

Chapter 3 Transport Layer



©

Transport Layer 3-1

Chapter 3: Transport Layer

our goals:

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

Transport Layer 3-2

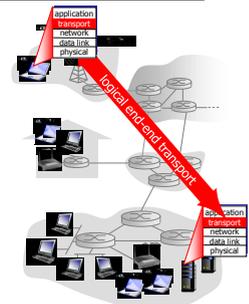
Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-3

Transport services and protocols

- ❖ provide *logical communication* between app processes running on different hosts
- ❖ *transport protocols run in end systems*
 - send side: breaks app messages into **segments**, passes to network layer
 - rcv side: reassembles **segments** into **messages**, passes to application layer



Transport Layer 3-4

Transport vs. network layer

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

household analogy:

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

Transport Layer 3-5

Internet transport-layer protocols

- ❖ reliable, in-order delivery (TCP)
 - congestion control
 - flow control
 - connection setup
- ❖ unreliable, unordered delivery: UDP
 - no-frills extension of "best-efforts" IP
- ❖ services not available:
 - delay guarantees
 - bandwidth guarantees



Transport Layer 3-6

Chapter 3 outline

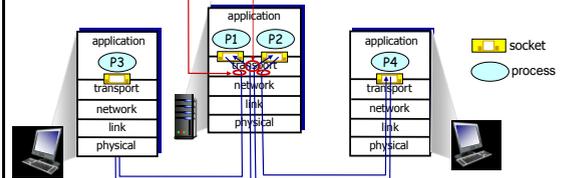
- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-7

Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add **transport header** (later used for demultiplexing)

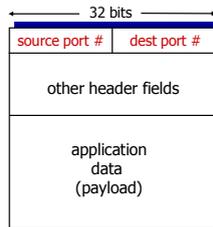
demultiplexing at receiver:
use header info to deliver received segments to correct socket



Transport Layer 3-8

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

Transport Layer 3-9

Connectionless demultiplexing

- ❖ 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

Transport Layer 3-10

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

Transport Layer 3-11

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

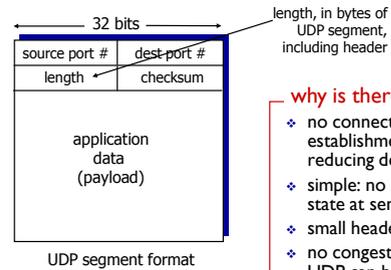
Transport Layer 3-12

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
- ❖ **connectionless:**
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others
- ❖ Typical apps using UDP:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
- ❖ reliable transfer over UDP:
 - add reliability at application layer
 - application-specific error recovery!

Transport Layer 3-13

UDP: segment header



why is there a UDP?

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

Transport Layer 3-14

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. *But maybe errors present, nonetheless*

Transport Layer 3-15

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

Transport Layer 3-16

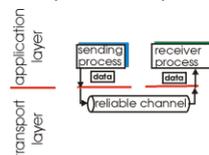
Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-17

Principles of reliable data transfer: Automatic Repeat reQuest (ARQ) Protocols

- ❖ important in application, transport, link layers
 - top-10 list of important networking topics!



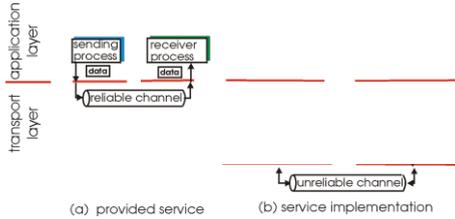
(a) provided service

- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Transport Layer 3-18

Principles of reliable data transfer

- ❖ important in application, transport, link layers
 - top-10 list of important networking topics!

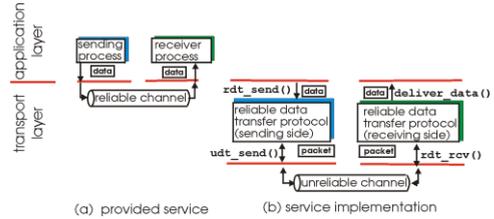


- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Transport Layer 3-19

Principles of reliable data transfer

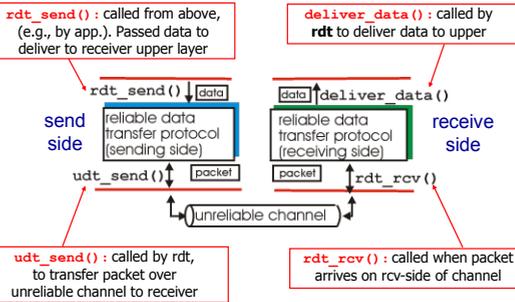
- ❖ important in application, transport, link layers
 - top-10 list of important networking topics!



(a) provided service (b) service implementation

Transport Layer 3-20

Reliable data transfer: getting started

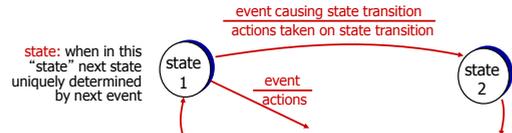


Transport Layer 3-21

Reliable data transfer: getting started

We will:

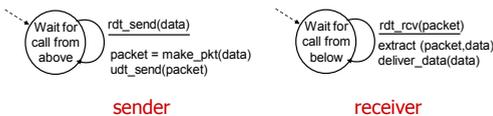
- ❖ incrementally develop Sender and Receiver sides of various reliable data transfer (rdt) protocols
- ❖ consider only unidirectional data transfer
 - but control info will flow on both directions!
- ❖ use Finite State Machines (FSM) to specify Sender and Receiver



Transport Layer 3-22

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



Transport Layer 3-23

rdt2.0: channel with bit errors

- ❖ underlying channel may flip bits in packet
 - checksum to detect bit errors
- ❖ question: how to recover from errors:

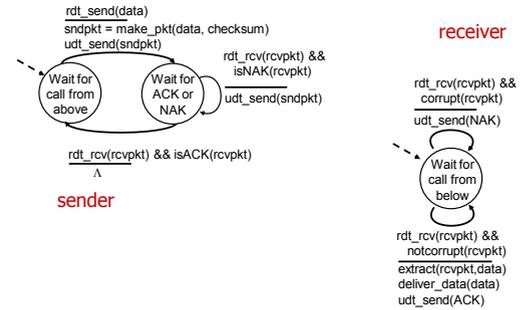
Transport Layer 3-24

rdt2.0: channel with bit errors

- ❖ underlying channel may flip bits in packet
 - checksum to detect bit errors
- ❖ 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**
- ❖ new mechanisms in rdt2.0 (beyond rdt1.0):
 - error detection
 - feedback: control msgs (ACK, NAK) from receiver to sender

Transport Layer 3-25

rdt2.0: FSM specification



Transport Layer 3-26

Break!

Transport Layer 3-27

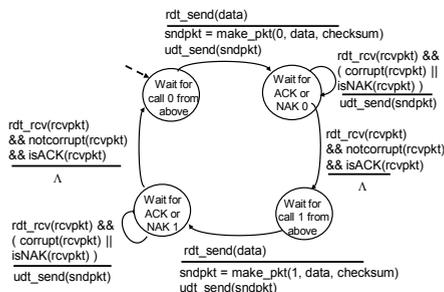
rdt2.0 has a fatal flaw!

- 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 ARQ protocol
sender sends one packet, then waits for receiver response

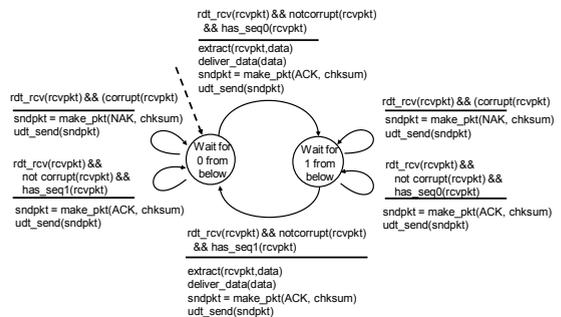
Transport Layer 3-28

rdt2.1: sender handles garbled ACK/NAKs



Transport Layer 3-29

rdt2.1: receiver handles garbled ACK/NAKs



Transport Layer 3-30

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 # 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

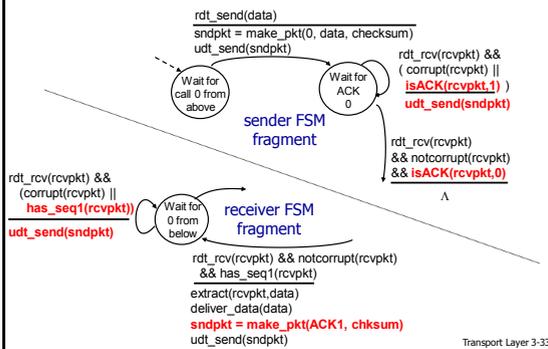
Transport Layer 3-31

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: i.e. **Retransmit current pkt**

Transport Layer 3-32

rdt2.2: sender, receiver fragments



Transport Layer 3-33

rdt3.0: channels with errors and loss

new assumption:

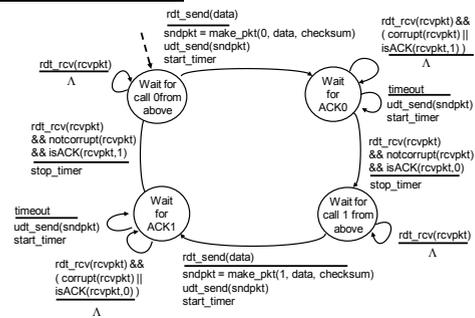
- underlying channel can **also lose** packets, in addition to flipping bits (data, ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

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 pkt being ACKed
- ❖ **requires countdown timer**

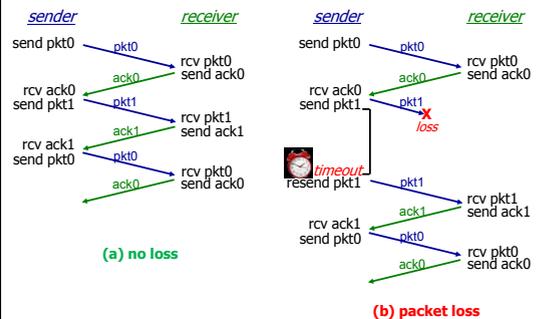
Transport Layer 3-34

rdt3.0 sender

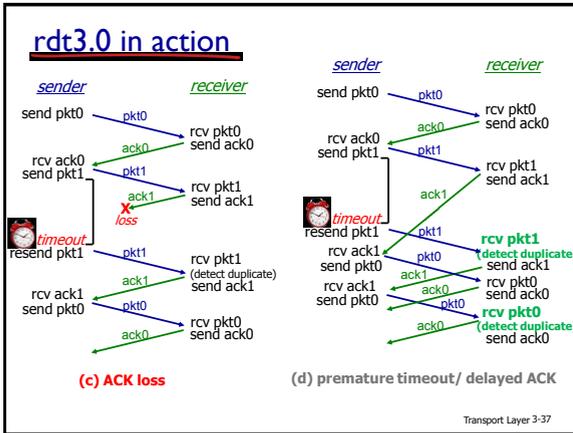


Transport Layer 3-35

rdt3.0 in action



Transport Layer 3-36



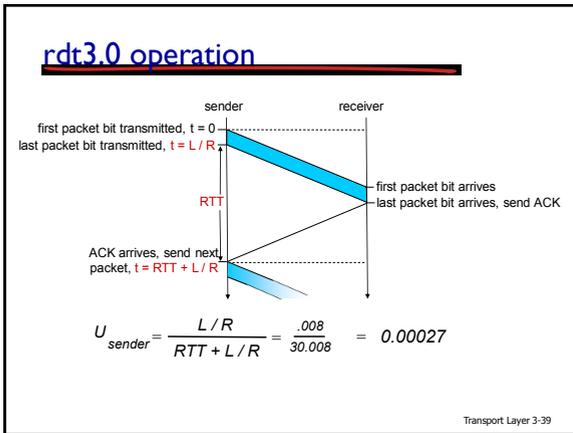
Performance of rdt3.0

- rdt3.0 is foolproof, but performance may be horrible!
- e.g.: 1Gbps link, 15ms propagation delay, 8000bit pkt:

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microseconds}$$
- U_{sender}: utilization (fraction of time sender is busy sending)

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$
- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- Thus network protocol limits use of physical resources!

Transport Layer 3-38



Pipelined protocols

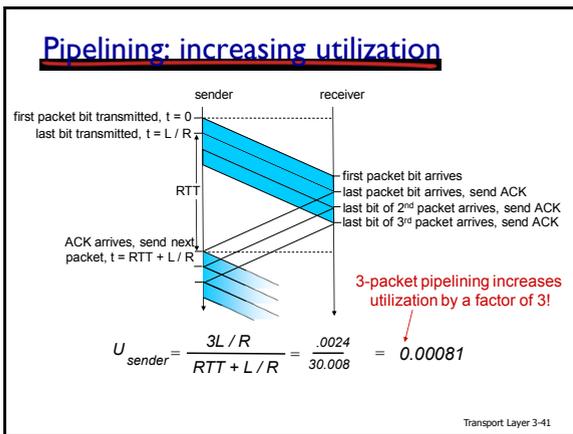
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

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

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

- two generic forms of pipelined protocols: **go-back-n** and **selective repeat**

Transport Layer 3-40



Pipelined ARQ protocols

Go-back-N:

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

Transport Layer 3-42

Go-Back-N: sender

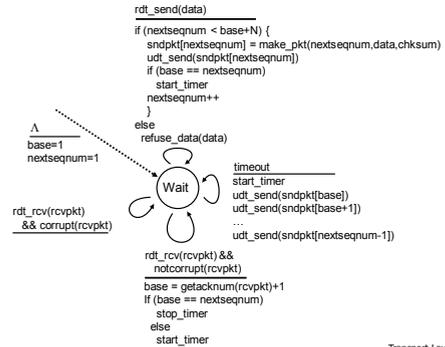
- ❖ k -bit seq # in pkt header
- ❖ "window" of up to N , consecutive unack'd pkts allowed



- ❖ $ACK(n)$: 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

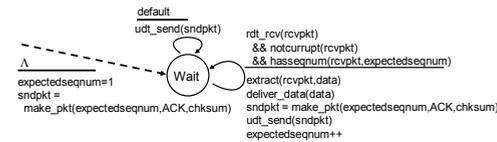
Transport Layer 3-43

GBN: sender extended FSM



Transport Layer 3-44

GBN: receiver extended FSM

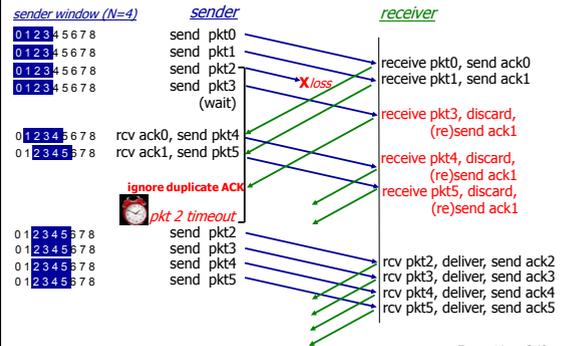


ACK-only: always send ACK for correctly-received pkt with **highest in-order seq #**

- may generate duplicate ACKs
- need only remember **expectedseqnum**
- ❖ out-of-order pkt:
 - **discard (don't buffer): no receiver buffering!**
 - **re-ACK pkt with highest in-order seq #**

Transport Layer 3-45

GBN in action



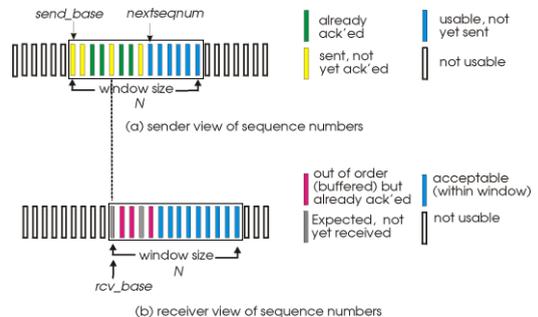
Transport Layer 3-46

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, unACK'ed pkts

Transport Layer 3-47

Selective repeat: sender, receiver windows (different views)



Transport Layer 3-48

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]

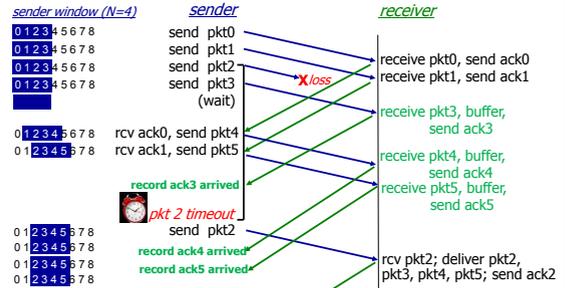
- ❖ ACK(n)

otherwise:

- ❖ ignore

Transport Layer 3-49

Selective repeat in action



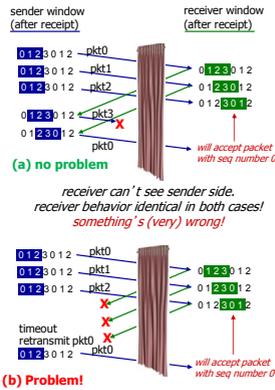
Transport Layer 3-50

Selective Repeat: problem

example:

- ❖ seq #'s: 0, 1, 2, 3
- ❖ window size=3
- ❖ receiver sees no difference in two scenarios!
- ❖ duplicate data accepted as new in (b)

Soln: Window Size must be \leq half the Seq # Space size to avoid problem in (b)



Transport Layer 3-51

Next Class!

Transport Layer 3-52

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer

- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Transport Layer 3-53

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

❖ point-to-point:

- one sender, one receiver

❖ reliable, in-order delivery

❖ pipelined:

- TCP congestion and flow control set window size

❖ full duplex data:

- bi-directional data flow
- MSS: maximum segment size

❖ connection-oriented:

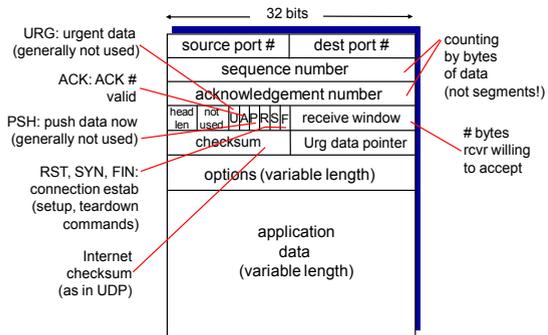
- handshaking before data exchange

❖ flow controlled:

- sender will not overwhelm receiver

Transport Layer 3-54

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

- byte stream "number" of first byte in segment's data

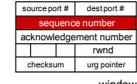
acknowledgements:

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

Q: how does receiver handle out-of-order segments?

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

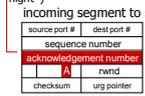
outgoing segment from sender



window size N

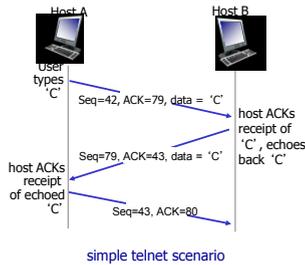


incoming segment to sender



Transport Layer 3-56

TCP seq. nums, ACKs: Telnet example



Transport Layer 3-57

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?

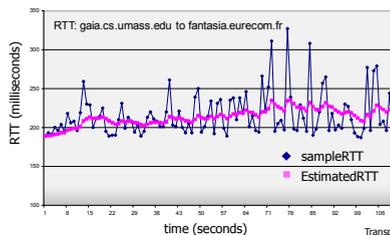
- SampleRTT: measured time from segment transmission until ACK receipt
 - ignore retransmissions
- SampleRTT will vary, want estimated RTT "smoother"
 - average several recent measurements, not just current SampleRTT

Transport Layer 3-58

TCP round trip time, timeout

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

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



Transport Layer 3-59

TCP round trip time, timeout

- timeout interval: EstimatedRTT plus "safety margin"
 - large variation in EstimatedRTT -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

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



↑
estimated RTT

↑
"safety margin"

Transport Layer 3-60

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-61

TCP reliable data transfer

- ❖ TCP creates rdt service on top of IP's unreliable service
 - pipelined segments
 - cumulative acks
 - single retransmission timer
- ❖ retransmissions triggered by:
 - timeout events
 - duplicate acks

Transport Layer 3-62

TCP sender events:

data rcvd from app:

- ❖ create segment with seq #
- ❖ seq # is byte-stream number of first data byte in segment
- ❖ start timer
- ❖ think of timer as for oldest unack'ed segment
 - expiration interval: `TimeoutInterval`

timeout:

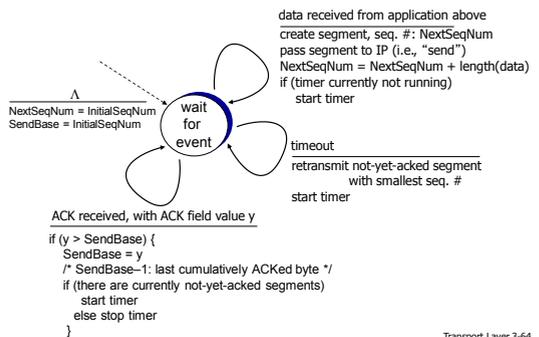
- ❖ retransmit segment that caused timeout
- ❖ restart timer

ack rcvd:

- ❖ if ack acknowledges previously unack'ed segments
 - update what is known to be ack'ed
 - start timer if there are still unack'ed segments

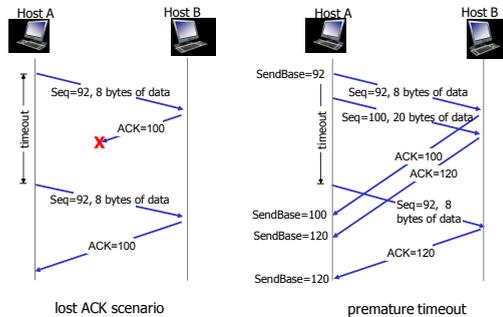
Transport Layer 3-63

TCP sender (simplified)



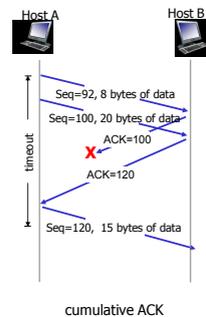
Transport Layer 3-64

TCP: retransmission scenarios



Transport Layer 3-65

TCP: retransmission scenarios



Transport Layer 3-66

TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	TCP receiver action
arrival of in-order segment with expected seq #. All data up to expected seq # already ACK'ed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expected seq. #. Gap detected	immediately send <i>duplicate ACK</i> , indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

Transport Layer 3-67

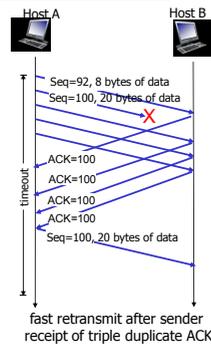
TCP fast retransmit

- time-out period often relatively long:
 - long delay before retransending 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 duplicate ACKs for ("triple duplicate ACKs"), resend unack'ed segment with smallest seq #
 likely that unack'ed segment lost, so don't wait for timeout

Transport Layer 3-68

TCP fast retransmit



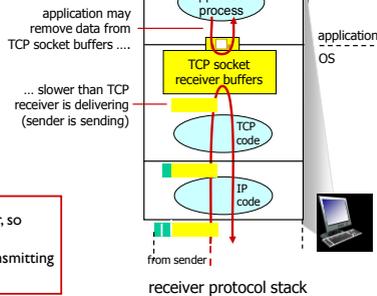
Transport Layer 3-69

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-70

TCP flow control

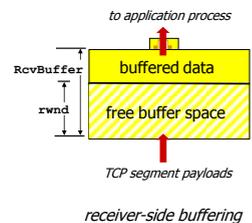


flow control
 receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

Transport Layer 3-71

TCP flow control

- receiver "advertises" free buffer space by including **rwnd** (Rcv Window) 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 unack'ed ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



Transport Layer 3-72

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-73

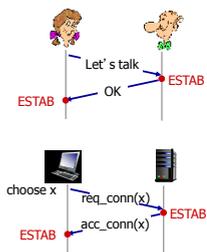
Connection Management

- before exchanging data, sender/receiver “handshake”:
- ❖ agree to establish connection (each knowing the other willing to establish connection)
 - ❖ agree on connection parameters

Transport Layer 3-74

Agreeing to establish a connection

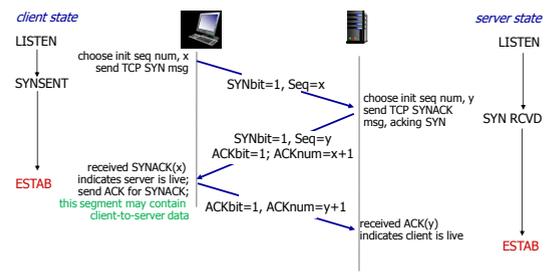
2-way handshake:



- 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

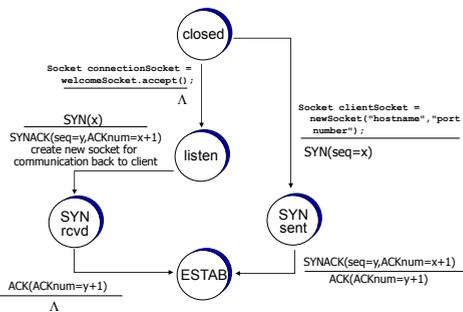
Transport Layer 3-75

TCP 3-way handshake



Transport Layer 3-76

TCP 3-way handshake: FSM



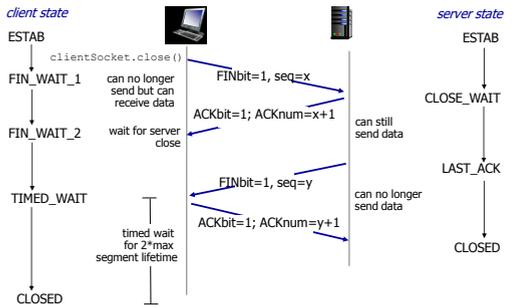
Transport Layer 3-77

TCP: closing a connection

- ❖ client, server each close their side of connection
 - send TCP segment with FIN bit = 1
- ❖ respond to received FIN with ACK

Transport Layer 3-78

TCP: closing a connection



Transport Layer 3-79

Break!

Transport Layer 3-80

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport Layer 3-81

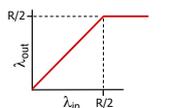
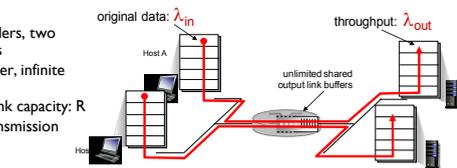
Principles of congestion control

- ❖ informally: “too many sources sending too much data too fast for *network* to handle”
- ❖ manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)

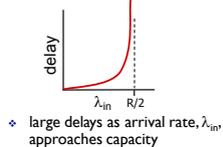
Transport Layer 3-82

Causes/costs of congestion: scenario 1

- ❖ two senders, two receivers
- ❖ one router, infinite buffers
- ❖ output link capacity: R
- ❖ no retransmission



- ❖ maximum per-connection throughput: $R/2$

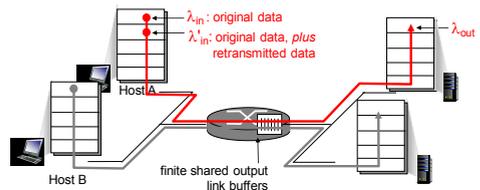


- ❖ large delays as arrival rate, λ_{in} , approaches capacity

Transport Layer 3-83

Causes/costs of congestion: scenario 2

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

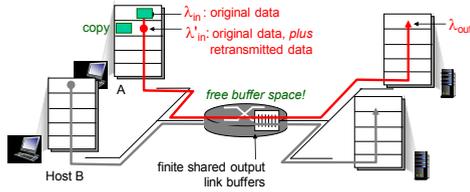
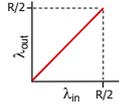


Transport Layer 3-84

Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- ❖ sender sends only when router buffers available



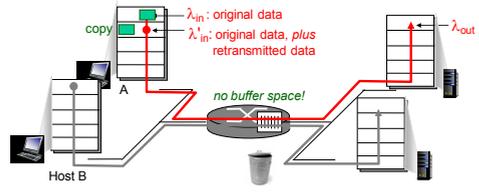
Transport Layer 3-85

Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- ❖ sender only resends if packet known to be lost



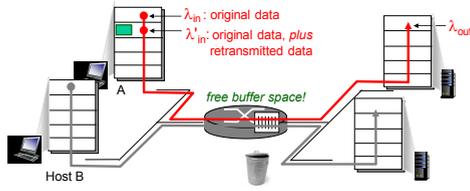
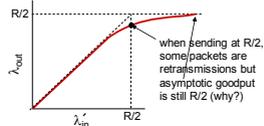
Transport Layer 3-86

Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- ❖ sender only resends if packet known to be lost



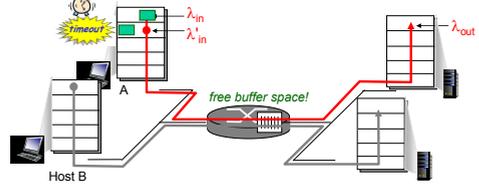
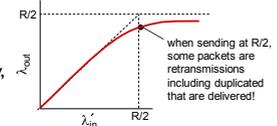
Transport Layer 3-87

Causes/costs of congestion: scenario 2

Realistic: duplicates

❖ packets can be lost, dropped at router due to full buffers

- ❖ sender times out prematurely, sending two copies, both of which are delivered

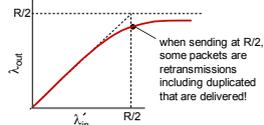


Transport Layer 3-88

Causes/costs of congestion: scenario 2

Realistic: duplicates

- ❖ packets can be lost, dropped at router due to full buffers
- ❖ sender times out prematurely, sending two copies, both of which are delivered



“costs” of congestion:

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

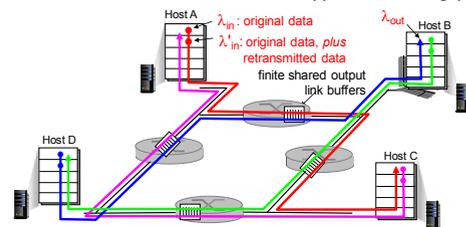
Transport Layer 3-89

Causes/costs of congestion: scenario 3

- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit

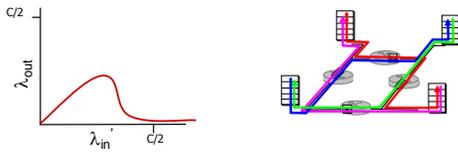
Q: what happens as λ_{in} and λ_{in}' increase?

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



Transport Layer 3-90

Causes/costs of congestion: scenario 3



another "cost" of congestion:

- ❖ when packet dropped, any "upstream" transmission capacity used for that packet was wasted!

Transport Layer 3-91

Approaches towards congestion control

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

network-assisted congestion control:

- ❖ routers provide feedback to end systems
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
 - explicit rate for sender to send at

Transport Layer 3-92

Chapter 3 outline

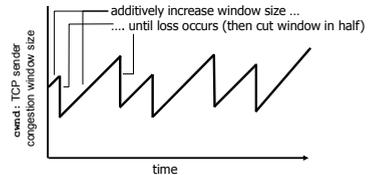
- | | |
|--|---|
| 3.1 transport-layer services | 3.5 connection-oriented transport: TCP <ul style="list-style-type: none"> ▪ segment structure ▪ reliable data transfer ▪ flow control ▪ connection management |
| 3.2 multiplexing and demultiplexing | |
| 3.3 connectionless transport: UDP | 3.6 principles of congestion control |
| 3.4 principles of reliable data transfer | 3.7 TCP congestion control |

Transport Layer 3-93

TCP congestion control: additive increase multiplicative decrease

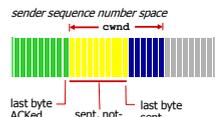
- ❖ **approach:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
 - **additive increase:** increase **cwnd** (Congestion Window) by 1 (one) MSS every RTT until loss detected
 - **multiplicative decrease:** cut **cwnd** in half after loss

AIMD sawtooth behavior: probing for bandwidth



Transport Layer 3-94

TCP Congestion Control: details



❖ sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

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

TCP sending rate:

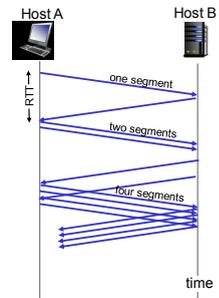
- ❖ **roughly:** send **cwnd** bytes, wait RTT for ACKs, then send more bytes

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

Transport Layer 3-95

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



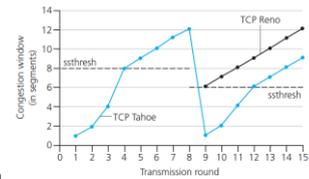
Transport Layer 3-96

TCP: detecting, reacting to loss

- ❖ loss indicated by timeout:
 - **cwnd** set to 1 MSS;
 - window then grows exponentially (as in slow start) to threshold, then grows linearly
- ❖ loss indicated by 3 duplicate ACKs: TCP RENO
 - dup ACKs indicate network capable of delivering some segments
 - **cwnd** is cut in half window then grows linearly
- ❖ TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

Transport Layer 3-97

TCP: switching from slow start to CA

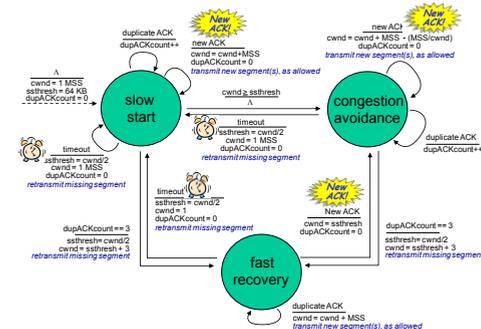


Implementation:

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

Transport Layer 3-98

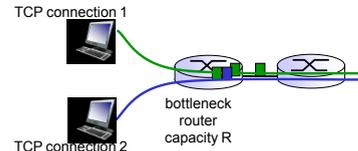
Summary: TCP Congestion Control



Transport Layer 3-99

TCP Fairness

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

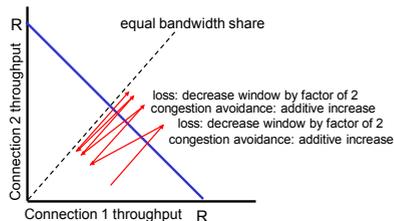


Transport Layer 3-100

Why is TCP fair?

two competing sessions:

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



Transport Layer 3-101

Fairness (more)

Fairness and UDP

- ❖ multimedia apps often do not use TCP
 - do not want rate throttled by congestion control
- ❖ instead use UDP:
 - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

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

Transport Layer 3-102