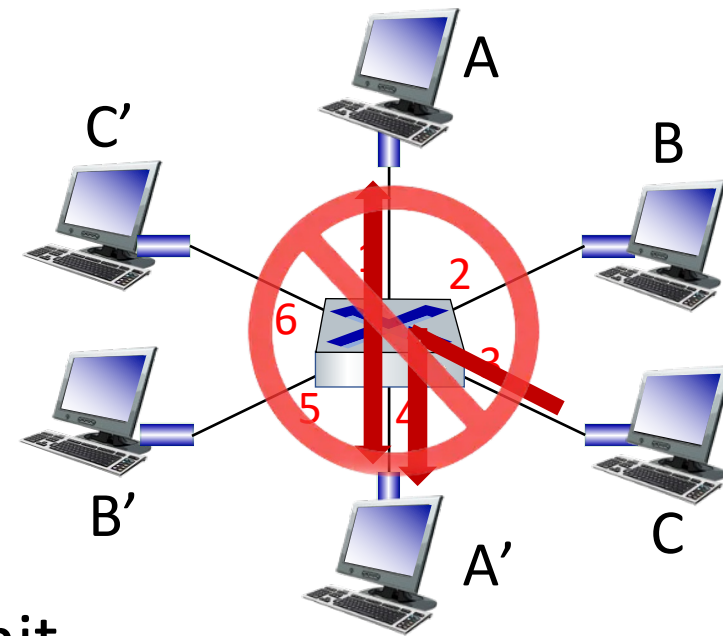
- hosts have dedicated, direct connection to switch
- switches buffer packets
- Ethernet protocol used on *each* incoming link, so:
 - no collisions; full duplex
 - each link is its own collision domain
- **switching**: A-to-A' and B-to-B' can transmit simultaneously, without collisions



switch with six interfaces (1,2,3,4,5,6)

Switch: multiple simultaneous transmissions

- hosts have dedicated, direct connection to switch
- switches buffer packets
- Ethernet protocol used on *each* incoming link, so:
 - no collisions; full duplex
 - each link is its own collision domain
- **switching:** A-to-A' and B-to-B' can transmit simultaneously, without collisions
 - but A-to-A' and C to A' can *not* happen simultaneously



switch with six interfaces (1,2,3,4,5,6)

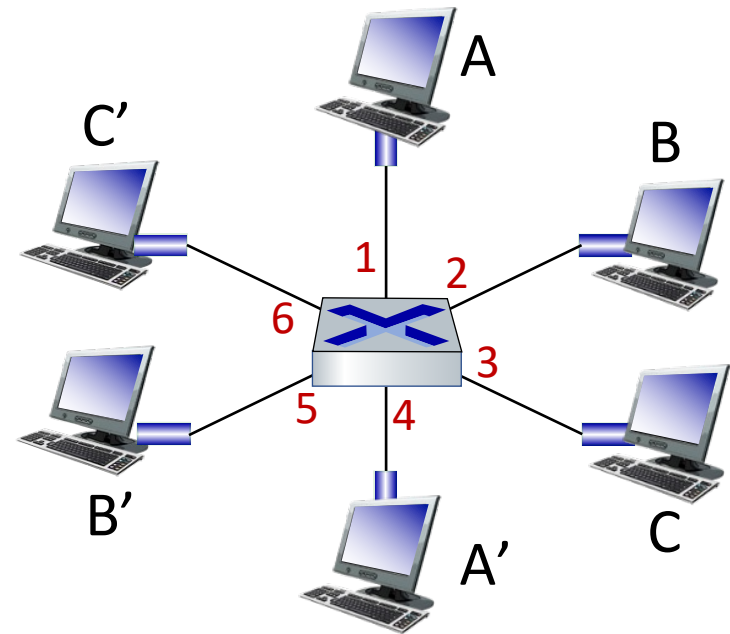
Switch forwarding table

Q: how does switch know A' reachable via interface 4, B' reachable via interface 5?

A: each switch has a **switch table**, each entry:

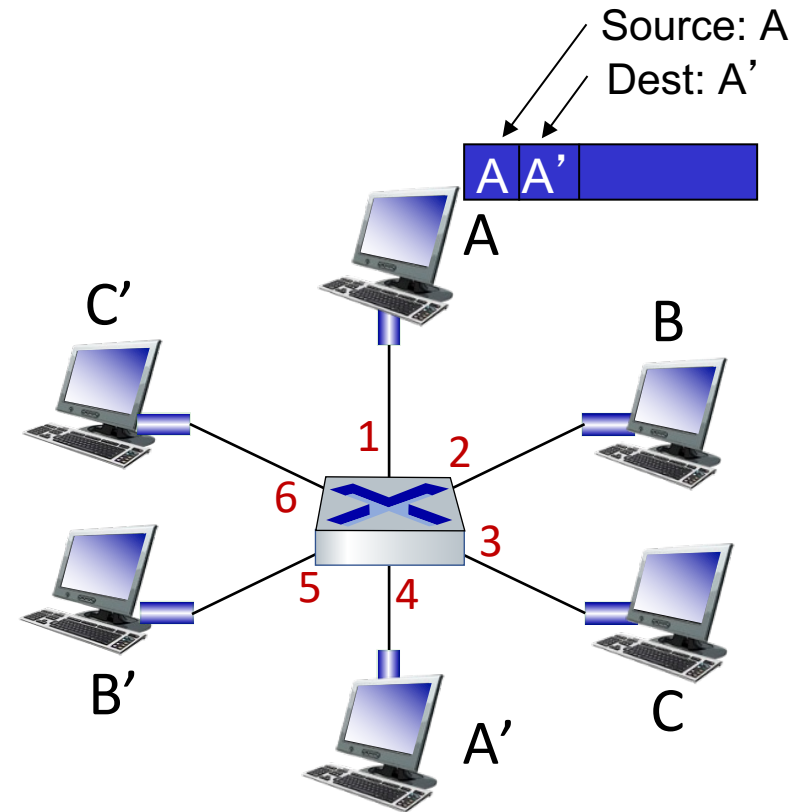
- (MAC address of host, interface to reach host, time stamp)
- looks like a routing table!

Q: how are entries created and maintained in a switch table?



Switch: self-learning

- switch *learns* which hosts can be reached through which interfaces
- when frame received, switch “learns” location of sender: incoming LAN segment
- records sender/location pair in switch table



MAC addr	interface	TTL
A	1	60

*Switch table
(initially empty)*

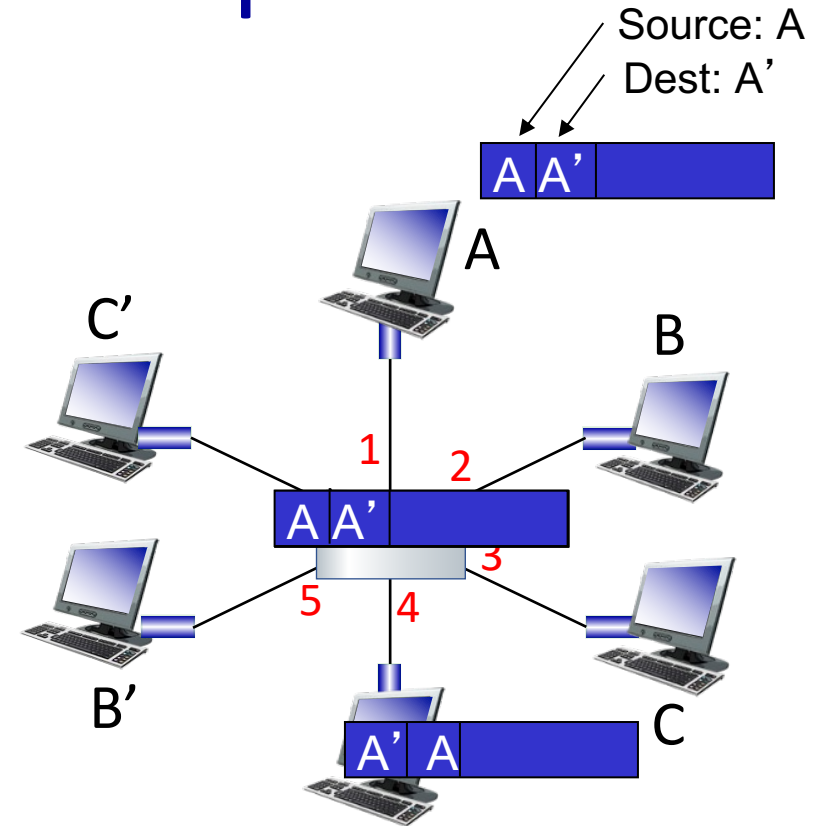
Switch: frame filtering/forwarding

when frame received at switch:

1. record incoming link, MAC address of sending host
2. index switch table using MAC destination address
3. if entry found for destination
then {
if destination on segment from which frame arrived
then drop frame
else forward frame on interface indicated by entry
}
else flood /* forward on all interfaces except arriving interface */

Self-learning, forwarding: example

- frame destination, A', location unknown: **flood**
- destination A location known: **selectively send on just one link**

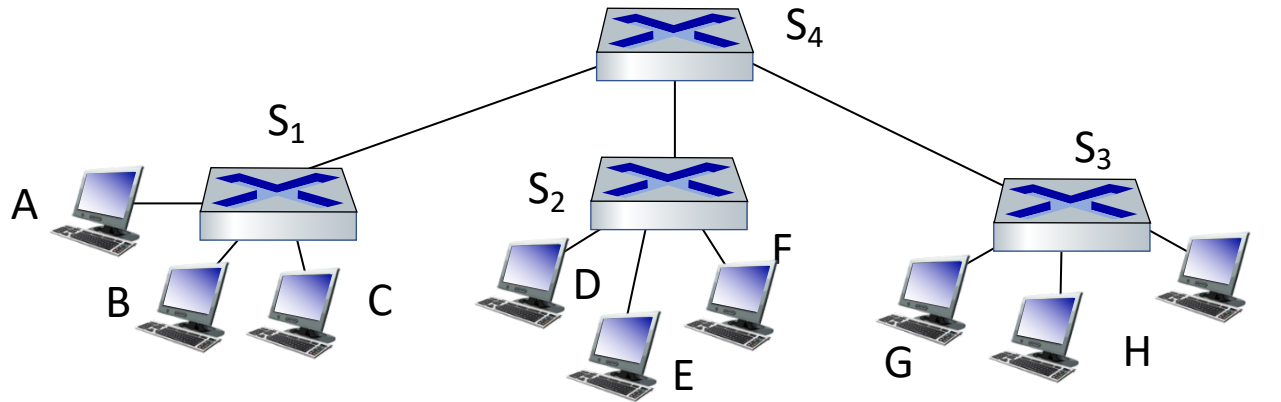


MAC addr	interface	TTL
A	1	60
A'	4	60

*switch table
(initially empty)*

Interconnecting switches

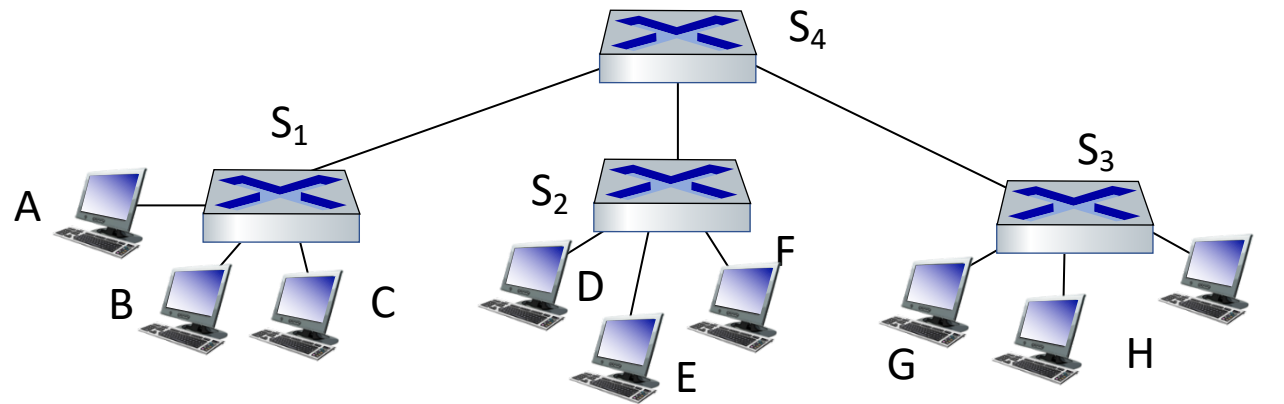
self-learning switches can be connected together:



Q: sending from A to G - how does S₁ know to forward frame destined to G via S₄ and S₃?

Interconnecting switches

self-learning switches can be connected together:



Q: sending from A to G - how does S₁ know to forward frame destined to G via S₄ and S₃?

- **A:** self learning! (works exactly the same as in single-switch case!)

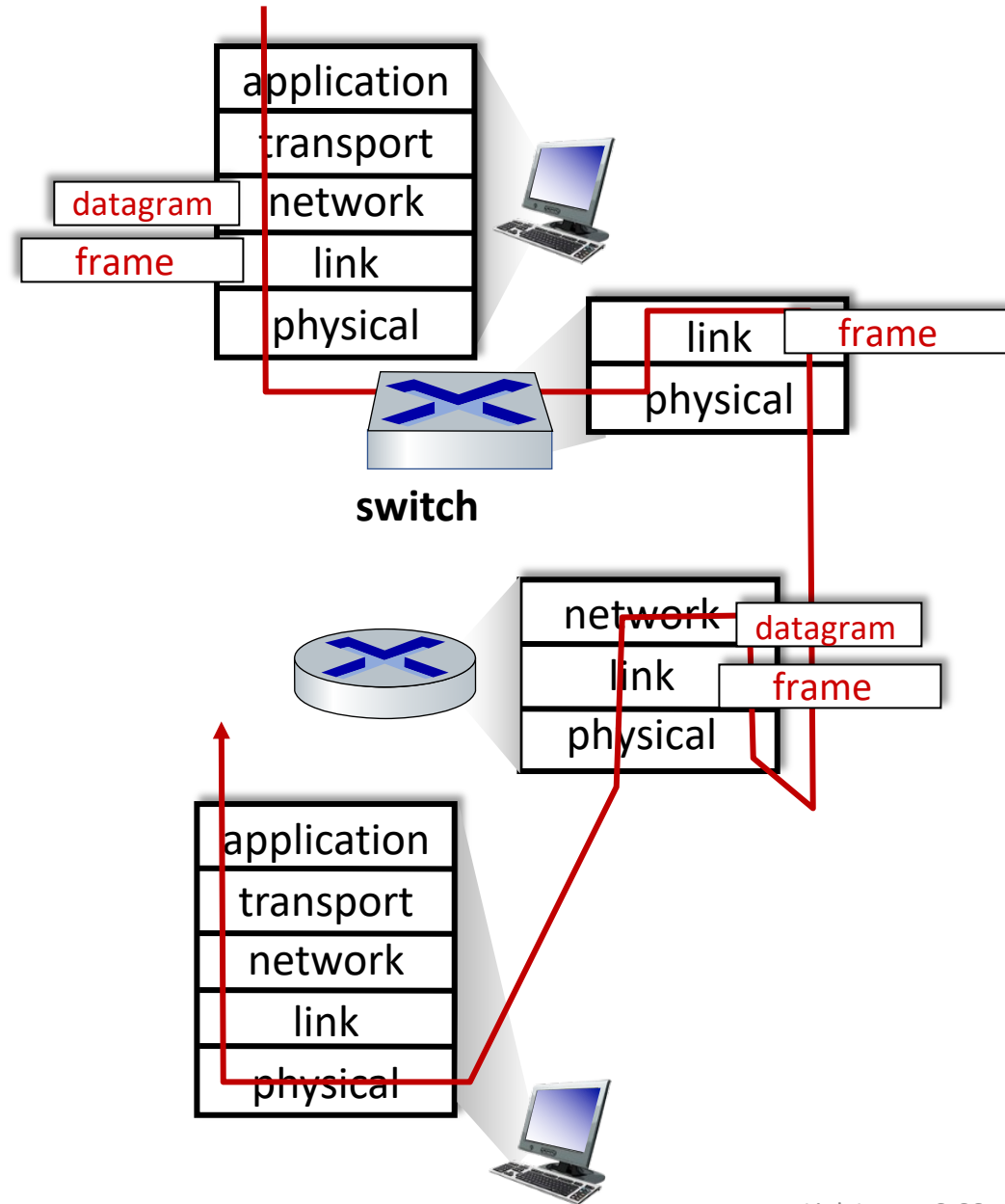
Switches vs. routers

both are store-and-forward:

- *routers*: network-layer devices (examine network-layer headers)
- *switches*: link-layer devices (examine link-layer headers)

both have forwarding tables:

- *routers*: compute tables using routing algorithms, IP addresses
- *switches*: learn forwarding table using flooding, learning, MAC addresses



Summary

- principles behind data link layer services:
 - error detection, correction
 - sharing a broadcast channel: multiple access
 - link layer addressing
- instantiation, implementation of various link layer technologies
 - Ethernet
 - switched LANS
- Self-learning Switches