TCP, Lecture 3

Internet Protocols

CSC / ECE 573

Fall, 2005

N. C. State University

Today's Lecture

I. TCP Error Detection and Correction

II. "Lost" ACKs and the Persist Timer

III. Estimating the Round Trip Time

IV. Calculating Retransmission Timeout Intervals

V. TCP Congestion Control

VI. The "Slow Start" Algorithm

TCP ERROR DETECTION AND CORRECTION

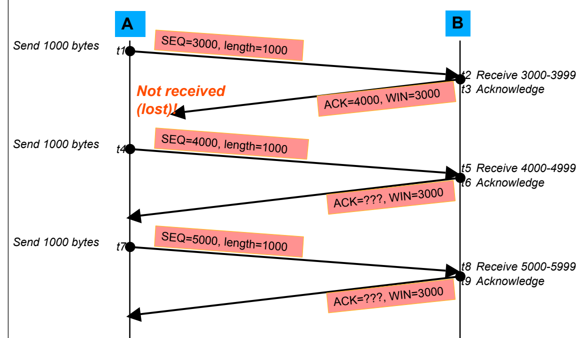
TCP Reliability

- Lost segments are detected and recovered
- Corrupted segments (invalid checksum) are detected and recovered
- Lost and corrupted segments are detected by failure to receive an acknowledgment
  - sender has to decide how long to wait before retransmitting (and how many times to retransmit)

TCP Reliability (cont'd)

- Duplicate segments are detected (using Sequence Number) and ignored by receiver (but acknowledgment will be sent to sender)
- Out-of-order segments are detected (using Sequence Number) and reordered (in reordering buffer)
  - receiver acknowledges only consecutively received segments
- Lost Acknowledgments may also lead (incorrectly) to retransmission of data
  - usually not a problem; why not?

Example of Lost ACKs



In-Class Exercise

A sends 1000 byte segment	SEQ=5000, ACK=4444
B sends 100 byte segment	SEQ=4444, ACK=6000
A sends 1000 byte segment	SEQ=6000, ACK=4544
A sends 1000 byte segment	SEQ=7000, ACK=4544
A sends 1000 byte segment	SEQ=8000, ACK=4544
B sends 100 byte segment	SEQ=4544, ACK=7000
A sends 1000 byte segment	SEQ=9000, ACK=4544
B sends 0 byte segment	SEQ=4644, ACK=7000
B sends 0 byte segment	SEQ=4644, ACK=9000
A sends 1000 byte segment	SEQ=10000, ACK=4644

LOST ACKS AND THE PERSIST TIMER

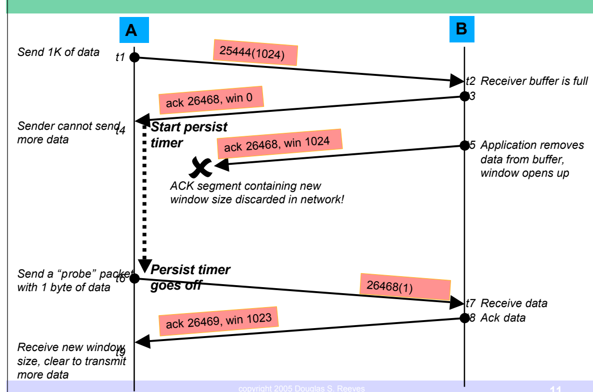
Problem Caused by A Lost ACK

- It is possible for receiver to advertise a Window Size = 0
  - fast sender + slow receiver = full receiver buffer
- When receiver (eventually) empties the buffer, requests more data by sending an ACK with Window Size > 0
- If this ACK is lost by network → deadlock!
  - sender can't send anything
  - receiver has no reason to send another ACK

The Persist Timer

- Solution: after receiving any ACK with Window Size = 0, start a *persist timer*
  - upon expiration of the persist timer, send a *probe* packet
  - probe packet = 1 more byte of data (technically, violates Window Size advertisement)
- Normal persist timer intervals
  - 1.5s, 3s, 6s, 12s, 24s, 48s, 60s, 60s, 60s, …
- The ACK of this probe replaces the lost ACK
  - i.e., transmitting the probe "stimulates" new advertisements

Persist Timer, Illustrated



ESTIMATING THE ROUND-TRIP TIME (RTT)

Motivation

- How long should we wait before deciding a segment has been lost and should be retransmitted?
  - too short → ?
  - too long → ?
  - just right → just a little bit longer than the longest time it could take to transmit a segment and receive an **ACK**

copyright 2005 Douglas S. Reeves

Estimating RTT

- Notation
  - $\mathtt{rtt}_i$ = time between transmission of ith packet until receive ACK of ith packet
  - $\mathtt{RTT}_i$ = estimate of average round trip time after i^{th} packet
- Exponentially weighted moving average (EWMA):

$$RTT_i = \alpha * RTT_{i-1} + (1-\alpha) * \mathtt{rtt}_i$$

  - (assume $RTT_0 = 0$)
  - Jacobson's algorithm: $\alpha = .875$

copyright 2005 Douglas S. Reeves

Example of RTT Calculation

- Notes: calculations in full precision, results shown only to nearest integer, time before packet #20 not shown

Pkt #	$\mathtt{rtt}_i$ (ms)	$\mathtt{RTT}_i$ (ms) $\alpha = .875$
20		40
21	30	39
22	80	44
23	40	43
24	130	54
25	40	?

copyright 2005 Douglas S. Reeves

Small Correction

- Problem: if a segment is retransmitted, is an ACK for the first, second, … or n^{th} transmission of that segment?
  - can't tell how to compute $\mathtt{rtt}_i$ in this case
- Solution ("Karn's Algorithm"): don't use (i.e., ignore) $\mathtt{rtt}_i$ from any retransmitted segments to update $\mathtt{RTT}_i$

copyright 2005 Douglas S. Reeves

Calculating the Variation in the RTT

- Motivation: ???
- True standard deviation (expensive to compute):

$$\mathtt{STDDEV} = ((\mathtt{rtt}_1 - \mathtt{avg(rtt)})^2 + (\mathtt{rtt}_2 - \mathtt{avg(rtt)})^2 + \ldots)^{1/2}$$

- The "*mean deviation*" (cheap to compute):

$$\mathtt{MDEV}_i = (1-\rho) * \mathtt{MDEV}_{i-1} + \rho * |\mathtt{rtt}_i - \mathtt{RTT}_{i-1}|$$

  - another EWMA!
  - recommended value for $\rho = .25$

copyright 2005 Douglas S. Reeves

Example of MDEV Calculation

Pkt #	$\mathtt{rtt}_i$ (ms)	$\mathtt{RTT}_i$ (ms) $\alpha = .875$	$\mathtt{MDEV}_i$ (ms) $\rho = .25$
20		40	10
21	30	39	10
22	80	44	18
23	40	43	14
24	130	54	32
25	50	54	25

copyright 2005 Douglas S. Reeves

CALCULATING THE RETRANSMISSION TIMEOUT INTERVAL

ACKs and Retransmissions

- When a segment is sent, a retransmission timer is started
- If the segment is **ACK**'ed before the timer expires, the timer is turned off
- Otherwise, the segment is retransmitted and timer is started again
- Notation
  - RTO_i = retransmission timeout interval for the ith packet

copyright 2005 Douglas S. Reeves

Determining RTO

- RFC 793 originally recommended:

  $RTO_i = 2 * RTT_{i-1}$

  - (i.e. allow two roundtrip times before timing out)
- Problem
  - when RTT_i has a high standard deviation, this method frequently times out too quickly (doesn't wait long enough)

copyright 2005 Douglas S. Reeves

Example of RTO Calculation (Original)

Pkt #	rtt_i (ms)	RTO_i	RTT_i (ms) $\alpha = .875$
20			40
21	30	80	39
22	80	78	44
23	40	88	43
24	130	86	54
25	50	108	54

Retransmission timeouts would occur! (retransmissions not shown, Karn's algorithm not used)

copyright 2005 Douglas S. Reeves

Improved Method of Computing RTO

- Solution
  - set **RTO** = **RTT** + a multiple of the mean deviation
- Jacobson's algorithm:

  $RTO_i = RTT_{i-1} + 4 * MDEV_{i-1}$

  - calculations can be implemented in a very efficient way

copyright 2005 Douglas S. Reeves

Example of RTO Calculation (Improved)

Pkt #	rtt_i (ms)	RTO_i	RTT_i (ms) $\alpha = .875$	$MDEV_i$ (ms) $\rho = .25$
20			40	20
21	30	120	39	18
22	80	109	44	23
23	40	138	43	19
24	130	118	54	36
25	50	196	54	28

Retransmission still occurs! (no retransmissions shown, karn's algorithm not used)

copyright 2005 Douglas S. Reeves

The TCP "Coarse-Grained" Timer

- For efficiency, TCP timers tick in "large" increments
  - e.g., `rtt` is measured to the nearest 500ms
  - result: RTTs and RTOs may differ from expected times due to this discretization
- See Stephens Vol I for more implementation details (fixed point arithmetic, etc.)

Retransmission Timer Backoff

- Retransmission may occur because `RTO` is too short
  - e.g., if the network congestion increases, RTO should be increased
  - problem: Karn's algorithm says don't modify RTT / MDEV for retransmitted segments!
- Solution: *exponential backoff*
  - upon timeout, retransmit and compute:
    $RTO_i = 2 * RTO_{i-1}$
    and no change to RTT

Retransmission Timer Backoff (cont'd)

- Example: RTO = 2s,
  after first timeout = 4s,
  after second timeout = 8s, etc.
- Questions
  - why is this necessary?
  - similar to anything else you know about?
  - how long should TCP keep trying until giving up?

TCP CONGESTION CONTROL

Fairness

- When congestion occurs, everyone using the same bottleneck link should reduce their transmission rate in a way that is fair
  - the most ambiguous word in English language?
  - TCP congestion control is *proportional-fair*
- Control mechanism is end-to-end, operated by end systems
  - who enforces fairness and what's the punishment for violating fairness?

TCP Congestion Control

- *Congestion control*: reduce transmission rate to match the current network bandwidth available
  - depends on the path to the destination
  - dictated by the "bottleneck" link
- Assumption: packet loss is due to congested routers, not transmission errors
  - may be incorrect for some technologies, such as wireless
  - what difference does it make?

TCP Feedback

- Implicit feedback: the source infers network conditions from the feedback (acknowledgments) it receives
- Info available to the source
  - ACKs received
  - measured **`rtt's`**
  - retransmission timeouts

copyright 2005 Douglas S. Reeves

TCP Sliding Windows

- Adapts rate by controlling the *congestion window*
  - smaller congestion window ➔ send at lower rate
  - larger congestion window ➔ send at higher rate
- Sliding windows are used for two purposes
  - flow control: **`Window Size`** advertisement
  - congestion control: **`cwnd`** (congestion window)
    - computed by source, not explicitly communicated
- **`Transmission_Window`** = min(**`Window Size`**, **`cwnd`**)

  → sender slows down to the rate of the slowest component (network or receiver)

copyright 2005 Douglas S. Reeves

TCP "SLOW START"

TCP Congestion Control Overview

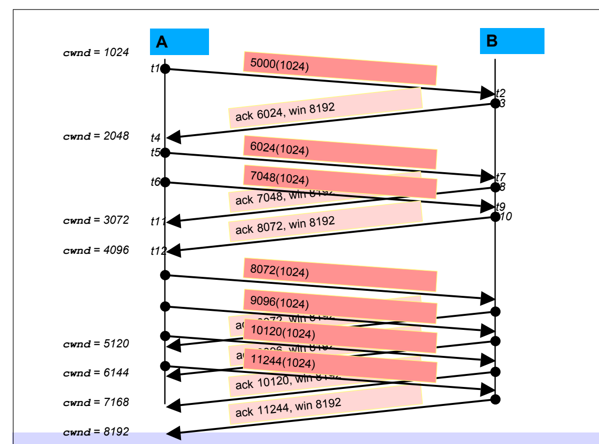
1. Start with very slow transmission rate (**`cwnd`** = 1 segment per rtt)
2. Probe for available bandwidth by increasing **`cwnd`** until packet losses start occurring
   - *Slow Start* = increase rapidly
   - *congestion avoidance* = increase more slowly
   - Slow Start threshold (**`ssthresh`**) marks transition from rapid to slower increase phase
3. When a packet loss is detected (whoops, increased too far), start over (reset **`cwnd`** to 1)

copyright 2005 Douglas S. Reeves

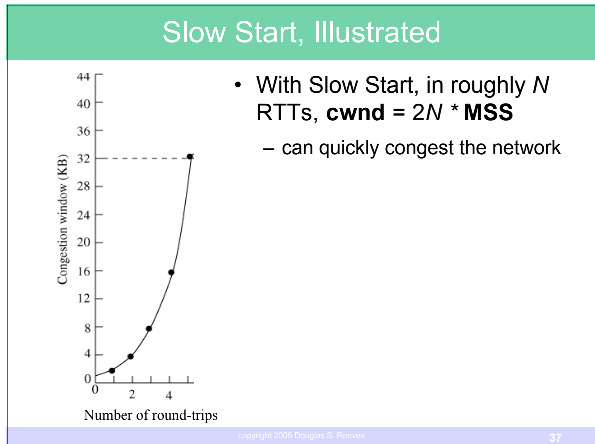
Slow Start

- Notes
  - **`MSS`** = maximum segment size
  - all quantities in units of bytes
- At connection establishment:
  `cwnd` ← MSS
  `ssthresh` ← 65535
- Upon arrival of each new ACK:
  `cwnd` ← **`cwnd`** + MSS
- Not a particularly slow increase!
  - **`cwnd`** doubles once every round-trip time (**`RTT`**)

copyright 2005 Douglas S. Reeves



Slow Start, Illustrated



- With Slow Start, in roughly *N* RTTs, **cwnd** = 2*N* * **MSS**
  - can quickly congest the network

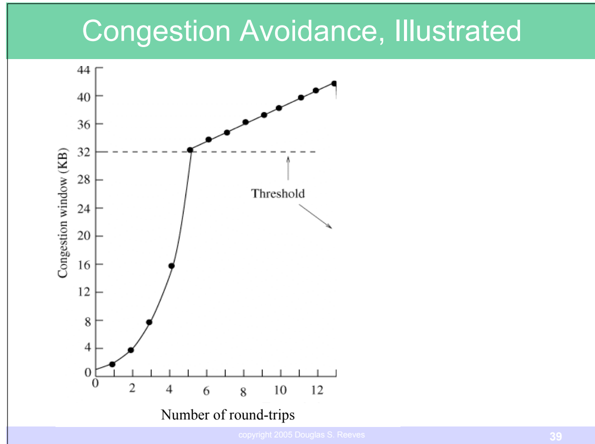
copyright 2005 Douglas S. Reeves

Congestion Avoidance

- Slow down the rate of increase as you approach the most recent "point of congestion"
  - **`ssthresh`** is used for this purpose
- If **`cwnd`** > **`ssthresh`** , upon arrival of each new ACK:
  `cwnd` ← **`cwnd`** + **`MSS`**^{2} / **`cwnd`**
- Result: **`cwnd`** approximately increases by one segment every **`cwnd`** ACKS

copyright 2005 Douglas S. Reeves

Congestion Avoidance, Illustrated



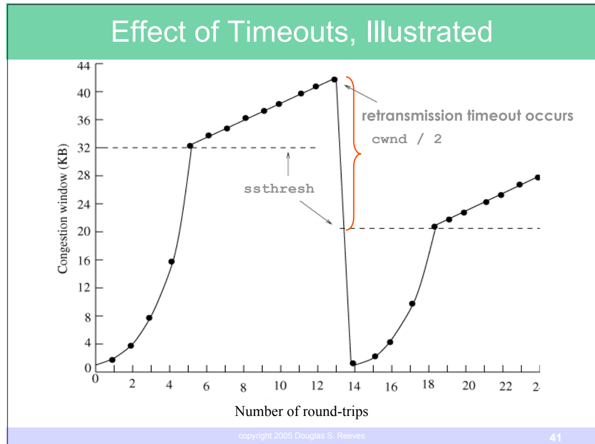
copyright 2005 Douglas S. Reeves

Reacting to Congestion

- Congestion may affect delivery of data and/or acknowledgments
- When a retransmission timeout occurs,
  `cwnd` ← **`MSS`** */* start all over again */*
  `ssthresh` ← max(2, ½ ***`Transmission_Window`**)
- Result: start over, but switch to congestion avoidance halfway to point where congestion previously occurred

copyright 2005 Douglas S. Reeves

Effect of Timeouts, Illustrated



copyright 2005 Douglas S. Reeves

Example

Time	Event	cwnd	ssthresh
t1	(assume)	1024	2048
t2	send S1		
t3	receive ACK of S1	2048 (=1024+1024)	
t4	send S2		
t5	send S3		
t6	receive **ACK** of S2	3072 (=2048+1024)	
t7	send of S4		
t8	send of S5		
t9	receive **ACK** of S3	3413 (=3072+341)	
t10	receive **ACK** of S4	3720 (= 3413+307)	
t11	send of S6		
t12	send of S7		
t13	receive **ACK** of S5	4002 (=3720+282)	
t14	receive **ACK** of S6	4264 (=4002+262)	
t15	timeout of S7 occurs	1024	2132

Summary

- TCP uses **`Sequence #`** and **`ACK #`** to detect errors
- Calculating the retransmission timeout interval is surprisingly complicated
  - RTT average computed as EWMA of `rtts`
  - Mean deviation computed
  - RTO = RTT + 4*MDEV
  - exponential backoff when losses occur

Summary (cont'd)

- TCP uses end-to-end congestion control with implicit feedback
- The congestion window provides congestion control
  - Slow start uses rapid initial increase, then slows down to avoid congestion

Next Lecture

- TCP, lecture 4