Congestion Avoidance and Control[*]

Van Jacobson[†]
Lawrence Berkeley Laboratory
Michael J. Karels[‡]
University of California at Berkeley

November, 1988

Introduction

Computer networks have experienced an explosive growth over the past few years and with that growth have come severe congestion problems. For example, it is now common to see internet gateways drop 10% of the incoming packets because of local buffer overflows. Our investigation of some of these problems has shown that much of the cause lies in transport protocol implementations (*not* in the protocols themselves): The 'obvious' ways to implement a window-based transport protocol can result in exactly the wrong behavior in response to network congestion. We give examples of 'wrong' behavior and describe some simple algorithms that can be used to make right things happen. The algorithms are rooted in the idea of achieving network stability by forcing the transport connection to obey a 'packet conservation' principle. We show how the algorithms derive from this principle and what effect they have on traffic over congested networks.

In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. In particular, we wondered if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

Since that time, we have put seven new algorithms into the 4BSD TCP:

- *(i)* round-trip-time variance estimation
- *(ii)* exponential retransmit timer backoff
- *(iii)* slow-start
- *(iv)* more aggressive receiver ack policy
- *(v)* dynamic window sizing on congestion
- *(vi)* Karn's clamped retransmit backoff
- *(vii)* fast retransmit

Our measurements and the reports of 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 *(i)* – *(v)* and the rationale behind them. *(vi)* is an algorithm recently developed by Phil Karn of Bell Communications Research, described in [15]. *(vii)* is described in a soon-to-be-published RFC (ARPANET "Request for Comments").

Algorithms *(i)* – *(v)* spring from one observation: The flow on a TCP connection (or ISO TP-4 or Xerox NS SPP connection) should 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 involves finding places that violate conservation and fixing them.

By 'conservation of packets' we mean that for a connection 'in equilibrium', i.e., running stably with a full window of data in transit, the packet 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 robust in the face of congestion.[1] Observation of the Internet suggests that it was not particularly robust. Why the discrepancy?

---

[*] **Note:** This is a very slightly revised version of a paper originally presented at SIGCOMM '88 [11]. If you wish to reference this work, please cite [11].

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

[‡] This work was supported by the U.S. Department of Commerce, National Bureau of Standards, under Grant Number 60NANB8D0830.

[1] A conservative flow means that for any given time, the integral of the packet density around the sender–receiver–sender loop is a constant. Since

There are only three ways for packet conservation to fail:

1. The connection doesn't get to equilibrium, or
2. A sender injects a new 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 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' (fig. 1). Self clocking systems automatically adjust to bandwidth and delay variations and have a wide dynamic range (important considering that TCP spans a range from 800 Mbps Cray channels to 1200 bps packet radio links). But the same thing that makes a self-clocked system stable when it's running makes it hard to start — to get data flowing there must be acks to clock out packets but to get acks there must be data flowing.

To start the 'clock', we developed a *slow-start* algorithm to gradually increase 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 that slow: it takes time $R \log_2 W$ where R is the round-trip-time and W is the window size in packets (fig. 2). This means the window opens quickly enough to have a negligible effect on performance, even on links with a 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 10 Mbps Ethernet hosts talk over the 56 Kbps Arpanet via IP gateways, the first-hop gateway sees a burst of eight packets delivered at 200 times the path bandwidth. This burst of packets often puts the connection into a persistent failure mode of continuous retransmissions (figures 3 and 4).

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 is frequently botched ([26] and [12] describe typical problems).

One mistake is not estimating the variation, σ_R, of the round trip time, R. From queuing theory we know that R and the variation in R increase quickly with load. If the load is ρ (the ratio of average arrival rate to average departure rate), 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, one should expect round-trip-time to vary by a factor of sixteen (-2σ to $+2\sigma$).

The TCP protocol specification[23] suggests estimating mean round trip time via the low-pass filter

$$R \leftarrow \alpha R + (1-\alpha) M$$

where R is the average RTT estimate, M is a round trip time measurement from the most recently acked data packet, and α is a filter gain constant with a suggested value of 0.9. Once the R estimate is updated, the retransmit timeout interval, rto, for the next packet sent is set to βR.

The parameter β accounts for RTT variation (see [4], section 5). The suggested $\beta = 2$ 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,

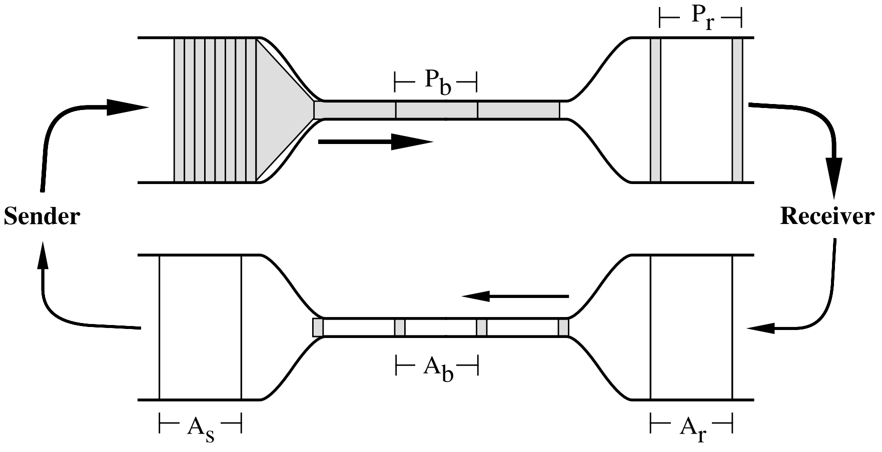
---

packets have to 'diffuse' around this loop, the integral is sufficiently continuous to be a Lyapunov function for the system. A constant function trivially meets the conditions for Lyapunov stability so the system is stable and any superposition of such systems is stable. (See [2], chap. 11–12 or [20], chap. 9 for excellent introductions to system stability theory.)

[2] Slow-start is quite similar to the CUTE algorithm described in [13]. We didn't know about CUTE at the time we were developing slow-start but we should have—CUTE preceded our work by several months.

When describing our algorithm at the Feb., 1987, Internet Engineering Task Force (IETF) meeting, we called it *soft-start*, a reference to an electronics engineer's technique to limit in-rush current. The name *slow-start* was coined by John Nagle in a message to the IETF mailing list in March, '87. This name was clearly superior to ours and we promptly adopted it.

Figure 1: Window Flow Control 'Self-clocking'



This is a schematic representation of a sender and receiver 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 dimension is bandwidth, the horizontal dimension is time. Each of the shaded boxes is a packet. Bandwidth × 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 a packet squeezed into the smaller long-haul bandwidth 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, nothing changes the inter-packet interval so on the receiver's net packet spacing $P_r = P_b$. If the receiver processing time is the same for all packets, the spacing between acks on the receiver's net $A_r = P_r = P_b$. 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 exactly match the packet time on the slowest link in the path.

wasting bandwidth on duplicates of packets that will eventually 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 (figures 5 and 6).

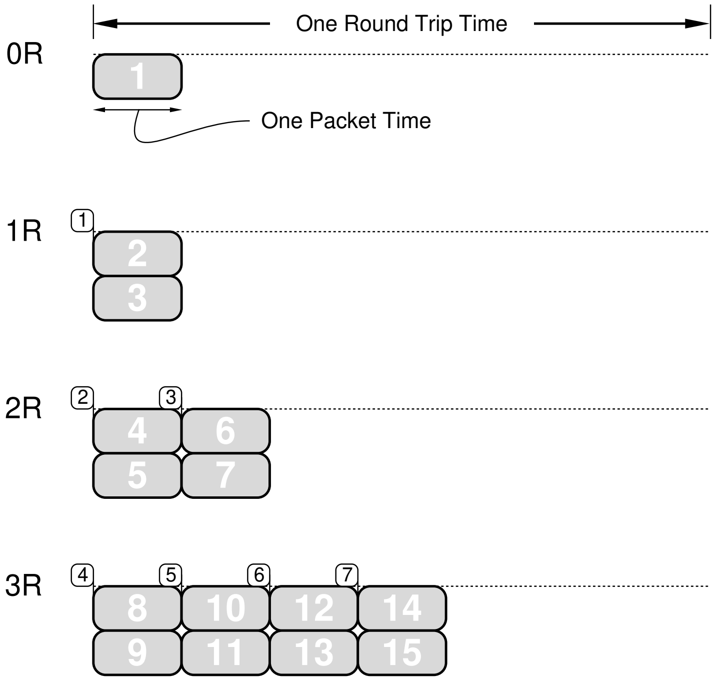
Another timer mistake is in the backoff after a retransmit: If a packet has to be retransmitted more than once, how should the retransmits be spaced? For a transport endpoint embedded in a network of unknown topology and with an unknown, unknowable and constantly changing population of competing conversations, only one scheme has any hope of working—exponential backoff—but a proof of this is beyond the scope of this paper.[4] To finesse a proof, note 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 suggests that an unstable system (a network subject

---

[3] We are far from the first to recognize that transport needs to estimate both mean and variation. See, for example, [5]. But we do think our estimator is simpler than most.

[4] See [7]. Several authors have shown that backoffs 'slower' than exponential are stable given finite populations and knowledge of the global traffic. However, [16] shows that nothing slower than exponential behavior will work in the general case. To feed your intuition, consider that an IP gateway has essentially the same behavior as the 'ether' in an ALOHA net or Ethernet. Justifying exponential retransmit backoff is the same as showing that no collision backoff slower than an exponential will guarantee stability on an Ethernet. Unfortunately, with an infinite user population even exponential backoff won't guarantee stability (although it 'almost' does—see [1]). Fortunately, we don't (yet) have to deal with an infinite user population.

Figure 2: The Chronology of a Slow-start



The horizontal direction is time. The continuous time line has been chopped into one-round-trip-time pieces stacked vertically with increasing time going down the page. The grey, numbered boxes are packets. The white numbered boxes are the corresponding acks. As each ack arrives, two packets are generated: one for the ack (the ack says a packet has left the system so a new packet is added to take its place) and one because an ack opens the congestion window by one packet. It may be clear from the figure why an add-one-packet-to-window policy opens the window exponentially in time. If the local net is much faster than the long haul net, the ack's two packets arrive at the bottleneck 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.

to random load shocks and prone to congestive collapse[5]) can be stabilized by adding some exponential damping (exponential timer backoff) to its primary excitation (senders, traffic sources).

3 Adapting to the path: congestion avoidance

If the timers are in good shape, it is possible to state with some confidence that a timeout indicates a lost packet and not a broken timer. At this point, something can be done about (3). Packets get lost for two reasons: they are damaged in transit, or the network is congested and somewhere on the path there was insufficient buffer capacity. On most network paths, loss due to damage is rare ($\ll 1\%$) so it is probable that a packet loss is due to congestion in the network.[6]

---

[5] The phrase *congestion collapse* (describing a positive feedback instability due to poor retransmit timers) is again the coinage of John Nagle, this time from [22].

[6] Because a packet loss empties the window, the throughput of any window flow control protocol is quite sensitive to damage loss. For an RFC793 standard TCP running with window w (where w is at most the bandwidth-delay product), a loss probability of p degrades throughput by a factor of $(1 + 2pw)^{-1}$. E.g., a 1% damage loss rate on an Arpanet path (8 packet window) degrades TCP throughput by 14%.

The congestion control scheme we propose is insensitive to damage loss until the loss rate is on the order of the window equilibration length (the number of packets it takes the window to regain its original size after a loss). If the pre-loss size is w, equilibration takes roughly $w^2/3$ packets so, for the

A ‘congestion avoidance’ strategy, such as the one proposed in [14], will 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 [14] which requires a new bit in the packet headers and a modification 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[7] and follows directly from a first-order time-series model of the network:[8] Say network load is measured 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, an uncongested network 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 and the effect of this left-over traffic (e.g., induced retransmits):

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

(These are the first two terms in a Taylor series expansion of $L(t)$. There is reason to believe one might eventually need a three term, second order model, but not until the Internet has grown substantially.)

When the network is congested, γ must be large and the queue lengths will start increasing exponentially.[9] The system will stabilize only if the traffic sources throttle back at least as quickly 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

On congestion:

$$W_i = dW_{i-1} \qquad (d < 1)$$

I.e., a multiplicative decrease of the window size (which becomes an exponential decrease over time if the congestion persists).

If there’s no congestion, γ must be near zero and the load approximately constant. The network announces, via a dropped packet, when demand is excessive but says nothing if a connection is using less than its fair share (since the network is stateless, it cannot know this). Thus a connection has to increase its bandwidth utilization 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 be wasted unless your window size is increased. What should the increase policy be?

The first thought is to use a symmetric, multiplicative increase, possibly with a longer time constant, $W_i = bW_{i-1}$, $1 < b \leq 1/d$. This is a mistake. The result will oscillate wildly and, on the average, deliver poor throughput. The analytic reason for this has to do with that fact that it is easy to drive the net into saturation but hard for the net to recover (what [17], chap. 2.1, calls the *rush-hour effect*).[10] Thus

---

Arpanet, the loss sensitivity threshold is about 5%. At this high loss rate, the empty window effect described above has already degraded throughput by 44% and the additional degradation from the congestion avoidance window shrinking is the least of one’s problems.

We are concerned that the congestion control noise sensitivity is quadratic in w but it will take at least another generation of network evolution to reach window sizes where this will be significant. If experience shows this sensitivity to be a liability, a trivial modification to the algorithm makes it linear in w. An in-progress paper explores this subject in detail.

[7] This is not an accident: We copied Jain’s scheme after hearing his presentation at [9] and realizing that the scheme was, in a sense, universal.

[8] See any good control theory text for the relationship between a system model and admissible controls for that system. A nice introduction appears in [20], chap. 8.

[9] 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 $\gamma > 1$.

[10] In fig. 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 queue would shrink only if the gateway could move packets into the skinny pipe faster than the sender dumped packets into the fat pipe. But the system tunes itself so each 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 will scale like $(1 - \rho)^{-2}$ ([17], 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 free bandwidth to dissipate queues that inevitably form due to path testing and 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.

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

Without justification, we'll state that the best increase policy is to make small, constant changes to the window size:[11]

On no congestion:

$$W_i = W_{i-1} + u \qquad (u \ll W_{max})$$

where W_{max} is the *pipesize* (the delay-bandwidth product of the path minus protocol overhead — i.e., the largest sensible window for the unloaded path). This is the additive increase / multiplicative decrease policy suggested in [14] 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 partially explained in appendix D. 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's 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).[12]
- 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 to the above. But, because both congestion avoidance and slow-start are triggered by a timeout and both manipulate the congestion window, they are frequently confused. 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.[13]

Figures 7 through 12 show the behavior of TCP connections with and without congestion avoidance. Although the test conditions (e.g., 16 KB windows) were deliberately chosen to stimulate congestion, the test scenario isn't far from common practice: 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 (4 KB). Thus simultaneous conversations between, say, any two hosts at Berkeley and any two hosts at MIT would exceed the network capacity of the UCB–MIT IMP path and would lead[14] to the type of behavior shown.

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 allocation. Thus, we view 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 becomes

---

[11] See [3] for a complete analysis of these increase and decrease policies. Also see [7] and [8] for a control-theoretic analysis of a similar class of control policies.

[12] This increment rule may be 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', 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, the sender has effective *silly window* avoidance (see [4], section 3) 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 maximum segment size and *cwnd* is expressed in bytes, not packets.

[13] 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 approach has been tried, e.g., [18] and [24], and appears to oscillate badly. To see why, note that for an inter-packet 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, 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 simulated the above algorithm and it appears to perform well. To test the predictions of that simulation against reality, we have a cooperative project with Sun Microsystems to prototype RPC dynamic congestion control algorithms using NFS as a test-bed (since NFS is known to have congestion problems yet it would be desirable to have it work over the same range of networks as TCP).

[14] *did* lead.

starved for traffic. Since we plan to continue using packet drops as a congestion signal, gateway 'self protection' from a mis-behaving host should 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 detected early, small adjustments to the senders' windows will cure it. Otherwise massive adjustments are 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. Jain[14] 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 or significant second-order dynamics in the traffic.[15] We plan to use some of our earlier work on ARMAX models for round-trip-time/queue length prediction as the basis of detection. Preliminary results suggest that this approach works well at high load, is immune to second-order effects in the traffic and is computationally cheap enough to not slow down kilopacket-per-second gateways.

Acknowledgements

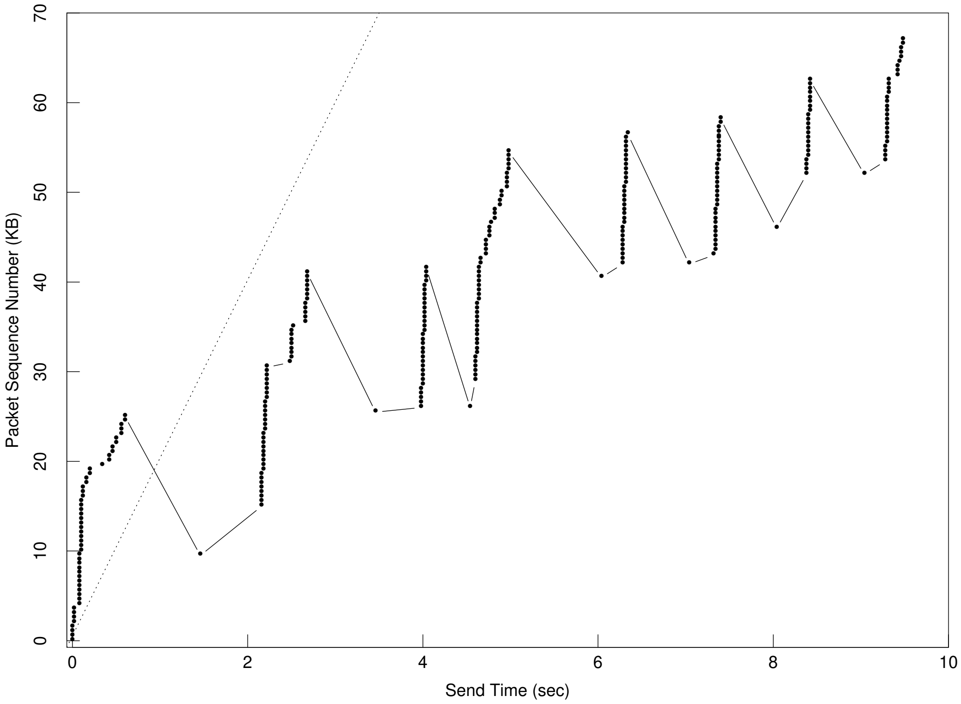
We are grateful to the members of the Internet Activity Board's End-to-End and Internet-Engineering task forces for this past year's interest, encouragement, cogent questions and network insights. Bob Braden of ISI and Craig Partridge of BBN were particularly helpful in the preparation of this paper: their careful reading of early drafts improved it immensely.

The first author is also deeply in debt to Jeff Mogul of DEC Western Research Lab. Without Jeff's interest and patient prodding, this paper would never have existed.

---

[15] These problems stem from the fact that the average time between regeneration points scales like $(1 - \rho)^{-1}$ and the variance like $(1 - \rho)^{-3}$ (see Feller[6], chap. VI.9). Thus the congestion detector becomes sluggish as congestion increases and its signal-to-noise ratio decreases dramatically.

Figure 3: Startup behavior of TCP without Slow-start



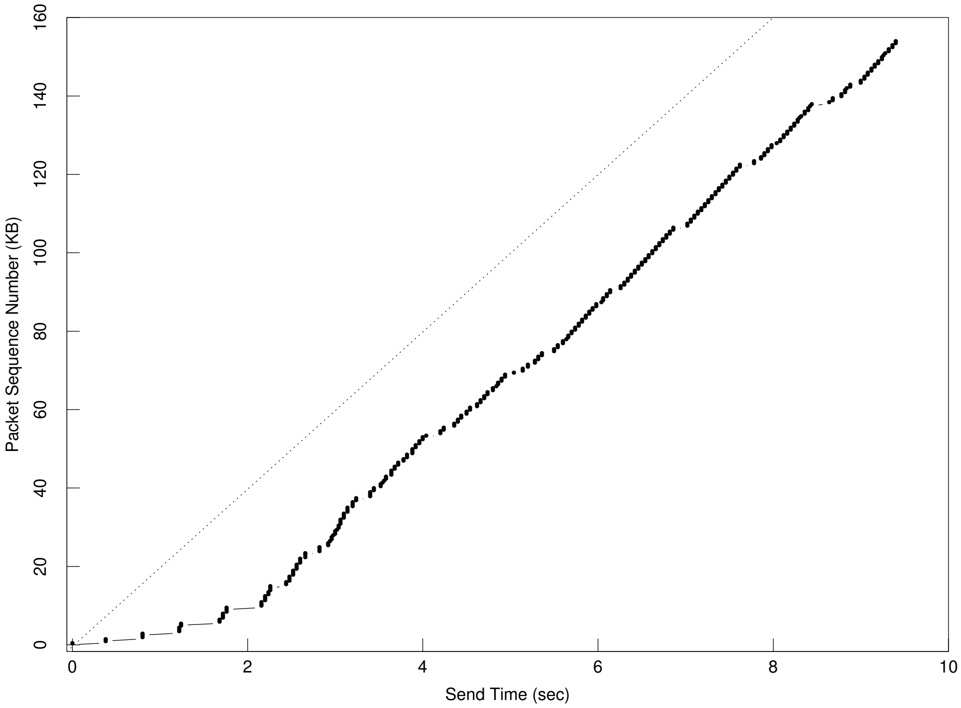
Trace data of the start of a TCP conversation between two Sun 3/50s running Sun OS 3.5 (the 4.3BSD TCP). The two Suns were on different Ethernets connected by IP gateways driving a 230.4 Kbps point-to-point link (essentially the setup shown in fig. 7). The window size for the connection was 16KB (32 512-byte packets) and there were 30 packets of buffer available at the bottleneck gateway. The actual path contains six store-and-forward hops so the pipe plus gateway queue has enough capacity for a full window but the gateway queue alone does not.

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.

'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.

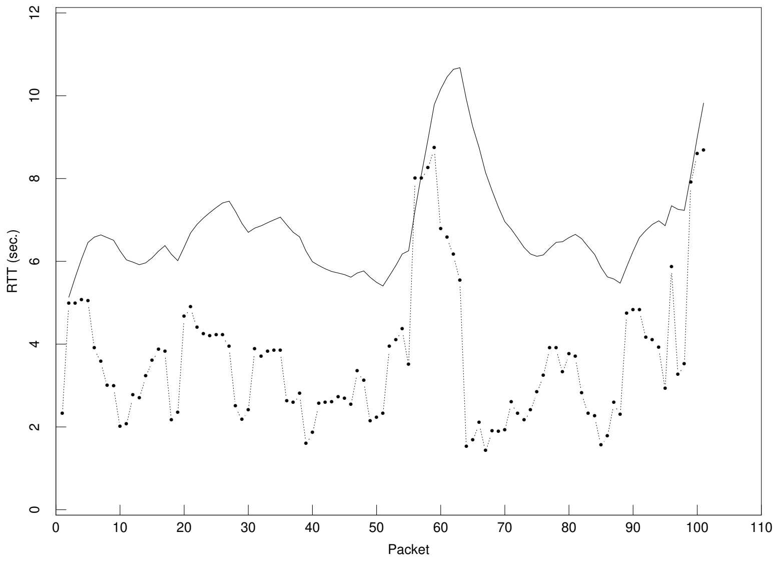
The dashed line shows the 20 KBps bandwidth available for this connection. Only 35% of this bandwidth was used; the rest was wasted on retransmits. Almost everything is retransmitted at least once and data from 54 to 58 KB is sent five times.

Figure 4: Startup behavior of TCP with Slow-start



Same conditions as the previous figure (same time of day, same Suns, same network path, same buffer and window sizes), except the machines were running the 4.3^{+} TCP with slow-start. 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 16 KBps — two times better than figure 3. (This is slightly misleading: Unlike the previous figure, the slope of the trace is 20 KBps and the effect of the 2 second offset decreases as the trace lengthens. E.g., if this trace had run a minute, the effective bandwidth would have been 19 KBps. The effective bandwidth without slow-start stays at 7 KBps no matter how long the trace.)

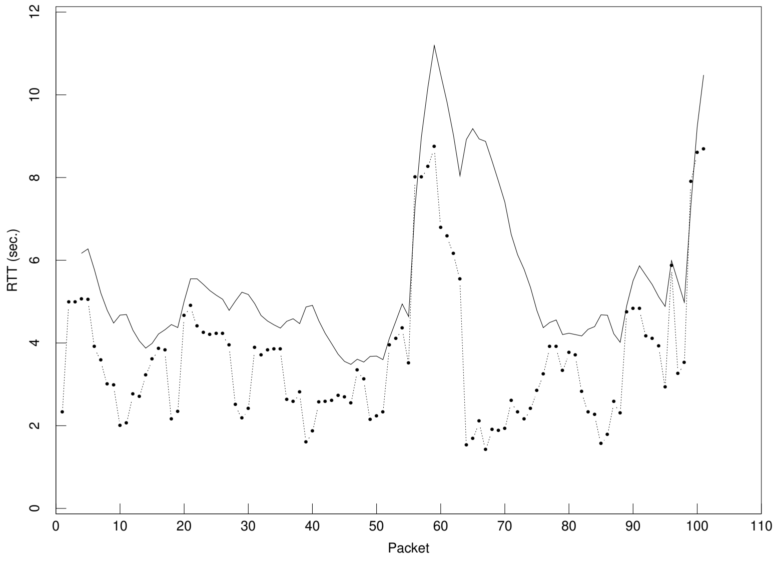
Figure 5: Performance of an RFC793 retransmit timer



Trace data showing 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 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.

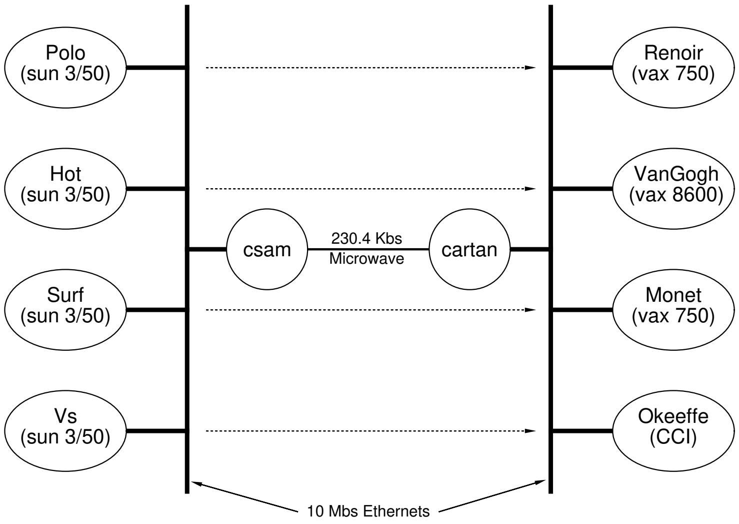
The packets are indicated by a dot. A dashed line connects them to make the sequence easier to follow. The solid line shows the behavior of a retransmit timer computed according to the rules of RFC793.

Figure 6: Performance of a Mean+Variance retransmit timer



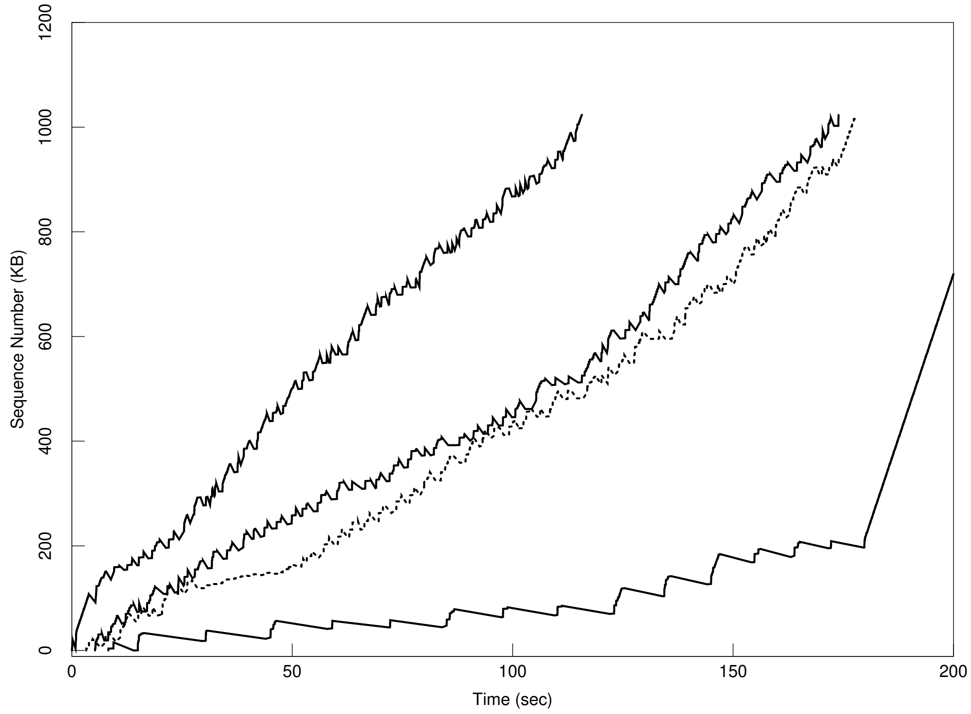
Same data as above but the solid line shows a retransmit timer computed according to the algorithm in appendix A.

Figure 7: Multiple conversation test setup


Polo
(sun 3/50)
Hot
(sun 3/50)
Surf
(sun 3/50)
Vs
(sun 3/50)
csam
230.4 Kbs
Microwave
cartan
Renoir
(vax 750)
VanGogh
(vax 8600)
Monet
(vax 750)
Okeeffe
(CCI)
10 Mbs Ethernets

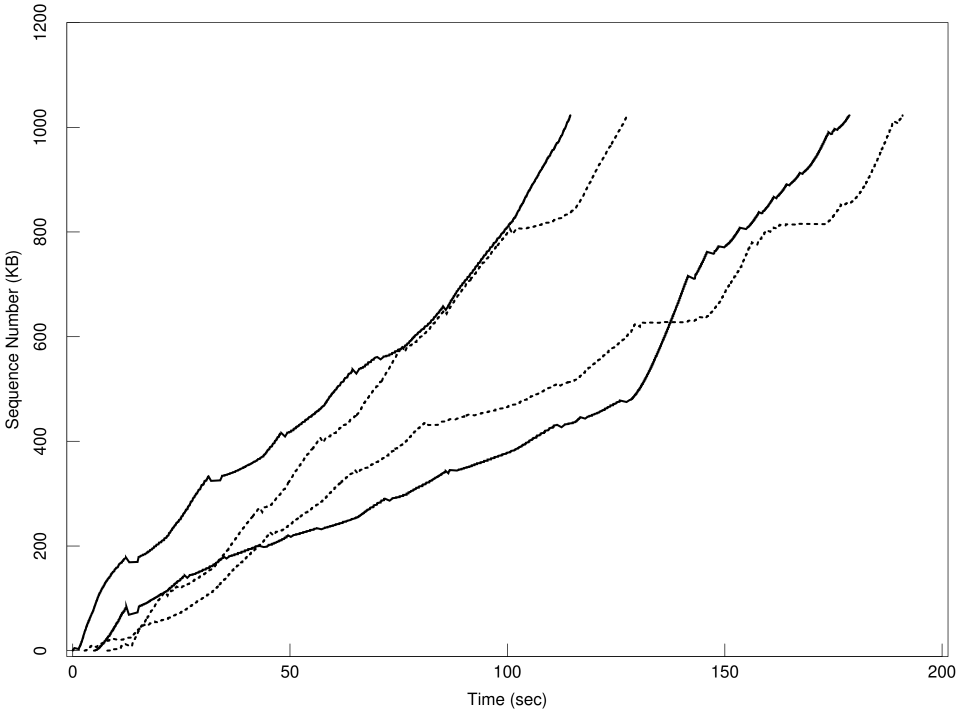
Test setup to examine the interaction of multiple, simultaneous TCP conversations sharing a bottleneck link. 1 MByte transfers (2048 512-data-byte packets) were initiated 3 seconds apart from four machines at LBL to four machines at UCB, one conversation per machine pair (the dotted lines above show the pairing). All traffic went via a 230.4 Kbps link connecting IP router **csam** at LBL to IP router **cartan** at UCB. The microwave link queue 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 queue capacity by 160%.

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 KBps, each of the four conversations should have received 6 KBps. Instead, one conversation got 8 KBps, two got 5 KBps, one got 0.5 KBps and 6 KBps has vanished.

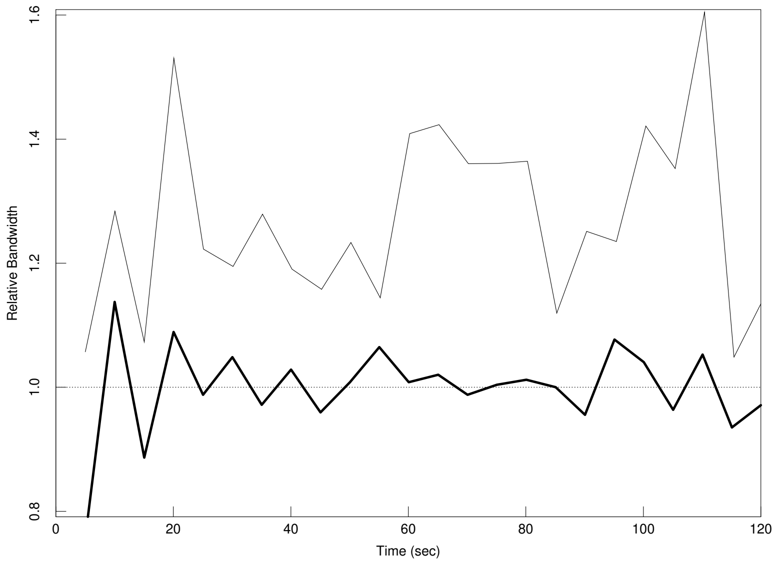
Figure 9: Multiple, simultaneous TCPs with congestion avoidance



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 KBps and two got 4.5 KBps (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 KBps senders were talking to 4.3BSD receivers which would delay an ack until 35% of the window was filled or 200 ms 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 KBps senders were talking to 4.3^{+} BSD receivers which would delay an ack for at most one packet (because of an ack's 'clock' rôle, the authors believe that the minimum ack frequency should be every other packet). I.e., the sender would deliver bursts of at most two packets. The probability of loss increases 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.

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 25 KBps 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 second interval is low (because of the slow-start), then there is about 20 seconds 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 seconds is a bandwidth 're-negotiation' due to connection one shutting down. The activity around 80 seconds is a reflection of the 'flat spot' in fig. 9 where most of conversation two's bandwidth is suddenly shifted to conversations three and four — competing conversations frequently exhibit this type of 'punctuated equilibrium' behavior and we hope to investigate its dynamics in a future paper.)

Figure 11: Effective bandwidth of old and new TCPs

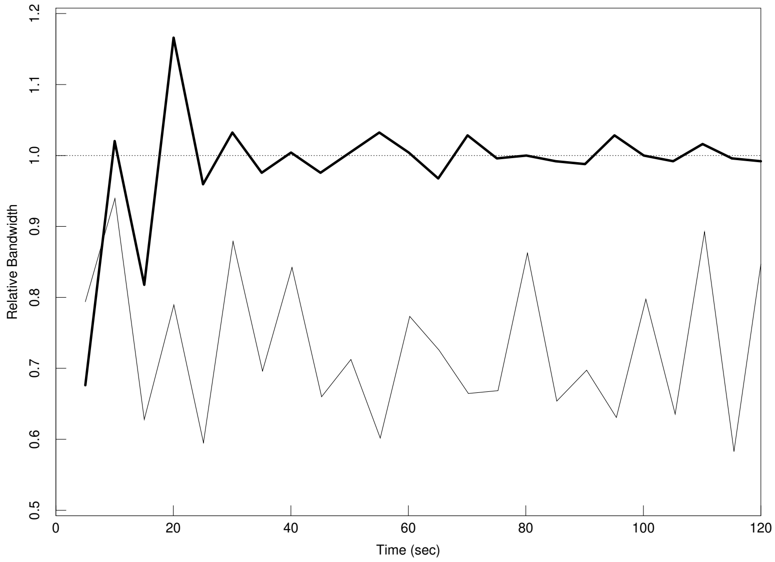
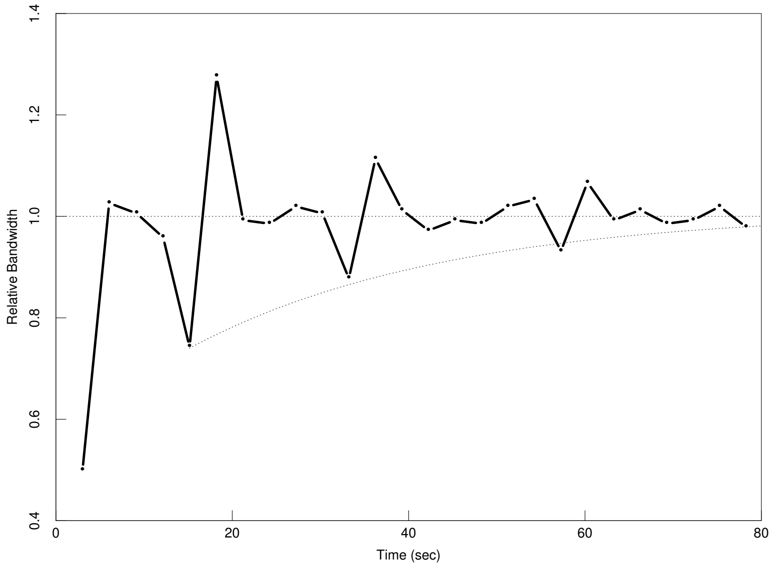


Figure 10 showed the old TCPs were using 25% more than the bottleneck link bandwidth. Thus, once the bottleneck queue filled, 25% of the the senders' packets were being discarded. If the discards, and only the discards, were retransmitted, the senders would have received the full 25 KBps link bandwidth (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.

Figure 12: Window adjustment detail



Because of the five second averaging time (needed to smooth out the spikes in the old TCP data), 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 a packet is dropped, the sender sends until it fills the window, then stops until the retransmission timeout. Since the receiver cannot ack 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 the out-of-order data it cached. 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 is a least squares fit to the six window size measurements obtained from these events. The fit time constant was 28 seconds. (The long time constant is due to lack of a congestion avoidance algorithm in the gateway. With a 'drop' algorithm running in the gateway, the time constant should be around 4 seconds)

A A fast algorithm for rtt mean and variation

A.1 Theory

The RFC793 algorithm for estimating the mean round trip time is one of the simplest examples 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 [19] and it's probably worth looking 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 more sense, and computes faster, if we rearrange and collect terms multiplied by g to get

$$a \leftarrow a + g(m-a)$$

Think of a as a prediction of the next measurement. $m - a$ 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 (random, unpredictable effects like fluctuations in competing traffic) and (2) error due to a bad choice of a. Calling the random error E_r and the estimation error E_e,

$$a \leftarrow a + gE_r + gE_e$$

The gE_e term gives a a kick in the right direction while the gE_r 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 E_e but a small g to minimize the damage from E_r. Since the E_e 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 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 g sdev(m). Also that a converges to the true average exponentially with time constant $1/g$. So a smaller g gives a stabler a at the expense of taking a much longer time to get to the true average.

If we want some measure of the variation in m, say to compute a good value for the TCP retransmit timer, there are several alternatives. Variance, σ^2, is the conventional choice because it has some nice mathematical properties. But computing 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 is the mean prediction error or mean deviation, the average of $|m-a|$. Also, since

$$mdev^2 = \left(\sum |m-a|\right)^2 \;\geq\; \sum |m-a|^2 = \sigma^2$$

mean deviation is a more conservative (i.e., larger) estimate of variation than standard deviation.[16]

There's often 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

Fast estimators for average a and mean deviation v given measurement m follow directly from the above. Both estimators compute means so there are two instances of the RFC793 algorithm:

$$Err \equiv m - a$$

$$a \leftarrow a + gErr$$

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

To be computed quickly, the above should be done in integer arithmetic. But the expressions contain fractions ($g < 1$) so some scaling is needed to keep everything integer. A reciprocal power of 2 (i.e., $g = 1/2^n$ for some n) is a particularly good choice for g since the scaling can be implemented with shifts. Multiplying through by $1/g$ gives

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

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

To minimize round-off error, the scaled versions of a and v, sa and sv, should be kept rather than the unscaled versions. Picking $g = .125 = \frac{1}{8}$ (close to the .1 suggested in RFC793) and expressing the above in C:

---

[16] Purists may note that we elided a factor of $1/n$, the number of samples, from the previous inequality. It makes no difference to the result.

```
/* update Average estimator */
m -= (sa >> 3);
sa += m;
/* update Deviation estimator */
if (m < 0)
        m = -m;
m -= (sv >> 3);
sv += m;
```

It's not necessary to use the same gain for a and v. To force the timer to go up quickly in response to changes in the RTT, it's a good idea to give v a larger gain. In particular, because of window–delay mismatch there are often RTT artifacts at integer multiples of the window size.[17] To filter these, one would like $1/g$ in the a estimator to be at least as large as the window size (in packets) and $1/g$ in the v estimator to be less than the window size.[18]

Using a gain of .25 on the deviation and computing the retransmit timer, rto, as $a + 4v$, the final timer code looks like:

```
m -= (sa >> 3);
sa += m;
if (m < 0)
        m = -m;
m -= (sv >> 2);
sv += m;
rto = (sa >> 3) + sv;
```

In general this computation will correctly round rto: Because of the sa truncation when computing $m-a$, sa will converge to the true mean rounded up to the next tick. Likewise with sv. Thus, on the average, there is half a tick of bias in each. The rto computation should be rounded by half a tick and one tick needs to be added to account for sends being phased randomly with respect to the clock. So, the 1.75 tick bias contribution from $4v$ approximately equals the desired half tick rounding plus one tick phase correction.

B The combined slow-start with congestion avoidance algorithm

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

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

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.

Note that the **else** clause of the above code will malfunction if *cwnd* is an integer in unscaled, one-packet units. I.e., if the maximum window for the path is w packets, *cwnd* must cover the range $0..w$ with resolution of at least $1/w$.[19] Since sending packets smaller than the maximum transmission unit for the path lowers efficiency, the implementor must take care that the fractionally sized *cwnd* does *not* result in small packets being sent. In reasonable TCP implementations, existing silly-window avoidance code should prevent runt packets but this point should be carefully checked.

C Window adjustment interaction with round-trip timing

Some TCP connections, particularly those over a very low speed link such as a dial-up SLIP line[25], may experience an unfortunate interaction between congestion window adjustment and retransmit timing: Network paths tend to divide into two classes: *delay-dominated*, where the store-and-forward and/or transit delays determine the RTT, and *bandwidth-dominated*, where (bottleneck) link bandwidth and average packet size determine the RTT.[20] On a bandwidth-dominated path of bandwidth b, a congestion-avoidance window increment of Δw will increase the RTT of post-increment packets by

---

[17] E.g., see packets 10–50 of figure 5. Note that these window effects are 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).

[18] Although it may not be obvious, the absolute value in the calculation of v introduces an asymmetry in the timer: Because v has the same sign as an increase and the opposite sign of a decrease, more gain in v makes the timer go up quickly and come down slowly, 'automatically' giving the behavior suggested in [21]. E.g., see the region between packets 50 and 80 in figure 6.

[19] For TCP this happens automatically since windows are expressed in bytes, not packets. For protocols such as ISO TP4, the implementor should scale *cwnd* so that the calculations above can be done with integer arithmetic and the scale factor should be large enough to avoid the fixed point (zero) of $\lfloor 1/cwnd \rfloor$ in the congestion avoidance increment.

$$\Delta R \approx \frac{\Delta w}{b}$$

If the path RTT variation V is small, ΔR may exceed the $4V$ cushion in rto, a retransmit timeout will occur and, after a few cycles of this, *ssthresh* (and, thus, *cwnd*) end up clamped at small values.

The rto calculation in appendix A was designed to prevent this type of spurious retransmission timeout during slow-start. In particular, the RTT variation V is multiplied by four in the rto calculation because of the following: A spurious retransmit occurs if the retransmit timeout computed at the end of slow-start round i, rto_i, is ever less than or equal to the actual RTT of the next round. In the worst case of all the delay being due the window, R doubles each round (since the window size doubles). Thus $R_{i+1} = 2R_i$ (where R_i is the measured RTT at slow-start round i). But

$$\begin{aligned} V_i &= R_i - R_{i-1} \\ &= R_i/2 \end{aligned}$$

and

$$\begin{aligned} rto_i &= R_i + 4V_i \\ &= 3R_i \\ &> 2R_i \\ &> R_{i+1} \end{aligned}$$

so spurious retransmit timeouts cannot occur.[21]

Spurious retransmission due to a window increase can occur during the congestion avoidance window increment since the window can only be changed in one packet increments so, for a packet size s, there may be as many as $s - 1$ packets between increments, long enough for any V increase due to the last window increment to decay away to nothing. But this problem is unlikely on a bandwidth-dominated path since the increments would have to be more than twelve packets apart (the decay time of the V filter times its gain in the rto calculation) which implies that a ridiculously large window is being used for the path.[22] Thus one should regard these timeouts as appropriate punishment for gross mis-tuning and their effect will simply be to reduce the window to something more appropriate for the path.

Although slow-start and congestion avoidance are designed to not trigger this kind of spurious retransmission, an interaction with higher level protocols frequently does: Application protocols like SMTP and NNTP have a 'negotiation' phase where a few packets are exchanged stop-and-wait, followed by data transfer phase where all of a mail message or news article is sent. Unfortunately, the 'negotiation' exchanges open the congestion window so the start of the data transfer phase will dump several packets into the network with no slow-start and, on a bandwidth-dominated path, faster than rto can track the RTT increase caused by these packets. The root cause of this problem is the same one described in sec. 1: dumping too many packets into an empty pipe (the pipe is empty since the negotiation exchange was conducted stop-and-wait) with no ack 'clock'. The fix proposed in sec. 1, slow-start, will also prevent this problem if the TCP implementation can detect the phase change. And detection is simple: The pipe is empty because we haven't sent anything for at least a round-trip-time (another way to view RTT is as the time it takes the pipe to empty after the most recent send). So, if nothing has been sent for at least one RTT, the next send should set *cwnd* to one packet to force a slow-start. I.e., if the connection state variable *lastsnd* holds the time the last packet was sent, the following code should appear early in the TCP output routine:

```
int idle = (snd_max == snd_una);
if (idle && now − lastsnd > rto)
        cwnd = 1;
```

The boolean *idle* is true if there is no data in transit (all data sent has been acked) so the **if** says "if there's nothing in transit and we haven't sent anything for 'a long time', slow-start." Our experience has been that either the current RTT estimate or the rto estimate can be used for 'a long time' with good results[23]

---

[20] E.g., TCP over a 2400 baud packet radio link is bandwidth-dominated since the transmission time for a (typical) 576 byte IP packet is 2.4 seconds, longer than any possible terrestrial transit delay.

[21] The original SIGCOMM '88 version of this paper suggested calculating rto as $R+2V$ rather than $R+4V$. Since that time we have had much more experience with low speed SLIP links and observed spurious retransmissions during connection startup. An investigation of why these occured led to the analysis above and the change to the rto calculation in app. A.

[22] The the largest sensible window for a path is the bottleneck bandwidth times the round-trip delay and, by definition, the delay is negligible for a bandwidth-dominated path so the window should only be a few packets.

[23] The rto estimate is more convenient since it is kept in units of time while RTT is scaled. Also, because of send/receive symmetry, the time of the last receive can be used rather than the last send — If the protocol implements 'keepalives', this state variable may already exist.

D Window Adjustment Policy

A reason for using $\frac{1}{2}$ as a the decrease term, as opposed to the $\frac{7}{8}$ in [14], was the following handwaving: 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 (slow-start 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 by 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 a large performance penalty, in system terms the cost is negligible: Currently, packets are dropped only when a large queue has formed. Even with the ISO IP 'congestion experienced' bit [10] to force senders to reduce their windows, we're stuck with the queue because the bottleneck is running at 100% utilization with no excess bandwidth available to dissipate the queue. If a packet is tossed, some sender shuts 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 than 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 [14], we are just to the left of the cliff rather than just to the right of the knee. If the gateways are fixed so they start dropping packets when the queue gets pushed past the knee, our increment will be much too aggressive and should be dropped by about a factor of four (since our measurements on an unloaded Arpanet place its 'pipe size' at 4–5 packets). It appears trivial to implement a second order control loop to adaptively determine the appropriate increment to use for a path. But second order problems are on hold until we've spent some time on the first order part of the algorithm for the gateways.

References

[1] ALDOUS, D. J. Ultimate instability of exponential back-off protocol for acknowledgment based transmission control of random access communication channels. *IEEE Transactions on Information Theory IT-33*, 2 (Mar. 1987).

[2] BORRELLI, R., AND COLEMAN, C. *Differential Equations*. Prentice-Hall Inc., 1987.

[3] CHIU, D.-M., AND JAIN, R. Networks with a connectionless network layer; part iii: Analysis of the increase and decrease algorithms. Tech. Rep. DEC-TR-509, Digital Equipment Corporation, Stanford, CA, Aug. 1987.

[4] CLARK, D. *Window and Acknowlegement Strategy in TCP*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, July 1982. RFC-813.

[5] EDGE, S. W. An adaptive timeout algorithm for retransmission across a packet switching network. In *Proceedings of SIGCOMM '83* (Mar. 1983), ACM.

[6] FELLER, W. *Probability Theory and its Applications*, second ed., vol. II. John Wiley & Sons, 1971.

[7] HAJEK, B. Stochastic approximation methods for decentralized control of multiaccess communications. *IEEE Transactions on Information Theory IT-31*, 2 (Mar. 1985).

[8] HAJEK, B., AND VAN LOON, T. Decentralized dynamic control of a multiaccess broadcast channel. *IEEE Transactions on Automatic Control AC-27*, 3 (June 1982).

[9] *Proceedings of the Sixth Internet Engineering Task Force* (Boston, MA, Apr. 1987). Proceedings available as NIC document IETF-87/2P from DDN Network Information Center, SRI International, Menlo Park, CA.

[10] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO International Standard 8473, Information Processing Systems — Open Systems Interconnection — Connectionless-mode Network Service Protocol Specification*, Mar. 1986.

[11] JACOBSON, V. Congestion avoidance and control. In *Proceedings of SIGCOMM '88* (Stanford, CA, Aug. 1988), ACM.

[12] JAIN, R. Divergence of timeout algorithms for packet retransmissions. In *Proceedings Fifth Annual International Phoenix Conference on Computers and Communications* (Scottsdale, AZ, Mar. 1986).

[13] JAIN, R. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications SAC-4*, 7 (Oct. 1986).

[14] JAIN, R., RAMAKRISHNAN, K., AND CHIU, D.-M. Congestion avoidance in computer networks with a connectionless network layer. Tech. Rep. DEC-TR-506, Digital Equipment Corporation, Aug. 1987.

[15] KARN, P., AND PARTRIDGE, C. Estimating round-trip times in reliable transport protocols. In *Proceedings of SIGCOMM '87* (Aug. 1987), ACM.

[16] KELLY, F. P. Stochastic models of computer communication systems. *Journal of the Royal Statistical Society B 47*, 3 (1985), 379–395.

[17] KLEINROCK, L. *Queueing Systems*, vol. II. John Wiley & Sons, 1976.

[18] KLINE, C. Supercomputers on the Internet: A case study. In *Proceedings of SIGCOMM '87* (Aug. 1987), ACM.

[19] LJUNG, L., AND SODERSTROM, T. *Theory and Practice of Recursive Identification.* MIT Press, 1983.

[20] LUENBERGER, D. G. *Introduction to Dynamic Systems.* John Wiley & Sons, 1979.

[21] MILLS, D. *Internet Delay Experiments.* ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Dec. 1983. RFC-889.

[22] NAGLE, J. *Congestion Control in IP/TCP Internetworks.* ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Jan. 1984. RFC-896.

[23] POSTEL, J., Ed. *Transmission Control Protocol Specification.* SRI International, Menlo Park, CA, Sept. 1981. RFC-793.

[24] PRUE, W., AND POSTEL, J. *Something A Host Could Do with Source Quench.* ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, July 1987. RFC-1016.

[25] ROMKEY, J. *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: Slip.* ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, June 1988. RFC-1055.

[26] ZHANG, L. Why TCP timers don't work well. In *Proceedings of SIGCOMM '86* (Aug. 1986), ACM.