Computer Networking

Lent Term M/W/F 11-midday
LT1 in Gates Building

Slide Set 5

Andrew W. Moore
andrew.moore@cl.cam.ac.uk
February 2014

1

Topic 5b – 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

Automatic Repeat Request (ARQ)

+ Self-clocking (Automatic)

+ Adaptive

+ Flexible

- Slow to start / adapt
consider high Bandwidth/Delay product

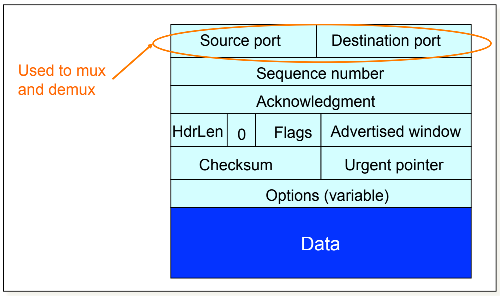
Next lets move from the generic to the specific....

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

its an ARQ protocol

3

TCP Header



4

Last time: Components of a solution for reliable transport

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Replay (SR)

5

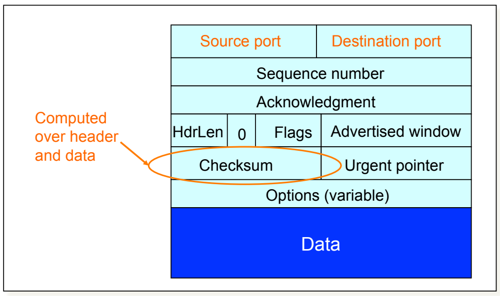
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

6

TCP Header



7

What does TCP do?

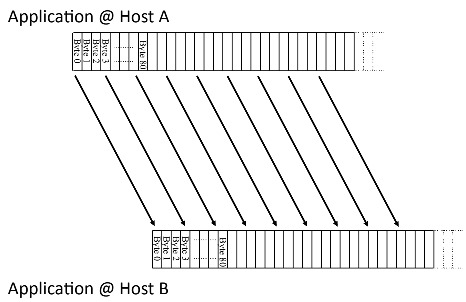
Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**

TCP: Segments and Sequence Numbers

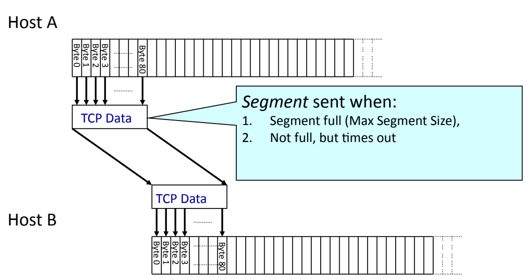
9

TCP "Stream of Bytes" Service…



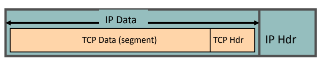
10

… Provided Using TCP "Segments"



11

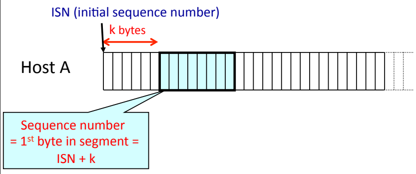
TCP Segment



- IP packet
  – No bigger than Maximum Transmission Unit (MTU)
  – E.g., up to 1500 bytes with Ethernet
- TCP packet
  – IP packet with a TCP header and data inside
  – TCP header ≥ 20 bytes long
- **TCP segment**
  – No more than Maximum Segment Size (MSS) bytes
  – E.g., up to 1460 consecutive bytes from the stream
  – MSS = MTU – (IP header) – (TCP header)

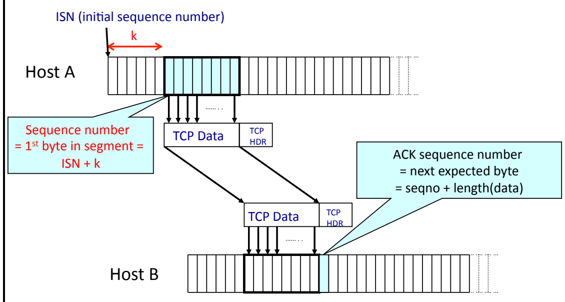
12

Sequence Numbers



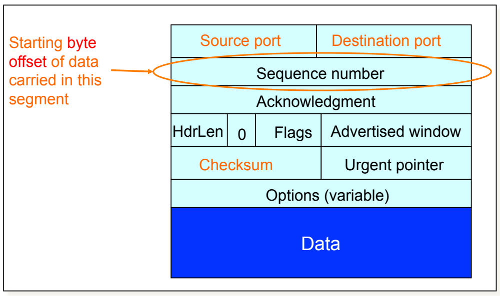
13

Sequence Numbers



14

TCP Header



15

- What does TCP do?

16

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)

ACKing and Sequence Numbers

- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, ...X+B-1]
- Upon receipt of packet, receiver sends an ACK
  - If all data prior to X already received:
    - ACK acknowledges X+B (because that is next expected byte)
  - If highest in-order byte received is Y s.t. (Y+1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

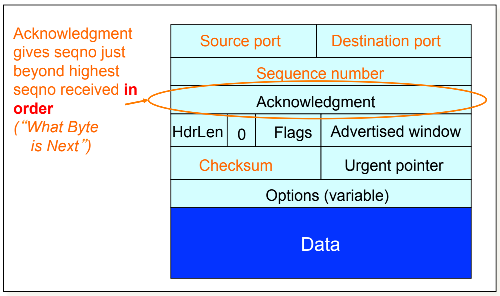
18

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field

TCP Header



What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers can buffer out-of-sequence packets (like SR)

Loss with cumulative ACKs

- Sender sends packets with 100B and seqnos.:
  – 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- Assume the fifth packet (seqno 500) is lost, but no others
- Stream of ACKs will be:
  – 200, 300, 400, 500, 500, 500, 500,…

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers may not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

Loss with cumulative ACKs

- "Duplicate ACKs" are a sign of an isolated loss
  – The lack of ACK progress means 500 hasn't been delivered
  – Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
  - TCP uses k=3
- But response to loss is trickier….

Loss with cumulative ACKs

- Two choices:
  - Send missing packet and increase W by the number of dup ACKs
  - Send missing packet, and wait for ACK to increase W
- Which should TCP do?

25

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

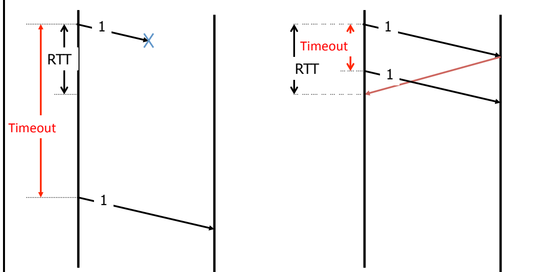
26

Retransmission Timeout

- If the sender hasn't received an ACK by timeout, retransmit the first packet in the window
- How do we pick a timeout value?

27

Timing Illustration



Timeout too long → inefficient

Timeout too short → duplicate packets

28

Retransmission Timeout

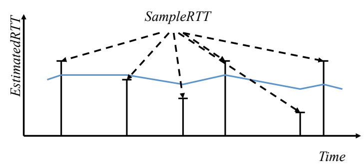
- If haven't received ack by timeout, retransmit the first packet in the window
- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
- But how do we measure RTT?

29

RTT Estimation

- Use exponential averaging of RTT samples

$$SampleRTT = AckRcvdTime - SendPacketTime$$
$$EstimatedRTT = \alpha \times EstimatedRTT + (1-\alpha) \times SampleRTT$$
$$0 < \alpha \leq 1$$

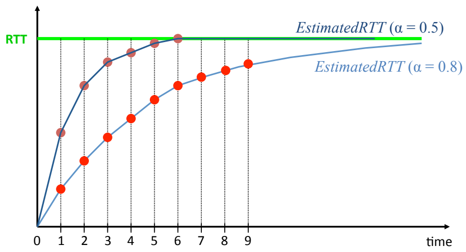


30

Exponential Averaging Example

*EstimatedRTT = α*EstimatedRTT + (1 – α)*SampleRTT*

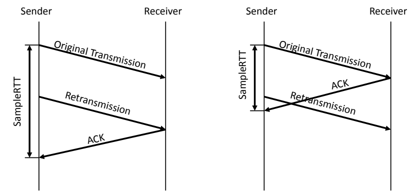
Assume RTT is constant → *SampleRTT* = RTT



31

Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?



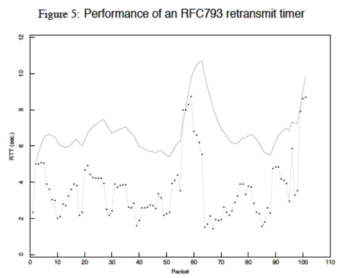
32

Karn/Partridge Algorithm

- Measure *SampleRTT* only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Computes EstimatedRTT using *α = 0.875*
- Timeout value (RTO) = 2 × EstimatedRTT
- Employs exponential backoff
  - Every time RTO timer expires, set RTO ← 2·RTO
  - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

33

Karn/Partridge in action



from Jacobson and Karels, SIGCOMM 1988

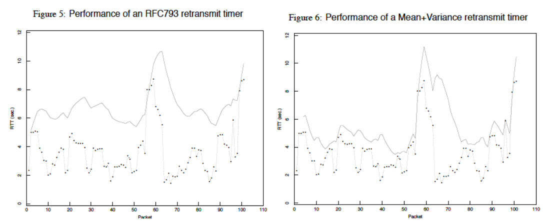
34

Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure deviation
- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation
- RTO = EstimatedRTT + 4 x EstimatedDeviation

35

With Jacobson/Karels



36

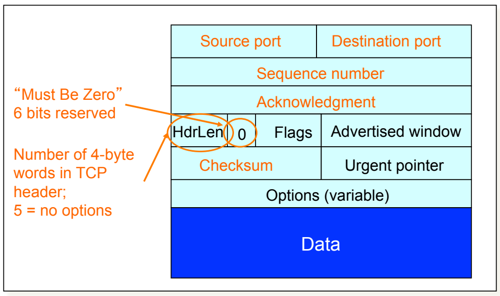
What does TCP do?

Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

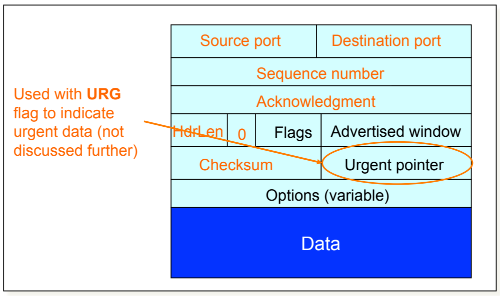
37

TCP Header: What's left?



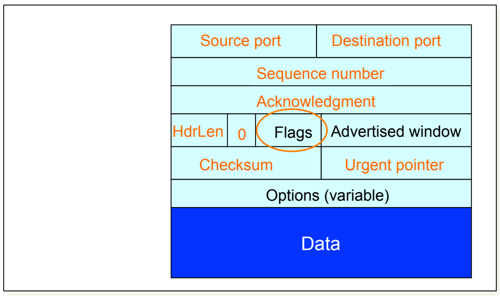
38

TCP Header: What's left?



39

TCP Header: What's left?



40

TCP Connection Establishment and Initial Sequence Numbers

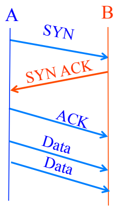
41

Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection

42

Establishing a TCP Connection

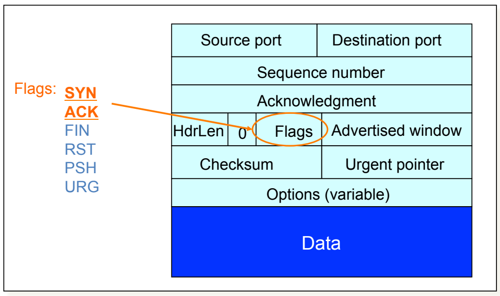


Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

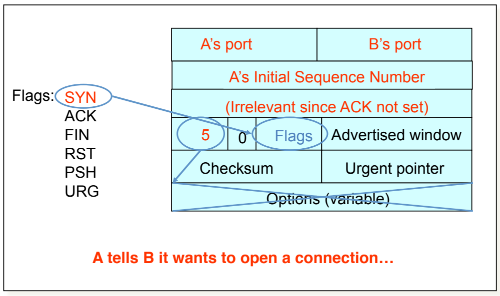
43

TCP Header



44

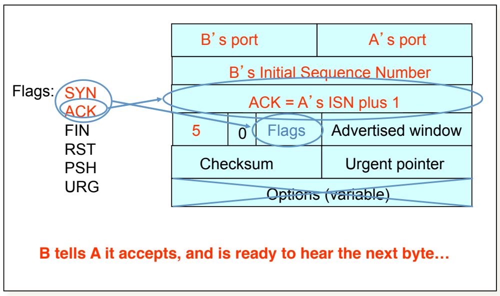
Step 1: A's Initial SYN Packet



A tells B it wants to open a connection…

45

Step 2: B's SYN-ACK Packet

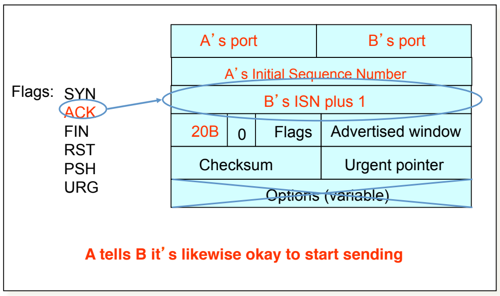


B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

46

Step 3: A's ACK of the SYN-ACK

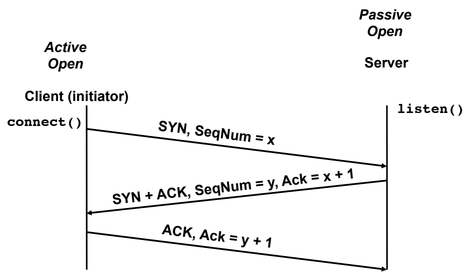


A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

47

Timing Diagram: 3-Way Handshaking



48

What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it's too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 6 seconds

49

SYN Loss and Web Downloads

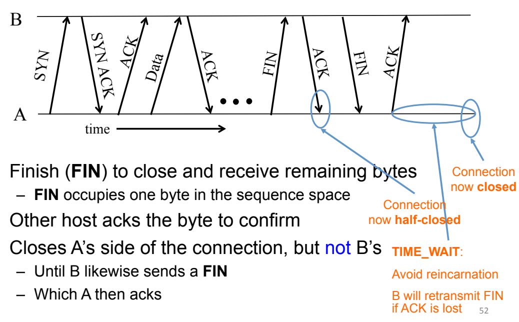
- User clicks on a hypertext link
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- If the SYN is lost…
  - 3-6 seconds of delay: can be very long
  - User may become impatient
  - … and click the hyperlink again, or click "reload"
- User triggers an "abort" of the "connect"
  - Browser creates a new socket and another "connect"
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly

50

Tearing Down the Connection

51

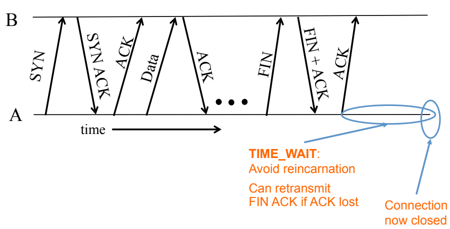
Normal Termination, One Side At A Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

52

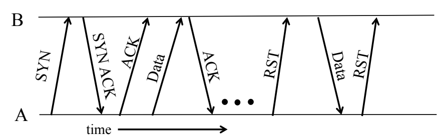
Normal Termination, Both Together



- Same as before, but B sets **FIN** with their ack of A's **FIN**

53

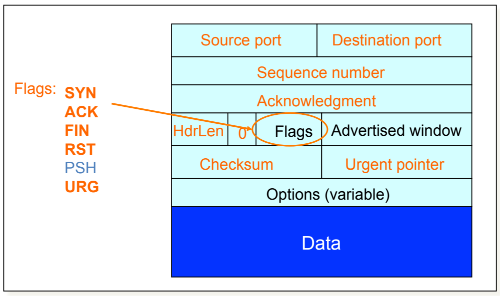
Abrupt Termination



- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

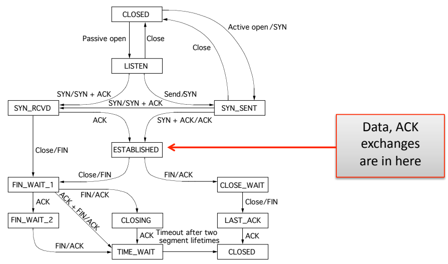
54

TCP Header



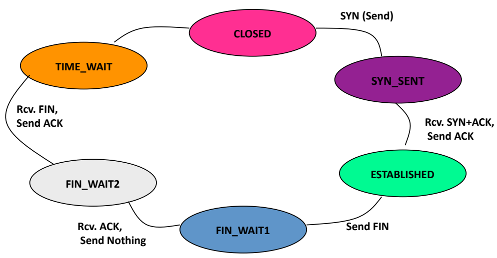
55

TCP State Transitions



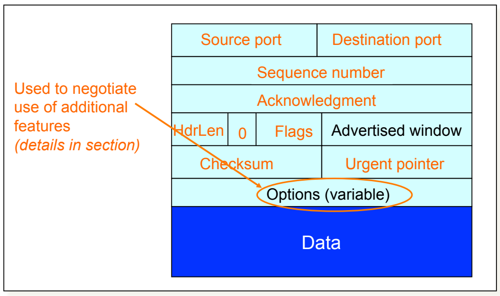
56

An Simpler View of the Client Side



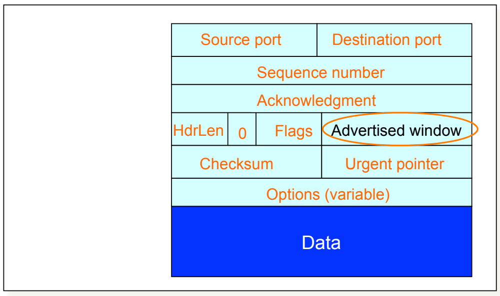
57

TCP Header



58

TCP Header



59

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP

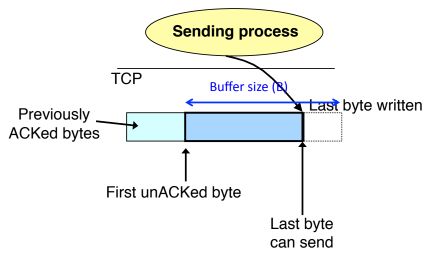
60

Recap: Sliding Window (so far)

- Both sender & receiver maintain a **window**
- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of undelivered data
- Right edge: Left edge + *constant*
  - constant only limited by buffer size in the transport layer

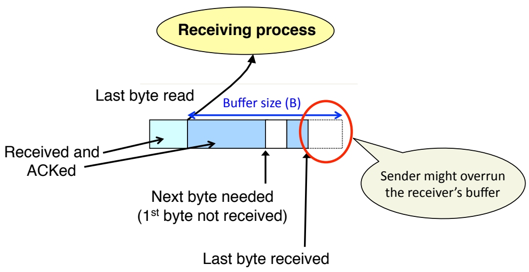
61

Sliding Window at Sender (so far)



62

Sliding Window at Receiver (so far)



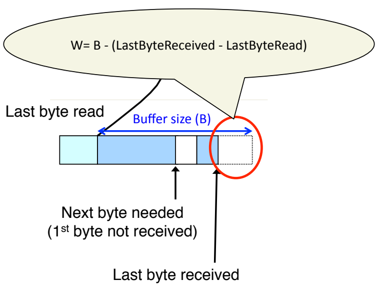
63

Solution: Advertised Window (Flow Control)

- Receiver uses an "Advertised Window" (W) to prevent sender from overflowing its window
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W

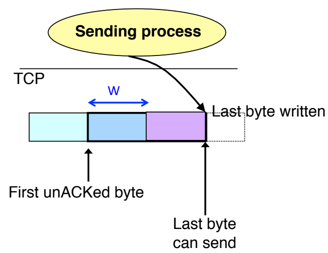
64

Sliding Window at Receiver



65

Sliding Window at Sender (so far)



66

Sliding Window w/ Flow Control

- Sender: window advances when new data ack'd
- Receiver: window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
  - Sender agrees not to exceed this amount

67

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- What's missing?

68

TCP

- The concepts underlying TCP are simple
  - acknowledgments (feedback)
  - timers
  - sliding windows
  - buffer management
  - sequence numbers

69

TCP

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers?
  - What is the seqno for an ACK-only packet?
  - What happens if advertised window = 0?
  - What if the advertised window is ½ an MSS?
  - Should receiver acknowledge packets right away?
  - What if the application generates data in units of 0.1 MSS?
  - What happens if I get a duplicate SYN? Or a RST while I'm in FIN_WAIT, *etc., etc., etc.*

70

TCP

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?

71

Sizing Windows for Congestion Control

- What are the problems?
- How might we address them?

72

- What does TCP do?
  – ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP

73

We have seen:

– Flow control: adjusting the sending rate to keep from overwhelming a slow *receiver*

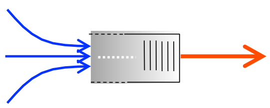
Now lets attend…

– Congestion control: adjusting the sending rate to keep from overloading the *network*

74

Statistical Multiplexing → Congestion

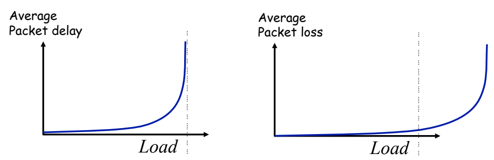
- If two packets arrive at the same time
  – A router can only transmit one
  – … and either buffers or drops the other
- If many packets arrive in a short period of time
  – The router cannot keep up with the arriving traffic
  – … delays traffic, and the buffer may eventually overflow
- Internet traffic is bursty



75

Congestion is undesirable

Typical queuing system with bursty arrivals



Must balance utilization versus delay and loss

76

Who Takes Care of Congestion?

- Network? End hosts? Both?

- TCP's approach:
  – **End hosts** adjust sending rate
  – Based on **implicit feedback** from network

- Not the only approach
  – A consequence of history rather than planning

77

Some History: TCP in the 1980s

- Sending rate only limited by flow control
  – Packet drops → senders (repeatedly!) retransmit a full window's worth of packets
- Led to "congestion collapse" starting Oct. 1986
  – Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec
- "Fixed" by Van Jacobson's development of TCP's congestion control (CC) algorithms

78

Jacobson's Approach

- Extend TCP's existing window-based protocol but adapt the window size in response to congestion
  - required no upgrades to routers or applications!
  - patch of a few lines of code to TCP implementations
- A pragmatic and effective solution
  - but many other approaches exist
- Extensively improved on since
  - topic now sees less activity in ISP contexts
  - but is making a comeback in datacenter environments

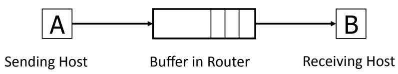
79

Three Issues to Consider

- Discovering the available (bottleneck) bandwidth
- Adjusting to variations in bandwidth
- Sharing bandwidth between flows

80

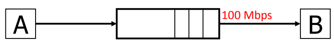
Abstract View



- Ignore internal structure of router and model it as having a single queue for a particular input-output pair

81

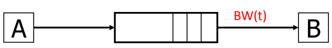
Discovering available bandwidth



- Pick sending rate to match bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be gigabit link, could be a modem

82

Adjusting to variations in bandwidth



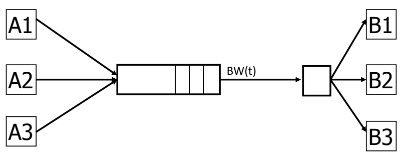
- Adjust rate to match instantaneous bandwidth
  - Assuming you have rough idea of bandwidth

83

Multiple flows and sharing bandwidth

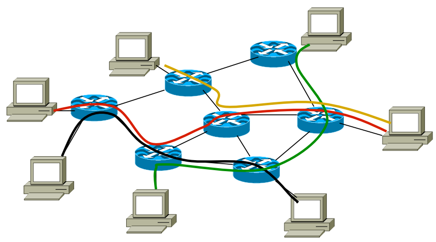
Two Issues:

- Adjust total sending rate to match bandwidth
- Allocation of bandwidth between flows



84

Reality

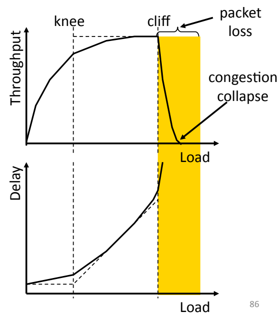


Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

85

View from a single flow

- Knee – point after which
  - Throughput increases slowly
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to drop to zero (congestion collapse)
  - Delay approaches infinity



86

General Approaches

(0) Send without care
- Many packet drops

87

General Approaches

(0) Send without care

(1) Reservations
- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

88

General Approaches

(0) Send without care

(1) Reservations

(2) Pricing
- Don't drop packets for the high-bidders
- Requires payment model

89

General Approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment
- Hosts probe network; infer level of congestion; adjust
- Network reports congestion level to hosts; hosts adjust
- Combinations of the above
- Simple to implement but suboptimal, messy dynamics

90

General Approaches

(0) Send without care
(1) Reservations
(2) Pricing
(3) Dynamic Adjustment

All three techniques have their place
- *Generality* of dynamic adjustment has proven powerful
- Doesn't presume business model, traffic characteristics, application requirements; does assume good citizenship

91

TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight
- Sending rate: ~Window/RTT
- Vary window size to control sending rate

92

All These Windows…

- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm
- Flow control window: AdvertisedWindow (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender
- Sender-side window = minimum{CWND,RWND}
  - Assume for this lecture that RWND >> CWND

93

Note

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes
- Keep in mind that real implementations maintain CWND in bytes

94

Two Basic Questions

- How does the sender detect congestion?
- How does the sender adjust its sending rate?
  - To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

95

Detecting Congestion

- Packet delays
  - Tricky: noisy signal (delay often varies considerably)
- Router tell endhosts they're congested
- Packet loss
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (checksum errors)
- Two indicators of packet loss
  - No ACK after certain time interval: timeout
  - Multiple duplicate ACKs

96

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough dupacks
  - Must have suffered several losses
- Will adjust rate differently for each case

97

Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate
- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth *vs.*
  - Adjusting to bandwidth variations

98

Bandwidth Discovery with Slow Start

- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)
- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

99

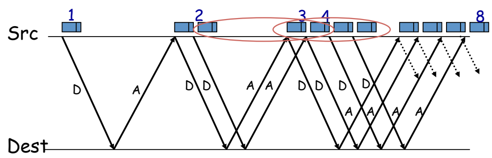
"Slow Start" Phase

- Sender starts at a slow rate but increases **exponentially** until first loss
- Start with a small congestion window
  - Initially, CWND = 1
  - So, initial sending rate is MSS/RTT
- Double the CWND for each RTT with no loss

100

Slow Start in Action

- For each RTT: double CWND
- Simpler implementation: for each ACK, CWND += 1



101

Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

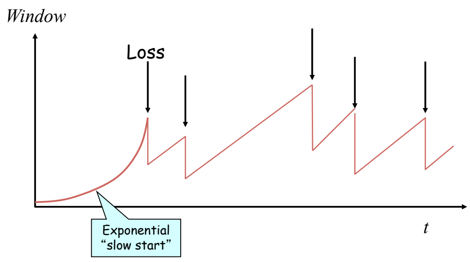
102

AIMD

- Additive increase
  - Window grows by one MSS for every RTT with no loss
  - For each successful RTT, CWND = CWND + 1
  - Simple implementation:
    - for each ACK, CWND = CWND+ 1/CWND
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**
  - On loss, CWND = CWND/2

103

Leads to the TCP "Sawtooth"



104

Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?
- Introduce a "slow start threshold" (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2
- When CWND = ssthresh, sender switches from slow-start to AIMD-style increase

105
105

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD

106

Why AIMD?

107

Recall: Three Issues

- Discovering the available (bottleneck) bandwidth
  - Slow Start
- Adjusting to variations in bandwidth
  - AIMD
- Sharing bandwidth between flows

108

Goals for bandwidth sharing

- Efficiency: High utilization of link bandwidth
- Fairness: Each flow gets equal share

109

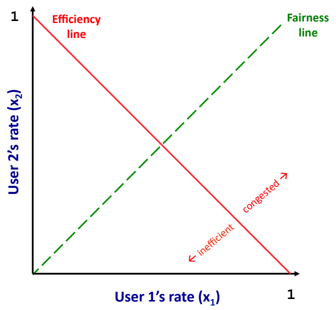
Why AIMD?

- Some rate adjustment options: Every RTT, we can
  – Multiplicative increase or decrease: CWND→ a*CWND
  – Additive increase or decrease: CWND→ CWND + b
- Four alternatives:
  – AIAD: gentle increase, gentle decrease
  – AIMD: gentle increase, drastic decrease
  – MIAD: drastic increase, gentle decrease
  – MIMD: drastic increase and decrease

110

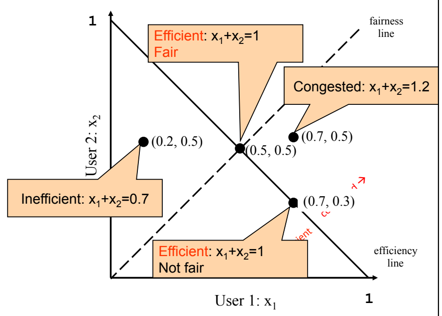
Simple Model of Congestion Control

- Two users
  – rates x_1 and x_2
- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$
- Fair when $x_1 = x_2$



111

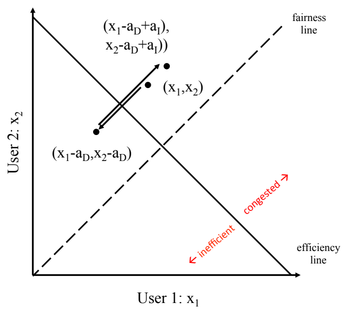
Example



112

AIAD

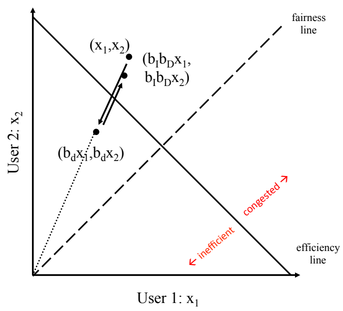
- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



113

MIMD

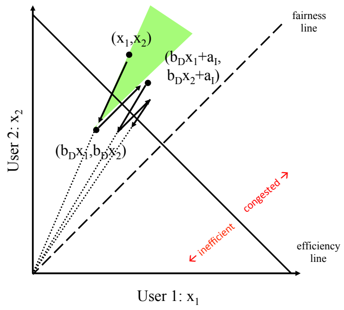
- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness



114

AIMD

- Increase: $x + a_I$
- Decrease: $x * b_D$
- Converges to fairness

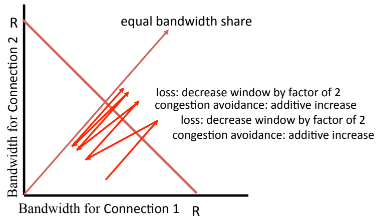


115

Why is AIMD fair?
(a pretty animation…)

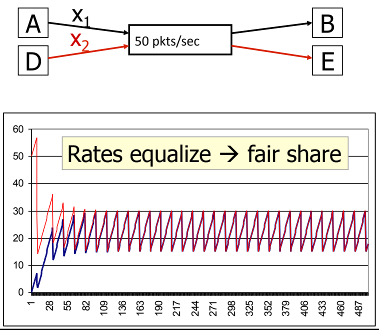
Two competing sessions:

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



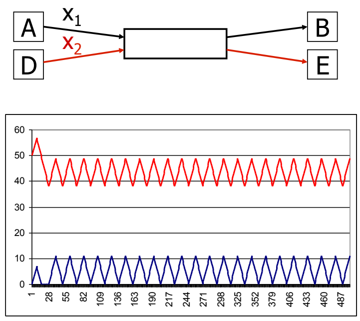
116

AIMD Sharing Dynamics



117

AIAD Sharing Dynamics



118

TCP Congestion Control Details

119

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - [Also dupACKcount and timer, as before]
- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

120

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = 2xCWND

121

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = CWND + 1

122

Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

123

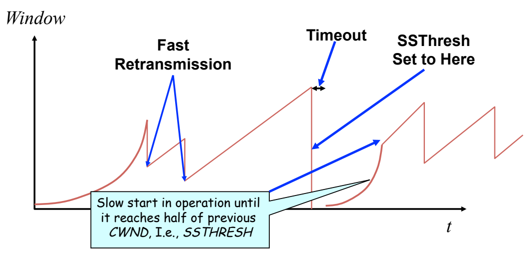
Event: dupACK

- dupACKcount ++

- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

124

Example



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

125

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery

126
126

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

127

Example (in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

128

Timeline

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

129
129

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

130

Example

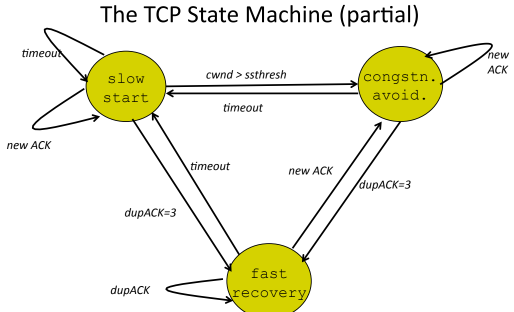
- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

131

Timeline

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

Putting it all together: The TCP State Machine (partial)



- How are ssthresh, CWND and dupACKcount updated for each event that causes a state transition?

TCP Flavors

- TCP-Tahoe
  - cwnd =1 on triple dupACK
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

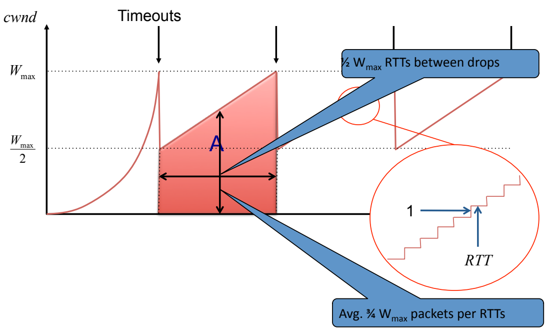
- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput

135

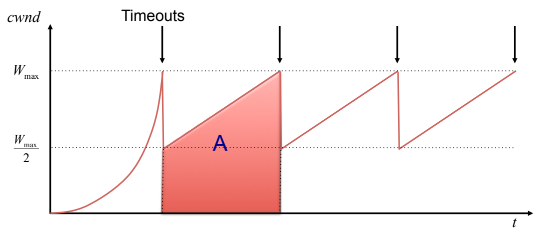
TCP Throughput Equation

136

A Simple Model for TCP Throughput



A Simple Model for TCP Throughput



Packet drop rate, $p = 1/A$, where $A = \frac{3}{8}W_{max}^2$

Throughput, $B = \frac{A}{\left(\frac{W_{max}}{2}\right)RTT} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$

138

Some implications: (1) Fairness

$$\text{Throughput}, B = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
  - Is this fair?

Some Implications: (2) How does this look at high speed?

- Assume that RTT = 100ms, MSS=1500bytes
- What value of *p* is required to go 100Gbps?
  - Roughly 2 x 10^{-12}
- How long between drops?
  - Roughly 16.6 hours
- How much data has been sent in this time?
  - Roughly 6 petabits
- These are not practical numbers!

140

Some implications: (3) Rate-based Congestion Control

$$\text{Throughput}, B = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- One can dispense with TCP and just match eqtn:
  - Equation-based congestion control
  - Measure drop percentage *p*, and set rate accordingly
  - Useful for streaming applications

Some Implications: (4) Lossy Links

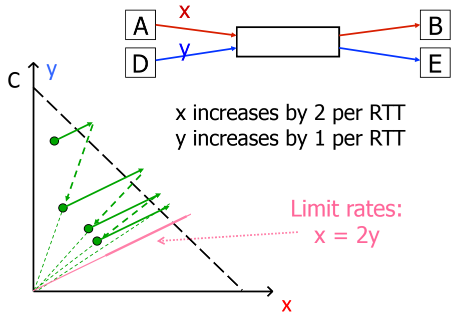
- TCP assumes all losses are due to congestion
- What happens when the link is lossy?
- Throughput ~ 1/sqrt(p) where p is loss prob.
- This applies even for non-congestion losses!

142

Other Issues: Cheating

- Cheating pays off
- Some favorite approaches to cheating:
  - Increasing CWND faster than 1 per RTT
  - Using large initial CWND
  - Opening many connections

Increasing CWND Faster

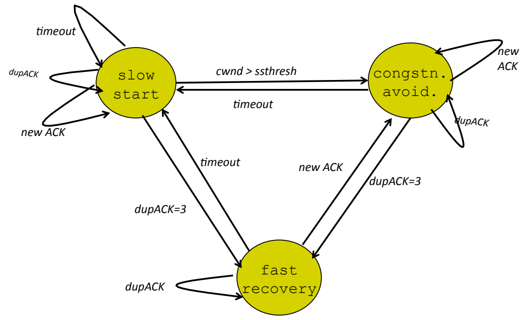


144

A Closer look at problems with TCP Congestion Control

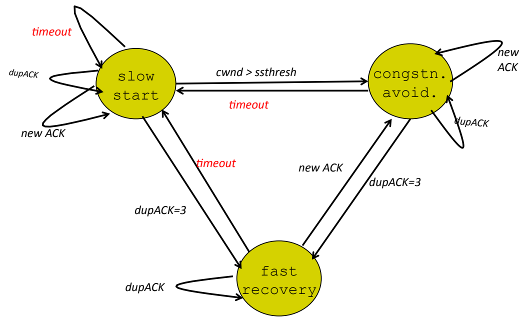
145

TCP State Machine

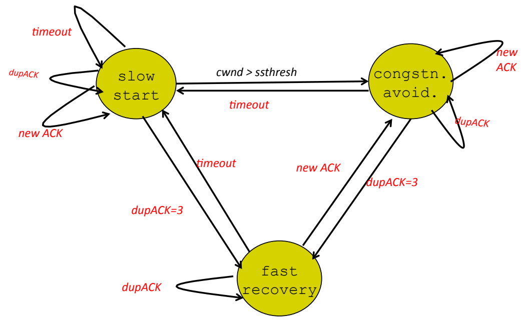


146

TCP State Machine

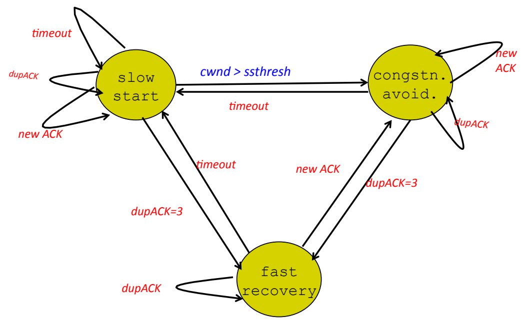


TCP State Machine



148

TCP State Machine



TCP Flavors

- TCP-Tahoe
  - CWND =1 on triple dupACK
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

Our default assumption

150

Interoperability

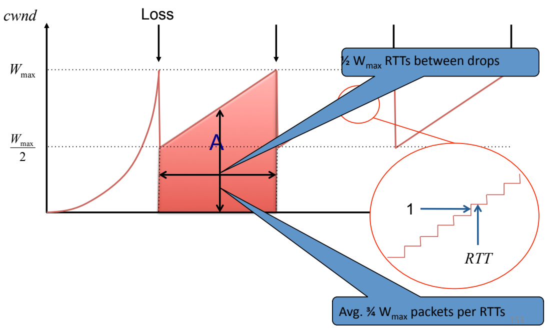
- How can all these algorithms coexist? Don't we need a single, uniform standard?
- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?

151

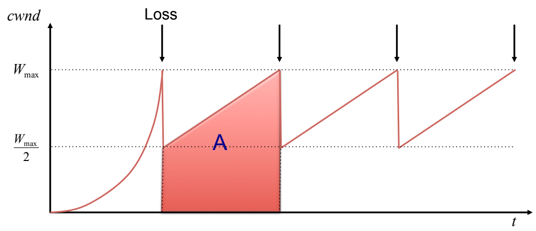
TCP Throughput Equation

152

A Simple Model for TCP Throughput



A Simple Model for TCP Throughput



Packet drop rate, $p = 1/A$, where $A = \frac{3}{8}W_{max}^2$

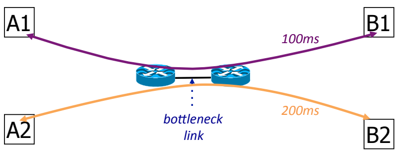
Throughput, $B = \frac{A}{\left(\frac{W_{max}}{2}\right)RTT} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$

154

Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



155

Implications (2): High Speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Assume RTT = 100ms, MSS=1500bytes
- What value of *p* is required to reach 100Gbps throughput
  - ~ 2×10^{-12}
- How long between drops?
  - ~ 16.6 hours
- How much data has been sent in this time?
  - ~ 6 petabits
- These are not practical numbers!

156

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND
- Other approaches?
  - Multiple simultaneous connections (hack but works today)
  - Router-assisted approaches (will see shortly)

157

Implications (3): *Rate*-based CC

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- TCP throughput is "choppy"
  - repeated swings between W/2 to W
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps
- A solution: "Equation-Based Congestion Control"
  - ditch TCP's increase/decrease rules and just follow the equation
  - measure drop percentage *p*, and set rate accordingly
- Following the TCP equation ensures we're "TCP friendly"
  - i.e., use no more than TCP does in similar setting

158

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput
- Limitations of TCP Congestion Control

159
159

Other Limitations of TCP Congestion Control

160

(4) Loss not due to congestion?

- TCP will confuse **any loss event** with congestion
- Flow will cut its rate
  - Throughput ~ 1/sqrt(p) where p is loss prob.
  - Applies even for non-congestion losses!
- We'll look at proposed solutions shortly…

161

(5) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (1): short flows never leave slow start!
  - short flows never attain their fair share
- Implication (2): too few packets to trigger dupACKs
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

162
162

(6) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Means that delays are large for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B

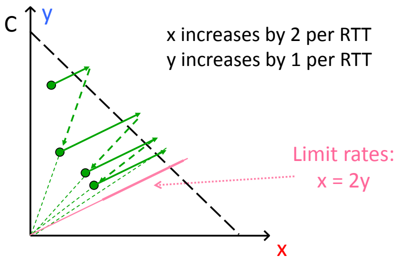
163

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT

164

Increasing CWND Faster



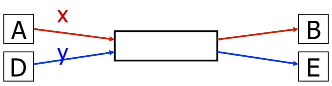
165

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections

166

Open Many Connections



Assume
- A starts 10 connections to B
- D starts 1 connection to E
- Each connection gets about the same throughput

Then A gets 10 times more throughput than D

167

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - Using large initial CWND
- Why hasn't the Internet suffered a congestion collapse yet?

168

(8) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering
- Sometimes we want CC but not reliability
  - e.g., real-time applications
- Sometimes we want reliability but not CC (?)

169
169

Recap: TCP problems

Routers tell endpoints if they're congested

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

170

- What does TCP do?
  – ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  – AIMD, Fast-Recovery, Throughput
- Limitations of TCP Congestion Control
- Router-assisted Congestion Control

171
171

Router-Assisted Congestion Control

- Three tasks for CC:
  – Isolation/fairness
  – Adjustment
  – Detecting congestion

172

How can routers ensure each flow gets its "fair share"?

173

Fairness: General Approach

- Routers classify packets into "flows"
  – (For now) flows are packets between same source/destination
- Each flow has its own FIFO queue in router
- Router services flows in a fair fashion
  – When line becomes free, take packet from next flow in a fair order
- What does "fair" mean exactly?

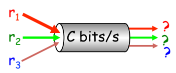
174

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

  $a_i = \min(f, r_i)$

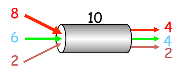
  where f is the unique value such that $\mathrm{Sum}(a_i) = C$



175
175

Example

- $C = 10$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$; $N = 3$
- $C/3 = 3.33 \rightarrow$
  - Can service all of r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8$; $N = 2$
- $C/2 = 4 \rightarrow$
  - Can't service all of r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$



$f = 4$:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

176

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

  $a_i = \min(f, r_i)$
- where f is the unique value such that $\mathrm{Sum}(a_i) = C$
- Property:
  - If you don't get full demand, no one gets more than you
- This is what round-robin service gives if all packets are the same size

177

How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin ("fluid flow")
- Can you do this in practice?
- No, packets cannot be preempted
- But we can approximate it
  - This is what "fair queuing" routers do

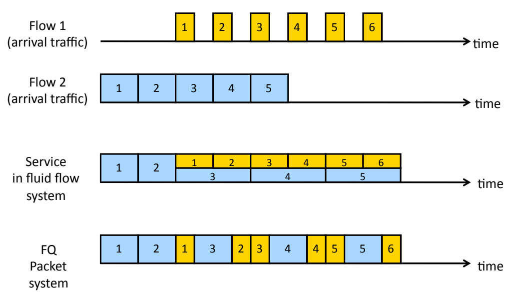
178

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit
- Then serve packets in the increasing order of their deadlines

179

Example



180

Fair Queuing (FQ)

- Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized
- Weighted fair queuing (WFQ): assign different flows different shares
- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90s, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

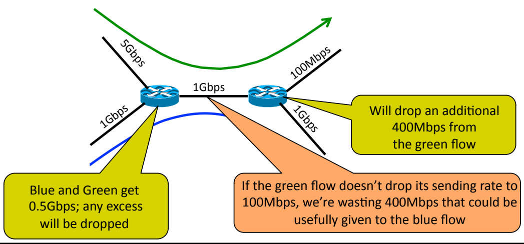
181

FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want
- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
  - robust to cheating, variations in RTT, details of delay, reordering, retransmission, *etc.*
- But congestion (and packet drops) still occurs
- And we still want end-hosts to discover/adapt to their fair share!
- What would the end-to-end argument say w.r.t. congestion control?

Fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Why shouldn't you be penalized for using more scarce bandwidth?
- And what is a flow anyway?
  - TCP connection
  - Source-Destination pair?
  - Source?

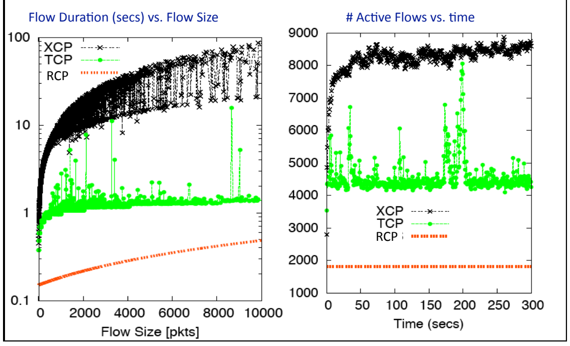
Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
  - Rate adjustment
  - Detecting congestion

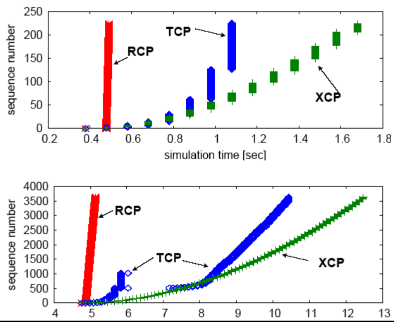
Why not just let routers tell endhosts what rate they should use?

- Packets carry "rate field"
- Routers insert "fair share" *f* in packet header
  - Calculated as with FQ
- End-hosts set sending rate (or window size) to *f*
  - hopefully (still need some policing of endhosts!)
- This is the basic idea behind the "Rate Control Protocol" (RCP) from Dukkipati *et al.* '07

Flow Completion Time: TCP vs. RCP (Ignore XCP)



Why the improvement?



189

Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
  - Rate adjustment
  - Detecting congestion

190

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - I.e., endhost reacts as though it saw a drop
- Advantages:
  - Don't confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - defined as extension to TCP/IP in RFC 3168 (uses diffserv bits in the IP header)

191

One final proposal: Charge people for congestion!

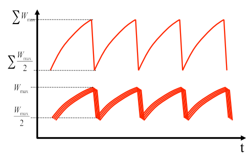
- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$
- Now, there's no debate over what a flow is, or what fair is...
- Idea started by Frank Kelly here in Cambridge
  - "optimal" solution, backed by much math
  - Great idea: simple, elegant, effective
  - Unclear that it will impact practice – although London congestion works

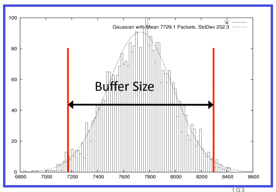
192

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





193

TCP in detail

- What does TCP do?
  - ARQ windowing, set-up, tear-down
- Flow Control in TCP
- Congestion Control in TCP
  - AIMD, Fast-Recovery, Throughput
- Limitations of TCP Congestion Control
- Router-assisted Congestion Control

194

Recap

- TCP:
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new, more optimal, approaches
  - though the needs of datacenters might change the status quos

195