

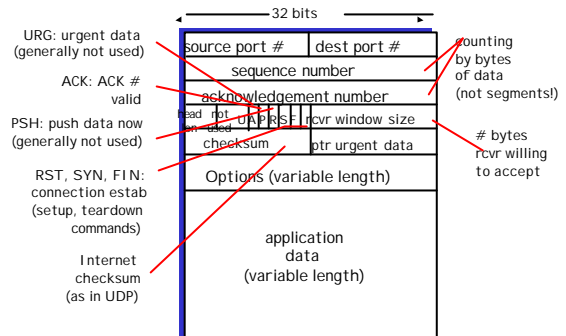
TCP: Overview

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
- **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

1

TCP segment structure



2

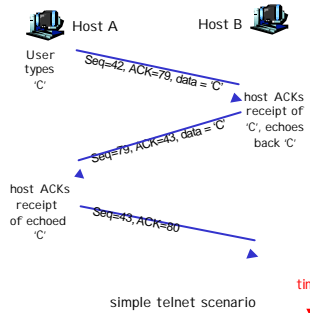
TCP seq. #'s and ACKs

Seq. #'s:

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

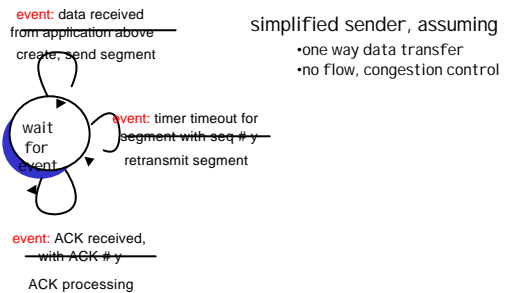
ACKs:

- seq # of next byte expected from other side
- cumulative ACK
- A: TCP spec doesn't say, - up to implementor



3

TCP: reliable data transfer



4

TCP: reliable data transfer

Simplified TCP sender

```

00 sendbase = initial_sequence number
01 nextseqnum = initial_sequence number
02
03 loop (forever) {
04   switch(event)
05   event: data received from application above
06     create TCP segment with sequence number nextseqnum
07     start timer for segment nextseqnum
08     pass segment to IP
09     nextseqnum = nextseqnum + length(data)
10   event: timer timeout for segment with sequence # y
11     retransmit segment with sequence number y
12     compute new timeout interval for segment y
13     restart timer for sequence number y
14   event: ACK received, with ACK field value of y
15     if (y > sendbase) { /* cumulative ACK of all data up to y */
16       cancel all timers for segments with sequence numbers < y
17       sendbase = y
18     }
19     else { /* a duplicate ACK for already ACKed segment */
20       increment number of duplicate ACKs received for y
21       if (number of duplicate ACKs received for y == 3) {
22         /* TCP fast retransmit */
23         resend segment with sequence number y
24         restart timer for segment y
25       }
26     } /* end of loop forever */

```

5

TCP: reliable data transfer

```

00 sendbase = initial_sequence number
01 nextseqnum = initial_sequence number
02
03 loop (forever) {
04   switch(event)
05   event: data received from application above
06     create TCP segment with sequence # nextseqnum
07     start timer for segment nextseqnum
08     pass segment to IP
09     nextseqnum = nextseqnum + length(data)
10   event: timer timeout for segment with sequence # y
11     retransmit segment with sequence number y
12     compute new timeout interval for segment y
13     restart timer for sequence number y

```

6

TCP: reliable data transfer

```

14  event: ACK received, with ACK field value of y
15  if (y > sendbase) { /* cum ACK of data up to y */
16    cancel all timers for segments with seq # < y
17    sendbase = y
18  }
19  else { /* a duplicate ACK for ACKed segment */
20    increment # of duplicate ACKs received for y
21    if (# of duplicate ACKs received for y == 3) {
22      /* TCP fast retransmit */
23      resend segment with sequence number y
24      restart timer for segment y
25    }
26  } /* end of loop forever */

```

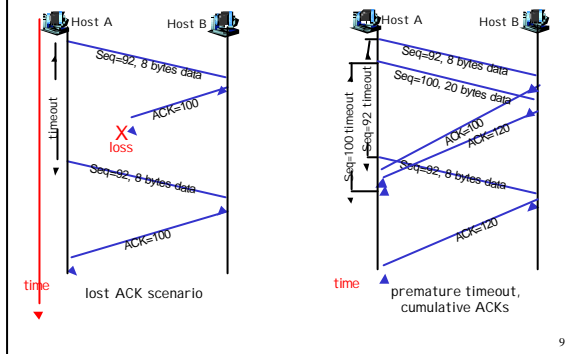
7

TCP ACK generation [RFC 1122, RFC 2581]

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

8

TCP: retransmission scenarios

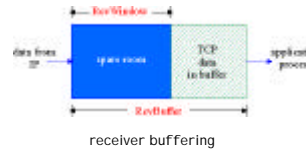


9

TCP Flow Control

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

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space
 ○ rcvr window size field in TCP segment



sender: amount of transmitted, unACKed data less than most recently-receiver rcvr window size

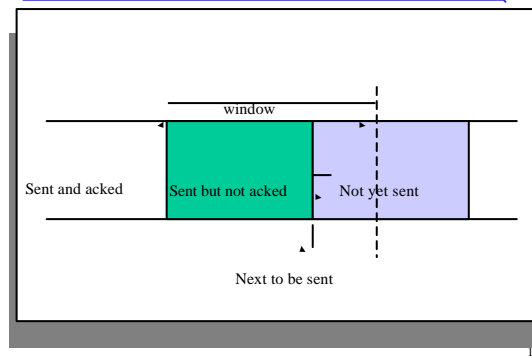
10

TCP Flow Control

- TCP is a sliding window protocol
 - For window size n , can send up to n bytes without receiving an acknowledgement
 - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
 - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
 - Congestion control now limits this

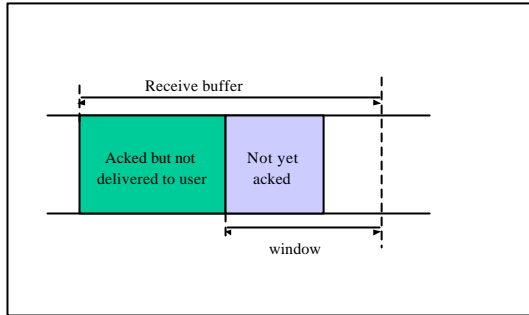
11

Window Flow Control: Send Side



12

Window Flow Control: Receive Side



13

TCP Persist

- What happens if window is 0?
 - Receiver updates window when application reads data
 - What if this update is lost?
- TCP Persist state
 - Sender periodically sends 1 byte packets
 - Receiver responds with ACK even if it can't store the packet

14

TCP Round-trip Time and Timeout Estimation

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
 - Low RTT → unneeded retransmissions
 - High RTT → poor throughput
- RTT estimator must adapt to change in RTT
 - But not too fast, or too slow!
- Spurious timeouts
 - "Conservation of packets" principle - more than a window worth of packets in flight

15

Initial Round-trip Estimator

- Round trip times exponentially averaged:
 - $\text{New RTT} = \alpha (\text{old RTT}) + (1 - \alpha) (\text{new sample})$
 - Recommended value for α : 0.8 - 0.9
 - 0.875 for most TCP's
- Retransmit timer set to β RTT, where $\beta = 2$
 - Every time timer expires, RTO exponentially backed-off
 - Like Ethernet
- Not good at preventing spurious timeouts

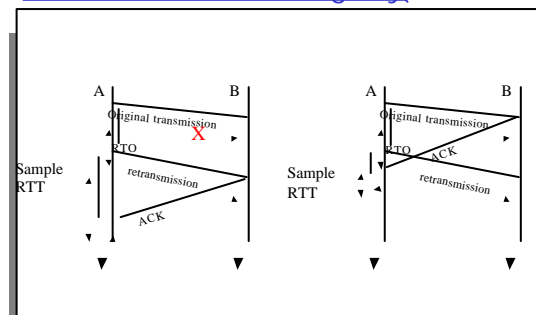
16

Jacobson's Retransmission Timeout

- Key observation:
 - At high loads round trip variance is high
- Solution:
 - Base RTO on RTT and standard deviation or RRTT
 - $\text{rttvar} = \gamma * \text{dev} + (1 - \gamma) \text{rttvar}$
 - dev = linear deviation
 - I inappropriately named - actually smoothed linear deviation

17

Retransmission Ambiguity



18

Karn's RTT Estimator

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
 - Don't count RTT sample on ACKs for this segment
 - Keep backed off time-out for next packet
 - Reuse RTT estimate only after one successful transmission

19

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
 - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
 - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
 - Can get RTT sample on any packet

20

Timer Granularity

- Many TCP implementations set RTO in multiples of 200, 500, 1000ms
- Why?
 - Avoid spurious timeouts - RTTs can vary quickly due to cross traffic
 - Make timers interrupts efficient

21

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");
```

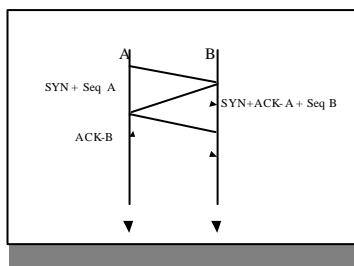
- *server*: contacted by client

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

22

Connection Establishment

- Use 3-way handshake



23

Sequence Number Selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation

24

TCP Connection Management

Three way handshake:

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

- o specifies initial seq #
- o specifies initial window #

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

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

Step 3: client system receives SYNACK

25

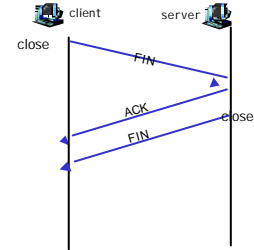
TCP Connection Management (cont.)

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.



26

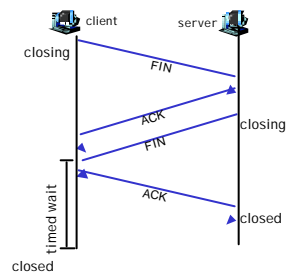
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

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

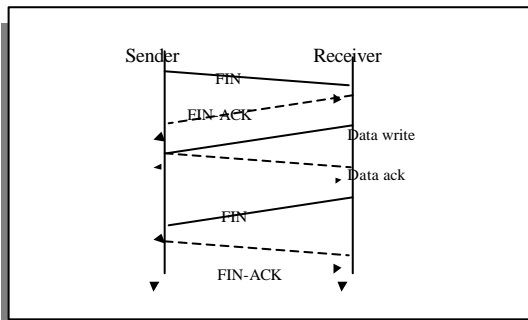
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



27

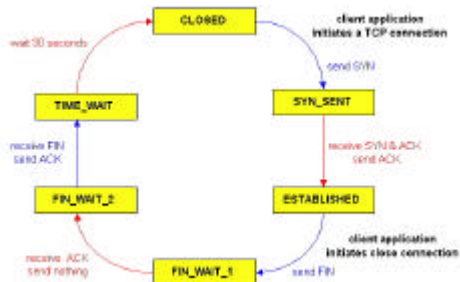
Tear-down Packet Exchange



28

TCP Connection Management (cont)

TCP client lifecycle



29

TCP Connection Management (cont)

TCP server lifecycle



30

Detecting Half-open Connections

TCP A	TCP B
1. (CRASH)	(send 300, receive 100)
2. CLOSED	ESTABLISHED
3. SYN-SENT → <SEQ=400><CTL=SYN>	→ (??)
4. (!!) ← <SEQ=300><ACK=100><CTL=ACK>	← ESTABLISHED
5. SYN-SENT → <SEQ=100><CTL=RST>	→ (Abort!!)
6. SYN-SENT	CLOSED
7. SYN-SENT → <SEQ=400><CTL=SYN>	→

31

Observed TCP Problems

- ❑ Too many small packets
 - Silly window syndrome
 - Nagel's algorithm
- ❑ Initial sequence number selection
- ❑ Amount of state maintained

32

Silly Window Syndrome

- ❑ Problem: (Clark, 1982)
 - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- ❑ Solution
 - Receiver must not advertise small window increases
 - Increase window by $\min(\text{MSS}, \text{RecvBuffer}/2)$

33

Nagel's Algorithm

- ❑ Small packet problem:
 - Don't want to send a 41 byte packet for each keystroke
 - How long to wait for more data?
- ❑ Solution:
 - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged

34

Why is Selecting I SN Important?

- ❑ Suppose machine X selects I SN based on predictable sequence
- ❑ Fred has .rhosts to allow login to X from Y
- ❑ Evil Ed attacks
 - Disables host Y - denial of service attack
 - Make a bunch of connections to host X
 - Determine I SN pattern and guess next I SN
 - Fake pkt1: [<src Y><dst X>, guessed I SN]
 - Fake pkt2: desired command

35

Time Wait Issues

- ❑ Web servers not clients close connection first
 - Established → Fin-Waits → Time-Wait → Closed
 - Why would this be a problem?
- ❑ Time-Wait state lasts for $2 * \text{MSL}$
 - MSL is should be 120 seconds (is often 60s)
 - Servers often have order of magnitude more connections in Time-Wait

36

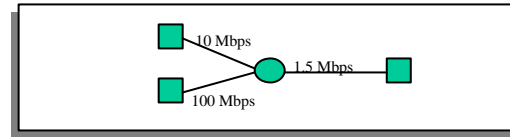
Principles of Congestion Control

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!

37

Congestion



- Different sources compete for resources inside network
- Why is it a problem?
 - Sources are unaware of current state of resource
 - Sources are unaware of each other
 - In many situations will result in < 1.5 Mbps of throughput (congestion collapse)

38

Congestion Collapse

- Definition: *Increase in network load results in decrease of useful work done*
- Many possible causes
 - Spurious retransmissions of packets still in flight
 - Classical congestion collapse
 - How can this happen with packet conservation
 - Solution: better timers and TCP congestion control
 - Undelivered packets
 - Packets consume resources and are dropped elsewhere in network
 - Solution: congestion control for ALL traffic

39

Other Congestion Collapse Causes

- Fragments
 - Mismatch of transmission and retransmission units
 - Solutions
 - Make network drop all fragments of a packet (early packet discard in ATM)
 - Do path MTU discovery
- Control traffic
 - Large percentage of traffic is for control
 - Headers, routing messages, DNS, etc.
- Stale or unwanted packets
 - Packets that are delayed on long queues
 - "Push" data that is never used

40

Where to Prevent Collapse?

- Can end hosts prevent problem?
 - Yes, but must trust end hosts to do right thing
 - E.g., sending host must adjust amount of data it puts in the network based on detected congestion
- Can routers prevent collapse?
 - No, not all forms of collapse
 - Doesn't mean they can't help
 - Sending accurate congestion signals
 - Isolating well-behaved from ill-behaved sources

41

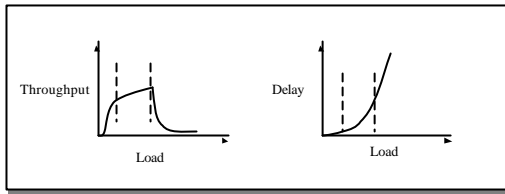
Congestion Control and Avoidance

- A mechanism which:
 - Uses network resources efficiently
 - Preserves fair network resource allocation
 - Prevents or avoids collapse
- Congestion collapse is not just a theory
 - Has been frequently observed in many networks

42

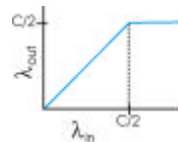
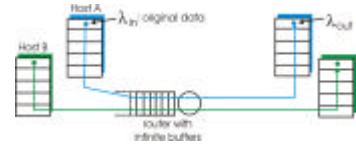
Congestion Control vs. Avoidance

- Avoidance keeps the system performing at the knee
- Control kicks in once the system has reached a congested state

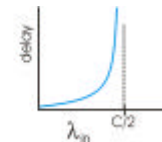


Causes/costs of congestion: scenario 1

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



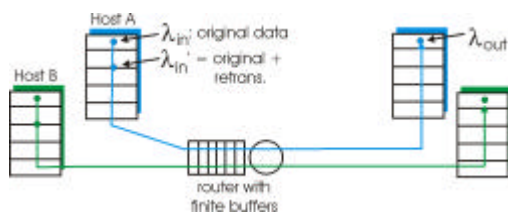
- maximum achievable throughput
- large delays when congested



44

Causes/costs of congestion: scenario 2

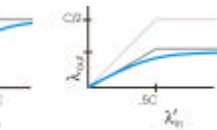
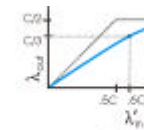
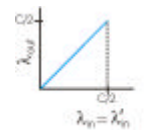
- one router, *finite* buffers
- sender retransmission of lost packet



45

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:

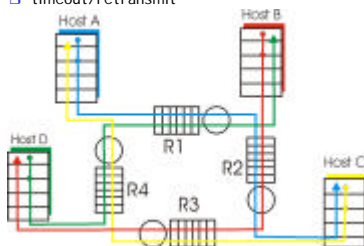
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

46

Causes/costs of congestion: scenario 3

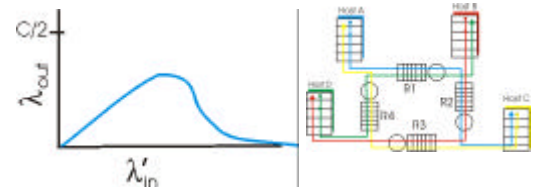
- four senders
- multihop paths
- timeout/retransmit

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



47

Causes/costs of congestion: scenario 3



Another "cost" of congestion:

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

48

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
 - choke packet router to sender
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
 - explicit rate sender should send at

49

Case study: ATM ABR congestion control

ABR: available bit rate:

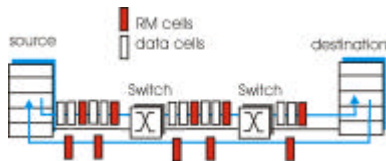
- "elastic service"
- if sender's path "underloaded":
 - sender should use available bandwidth
- if sender's path congested:
 - sender throttled to minimum guaranteed rate

RM (resource management) cells:

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

50

Case study: ATM ABR congestion control



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

51

End-to-end congestion control - objectives

- Simple router behavior
- Distributedness
- Efficiency: $X_{knee} = \sum x_i(t)$
- Fairness: $(\sum x_i)^2 / n(\sum x_i^2)$
- Power: (throughput^α/delay)
- Convergence: control system must be stable

52

Basic Control Model

- Let's assume window-based control
- Reduce window when congestion is perceived
 - How is congestion signaled?
 - Either mark or drop packets
 - When is a router congested?
 - Drop tail queues - when queue is full
 - Average queue length - at some threshold
- Increase window otherwise
 - Probe for available bandwidth - how?

53

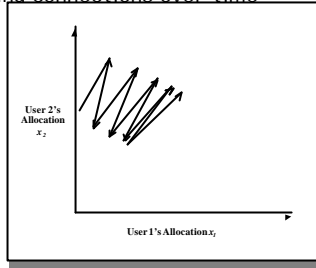
Linear Control

- Many different possibilities for reaction to congestion and probing
 - Examine simple linear controls
 - $Window(t+1) = a + b \cdot Window(t)$
 - Different a_i/b_i for increase and a_d/b_d for decrease
- Supports various reaction to signals
 - Increase/decrease additively
 - Increased/decrease multiplicatively
 - Which of the four combinations is optimal?

54

Phase plots

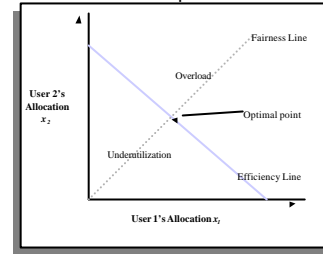
- Simple way to visualize behavior of competing connections over time



55

Phase plots

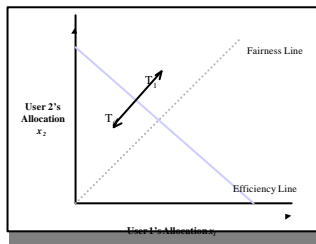
- What are desirable properties?
- What if flows are not equal?



56

Additive Increase/Decrease

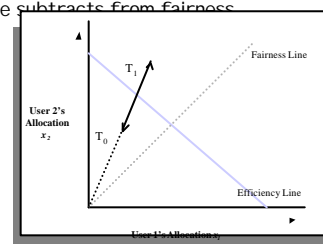
- X_1 and X_2 in-/decrease by the same amount over time
 - Additive increase/decrease - constant fairness



57

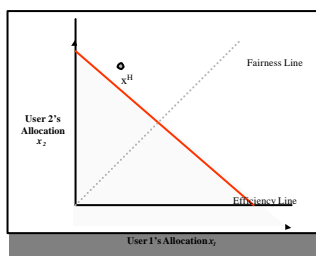
Multiplicative Increase/Decrease

- X_1 and X_2 in-/decrease by the same factor
 - Extension from origin - decrease adds to fairness, increase subtracts from fairness



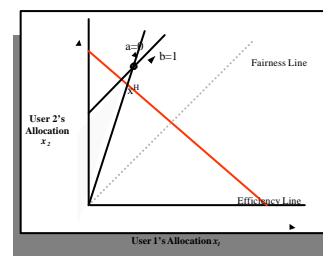
58

Convergence to Efficiency



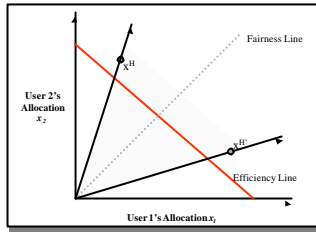
59

Distributed Convergence to Efficiency



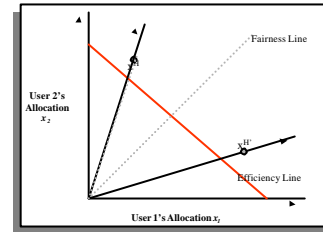
60

Convergence to Fairness



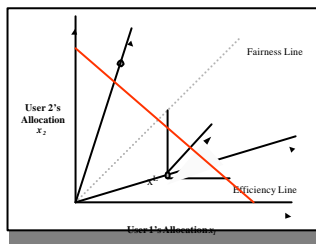
61

Convergence to Efficiency & Fairness



62

Increase



63

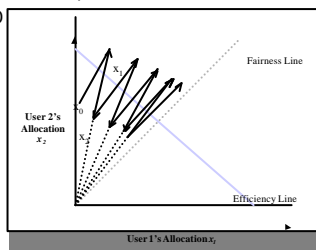
Constraints

- Distributed efficiency
 - I.e., $\sum \text{Window}(t+1) > \sum \text{Window}(t)$ during increase
 - $a_i > 0$ & $b_i \leq 1$
 - Similarly, $a_d < 0$ & $b_d \leq 1$
- Must never decrease fairness
 - $a_i/b_i > 0$ and $a_d/b_d \leq 0$
- Full constraints
 - $a_d = 0$, $0 \leq b_d < 1$, $a_i > 0$ and $b_i \leq 1$

64

What is the Right Choice?

- Constraints limit us to AIMD
 - Can have multiplicative term in increase
 - AIMD



65

TCP Congestion Control

- End-to-end control (no network assistance)
- Motivated by ARPANET congestion collapse
- Underlying design principle: packet conservation
 - At equilibrium, inject packet into network only when one is removed
 - Basis for stability of physical systems
- Why was this not working?
 - Connection doesn't reach equilibrium
 - Spurious retransmissions
 - Resource limitations prevent equilibrium

66

TCP Congestion Control - Solutions

- Reaching equilibrium
 - Slow start
- Eliminates spurious retransmissions
 - Accurate RTO estimation
 - Fast retransmit
- Adapting to resource availability
 - Congestion avoidance

67

TCP Congestion Control Basics

- Keep a congestion window, cwnd
 - Denotes how much network is able to absorb
- Sender's maximum window:
 - Min (advertised window, cwnd)
- Sender's actual window:
 - Max window - unacknowledged segments



68

TCP Congestion Control Basics (cont.)

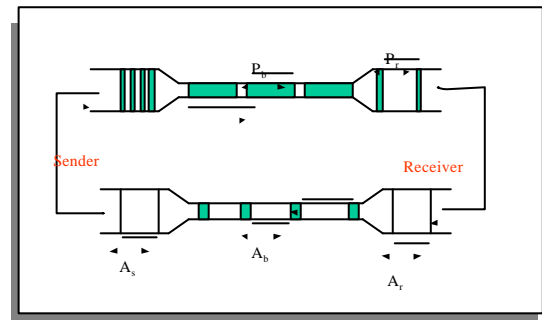
- TCP throughput limited by window
- w segments, each with MSS bytes sent in one RTT

$$\text{throughput} = \frac{w \cdot \text{MSS}}{\text{RTT}} \text{ Bytes/sec}$$

- If we have large actual window, should we send data in one shot?
 - No, use acks to clock sending new data

69

Self-clocking



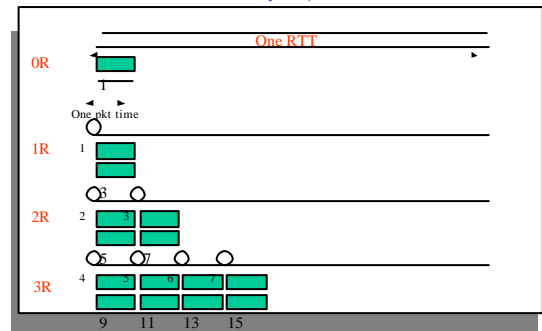
70

Slow Start

- How do we get this clocking behavior to start?
 - Initialize cwnd = 1
 - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
 - Window actually increases to W in $\text{RTT} \cdot \log_2(W)$
 - Can overshoot desired window and cause packet loss

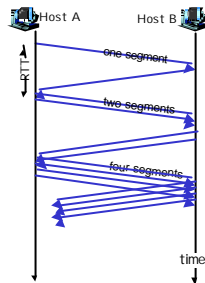
71

Slow Start Example



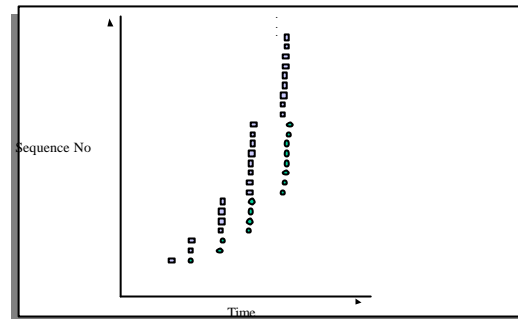
72

Slow Start Example (cont.)



73

Slow Start Sequence Number Plot



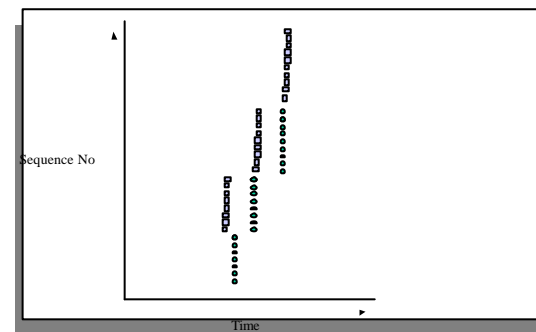
74

Congestion Avoidance

- Loss implies congestion - why?
 - Not necessarily true on all link types
- If loss occurs when $cwnd = W$
 - Network can handle $0.5W \sim W$ segments
 - Set $cwnd$ to $0.5W$ (multiplicative decrease)
- Upon receiving ACK
 - Increase $cwnd$ by $1/cwnd$
 - Results in additive increase

75

Congestion Avoidance Sequence Plot



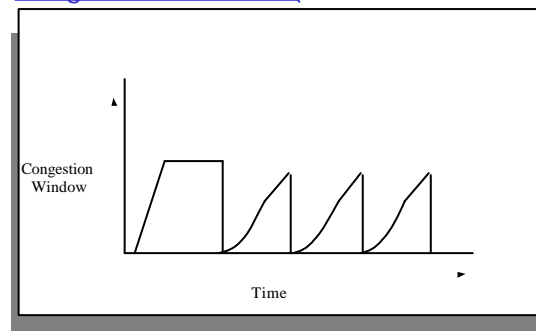
76

Return to Slow Start

- If packet is lost we lose our self clocking as well
 - Need to implement slow-start and congestion avoidance together

77

Congestion Window



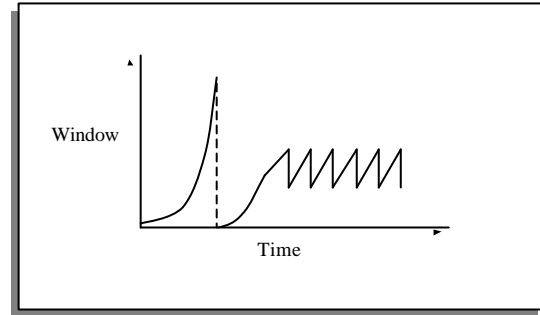
78

Return to Slow Start vs. Congestion avoidance

- If packet is lost we lose our self clocking as well
 - Need to implement slow-start and congestion avoidance together
- When timeout occurs set ssthresh to 0.5w
 - If cwnd < ssthresh, use slow start
 - Else use congestion avoidance

79

Overall TCP Behavior



80

How to Change Window

- When a loss occurs have W packets outstanding
- New cwnd = 0.5 * cwnd
 - How to get to new state?

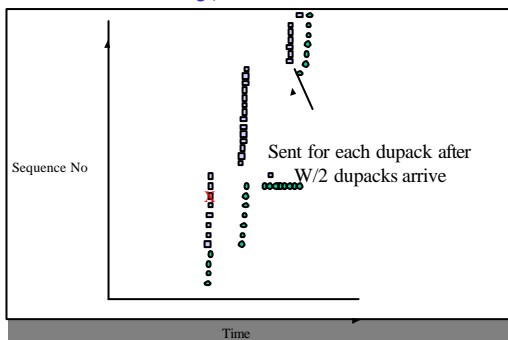
81

Fast Recovery

- Each duplicate ack notifies sender that single packet has cleared network
- When < cwnd packets are outstanding
 - Allow new packets out with each new duplicate acknowledgement
- Behavior
 - Sender is idle for some time – waiting for 1/2 cwnd worth of dupacks
 - Transmits at original rate after wait
 - Ack clocking rate is same as before loss

82

Fast Recovery



83

TCP congestion control summary:

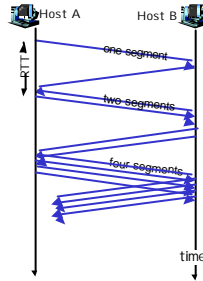
- "probing" for usable bandwidth:
 - ideally: transmit as fast as possible (Congwin as large as possible) without loss
 - increase Congwin until loss (congestion)
 - loss: decrease Congwin, then begin probing (increasing) again
- two "phases"
 - slow start
 - congestion avoidance
- important variables:
 - Congwin
 - threshold: defines threshold between two slow start phase, congestion control phase

84

TCP Slowstart

Slowstart algorithm
 initialize: Congwin = 1
 for (each segment ACKed)
 Congwin++
 until (loss event OR
 CongWin > threshold)

- exponential increase (per RTT in window size (not so slow!))
- loss event: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)



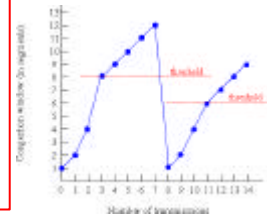
85

TCP Congestion Avoidance

Congestion avoidance

```

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
    Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
    
```



¹: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

86

TCP Flavors

- Tahoe, Reno, Vegas
- TCP Tahoe (distributed with 4.3BSD Unix)
 - Original implementation of Van Jacobson's mechanisms (VJ paper)
 - Includes:
 - Slow start
 - Congestion avoidance
 - Fast retransmit

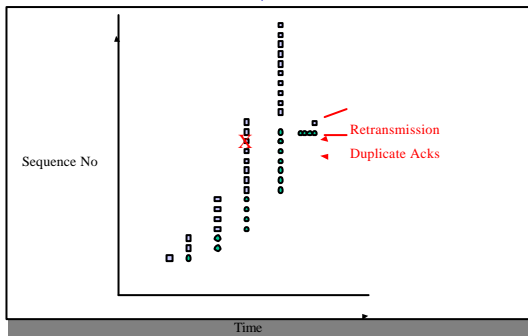
87

Fast Retransmit

- What are duplicate acks (dupacks)?
 - Repeated acks for the same sequence
- When can duplicate acks occur?
 - Loss
 - Packet re-ordering
 - Window update - advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
 - Use receipt of 3 or more duplicate acks as indication of loss
 - Don't wait for timeout to retransmit packet

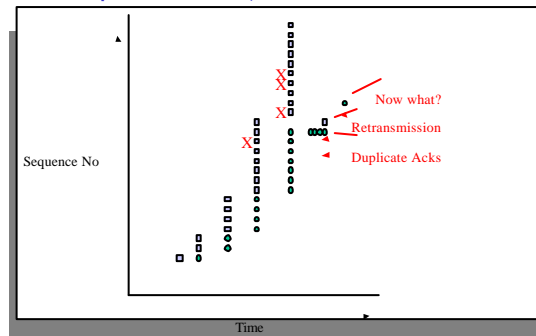
88

Fast Retransmit



89

Multiple Losses



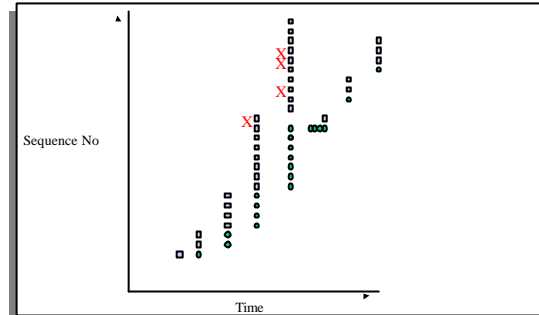
90

TCP Tahoe

- ❑ distributed with 4.3BSD Unix
- ❑ Original implementation of Van Jacobson's mechanisms (VJ paper)
- ❑ Includes:
 - Slow start
 - Congestion avoidance
 - Fast retransmit

91

Tahoe



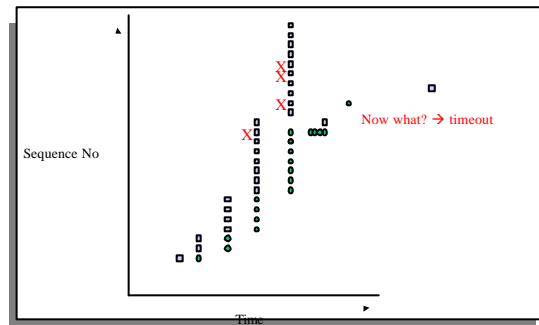
92

TCP Reno (1990)

- ❑ All mechanisms in Tahoe
- ❑ Addition of fast-recovery
 - Opening up congestion window after fast retransmit
- ❑ Delayed acks
- ❑ Header prediction
 - Implementation designed to improve performance
 - Has common case code inlined
- ❑ With multiple losses, Reno typically timeouts because it does not see duplicate acknowledgements

93

Reno



94

NewReno

- ❑ The ack that arrives after retransmission (partial ack) should indicate that a second loss occurred
- ❑ When does NewReno timeout?
 - When there are fewer than three dupacks for first loss
 - When partial ack is lost
- ❑ How fast does it recover losses?
 - One per RTT

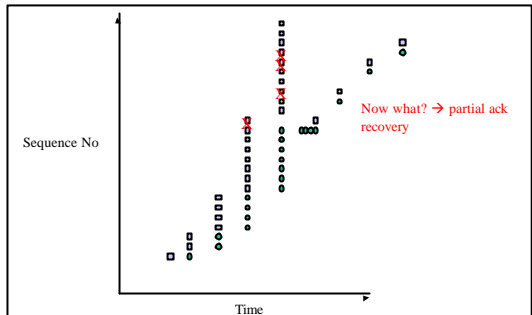
95

NewReno Changes

- ❑ Send a new packet out for each pair of dupacks
 - Adapt more gradually to new window
- ❑ Will not halve congestion window again until recovery is completed
 - Identifies congestion events vs. congestion signals
- ❑ Initial estimation for ssthresh

96

NewReno



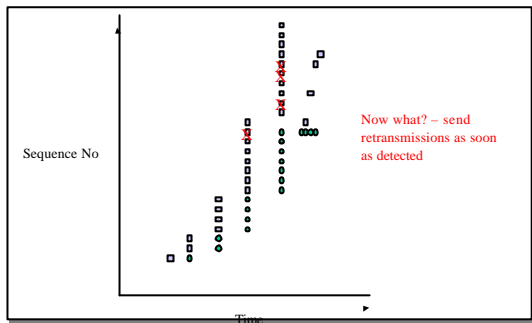
97

SACK

- Basic problem is that cumulative acks only provide little information
 - Ack for just the packet received
 - What if acks are lost? → carry cumulative also
 - Not used
 - Bitmask of packets received
 - Selective acknowledgement (SACK)
- How to deal with reordering

98

SACK



99

Performance Issues

- Timeout \gg fast retransmit
 - Need 3 dupacks/sacks
 - Not great for small transfers
 - Don't have 3 packets outstanding
 - What are real loss patterns like?
- Right edge recovery
 - Allow packets to be sent on arrival of first and second duplicate ack
 - Helps recovery for small windows
- How to deal with reordering?

100

TCP Extensions

- Implemented using TCP options
 - Timestamp
 - Protection from sequence number wraparound
 - Large windows

101

Protection From Wraparound

- Wraparound time vs. Link speed
 - 1.5Mbps: 6.4 hours
 - 10Mbps: 57 minutes
 - 45Mbps: 13 minutes
 - 100Mbps: 6 minutes
 - 622Mbps: 55 seconds → < MSL!
 - 1.2Gbps: 28 seconds
- Use timestamp to distinguish sequence number wraparound

102

Large Windows

- Delay-bandwidth product for 100ms delay
 - 1.5Mbps: 18KB
 - 10Mbps: 122KB > max 16bit window
 - 45Mbps: 549KB
 - 100Mbps: 1.2MB
 - 622Mbps: 7.4MB
 - 1.2Gbps: 14.8MB
- Scaling factor on advertised window
 - Specifies how many bits window must be shifted to the left
 - Scaling factor exchanged during connection setup

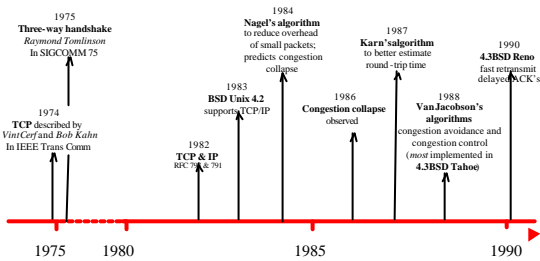
103

Maximum Segment Size (MSS)

- Exchanged at connection setup
 - Typically pick MTU of local link
- What all does this effect?
 - Efficiency
 - Congestion control
 - Retransmission
- Path MTU discovery
 - Why should MTU match MSS?

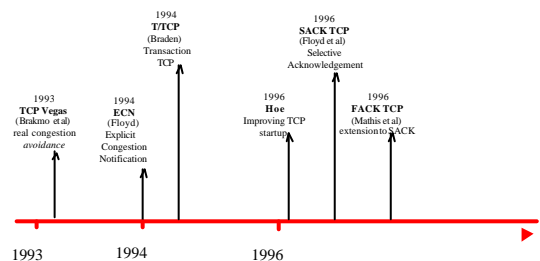
104

Evolution of TCP



105

TCP Through the 1990s



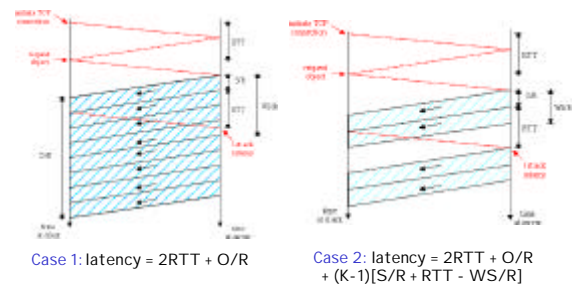
106

Effects of TCP latencies

- Q: client latency from object request from WWW server to receipt?
- TCP connection establishment
 - data transfer delay
- Notation, assumptions:**
- Assume: fixed congestion window, W , giving throughput of R bps
 - S : MSS (bits)
 - O : object size (bits)
 - no retransmissions (no loss, no corruption)
- Two cases to consider:**
- $WS/R > RTT + S/R$: ACK for first segment in window before window's worth of data sent
 - $WS/R < RTT + S/R$: wait for ACK after sending window's worth of data sent

107

Effects of TCP latencies



108

Delayed Ack Impact

- TCP congestion control triggered by acks
 - If receive half as many acks → window grows half as fast
- Slow start with window = 1
 - Will trigger delayed ack timer
 - First exchange will take at least 200ms
 - Start with > 1 initial window
 - Bug in BSD, now a "feature"/standard

109

Transport Layer: Summary

- principles behind transport layer services:
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
 - instantiation and implementation in the Internet
 - UDP
 - TCP
- Next:**
- leaving the network "edge" (application transport layer)
 - into the network "core"

110