Chapter 21, 22, 23, 24 – TCP Timeout, Retransmission, and Congestion Avoidance

TCP's timeout routines are driven by 2 physical timers

The 200 ms timer
- ACK generation

The 500 ms timer
- Connection timeout
- Retransmission timeout
- Persist timeout

The OS invokes a TCP timer exit each time one of these ticks.

Logical timers that will be discussed:

Retransmission:	How long to wait after a packet for an ACK
Persist:	Keeps window size information flowing
Keepalive:	A watchdog timer on idle connections.
2MSL:	Keeps track of connections in the TIME_WAIT state

Retransmission timeouts:

Two time values are at the core of the problem

Round trip time (RTT)
Retransmission timeout (RTO)

RTO < RTT => Protocol won't work at all since EVERY packet times out

Retransmission timeout must be a function of round trip delay

At the datalink layer RTT is basically fixed
At the transport layer it can have *extreme* variability

Too fast a timeout results in unneeded retransmissions and wastes bandwidth
Too slow a timeout hurts responsiveness..

A transport layer protcol in which error recovery is timeout driven
must use a dynamic estimate of RTT and RTO

Retransmissions must use an exponential backoff strategy

Backoff strategy slower than exponential can lead to *congestion collapse*
Double the delay after each retransmission
Max delay is 64 seconds
Max attempts is 12 tries
After that close the connection

Determining the base retransmission timeout (RTO) value

We need a dynamic procedure that takes into account *current* congestion

The original algorithm

M = measured value of last ack's RTT delay.
R = averaged round trip time
$R = aR + (1 - a) M$ (with a recommended to be 0.9)
$RTO = Rb$ (b = a variance fudge factor recommended to be 2)

Problems with the original algorithm

Not sensitive enough to fluctuations in load
Doesn't dynamically consider the variance (c.f. open queueing systems)
Short term overloads ==> Many packet timeouts ==> Many retransmissions

Jacobsons's new algorithm for estimating RTO.

M = *Measured* value of a single round trip transmission time.
R = an *Estimator* of the current RTT.
D = a smoothed estimator of the mean deviation.

For each Ack received compute:

$Err = M - R$
$R = R + g * Err = gM + (1 - g)R$ (g = gain = 0.125)
$D = D + h * (|Err| - D)$ (h = gain = 0.25)
$RTO = R + 4D$

Thus variance is explicitly and dynamically considered in the calculation

Karn's improvements

Don't adjust R or RTO for acks for retransmitted packets
(because you don't know if the ACK is actually for the original)

Could lead to deadlock like situation..
Suppose time required increases radically..
Every packet would have to be retransmitted multiple times (until backoff passed new time.)
A new RTO would thus never be computed.

Solution to possible deadlock
Continue to use a (successful) backoff value until a valid new sample is obtained.

Further complications

TCP uses the 500ms timer to compute R
Computation is performed by counting ticks while waiting for ACK's.
If multiple packets are outstanding at any point in time,
only one is being timed and
the timing precision is 1/2 second.

How does all this work for high performance nets or even uncongested LAN's?

Congestion aviodance

Recall *Slow Start* from Chpt 20.
Sender can only send the min(usable_window, congestion_avoidance_window *cwnd*).
Initial value is of *cwnd* 1 segment.
Value is increased by one each time a segment is acked

	cwnd
Send	1
wait	(1 ack recvd)
Send	2
Send	2
wait	(2 acks rcvd)
Send	4

Value quickly climbs up to the offered window.

The congestion avoidance algorithm:

The congestion avoidance algorithm adds another variable *ssthresh* (initially 64KB).

Congestion indicators:

```
Receipt of multiple acks (typically 3) for the same packet
    Possible causes of duplicate ack include
        Lost segment
        Out of order delivery
    Lost segments are caused by congestion
    (TCP must ack on receipt of out−of−sequence packet)

Occurrence of a timeout.
```

When congestion is detected set *ssthresh* to max(min(offered window, *cwnd*) / 2, 2).
Retransmit the packet thought to be lost.

```
If timeout was the cause
    set cwnd to 1 segment
else (triple acks)
    The SigComm paper didn't address the adjustment of cwnd
    Presumably either:
        triple acks weren't responded to and timeout ensued
        or
        cwnd /= 2


When an ack for new data is received
    if (cwnd < ssthresh)
        cwnd += 1                   /* The slow start algo          */
    else
        cwnd += (1 /cwnd) ;         /* The cong avoidance alg.      */

Transmit a new segment if allowed by cwnd.
```

Growth of *cwnd* during congestion avoidance.

The objective is that after a full window of ACKs are received cwnd += 1 seg;
(Note that book includes a discussion relating to a Berkeley inspired bug that added 1/8 seg each time.)

Suppose that we view the cwnd in segs.

Let $cwnd_s = 4$ Segs.
Then $1/\ cwnd_s = 1/4$
After 4 segs are received
$cwnd_s \sim= 4 + 1/4 + 1/4 + 1/4 + 1/4 = 5$
The value 5 is something of an over estimate since at the second addition
we should really add 1 / 4.25 etc...

Some rescaling is clearly necessary here because *cwnd* is actually maintained in *bytes*

$cwnd_b = cwnd_s * MSS \Rightarrow 1/\ cwnd_s = MSS\ /\ cwnd_b$

$cwnd_b$ += $(MSS / cwnd_b) * MSS$ + (MSS/8 <= Th BSD bug/hack)

cwnd	MSS	Segs/wnd
4096	512	8
After Ack	New Cwnd	W/o 1/8 Adjust
0	4096	4097
1	4224	4161
2	4350	4224
3	4474	4286
4	4597	4347
5	4718	4408
6	4838	4467
7	4956	4526
8	5073	4584

Fast Recovery

TCP Tahoe implemented slow start, congestion avoidance, and fast retransmit as just described..

Tahoe solved:

The problem of extreme congestion build up at bottleneck routers

Tahoe didn’t solve:

Major reduction in packet flow at each loss..
In congested nets connections tended to keep a full cwnd of packets in flight.
Packet loss led to pipeline drains.

Yet another refinement (TCP Reno – fast retransmit / fast recovery algorithm)

Documented in "Modified TCP Congestion Avoidance Alg" , end2end mailing list, Apr 1990 (V. Jacobson)

Receipt of multiple ACKs a potential indicator of lost segment.

Receiver of out of order segments *requires* generation of the expected ack.
Probability of out of order delivery is inversely proportional to # duplicate acks.
Thus, the 3rd duplicate ACK is considered lost segment indication.

When 3rd duplicate ACK is received

Set *ssthresh* to max(min(offered window, *cwnd*) / 2, 2)
Retransmit the missing segment
Set *cwnd* to *ssthresh + 3 * MSS*

For each additional duplicate ACK
Increment *cwnd* by *MSS* and transmit if allowed

At the next ACK of new data
Set *cwnd* to *ssthresh*

Points to note in figure 21.10 and 21.11

1 – Value of ssthresh stays at previous setting of 512 during congestion avoidance.
2 – At third dup ACK, ssthresh is set to cwnd/2 rounded down to next MSS multiple.
3 – *cwnd* is set to 1024 + 3 * 256, and the missing segment is retransmitted. Note that the sender is now forced to stop because usable window = 0
4 – As each ack arrives add 1 MSS (as would be done in Slow start)
5 – When cwnd reaches 2560, usable window right edge is 6657+2560=9217. Hence 6961:9217 can be sent. The idea is to try to minimize the pipe draining effects. Thus for each ack we increase *cwnd* by 1 MSS and can keep sending.
6 – When ack for retransmitted segment finally arrives, *cwnd* is set to 1024 + 256?,
but the right edge of the usable window is 8961+1280 = 10241 (a good jump up
from 9473)

TCP Reno implements slow start, congestion avoidance, fast retransmit and fast recovery
TCP Reno can (sometimes) recover from a single segment per window without blocking.
TCP Reno can almost always recover from a single segment loss without a pipeline drain.
(In the example in the book... some blocking does occur but total drain doesn't happen).
TCP Reno cannot generally recover from multiple drops per window / RTT.

Other proposals:

TCP NewReno (Janey Hoe – > Sally Floyd)
On a partial ack retransmit the segment acked
SACKs (Sally Floyd)
TCP Vegas

6. Suppose MSS = 1000 bytes, cwnd = 9000, snd.seq = 17000, snd.ack = 12000
and the offered window is 32000.

a. If the segment that has sequence 12000 was actually lost, how
many more segments can the sender send before having to stop.
(Assume NO timeout occurs).

```
The right edge of the congestion window is  12000+9000 = 21000.
The segments at 17000, 18000,  19000, and 20000 can be sent.
Thus the answer is 4.
```

b. Suppose that the sender has stopped when the 3rd duplicate ack
arrives. What the receipt of the third duplicate ack cause
ssthresh and cwnd be set to.

ssthresh – `9000/2 = 4000 (rounded down)`

cwnd –`4000 + 3000 = 7000`

c. How many additional duplicate acks will have to be received
before a NEW segment can be sent.

```
Now the right edge of the congestion window is
12000 + 7000 = 19000

Assuming the sender stopped after sending 20000 the
right edge of the congestion window must be 22000
before it can send again. Thus three additional
duplicate acks must be received.
```

d. Assuming that ONLY the 12000 segment was lost what will be the
leading edge of the usable window when the first ack for new
data is received.

```
The first ack for new data will have ack=21000
since the sender sent the packet with seq # 2000
before the retransmission.

Thus the right edge of the congestion window will be either
21000+4000 = 25000 or 21000+4000+1000 = 26000
```

TCP Dynamics

For TCP connections on WANs

the available bandwidth is typically << the bit rate of the LAN to which the host is attached.
the latency is typically 60 msec or more

Suppose *cwnd = 20,000 bytes.* For a 100 Mbps LAN the time to send a complete window is

8 * 20000 / 10^8 = 1.6 * 10^5 / 10^ 8 = 1.6 msec

A sender will typically burst the whole window and then wait 58.4 msec for the ack to return.

The effective throughput is 1.6 * 10^5 / 60 * 10^–3 = 2.666 Mbps

TCP Vegas

Claimed advantages over Reno
40–70% improvement in throughput
20–50% fewer bytes transmitted.

RTT computation

Computed by sender for each segment sent using a decent resolution clock..
No timestamps to send and echo..

On *first* duplicate ack

Check difference in current time and stamp for identified packet.
If greater than RTT retransmit immediately..

On first or second non duplicate ack after a retransmission

Check difference in current time and stamp for identified packet.
If greater than RTT retransmit immediately..
(This is partial ack handling)

Claim on cwnd managment

Reno will shrink cwnd more than once because of multiple drops / window
Vegas will shrink cwnd only once.

The main innovation is anticipatory window size adjustment..

Recall Little's law: N = RX

N = population (bytes in the pipe)

R = response time (RTT)

X = throughput (bytes per second received)

X = N / R might seem to indicate that by increasing N one can increase X but throughput is bounded by the available bandwidth.. At some point X ceases to increase and R increases linearly with N.

Vegas maintains a variable called BaseRTT = min{all RTT's}

An "expected" X = Window size / Base RTT is computed..

An "actual" X is computed as follows

A distinguished segment is identified

When its ack arrives N is set to nxtsnd – acknum (number of bytes sent since the distinguished segment was sent (this is typically = *cwnd* since the sender typically always sends a full window and then waits.

R is set to the RTT of the distinguished segment

"actual X" = N / R ~= *cwnd* / R < *cwnd* / BaseRTT

If the actual RTT > Base RTT the actual throughput will be < expected.
diff = expected – actual and thus *diff* >= 0.

Two thresholds measured in bytes / second are used to adjust cwnd
diff is a throughput measure that indirectly identifies the number of extra bytes in the network...

```
N_extra  = diff * Base_RTT

If (N_extra <  alpha)
        linearly increase cwnd during the next RTT
if (N_extra  > beta)
        linearly decrease cwnd during next RTT
```

Recommended values for alpha and beta are 2 MSS and 4 MSS

Example:

Base RTT = 10 ms
Window = 12,000
Actual RTT = 15 ms
Expected X = 12,000 / 0.010 = 1200000
Actual X = 12,000 / 0.015 = 800000
Diff = 400000
N_extra = 400000 * 0.010 = 4000

Chapter 22 – TCP Persist Timer and the Silly Window Syndrome

Persist timer

- Persist timer is driven by the 500 ms timer
- Used by a sender with a size 0 usable window to ask for a window update
- Prevents a possible deadlock should the window update be lost
- Exponential backoff is used
  - An initial value of 1.5 sec for a LAN is typical
  - Actual delay is constrained to [5 sec, 60 sec]
- Persist messages just send 1 byte of data
  - If no receiver space is available the receiver discards it.
- Persist is infinitely persistent.

Silly window syndrome

- Window full of small segments
- Each ack allows one more small segment to be transmitted
- Can be triggered by a receiving application that slowly consumes small chunks.

Effects can be mitigated by:

- A non greedy sending policy that waits until
  - a – at least one MSS can be sent.
  - b – 1/2 max size window ever advertised can be sent
  - c – We have no unacked data and can send everything we have queued.

- A receiving policy that witholds window updates (withhold ACKs at your peril)
- Don’t open the window until it can move at least
  - a – 1 MSS or
  - b – 1/2 receiver buffer space whichever is smaller.

Persist timer can cause transmission of small segments.

Chapter 23 – Keepalive timer

Not part of the spec but can be used to detect failed hosts on idle connections

Send a garbage byte (one with bad seq number) or (no byte at all) every 2 hours

Receiver is forced to ACK with next expected byte
Receiver will send a RST if it has crashed and come back up

Disadvantages

Can terminate a connection because of a transient failure.
Causes extra packets
Causes extra $$$'s

Advantage

Can detect failure of the other end of a lightly used connection
If failure is undetected a half–open connection can persist *forever.*
Intentional creation of a zillion such connections can be an effective denial of service attack.

Controversy:

Shouldn't this function be in the Application Layer if the Application wants it.

Chapter 24 – Futures

Bandwidth versus latency limited communications

Suppose we have a fixed latency of 30ms to send a 1 MB file across the US

Bandwidth	Time	Effective Rate
1.544 Mb	5.21	1.535 Mb
45Mb	0.28	35.71 Mb
1Gb	0.038	211 Mb
2Gb	0.034	235 Mb

Morals:

No matter how large your file you eventually become latency limited.

Effects on Stop and Wait protocols such as:
3 way handshakes and
slow start are particularly bad

In summary:

Speed of light limit imposes a lower bound on the minimum service time you can provide any customer

Increasing band width allows you to provide a service time that is arbitrarily close to that minimum to an arbitrarily large number of customers

So... LFN's pose two problems to TCP

1 – Performance ––> small max window size makes effective bandwidth ridiculously small.

2 – Reliability ––> big window size that eliminates the performance bottleneck can lead to sequence number wraps and cause the delayed duplicate problem.

PERFORMANCE ISSUES

Impact of 64K window size limit on performance in LFN’s

For a 1GBit transcon network

assume RTT = 60 msec
the bandwidth delay product is 7.5 MB

64 KByte window size would greatly reduce throughput

2^{19} bits / 2^{30} bps = 2^{-11} seconds to send a window.
This value is about .5 msec ==> 1 /120 of available bandwidth actually used

==> Effective bandwidth is limited by WindowSize / RTT
2^19 / 0.06 = 8738133 bits / sec

(But this is good throughput even today on the vBNS)

Possible solution: larger windows

Impact of lost packets on performance in LFN’s

A stop & wait exchange requires an RTT for *each packet sent.*
Thus TCP loses *at least* one RTT’s worth for every timeout and slow start..

Impact of latency bound RTT in LFN’s
10^9 bps * 30 * 10^{-3} sec = 30 * 10^6 bits
Thus a 1Gb network with a 30ms RTT is *guaranteed* to waste more than 30Mb of capacity for each timeout!!
Mitigating factor: Realistic transcon throughput is closer to a max of 10Mbps (even for the vBNS)

Fast transmit and fast recovery can recover from *1* packet drop per window without a stop and wait exchange.

For fixed probability of packet loss, larger windows ==> higher probability of multiple drops / window.

Possible solutions
- SACK’s (selective ACK’s)
  - Permit the sender to know exactly what’s missing.
  - Have questionable value in Non LFN’s and aren’t yet standard.
- NewReno
- Vegas

Large Window Sizes and Effective RTT measurements

Existing procedure is driven by 500 ms timer
Times only a single segment per window
Large window size ==> inadequate sampling.

Possible solution
    Vegas type timing (with multiple distinguished segments per window)
    Timestamps

RELIABILITY ISSUES

Limits on window size:

For correct operation of *any* sliding window protocol the leading edge of receiver window must not wrap and overlap trailing edge of the sender window.

If this should occur, the protocol will fail on a lost ack and retransmission. Therefore window size must be <= 2^31.

The above rule suffices for building a reliable *link layer* protocol.

However, a more stringent requirement is needed if delayed duplicates exist in the net.

Dealing with delayed duplicates

Two potential sources of delayed duplicates:

1 – Fast Wrap on the current connection.
2 – Carry over from earlier incarnation of *this* connection.
(source IP, source port, dest IP, dest port)

Problem 2 is addressed by the Time–Wait state in closing a connection

For problem 1, RFC 1323 states that a constraint on the maximum *effective* bandwidth for error free operation is:

$$B * MSL\ (secs) < 2^{31}$$

Exercise: This constraint is clearly sufficient but is it truly necessary?

That is:
If you transmit at max rate you can only consume 1/2 the sequence number space within an MSL (Maximum segment Lifetime)

Said another way:
if MSL is 2 minutes, the max safe bandwidth is 2^31 / 120 or about 2^24 Bps

Example: Here T−Wrap is 2^31 and corresponds to max MSL for safe operation.

Network	bps	T−Wrap (Seconds)	
ArpaNet	56kb	300000	3.6 Days
T1	1.544Mb	10000	3 Hours
E−net	10Mb	1700	30 Min
T−3	45Mb	380	6 Min
Fddi	100Mb	170	3 Min
Gigabit	1Gb	17	

Window size has *no* impact on the fast sequence number wrap problem

A large window is necessary to get reasonable performance when bandwidth−delay product is large

but... the sequence number wrap problem can occur even with a small window.

Example:

FDDI Lan with diameter of 1km
RTT = $2 \times 10^{3} / 3 \times 10^{8} = 6.7 \times 10^{-6}$ seconds
Bandwidth delay product is 12.5×10^{6} bytes / sec x 6.7×10^{-6} = 83.7 bytes
==> 100 % utilization possible at an 83.7 byte window
Sequence number wrap in a perilously low 3 minutes

Possible solutions to sequence number wrap problem:

Increase sequence # to 64 bits
Use a time stamp to "augment" the sequence number.

RFC 1323 recommended solutions.

Window size:

Window scale factor option provides *very* large windows.

3 bytes

Kind	– 3
Length	– 3
Shift	– 0 – 14

Option is exchanged at startup time in syns and not thereafter.
Value *transmitted* is the value to be used for the transmitters receive window

Maximum value of 14 is chosen so that sender + receiver window space is $< 2^{31}$ (which make sense if you buy the argument that no more than 2^{31} bytes can safely be in flight within an MSL)

Timestamp option:

Used for both RTT measurement and for PAWS

10 bytes

Kind	– 8
Length	– 10
TSval	– 4 bytes
TSecr	– 4 bytes

Sender sends a timestamp and receiver sends it back..
Allows for better calculation of RTT
Why not just *remember* it at the sender... (c.f. Vegas).

Determining which timestamp to echo:

Normally receiver just remembers last TS received and echos it.

Delayed ACK's:
    Use TSVal from earliest unacked seg

Our of order segment
    Use TSVal from last segment that advanced the window
    This will lead to overestimation of RTT.. which is probably good when congestion is occuring.

Filled hole in the sequence number space
    Use TSval from the seg that filled the hole

PAWS (Protection against wrapped sequence number)

The main benefit of actually *sending* the TS is that it *totally defeats* the fast sequence number wrap delayed duplicate problem.

Assume time stamps are non decreasing.

As segments are received timestamps are remembered (the TS used in the echo reply is the one remembered.)

Segment with a decrease in timestamp can be discarded as a dup.

Constraints on the Timestamp clock..

    Not too slow –– must tick at least once for each 2^{31} bytes sent.
    Not too fast –– must not recycle in less than MSL segments..
    BSD uses 1 tick per 500 ms

Congestion management in gateways

Buffering in gateways (a.k.a. routers)

- Gateways have multiple ports
- Congestion results when input and output loads are unevenly distributed...
  - either long term
  - and/or
  - due to the bursty nature of network traffic
- *Some* buffering is desirable for handling bursty traffic
- Persistent congestion should *not* be addressed by additional buffering
- Arbitrarily large numbers of buffers can lead to arbitrarily large delays

Buffer control in congestion management

- Drop tail
  - When queue length >= N, drop packets instead of queueing them for transmission..
  - Easy but..
    - Induces synchronized behaviour in the network.
      - ==> Links become idle & throughput goes down
    - Doesn’t target aggressive users.

Smarter Dropping algorithms:

- Random Drop
  - Drop a random packet from the queue
- Early Random Drop
  - When queue length reaches N/2 drop a packet with Prob $p = 0.02$ (in one study
- Effectiveness in targeting agressive users and reducing synchronization
  - remains a point of contention

- Random Early Detection
  - Driven by AVERAGE rather than instantaneous queue length
  - Average is typical exponential moving average
  - Two thresholds are used
    - T_1 at this level begin marking /dropping
    - T_2 mark/drop every packet (drop tail behavior)
    - P_m is varied linearly from 0.0 to $\max_p$ as average rises from T_1 to T_2
    - P_a = actual marking probability = $P_m / (1 - \text{count } P_m)$
      - where count increases by one for each packet sent.

- Stateful versus stateless gateways