Network Working Group
Request for Comments: 1016

W. Prue
J. Postel
ISI

July 1987

Something a Host Could Do with Source Quench:

The Source Quench Introduced Delay (SQuID)

Status of this Memo

This memo is intended to explore the issue of what a host could do with a source quench. The proposal is for each source host IP module to introduce some delay between datagrams sent to the same destination host. This is an "crazy idea paper" and discussion is essential. Distribution of this memo is unlimited.

Introduction

A gateway may discard Internet datagrams if it does not have the buffer space needed to queue the datagrams for output to the next network on the route to the destination network. If a gateway discards a datagram, it may send a source quench message to the Internet source host of the datagram. A destination host may also send a source quench message if datagrams arrive too fast to be processed. The source quench message is a request to the host to cut back the rate at which it is sending traffic to the Internet destination. The gateway may send a source quench message for every message that it discards. On receipt of a source quench message, the source host should cut back the rate at which it is sending traffic to the specified destination until it no longer receives source quench messages from the gateway. The source host can then gradually increase the rate at which it sends traffic to the destination until it again receives source quench messages [1,2].

The gateway or host may send the source quench message when it approaches its capacity limit rather than waiting until the capacity is exceeded. This means that the data datagram which triggered the source quench message may be delivered.

The SQuID Concept

Suppose the IP module at the datagram source has a queue of datagrams to send, and the IP module has a parameter "D". D is the introduced delay between sending datagrams from the queue to the network. That is, when the IP module discovers a datagram waiting to be sent to the network, it sends it to the network then waits time D before even looking at the datagram queue again. Normally, the value of D is zero.

Imagine that when a source quench is received (or any other signal is received that the host should slow down its transmissions to the network), the value of D is increased. As time goes by, the value of D is decreased.

The SQuID Algorithm

```
        on increase event:

             D <-- maximum (D + K, I)
                                         (where K = .020 second,
                                                I = .075 second)

        on decrease event:

             D <-- maximum (D - J, 0)
                                         (where J = .001 second)
```

An increase event is receipt of one or more source quenches in a event period E, (where E is 2.000 seconds).

A decrease event is when S time has passed since D was decreased and there is a datagram to send (where S is 1.000 seconds).

A cache of D’s is kept for the last M hosts communicated with.

Note that when no datagrams are sent to a destination for some time the D for that destination is not decreased, but, if a destination is not used for a long time that D for that destination may fall out of the cache.

Possible Refinements

Keep a separate outgoing queue of datagrams for each destination host, local subnet, or network.

Keep the cache of D’s per network or local subnet, instead of per host.

"I" could be based upon the basic speed of the slowest intervening network (see Appendix A).

"D" could be limited to never go below "I" if the above refinement were implemented.

"S" could be based upon the round trip time.

"D" could be adjusted datagram by datagram based upon the length of the datagrams. Wait longer after a long datagram.

The delay algorithm could be implemented such that if a source doesn’t send a datagram when it is next allowed (the introduced delay interval) or for N such intervals that the source gets a credit for one and only one free (no delay) datagram.

Implementation Ideas

Since IP does not normally keep much state information about things, we want the default or idle IP to have no state about these D values. Since the default D value is zero, let us propose that the IP will keep a list of only those destinations with non zero D’s.

When the IP wants to send a datagram, it searches the D-list to see if the destination is noted. If it is not, the D value is zero, so the IP sends the datagram at once. If the destination is listed, the IP must wait D time indicated before sending that particular datagram. It could look at a datagram addressed to a different destination, and possibly send it in the mean time.

When the IP receives a source quench, it checks to see if the destination in the datagram that caused the source quench is on the list. If so, it adds K to the D value. If not, it appends the destination to the list with the D value set to "I".

A Closer Look At the Problem

Some implementations of IP send one SQ for every N datagrams they discard (for example, N=20) so the SQ messages will not make the congestion problem much worse [3]. In such situations any of the sources of the 20 datagrams may get the SQ not necessarily the one causing the most traffic. However if a host continues to send datagrams at a high rate it has a high probability of receiving a SQ message sooner or later. It is much like a speeder on a highway. Not all speeders get speeding tickets but the ones who speed most often or most excessively are most likely to be ticketed. In this case they will get a ticket and their car may be destroyed.

With memory becoming so inexpensive many IP nodes put an artificially low limit on the size of their queues so that through node delay will not be excessive [4]. For example, if one megabyte of data is buffered to be sent over a 56 kb/s line the last datagram will wait over 2 minutes before being sent.

One problem with SQ is that the IP or ICMP specification does not have a well defined event to indicate receipt of SQ to higher level protocols. Therefore many TCP implementations do not get notified about SQ events and thus do not react to SQ. TCP is not the only source of IP datagrams either. Other protocols should also respond to SQ events in some appropriate way. TCP and other protocols at that level should do something about a source quench, however, discussion of their behavior is beyond the scope of this memo. Note that implementation of SQ processing at one level of protocol should not interfere with the behavior of higher level protocols. This however, is difficult to do.

For protocols using IP which are trying to transfer large amounts of data the data flow is most typically very bursty. TCP for example, might send 5-10 segments into a window of 5-10 K bytes then wait for the acknowledgment of the data which opens the window again. NETBLT as defined by RFC-998 is a rate based protocol which has parameters for burst size and burst rate.

One purpose of the bursts is to allow the source computer to generate several datagrams at once to provide more efficient scheduling. An other reason is to keep the network busy accepting data to maximize effective throughput in spite of a potentially large network round trip delay. To send a datagram then wait for an acknowledgment is a simple but not efficient protocol on a large wide area network.

The reasons for efficiencies obtained at the source node by generating many datagrams at once are not as applicable in an intermediate IP node. Since each datagram is potentially from a different node they must all be treated individually. Datagrams received in a burst may also overload the queue of an intermediate node losing datagrams and causing SQs to be generated. If the queue is near a threshold and a burst comes, possibly all of the datagrams will be lost. When datagrams arrive evenly spaced, less datagrams are likely to be lost because the inter-arrival time allows the queue a little time to empty out. Therefore datagrams spaced with some delay between them may be better for intermediate IP nodes.

Congestion is most likely to occur at IP nodes which are gateways between a slower network and a faster one. The congestion will be in the send queue from the slow network to the fast network. An SQ being returned to the sender will return on the faster network. (See diagram below.)

A Gateway Source Quench Concept

In order for the SQuID algorithm to work we rely upon the gateways to send SQs to us to tell us how we are doing. Because the loss of a single datagram affects data flow so much (see lost datagram discussion in Observed Results below) it would be much better for the source IP node if it got a warning before datagrams were discarded.

We propose gateway IP nodes start SQing before the node is flooded at a level we call SQ Keep (SQK) but forward the datagram. If the queue level reaches a critical level, SQ Toss level (SQT), the gateway should toss datagrams to resolve the problem unless the datagram is an ICMP message. Even ICMP messages will be tossed if the MaxQ level is reached. Once the gateway starts sending SQs it should continue to do so until the queue level goes below a low water mark level (SQLW) as shown below. This is analogous to methods some operating systems use to handle memory space management.

The gateway should try to send SQ to as many of the contributors of the congestion as possible but only once per contributor per second or two.

Source Quench Queue Levels

```
      +--------------+ MaxQ level
      |              |> datagrams tossed & SQed (but not ICMP msgs.)
      +--------------+ SQT level (95%)
      |              |\
      |              | > datagrams SQed but forwarded
      |              |/
      +--------------+ SQK level (70%)
      |              |\
      |              | \ datagrams SQed but forwarded if SQK level
      |              | / exceeded & SQLW or lower not yet reached
      |              |/
      +--------------+ SQLW level (50%)
      |              |\
      |              | \
      |              |  \
      |              |   \ datagrams forwarded
      |              |   /
      |              |  /
      |              | /
      |              |/
      +--------------+
```

Description of the Test Model

We needed some way of testing our algorithm and its various parameters. It was important to check the interaction between IP with the SQuID algorithm and TCP. We also wanted to try various combinations of retransmission strategy and source quench strategy which required control of the entire test network. We therefore decided to build an Internet model.

Using this example configuration for illustration:

```
 _______    LAN     _______    WAN     _______    LAN     _______
|   1   |            |   2   |            |   3   |            |   4   |
|TCP/IP |---10 Mb/s--|  IP   |---56 kb/s--|  IP   |---10 Mb/s--|TCP/IP |
|_______|            |_______|            |_______|            |_______|
```

A program was written in C which created queues and structures to put on the queues representing datagrams carrying data, acknowledgments and SQs. The program moved datagrams from one queue to the next based upon rules defined below

A client fed the TCP in node 1 data at the rate it would accept. The TCP function in node 1 would chop the data up into fixed 512 byte datagrams for transmission to the IP in node 1. When the datagrams were given to IP for transmission, a timestamp was put on it and a copy of it was put on a TCP ack-wait queue (data sent but not yet acknowledged). In particular TCP assumed that once it handed data to IP, the data was sent immediately for purposes of retransmission timeouts even though our algorithm has IP add delay before transmission.

Each IP node had one queue in each direction (left and right). For each IP in the model IP would forward datagrams at the rate of the communications line going to the next node. Thus the fifth datagram on IP 2’s queue going right would take 5 X 73 msec or 365 msec before it would appear at the end of IP 3’s queue. The time to process each datagram was considered to be less than the time it took for the data to be sent over the 56 kb/s lines and therefore done during those transmission times and not included in the model. For the LAN communications this is not the case but since they were not at the bottleneck of the path this processing time was ignored. However because LAN communications are typically shared band width, the LAN band width available to each IP instance was considered to be 1 Mb/s, a crude approximation.

When the data arrived at node 4 the data was immediately given to the TCP receive function which validated the sequence number. If the datagram was in sequence the datagram was turned into an ack datagram and sent back to the source. An ack datagram carries no data and will move the right edge of the window, the window size past the just acked data sequence number. The ack datagram is assumed to be 1/8 of the length of a data datagram and thus can be transmitted from one node to the next 8 times faster. If the sequence number is less than expected (a retransmission due to a missed ack) then it too is turned into an ack. A larger sequence number datagram is queued indefinitely until the missing datagrams are received.

We also modeled the gateway source quench algorithm. When a datagram was put on an IP queue the number on the queue was compared to an SQ keep level (SQK). If it was greater, an SQ was generated and returned to the sender. If it was larger than the SQ toss (SQT) level it was also discarded. Once SQs were generated they would continue to be sent until the queue level went below SQ Low Water (SQLW) level which was below the original SQK level. These percentages were modifiable as were many parameters. An SQ could be lost if it exceeded the maximum queue size (MaxQ), but a source quench was never sent about tossing a source quench.

Upon each transition from one node to the next, the datagram was vulnerable to datagram loss due to errors. The loss rate could be set as M losses out of N datagrams sent, thus the model allowed for multi-datagram loss bursts or single datagram losses. We used a single datagram loss rate of 1 lost datagram per 300 datagrams sent for much of our testing. While this may seem low for Internet simulation, remember it does not include losses due to congestion.

Some network parameters we used were a maximum queue length of 15 datagrams per IP direction left and right. We started sending SQ if the queue was 70% full, SQK level, tossed data datagrams, but not SQ datagrams, if 95% of the queue was reached, SQT level, and stopped SQing when a 50% SQLW level was reached (see above).

We ignored additional SQs for 2 seconds after receipt of one SQ. This was done because some Internet nodes only send one SQ for every 20 datagrams they discard even though our model sent SQs for every datagram discarded. Other IP node may send one SQ per discarded packet. The SQuID algorithm needed a way to handle both types of SQ generation. We therefore treated one or a burst of SQs as a single event and incremented our D by a larger amount than would be appropriate for responding individually to the multiple SQs of the verbose nodes.

The simulation did not do any fragmenting of datagrams. Silly window syndrome was avoided. The model did not implement nor simulate the TTL (time-to-live) function.

The model allowed for a flexible topology definition with many TCP source/destination pairs on host IP nodes or gateway IP nodes with various windows allowed. An IP node could have any number of TCPs assigned to it. Each line could have an individually set speed. Any TCP could send to any other TCP. The routing from one location to another was fixed. Therefore datagrams did not arrive out of sequence. However, datagrams arrived in ascending order, but not consecutively, on a regular basis because of datagram losses. Datagrams going "left" through a node did not affect the queue size, or SQ chances, of data going "right" through the node.

The TCP retransmission timer algorithm used an Alpha of .15 and a Beta of 1.5. The test was run without the benefit of the more sophisticated retransmission timer algorithm proposed by Van Jacobson [5].

The program would display either the queue sizes of the various IP nodes and the TCP under test as time passed or do a crude plot of various parameters of interest including SRTT, perceived round trip time, throughput, and the critical queue size.

As we observed the effects of various algorithms for responding to SQ we adapted our model to better react to SQ. Initial tests showed if we incremented slowly and decremented quickly we observed oscillations around the correct value but more of the time was spent over driving the network, thus losing datagrams, than at a value which helped the congestion situation.

A significant problem is the delay between when some intermediate node starts dropping datagrams and sending source quenches to the time when the source quenches arrive at the source host and can begin to effect the behavior at the data source. Because of this and the possibility that a IP might send only one SQ for each 20 datagrams lost, we decided that the increase in D per source quench should be substantial (for example, D should increase by 20 msec for every source quench), and the decrease with time should be very slow (for example, D should decrease 1 msec every second). Note that this is the opposite behavior than suggested in an early draft by one of the authors.

However, when many source quenches are received (for example, when a source quench is received for every datagram dropped) in a short time period the D value is increased excessively. To prevent D from growing too large, we decided to ignore subsequent source quenches for a time (for example, 2 seconds) once we had increased D.

Tests were run with only one TCP sending data to learn as much as possible how an unperturbed session might run. Other test runs would introduce and eliminate competing traffic dynamically between other TCP instances on the various nodes to see how the algorithms reacted to changes in network load. A potential flaw in the model is that the defined TCPs with open windows always tried to forward data. Their clients feeding them data never paused to think what they were going to type nor got swapped out in favor of other applications nor turned the session around logically to listen to the other end for more user commands. In other words all of the simulated TCP sessions were doing file transfers.

The model was defined to allow many mixes of competing algorithms for responding to SQ. It allowed comparing effective throughput between TCPs with small windows and large windows and those whose IP would introduce inter-datagram delays and those who totally ignored SQ. It also allowed comparisons with various inter-datagram increment amounts and decrement amounts. Because of the number of possible configurations and parameter combinations only a few combinations of parameters were tested. It is hoped they were the most appropriate ones upon which to concentrate.

Observed Results

All of our algorithms oscillate, some worse than others.

If we put in just the right amount of introduced delay we seem to get the best throughput. But finding the right amount is not easy.

Throughput is adversely affected, heavily, by a single lost datagram at least for the short time. Examine what happens when a window is 35 datagrams wide with an average round trip delay of 2500 msec using 512 byte datagrams when a single datagram is lost at the beginning. Thirty five datagrams are given by TCP to IP and a timer is started on the first datagram. Since the first datagram is missing, the receiving TCP will not sent an acknowledgment but will buffer all 34 of the out-of-sequence datagrams. After 1.5 X 2500 msec, or 3750 msec, have elapsed the datagram times out and is resent. It arrives and is acked, along with the other 34, 2500 msec later. Before the lost datagram we might have been sending at the average rate a 56 kb/s line could accept, about one every 75 msec. After loss of the datagram we send at the rate of one in 6250 msec over 83 times slower.

If the lost datagram in the above example is other than the first datagram the situation becomes the same when all of the datagrams before the lost datagram are acknowledged. The example holds true then for any single lost datagram in the window.

When SQ doesn’t always cause datagram loss the sender continues to send too fast (queue size oscillates a lot). It is important for the SQ to cause feed-back into the sending system as soon as possible, therefore when the source host IP receives an SQ it must make adjustments to the send rate for the datagrams still on the send queue not just datagrams IP is requested to send after the SQ.

Through network delay goes up as the network queue lengths go up.

Window size affect the chance of getting SQed. Look at our model above using a queue level of 15 for node 2 before SQs are generated and a window size of 20 datagrams. We assumed that we could send data over the LAN at a sustained average rate of 1 Mb/s or about 18 times as fast as over the WAN. When TCP sends a burst of 20 datagrams to node 1 they make it to node 2 in 81 msec. The transition time from node 2 to node 3 is 73 msec, therefore, in 81 msec, only one datagram is forwarded to node 3. Thus the 17th, 18th, 19th, and 20th datagram are lost every time we send a whole window. More are lost when the queue is not empty. If a sequence of acks come back in response to the sent data, the acks tend to return at the rate at which data can traverse the net thus pacing new send data by opening the window at the rate which the network can accept it. However as soon as one datagram is lost all of the subsequent acks are deferred and batched until receipt of the missing data block which acks all of the datagrams and opens the window to 20 again. This causes the max queue size to be exceeded again.

If we assume a window smaller than the max queue size in the bottleneck node, any time we send a window's worth of data, it is done independently of the current size of the queue. The larger the send window, the larger a percentage of the stressed queue we send. If we send 50% of the stressed queue size any time that queue is more than 50% we threaten to overflow the queue. Evenly spaced single datagram bursts have the least chance of overflowing the queue since they represent the minimum percentage of the max queue one may send.

When a big window opens up (that is, a missing datagram at the head of a 40 datagram send queue gets retransmitted and acked), the perceived round trip time for datagrams subsequently sent hits a minimum value then goes up linearly. The SRTT goes down then back up in a nice smooth curve. This is caused by the fact IP will not add delay if the queue is empty and IP has not sent any datagrams to the destination for our introduced delay time. But as many datagrams are added to the IP pre-staged send queue in bursts all have the same send time as far as TCP is concerned. IP will delay each datagram on the head of the queue by the introduced delay amount. The first may be undelayed as just described but all of the others are delayed by their ordinal number on the queue times the introduced delay amount.

It seems as though in a race between a TCP session which delays sending to IP and one who does not, the delayer will get better throughput because less datagrams are lost. The send window may also be increased to keep the pipeline full. If however the non delayer uses windowing to reduce the chance of SQ datagram loss his throughput may possibly be better because no fair queuing algorithm is in place.

If gateways send SQs early and don't toss data until its critical and keep sending SQs until a low water mark is hit, effective throughput seems to go up.

At the startup of our tests throughput was very high, then dropped off quickly as the last of the window got clobbered. Our model should have used a slow start up algorithm to minimize the startup shock. However the learning curve to estimate the proper value for D was probably quicker.

A large part of the perceived RTT is due to the delay getting off the TCP2IP (TCP transitional) queue when we used large windows. If IP would invoke some back-pressure to TCP in a real implementation this can be significantly reduced. Reducing the window would do this for us at the expense of throughput.

After an SQ burst which tosses datagrams the sender gets in a mode where TCP may only send one or two datagrams per RTT until the queued but not acked segments fall into sequence and are acked. This assumes only the head of the retransmission queue is retransmitted on a timeout. We can send one datagram upon timeout. When the ack for the retransmission is received the window opens allowing sending a second. We then wait for the next lost datagram to time out.

If we stop sending data for a while but allow D to be decreased, our algorithm causes the introduced delay to dwindle away. We would thus go through a new startup learning curve and network oscillation sequence.

One thing not observed often was TCP timing out a segment before the source IP even sent the datagram the first time. As discussed above the first datagram on the queue of a large burst is delayed minimally and succeeding datagrams have linearly increasing delays. The smoothed round trip delay algorithm has a chance to adapt to the perceived increasing round trip times.

Unstructured Thoughts and Comments

The further down a route a datagram traverses before being clobbered the greater the waste of network resources. SQs which do not destroy the datagram referred to are better than ones that do if return path resources are available.

Any fix must be implementable piecemeal. A fix can not be installed in all or most nodes at one time. The SQuID algorithm fulfills this requirement. It could be implemented, installed in one location, and used effectively.

If it can be shown that by using the new algorithm effective throughput can be increased over implementations which do not implement it that may well be effective impetus to get vendors to implement it.

Once a source host has an established average minimum inter-datagram delay to a destination (see Appendix A), this information should be stored across system restarts. This value might be used each time data is sent to the given host as a minimum inter-datagram delay value.

Window closing algorithms reduce the average inter-datagram delay and the burst size but do not affect the minimum inter-datagram spacing by TCP.

Currently an IP gateway node can know if it is in a critical path because its queues stay high or keep building up. Its optimum queue size is one because it always has something to do and the through node delay is at a minimum. It is very important that the gateway at the critical path not so discourage data flow that its queue size drops to zero. If the gateway tosses datagrams this stops data flow for TCP for a while (as described in Observed Results above). This argues for the gateway algorithm described above which SQs but does not toss datagrams unless necessary. Optimally we should try to have a queue size somewhat larger than 1 but less than say 50% of the max queue size. Large queues lead to large delay.

TCP’s SRTT is made artificially large by introducing delay at IP but the perceived round trip time variance is probably smaller allowing a smaller Beta value for the timeout value.

So that a decrease timer is not needed for the "D" decrease function, upon the next sent datagram to a delayed destination just decrease the delay by the amount of time since we last did this divided by the decrease timer interval. An alternate algorithm would be to decrease it by only one decrease unit amount if more than the timer interval has gone by. This eliminates the problem caused by the delay, "D", dwindling away if we stop sending for a while. The longer we send using this "D", the more likely it is that it is too large a delay and the more we should decrease it.

It is better for the network and the sender for our introduced delay to be a little on the high side. It minimizes the chances of getting a datagram clobbered by sending it into a congested gateway. A lost datagram scenario described above showed that one lost datagram can reduce our effective delay by one to two orders of magnitude temporarily. Also if the delay is a little high, the net is less stressed and the queues get smaller, reducing through network delay.

The RTT experienced at a given time verses the minimum RTT possible for the given route does give a good measure of congestion. If we ever get congestion control working RTT may have little to do with the amount of congestion. Effective throughput when compared with the possible throughput (or some other measure) is the only real measure of congestion.

Slow startup of TCP is a good thing and should be encouraged as an additional mechanism for alleviating network overload.

The network dynamics tends to bunch datagrams. If we properly space data instead of bunching it like windowing techniques to control overflow of queues then greater throughput is accomplished because the absolute rate we can send is pacing our sending not the RTT. We eliminate "stochastic bunching" [6].

The longer the RTT the more network resources the data takes to traverse the net.

Should "fair queuing" say that a longer route data transfer should get less band width than a shorter one (since it consumes more of the net)? Being fair locally on each node may be unfair overall to datagrams traversing many nodes.

If we solve congestion problems today, we will start loading up the net with more data tomorrow. When this causes congestion in a year will that type of congestion be harder to solve than todays or is it not our problem? John Nagle suggests "In a large net, we may well try to force congestion out to the fringes and keep the interior of the net uncongested by controlling entry to the net. The IMP-based systems work that way, or at least used to. This has the effect of concentrating congestion at the entrance to the long-haul system. That’s where we want it; the Source Quench / congestion window / fair queuing set of strategies are able to handle congestion at the LAN to WAN bottleneck [7]. Our algorithm should try to push the network congestion out to the extremities and keep the interior network congestion free.

Use of the algorithm is aesthetically appealing because the data is sitting in our local queue instead of consuming resources inside the net. We give data to the network only when it is ready to accept it.

An averaged minimum inter-datagram arrival value will give a measure of the network bottleneck speed at the receiver. If the receiver does not defer or batch together acks the same would be learned from the inter-datagram arrival time of the acks. A problem is that IP doesn’t have knowledge of the datagram contents. However IP does know from which host a datagram comes.

If SQuID limits the size of its pre-net buffering properly (causes back-pressure to TCP) then artificially high RTT measurements would not occur.

TCP might, in the future, get a way to query IP for the current introduced delay, D, for a given destination and if the value is too excessive abort or not start a session.

With the new algorithm TCP could have an arbitrarily large window to send into without fear of over running queue sizes in intermediate nodes (not that any TCP ever considered having this fear before). Thus it could have a window size which would allow it to always be sending; keeping the pipe full and seldom getting in the stop-and-wait mode of sending. This presupposes that the local IP is able to cause some sort of back pressure so that the local IPs queues are not over run. TCP would still be operating in the burst mode of sending but the local IP would be sending a datagram for the TCP as often as the network could accept it keeping the data flow continuous though potentially slow.

Experience implementing protocols suggests avoiding timers in protocols whenever possible. IP, as currently defined, does not use timers. The SQuID algorithm uses two at the IP level. A way to eliminate the introduced delay decrease timer is to decrease the D value when we check the send queue for data to send. We would decrease "D" by one "J" unit if "S" time, or more, has elapsed. The other timer is not so easily eliminated. If the IP implementation is periodically awakened to check for work to do, it could check the time stamps of the head of the queues to see if any datagrams have waited long enough. This would mean we would necessarily wait too long before sending, but it may not be too large of a variance from our desired rates. The additional delay would help congestion and reduce our chances of SQ. Another option is setting a real timer which is say 25-50% too large and hope that our periodic work in IP will allow us to notice a datagram is delayed enough, and send it. Upon sending the datagram we would cancel the timer, avoiding the timer interrupt and environment swap. In other implementations the communications interface or the link level protocol may be able to provide the timing needed without interrupts to the main processor.

Background tasks like some file transfers could query IP for the latest delay characteristics for a given destination to determine if it is appropriate to consume network resources now or if it would be better to wait until later.

We should consider what would happen if IP, using the SQuID algorithm, and TCP both introduced delay between the datagrams. If TCPs delay was greater than IP’s, then when IP got the datagrams it would forward them immediately as described in the algorithm above. This is because when the IP send queue is empty and it has been at least as long as IP wants the higher level protocol, TCP, gets one free (no delay) send. Note also that IP will be decreasing the amount of delay it wants introduced because of the successful transmissions without SQs. This would affect other protocols who might also send to the same destination. If TCP sends too quickly then IP will protect the network from its indiscretion as described in the basic algorithm however TCPs round trip time estimates will be much closer to reality. A lost datagram will thus be detected more quickly. If TCP also uses windowing to limit its sending rate, it might, because of its success with a smaller window, increase the window size increasing TCPs efficiency.

As this algorithm is implemented everywhere, the SQ Keep (SQK) and SQ Low Water (SQLW) queue level percentages should be dropped to reduce queue sizes and thus the through net delay. The percentage we lower SQK and SQLW to should be adjusted based upon the percentage of time the queue is empty. The more the queue is empty the more likely it is that the node is discouraging data flow too much. The more time the queue is not empty but not too full, the more likely it is the node is not excessively discouraging data flow. How uniform the queue size is, is a measure of how well the network citizens are behaved.

As the congestion is pushed to the sources, gateways which are bottlenecks can more easily detect someone not playing by the rules (sending datagrams in bursts) and deal with the offender.

Appendix A -- Determination of the Value for the Parameter "I"

To get to the correct value for the delay needed quickly, when an event occurred and the currently used delay was minimal, the transmission time for an average sized datagram across the slowest communications link was used for a first value. How a real IP node is to guess this value is discussed below. In our example the transition between node 2 and node 3 is the bottleneck. Using the 56 kb/s line, a 512 byte datagram would take 73 msec with no queuing or processing time is considered. This value is defined to be the minimum inter-datagram arrival time (MIAT). Assuming a perfect network, ignoring factors other than transmission speed, this is the minimum time one could expect between receipt of datagrams at the destination, because of the slowed data rate through the bottleneck. This won't hold true if the datagrams do not follow the same path.

The MIAT, minimum inter-datagram arrival time, may be guessed at by comparing the arrival timestamps of consecutive datagrams from a source of data. Each value to be considered needs to be adjusted up or down based upon the size between the second datagram received and the typical datagram size. More simply stated, a datagram which is half the size of the average datagram can be transmitted across a line in half the time. Therefore, double its IAT before considering it for an MIAT. If the timestamp of a datagrams is taken based upon an event caused by the start of the datagram arriving, not the completion of the datagram arriving, then the first datagram's size is the limiting length and should be used to adjust its IAT. In order to keep the algorithm simple, arrival times for short datagram could be ignored as could IATs which were orders of magnitude too large (see below).

Once a minimal value is found based upon looking for small values over a minute or more, the value might be time averaged with a value kept like TCP's SRTT in order to reduce the effects of a false MIAT. We could assume the limiting facility would be a 9.6 kb/s line, a 56-64 kb/s line, or a 1.5 Mb/s line. These facilities would provide a MIAT of 427 msec, 73-64 msec, or 3 msec respectively, for a datagram 512 bytes long. These are almost orders of magnitude in differences. If the MIAT a node measures is not in this range but close, the value it is closest to may be used for its MIAT from that source.

One of the good things about this measurement is that it is an entirely passive measurement. No additional traffic is needed to measure it. If a source is not sending data continuously then the longer measured values won't be validated as minimal values. The assumption is that at least sometimes the source will send multiple datagrams at a time.

The MIAT measurement is totally unaffected by satellite delay characteristics of any intervening facilities. The chance of getting a valid minimal reading is affected by the number of nodes traversed, but the value measured if it is valid will not be affected by the number of nodes traversed. Stated another way, when a pair of datagrams traverse from one node to the next the datagrams are susceptible to being separated by a datagram from another source. Both of these factors affect SRTT. The value obtained requires no topological knowledge of the route.

A potential problem with the measurement is, it will not be the proper value for some forms of alternate routes. If a T1 link is the bottleneck route some times and other times it is a 56 kb/s link our first guess for inter-datagram delay would be too small for the 56 kb/s line route. Another problem is that if one datagram goes via one route and the next goes via another, their inter-datagram arrival difference could lead to a small false measurement. If intervening networks fragment datagrams then the measured IAT between segments could be misleading. A solution to this problem is to ignore fragmented datagram IATs.

This number represents the minimum inter-datagram delay the sending IP should use to send to us, the measuring site, for the given datagram size. If we assume that the return path will be through the same facilities or the same type, then as described above this value can be the first guess for inter-datagram introduced delay, "D" (in the algorithm). It represents the "I" parameter.

These MIATs may be cached on a host, subnet, or network basis. The last "n" hosts MIATs could be kept. For infrequent destinations an entry per destination network would be applicable to many destinations. If the local net is in fact a subnet, then the other local subnet MIATs could be kept.

If a good algorithm is found for MIAT, comparing it to the average IAT (during data transfer) would give a good measure of the amount of network traffic load. Since IP has no idea when the source of data is sending as fast as possible, to get a valid average, upper layer protocols would have to figure this out for themselves. IP could however provide an interface to get the current MIAT for a given destination.

References

[1] Postel, Jon, "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, ISI, September 1981.

[2] Postel, Jon, "Internet Control Message Protocol - DARPA Internet Program Protocol Specification", RFC 792, ISI, September 1981.

[3] Karels, M., "Re: Source Quench", electronic mail message to J. Postel and INENG-INTEREST, Tue, 24 Feb 87.

[4] Nagle, John B., "On Packet Switches With Infinite Storage", RFC 970, FACC Palo Alto, December 1985.

[5] Jacobson, Van, "Re: interpacket arrival variance and mean", electronic mail message to TCP-IP, Mon, 15 Jun 87 06:08:01 PDT

[6] Jacobson, Van, "Re: Appropriate measures of gateway performance" electronic mail message to J. Noel Chiappa and INENG-INTEREST, Sun, 22 Mar 87 15:04:44 PST.

[7] Nagle, John B., "Source quench, and congestion generally", electronic mail message to B. Braden and INENG-INTEREST, Thu, 5 Mar 87 11:08:49 PST.

[8] Nagle, John B., "Congestion Control in IP/TCP Internetworks", RFC 896, FACC Palo Alto, 6 January 1984.