From van@helios.ee.lbl.gov Mon Apr 30 01:44:05 1990
To: end2end-interest@ISI.EDU
Subject: modified TCP congestion avoidance algorithm
Date: Mon, 30 Apr 90 01:40:59 PDT
From: Van Jacobson <van@helios.ee.lbl.gov>
Status: RO

This is a description of the modified TCP congestion avoidance algorithm that I promised at the teleconference.

BTW, on re-reading, I noticed there were several errors in Lixia's note besides the problem I noted at the teleconference. I don't know whether that's because I mis-communicated the algorithm at dinner (as I recall, I'd had some wine) or because she's convinced that TCP is ultimately irrelevant :). Either way, you will probably be disappointed if you experiment with what's in that note.

First, I should point out once again that there are two completely independent window adjustment algorithms running in the sender: Slow-start is run when the pipe is empty (i.e., when first starting or re-starting after a timeout). Its goal is to get the "ack clock" started so packets will be metered into the network at a reasonable rate. The other algorithm, congestion avoidance, is run any time *but* when (re-)starting and is responsible for estimating the (dynamically varying) pipesize. You will cause yourself, or me, no end of confusion if you lump these separate algorithms (as Lixia's message did).

The modifications described here are only to the congestion avoidance algorithm, not to slow-start, and they are intended to apply to large bandwidth-delay product paths (though they don't do any harm on other paths). Remember that with regular TCP (or with slow-start/c-a TCP), throughput really starts to go to hell when the probability of packet loss is on the order of the bandwidth-delay product. E.g., you might expect a 1% packet loss rate to translate into a 1% lower throughput but for, say, a TCP connection with a 100 packet b-d p. (= window), it results in a 50-75% throughput loss. To make TCP effective on fat pipes, it would be nice if throughput degraded only as function of loss probability rather than as the product of the loss probabilty and the b-d p. (Assuming, of course, that we can do this without sacrificing congestion avoidance.)

These mods do two things: (1) prevent the pipe from going empty after a loss (if the pipe doesn't go empty, you won't have to waste round-trip times re-filling it) and (2) correctly account for the amount of data actually in the pipe (since that's what congestion avoidance is supposed to be estimating and adapting to).

For (1), remember that we use a packet loss as a signal that the pipe is overfull (congested) and that packet loss can be detected one of two different ways: (a) via a retransmit timeout or (b) when some small number (3-4) of consecutive duplicate acks has been received (the "fast retransmit" algorithm). In case (a), the pipe is guaranteed to be empty so we must slow-start. In case (b), if the duplicate ack threshhold is small compared to the bandwidth-delay product, we will detect the loss with the pipe almost full. I.e., given a threshhold of 3 packets and an LBL-MIT bandwidth-delay of around 24KB or 16 packets (assuming 1500 byte MTUs), the pipe is 75% full when fast-retransmit detects a loss (actually, until gateways start doing some sort of congestion control, the pipe is overfull when the loss is detected so *at least* 75% of the packets needed for ack clocking are in transit when fast-retransmit happens). Since the pipe is full, there's no need to slow-start after a fast-retransmit.

For (2), consider what a duplicate ack means: either the network duplicated a packet (i.e., the NSFNet braindead IBM token ring adapters) or the receiver got an out-of-order packet. The usual cause of out-of-order packets at the receiver is a missing packet. I.e., if there are W packets in transit and one is dropped, the receiver will get W-1 out-of-order and (4.3-tahoe TCP will) generate W-1 duplicate acks. If the `consecutive duplicates' threshhold is set high enough, we can reasonably assume that duplicate acks mean dropped packets.

But there's more information in the ack: The receiver can only generate one in response to a packet arrival. I.e., a duplicate ack means that a packet has left the network (it is now cached at the receiver). If the sender is limitted by the congestion window, a packet can now be sent. (The congestion window is a count of how many packets will fit in the pipe. The ack says a packet has left the pipe so a new one can be added to take its place.) To put this another way, say the current congestion window is C (i.e, C packets will fit in the pipe) and D duplicate acks have been received. Then only C-D packets are actually in the pipe and the sender wants to use a window of C+D packets to fill the pipe to its estimated capacity (C+D sent - D received = C in pipe).

So, conceptually, the slow-start/cong.avoid/fast-rexmit changes are:

- The sender's input routine is changed to set `cwnd' to `ssthresh' when the dup ack threshhold is reached. [It used to set cwnd to mss to force a slow-start.] Everything else stays the same.

- The sender's output routine is changed to use an effective window of min(snd_wnd, cwnd + dupacks*mss) [the change is the addition of the `dupacks*mss' term.] `Dupacks' is zero until the rexmit threshhold is reached and zero except when receiving a sequence of duplicate acks.

The actual implementation is slightly different than the above because I wanted to avoid the multiply in the output routine (multiplies are expensive on some risc machines). A diff of the old and new fastrexmit code is attached (your line numbers will vary).

Note that we still do congestion avoidance (i.e., the window is reduced by 50% when we detect the packet loss). But, as long as the receiver's offered window is large enough (it needs to be at most twice the bandwidth-delay product), we continue sending packets (at exactly half the rate we were sending before the loss) even after the loss is detected so the pipe stays full at exactly the level we want and a slow-start isn't necessary.

Some algebra might make this last clear: Say U is the sequence number of the first un-acked packet and we are using a window size of W when packet U is dropped. Packets [U..U+W) are in transit. When the loss is detected, we send packet U and pull the window back to W/2. But in the round-trip time it takes the U retransmit to fill the receiver's hole and an ack to get back, W-1 dup acks will arrive (one for each packet in transit). The window is effectively inflated by one packet for each of these acks so packets [U..U+W/2+W-1) are sent. But we don't re-send packets unless we know they've been lost so the amount actually sent between the loss detection and the recovery ack is U+W/2+W-1 - U+W = W/2-1 which is exactly the amount congestion avoidance allows us to send (if we add in the rexmit of U). The recovery ack is for packet U+W so when the effective window is pulled back from W/2+W-1 to W/2 (which happens because the recovery ack is `new' and sets dupack to zero), we are allowed to send up to packet U+W+W/2 which is exactly the first packet we haven't yet sent.  (I.e., there is no sudden burst of packets as the `hole' is filled.) Also, when sending packets between the loss detection and the recovery ack, we do nothing for the first W/2 dup acks (because they only allow us to send packets we've already sent) and the bottleneck gateway is given W/2 packet times to clean out its backlog. Thus when we start sending our W/2-1 new packets, the bottleneck queue is as empty as it can be.

[I don't know if you can get the flavor of what happens from this description -- it's hard to see without a picture. But I was delighted by how beautifully it worked -- it was like watching the innards of an engine when all the separate motions of crank, pistons and valves suddenly fit together and everything appears in exactly the right place at just the right time.]

Also note that this algorithm interoperates with old tcp's: Most pre-tahoe tcp's don't generate the dup acks on out-of-order packets. If we don't get the dup acks, fast retransmit never fires and the window is never inflated so everything happens in the old way (via timeouts). Everything works just as it did without the new algorithm (and just as slow).

If you want to simulate this, the intended environment is:

- large bandwidth-delay product (say 20 or more packets)

- receiver advertising window of two b-d p (or, equivalently, advertised window of the unloaded b-d p but two or more connections simultaneously sharing the path).

- average loss rate (from congestion or other source) less than one lost packet per round-trip-time per active connection. (The algorithm works at higher loss rate but the TCP selective ack option has to be implemented otherwise the pipe will go empty waiting to fill the second hole and throughput will once again degrade at the product of the loss rate and b-d p. With selective ack, throughput is insensitive to b-d p at any loss rate.)

And, of course, we should always remember that good engineering practise suggests a b-d p worth of buffer at each bottleneck -- less buffer and your simulation will exhibit the interesting pathologies of a poorly engineered network but will probably tell you little about the workings of the algorithm (unless the algorithm misbehaves badly under these conditions but my simulations and measurements say that it doesn't). In these days of $100/megabyte memory, I dearly hope that this particular example of bad engineering is of historical interest only.

```
 - Van

-----------------
*** /tmp/,RCSt1a26717   Mon Apr 30 01:35:17 1990
--- tcp_input.c Mon Apr 30 01:33:30 1990
***************
*** 834,850 ****
                                 * Kludge snd_nxt & the congestion
                                 * window so we send only this one
!                                * packet.  If this packet fills the
!                                * only hole in the receiver's seq.
!                                * space, the next real ack will fully
!                                * open our window.  This means we
!                                * have to do the usual slow-start to
!                                * not overwhelm an intermediate gateway
!                                * with a burst of packets.  Leave
!                                * here with the congestion window set
!                                * to allow 2 packets on the next real
!                                * ack and the exp-to-linear thresh
!                                * set for half the current window
!                                * size (since we know we're losing at
!                                * the current window size).
                                 */
                                if (tp->t_timer[TCPT_REXMT] == 0 ||
--- 834,850 ----
                                 * Kludge snd_nxt & the congestion
                                 * window so we send only this one
!                                * packet.
!                                *
!                                * We know we're losing at the current
!                                * window size so do congestion avoidance
!                                * (set ssthresh to half the current window
!                                * and pull our congestion window back to
!                                * the new ssthresh).
!                                *
!                                * Dup acks mean that packets have left the
!                                * network (they're now cached at the receiver)
!                                * so bump cwnd by the amount in the receiver
!                                * to keep a constant cwnd packets in the
!                                * network.
                                 */
                                if (tp->t_timer[TCPT_REXMT] == 0 ||
***************
*** 853,864 ****
                                else if (++tp->t_dupacks == tcprexmtthresh) {
                                        tcp_seq onxt = tp->snd_nxt;
!                                       u_int win =
!                                           MIN(tp->snd_wnd, tp->snd_cwnd) / 2 /
!                                               tp->t_maxseg;

                                        if (win < 2)
                                                win = 2;
                                        tp->snd_ssthresh = win * tp->t_maxseg;
-
                                        tp->t_timer[TCPT_REXMT] = 0;
                                        tp->t_rtt = 0;
--- 853,864 ----
                                else if (++tp->t_dupacks == tcprexmtthresh) {
                                        tcp_seq onxt = tp->snd_nxt;
!                                       u_int win = MIN(tp->snd_wnd,
!                                                       tp->snd_cwnd);
+                                       win /= tp->t_maxseg;
+                                       win >>= 1;
                                        if (win < 2)
                                                win = 2;
                                        tp->snd_ssthresh = win * tp->t_maxseg;
                                        tp->t_timer[TCPT_REXMT] = 0;
                                        tp->t_rtt = 0;
***************
*** 866,873 ****
                                        tp->snd_cwnd = tp->t_maxseg;
                                        (void) tcp_output(tp);
!
                                        if (SEQ_GT(onxt, tp->snd_nxt))
                                                tp->snd_nxt = onxt;
                                        goto drop;
                                }
                        } else
--- 866,879 ----
                                        tp->snd_cwnd = tp->t_maxseg;
                                        (void) tcp_output(tp);
!                                       tp->snd_cwnd = tp->snd_ssthresh +
!                                                      tp->t_maxseg *
!                                                      tp->t_dupacks;
                                        if (SEQ_GT(onxt, tp->snd_nxt))
                                                tp->snd_nxt = onxt;
                                        goto drop;
+                               } else if (tp->t_dupacks > tcprexmtthresh) {
+                                       tp->snd_cwnd += tp->t_maxseg;
+                                       (void) tcp_output(tp);
+                                       goto drop;
                                }
                        } else
***************
*** 874,877 ****
--- 880,890 ----
                                tp->t_dupacks = 0;
                        break;
+               }
+               if (tp->t_dupacks) {
+                       /*
+                        * the congestion window was inflated to account for
+                        * the other side's cached packets - retract it.
+                        */
+                       tp->snd_cwnd = tp->snd_ssthresh;
                }
                tp->t_dupacks = 0;
*** /tmp/,RCSt1a26725   Mon Apr 30 01:35:23 1990
--- tcp_timer.c Mon Apr 30 00:36:29 1990
***************
*** 223,226 ****
--- 223,227 ----
                tp->snd_cwnd = tp->t_maxseg;
                tp->snd_ssthresh = win * tp->t_maxseg;
+               tp->t_dupacks = 0;
                }
                (void) tcp_output(tp);

From van@helios.ee.lbl.gov Mon Apr 30 10:37:36 1990
To: end2end-interest@ISI.EDU
Subject: modified TCP congestion avoidance algorithm (correction)
Date: Mon, 30 Apr 90 10:36:12 PDT
From: Van Jacobson <van@helios.ee.lbl.gov>
```

Status: RO

I shouldn't make last minute 'fixes'. The code I sent out last night had a small error:

```
*** t.c Mon Apr 30 10:28:52 1990
--- tcp_input.c Mon Apr 30 10:30:41 1990
***************
*** 885,893 ****
                         * the congestion window was inflated to account for
                         * the other side's cached packets - retract it.
                         */
!                       tp->snd_cwnd = tp->snd_ssthresh;
                }
-               tp->t_dupacks = 0;
                if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
                        tcpstat.tcps_rcvacktoomuch++;
                        goto dropafterack;
--- 885,894 ----
                         * the congestion window was inflated to account for
                         * the other side's cached packets - retract it.
                         */
!                       if (tp->snd_cwnd > tp->snd_ssthresh)
!                               tp->snd_cwnd = tp->snd_ssthresh;
!                       tp->t_dupacks = 0;
                }
                if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
                        tcpstat.tcps_rcvacktoomuch++;
                        goto dropafterack;
```