50.012 Networks

Lecture 11: Midterm Recap

2021 Term 6

Assoc. Prof. CHEN Binbin

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Midterm Exam Info

- Weight of midterm in overall assessment: 20%
- Date: 22 Oct 2021, Friday
- Time: 2:30pm – 4:00pm (90mins)
- Venue: TT9 & TT10 (1.415 &1.416), and TT23 (2.413)
- Scope: All topics covered in the first five weeks

The session on 22 Oct 9am to 11am will be reserved for consultation (slots can be booked in advance by appointment). Attendance is optional.

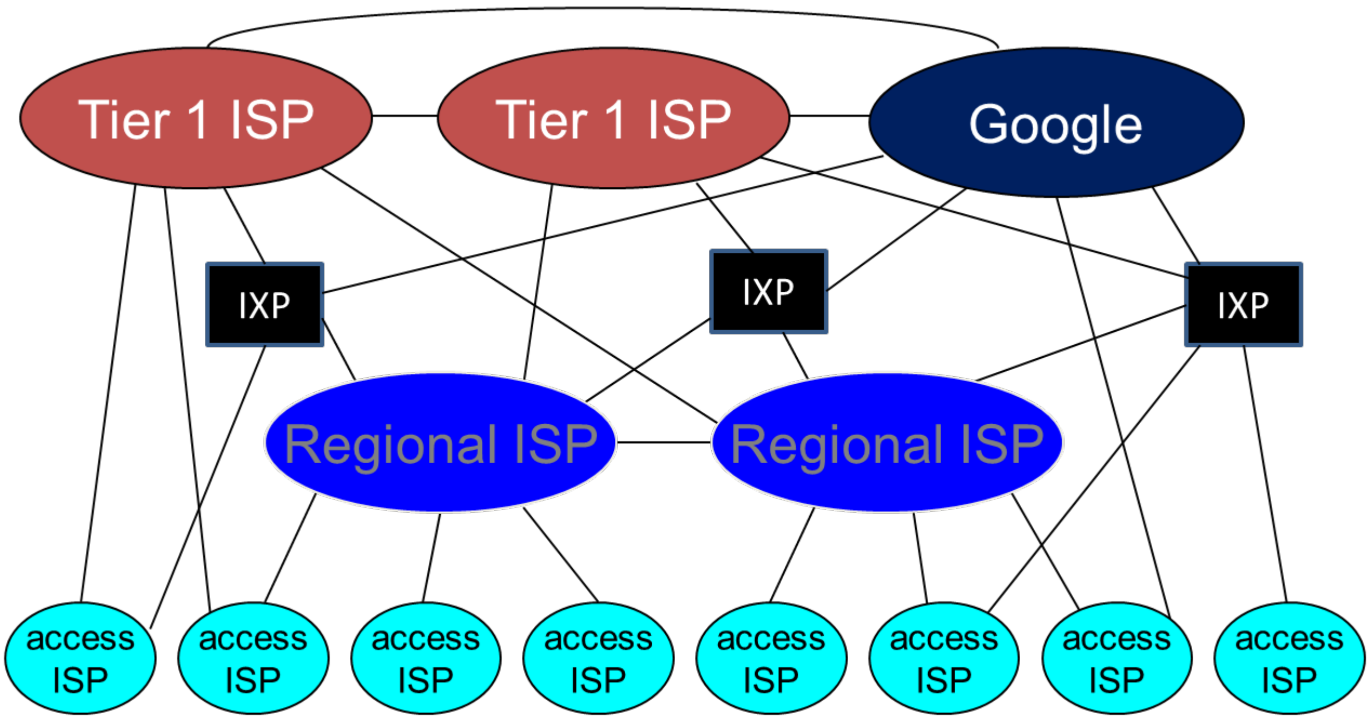
Midterm Exam Format

- Close book; no Internet access
- Types of questions similar to our quizzes, homework assignments, and in-class activities
- No e-devices (except for calculator) allowed during the exam
  - No extended coding, nor difficult calculation needed in the exam
- One A4 cheat sheet (double sided) allowed – hand-written or printed

Outline

- Fundamentals
  - Internet structure, protocol stack, encapsulation, performance, packet switching
- Applications
- Transport Layer

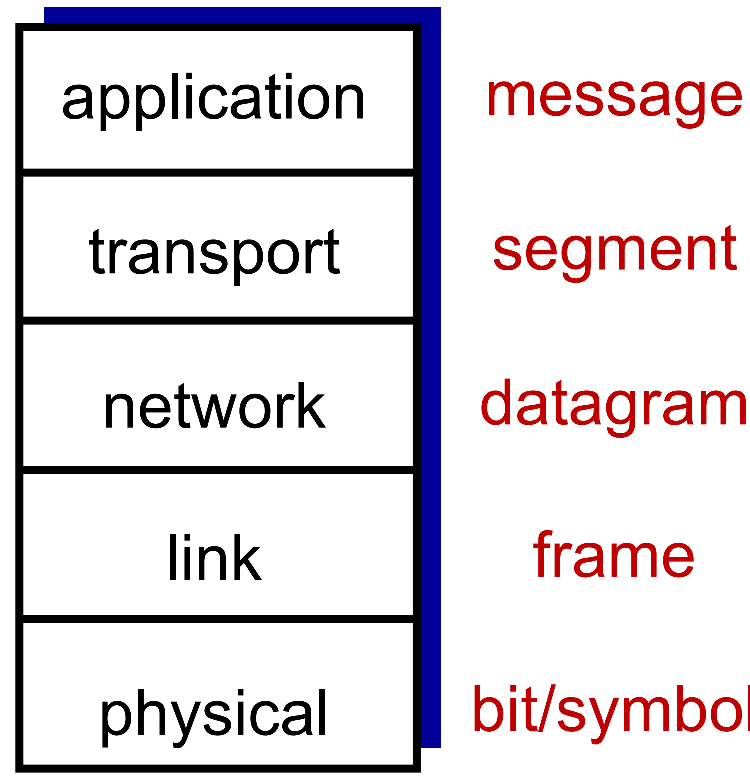
Internet structure



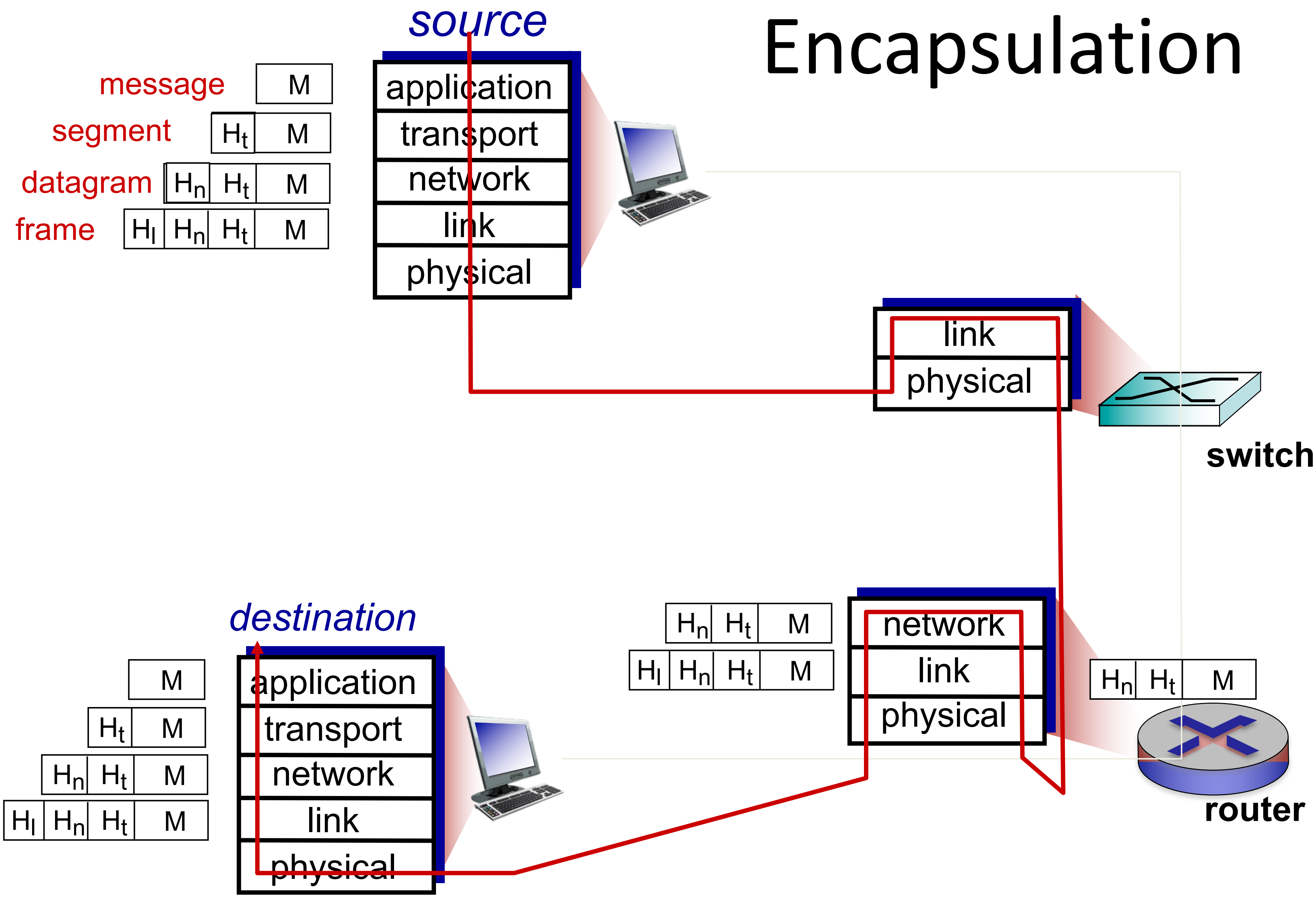
Points of presence (PoPs), multi-homing, peering, Internet exchange points (IXP), content provider networks

Internet protocol stack

- *application:* supporting network applications
  - HTTP (web), SMTP (E-Mail), 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)
- *physical:* bits "on the wire"



Encapsulation



Circuit switching vs. packet switching

Consider a number of users sharing a 100 Mbps link. Each user requires 2.5Mbps when transmitting, but transmits only 5 percent of the time (independently).

a) When circuit switching is used, how many users can be supported? What is the average link utilization when the maximum number of users are supported?

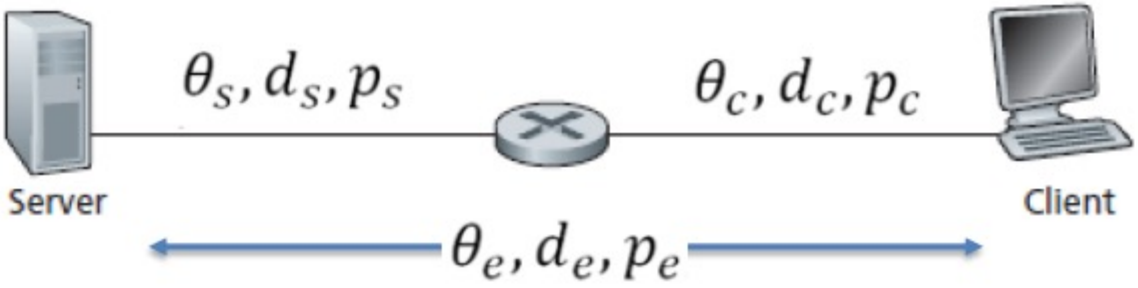
b) For the remainder of this problem, suppose packet switching is used. Suppose there are 500 users in total. Find the probability that at any given time, exactly x users are transmitting. Hint: use BinomialCoefficient(N, i) (or just (N,i) in short) to denote the "N choose i", i.e., the number of ways to choose an (unordered) subset of i items from N items.

c) Find the probability that there are more than 40 users transmitting. Compute your answer using https://homepage.divms.uiowa.edu/~mbognar/applets/bin.html

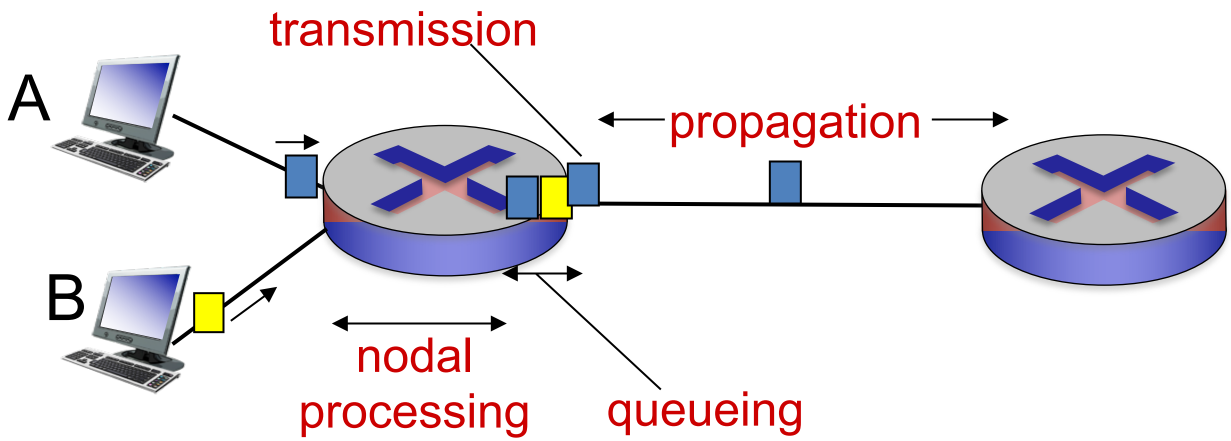
d) Estimate the average link utilization when 500 users are supported by packet switching

Network performance

- Throughput, delay, and loss probability



Four sources of packet delay



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

d_{trans}: transmission delay:

- L: packet length (bits)
- R: link *bandwidth (bps)*
- $d_{trans} = L/R$

d_{prop}: propagation delay:

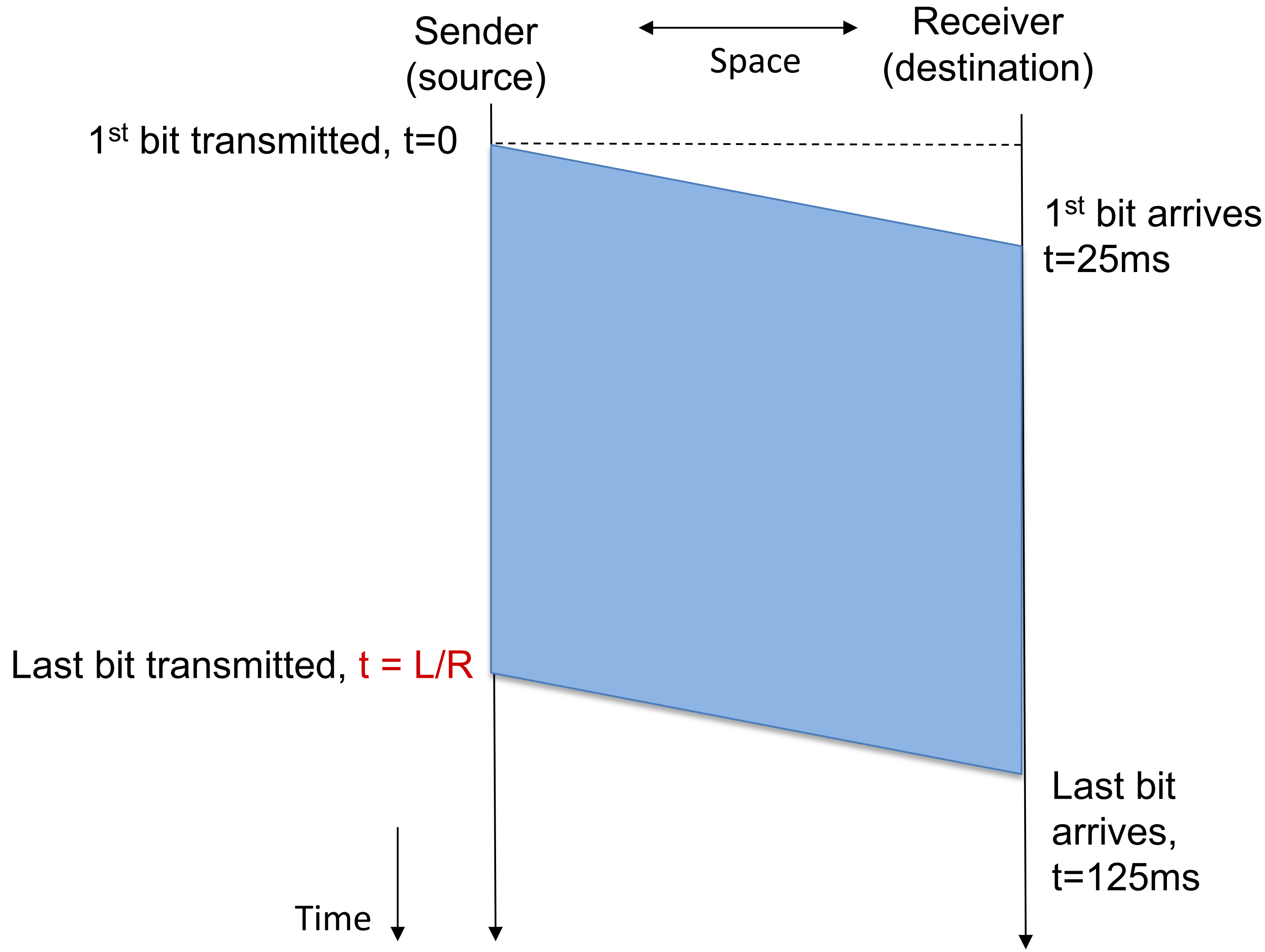
- d: length of physical link
- s: propagation speed (~2×10^8 m/sec)
- $d_{\text{prop}} = d/s$

d_{trans} and d_{prop} *very* different

- http://gaia.cs.umass.edu/kurose_ross/interactive/end-end-delay.php
- https://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198700.cw/index.html

Space-Time Diagram

Transfer of a $L=10^4$bits (1250 B) packet on a 5000 km link (with propagation speed=$2x10^8$m/s) at transmission rate (capacity / bandwidth) of $R=10^5$b/s



Outline

- Fundamentals
- Applications
  - Application architecture, transport protocols services and API (socket), Web (REST API), streaming, CDN, P2P
- Transport Layer

Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Processes communicating

process: program running within a host

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

clients, servers

client process: process that initiates communication

server process: process that waits to be contacted

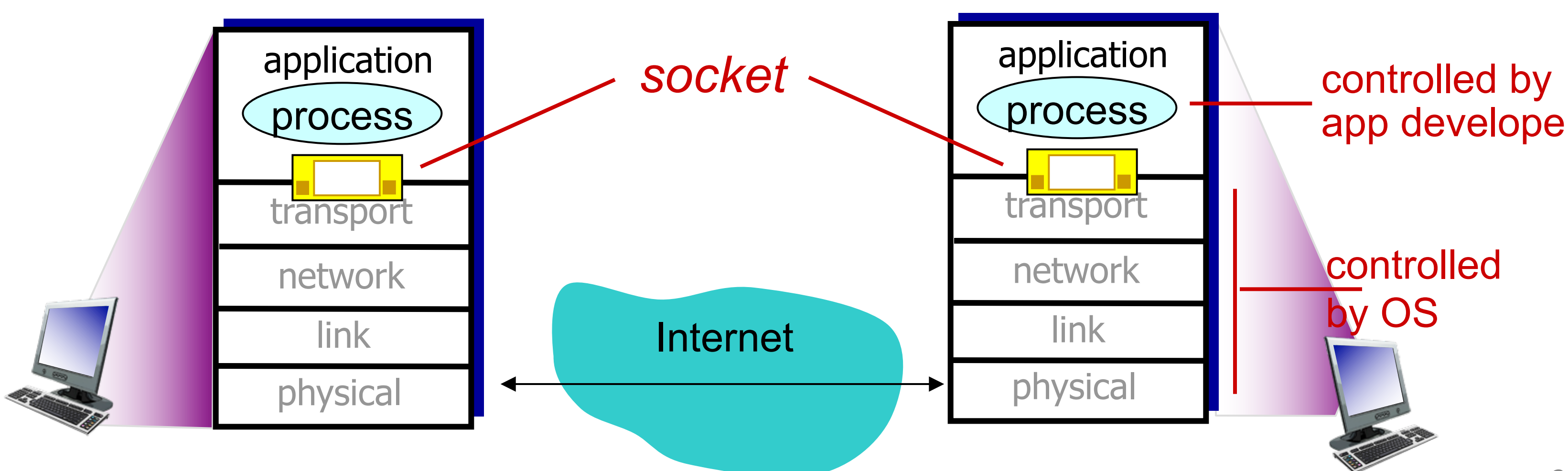
- aside: applications with P2P architectures have client processes & server processes

Addressing processes

- to receive messages, process must have *identifier*
- *identifier* includes (type of transport protocol), IP address, and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door (*a door between a factory's manufacturing area & its packaging area*)
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Internet transport protocols services

TCP service:

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

UDP service:

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

why is there a UDP?

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

Socket: UDP vs. TCP

- Socket establishment

UDP client

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

TCP client

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
```

UDP server

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
```

TCP server

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
```

Socket: UDP vs. TCP

- Use of socket

UDP client

```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

TCP client

```
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
```

UDP server

```
message, clientAddress =
serverSocket.recvfrom(2048)
serverSocket.sendto(message, clientAddress)
```

TCP server

```
connectionSocket, addr = serverSocket.accept()
message = connectionSocket.recv(1024)
connectionSocket.send(message)
```

App-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
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

Web: URI, URL, and URNs

- Terminology:
  - URI = Uniform Resource Identifier
  - URL = Uniform Resource Locator
  - URN = Uniform Resource Name
- URNs and URLs are both URIs
- What is the difference?
  - URI: Uniquely identifies a resource
  - URL: identifies + provides location of resource
  - URN: identifies but not locates
    - For example, in the International Standard Book Number (ISBN) system, ISBN 0-486-27557-4 identifies a specific edition of Shakespeare's play Romeo and Juliet. The URN for that edition would be urn:isbn:0-486-27557-4.

- URI generic syntax (finalized in RFC 3986, 2005)
  - scheme:[//authority]path[?query][#fragment]

Idempotence and safe methods

- HTTP uses concepts of idempotence and safe methods
  - Safe methods enable caching and load distribution
  - Idempotence allows to handle lost confirmations by re-sending
- Safe methods will not modify (non-trivial) resources on server
  - Example: GET and HEAD do not change the resource on server
- Idempotent methods may modify resources on the server
  - But can be executed multiple times without changing outcome
  - Example: duplicate DELETE operations have no additional effect
- POST is *not* idempotent, multiple POSTs have multiple effects
  - Example: multiple rooms are created for POST /rooms

Web API: REST

- REpresentational State Transfer
  - Roy Thomas Fielding Ph.D. thesis 2000 https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- An architectural style, not a protocol
- Request (client -> server): to get / update the state of a specified resource
  - Uniform Resource Identifier (URI)
  - The verb specified by standard HTTP methods: GET, PUT, POST, PATCH, DELETE
- Respond (server->client): representation(s) of the (current) state of the resource
  - State often represented in JSON / XML

Resources in REST

- Two types of resources
  - Collections: /rooms
    - Container, referencing other things
  - Instances: /rooms/2.506
    - Single instance
- Resources are referenced in the HTTP header
  - GET /rooms/2.506 HTTP/1.1

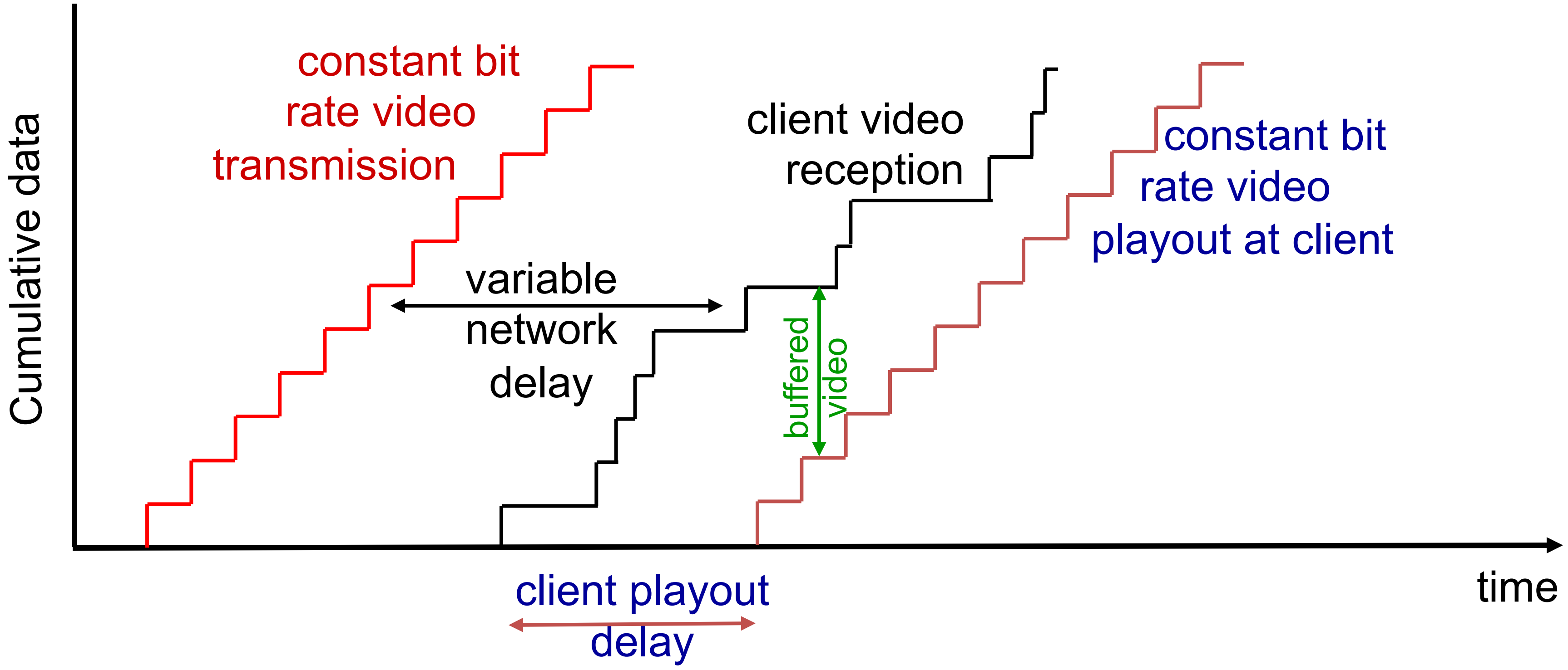
PUT vs POST

- Normally, POST is used to create new resources (and get ID), PUT is used to update
- POST can be used to create an element in a collection, without explicit name
  - Server will reply with 201 message with URL of created element
- PUT can be used to update an existing resource
  - Reply will be 200

Multimedia networking: 3 application types

- *streaming, stored* audio, video
  - *streaming:* can begin playout before downloading entire file
  - *stored (at server):* can transmit faster than audio/video will be rendered (implies storing/buffering at client)
  - e.g., YouTube, Netflix, Hulu
- *conversational* voice/video over IP
  - interactive nature of human-to-human conversation limits delay tolerance
  - e.g., Skype
- *streaming live* audio, video
  - e.g., live sporting event (futbol)

Streaming stored video



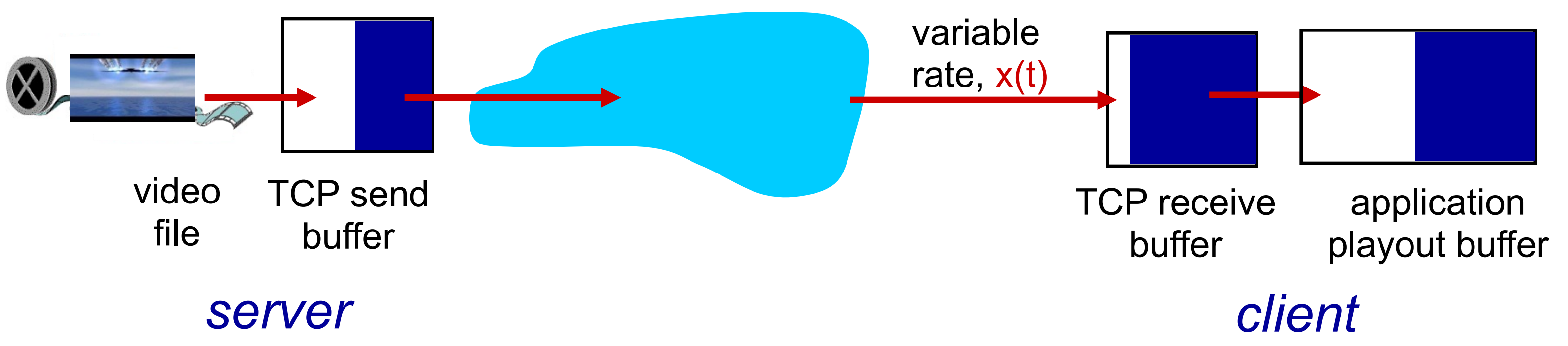
- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

Streaming multimedia: UDP

- server sends at rate appropriate for client
  - often: send rate = encoding rate = playback rate
  - send rate can be oblivious to congestion levels
- short playout delay (2-5 seconds) to remove network jitter
- error recovery: application-level, time permitting
- Real-time Transport Protocol (RTP) [RFC 2326]: multimedia payload types
- UDP may *not* go through firewalls

Streaming multimedia: HTTP

- multimedia file retrieved via HTTP GET
- send at maximum possible rate under TCP



- fill rate fluctuates due to TCP congestion control, retransmissions (in-order delivery)
- larger playout delay: smooth TCP delivery rate
- HTTP/TCP passes more easily through firewalls

Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
  - Other adaptive solutions: Apple's HTTP Live Streaming (HLS) solution, Adobe Systems HTTP Dynamic Streaming, Microsoft Smooth Streaming
- *Server:*
  - encodes video file into multiple versions
  - each version stored, encoded at a different rate
  - *manifest file:* provides URLs for different versions
- *Client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate (highest quality version) sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at the time)

Voice-over-IP (VoIP)

- *VoIP end-end-delay requirement*: needed to maintain "conversational" aspect
  - higher delays noticeable, impair interactivity
  - < 150 msec: good
  - > 400 msec: bad
  - includes application-level (packetization, playout), network delays

Adaptive playout delay (1)

- *goal:* low playout delay, low late loss rate
- *approach:* adaptive playout delay adjustment:
  - estimate network delay, adjust playout delay at beginning of each talk spurt
  - silent periods compressed and elongated
  - chunks still played out every 20 msec during talk spurt
- adaptively estimate packet delay: (EWMA - exponentially weighted moving average):

$$d_i = (1-\alpha)d_{i-1} + \alpha\,(r_i - t_i)$$

delay estimate after ith packet

small constant, e.g. 0.1

time received - time sent (timestamp)

measured delay of ith packet

Adaptive playout delay (2)

- also useful to estimate average deviation of delay, v_i :

$$v_i = (1-\beta)v_{i-1} + \beta\,|r_i - t_i - d_i|$$

- estimates d_i, v_i calculated for every received packet, but used only at start of talk spurt

- for first packet in talk spurt, playout time is:

$$playout\text{-}time_i = t_i + d_i + Kv_i$$

- remaining packets in talkspurt are played out periodically

VoIP: recovery from packet loss

Challenge: recover from packet loss given small tolerable delay between original transmission and playout

- each ACK/NAK takes ~ one RTT
- alternative: *Forward Error Correction (FEC)*
  - send enough bits to allow recovery without retransmission

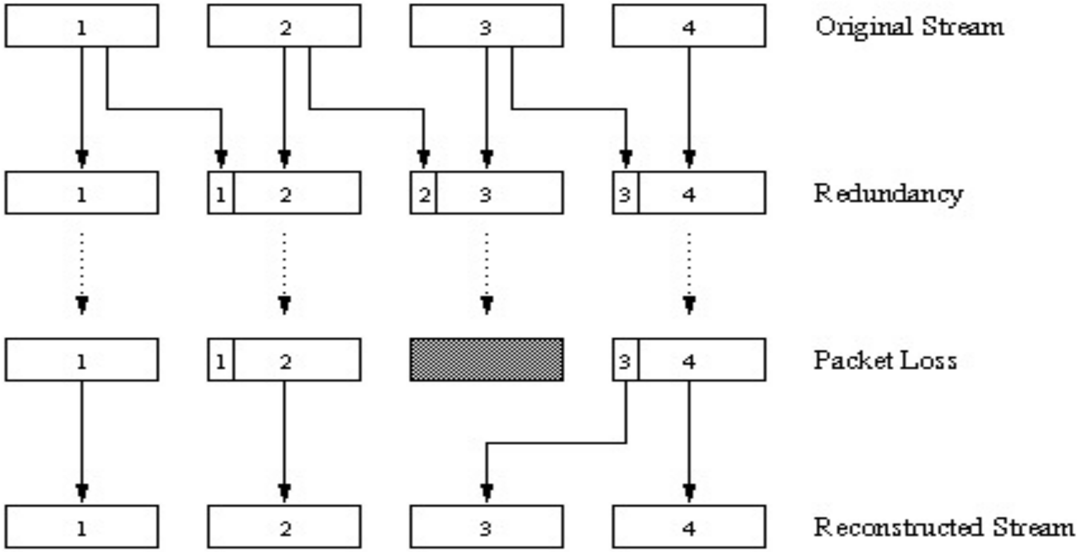
simple FEC

- for every group of *n* chunks, create redundant chunk by exclusive OR-ing *n* original chunks
- send *n+1* chunks, increasing bandwidth by factor *1/n*
- can reconstruct original *n* chunks if at most one lost chunk from *n+1* chunks
- Note: playout delay increases with *n*

VoIP: recovery from packet loss (2)

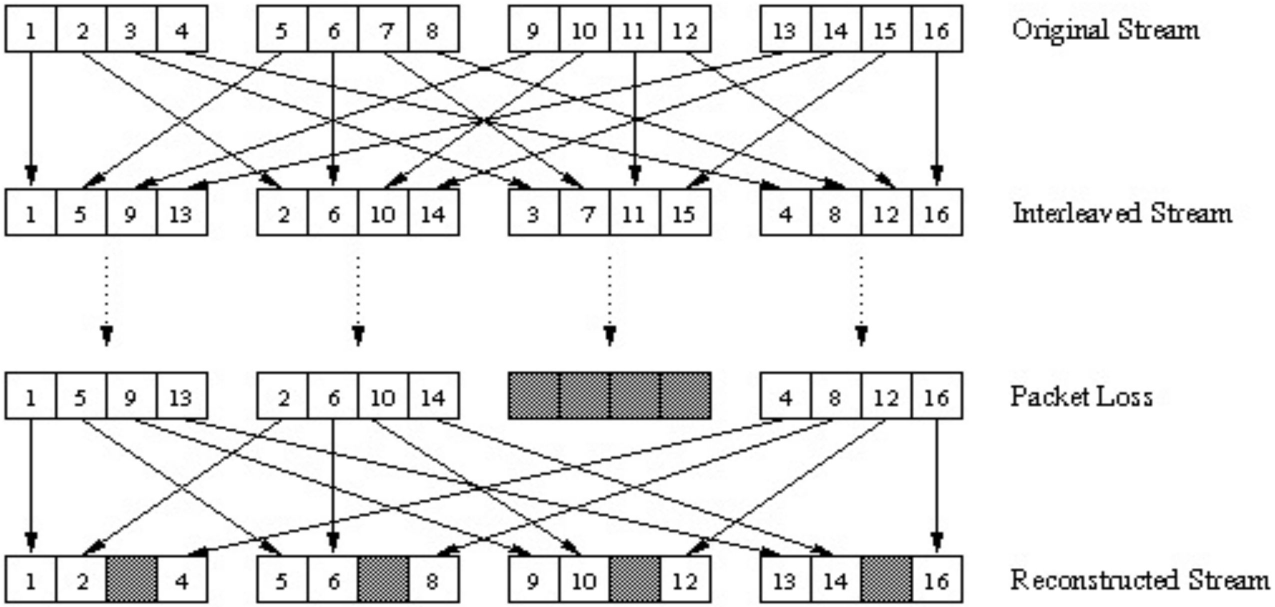
another FEC scheme:

- "piggyback lower quality stream"
- send lower resolution audio stream as redundant information
- e.g., nominal stream PCM at 64 kbps and redundant stream GSM at 13 kbps



- non-consecutive loss: receiver can conceal loss
- generalization: can also append (n-1)st and (n-2)nd low-bit rate chunk

VoIP: recovery from packet loss (3)



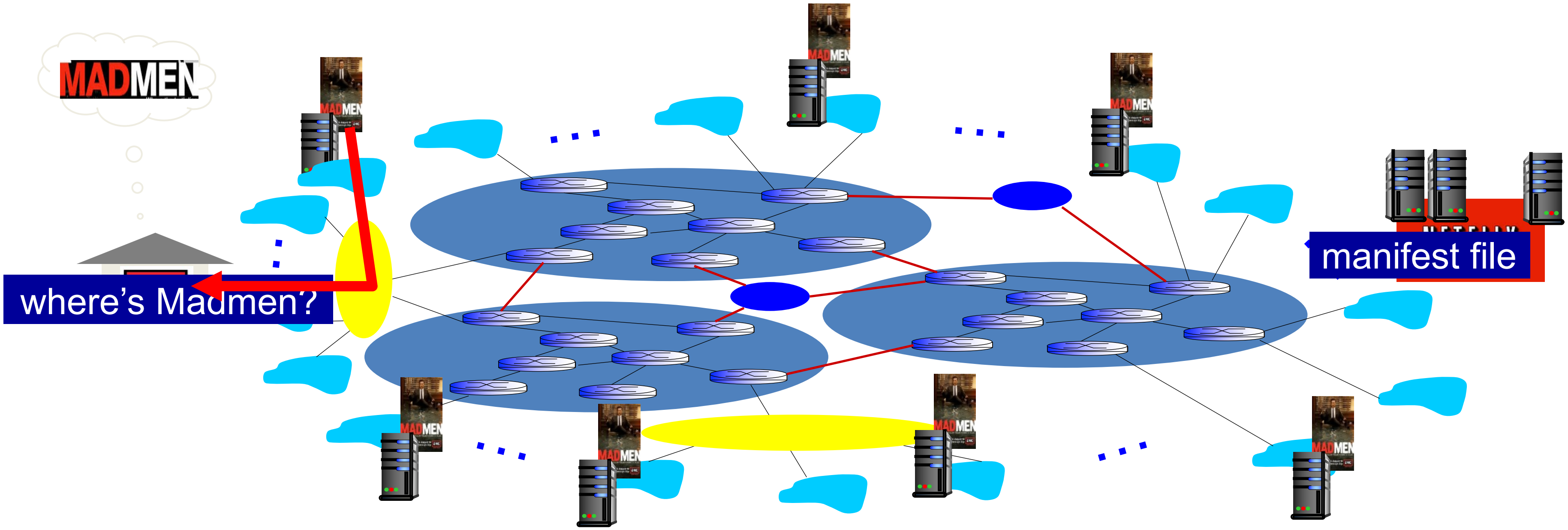
interleaving to conceal loss:

- audio chunks divided into smaller units, e.g. four 5 msec units per 20 msec audio chunk
- packet contains small units from different chunks

- if packet lost, still have *most* of every original chunk
- no redundancy overhead, but increases playout delay

Content distribution networks

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from Netflix
  - directed to nearby CDN copy (based on mapping IP address of request to geographical location), retrieves content

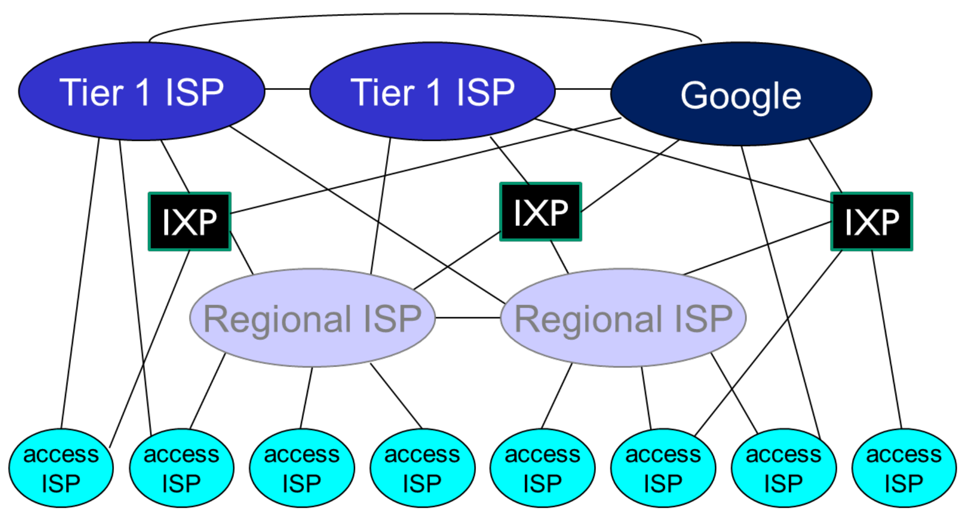


Questions for a CDN operator

- Where to put the servers?
- How to select the server?
- How to route the client to the server?
- How to replicate the content to the servers?

CDN Server Placement

- *Enter deep:* push CDN servers deep into many access ISPs
  - close to users
  - used by Akamai, 1700+ locations
- *Bring home:* smaller number (10's) of larger clusters in IXPs (Internet Exchange Points) near (but not within) access networks
  - used by Limelight



Cluster (Server) Selection

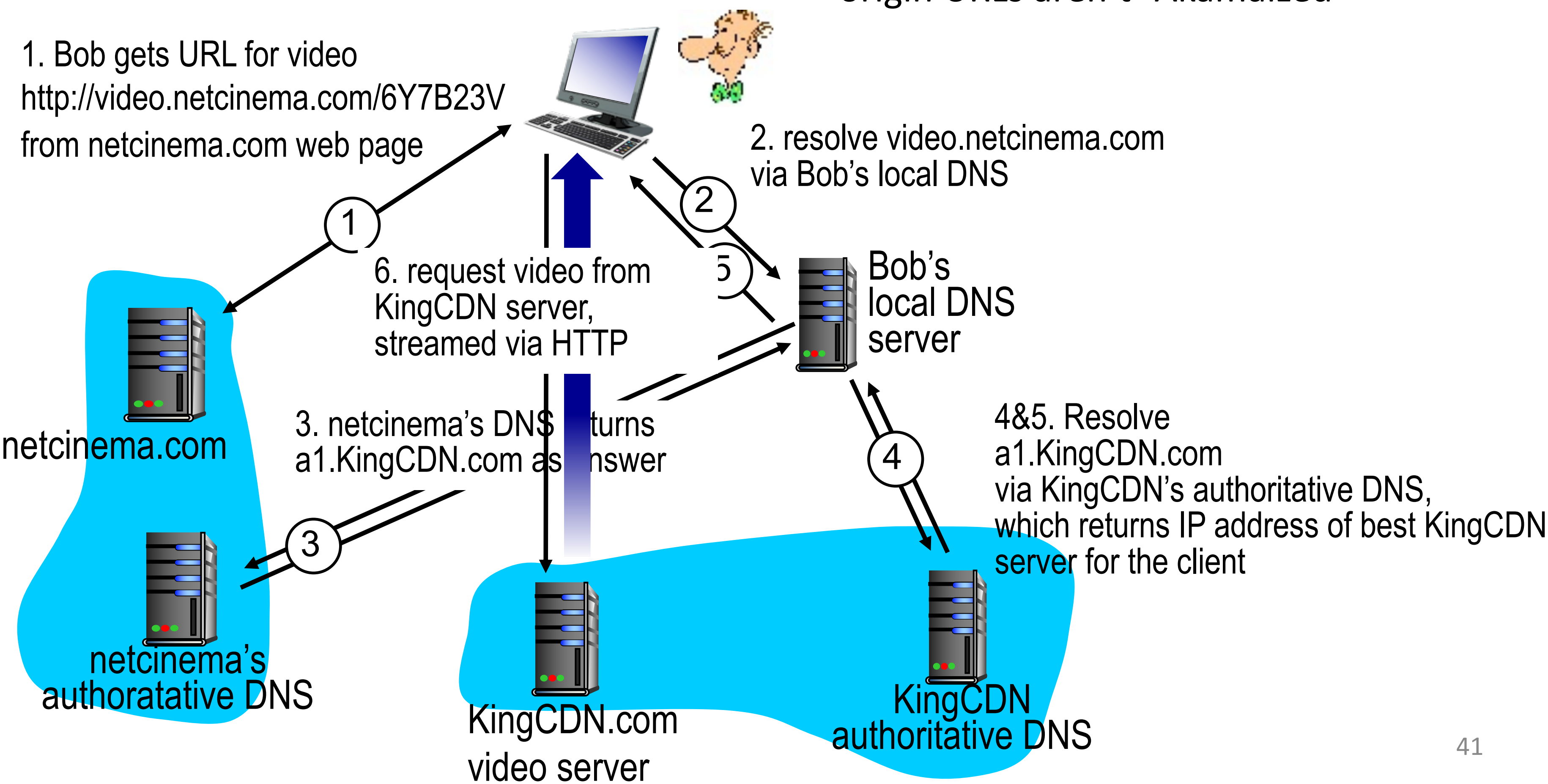
- Geographically close
- Best performance: real-time measurement
- Lowest load: Load-balancing
- Cheapest: CDN may need to pay its provider ISP
- Any alive node: fault-tolerant

CDN content access by DNS redirect

Bob (client) requests video http://video.netcinema.com/6Y7B23V

- video stored in CDN run by KingCDN.com

Used by: third-party CDNs like Akamai, if origin URLs aren't "Akamaized"



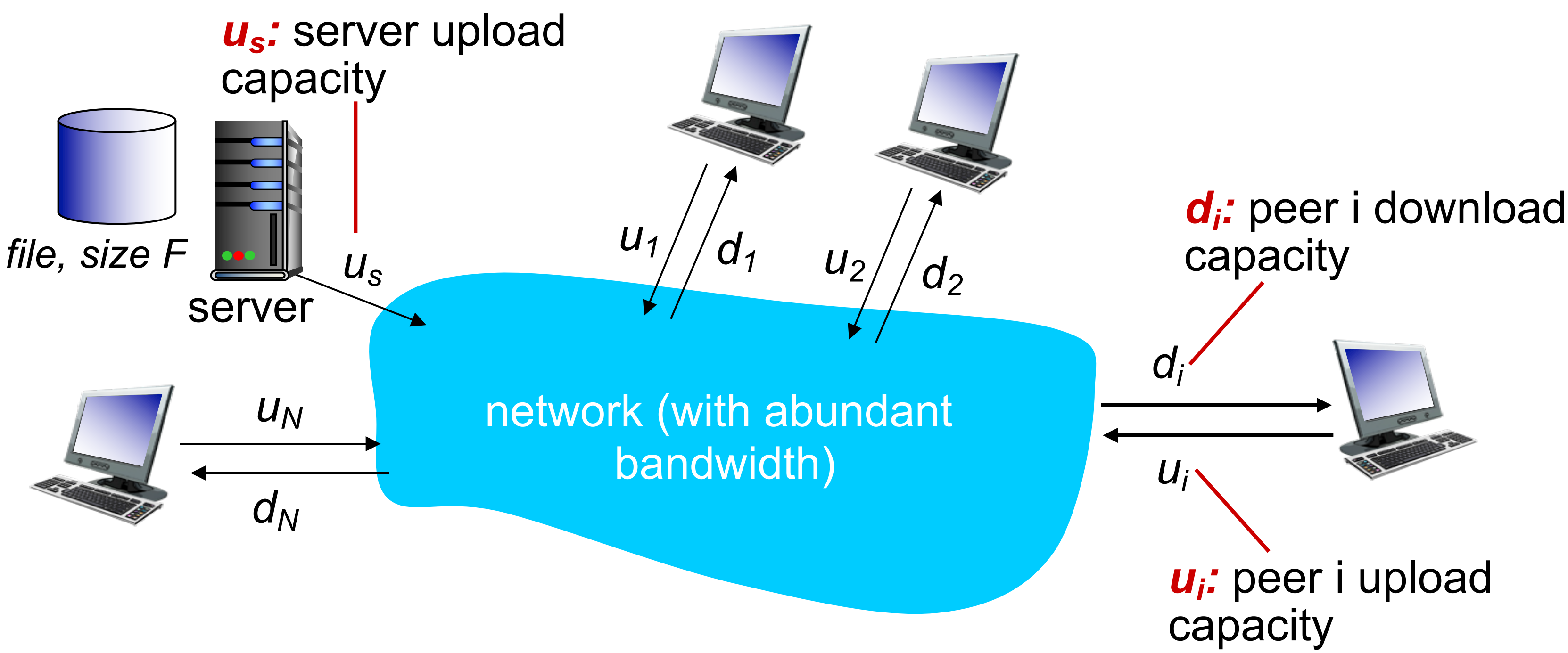
Content replication

	Mechanism	Pros.	Cons.
Netflix	Push	Use of off-peak bandwidth (Centralized) optimization	Less adaptive
Youtube	Pull	More adaptive	Initial access delay

File distribution: client-server vs P2P

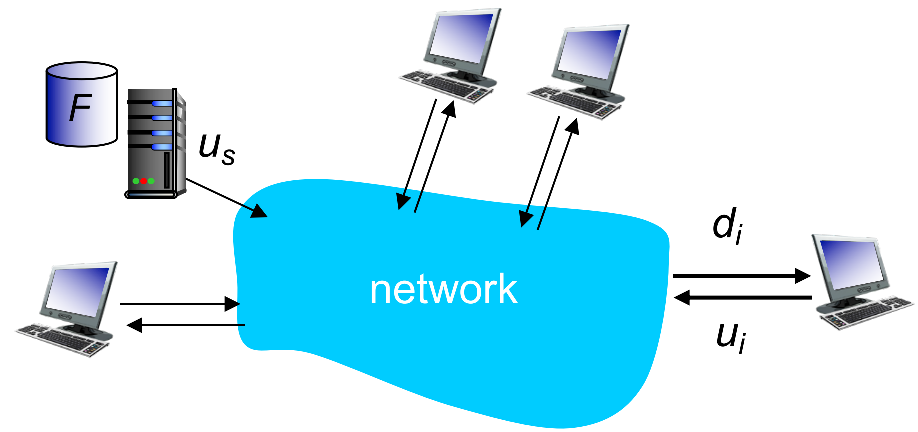
Question: How much time to distribute file (size *F*) from one server to *N* *peers*?

– peer upload/download capacity (access bandwidth) is limited resource, compared with core bandwidth



File distribution time (lower bound): client-server

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



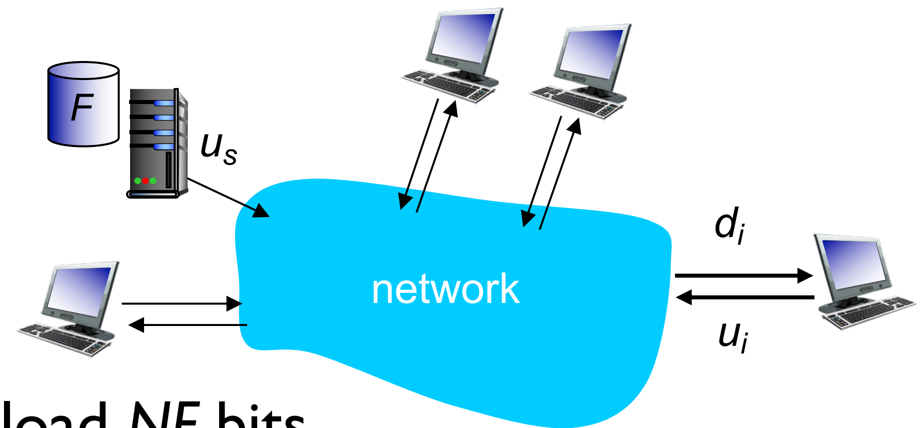
time to distribute F to N clients using client-server approach

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

increases linearly in N; susceptible to "flash crowds" particularly

File distribution time (lower bound): P2P

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



- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as aggregate must download *NF* bits
  - implies upload of same number of bits (why?)
  - max upload rate is $u_s + \Sigma u_i$ (if it's feasible to schedule the distribution to make all the uploads concurrently active)

time to distribute F to N clients using P2P approach

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

increases linearly in *N* …

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

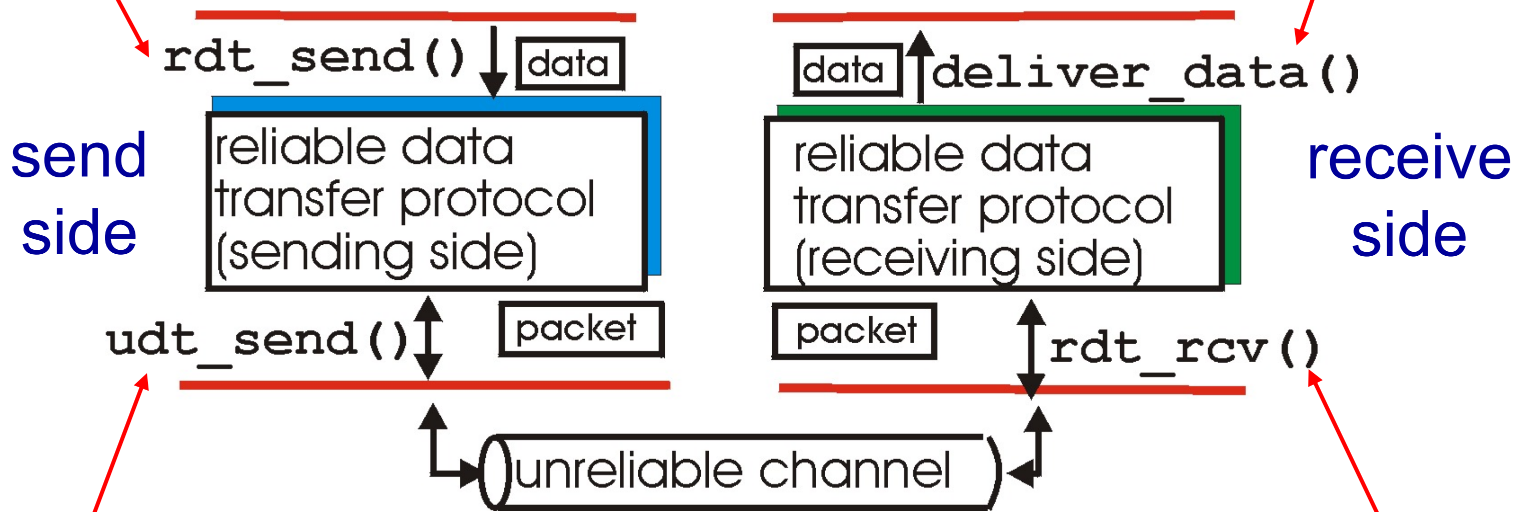
Outline

- Fundamentals
- Applications
- Transport Layer
  - reliable data transfer
  - Multiplexing, UDP
  - congestion control
  - flow control, TCP connection management,

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



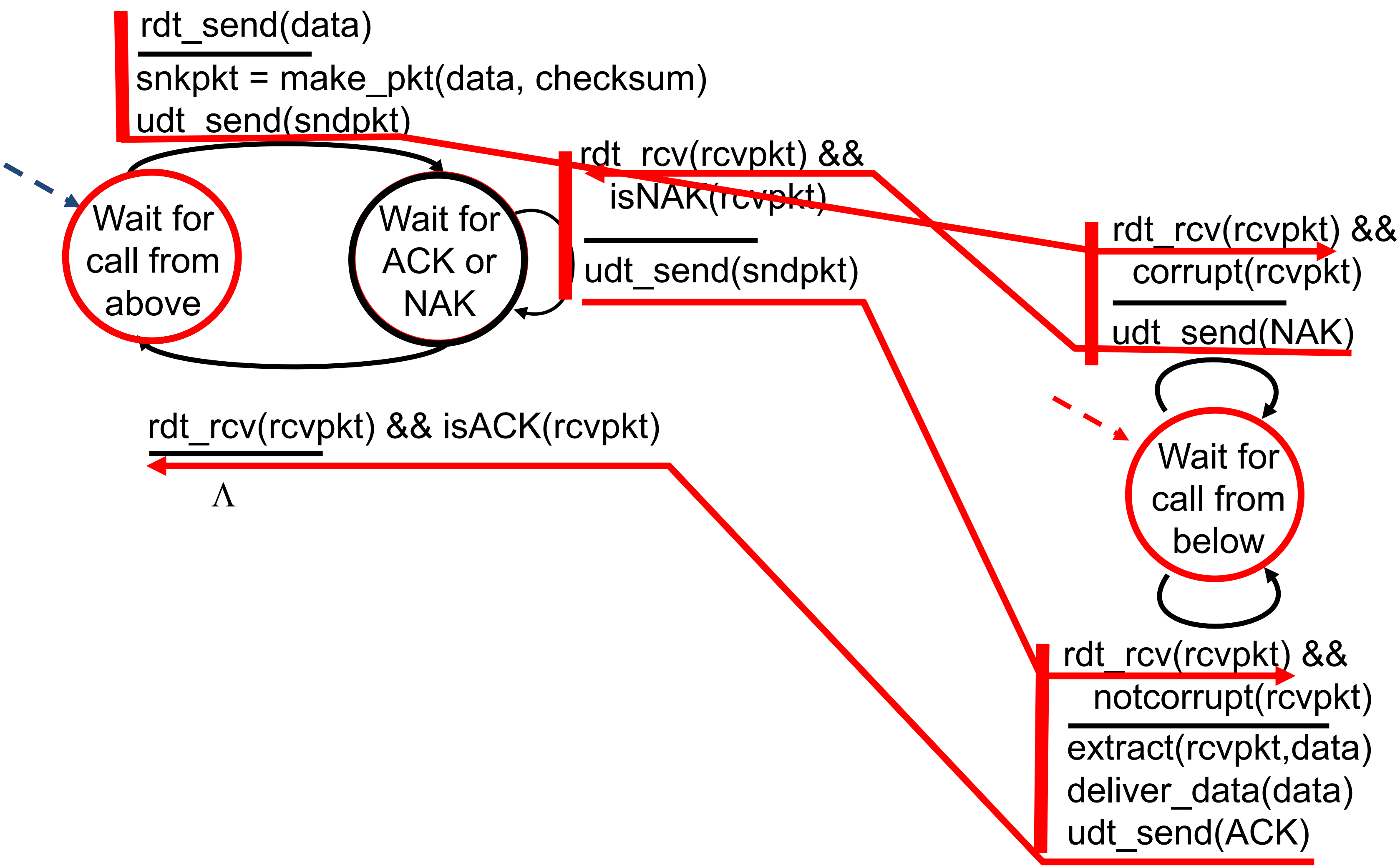
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

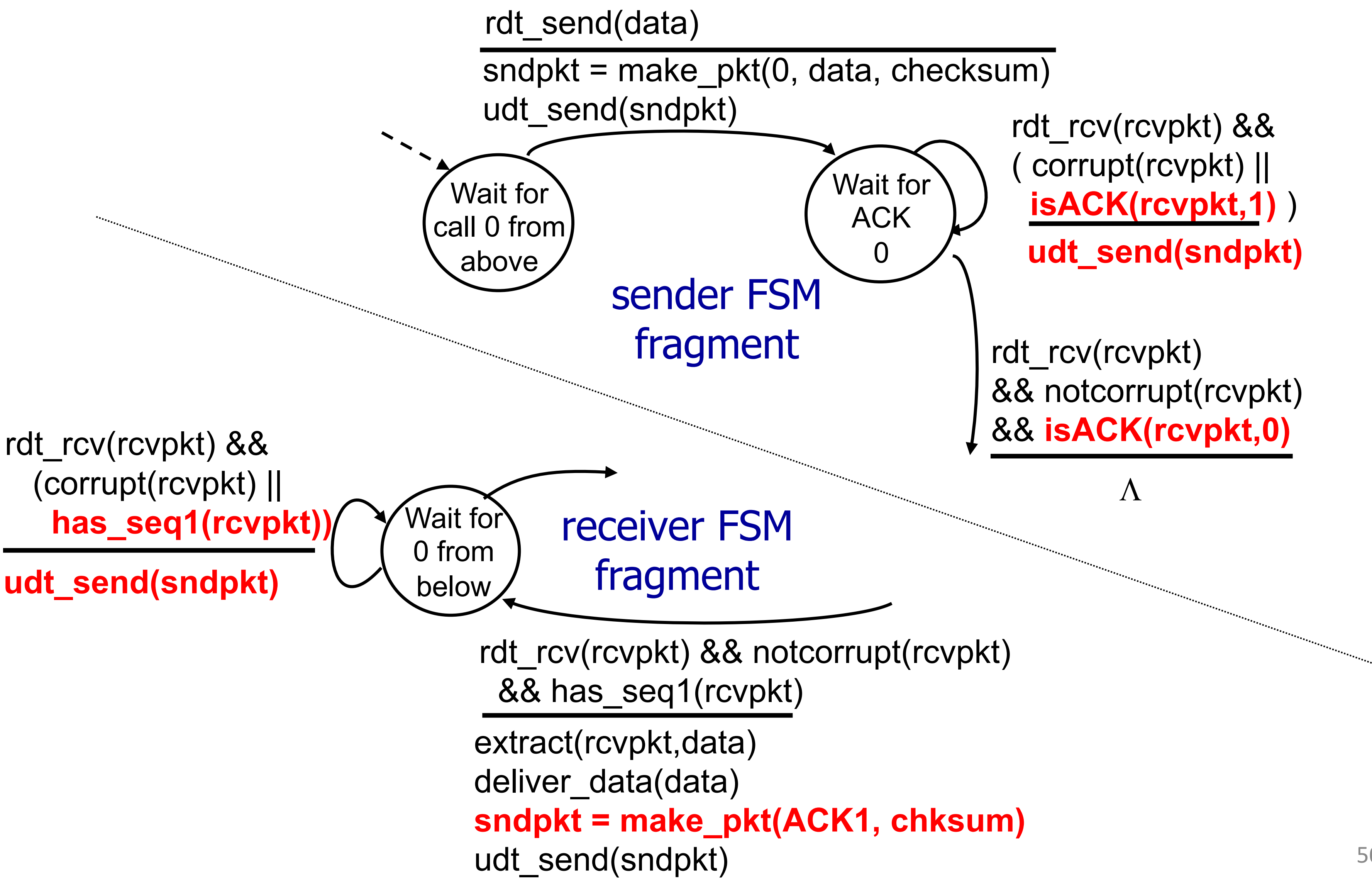
Reliable data transfer: roadmap

rdt	Assumptions about underlying channel	#State (sender/ receiver)	Mechanism
1.0	perfectly reliable	1 / 1	
2.0	Data packet corrupted with bit errors	2 / 1	Checksum ACK/NAK
2.1	Both data & ACK corrupted with bit errors	4 / 2	Seq#
2.2	Same as 2.1	4 / 2	Remove NAK
3.0	With both error and loss	4 / 2	Countdown timer

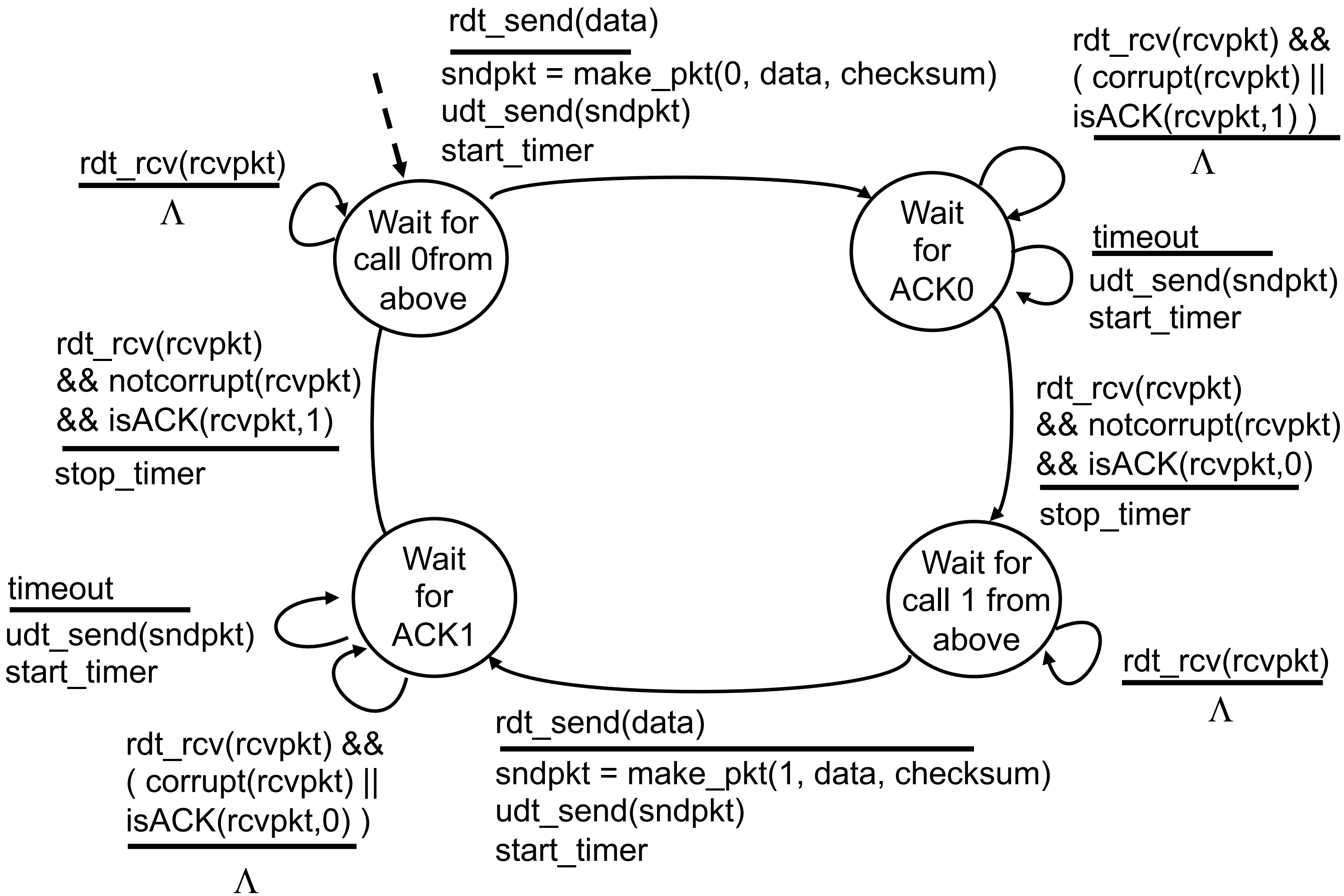
rdt2.0: error scenario



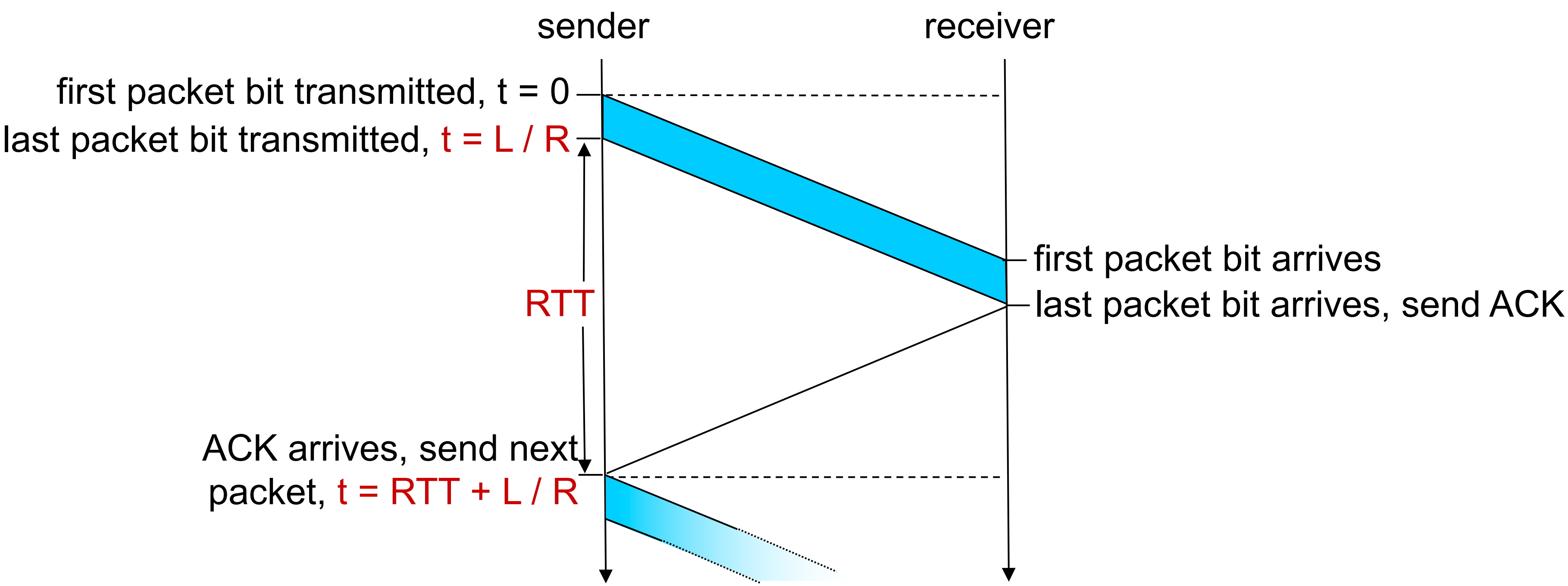
rdt2.2: sender, receiver fragments



rdt3.0 sender

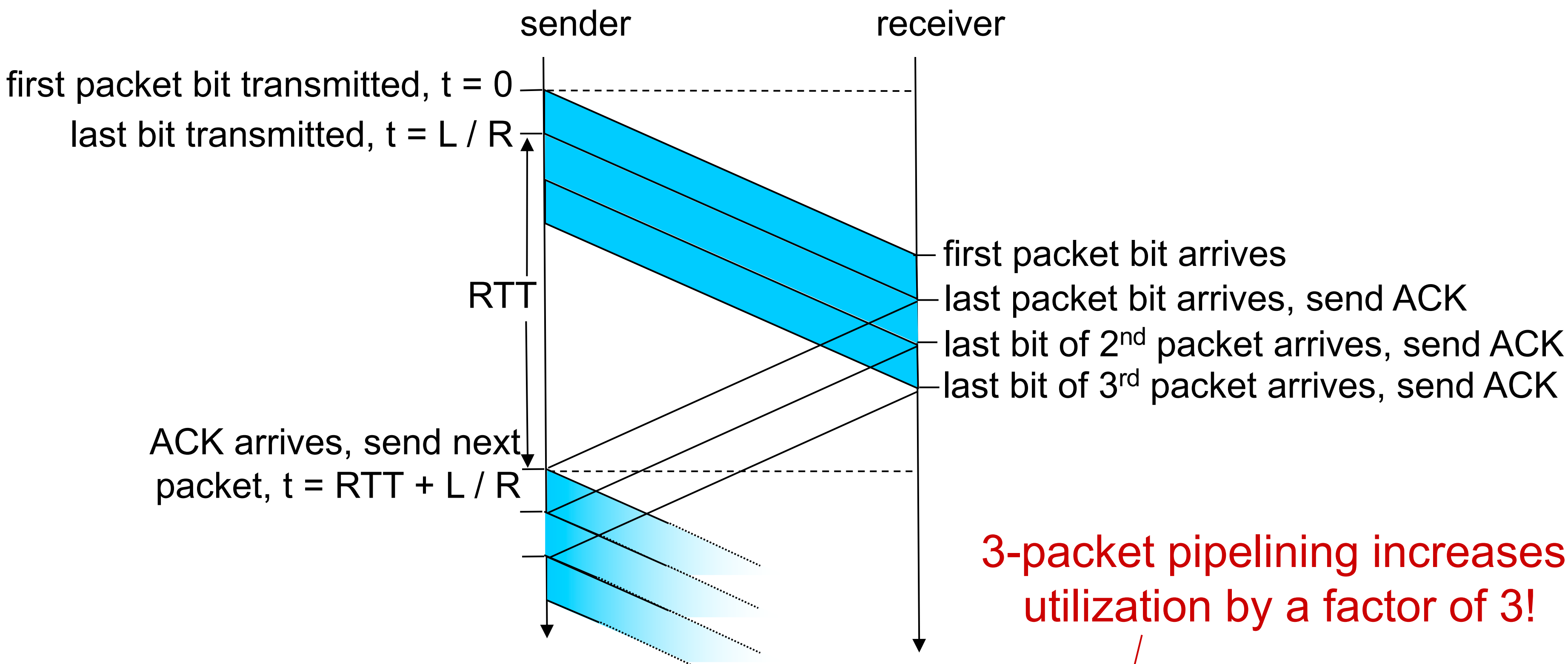


rdt3.0: stop-and-wait operation



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

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

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

Pipelined protocols: overview

Go-back-N:

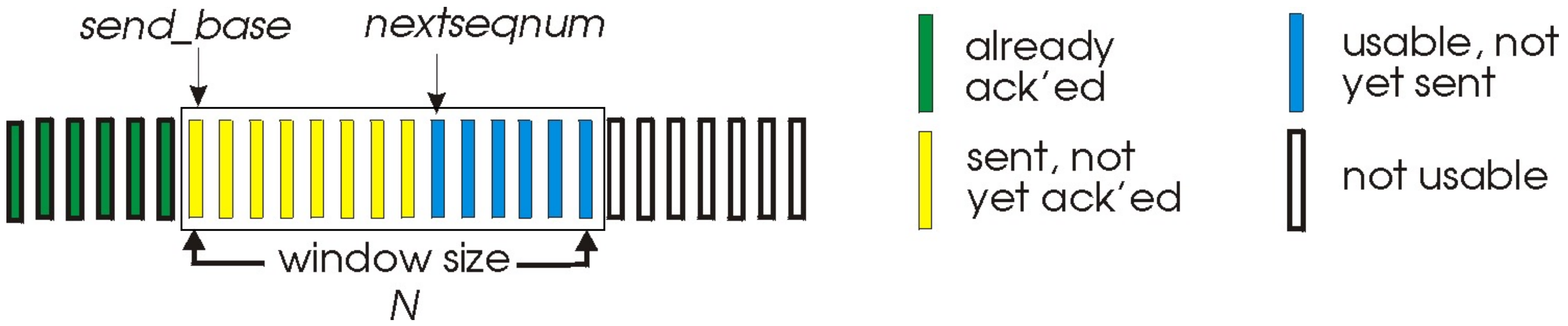
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

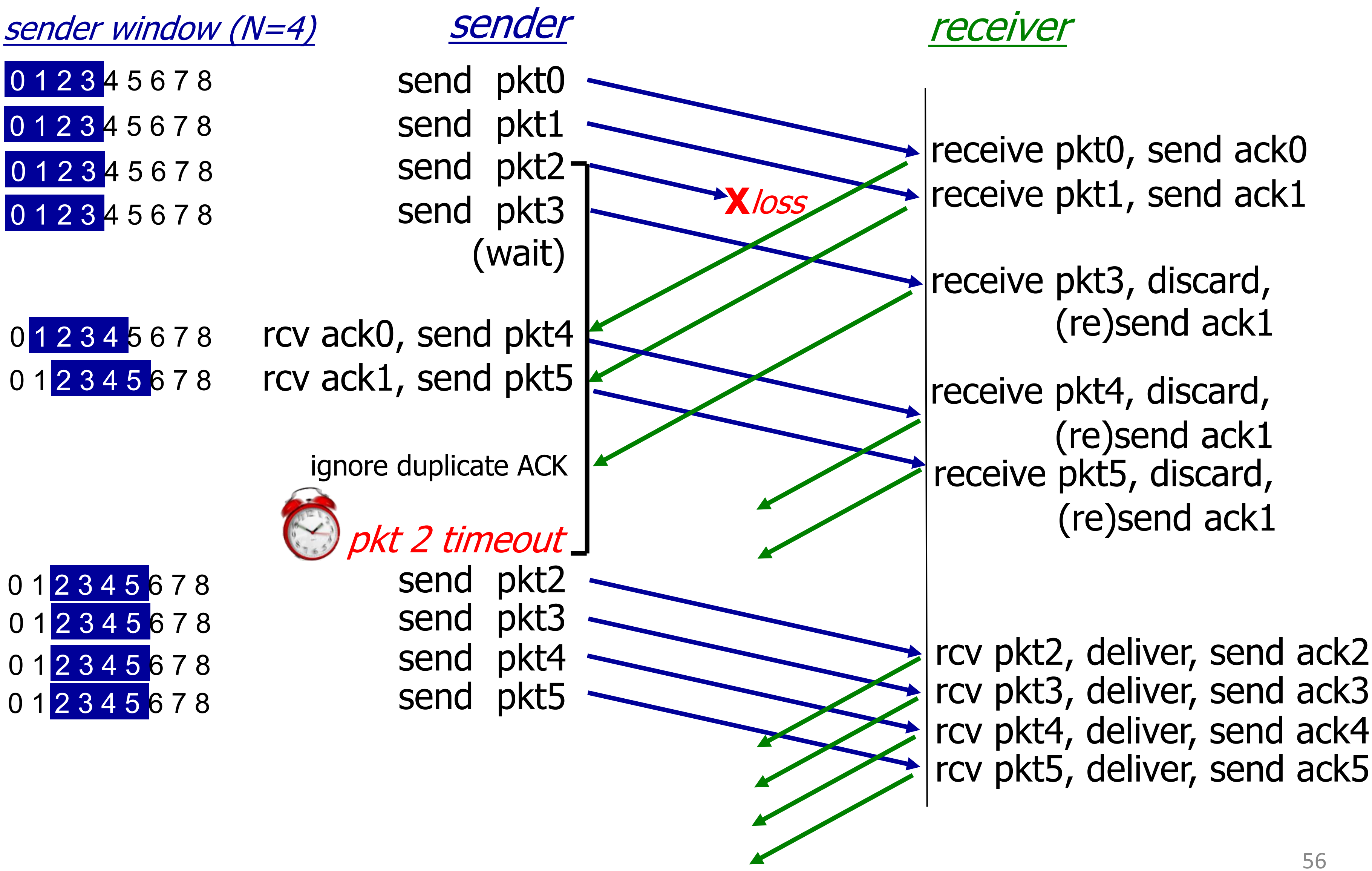
Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

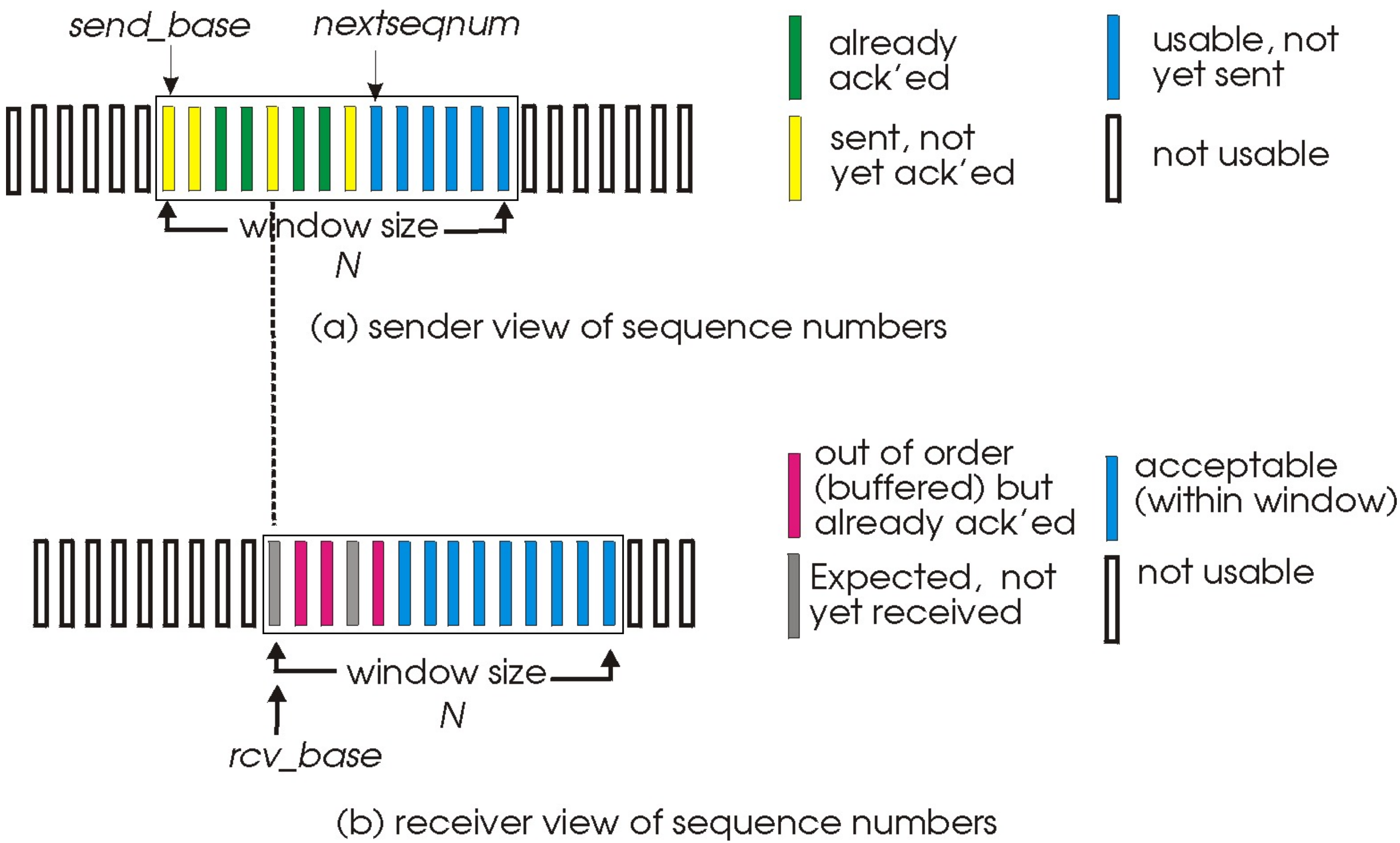
GBN in action



Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

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

receiver

pkt n in [rcvbase, rcvbase+N-1]

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

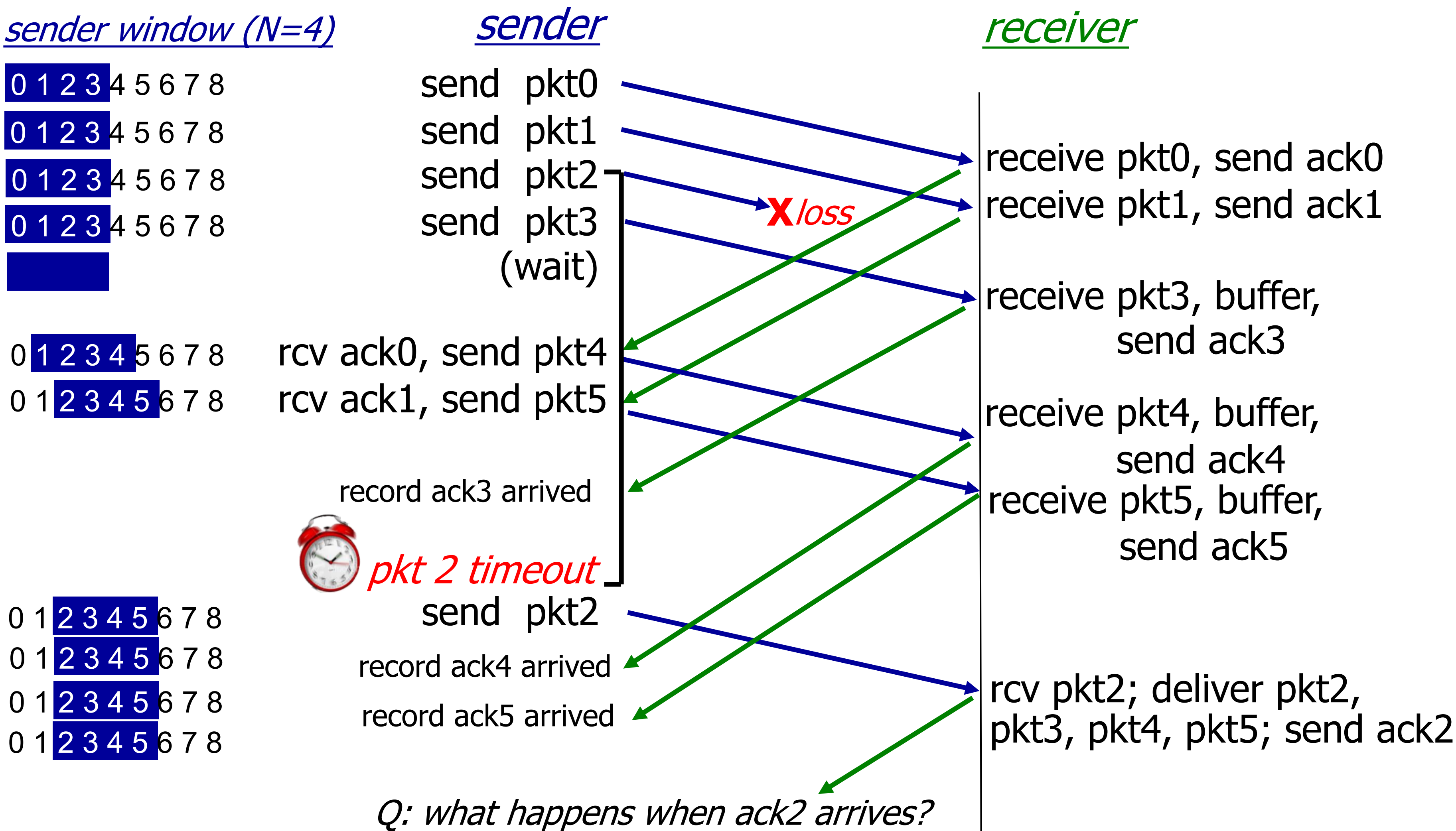
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



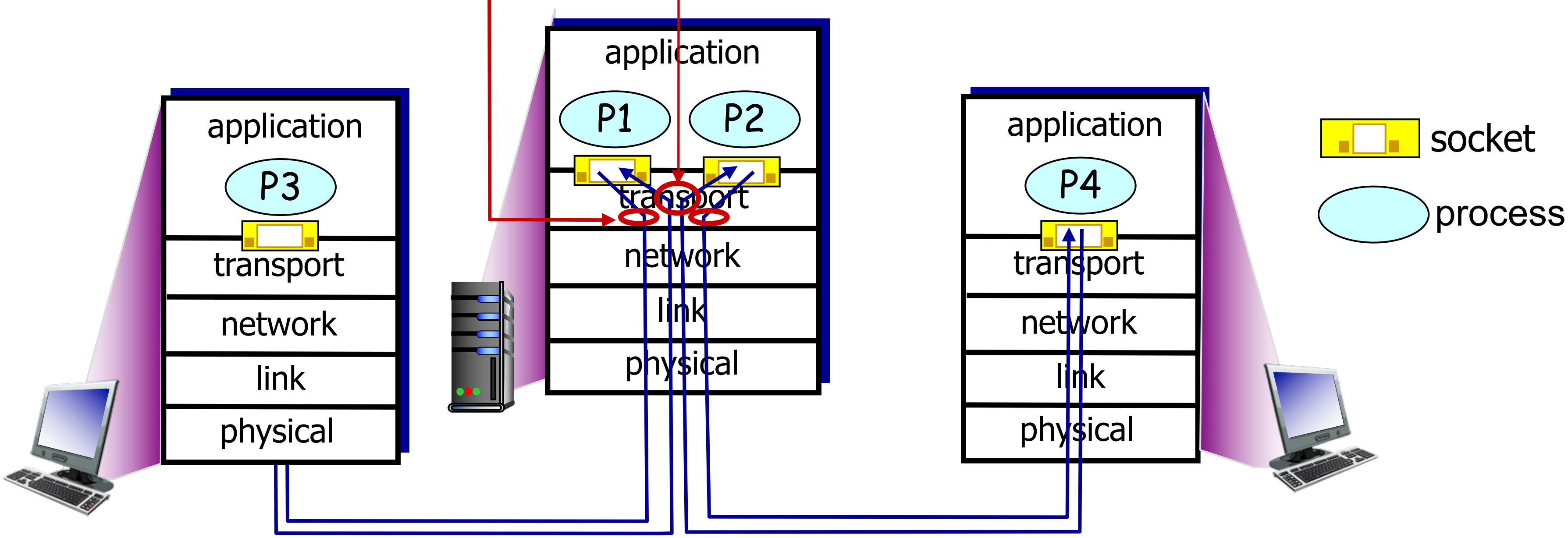
Multiplexing/demultiplexing

multiplexing at sender:

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

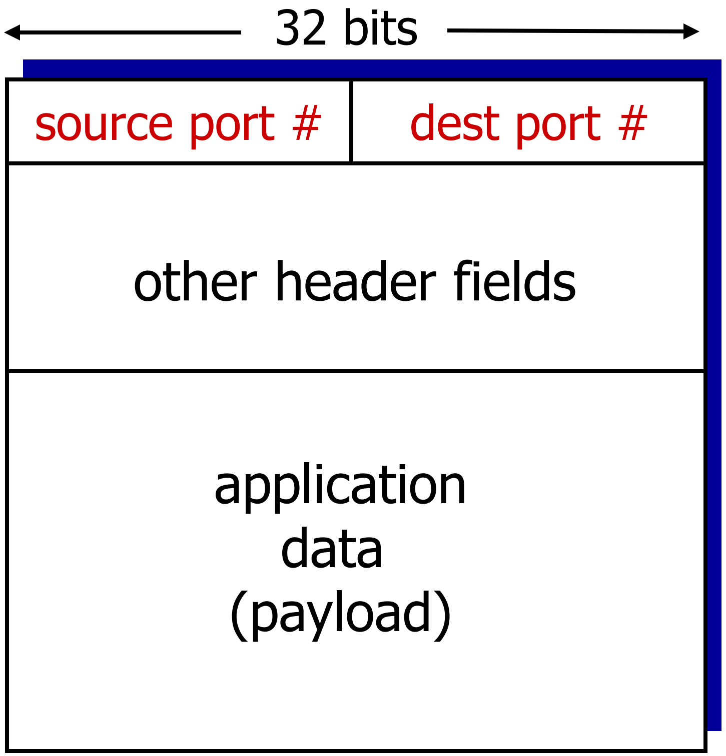
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - 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



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

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

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

Connection-oriented demux

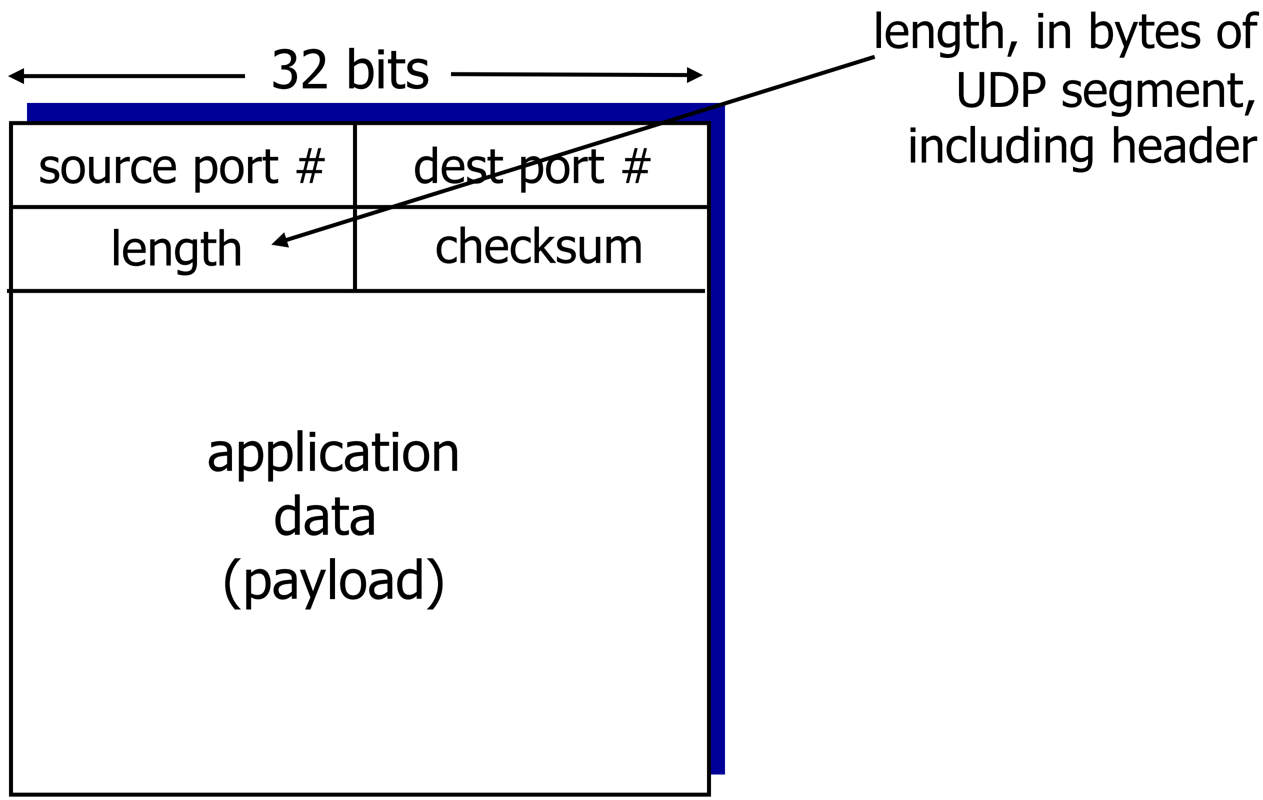
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP (remember *QUIC*):
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

UDP checksum

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

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

Internet checksum: example

example: add two 16-bit integers

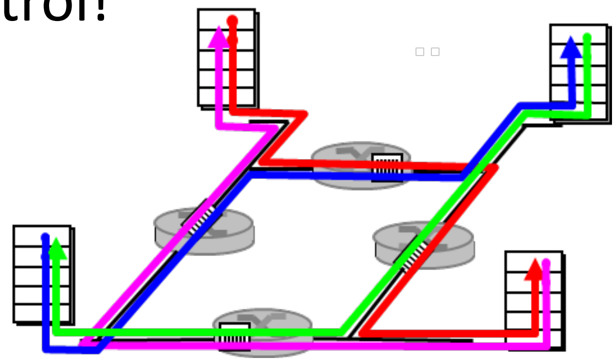
	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

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

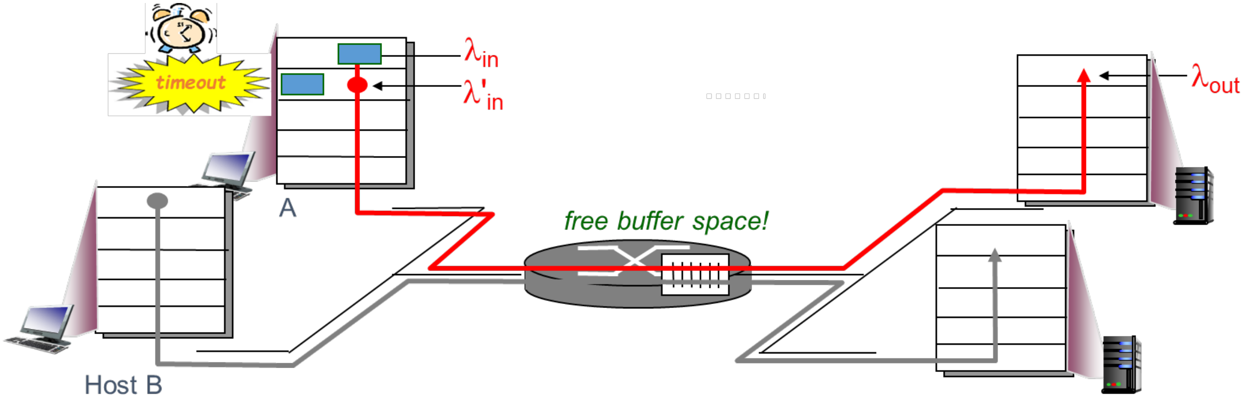
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Congestion

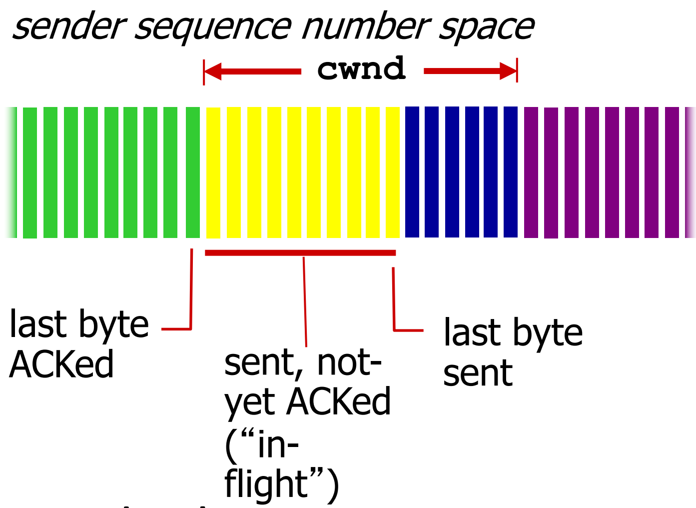
- Informally: "too many sources sending too much data too fast for *network* to handle". Different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)



- "Cost" of congestion:
  - more work (retrans) for given "goodput"
  - unneeded retransmissions: link carries multiple copies of pkt
  - when packet dropped, any "upstream" transmission capacity used for that packet was wasted!



TCP Congestion Control: details



- sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}}$$

- **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

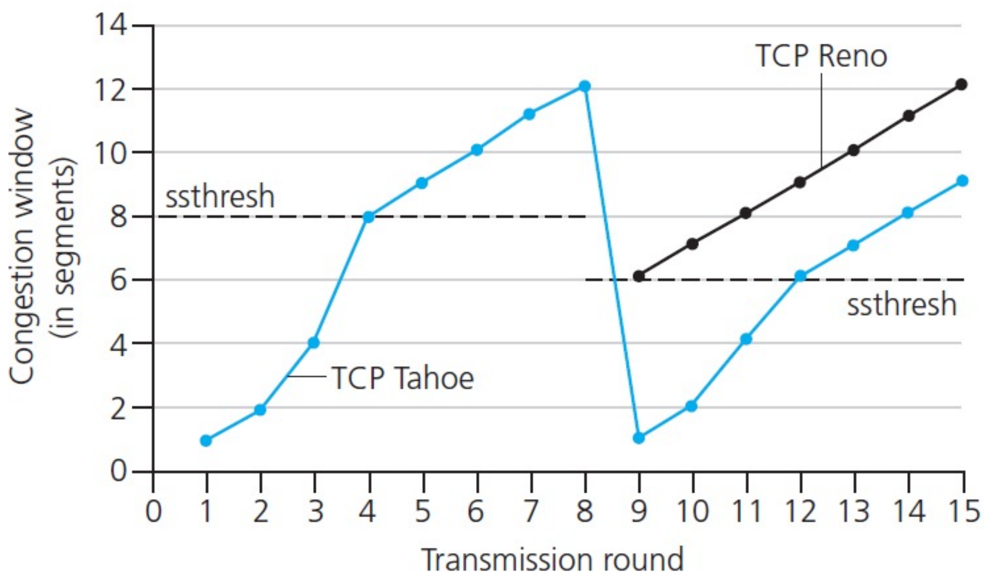
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

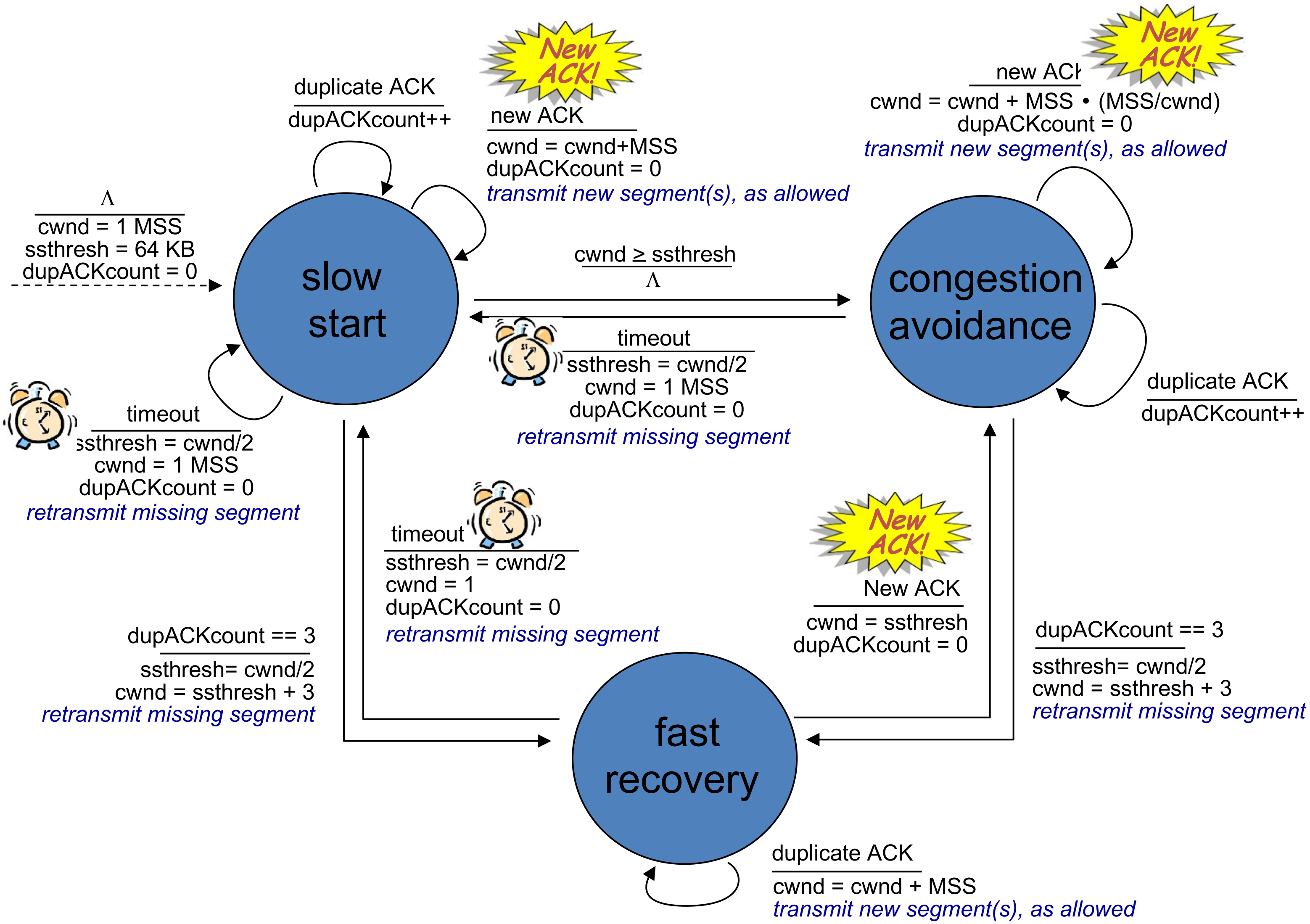


Implementation:

- 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/

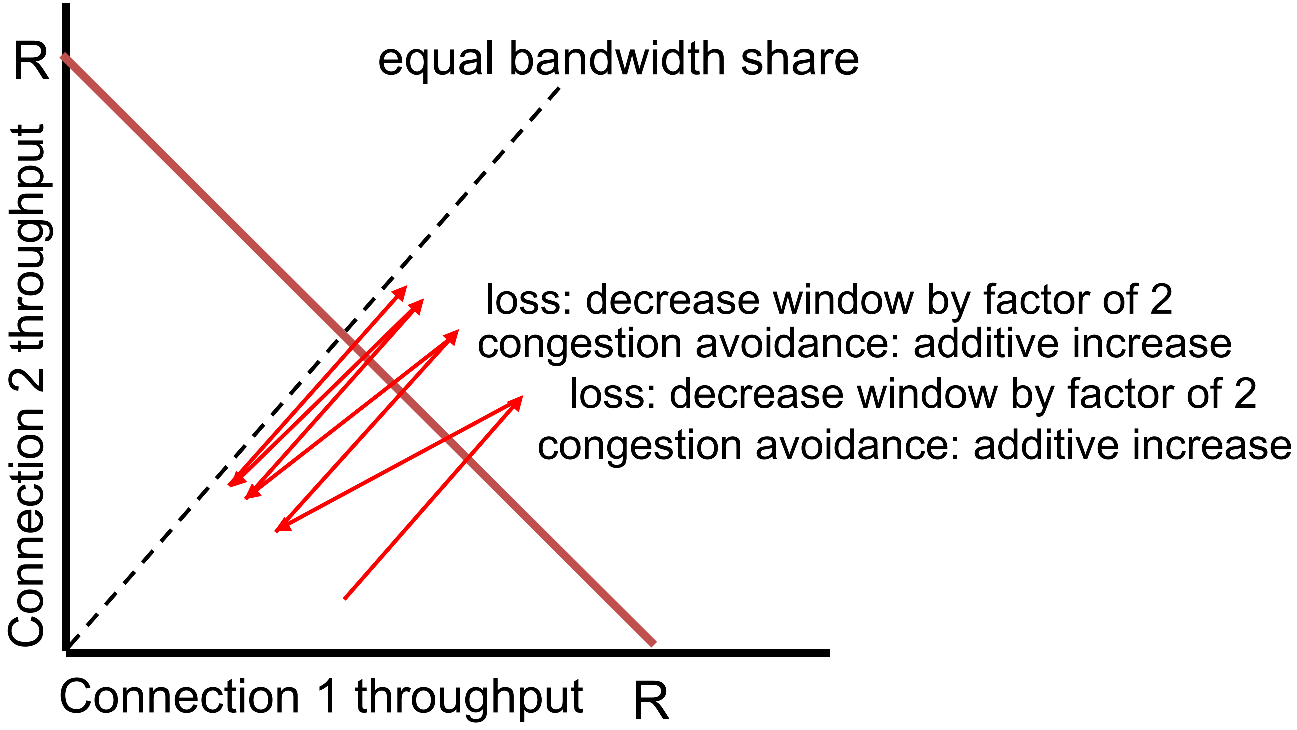
Summary: TCP Congestion Control



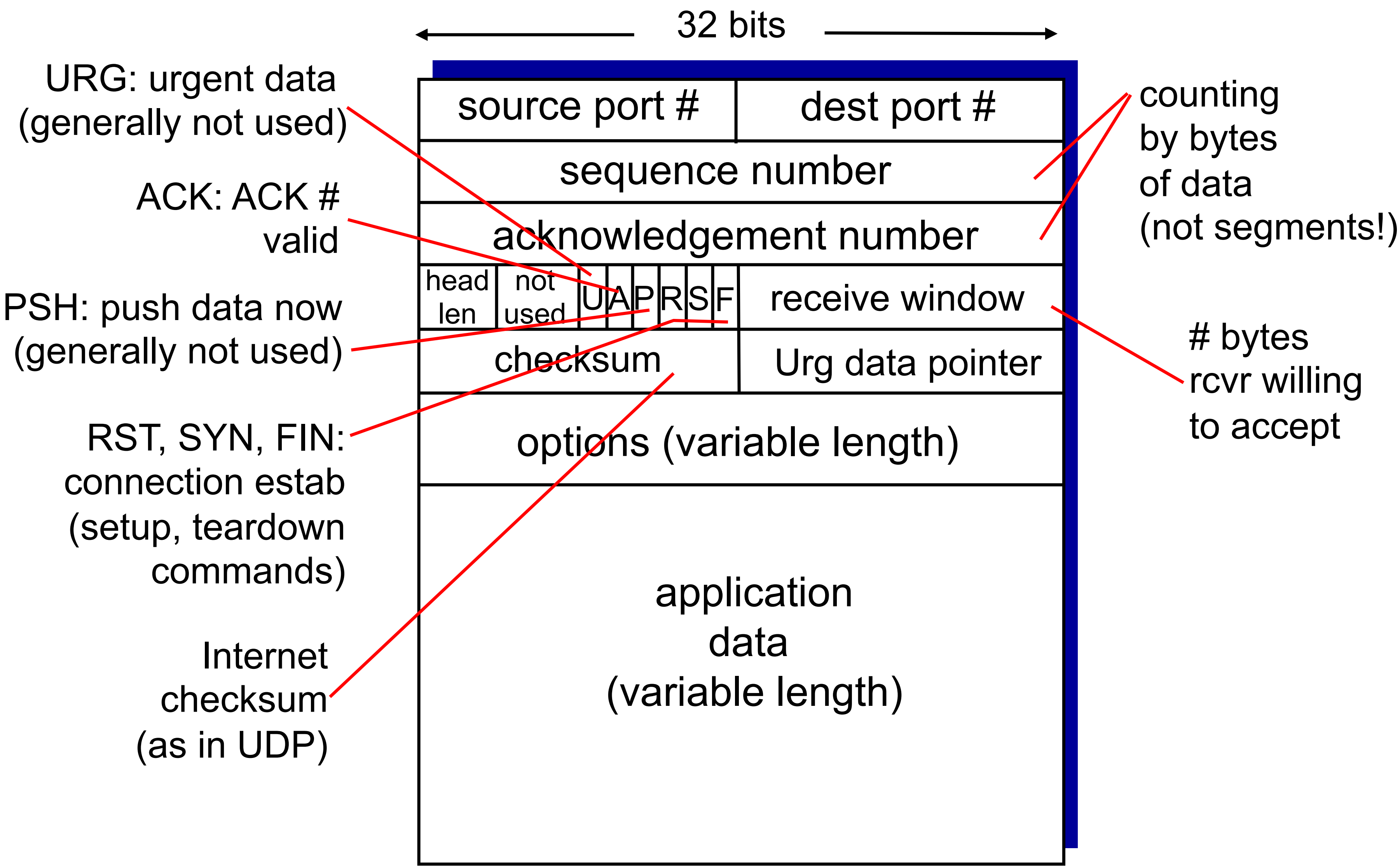
Why is TCP fair?

two competing sessions:

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



TCP segment structure



TCP seq. numbers, ACKs

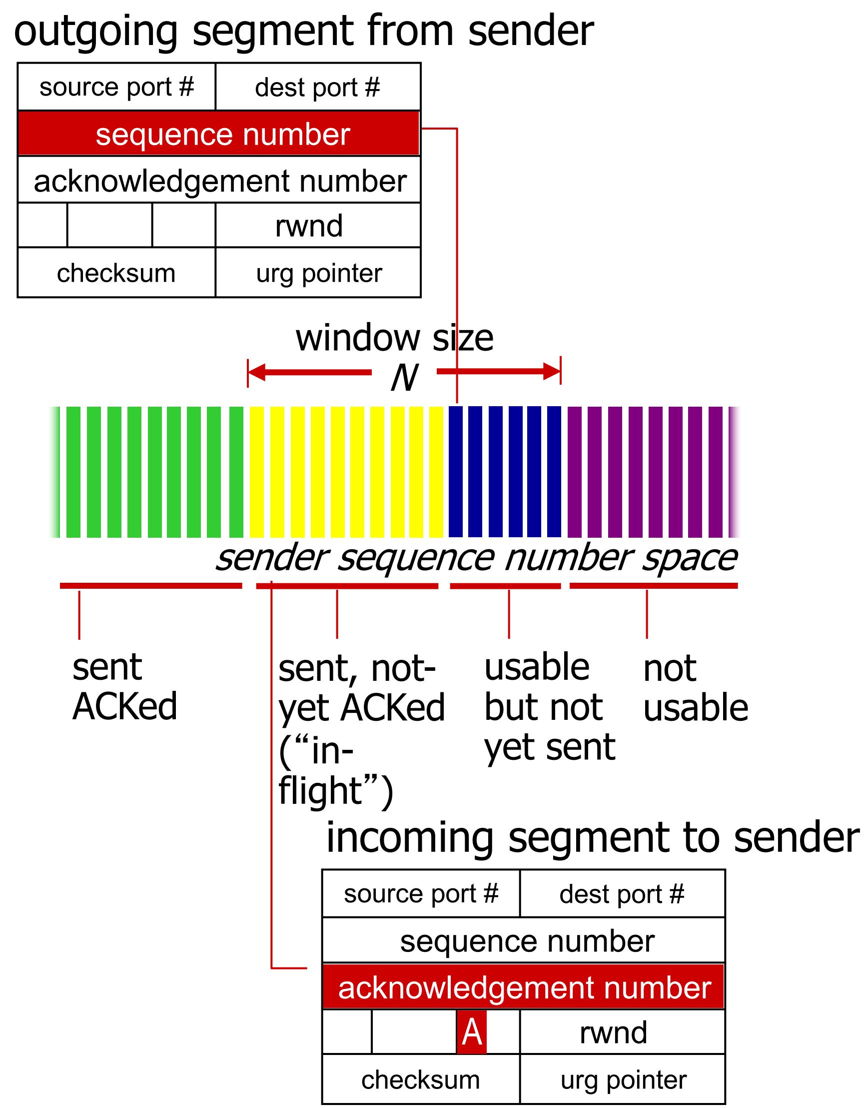
<u>sequence numbers:</u>

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

<u>acknowledgements:</u>

– seq # of next byte expected from other side

– cumulative ACK



TCP sender events:

data rcvd from app:

- 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`**

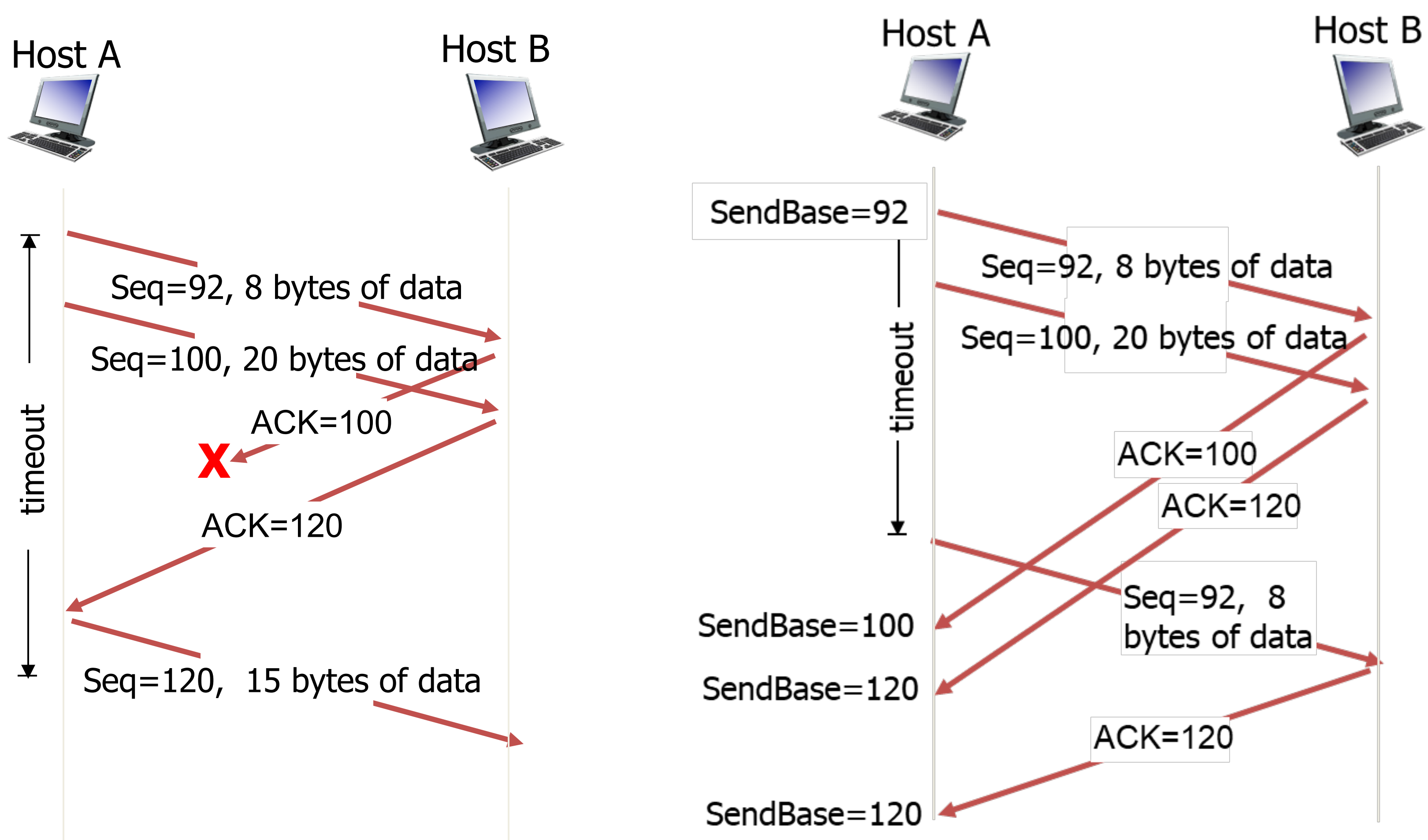
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

TCP: retransmission scenarios

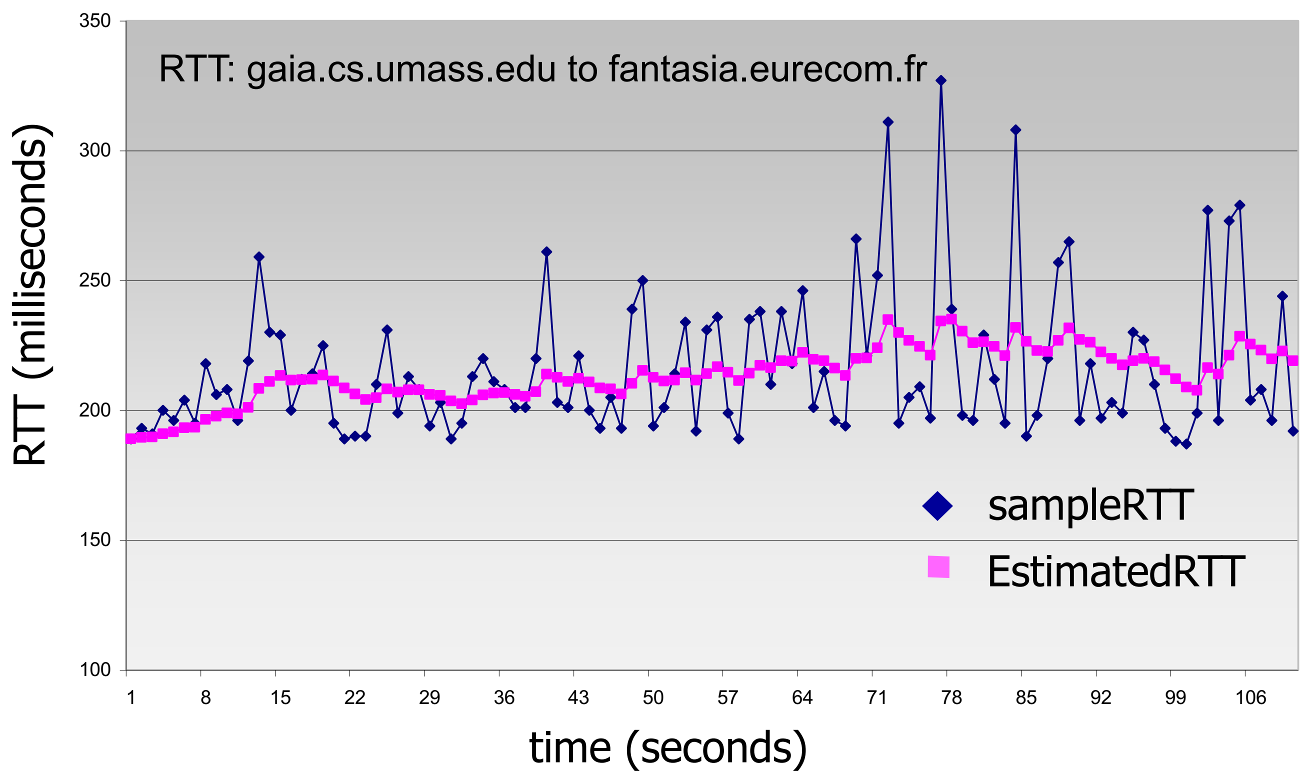


cumulative ACK

TCP round trip time, timeout

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- exponential weighted moving average
- 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 ->`** larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|

              (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

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

TCP fast retransmit

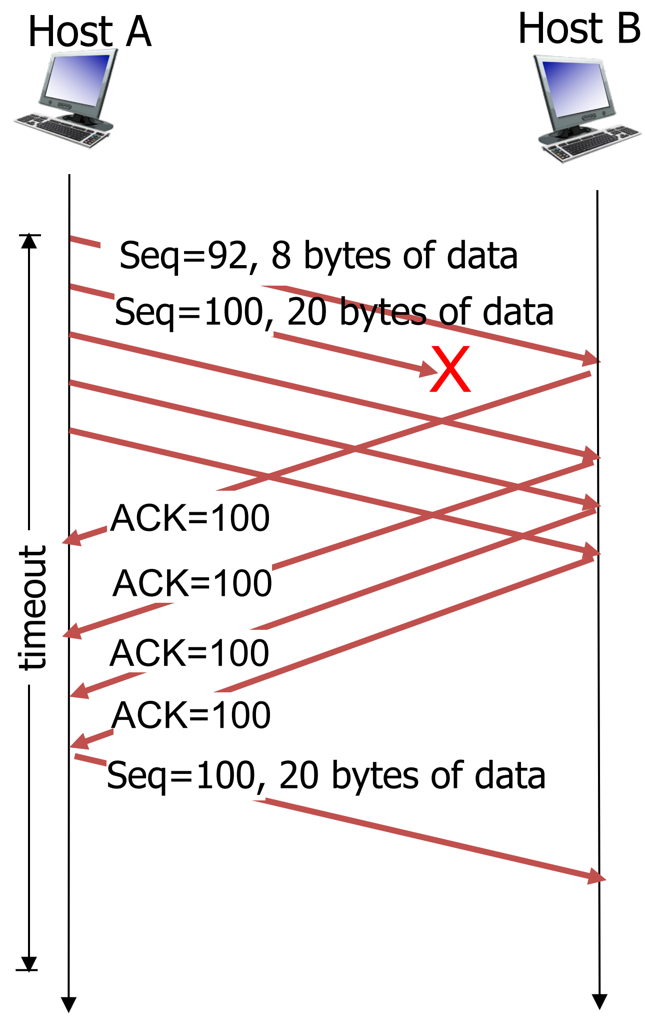
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

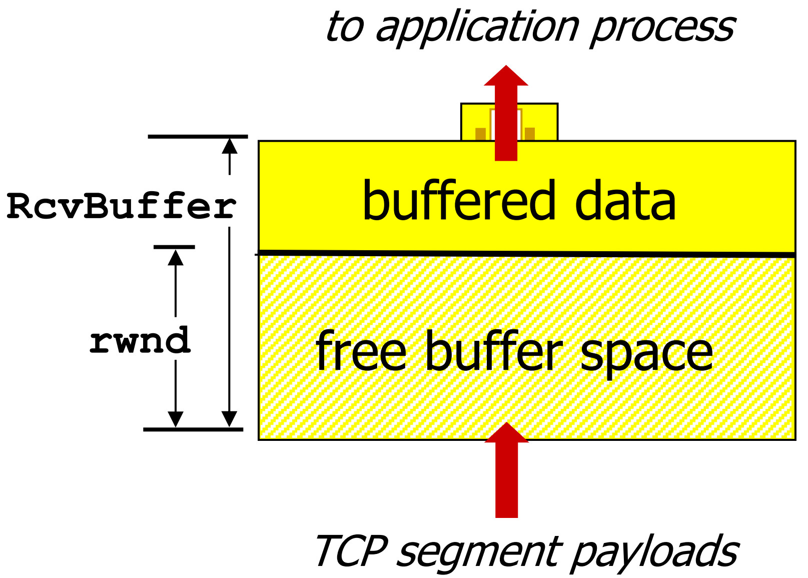
TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow

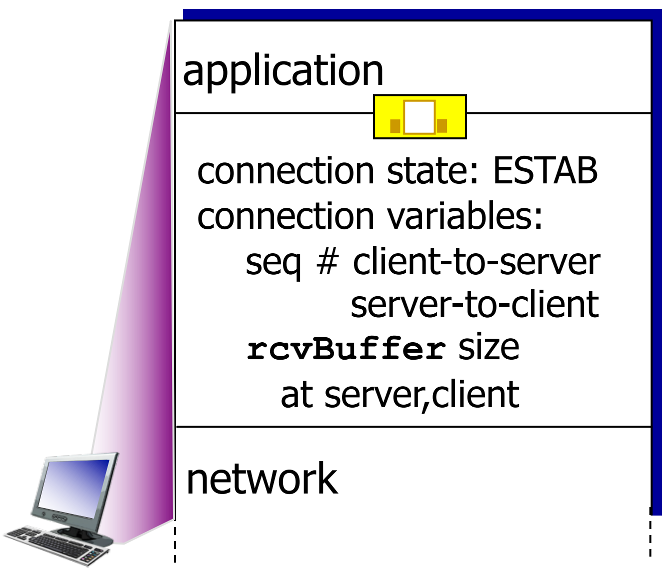


receiver-side buffering

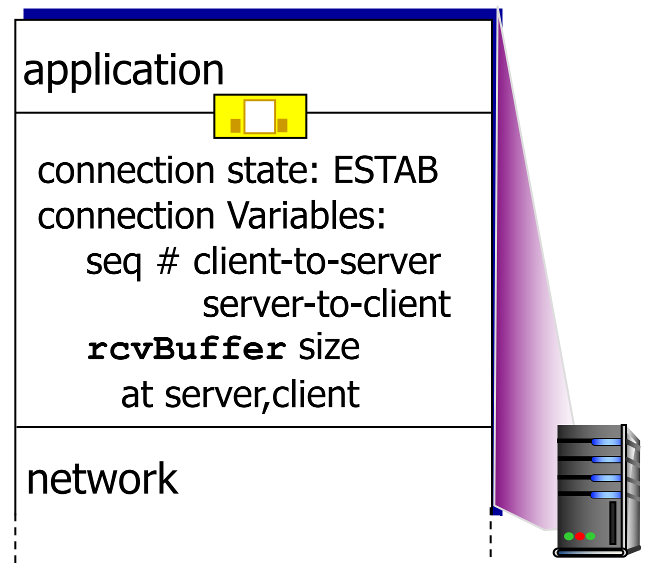
Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

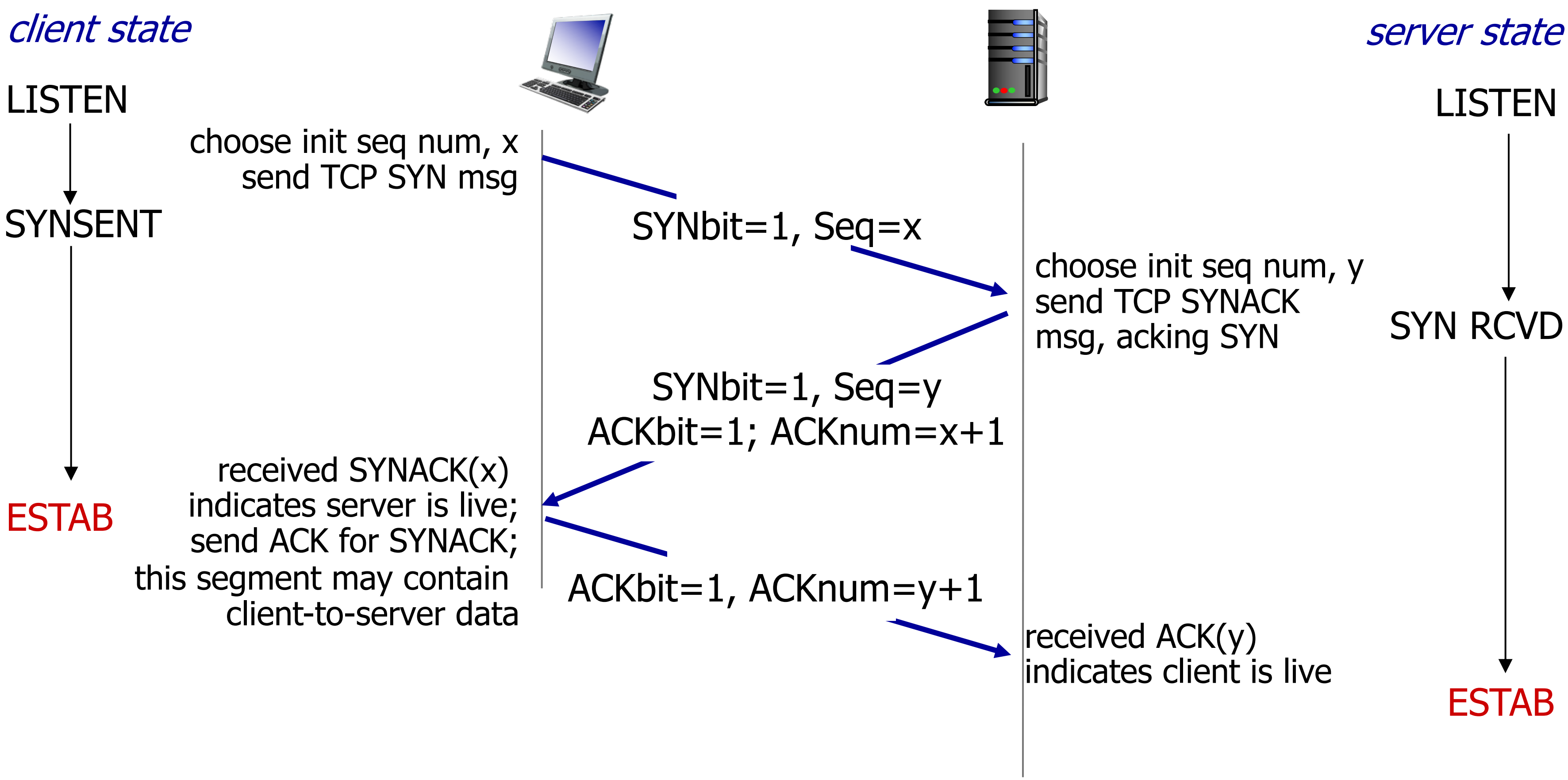



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```




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

TCP 3-way handshake



TCP: closing a connection