



Data Communications & Networks

Session 9 – Main Theme Network Congestion


Causes, Effects, Controls, and TCP Applications

Dr. Jean-Claude Franchitti

*New York University
Computer Science Department
Courant Institute of Mathematical Sciences*

*Adapted from course textbook resources
Computer Networking: A Top-Down Approach, 5/E
Copyright 1996-2013
J.F. Kurose and K.W. Ross, All Rights Reserved*

Agenda

- 
- 1 Session Overview
 - 2 Network Congestion Principles
 - 3 Internet Transport Protocols Review
 - 4 TCP Congestion Control
 - 5 Summary and Conclusion



- Course description and syllabus:

- » <http://www.nyu.edu/classes/jcf/csci-ga.2262-001/>

- » <http://cs.nyu.edu/courses/spring16/CSCI-GA.2262-001/index.html>

- Textbooks:

- » ***Computer Networking: A Top-Down Approach (6th Edition)***

James F. Kurose, Keith W. Ross

Addison Wesley

ISBN-10: 0132856204, ISBN-13: 978-0132856201, 6th Edition (02/24/12)





- Computer Networks and the Internet
- Application Layer
- Fundamental Data Structures: queues, ring buffers, finite state machines
- Data Encoding and Transmission
- Local Area Networks and Data Link Control
- Wireless Communications
- Packet Switching
- OSI and Internet Protocol Architecture
- Congestion Control and Flow Control Methods
- Internet Protocols (IP, ARP, UDP, TCP)
- Network (packet) Routing Algorithms (OSPF, Distance Vector)
- IP Multicast
- Sockets



- Introduction to Basic Networking Concepts (Network Stack)
- Origins of Naming, Addressing, and Routing (TCP, IP, DNS)
- Physical Communication Layer
- MAC Layer (Ethernet, Bridging)
- Routing Protocols (Link State, Distance Vector)
- Internet Routing (BGP, OSPF, Programmable Routers)
- TCP Basics (Reliable/Unreliable)
- Congestion Control
- QoS, Fair Queuing, and Queuing Theory
- Network Services – Multicast and Unicast
- Extensions to Internet Architecture (NATs, IPv6, Proxies)
- Network Hardware and Software (How to Build Networks, Routers)
- Overlay Networks and Services (How to Implement Network Services)
- Network Firewalls, Network Security, and Enterprise Networks



- Session Overview
- Network Congestion Principles
- Internet Transport Protocols Review
- TCP Congestion Control
- Summary & Conclusion



Information



Common Realization



Knowledge/Competency Pattern



Governance




Alignment



Solution Approach

Agenda

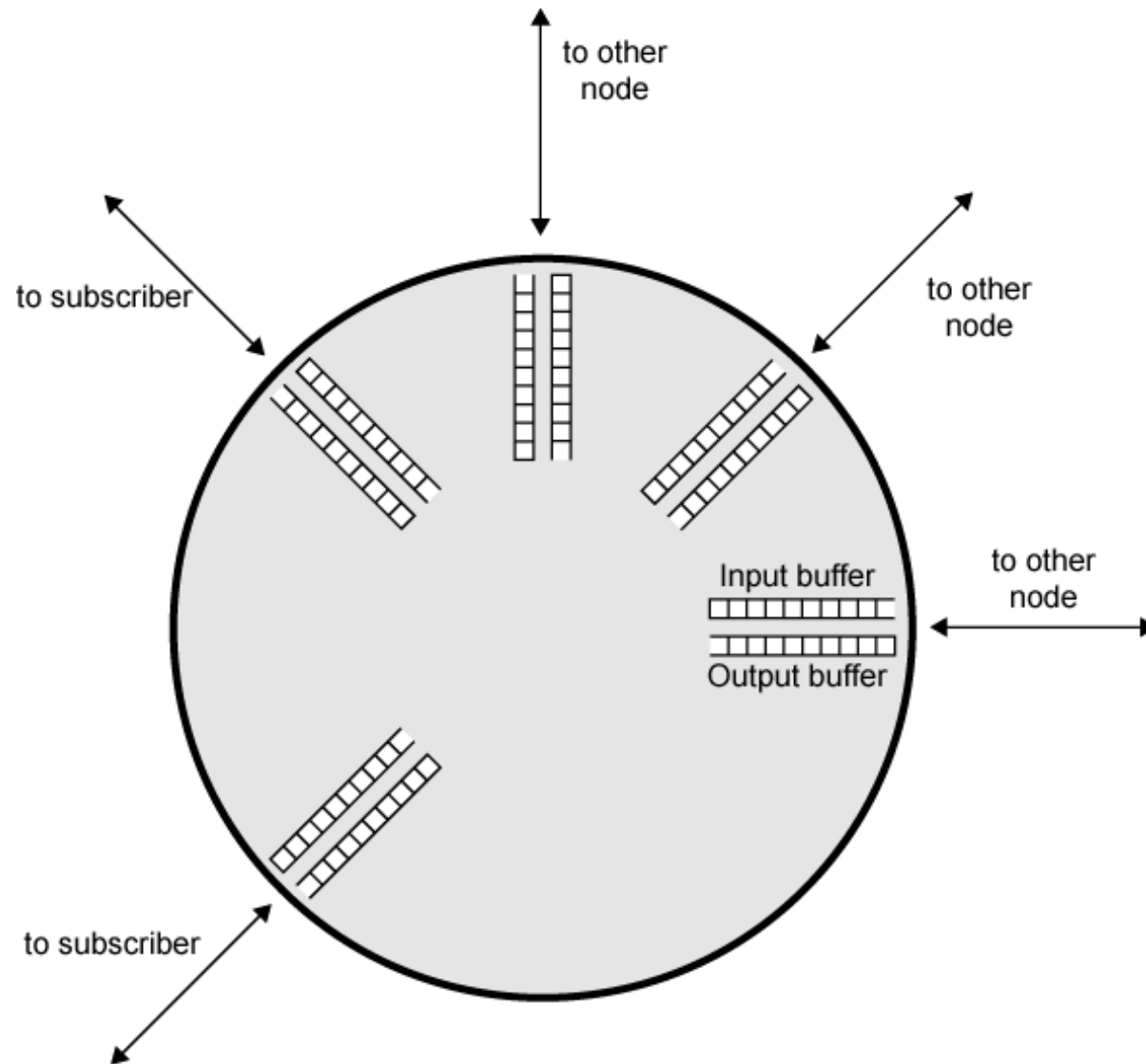
- 
- 1 Session Overview
 - 2 Network Congestion Principles
 - 3 Internet Transport Protocols Review
 - 4 TCP Congestion Control
 - 5 Summary and Conclusion



- What is Congestion?
- Effects of Congestion
- Causes/Costs of Congestion
- Approaches Towards Congestion Control

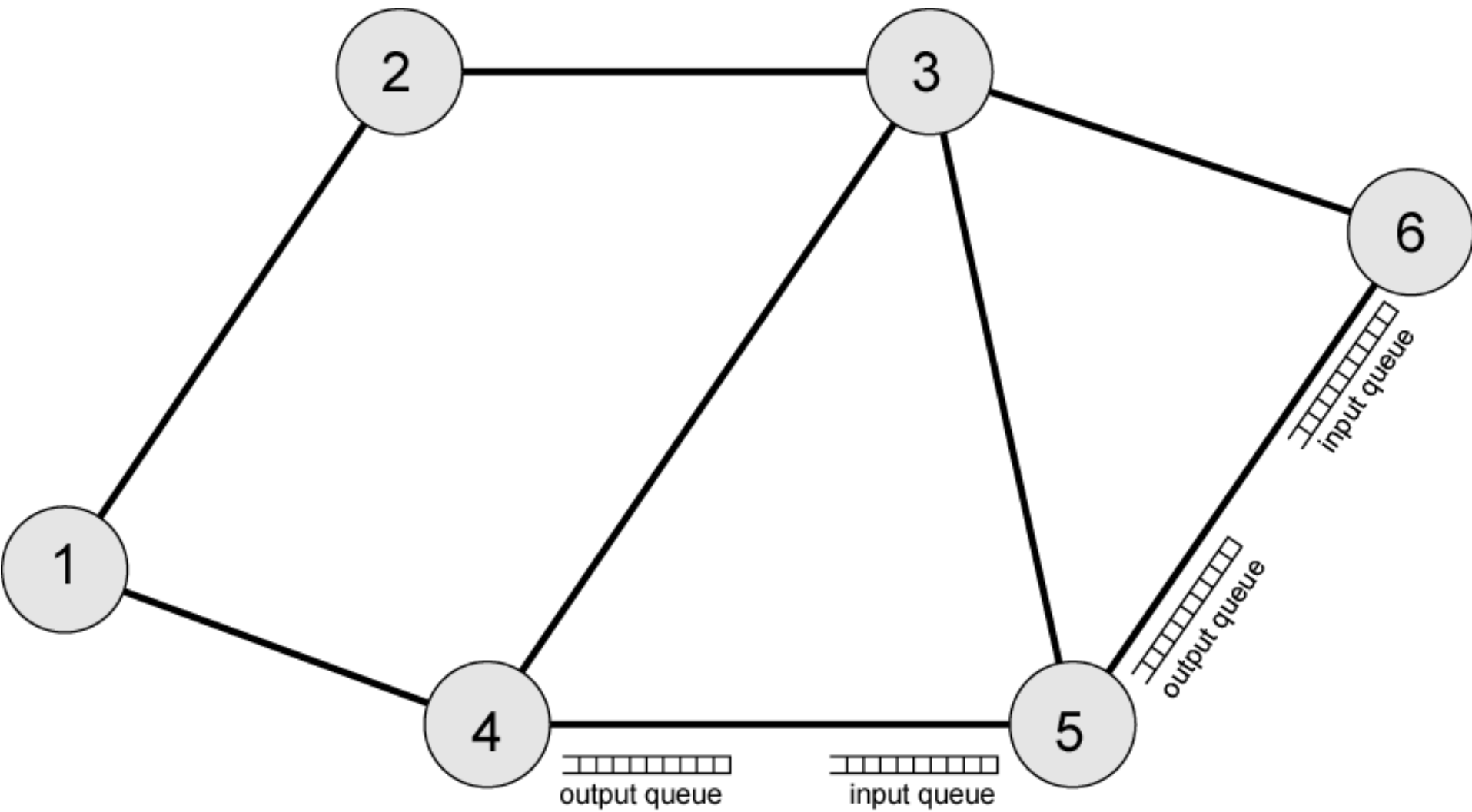


- Congestion occurs when the number of packets being transmitted through the network approaches the packet handling capacity of the network
- Congestion control aims to keep number of packets below level at which performance falls off dramatically
- Data network is a network of queues (e.g., router buffers)
- Generally 80% utilization is critical
- Finite queues mean data may be lost (e.g., as router buffers become congested)
- A top-10 problem!





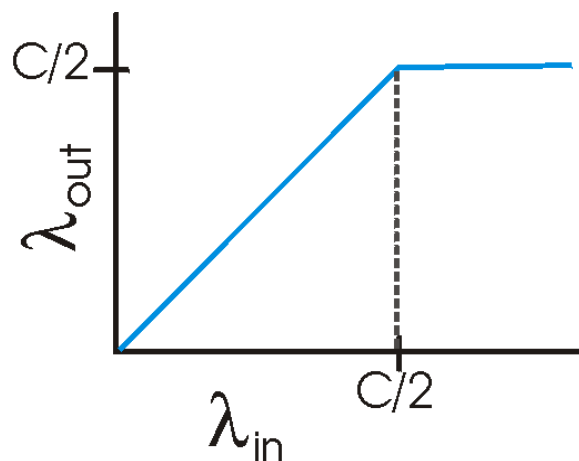
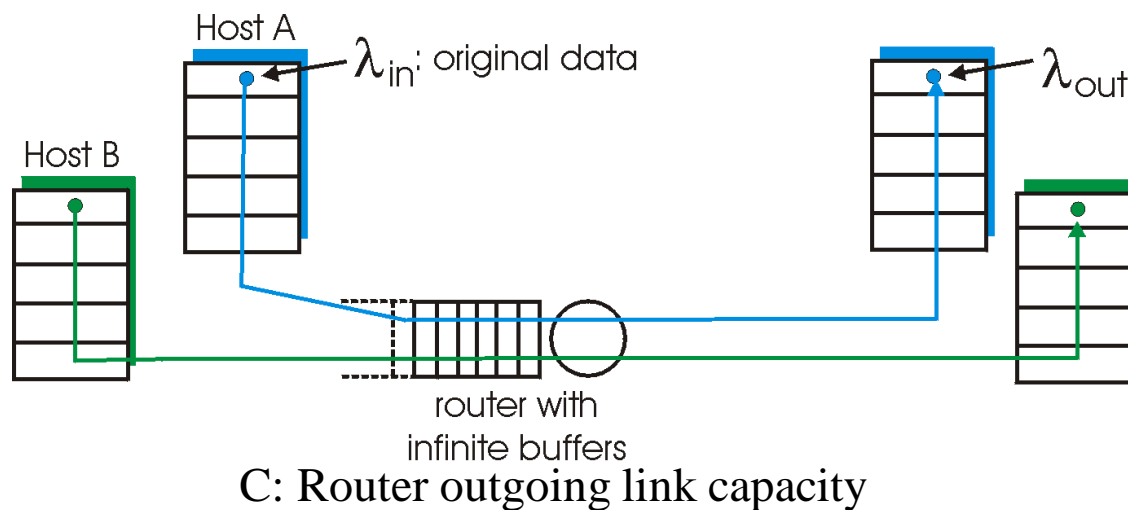
- Packets arriving are stored at input buffers
- Routing decision made
- Packet moves to output buffer
- Packets queued for output transmitted as fast as possible
 - Statistical time division multiplexing
- If packets arrive too fast to be routed, or to be output, buffers will fill
- Can discard packets
- Can use flow control
 - Can propagate congestion through network



Causes/Costs of Congestion: Scenario 1

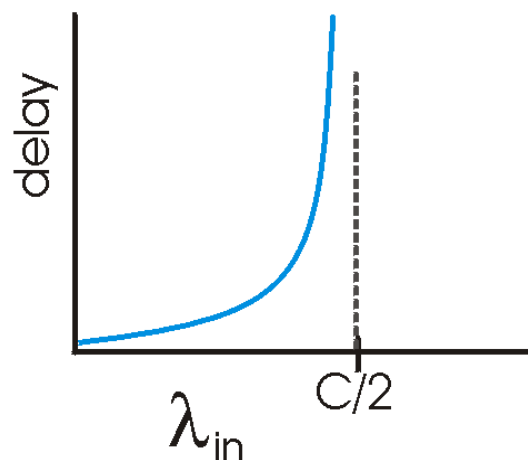


- two senders, two receivers
- one router, infinite buffers
- no retransmission
- no flow control
- no congestion control



Host A per connection throughput

(# of bytes/sec at receiver) as a function of the connection sending rate



Congestion cost:

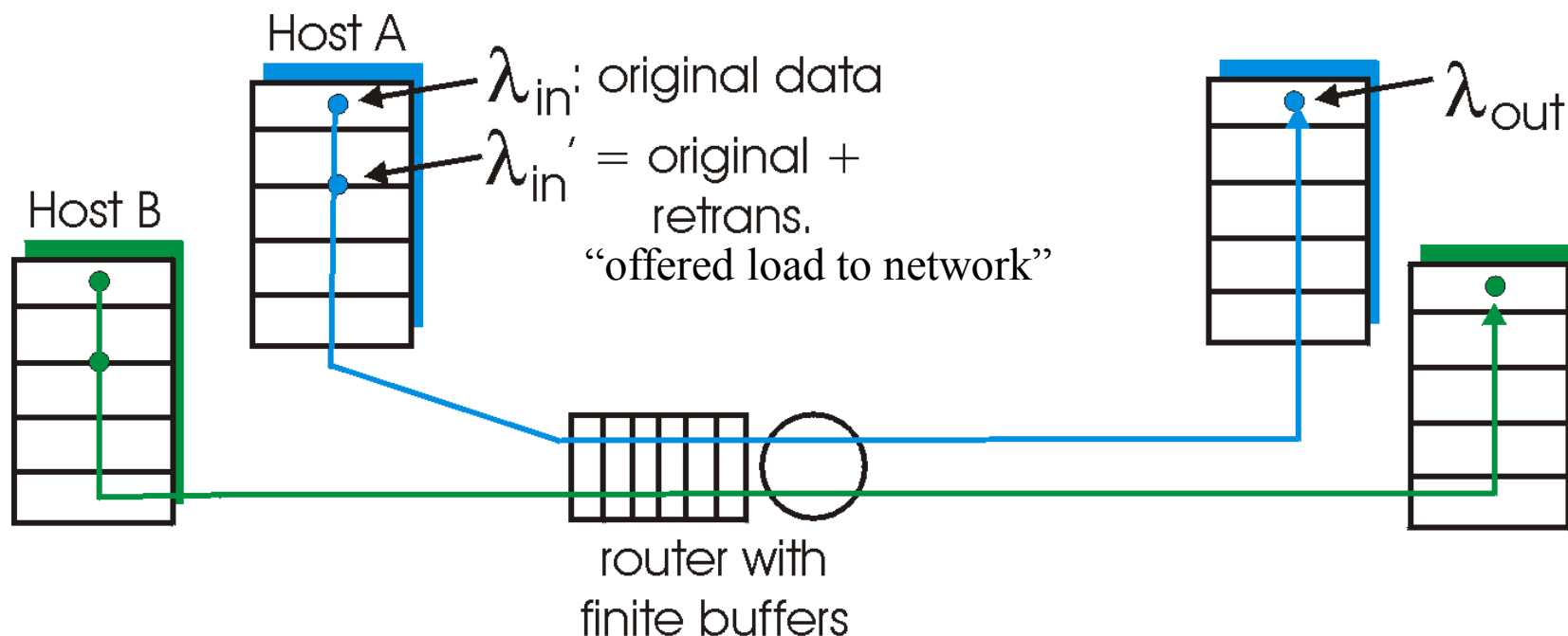
Average delay increases when operating near link capacity

- large delays when congested
- maximum achievable throughput

Causes/Costs of Congestion: Scenario 2 (1/2)



- one router, *finite* buffers
- sender retransmits lost packet (i.e. reliable connection assumed)

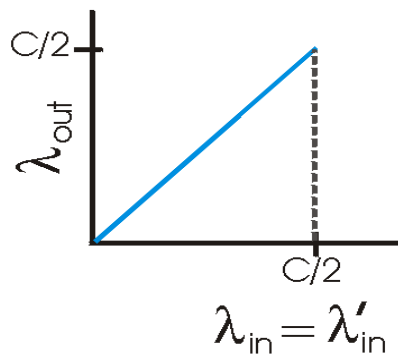


Performance depends on how retransmission is performed:

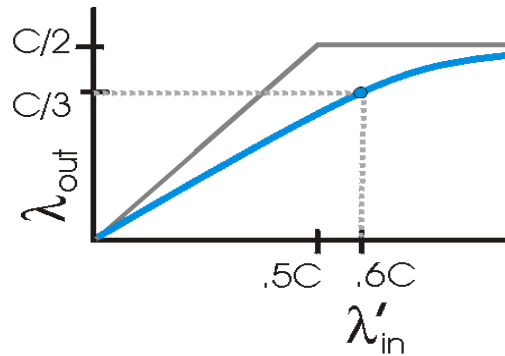
- Host A only sends a packet when a buffer is free -> no loss (offered load = sending rate)
- Sender only retransmits when a packet is known to be lost (timeout large enough...)
 - > **congestion cost: sender must retransmit to compensate for loss due to buffer overflow**
- Sender retransmits prematurely a delayed packet that is not lost
 - > **congestion cost: unneeded retransmissions in the face of large delays**



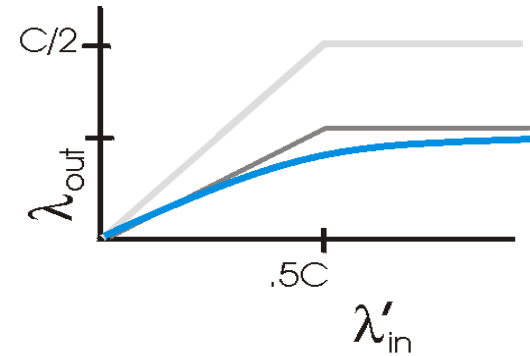
- always: $\lambda_{in} = \lambda_{out}$ ($\lambda'_{in} = \lambda_{in}$)
- “perfect” retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



(a)



(b)



(c)

“costs” of congestion:

Offered load is $C/2$

Throughput converges to $C/4$ if packets are forwarded twice

- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt



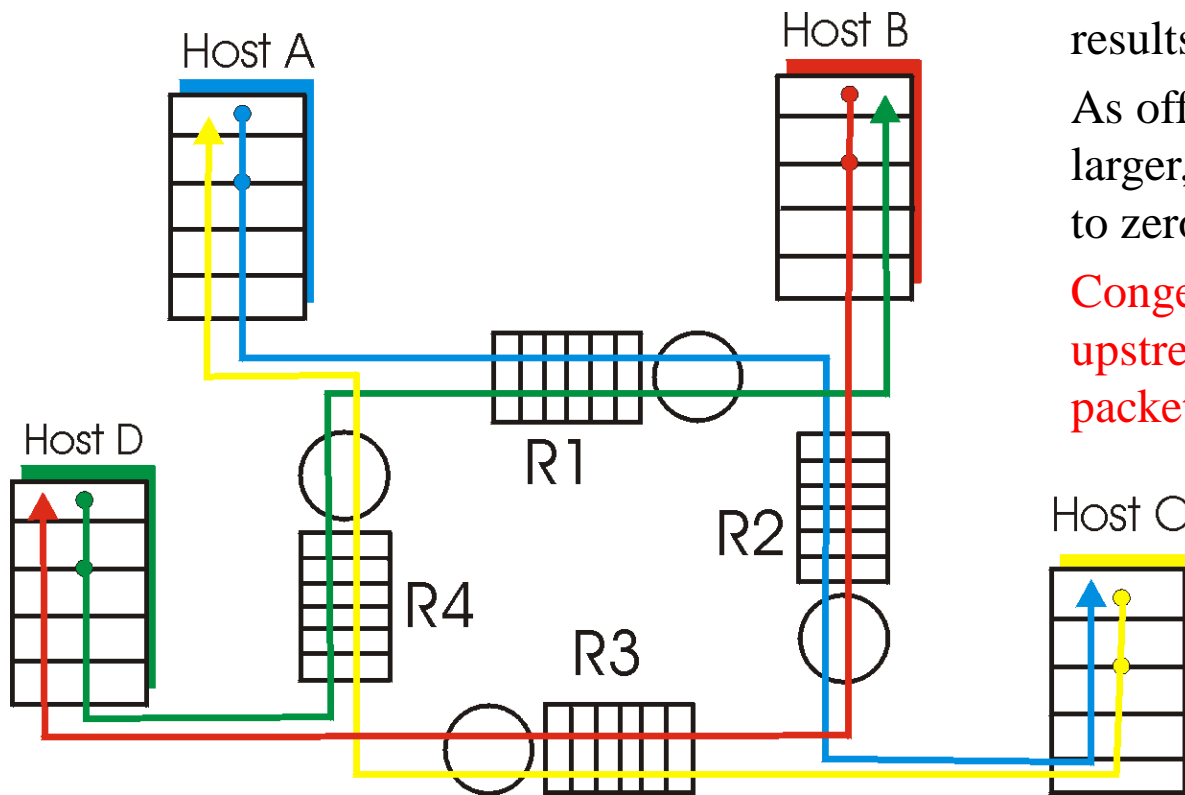
- four senders
- multihop paths
- timeout/retransmit

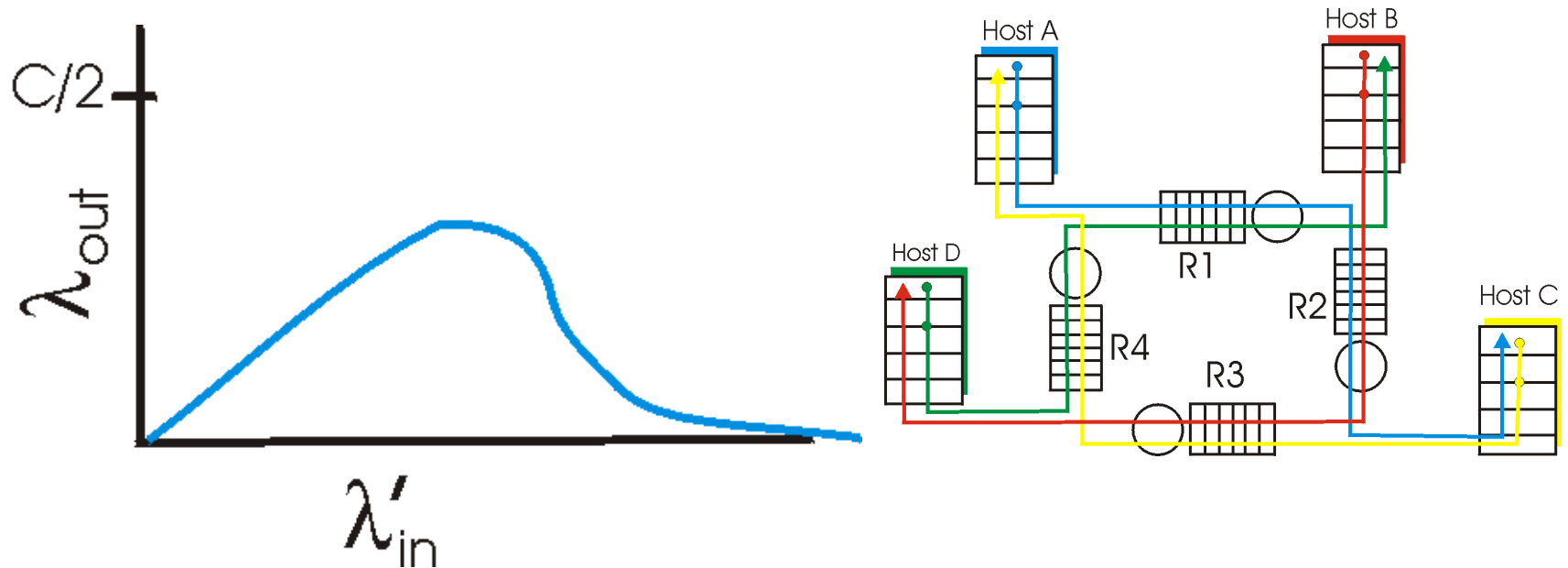
Q: what happens as λ_{in}
and λ'_{in} increase ?

Small increase in sending rate
results in a throughput increase

As offered load gets larger and
larger, throughput eventually goes
to zero

**Congestion cost: waste of
upstream transmission capacity to
packet drop point is wasted**





Another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet becomes wasted!



Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP (via indication of timeout of triple duplicate ack)

Network-assisted congestion control:

- routers provide feedback to end systems
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM ABR)
 - explicit rate sender should send at
 - indications: choke packets, packet field update

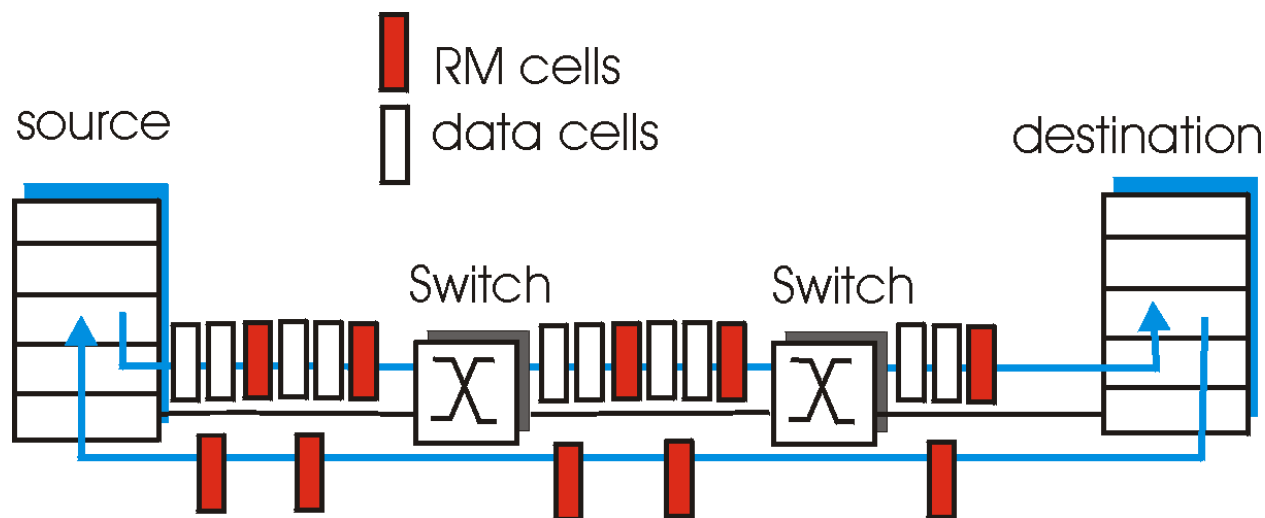


■ ABR: available bit rate:

- “elastic service”
- if sender’s path “underloaded”:
 - sender should use available bandwidth
- if sender’s path congested:
 - sender throttled to minimum guaranteed rate


RM (resource management) cells:

- sent by sender, interspersed with data cells (one per 32)
- bits in RM cell set by switches (“*network-assisted*”)
 - **NI bit**: no increase in rate (mild congestion)
 - **CI bit**: congestion indication
- RM cells returned to sender by receiver, with bits intact



- two-byte ER (explicit rate) field in RM cell
 - » congested switch may lower ER value in cell
 - » sender' send rate thus minimum supportable rate on path
- EFCI (Explicit Forward Congestion Indication) bit in data cells: set to 1 in congested switch
 - » if data cell preceding RM cell has EFCI set, destination sets CI bit in returned RM cell

Agenda

- 
- 1 Session Overview
 - 2 Network Congestion Principles
 - 3 Internet Transport Protocols Review
 - 4 TCP Congestion Control
 - 5 Summary and Conclusion



- Internet Transport Protocols
- Transport Layer Addressing
- Standard Services and Port Numbers
- TCP Overview
- Reliability in an Unreliable World
- TCP Flow Control
- Why Startup / Shutdown Difficult?
- TCP Connection Management
- Timing Problem
- Implementation Policy Options
- UDP: User Datagram Protocol



- Two Transport Protocols Available
 - Transmission Control Protocol (TCP)
 - *connection oriented*
 - most applications use TCP
 - RFC 793
 - User Datagram Protocol (UDP)
 - *Connectionless*
 - RFC 768



- Transport Layer Multiplexing and Demultiplexing
 - Extend IP's delivery svc between two end systems to a delivery svc between two processes running on the end systems
- Segment Integrity Checking
- TCP Only:
 - Reliable data transfer (flow control, seq #s, acknowledgements, and timers)
 - Congestion control



- Communications endpoint addressed by:
 - IP address (32 bit) in IP Header
 - Port numbers (16 bit) in TP Header¹
 - Transport protocol (TCP or UDP) in IP Datagram Header

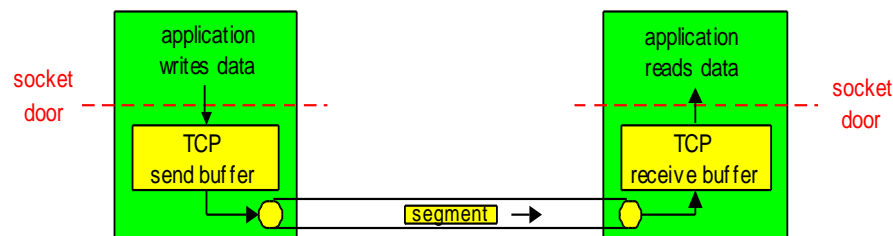
¹ TP => Transport Protocol (UDP or TCP)



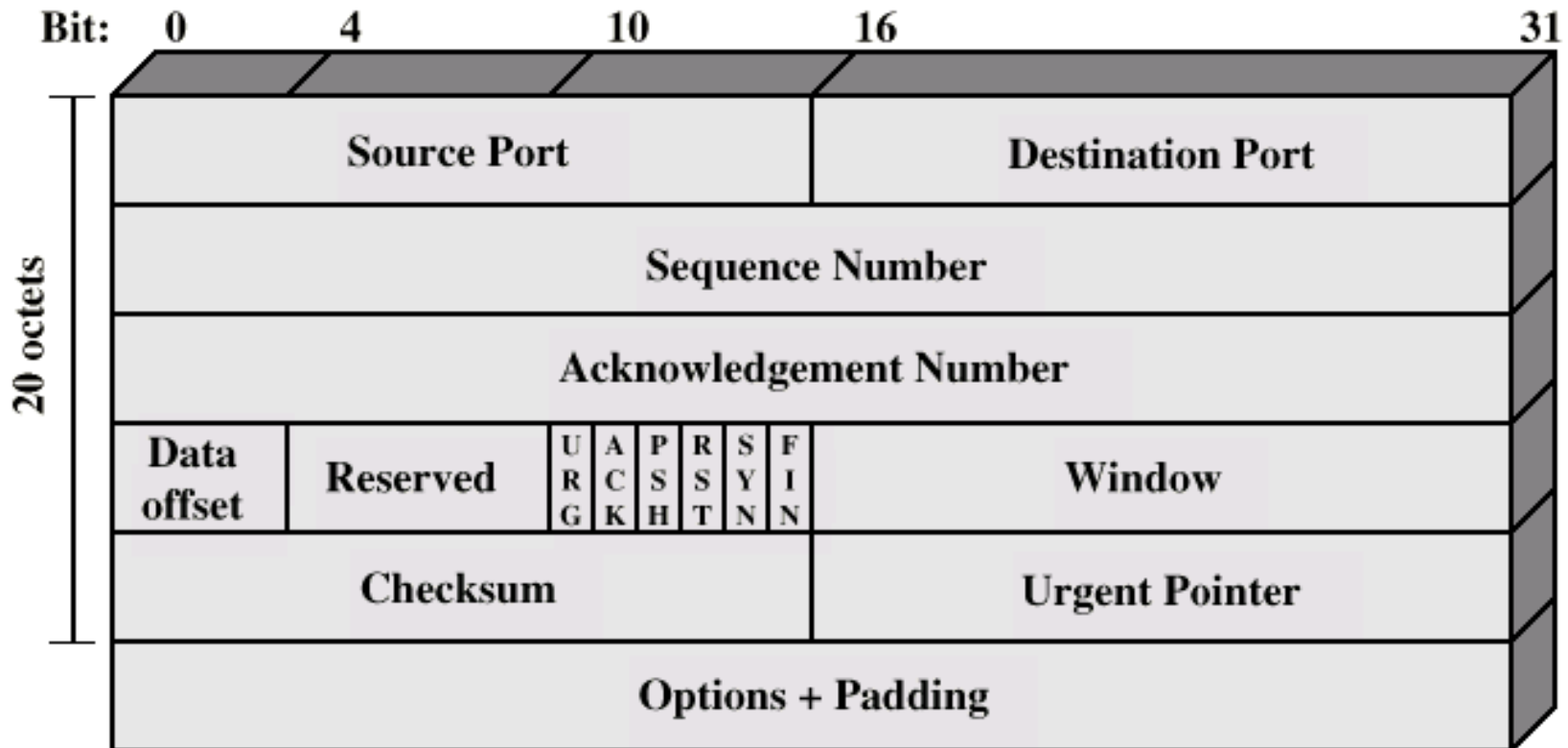
service	tcp	udp
echo	7	7
daytime	13	13
netstat	15	
ftp-data	20	
ftp	21	
telnet	23	
smtp	25	
time	37	37
domain	53	53
finger	79	
http	80	
pop-2	109	
pop	110	
sunrpc	111	111
uucp-path	117	
nntp	119	
talk		517



- **point-to-point:**
 - one sender, one receiver
(no multicasting possible)
- **reliable, in-order *byte stream*:**
 - no “message boundaries”
- **pipelined:**
 - TCP congestion and flow control set window size
- ***send & receive buffers***

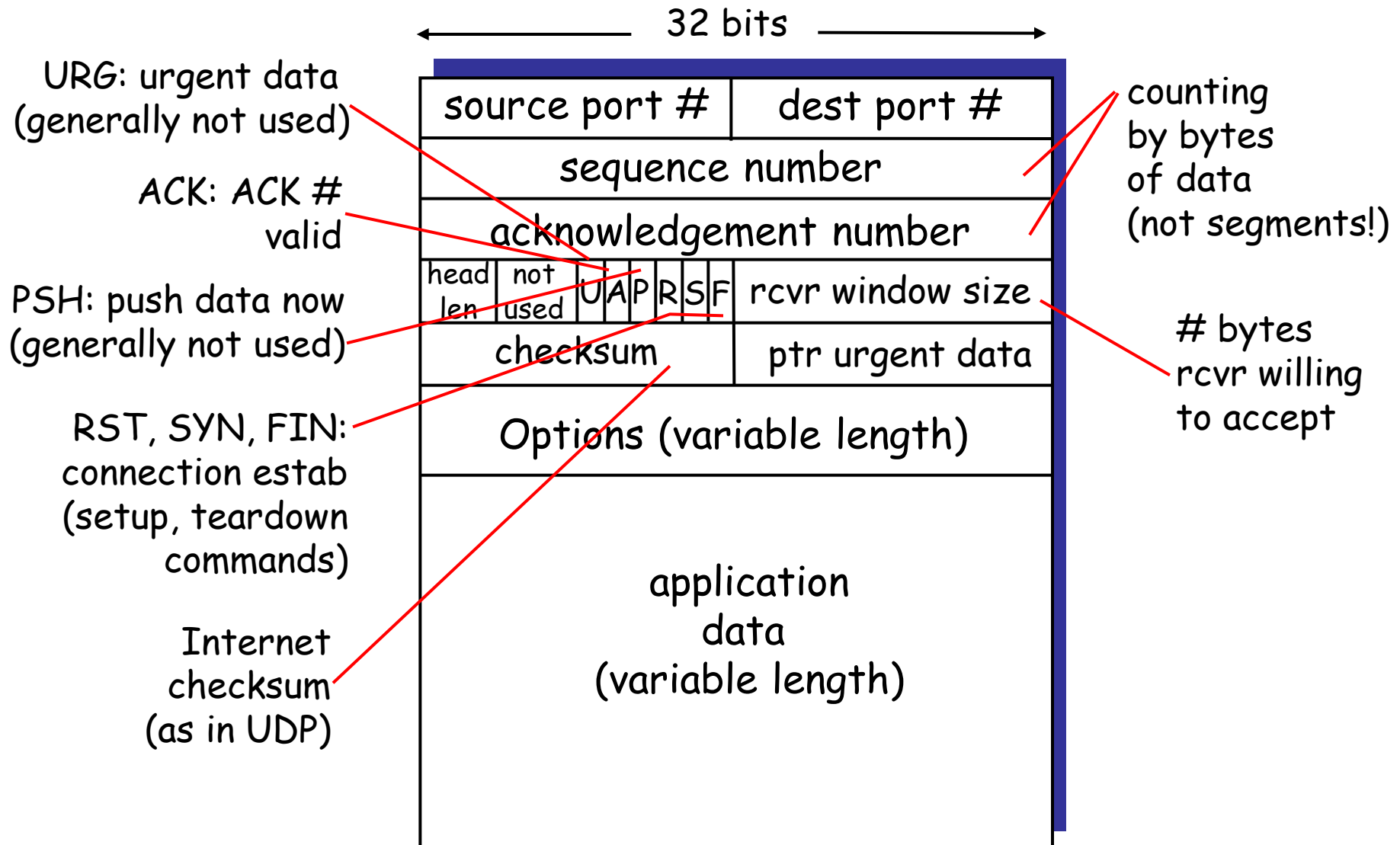


- **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size (app layer data size)
- **connection-oriented:**
 - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- **flow controlled:**
 - sender will not overwhelm receiver



- Data offset: specifies length of TCP header in 32-bit words
- Options field: used when a sender and receiver negotiate the MSS or as a window scaling factor for use in high-speed networks or for timestamping (RFC 854/1323)

TCP Segment Structure





- IP offers best-effort (unreliable) delivery
- TCP uses IP
- TCP provides completely reliable transfer
- How is this possible? How can TCP realize:
 - Reliable connection startup?
 - Reliable data transmission?
 - Graceful connection shutdown?



- Positive acknowledgment
 - Receiver returns short message when data arrives
 - Called *acknowledgment*
- Retransmission
 - Sender starts timer whenever message is transmitted
 - If timer expires before acknowledgment arrives, sender *retransmits* message
 - THIS IS NOT A TRIVIAL PROBLEM! – more on this later



- Receiver
 - Advertises available buffer space
 - Called *window*
 - This is known as a *CREDIT* policy
- Sender
 - Can send up to entire window before ACK arrives
- Each acknowledgment carries new window information
 - Called *window advertisement*
 - Can be zero (called *closed window*)
- Interpretation: I have received up through X, and can take Y more octets



- Decouples flow control from ACK
 - May ACK without granting credit and vice versa
- Each octet has sequence number
- Each transport segment has seq number, ack number and window size in header



- When sending, seq number is that of first octet in segment
- ACK includes $AN=i$, $W=j$
- All octets through $SN=i-1$ acknowledged
 - Next expected octet is i
- Permission to send additional window of $W=j$ octets
 - i.e. octets through $i+j-1$

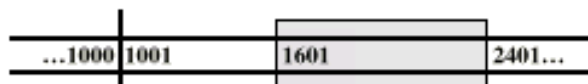


Credit Allocation

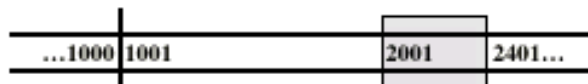
Transport Entity A



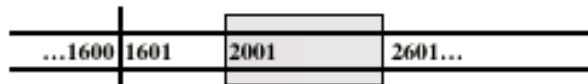
A may send 1400 octets



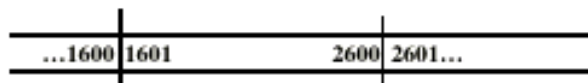
A shrinks its transmit window with each transmission



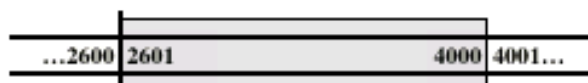
A adjusts its window with each credit



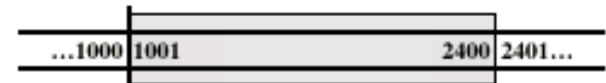
A exhausts its credit



A receives new credit



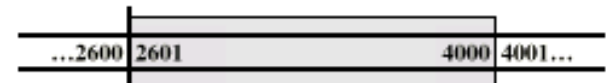
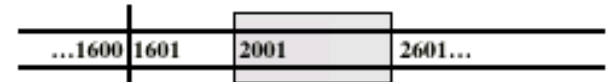
Transport Entity B



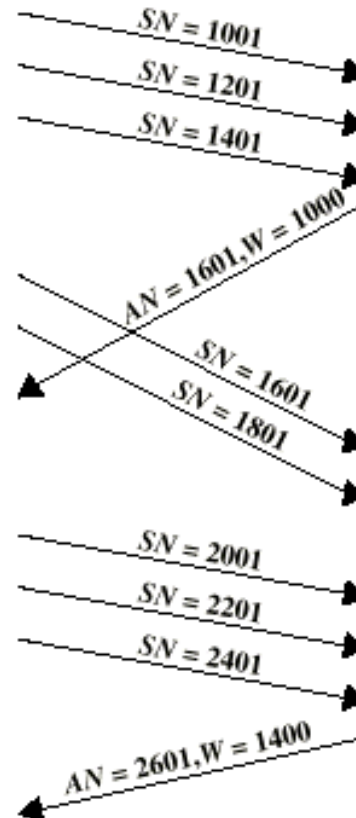
B is prepared to receive 1400 octets, beginning with 1001



B acknowledges 3 segments (600 octets), but is only prepared to receive 200 additional octets beyond the original budget (i.e., B will accept octets 1601 through 2600)



B acknowledges 5 segments (1000 octets) and restores the original amount of credit



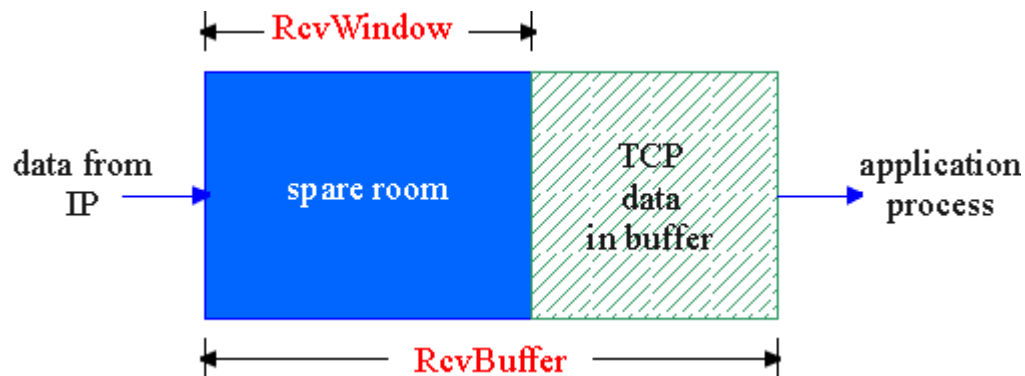


flow control

sender won't overrun
receiver's buffers by
transmitting too much,
too fast

RcvBuffer = size of TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

receiver: explicitly
informs sender of
(dynamically
changing) amount of
free buffer space

- **RcvWindow** field
in TCP segment

sender: keeps the amount
of transmitted,
unACKed data less
than most recently
received **RcvWindow**



Seq. #'s:

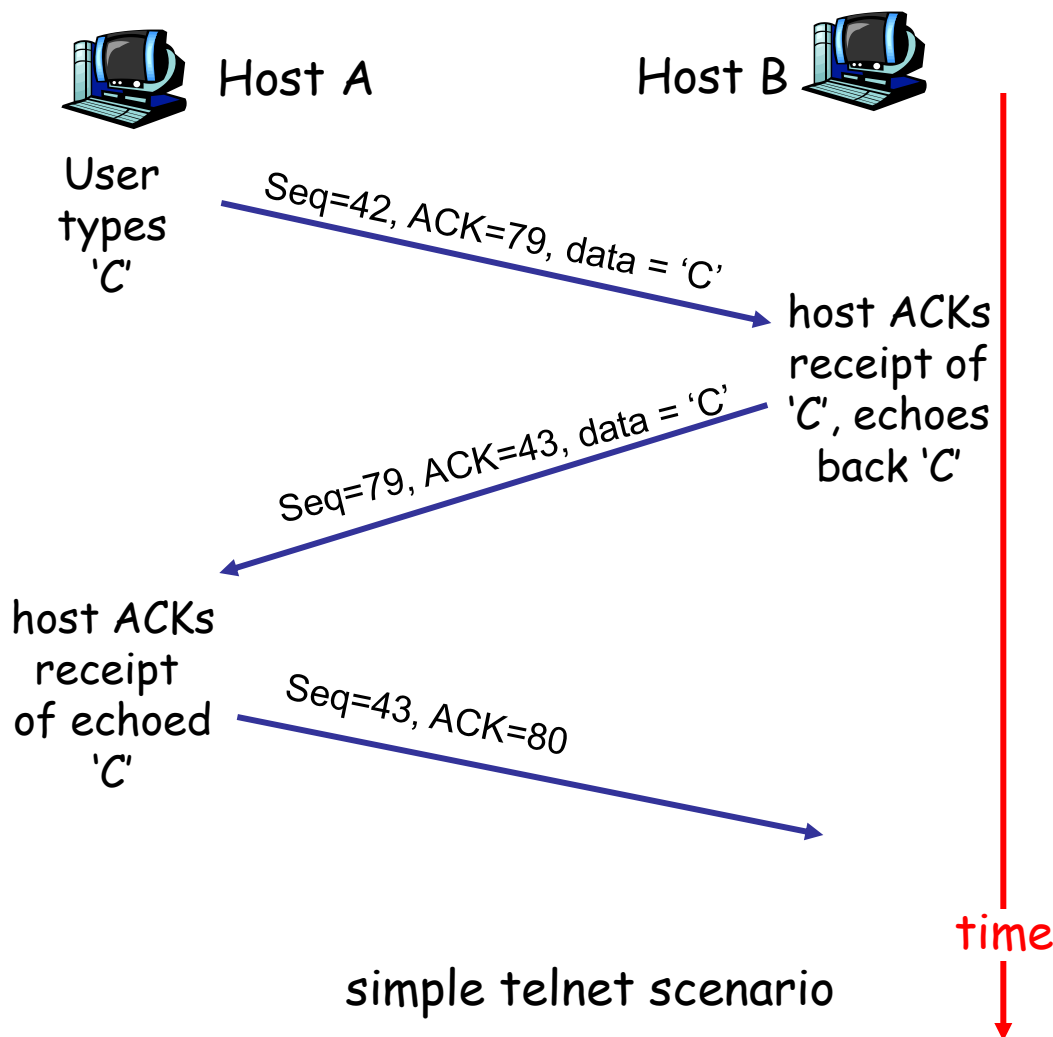
- byte stream
“number” of first
byte in segment's
data

ACKs:

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

Q: how receiver handles
out-of-order segments

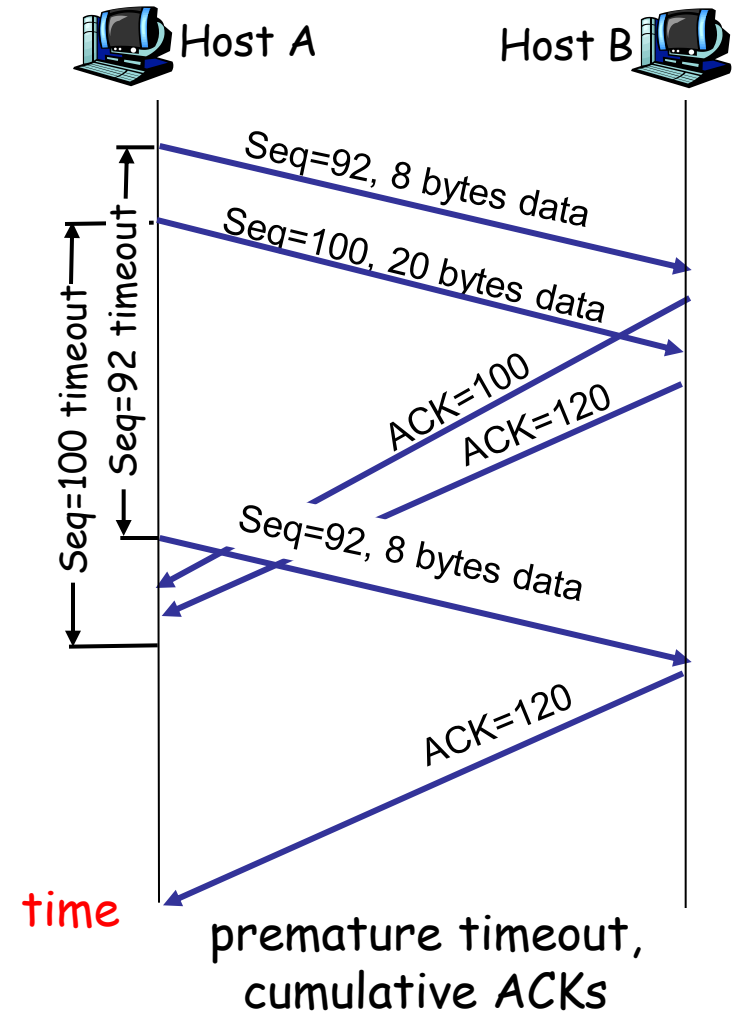
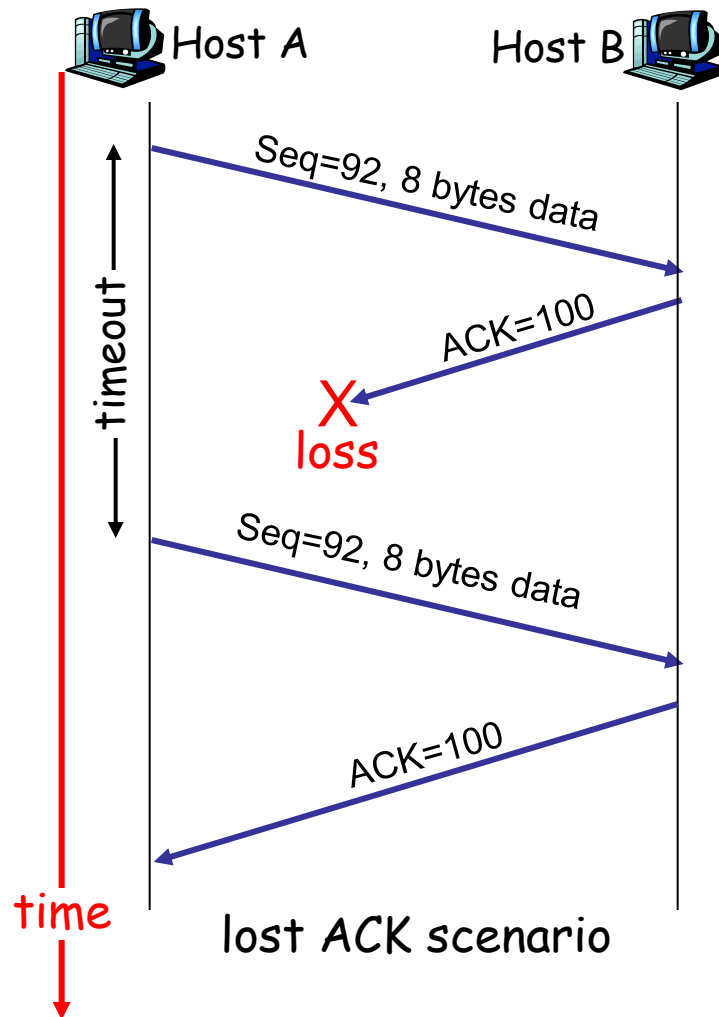
- A: TCP spec
doesn't say, - up to
implementor





Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: Retransmission Scenarios





- Segments can be
 - Lost
 - Duplicated
 - Delayed
 - Delivered out of order
 - Either side can crash
 - Either side can reboot
- Need to avoid duplicate “shutdown” message from affecting later connection



- Recall: TCP sender, receiver establish “connection” before exchanging data segments
- initialize TCP variables:
 - » seq. #s
 - » buffers, flow control info (e.g. **RcvWindow**)
- *client*: connection initiator
Socket clientSocket = new Socket("hostname", "port number");
- *server*: contacted by client
Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

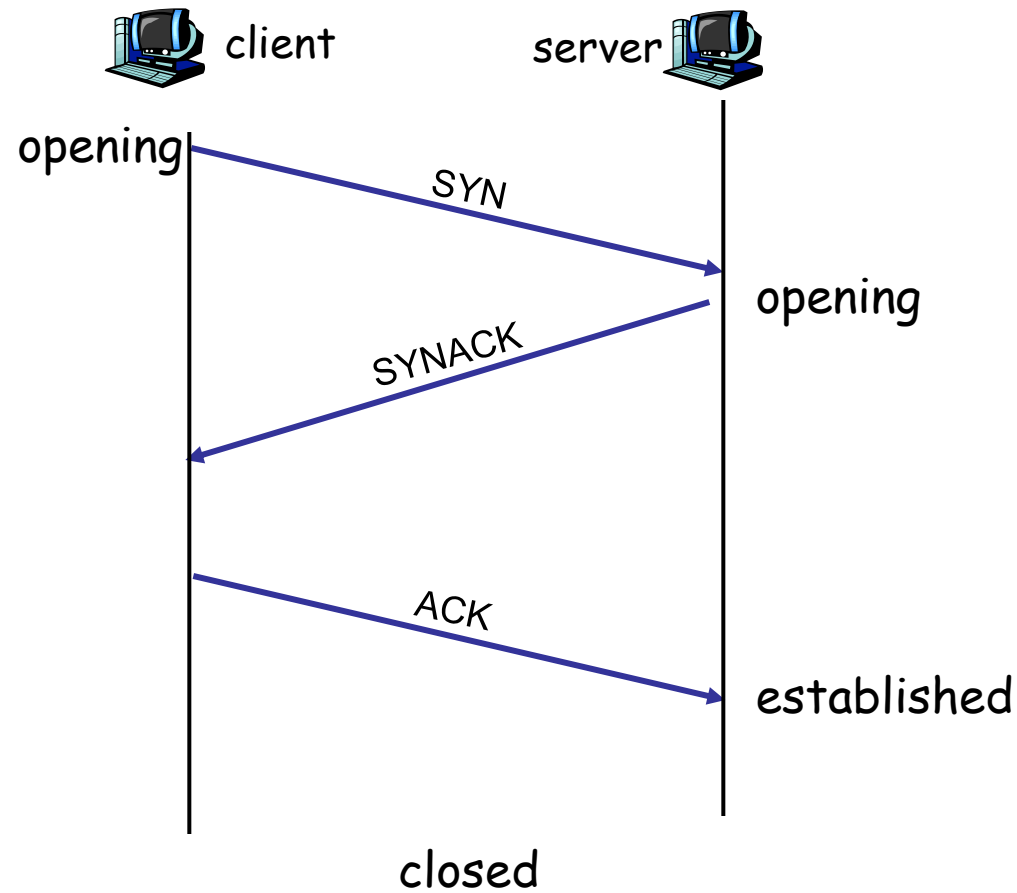
Step 1: client end system sends TCP SYN control segment to server

- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

TCP Connection Management (OPEN)





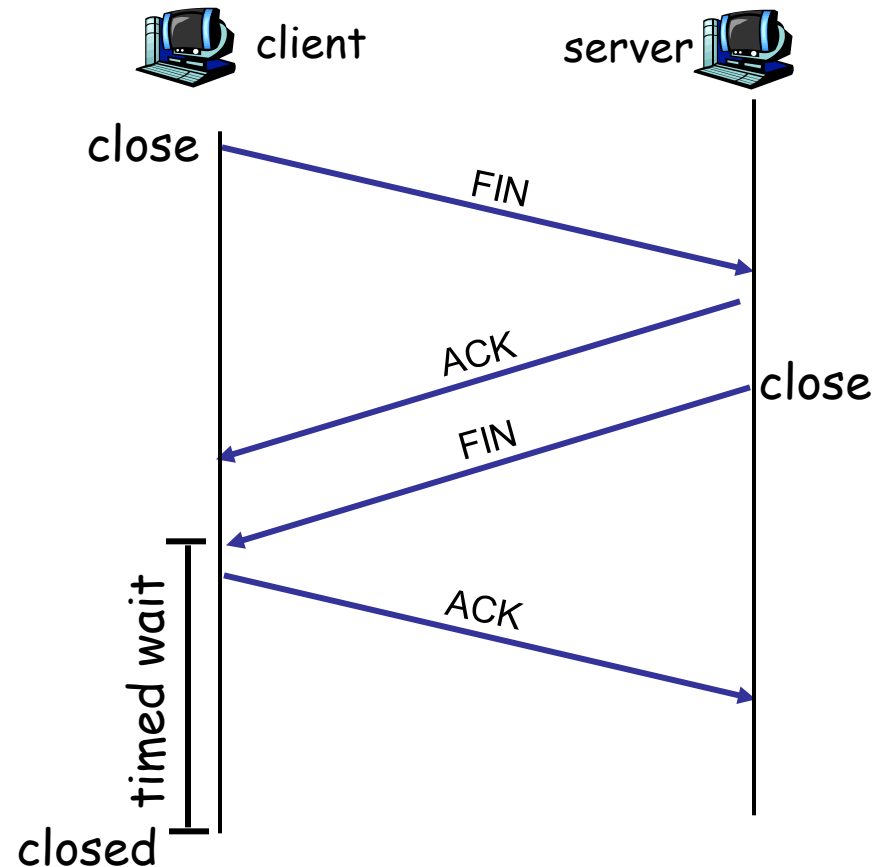
Closing a connection:

client closes socket:

```
clientSocket.close  
( ) ;
```

Step 1: client end system
sends TCP FIN control
segment to server

Step 2: server receives FIN,
replies with ACK. Closes
connection, sends FIN.



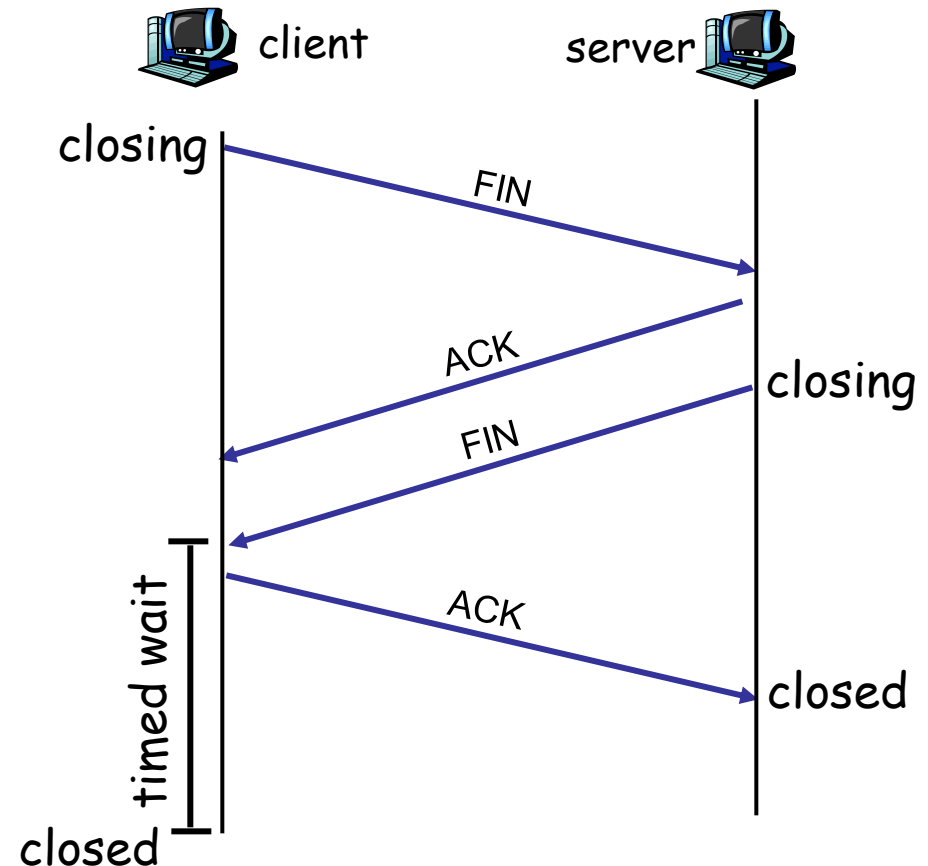


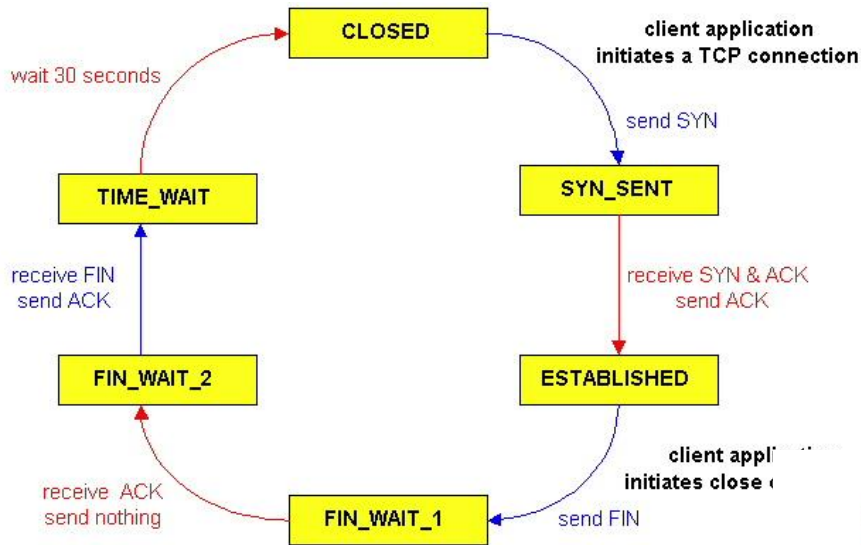
Step 3: client receives FIN,
replies with ACK.

- Enters “timed wait” -
will respond with ACK
to received FINs

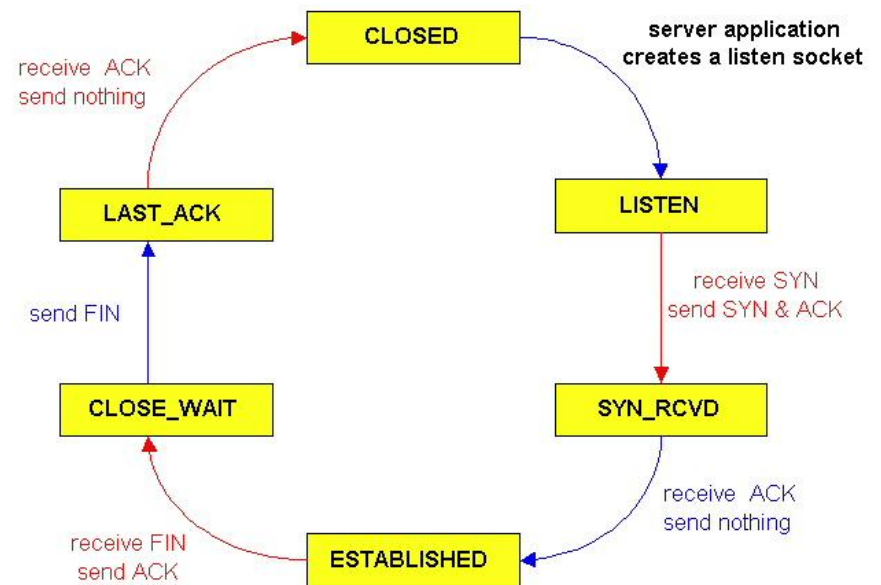
Step 4: server, receives ACK.
Connection closed.

Note: with small modification,
can handle simultaneous
FINs.





TCP server lifecycle





The delay required for data to reach a destination and an acknowledgment to return depends on traffic in the internet as well as the distance to the destination. Because it allows multiple application programs to communicate with multiple destinations concurrently, TCP must handle a variety of delays that can change rapidly.

How does TCP handle this



- Keep estimate of round trip time on each connection
- Use current estimate to set retransmission timer
- Known as *adaptive retransmission*
- Key to TCP's success



- Q: how to set TCP timeout value?
- longer than RTT
 - » note: RTT will vary
- 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, cumulatively ACKed segments
- `SampleRTT` will vary, want estimated RTT “smoother”
 - use several recent measurements, not just current `SampleRTT`



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

- Exponential weighted moving average (EWMA)
- Influence of given sample decreases exponentially fast
- Typical value of x: 1/8 (RFC 6298)

Setting the timeout

- **EstimatedRTT** plus “safety margin”
- large variation in **EstimatedRTT** -> larger safety margin (y typically 0.25)

$$\text{Timeout} = \text{EstimatedRTT} + 4 * \text{Deviation}$$

$$\begin{aligned} \text{Deviation} = & (1-y) * \text{Deviation} + \\ & y * |\text{SampleRTT} - \text{EstimatedRTT}| \end{aligned}$$



- Send
- Deliver
- Accept
- Retransmit
- Acknowledge



- If no push or close TCP entity transmits at its own convenience (IFF send window allows!)
- Data buffered at transmit buffer
- May construct segment per data batch
- May wait for certain amount of data



- In absence of push, deliver data at own convenience
- May deliver as each in-order segment received
- May buffer data from more than one segment



- Segments may arrive out of order
- In order
 - Only accept segments in order
 - Discard out of order segments
- In windows
 - Accept all segments within receive window



- TCP maintains queue of segments transmitted but not acknowledged
- TCP will retransmit if not ACKed in given time
 - First only
 - Batch
 - Individual



- Immediate
 - as soon as segment arrives.
 - will introduce extra network traffic
 - Keeps sender's pipe open
- Cumulative
 - Wait a bit before sending ACK (called “delayed ACK”)
 - Must use timer to insure ACK is sent
 - Less network traffic
 - May let sender's pipe fill if not timely!



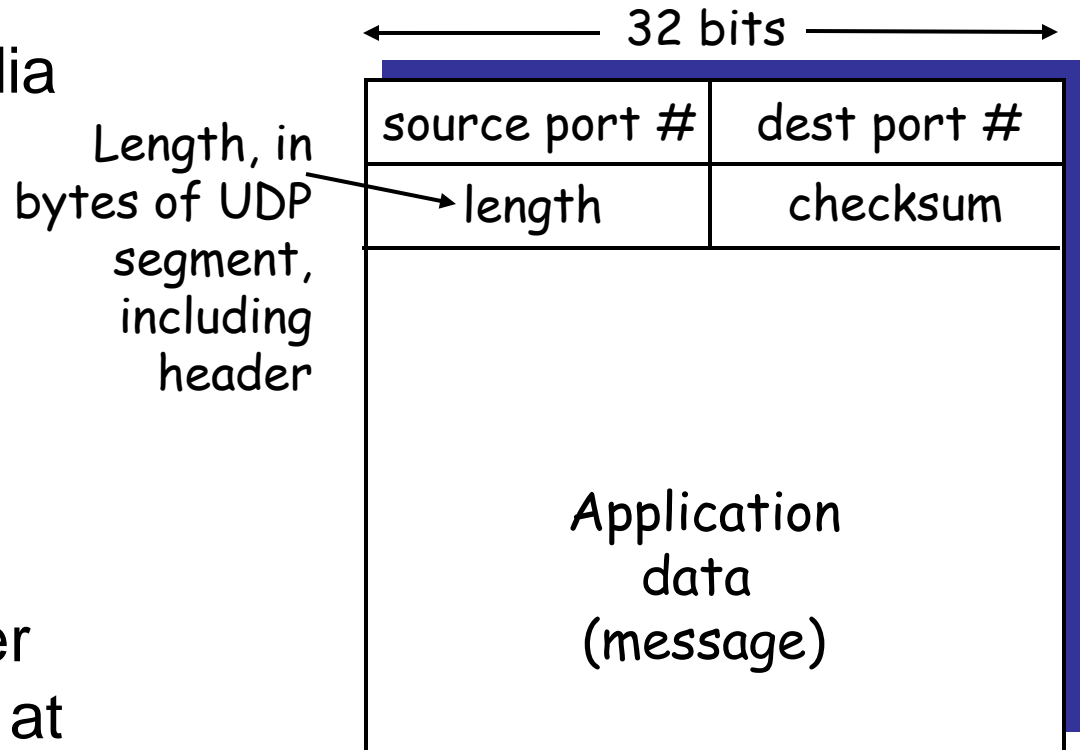
- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - » lost
 - » delivered out of order to app
- *connectionless*:
 - » no handshaking between UDP sender, receiver
 - » each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header (8 vs. 20 bytes)
- no congestion control: UDP can blast away as fast as desired
- No retransmission
- Good for real-time apps
 - Require min sending rate and reduced delays and tolerate loss



- often used for streaming multimedia apps
 - » loss tolerant
 - » rate sensitive
- other UDP uses
 - » DNS
 - » SNMP
- reliable transfer over UDP: add reliability at application layer
 - » application-specific error recover!



UDP segment format



- Inward data collection
- Outward data dissemination
- Request-Response
- Real time application
- Examples:
 - DNS
 - RIP
 - SNMP

Agenda

- 1 Session Overview
- 2 Network Congestion Principles
- 3 Internet Transport Protocols Review
- 4 TCP Congestion Control
- 5 Summary and Conclusion





- TCP Congestion Control
- TCP Fairness



- end-end control (no network assistance)
- sender limits transmission:
 $\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

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

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (CongWin) after loss event

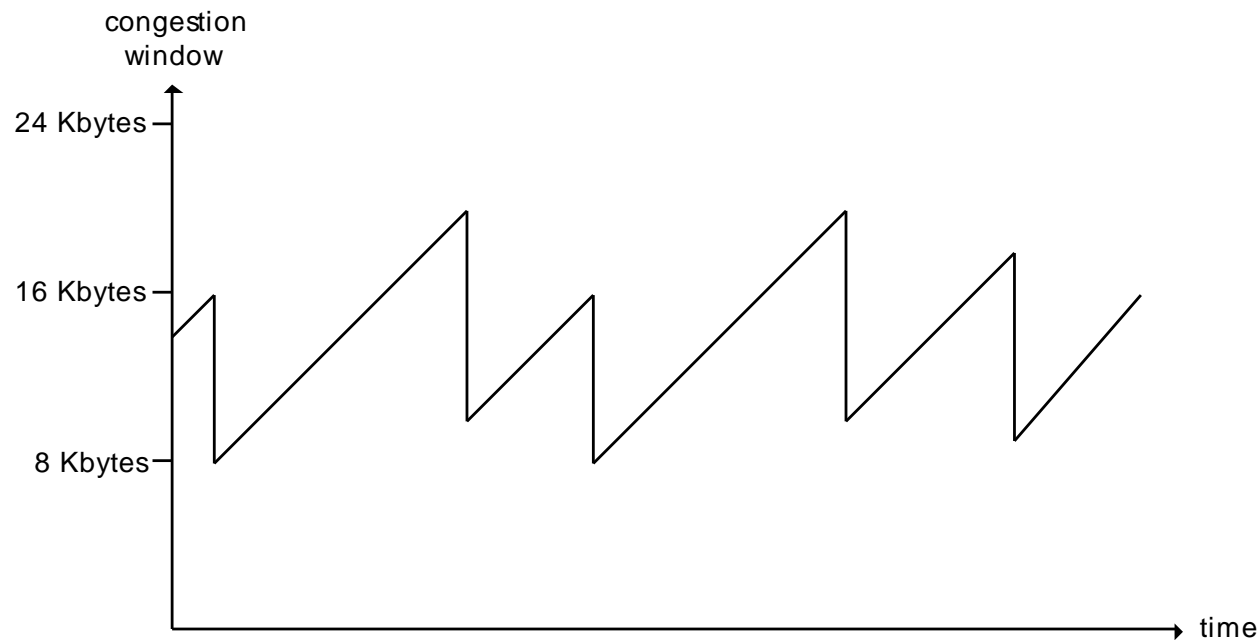
three mechanisms:

- AIMD
- slow start
- conservative after timeout events



multiplicative decrease: cut
CongWin in half after loss
event

additive increase: increase
CongWin by 1 MSS
every RTT in the absence
of loss events: *probing*

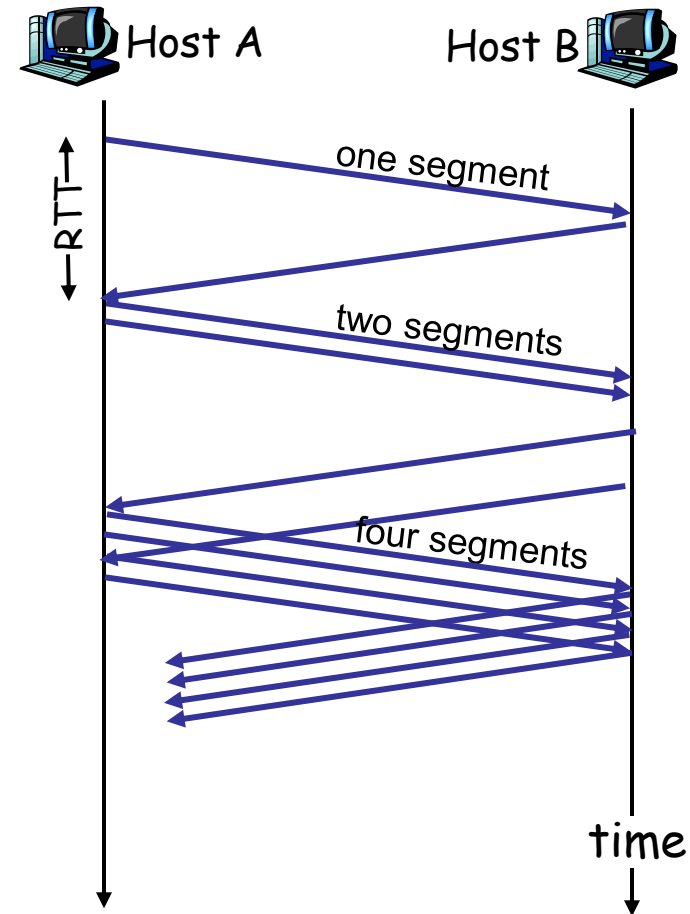




- When connection begins, **CongWin** = 1 MSS
 - Example: MSS = 500 bytes & RTT = 200 msec
 - initial rate = 20 kbps
- available bandwidth may be \gg MSS/RTT
 - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event



- When connection begins, increase rate exponentially until first loss event:
 - » double **CongWin** every RTT
 - » done by incrementing **CongWin** for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast





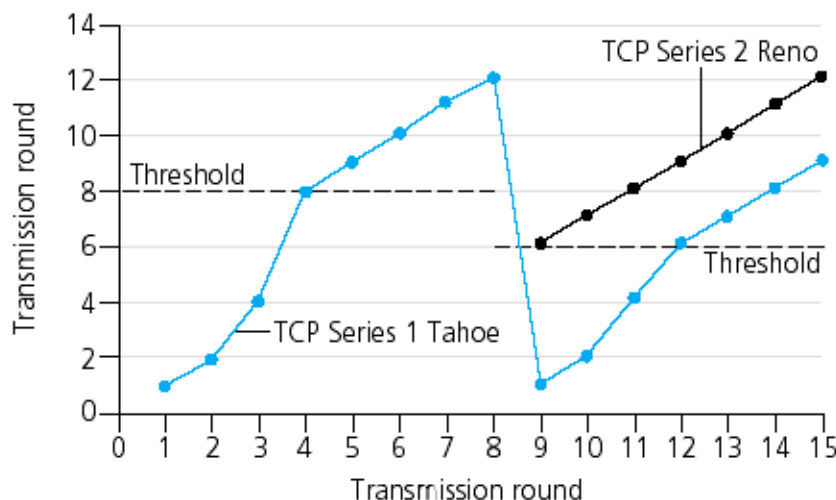
- After 3 dup ACKs:
 - » **CongWin** is cut in half
 - » window then grows linearly
- But after timeout event:
 - » **CongWin** instead set to 1 MSS;
 - » window then grows exponentially
 - » to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"



- **Q:** When should the exponential increase switch to linear?
- **A:** When **CongWin** gets to 1/2 of its value before timeout.



Implementation:

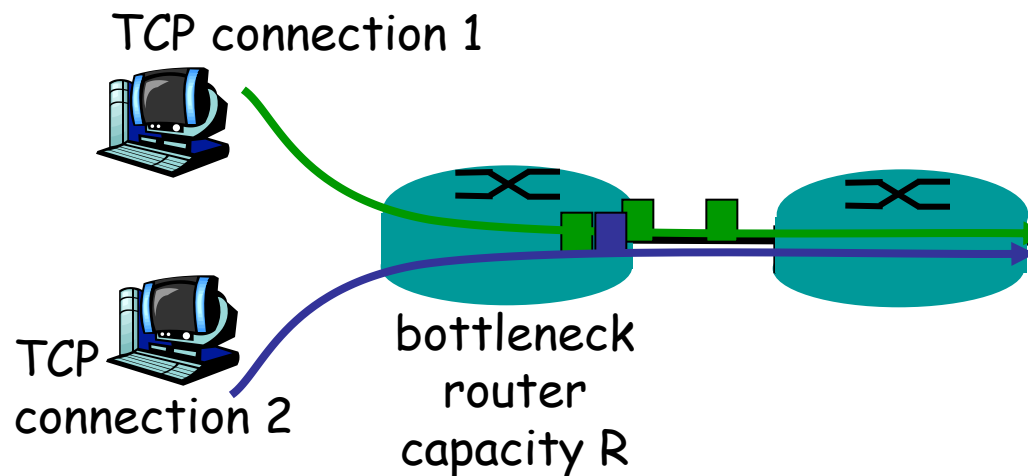
- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event



- When CongWin is below Threshold, sender in **slow-start** phase, window grows exponentially
- When CongWin is above Threshold, sender is in **congestion-avoidance** phase, window grows linearly
- When a **triple duplicate ACK** occurs, Threshold set to CongWin/2 and CongWin set to Threshold
- When **timeout** occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS



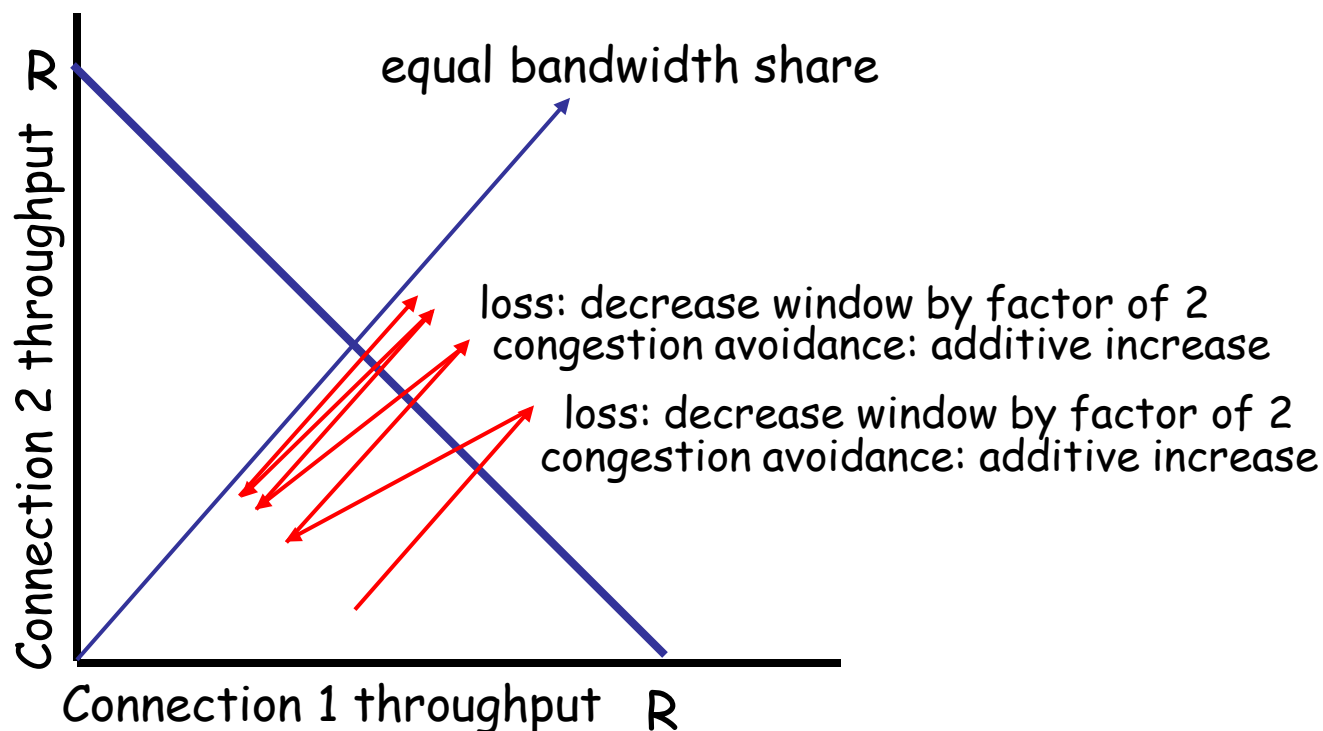
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K





Two competing sessions:

- » Additive increase gives slope of 1, as throughput increases
- » multiplicative decrease decreases throughput proportionally






Fairness and UDP

- Multimedia apps often do not use TCP
 - do not want rate throttled by congestion control
- Instead use UDP:
 - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
 - new app asks for 1 TCP, gets rate $R/10$
 - new app asks for 11 TCPs, gets $R/2$!

Agenda

- 1 Session Overview
- 2 Network Congestion Principles
- 3 Internet Transport Protocols Review
- 4 TCP Congestion Control
-  5 Summary and Conclusion



- Session Overview
- Network Congestion Principles
- Internet Transport Protocols Review
- TCP Congestion Control
- Summary & Conclusion



- Readings



- » Chapter 3 – Sections 3.3, 3.5, 3.6, and 3.7
 - » RFC 793 – Introduction, Sections 1 and 2
 - » RFC 2581
- Assignment #9 previously assigned is due on 05/05/16

