Topic 5 – Transport

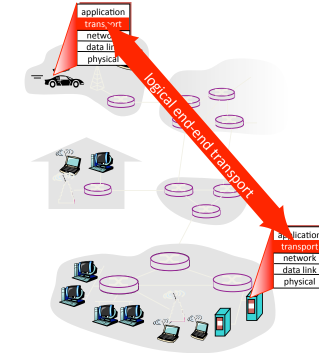
Our goals:

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

2

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 app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



3

Transport vs. network layer

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

Household analogy:

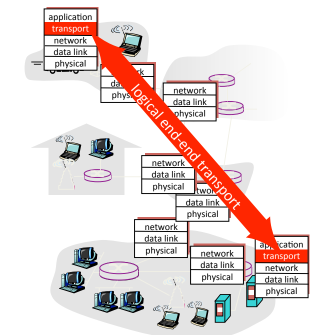
12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

4

Internet transport-layer protocols

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

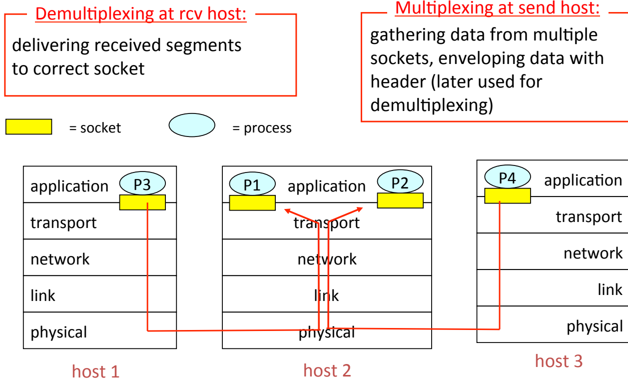


5

Multiplexing/demultiplexing
(Transport-layer style)

Demultiplexing at rcv host:
delivering received segments to correct socket

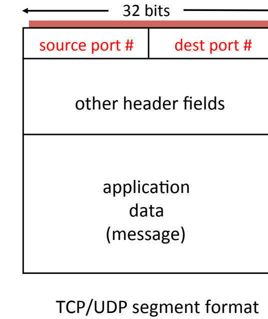
Multiplexing at send host:
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



6

How transport-layer demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 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

7

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

- UDP socket identified by two-tuple:

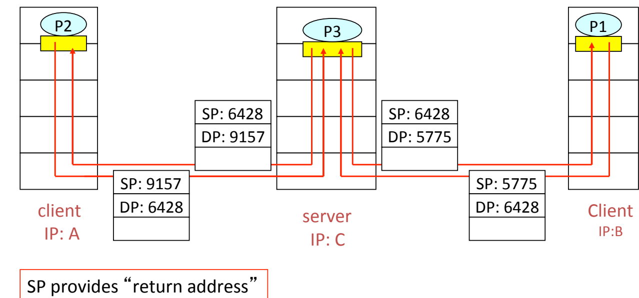
(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

8

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



SP provides “return address”

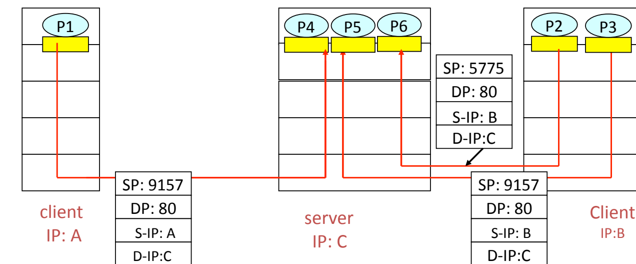
9

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host 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

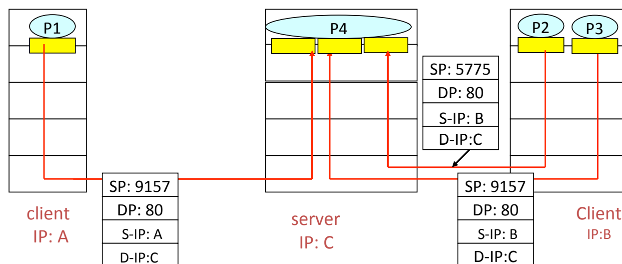
10

Connection-oriented demux (cont)



11

Connection-oriented demux: Threaded Web Server



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 to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

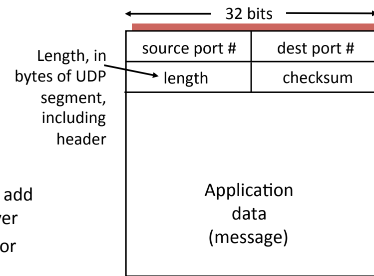
Why is there a UDP?

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

13

UDP: more

- 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 recovery!



UDP segment format

14

UDP checksum

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

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

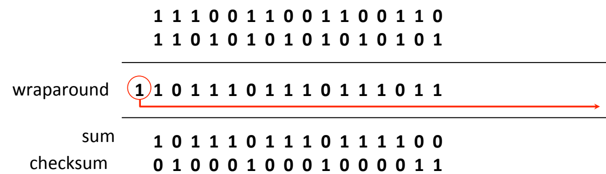
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....

15

Internet Checksum

(time travel warning – we covered this earlier)

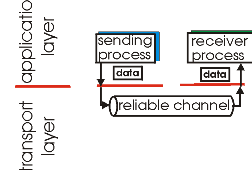
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers



16

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



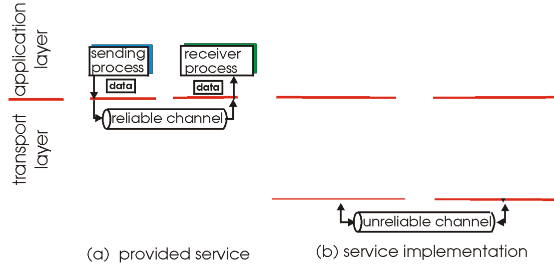
(a) provided service

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

17

Principles of Reliable data transfer

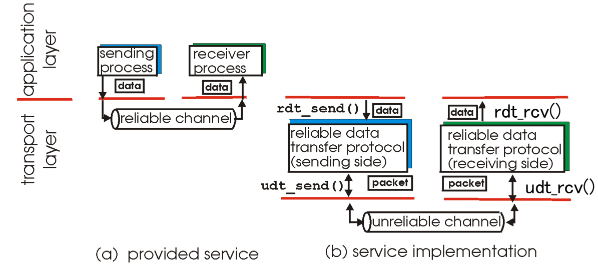
- important in app., transport, link layers
- top-10 list of important networking topics!



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

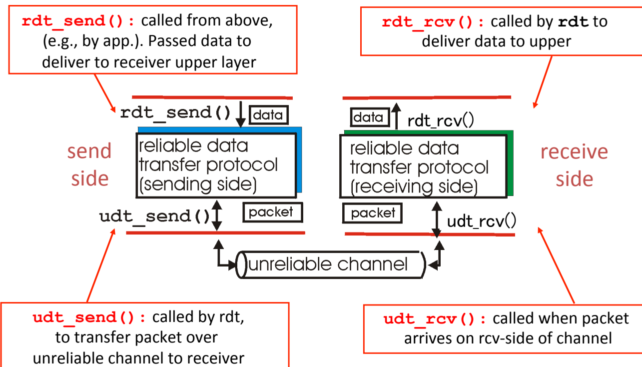
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!



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

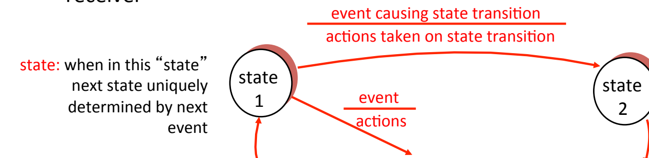
Reliable data transfer: getting started



Reliable data transfer: getting started

We'll:

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



KR state machines – a note.

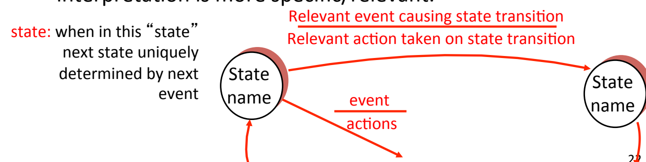
Beware

Kurose and Ross has a confusing/confused attitude to state-machines.

I've attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which the RDT example is taken.)

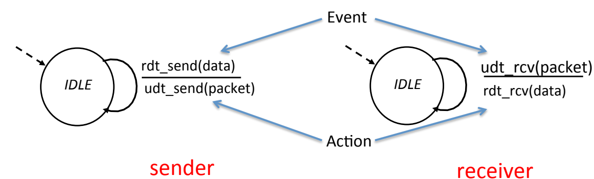
in KR "actions taken" appear wide-ranging, my interpretation is more specific/relevant.



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 read data from underlying channel



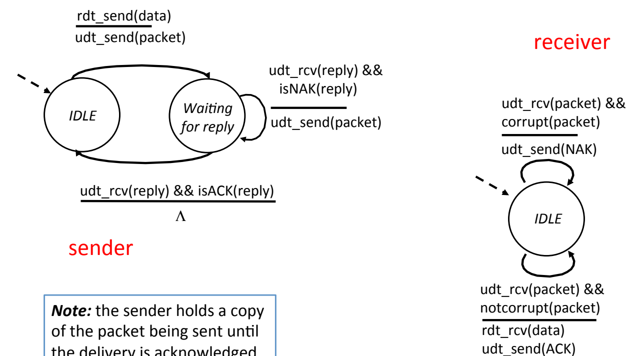
23

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) receiver->sender

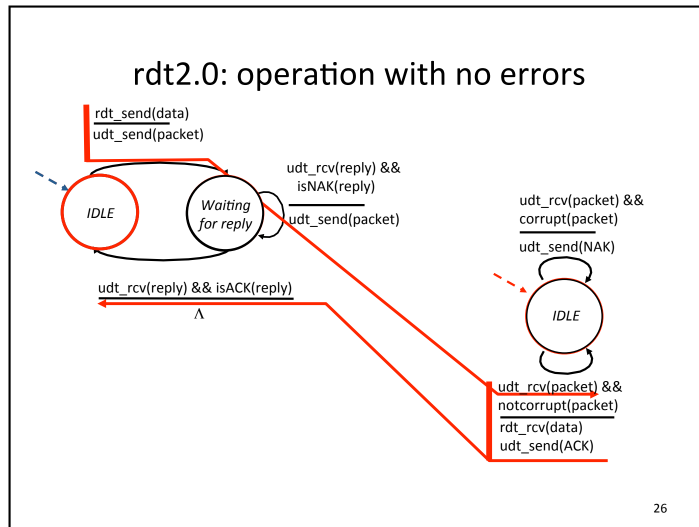
24

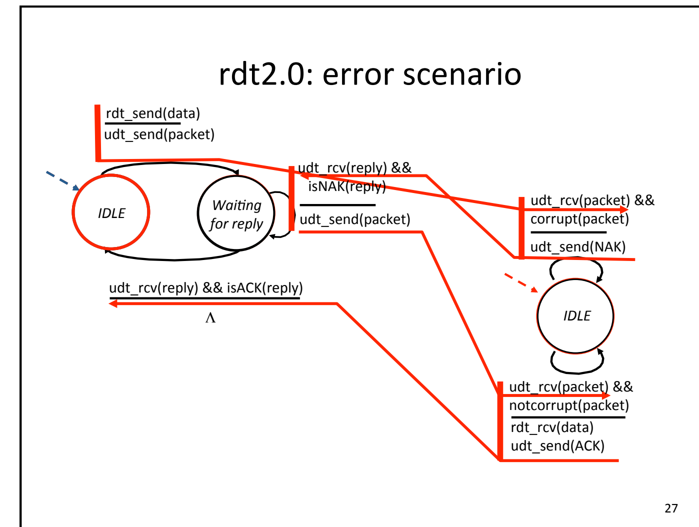
rdt2.0: FSM specification



Note: the sender holds a copy of the packet being sent until the delivery is acknowledged.

25





rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

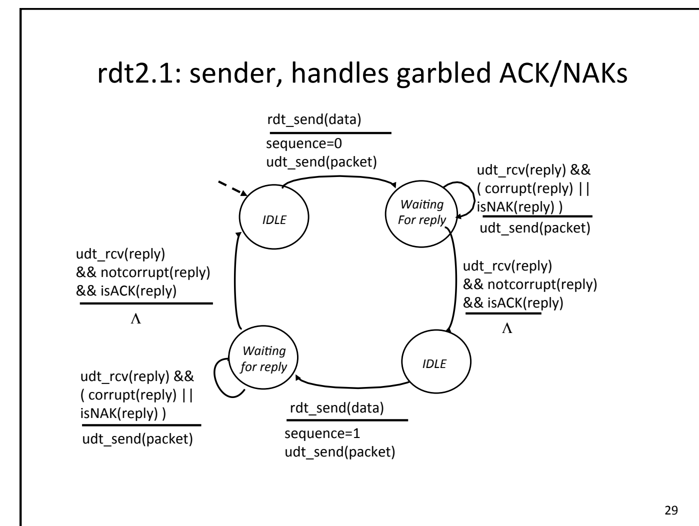
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

Handling duplicates:

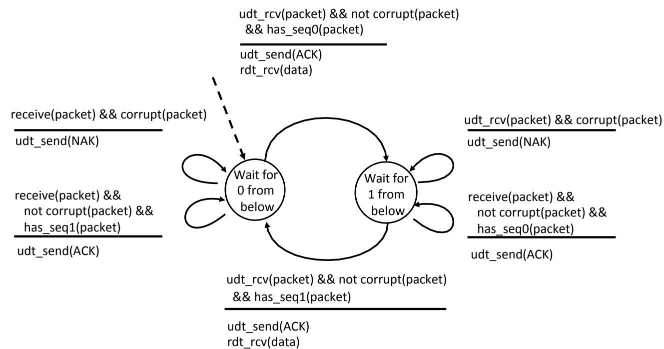
- sender retransmits current packet if ACK/NAK garbled
- sender adds *sequence number* to each packet
- receiver discards (doesn't deliver) duplicate packet

stop and wait
Sender sends one packet, then waits for receiver response

28



rdt2.1: receiver, handles garbled ACK/NAKs



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 "current" pkt has a 0 or 1 sequence number

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

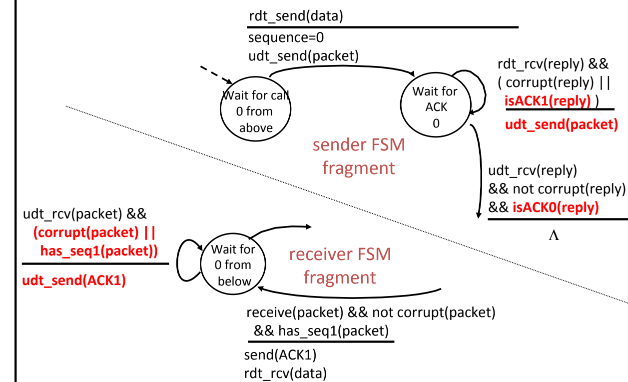
31

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: *retransmit current pkt*

32

rdt2.2: sender, receiver fragments



33

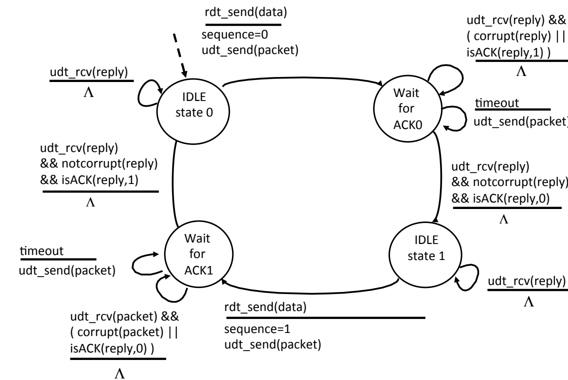
rdt3.0: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data or 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 use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

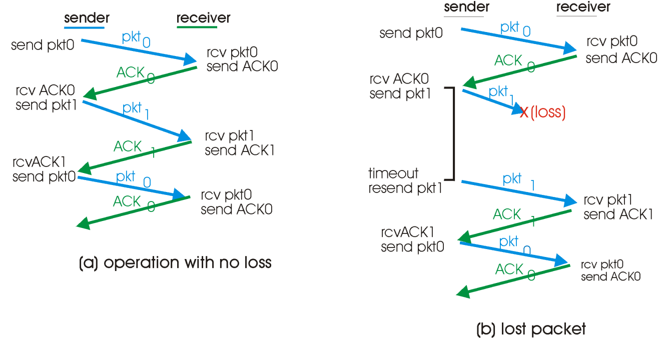
34

rdt3.0 sender



35

rdt3.0 in action

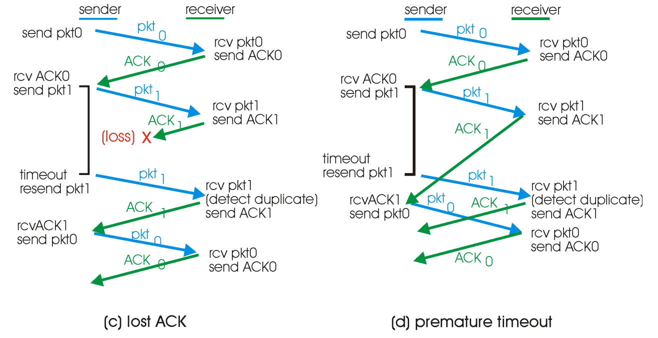


(a) operation with no loss

(b) lost packet

36

rdt3.0 in action



(c) lost ACK

(d) premature timeout

37

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

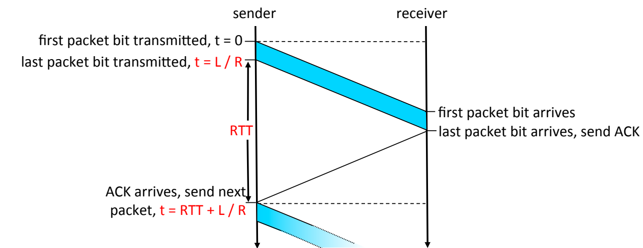
- U_{sender}: utilization – fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

38

rdt3.0: stop-and-wait operation

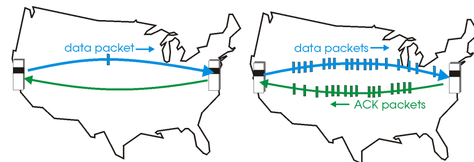


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

39

Pipelined (Packet-Window) 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

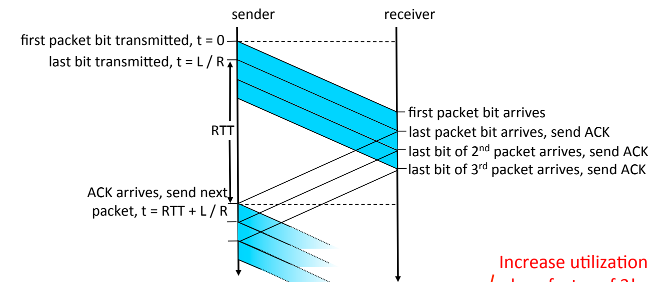


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*

40

Pipelining: increased utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

41

Pipelining Protocols

Go-back-N: big picture:
- Sender can have up to N unacked packets in pipeline
- Rcvr only sends cumulative acks
  - Doesn't ack packet if there's a gap
- Sender has timer for oldest unacked packet
  - If timer expires, retransmit all unacked packets

Selective Repeat: big pic
- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

42

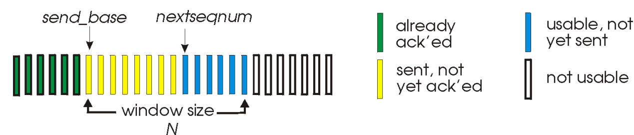
Selective repeat: big picture

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

43

Go-Back-N

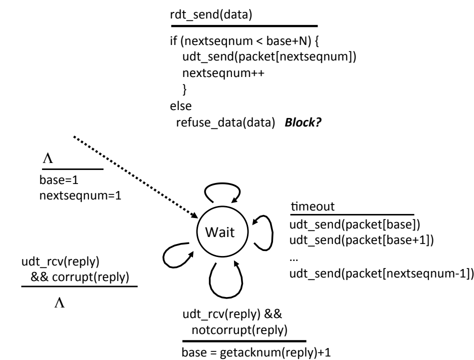
Sender:
- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



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

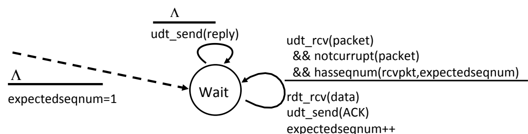
44

GBN: sender extended FSM



45

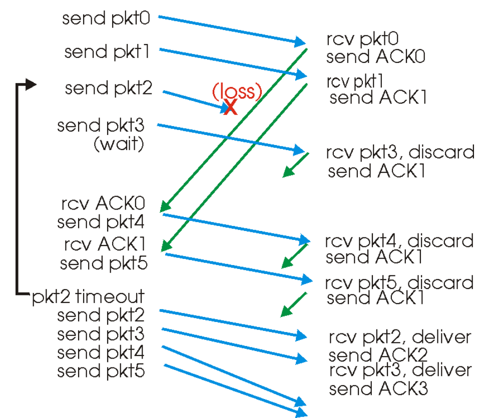
GBN: receiver extended FSM



ACK-only: always send an ACK for correctly-received packet with the highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order packet:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK packet with highest in-order seq #

46



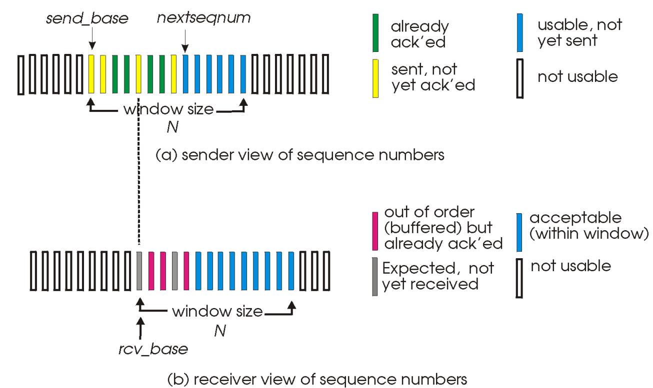
47

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
  - again limits seq #s of sent, unACKed pkts

48

Selective repeat: sender, receiver windows



49

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

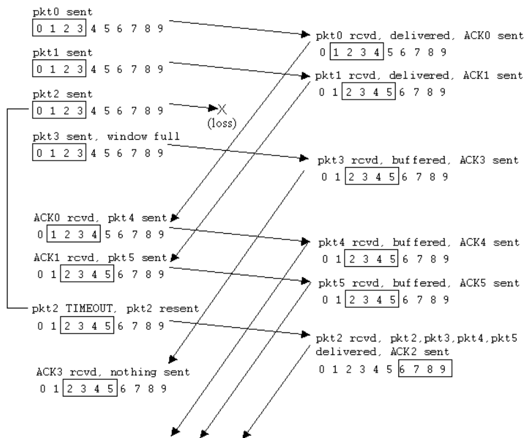
pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1]
- ACK(n)

otherwise:
- ignore

50

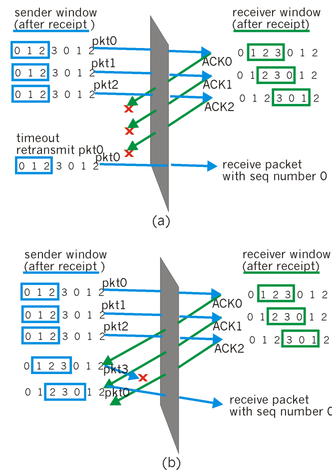
Selective repeat in action



51

Selective repeat: dilemma

Example:
- seq #'s: 0, 1, 2, 3
- window size=3



- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?

window size ≤ (½ of seq # size)

52

Automatic Repeat Request (ARQ)

+ Self-clocking (Automatic)

+ Adaptive

+ Flexible

- Slow to start / adapt

consider high Bandwidth/Delay product

Now lets move from the generic to the specific….

TCP arguably the most successful protocol in the Internet…..

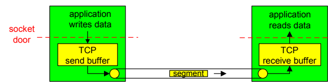
its an ARQ protocol

53

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581, …

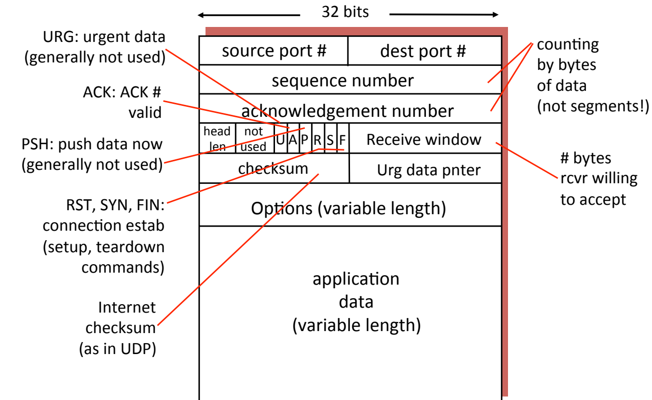
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



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

54

TCP segment structure



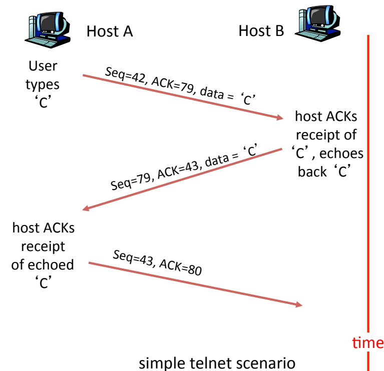
55

TCP seq. #'s and ACKs

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



simple telnet scenario

This has led to a world of hurt….

56

TCP out of order attack

- ARQ with SACK means recipient needs copies of all packets
- Evil attack one:
  send a long stream of TCP data to a server but don't send the first byte
- Recipient keeps all the subsequent data and waits…..
  - Filling buffers.
    - Critical buffers…

- Send a legitimate request

GET index.html

this gets through an intrusion-detection system

then send a new segment replacing bytes 4-13 with "password-file"

A dumb example.

Neither of these attacks would work on a modern system.

57

TCP Round Trip Time and 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`

58

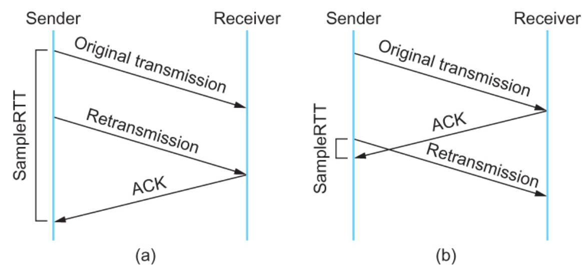
TCP Round Trip Time and Timeout

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

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

59

Some RTT estimates are never good



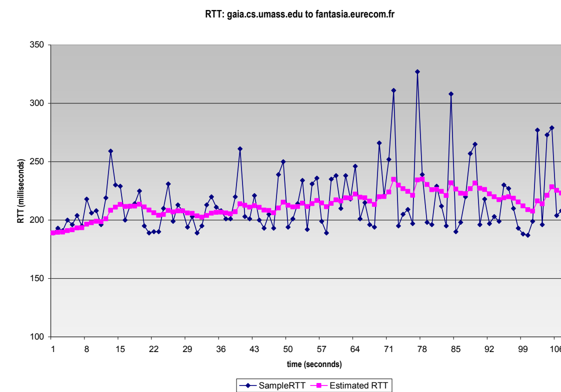
Associating the ACK with (a) original transmission versus (b) retransmission

Karn/Partridge Algorithm – Ignore retransmission in measurements

(and increase timeout; this makes retransmissions decreasingly aggressive)

60

Example RTT estimation:



61

TCP Round Trip Time and Timeout

Setting the timeout

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)
```

Then set timeout interval:

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

62

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

63

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 acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

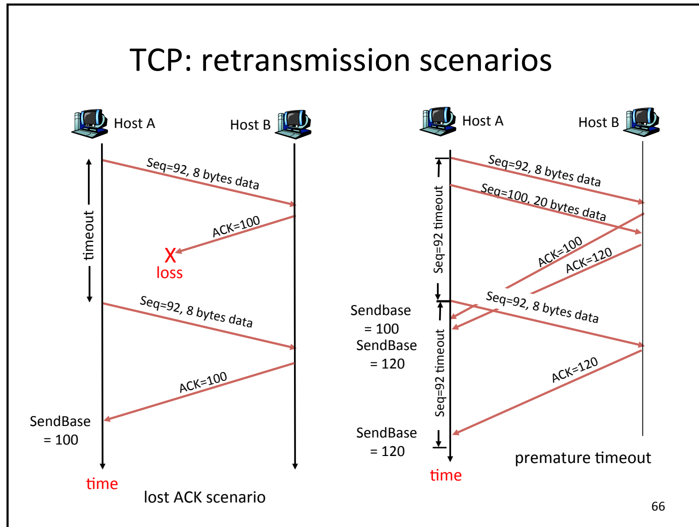
64

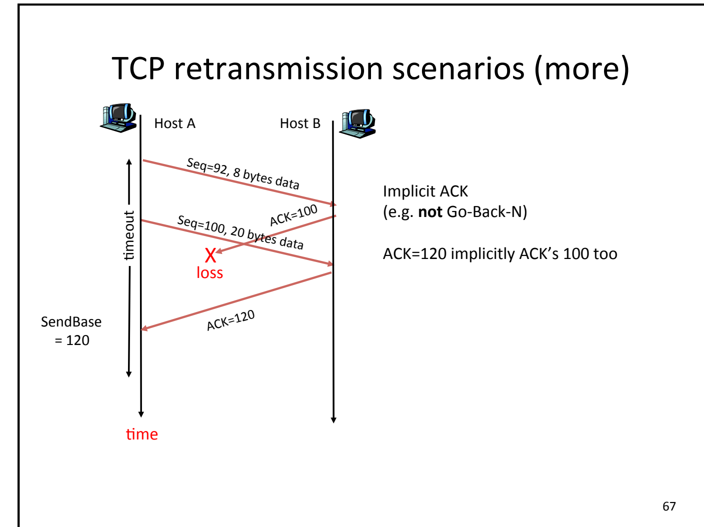
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

 } /* end of loop forever */
```

Comment:
• SendBase-1: last cumulatively ack'ed byte
Example:
• SendBase-1 = 71;
y= 73, so the rcvr wants 73+ ;
y > SendBase, so that new data is acked

65





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 ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, 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

68

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.
- If sender receives 3 duplicate ACKs, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

69

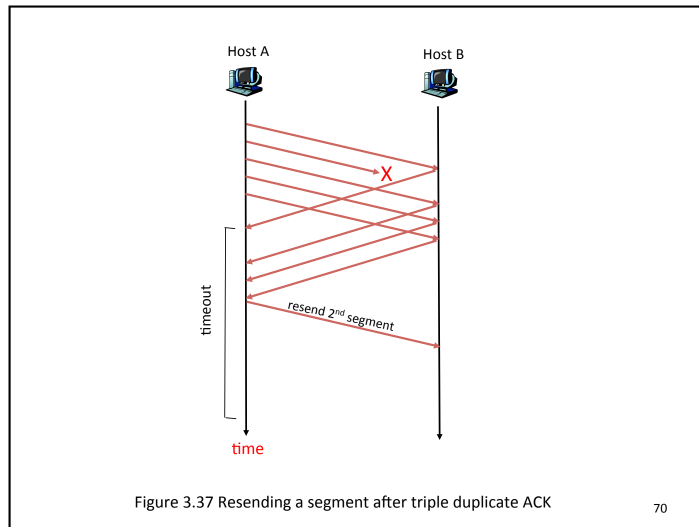


Figure 3.37 Resending a segment after triple duplicate ACK

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
         if (y > SendBase) {
             SendBase = y
             if (there are currently not-yet-acknowledged segments)
                 start timer
           }
         else {
              increment count of dup ACKs received for y
              if (count of dup ACKs received for y = 3) {
                  resend segment with sequence number y
              }
```

a duplicate ACK for already ACKed segment

fast retransmit

71

Silly Window Syndrome

(a) Sender — MSS MSS — Data — ACKS — MSS MSS — Receiver

(b) Sender — <MSS MSS — Data — ACKS — MSS <MSS — Receiver

MSS advertises the amount a receiver can accept

If a transmitter has something to send – it will.

This means small MSS values may persist
- indefinitely.

Solution

Wait to fill each segment, but don't wait indefinitely.

NAGLE's Algorithm

If we wait too long interactive traffic is difficult

If we don't want we get *silly window* syndrome

Solution: Use a timer, when the timer expires – send the (unfilled) segment.

72

Flow Control ≠ Congestion Control

- ***Flow control*** involves preventing senders from overrunning the capacity of the receivers
- ***Congestion control*** involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded

73

Flow Control – (bad old days?)

In-Line flow control

- **XON/XOFF** (^s/^q)
- **data-link** dedicated symbols aka Ethernet
  (more in the Advanced Topic on Datacenters)

Dedicated wires

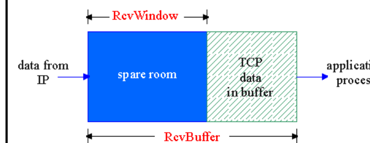
- **RTS/CTS** handshaking
- **Read (or Write) Ready**
  signals from memory interface saying slow-down/stop…

74

TCP Flow Control

- receive side of TCP connection has a receive buffer:

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

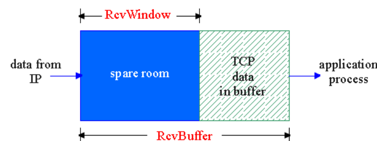


- speed-matching service: matching the send rate to the receiving app's drain rate

- app process may be slow at reading from buffer

75

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of `RcvWindow` in segments
- Sender limits unACKed data to `RcvWindow`
  - guarantees receive buffer doesn't overflow

76

TCP Connection Management

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 host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

77

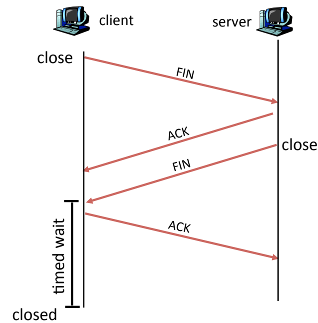
TCP Connection Management (cont.)

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.



78

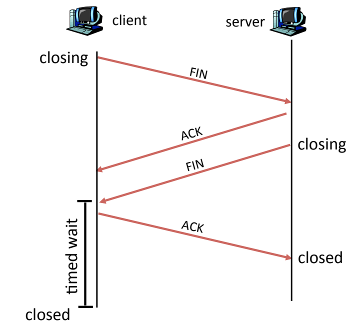
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

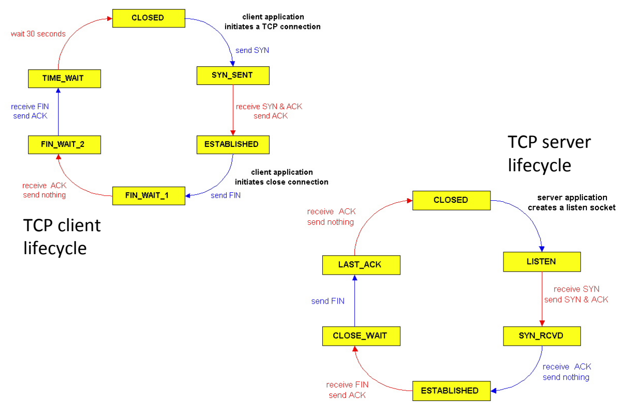
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



79

TCP Connection Management (cont)



80

Principles of Congestion Control

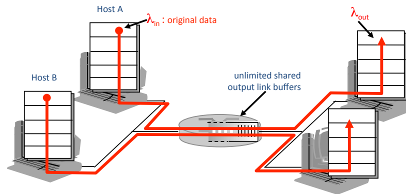
Congestion:

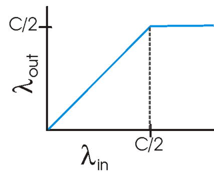
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

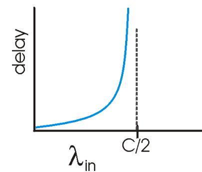
81

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission





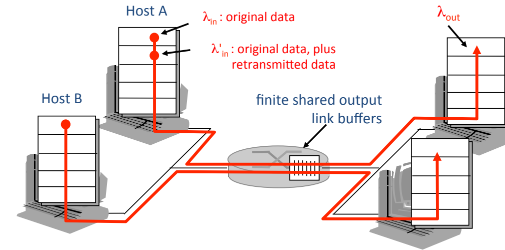


- large delays when congested
- maximum achievable throughput

82

Causes/costs of congestion: scenario 2

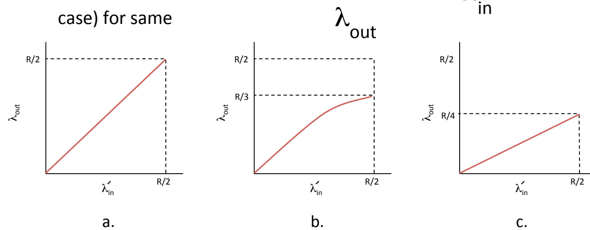
- one router, *finite* buffers
- sender retransmission of lost packet



83

Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "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}



"costs" of congestion:

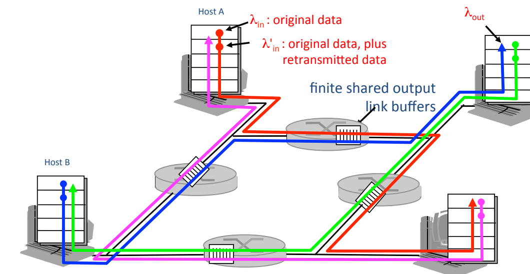
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

84

Causes/costs of congestion: scenario 3

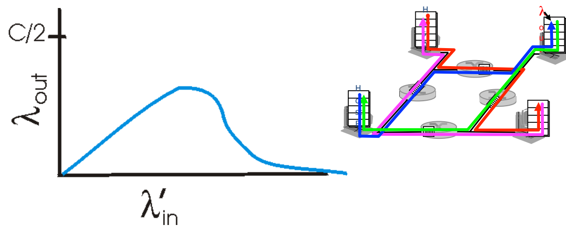
- four senders
- multihop paths
- timeout/retransmit

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



85

Causes/costs of congestion: scenario 3



Another "cost" of congestion:

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

Congestion Collapse example: Cocktail party effect

86

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:

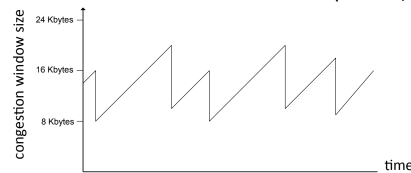
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

87

TCP congestion control: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **CongWin** by 1 MSS every RTT for each received ACK until loss detected (W ←W + 1/W)
  - *multiplicative decrease*: cut **CongWin** in half after loss (W←W/2)

Saw tooth behavior: probing for bandwidth



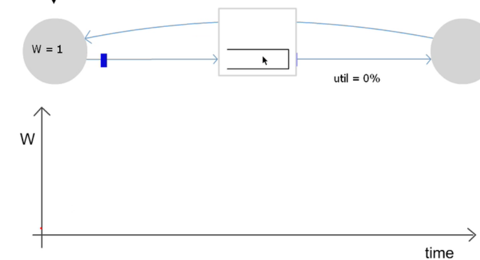
88

Continuous ARQ (TCP) adapting to congestion

Only W packets may be outstanding

Rule for adjusting W

- If an ACK is received: W ← W+1/W
- If a packet is lost: W ← W/2



SLOW START IS NOT SHOWN!

89

TCP Congestion Control: details

- sender limits transmission:
  LastByteSent-LastByteAcked ≤ 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

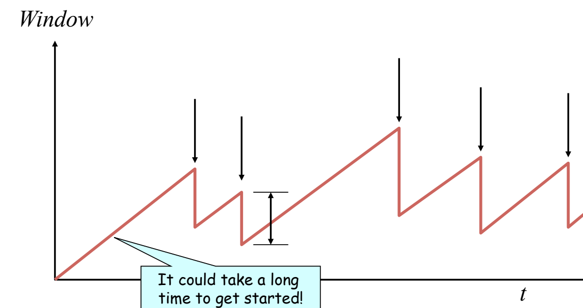
three mechanisms:

- AIMD
- slow start
- conservative after timeout events

90

AIMD Starts Too Slowly!

Need to start with a small CWND to avoid overloading the network.



91

TCP Slow Start

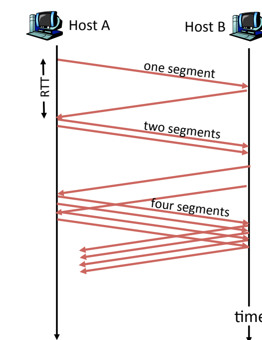
- When connection begins, **CongWin** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until first loss event

92

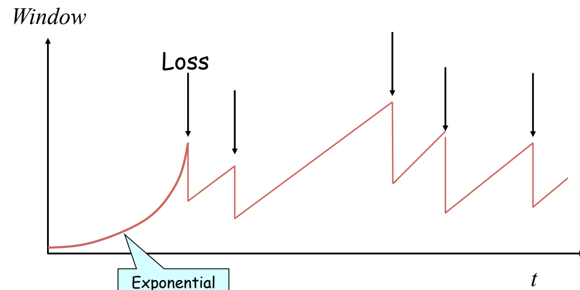
TCP Slow Start (more)

- 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



93

Slow Start and the TCP Sawtooth



Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

94

Refinement: inferring loss

- 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:

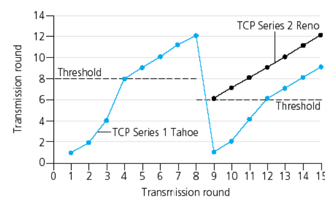
- ❑ 3 dup ACKs indicates network capable of delivering some segments
- ❑ timeout indicates a "more alarming" congestion scenario

95

Refinement

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

96

Summary: TCP Congestion Control

- 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.

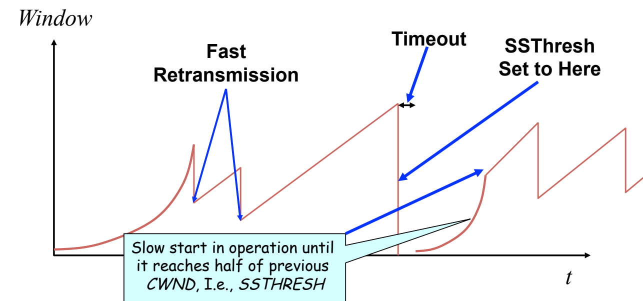
97

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

98

Repeating Slow Start After Timeout



Slow-start restart: Go back to CWND of 1 MSS, but take advantage of knowing the previous value of CWND.

99

TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

100

TCP Futures: TCP over "long, fat pipes"

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate *p*:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{p}}$$

- ➜ L = $2 \cdot 10^{-10}$ *Ouch!*
- New versions of TCP for high-speed

101

Calculation on Simple Model
(cwnd in units of MSS)

- Assume loss occurs whenever cwnd reaches W
  - Recovery by fast retransmit
- Window: W/2, W/2+1, W/2+2, ...W, W/2, ...
  - W/2 RTTs, then drop, then repeat
- Average throughput: .75W(MSS/RTT)
  - One packet dropped out of (W/2)*(3W/4)
  - Packet drop rate p = (8/3) W^{-2}
- Throughput = (MSS/RTT) sqrt(3/2p)

HINT: KNOW THIS SLIDE

102

Three Congestion Control Challenges – or Why AIMD?

- Single flow adjusting to bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be a Gbps link; could be a modem
- Single flow adjusting to variations in bandwidth
  - When bandwidth decreases, must lower sending rate
  - When bandwidth increases, must increase sending rate
- Multiple flows sharing the bandwidth
  - Must avoid overloading network
  - And share bandwidth “fairly” among the flows

103

Problem #1: Single Flow, Fixed BW

- Want to get a first-order estimate of the available bandwidth
  - Assume bandwidth is fixed
  - Ignore presence of other flows
- Want to start slow, but rapidly increase rate until packet drop occurs (“slow-start”)
- Adjustment:
  - cwnd initially set to 1 (MSS)
  - cwnd++ upon receipt of ACK

104

Problems with Slow-Start

- Slow-start can result in many losses
  - Roughly the size of cwnd ~ BW*RTT
- Example:
  - At some point, cwnd is enough to fill “pipe”
  - After another RTT, cwnd is double its previous value
  - All the excess packets are dropped!
- Need a more gentle adjustment algorithm once have rough estimate of bandwidth
  - Rest of design discussion focuses on this

105

Problem #2: Single Flow, Varying BW

Want to track available bandwidth

- Oscillate around its current value
- If you never send more than your current rate, you won't know if more bandwidth is available

Possible variations: (in terms of change per RTT)

- Multiplicative increase or decrease:
  cwnd→ cwnd * / a
- Additive increase or decrease:
  cwnd→ cwnd +- b

106

Four alternatives

- AIAD: gentle increase, gentle decrease
- AIMD: gentle increase, drastic decrease
- MIAD: drastic increase, gentle decrease
  – too many losses: eliminate
- MIMD: drastic increase and decrease

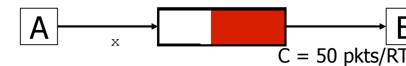
107

Problem #3: Multiple Flows

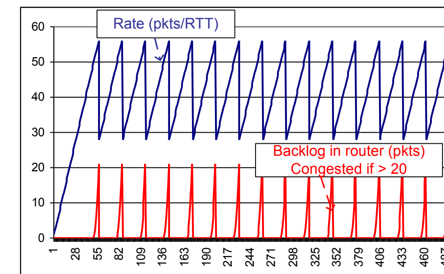
- Want steady state to be "fair"
- Many notions of fairness, but here just require two identical flows to end up with the same bandwidth
- This eliminates MIMD and AIAD
  – As we shall see…
- AIMD is the only remaining solution!
  – Not really, but close enough….

108

Recall Buffer and Window Dynamics

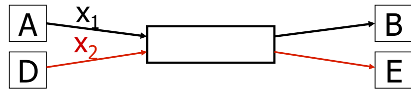


- No congestion → x increases by one packet/RTT every RTT
- Congestion → decrease x by factor 2

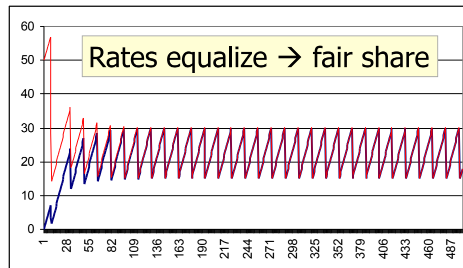


109

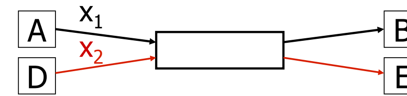
AIMD Sharing Dynamics



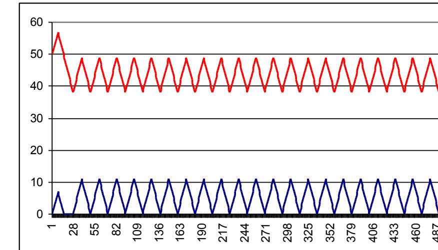
- No congestion → rate increases by one packet/RTT every RTT
- Congestion → decrease rate by factor 2



AIAD Sharing Dynamics

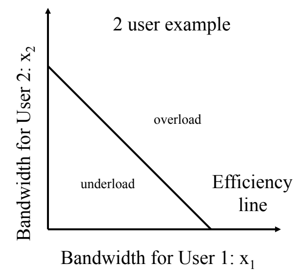


- No congestion → x increases by one packet/RTT every RTT
- Congestion → decrease x by 1



Simple Model of Congestion Control

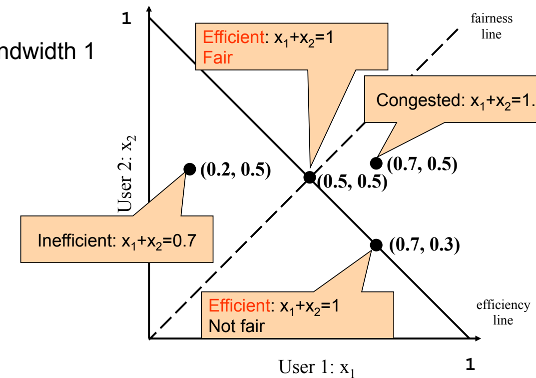
- Two TCP connections
  - Rates x_1 and x_2
- Congestion when sum>1
- Efficiency: sum near 1
- Fairness: x's converge



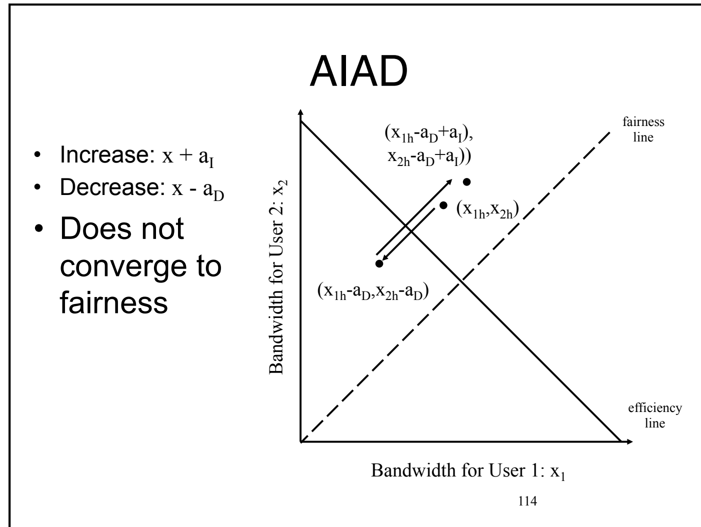
112

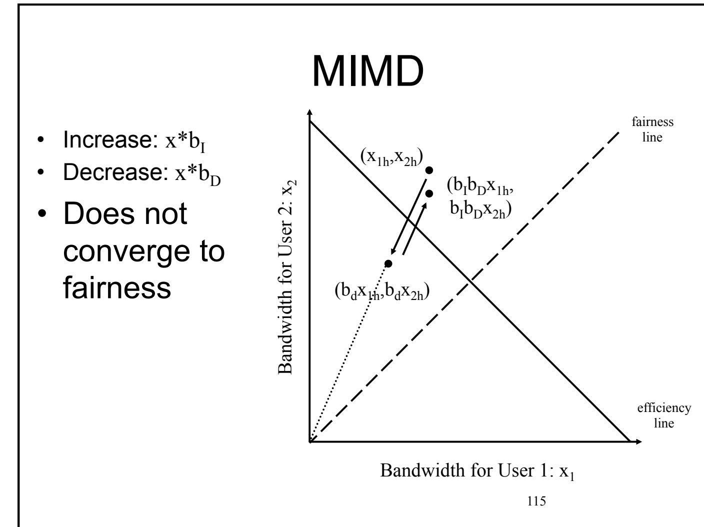
Example

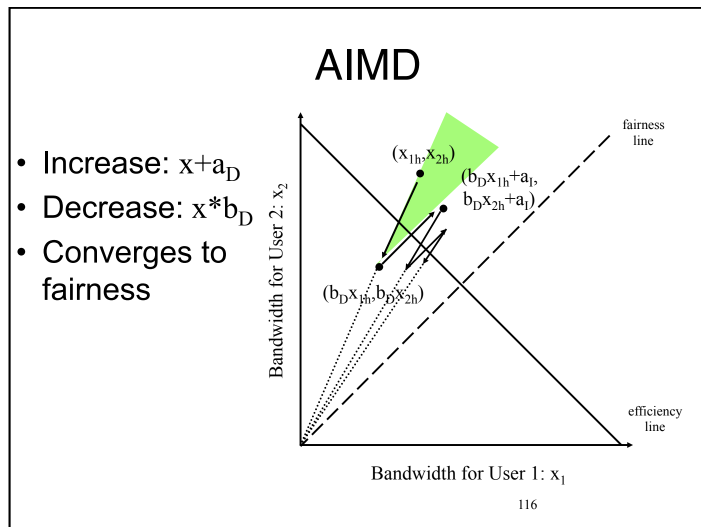
- Total bandwidth 1

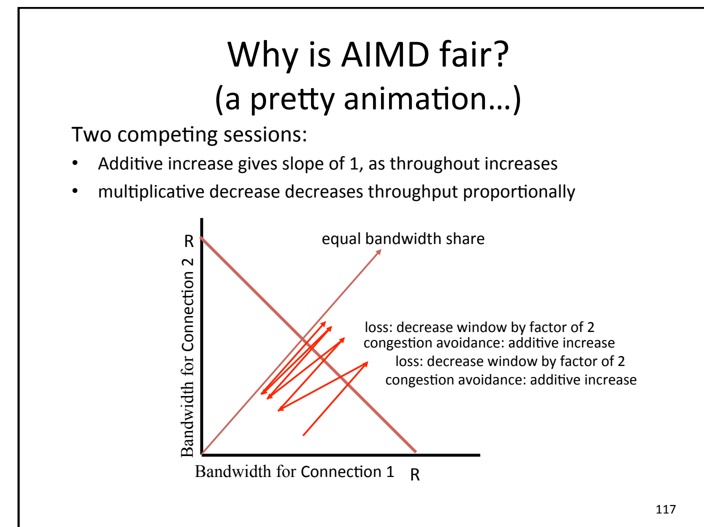


113


AIAD
Increase: x + a_I
Decrease: x - a_D
Does not converge to fairness
(x1h-aD+aI), x2h-aD+aI))
(x1h,x2h)
(x1h-aD,x2h-aD)
fairness line
efficiency line
Bandwidth for User 2: x2
Bandwidth for User 1: x1
114


MIMD
Increase: x*b_I
Decrease: x*b_D
Does not converge to fairness
(x1h,x2h)
(bIbDx1h, bIbDx2h)
(bdx1h,bdx2h)
fairness line
efficiency line
Bandwidth for User 2: x2
Bandwidth for User 1: x1
115


AIMD
Increase: x+a_D
Decrease: x*b_D
Converges to fairness
(x1h,x2h)
(bDx1h+aI, bDx2h+aI)
(bDx1h,bDx2h)
fairness line
efficiency line
Bandwidth for User 2: x2
Bandwidth for User 1: x1
116


Why is AIMD fair?
(a pretty animation…)
Two competing sessions:
Additive increase gives slope of 1, as throughout increases
multiplicative decrease decreases throughput proportionally
R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Bandwidth for Connection 2
Bandwidth for Connection 1
R
117

Fairness (more)

Fairness and UDP

- Multimedia apps may not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- (Ancient yet ongoing) 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 !

- ***Recall*** Multiple browser sessions (and the potential for syncronized loss)

118

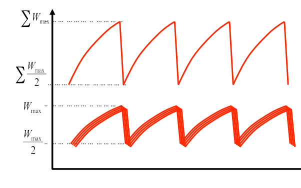
Some TCP issues outstanding…

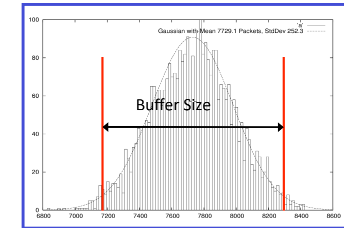
Synchronized Flows

- Aggregate window has same dynamics
- Therefore buffer occupancy has same dynamics
- Rule-of-thumb still holds.

Many TCP Flows

- Independent, desynchronized
- Central limit theorem says the aggregate becomes Gaussian
- Variance (buffer size) decreases as N increases





119