Draft

[Jai86b] Raj Jain. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications*, SAC-4(7), October 1986.

[JRC87] Raj Jain, K.K. Ramakrishnan, and Dah-Ming Chiu. Congestion avoidance in computer networks with a connectionless network layer. Technical Report DEC-TR-506, Digital Equipment Corporation, August 1987.

[Kle76] Leonard Kleinrock. *Queueing Systems*, volume II. John Wiley & Sons, 1976.

[Kli87] Charley Kline. Supercomputers on the Internet: A case study. In *Proceedings of SIGCOMM '87*. ACM, August 1987.

[KP87] Phil Karn and Craig Partridge. Estimating round-trip times in reliable transport protocols. In *Proceedings of SIGCOMM '87*. ACM, August 1987.

[Lue79] David G. Luenberger. *Introduction to Dynamic Systems*. John Wiley & Sons, 1979.

[Mil83] David Mills. *Internet Delay Experiments*. ARPANET Working Group Requests for Comments, Network Information Center (NIC), SRI International, Menlo Park, Calif., December 1983. RFC-889.

[NBS83] Institute for Computer Science and Technology, National Bureau of Standards. *Specification of a Transport Protocol for Computer Communications, Volume 5: Guidance for the Implementor*, January 1983.

[Pos81] Jon Postel, editor. *Transmission Control Protocol Specification*. ARPANET Working Group Requests for Comments, Network Information Center (NIC), SRI International, Menlo Park, Calif., September 1981. RFC-793.

[PP87] W. Prue and Jon Postel. *Something A Host Could Do with Source Quench*. ARPANET Working Group Requests for Comments, Network Information Center (NIC), SRI International, Menlo Park, Calif., July 1987. RFC-1016.

[Zha86] Lixia Zhang. Why TCP timers don't work well. In *Proceedings of SIGCOMM '86*. ACM, August 1986.

Draft

queue since the bottleneck would be running at 100% utilization with no excess bandwidth available to dissipate the queue. If a packet is tossed, some sender will shut up for two rtt, exactly the time needed to empty the queue. If that sender restarts with the correct window size the queue won't reform. Thus the delay has been reduced to minimum without the system losing any bottleneck bandwidth.

The 1-packet increase has less justification that the 0.5 decrease. In fact, it's almost certainly too large. If the algorithm converges to a window size of w, there are $O(w^2)$ packets between drops with an additive increase policy. We were shooting for an average drop rate of $< 1\%$ and found that on the Arpanet (the worst case of the four networks we tested), windows converged to 8–12 packets. This yields 1 packet increments for a 1% average drop rate.

But, since we've done nothing in the gateways, the window we converge to is the maximum the gateway can accept without dropping packets. I.e., in the terms of [JRC87], we are just to the left of the cliff rather than just to the right of the knee. If the gateways are now fixed so they start dropping packets when the queue gets pushed past the knee, our increment will be much too aggressive and we'll have to drop it by about a factor of four (since our measurements on an unloaded Arpanet place its 'pipe size' at 4–5 packets). Mike and I have talked about a second order control loop to adaptively determine the appropriate increment to use for a path. It appears trivial to implement such a loop (since changing the increment affects the frequency of oscillation but not the mean window size, the loop would affect rate of convergence but not convergence and (first-order) stability). But we're leaving second order stuff until we've spent some time on the first order part of the algorithm for the gateways.

References

[BC87] R. Borrelli and C. Coleman. *Differential Equations*. Prentice-Hall Inc., 1987.

[Edg83] Stephen William Edge. An adaptive timeout algorithm for retransmission across a packet switching network. In *SIGCOMM '83*. ACM, 1983.

[Fel71] William Feller. *Probability Theory and its Applications*, volume II. John Wiley & Sons, second edition, 1971.

[ISO84] International Organization for Standardization. *ISO International Standard 8073, Information Processing Systems — Open Systems Interconnection — Transport Protocol Specification*, 1984.

[Jai86a] Raj Jain. Divergence of timeout algorithms for packet retransmissions. In *Proceedings Fifth Annual International Phoenix Conference on Computers and Communications*, Scottsdale, AZ, March 1986.

Draft

B The combined slow-start with congestion avoidance algorithm

The sender keeps two state variables for congestion control: a congestion window, *cwnd*, and a threshold size, *ssthresh*, to switch between the two algorithms. The sender's output routine sends the minimum of cwnd and the window advertised by the receiver. The rest of the congestion control sender code looks like: On a timeout, record half the current window size as *ssthresh* (this is the multiplicative decrease part of the congestion avoidance algorithm), then set *cwnd* to 1 packet (this initiates slow-start). When new data is acked, the sender does

```
if (cwnd < ssthresh)  // if we're still doing slowstart
    cwnd += 1         // open the window exponentially
else
    cwnd += 1/cwnd    // otherwise do the Congestion
                      // Avoidance increment-by-1
```

Thus slow-start opens the window quickly to what congestion avoidance thinks is a safe operating point (half the window that got us into trouble), then congestion avoidance takes over and slowly increases the window size to probe for more bandwidth becoming available on the path.

C Window Adjustment Policy

One reason for using $\frac{1}{2}$ as a the decrease term, as opposed to the $\frac{7}{8}$ in [JRC87], was the following hand-waving: When a packet is dropped, you're either starting (or restarting after a drop) or steady-state sending. If you're starting, you know that half the current window size 'worked', i.e., that a window's worth of packets were exchanged with no drops (slowstart guarantees this). Thus on congestion you set the window to the largest size that you know works then slowly increase the size. If the connection is steady-state running and a packet is dropped, it's probably because a new connection started up and took some of your bandwidth. We usually run our nets with $\rho \leq 0.5$ so it's probable that there are now exactly two conversations sharing the bandwidth. I.e., you should reduce your window by half because the bandwidth available to you has been reduced to half. And, if there are more than two conversations sharing the bandwidth, halving your window is conservative — and being conservative at high traffic intensities is probably wise.

Although a factor of two change in window size seems like it would exact a large performance penalty, in system terms the cost isn't that high: Currently packets are dropped only when a large queue has formed. If we had the ISO-8473 bit to force senders to reduce their windows, we'd still be stuck with the

Draft

$$2^n D \leftarrow 2^n D + (|Err| - D)$$

To minimize round-off error, we probably want to keep the scaled versions of A and D rather than the unscaled versions. If we pick $g = .125 = \frac{1}{8}$ (close to the .1 that rfc793 suggests) and express the above in C:

```
M -= (A >> 3);     // Compute Err
A += M;            // Update the scaled average
if (M < 0)         // Compute abs(Err)
      M = -M;
M -= (D >> 3);     // Compute Mean Deviation error
D += M;            // Update the scaled mean dev.
```

It's not necessary to use the same gain for A and D. To force the timer to go up quickly in response to changes in the rtt, it's a good idea to give D a larger gain. In particular, because of window–delay mismatch, there are often rtt artifacts at integer multiples of the window size (e.g., see packets 10–50 of figure 5). To filter these, one would like $1/g$ in the A estimator to be $\geq$ the window size (in packets) and $1/g$ in the D estimator to be $<$ the window size.[10] [11]

Using a gain of .25 on the deviation and computing the retransmit timer, rto, as $A + 2D$, our final timer code looks like:

```
M -= (A >> 3);
A += M;
if (M < 0)
      M = -M;
M -= (D >> 2);
D += M;
rto = ((A >> 2) + D) >> 1;
```

Note that A and D are added before the final shift. In general, this will correctly round rto: Because of the A truncation when we compute $M - A$, the scaled A will converge to the true mean rounded up to the next tick. Likewise with D. Thus, on the average, there is half a tick of bias in each. We want the rto computation rounded by half a tick and we need to add one tick to account for the fact that our sends are phased randomly with respect to the clock. So, the 1.5 tick bias contribution we get from $A + 2D$ equals the desired half tick rounding plus one tick phase correction.

[10] Note that these window effects are largely due to characteristics of the Arpa/Milnet subnet. In general, window effects on the timer are at most a second-order consideration and depend a great deal on the underlying network. E.g., if one were using the Wideband with a 256 packet window, 1/256 would not be a good gain for A (1/16 might be).

[11] Although it may not be obvious, the absolute value in the calculation of D introduces an asymmetry in the timer: Because D has the same sign as an increase and the opposite sign of a decrease, more gain in D makes the timer go up fast and come down slow, 'automatically' giving the behavior Dave Mills suggests in [Mil83]. E.g., see the region between packets 50 and 80 in figure 6.

Draft

If we want some measure of the variation in M, say to compute a good value for the tcp retransmit timer, there are lots of choices. Statisticians like variance because it has some nice mathematical properties. But variance requires squaring $(M - A)$ so an estimator for it will contain a multiply with a danger of integer overflow. Also, most applications will want variation in the same units as A and M, so we'll be forced to take the square root of the variance to use it (i.e., at least a divide, multiply and two adds).

A variation measure that's easy to compute, and has a nice, intuitive justification, is the mean prediction error or mean deviation. This is just the average of $|M - A|$. Intuitively, this is an estimate of how badly we've blown our recent predictions (which seems like just the thing we'd want to set up a retransmit timer). Since

$$mdev^2 = \left(\sum |M - A|\right)^2 \geq \sum (M - A)^2 = \sigma^2$$

mean deviation is a more 'conservative' estimate of variation than standard deviation.

If for some reason you want standard deviation, there's usually a simple relation between mdev and sdev. E.g., if the prediction errors are normally distributed, $mdev = \sqrt{\pi/2}\, sdev$. For most common distributions the factor to go from sdev to mdev is near one ($\sqrt{\pi/2} \approx 1.25$). I.e., mdev is a good approximation of sdev and is much easier to compute.

A.2 Practice

So let's code estimators for average A and mean deviation D given measurement M. Both estimators compute means so we get two instances of the rfc793 algorithm:

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + g(|Err| - D)$$

If we want to do the above fast, it should probably be done in integer arithmetic. But the expressions contain fractions ($g < 1$) so we'll have to do some scaling to keep everything integer. If we chose g so that $1/g$ is an integer, the scaling is easy. A particularly good choice for g is a reciprocal power of 2 (i.e., $g = 1/2^n$ for some n). Multiplying through by $1/g$ gives

$$2^n A \leftarrow 2^n A + Err$$

Draft

A A fast algorithm for rtt mean and variance

A.1 Theory

You're probably familiar with the rfc793 algorithm for estimating the mean round trip time. It is the simplest example of a class of estimators called 'recursive prediction error' or 'stochastic gradient' algorithms. In the past 20 years these algorithms have revolutionized estimation and control theory and it's probably worth while to look at the rfc793 estimator in some detail.

Given a new measurement, M, of the rtt (round-trip-time), tcp updates an estimate of the average rtt, A, by

$$A \leftarrow (1 - g)A + gM$$

where g is a 'gain' ($0 < g < 1$) that should be related to the signal-to-noise ratio (or, equivalently, variance) of M. This makes a bit more sense (and computes faster) if we rearrange and collect terms multiplied by g to get

$$A \leftarrow A + g(M - A)$$

We can think of A as a prediction of the next measurement. So $M - A$ is the error in that prediction and the expression above says we make a new prediction based on the old prediction plus some fraction of the prediction error. The prediction error is the sum of two components: (1) error due to 'noise' in the measurement (i.e., random, unpredictable effects like fluctuations in competing traffic) and (2) error due to a bad choice of A. Calling the random error RE and the predictor error PE,

$$A \leftarrow A + gRE + gPE$$

The gPE term gives A a kick in the right direction while the gRE term gives it a kick in a random direction. Over a number of samples, the random kicks cancel each other out so this algorithm tends to converge to the correct average. But g represents a compromise: We want a large g to get mileage out of the PE term but a small g to minimize the damage from the RE term. Since the PE terms move A toward the real average no matter what value we use for g, it's almost always better to use a gain that's too small rather than one that's too large. Typical gain choices are 0.1–0.2 (though it's always a good idea to take long look at your raw data before picking a gain).

It's probably obvious that A will oscillate randomly around the true average and the standard deviation of A will be something like g sdev(M). Also that A converges to the true average exponentially with time constant $1/g$. So making g smaller gives a stabler A at the expense of taking a much longer time to get to the true average.

Draft

allocation. Thus, we view doing the gateway 'congestion detection' algorithm as the next big step.

The goal of this algorithm to send a signal to the endnodes as early as possible, but not so early that the gateway ends up starved for traffic. The way we anticipate doing the algorithm, gateway 'self protection' from a mis-behaving host will fall-out for free (that host will simply have most of its packets dropped as the gateway trys to tell it that it's using more than its fair share). Thus, like the endnode algorithm, the gateway algorithm should reduce congestion even if no endnode is modified to do congestion avoidance. And nodes that do implement congestion avoidance will get their fair share of bandwidth and a minimum number of packet drops.

Since congestion grows exponentially, detecting it early is important — If it's detected early, small adjustments to the senders' windows will cure it. Otherwise massive adjustments will be necessary to give the net enough spare capacity to pump out the backlog. But, given the bursty nature of traffic, reliable detection is a non-trivial problem. [JRC87] proposes a scheme based on averaging between queue regeneration points. This should yield good burst filtering but we think it might have convergence problems under high load and/or significant second-order dynamics in the traffic.[9] We plan to use some of our earlier work on ARMAX models for round-trip-time/queue length prediction as the basis of the detection. Preliminary results suggest that this approach works well at high load, is immune to second-order effects in the traffic and is cheap enough to compute that it wouldn't slow down kilo-packet-per-second gateways.

Acknowledgements

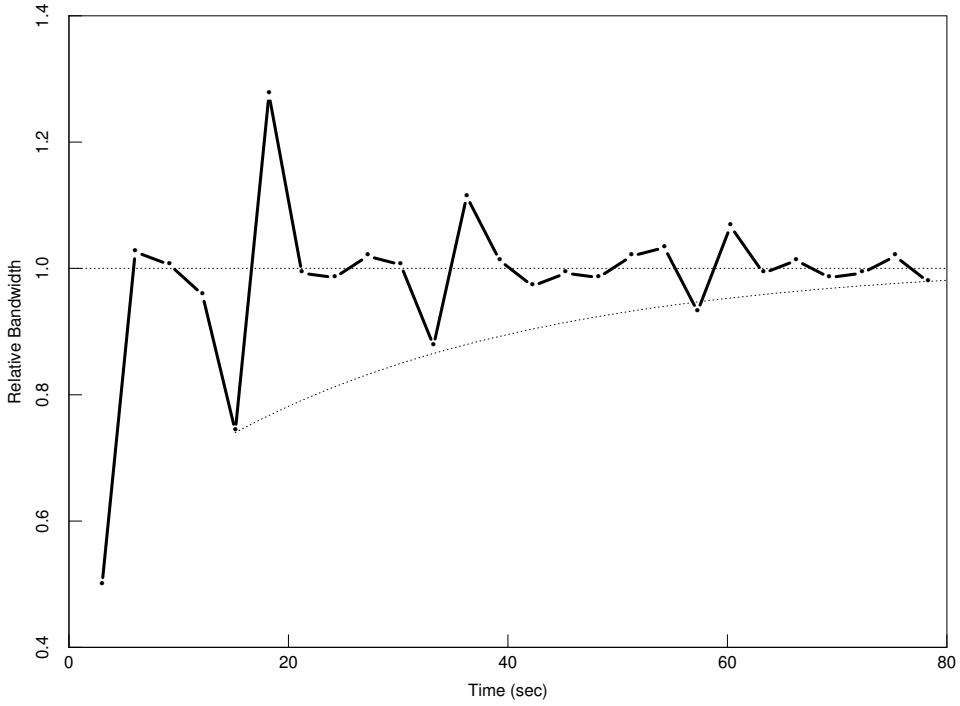
I am very grateful to the members of the Internet Activity Board's End-to-End and Internet-Engineering task forces for this past year's unstinting supply of encouragement, cogent questions and network insights.

I am also deeply in debt to Jeff Mogul of DEC. Without Jeff's interest, patient prodding and way-beyond-the-call-of-duty efforts to help me get a draft submitted before deadline, this paper would never have existed.

[9] The time between regeneration points increases like $(1-\rho)^{-1}$ and the variance of that time like $(1-\rho)^{-3}$ (see [Fel71], chap. VI.9). Thus the congestion detector becomes sluggish as congestion increases and its signal-to-noise ratio decreases dramatically.

Draft

Figure 12: Window adjustment detail



Because of the five second averaging time (needed to smooth out the spikes in the old TCP data), the convergence of the congestion avoidance window policy is difficult to make out in figures 10 and 11. Here we show effective throughput (data acked) for TCPs with congestion control, averaged over a three second interval.

When the gateway drops a packet, the sender will keep sending until it fills the window, then stop until the timeout for the dropped packet. Since the receiver cannot ack any of the data beyond the dropped packet, on this plot we'd expect to see a negative-going spike whose amplitude equals the sender's window size (minus one packet). If the retransmit happens in the next interval (the intervals were chosen to match the retransmit timeout), we'd expect to see a positive-going spike of the same amplitude when receiver acks its cached data. Thus the height of these spikes is a direct measure of the sender's window size.

The data clearly shows three of these events (at 15, 33 and 57 seconds) and the window size appears to be decreasing exponentially. The dotted line shows a least squares fit to the six window size measurements we get from these three events. The time constant of the fit exponential was 28 sec. (The long time constant is due to lack of a congestion avoidance algorithm in the gateway. With the gateway algorithm, the time constant should be around 4 sec.)

Draft

Figure 11: Effective bandwidth of old and new TCPs

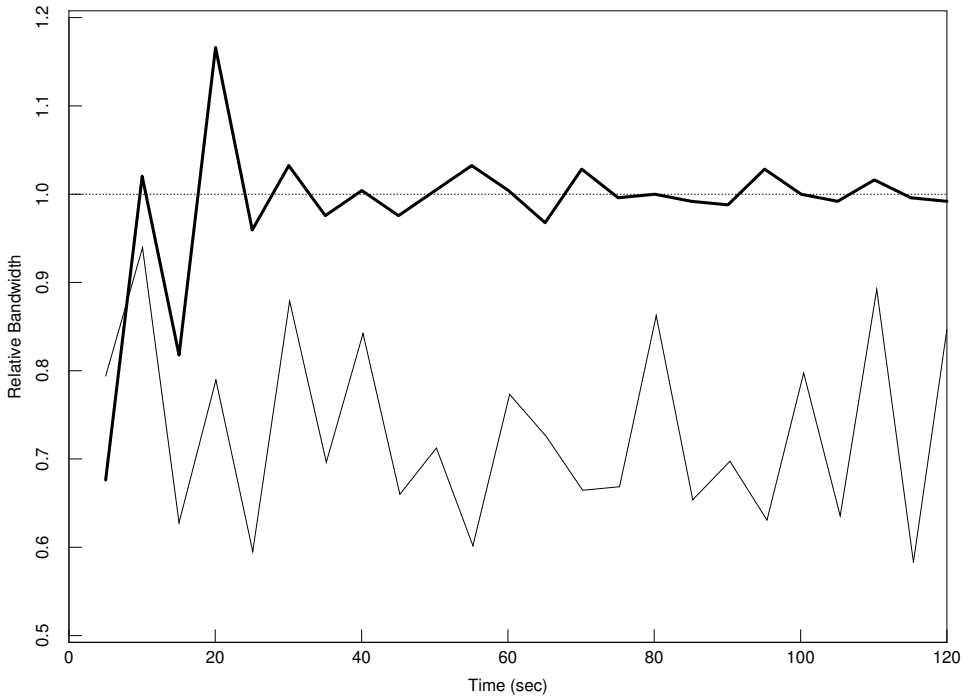


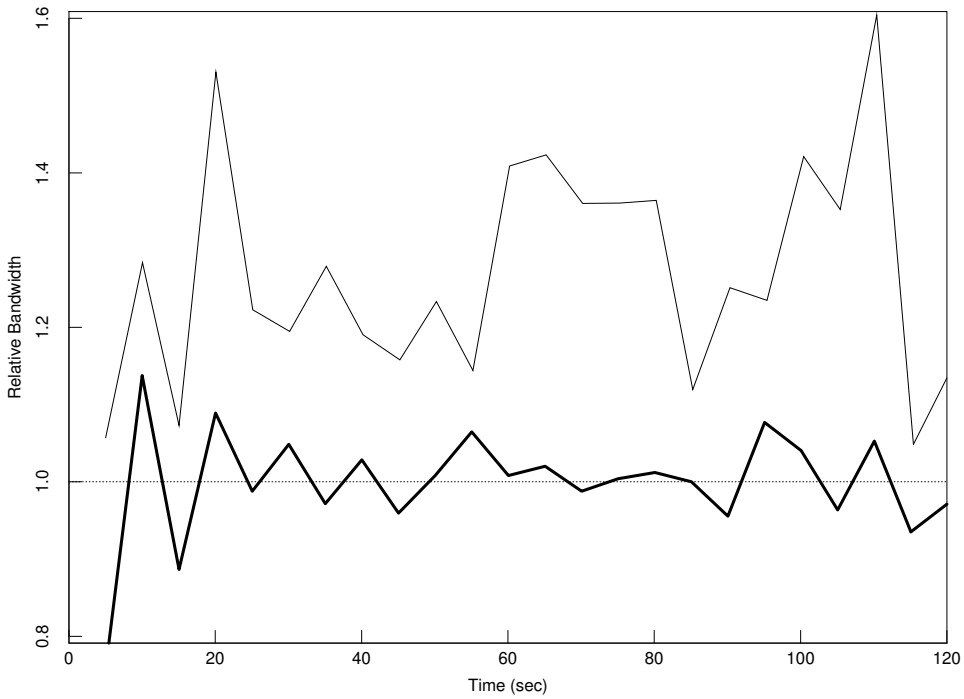
Figure 10 showed the old TCPs were using 25% more than the bottleneck link bandwidth. Thus, once the bottleneck queue filled, 25% of the the senders' packets were being discarded. If the discards, and only the discards, were retransmitted, the senders would have received the full 25KBs link bandwidth (i.e., their behavior would have been anti-social but not self-destructive). But fig. 8 noted that around 25% of the link bandwidth was unaccounted for.

Here we average the total amount of data acked per five second interval. (This gives the *effective* or *delivered* bandwidth of the link.) The thin line is once again the old TCPs. Note that only 75% of the link bandwidth is being used for data (the remainder must have been used by retransmissions of packets that didn't need to be retransmitted).

The thick line shows delivered bandwidth for the new TCPs. There is the same slow-start and turn-on transient followed by a long period of operation right at the link bandwidth. (Notice that the amplitude of the oscillations in the 40 to 120 sec. region is smaller than the corresponding region in fig. 10. This suggests that the bottleneck queue was buffering the wire from send-side transients.)

Draft

Figure 10: Total bandwidth used by old and new TCPs

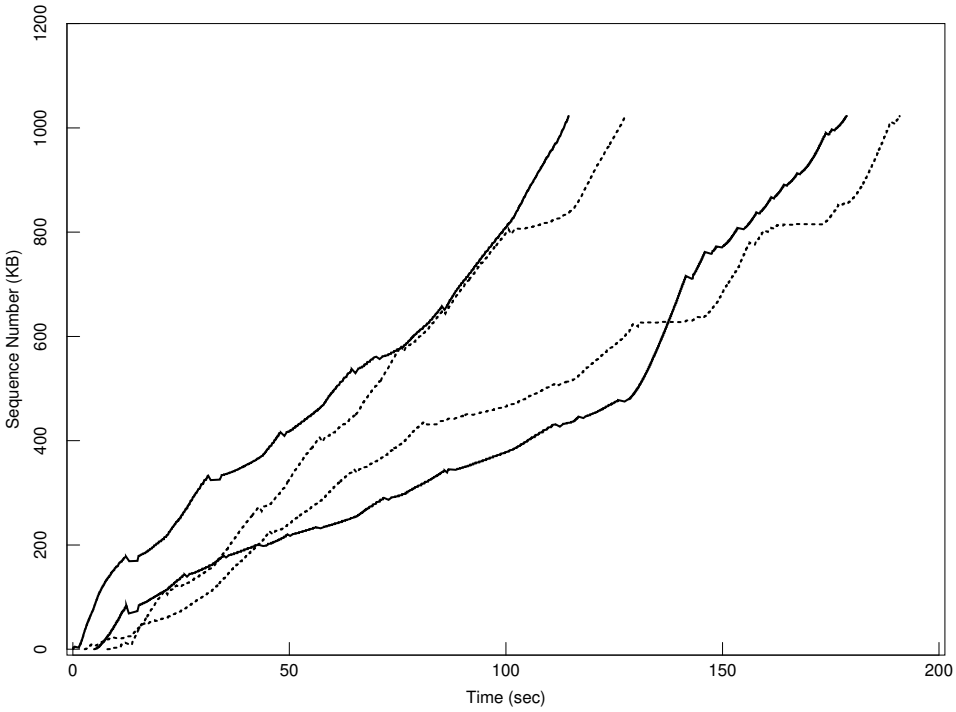


The thin line shows the total bandwidth used by the four senders without congestion avoidance (fig. 8), averaged over 5 second intervals and normalized to the 25KBs link bandwidth. Note that the senders send, on the average, 25% more than will fit in the wire.

The thick line is the same data for the senders with congestion avoidance (fig. 9). The first 5 sec. interval is low (because of the slow-start), then there is about 20 sec. of damped oscillation as the congestion control 'regulator' for each tcp finds the correct window size. The remaining time the senders run at the wire bandwidth. (The activity around 110 sec. is a bandwidth 're-negotiation' due to connection one shutting down. The activity around 80 sec. is a reflection of the 'flat spot' in fig. 9 where most of conversation two's bandwidth suddenly gets handed to conversations three and four — a colleague and I find this kind of 'punctuated equilibrium' fascinating and hope to investigate its dynamics in a future paper.)

Draft

Figure 9: Multiple, simultaneous TCPs with congestion avoidance



Trace data from four simultaneous TCP conversations using congestion avoidance over the paths shown in figure 7.

89 of 8281 packets sent were retransmissions (i.e., 1% of the data packets had to be retransmitted).

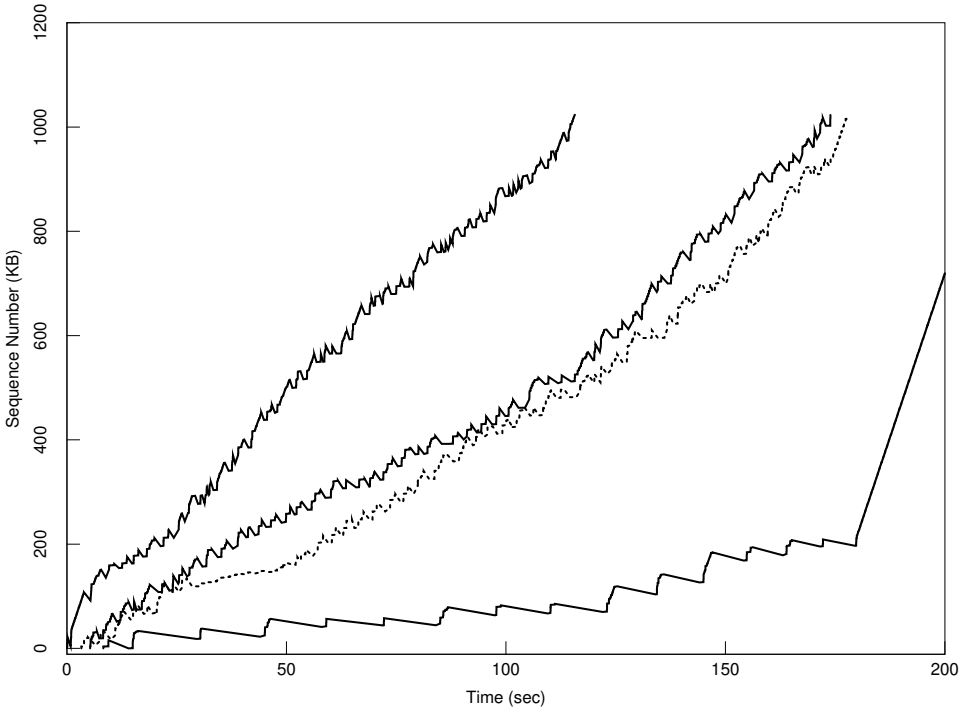
Two of the conversations got 8 KBs and two got 4.5 KBs (i.e., all the link bandwidth is accounted for — see fig. 11). The difference between the high and low bandwidth senders was due to the receivers. The 4.5 KBs senders were talking to 4.3BSD receivers which would delay an ack until 35% of the window was filled or 200ms had passed (i.e., an ack was delayed for 5–7 packets on the average). This meant the sender would deliver bursts of 5–7 packets on each ack.

The 8 KBs senders were talking to 4.3$^+$ BSD receivers which would delay an ack for at most one packet (the author doesn't believe that delayed acks are a particularly good idea). I.e., the sender would deliver bursts of at most two packets.

The probability of loss increases very rapidly with burst size so senders talking to old-style receivers saw three times the loss rate (1.8% vs. 0.5%). The higher loss rate meant more time spent in retransmit wait and, because of the congestion avoidance, smaller average window sizes.

Draft

Figure 8: Multiple, simultaneous TCPs with no congestion avoidance



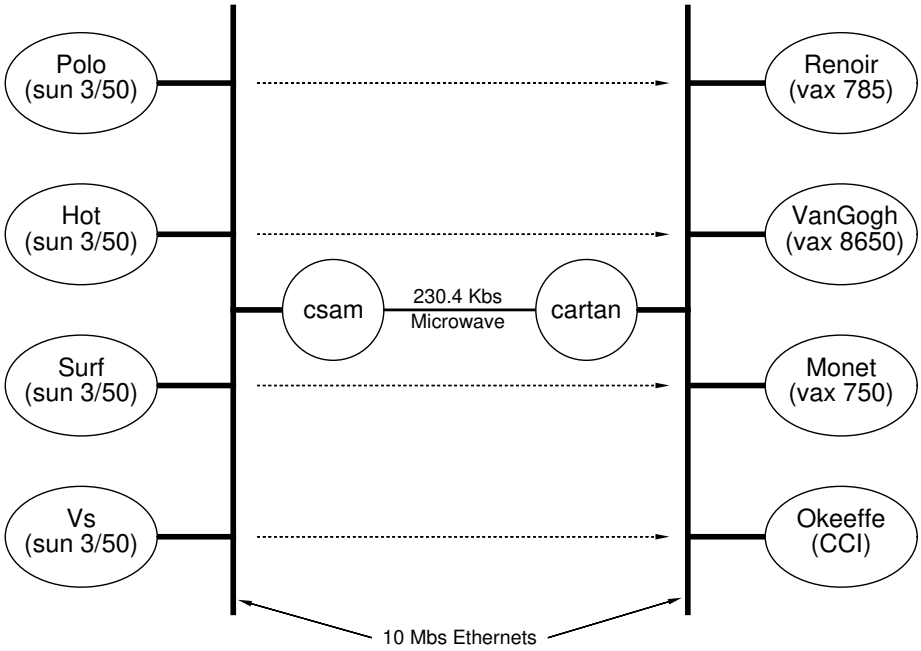
Trace data from four simultaneous TCP conversations without congestion avoidance over the paths shown in figure 7.

4,000 of 11,000 packets sent were retransmissions (i.e., half the data packets were retransmitted).

Since the link data bandwidth is 25 KB/sec, each of the four conversations should have received 6 KBs. Instead, one conversation got 8 KBs, two got 5 KBs, one got 0.5 KBs and 6 KBs has vanished.

Draft

Figure 7: Multiple conversation test setup



Test setup to examine the interaction of multiple, simultaneous TCP conversations sharing a bottleneck link. 1 MByte transfers (2048 512-data-byte packets) were initiated simultaneously (actually, 3 seconds apart) from four machines at LBL to four machines at UCB, one conversation per machine pair (the dotted lines above show the pairing). All traffic went via a 230.4 Kbs link connecting IP router **csam** at LBL to IP router **cartan** at UCB. (Actually, the topology was more complicated and an average of five IP hops separated the sender and receiver. The other hops were via either 10 MB Ethernets or 80 MB Proteon rings and thus the factor of 40 bandwidth difference between the microwave link and the local net(s) dominated the observed behavior.)

The microwave link queue at **csam** can hold up to 50 packets. Each connection was given a window of 16 KB (32 512-byte packets). Thus any two connections could overflow the available buffering and the four connections exceeded the bottleneck queue capacity by 260%.

Draft

to the above. But, because both congestion avoidance and slow-start are triggered by a timeout and both manipulate the congestion window, people frequently confuse the two. They are actually independent algorithms with completely different objectives. To emphasize the difference, the two algorithms have been presented separately even though in practise they should be implemented together. Appendix B describes a combined slow-start/congestion avoidance algorithm.[8]

Figures 7 through 12 show the behavior of TCP connections with and without congestion avoidance. Although the test conditions (e.g., 16KB windows) were deliberately chosen to stimulate congestion, the test scenario isn't far from common practice. E.g., the Arpanet IMP end-to-end protocol allows at most eight packets in transit between any pair of gateways. The default 4.3BSD window size is eight packets (4KB). Thus simultaneous conversations between any two hosts at Berkeley and any two hosts at MIT would exceed the buffer capacity of the UCB–MIT IMP path and would lead (actually, *did* lead) to the behavior shown in the following figures.

4 Future work: the gateway side of congestion control

While algorithms at the transport endpoints can insure the network capacity isn't exceeded, they cannot insure fair sharing of that capacity. Only in gateways, at the convergence of flows, is there enough information to control sharing and fair

[8] We have also developed a rate-based variant of the congestion avoidance algorithm to apply to connectionless traffic (e.g., domain server queries, RPC requests). Remembering that the goal of the increase and decrease policies is bandwidth adjustment, and that 'time' (the controlled parameter in a rate-based scheme) appears in the denominator of bandwidth, the algorithm follows immediately: The multiplicative decrease remains a multiplicative decrease (e.g., double the interval between packets). But subtracting a constant amount from interval does *not* result in an additive increase in bandwidth. (This mistake has been made frequently, e.g., see [Kli87] and [PP87]. For interval I and decrement c, the bandwidth change of a decrease-interval-by-constant policy is

$$\frac{1}{I} \rightarrow \frac{1}{I - c}$$

a non-linear, and destablizing, bandwidth increase.)

An update policy that does result in a linear increase of bandwidth over time is

$$I_i = \frac{\alpha I_{i-1}}{\alpha + I_{i-1}}$$

where I_i is the interval between sends when the ith packet is sent and α is the desired rate of increase in packets per packet/sec.

We have a cooperative project with Sun Microsystems to prototype RPC dynamic congestion control algorithms based on the above. We plan to use NFS as a test-bed (since NFS is known to have congestion problems yet it would be very desirable to have it work over the same range of networks as TCP).

Draft

bandwidth is costly. But an exponential, almost regardless of its time constant, increases so quickly that large overestimates are inevitable.

Without justification, I'll simply state that the best increase policy is to make small, constant changes to the window size:

$$\text{On no congestion:} \quad W_i = W_{i-1} + u \qquad (u \ll \text{delay-bandwidth product})$$

I.e., an additive increase policy. This is the policy suggested in [JRC87] and the policy we've implemented in TCP. The only difference between the two implementations is the choice of constants for d and u. We used 0.5 and 1 for reasons that are partially explained in appendix C. A more complete analysis is in yet another in-progress paper.

The preceding has probably made the congestion control algorithm sound hairy but it's not. Like slow-start, it turns out to be three lines of code:

- On any timeout, set cwnd to half the current window size (this is the multiplicative decrease).
- On each ack for new data, increase cwnd by 1/cwnd (this is the additive increase).[7]
- When sending, send the minimum of the receiver's advertised window and cwnd.

Note that this algorithm is *only congestion avoidance*, it doesn't include the previously described slow-start. Since the packet loss that signals congestion will result in a re-start, it will almost certainly be necessary to slow-start in addition

---

will scale like $(1-\rho)^{-2}$ ([Kle76], chap. 2 is an excellent discussion of this). Since at equilibrium our transport connection 'wants' to run the bottleneck link at 100% ($\rho = 1$), we have to be sure that during the non-equilibrium window adjustment, our control policy allows the gateway enough bandwidth and time to dissipate queues that inevitably form due to path testing and statistical traffic fluctuations. By an argument similar to the one used to show exponential timer backoff is necessary, it's possible to show that an exponential (multiplicative) window increase policy will be 'faster' than the dissipation time for some traffic mix and, thus, leads to an unbounded growth of the bottleneck queue.

[7] This increment rule may be slightly less than obvious. We want to increase the window by at most one packet over a time interval of length R (the round trip time). To make the algorithm 'self-clocked', and to remove some nasty end-case truncation effects, it's better to increment by a small amount on each ack rather than by a large amount at the end of the interval. (Assuming, of course, that the sender has effective 'silly window' code and doesn't attempt to send packet fragments because of the fractionally sized window.) A window of size cwnd packets will generate at most cwnd acks in one R. Thus an increment of 1/cwnd per ack will increase the window by at most one packet in one R. In TCP, windows and packet sizes are in bytes so the increment translates to maxseg*maxseg/cwnd where maxseg is the max. segment size and cwnd is the congestion window size in bytes.

Draft

(These are the first two terms in a Taylor series expansion of $L(t)$. As pragmatists, we extend the original model just enough to explain the new behavior when we find that taking just the first term is inadequate. There is reason to believe one might eventually need a three term, second order model, but not until the Internet has grown to several times its current size.)

When the network is congested, the γ term must be large and the load (queues) will start increasing exponentially.[5] The only way to stabilize this system is if the traffic sources throttle back at least as fast as the queues are growing. Since a source controls load in a window-based protocol by adjusting the size of the window, W, we end up with the sender policy

$$\text{On congestion:} \quad W_i = dW_{i-1} \qquad (d < 1)$$

I.e., a multiplicative decrease of the window size. (This will turn into an exponential decrease over time if the congestion persists.)

If there's no congestion, the γ term must be near zero and the network load is approximately constant. The network tells you (via a dropped packet) when the demand is excessive but says nothing if you are using less than your fair share of the available bandwidth (since the network is stateless, it cannot know this). Thus you have to try to increase the bandwidth you are using to find out the current limit. E.g., you could have been sharing the path with someone else and converged to a window that gives you each half the available bandwidth. If she shuts down, 50% of the bandwidth will get wasted unless you make some effort to increase your window size. What should the increase policy be?

The first thought is to use a symmetric, multiplicative increase, possibly with a longer time constant. E.g., $W_i = bW_{i-1}$, $1 < b \leq 1/d$. This is a mistake. The result will oscillate wildly and, on the average, deliver poor throughput. There is an analytic reason for this but it's tedious to derive. It has to do with that fact that it is easy to drive the net into saturation but hard for the net to recover (what [Kle76], chap. 2.1, calls the *rush-hour effect*).[6] Thus overestimating the available

[5] I.e., the system behaves like $L_i \approx \gamma L_{i-1}$, a difference equation with the solution

$$L_n = \gamma^n L_0$$

which goes exponentially to infinity for any $\gamma > 1$.

[6] Taking another look at figure 1, note that the 'pipesize' is 16 packets, 8 in each path, but the sender is using a window of 22 packets. The six excess packets will form a queue at the entry to the bottleneck and *that queue cannot shrink*, even though the sender carefully clocks out packets at the bottleneck link rate. This stable queue is another, unfortunate, aspect of conservation: The only way the queue could shrink would be if the gateway could move packets into the skinny pipe faster than the sender is dumping packets into the fat pipe. But the system always tunes itself so every time the gateway pulls a packet off the front of its queue, the sender lays a new packet on the end.

A gateway needs excess output capacity (i.e., $\rho < 1$) to dissipate a queue and the clearing time

Draft

suggests that if we have a system that is unstable (a network subject to random load shocks and prone to congestive collapse), the way to stabilize it is to add some exponential damping (read: exponential timer backoff) to its primary excitation (read: senders, traffic sources).

3 Adapting to the path: congestion avoidance

Once the timers are in good shape, it is possible to state with some confidence that a timeout means a lost packet and not a busted timer. At this point something can be done about (3). Packets get lost for two reasons: either they are damaged in transit or the network is congested and somewhere on the path there was insufficient buffer capacity. On most of the network paths we use, loss due to damage is rare ($\ll 1\%$) so it is very probable that a packet loss is due to congestion in the network.

Say we try to develop a 'congestion avoidance' strategy like the one proposed in [JRC87]. It must have two components: The network must be able to signal the transport endpoints that congestion is occurring (or about to occur). And the endpoints must have a policy that decreases utilization if this signal is received and increases utilization if the signal isn't received.

If packet loss is (almost) always due to congestion and if a timeout is (almost) always due to a lost packet, we have a good candidate for the 'network is congested' signal. Particularly since this signal is delivered automatically by all existing networks, without special modification (as opposed to, say, ISO-8473 which requires a new bit in the packet headers and a mod to *all* existing gateways to set this bit).

The other part of a congestion avoidance strategy, the endnode action, is almost identical in the DEC/ISO scheme and our TCP. This is not an accident (we copied Jain's scheme after hearing his presentation at the Boston IETF and realizing that the scheme was, in a sense, universal): The endnode strategy follows directly from a first-order time-series model of the network. Say we measure network 'load' by average queue length over fixed intervals of some appropriate length (something near the round trip time). If L_i is the load at interval i, a network not subject to congestion can be modeled by saying L_i changes slowly compared to the sampling time. I.e.,

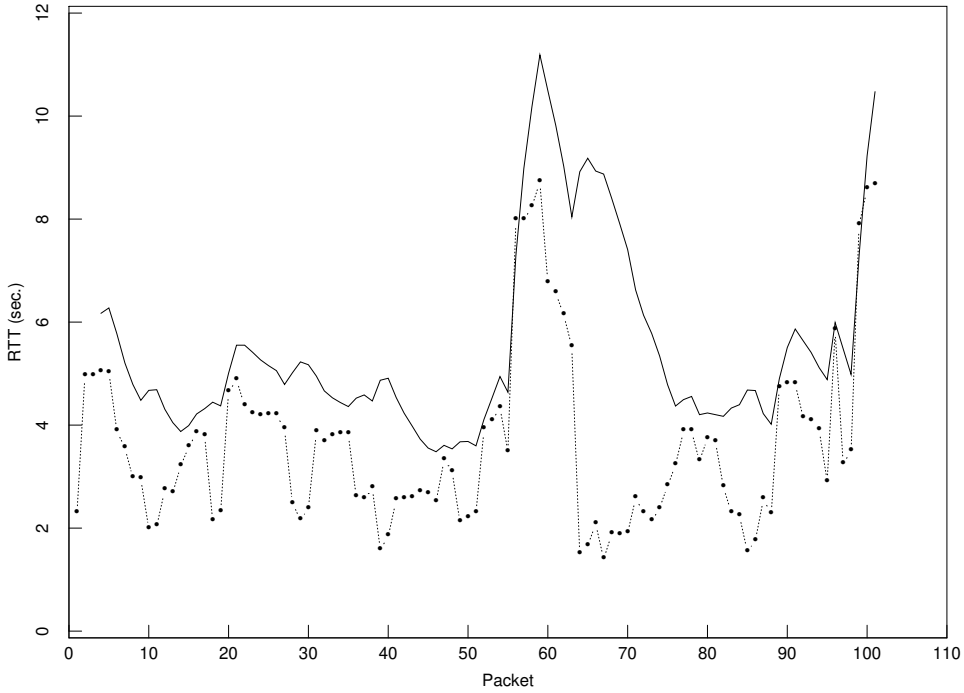
$$L_i = N$$

(N constant). If the network is subject to congestion, this zeroth order model breaks down. The average queue length becomes the sum of two terms, the N above that accounts for the average arrival rate of new traffic and intrinsic delay and a new term that accounts for the fraction of traffic left over from the last time interval:

$$L_i = N + \gamma L_{i-1}$$

Draft

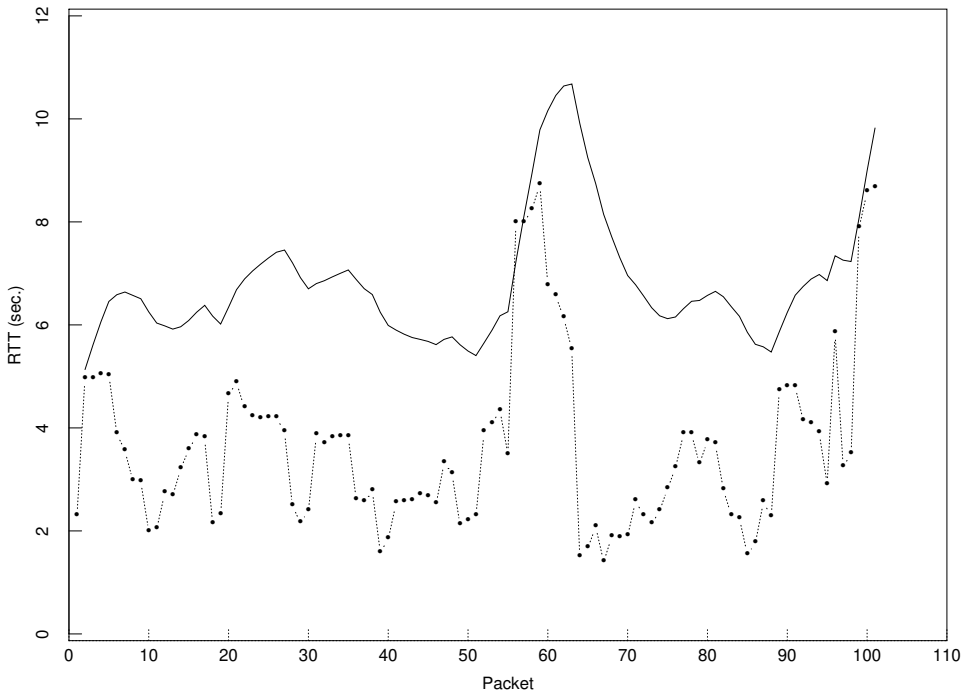
Figure 6: Performance of a Mean+Variance retransmit timer



Same data as above but the solid line shows a retransmit timer computed according to the algorithm in appendix A.

Draft

Figure 5: Performance of an RFC793 retransmit timer



This is trace data showing the per-packet round trip time on a well-behaved Arpanet connection. The x-axis is the packet number (packets were numbered sequentially, starting with one) and the y-axis is the elapsed time, in seconds, from the send of the packet to the sender's receipt of its ack. During this portion of the trace, no packets were dropped or retransmitted.

The packets are indicated by a dot. A dashed line connects them to make it easier to follow the sequence. The solid line shows the behavior of a retransmit timer computed according to the rules of RFC793.

Draft

2 Conservation at equilibrium: round-trip timing

Once data is flowing reliably, problems (2) and (3) should be addressed. Assuming that the protocol implementation is correct, (2) must represent a failure of sender's retransmit timer. A good round trip time estimator (the core of the retransmit timer) is the single most important feature of any protocol implementation that expects to survive heavy load. And it frequently gets botched (see [Zha86] and [Jai86a] for descriptions of typical problems).

One mistake seems to be not estimating the variance σ_R of the round trip time, R. From queuing theory we know that R and the variation in R increase very quickly with load. If the load is ρ (the ratio of average arrival rate to average departure rate), both R and σ_R scale like $(1-\rho)^{-1}$. To make this concrete, if the network is running at 75% of capacity (as the Arpanet was in last April's collapse), you should expect round-trip-time to vary by a factor of sixteen ($\pm 2\sigma$).

The RFC793 parameter β accounts for rtt variation. The suggested value of '2' can adapt to loads of at most 30%. Above this point, a connection will respond to load increases by retransmitting packets that have only been delayed in transit. This forces the network to do useless work (wasting bandwidth on duplicates of packets that will be delivered) at a time when it's known to be having trouble with useful work. I.e., this is the network equivalent of pouring gasoline on a fire.

We developed a cheap method for estimating variation (see appendix A)[3] and the resulting retransmit timer essentially eliminates spurious retransmissions. A pleasant side effect of estimating β rather than using a fixed value is that low load as well as high load performance improves, particularly over high delay paths such as satellite links (see figures 5 and 6).

Another timer mistake seems to be in the backoff after a retransmit: If a packet has to be retransmitted more than once, how should the retransmits be spaced? It turns out there's only one scheme that's likely to work, exponential backoff, but proving this is a bit involved.[4] We might finesse a proof by noting that a network is, to a very good approximation, a linear system. (That is, it is composed of elements that behave like linear operators — integrators, delays, gain stages, etc.). Linear system theory says that if a system is stable, the stability is exponential. This

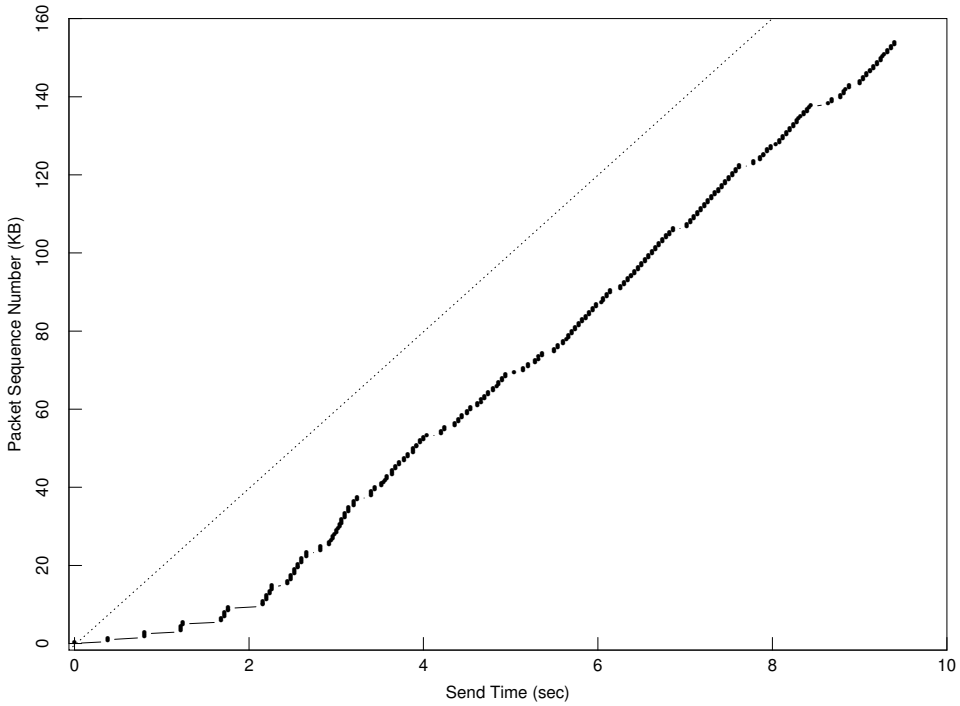
[3] We are far from the first to recognize that transport needs to estimate both mean and variation. See, for example, [Edg83]. But we do think our estimator is simpler than most.

Given history's recognition of this problem, we were surprised to learn from [ISO84] and [NBS83] that ISO standard networks would be so predictable that there was no need, or provision, for transport to estimate either mean or variance of the round-trip-time.

[4] An in-progress paper attempts a proof. If you view the bottleneck IP gateway as a 'shared resource with fixed capacity', it bears a remarkable resemblance to the 'ether' in an Aloha or Ethernet multi-access protocol. Thus the retransmit backoff problem is essentially the same as showing that no backoff 'slower' than an exponential will guarantee stability on an Ethernet.

Draft

Figure 4: Startup behavior of TCP with Slowstart

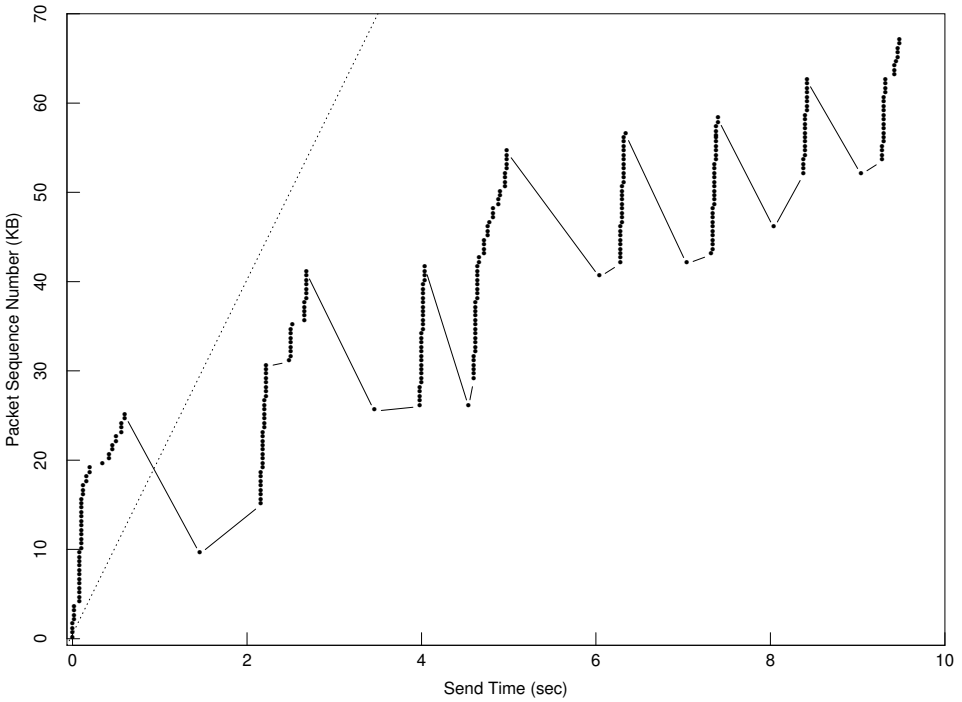


Same situation as the previous figure (same time of day, same Suns, same network path), except the machines were running the 4.3+ BSD TCP with slowstart.

No bandwidth is wasted on retransmits but two seconds is spent on the slow-start so the effective bandwidth of this part of the trace is 16KBs — two times better than figure 3. (This is slightly misleading: Unlike the previous figure, the slope of the trace is 20KBs and the effect of the 2 sec. offset decreases as the trace lengthens. E.g., if this trace had run a minute, the effective bandwidth would have been 19 KBs. The effective bandwidth without slow-start stays at 7KBs no matter how long the trace).

Draft

Figure 3: Startup behavior of TCP without Slowstart



Trace data of the start of a TCP conversation between two Sun 3/50s running Sun OS3.5 (the 4.3BSD TCP). The two Suns were on different Ethernets connected by IP gateways driving a 230.4kbs point-to-point link (essentially the setup shown in fig. 7).

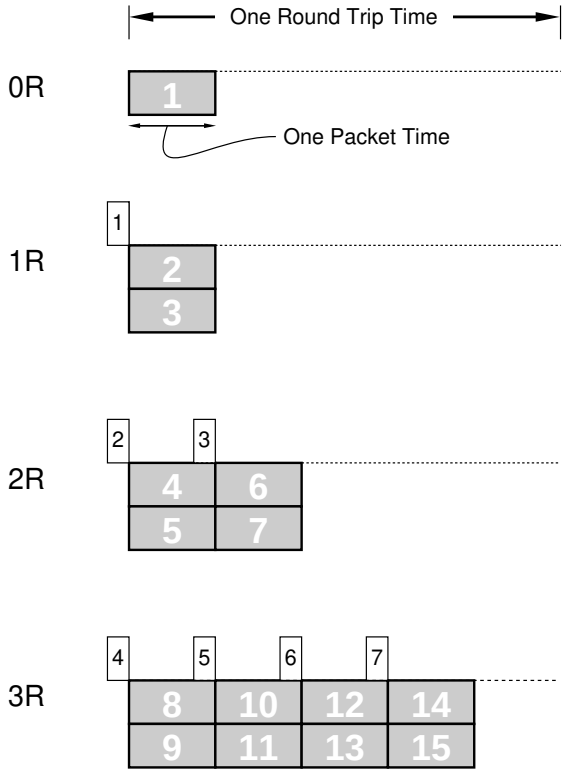
Each dot is a 512 data-byte packet. The x-axis is the time the packet was sent. The y-axis is the sequence number in the packet header. Thus a vertical array of dots indicate back-to-back packets and two dots with the same y but different x indicate a retransmit.

The dashed line shows the 20KBs available bandwidth for this connection. Only 35% of this bandwidth was used; the rest was wasted on retransmits (almost everything is retransmitted at least once and the stuff from 54 to 58KB is sent five times).

'Desirable' behavior on this graph would be a relatively smooth line of dots extending diagonally from the lower left to the upper right. The slope of this line would equal the available bandwidth. (Nothing in this trace resembles desirable behavior.)

Draft

Figure 2: The Chronology of a Slow-start



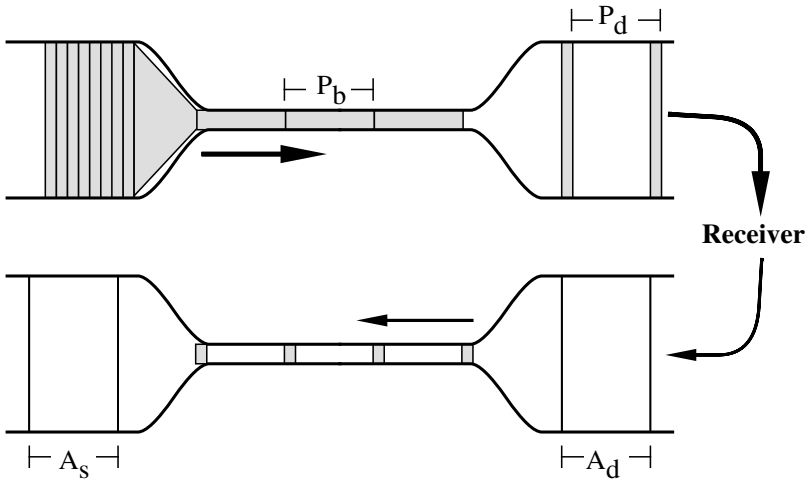
The horizontal direction is time. To fit things on the page, the continuous time line has been chopped into one-round-trip-time pieces which are stacked vertically with increasing time going down the page. The grey, numbered boxes are packets. The white numbered boxes are acks (the number tells which packet is being acked).

Notice that when the ack for a packet arrives, two packets are generated: one for the ack (the ack tells us a packet has left the system so we add a new packet to take its place) and a packet because the ack opens the congestion window by one. It may be clear from the figure why an add-one-packet-to-window policy opens the window exponentially in time.

Assuming that the local net is much faster than the long haul net, the bottleneck gateway sees the two packets that result from an ack arrive at essentially the same time. These two packets are shown stacked on top of one another (indicating that one of them would have to occupy space in the gateway's outbound queue). Thus the short-term queue demand on the gateway is increasing exponentially and opening a window of size W packets will require $W/2$ packets of buffer capacity at the bottleneck.

Draft

Figure 1: Window Flow Control 'Self-clocking'



This is a schematic representation of a sender (at the left) and receiver (at the right) on high bandwidth networks connected by a slower, long-haul net. The sender is just starting and has shipped a window's worth of packets, back-to-back. The ack for the first of those packets is about to arrive back at the sender (the vertical line at the mouth of the lower left funnel).

The vertical direction is bandwidth, the horizontal direction is time. Each of the shaded boxes is a packet. Bandwidth $\times$ Time $=$ Bits so the area of each box is the packet size.

The number of bits doesn't change as a packet goes through the network so, as each packet is squeezed into the smaller long-haul bandwidth, it must spread out in time. The time P_b represents the minimum packet spacing on the slowest link in the path (the *bottleneck*). As the packets leave the bottleneck for the destination net, there is nothing to change the interpacket interval so the destination net packet spacing $P_d = P_b$. If the receiver processing time is the same for all packets, the spacing between acks on the destination net $A_d = P_d = P_b$. If the time slot P_b was big enough for a packet, it's big enough for an ack so the ack spacing is preserved along the return path. Thus the ack spacing on the sender's net $A_s = P_b$.

So, if packets after the first burst are sent only in response to an ack, the sender's packet spacing will be exactly match the packet time on the slowest link in the path.

Draft

1 Getting to Equilibrium: Slow-start

Failure (1) has to be from a connection that is either starting or restarting after a packet loss. Another way to look at the conservation property is to say that the sender uses acks as a 'clock' to strobe new packets into the network. Since the receiver can generate acks no faster than data packets can get through the network, the protocol is 'self clocking' (see figure 1). Self clocking systems automatically adjust to bandwidth and delay variations and have a wide dynamic range (an important issue when you realize that TCP spans everything from 800 Mbit/sec Cray channels to 1200 bit/sec packet radio links). But the same thing that makes a self-clocked system stable when it's running makes it hard to get started — to get data flowing you must have acks to clock out packets but to get acks you must have data flowing.

To start the 'clock', we developed an algorithm called *slow-start* that gradually increases the amount of data in-transit.[2] Although we flatter ourselves that the design of this algorithm is rather subtle, the implementation is trivial — one new state variable and three lines of code in the sender:

- Add a *congestion window*, cwnd, to the per-connection state.
- When starting or restarting after a loss, set cwnd to one packet.
- On each ack for new data, increase cwnd by one packet.
- When sending, send the minimum of the receiver's advertised window and cwnd.

Actually, the slow-start window increase isn't all that slow: The window opening takes time equal to $R \log_2 W$ where R is the round-trip-time and W is the window size in packets (see figure 2). This opens the window quickly enough to have a negligible effect on performance, even on links with a very large bandwidth–delay product. And the algorithm guarantees that a connection will source data at a rate at most twice the maximum possible on the path. Without slow-start, by contrast, when 10Mb Ethernet hosts talk over the 56Kb Arpanet via IP gateways, the gateways see a burst of eight to sixteen packets delivered at 200 times the path bandwidth (see figures 3 and 4).

[2] Slow-start is quite similar to the CUTE algorithm described in [Jai86b]. We didn't know about CUTE at the time we were developing slowstart but we should have—CUTE preceded our work by several months.

Draft

Our own measurements and the reports of our beta testers suggest that the final product is fairly good at dealing with congested conditions on the Internet.

This paper is a brief description of (1)–(5) and the rationale behind them. (6) is an algorithm recently developed by Phil Karn, described well in [KP87]. (7) is described in a soon-to-be-published RFC.

Algorithms (1)–(5) all spring from one observation: The flow on a TCP connection (or ISO TP-4 or XNS SPP connection) should, by the design of the protocol, obey a 'conservation of packets' principle. And, if this principle were obeyed, congestion collapse would become the exception rather than the rule. Thus congestion control turns into finding places where conservation is violated and fixing them.

By 'conservation of packets' I mean the following: If you look at the connection 'in equilibrium', i.e., running stably with a full window of data in transit, you notice that the flow is what a physicist would call 'conservative': A new packet isn't put into the network until an old packet leaves. The physics of flow predicts that systems with this property should be extremely robust in the face of congestion.[1] Observation of the Arpanet or NSFNet suggests that they were not particularly robust in the face of congestion. From whence comes the discrepancy?

There are only three ways for packet conservation to fail:

1. The connection doesn't get to equilibrium, or
2. A sender injects a packet before an old packet has exited, or
3. The equilibrium can't be reached because of resource limits along the path.

In the following sections, we treat each of these in turn.

[1] The author realizes that thermodynamics and hydrodynamics are not prerequisites for a well-rounded background in networks and is groping, without success, for an intuitive explanation of why conservative flow should resist congestive collapse. A mathematician tells me that conservation implies three equations for the packet flow J:

$$\frac{\partial \rho(r,t)}{\partial t} = -\nabla \cdot J \qquad J = -D\nabla\rho(r,t) \qquad \int_{-\infty}^{+\infty} \rho(r,t)dr = W$$

The first, local conservation, says that any change in packet density ρ at position r and time t has to come from a flow (i.e., packets move but aren't created or destroyed). The second, Fick's law, says flow is determined by the density gradient (i.e., packets flow into holes). The third, global conservation, says that if you snapshot the entire system at any time and count all the packets, you always get the same number (the connection window size). Supposedly the third equation is a Lyapunov function for the system so it's easy to prove stability. Perhaps you would like to take this on faith. I did. (See [Lue79], chap. 9 or [BC87], chap. 11–12 if you're not as trusting.)

Congestion Avoidance and Control

Van Jacobson*

University of California
Lawrence Berkeley Laboratory
Berkeley, CA 94720

April 4, 1988

In October of '86, the Internet had the first of what turned into a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards or three IMP hops) dropped from 32kbps to 40bps. Fascinated by this sudden factor-of-thousand drop in bandwidth (and having nothing better to do since the net was effectively unusable), Mike Karels and I embarked on an investigation of why things had gotten so bad. In particular, we wondered if 4.3BSD (Berkeley Unix) TCP was mis-behaving and/or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions turned out to be yes.

Since that time, Mike and I have stared at a lot of trace data and were eventually motivated to put seven new algorithms into the 4BSD TCP:

1. round-trip-time variance estimation
2. exponential retransmit timer backoff
3. slow-start
4. more aggressive receiver ack policy
5. dynamic window sizing based on congestion
6. Karn's clamped retransmit backoff
7. fast retransmit

*This work was supported in part by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

Draft