RFC: 813

WINDOW AND ACKNOWLEDGEMENT STRATEGY IN TCP

David D. Clark
MIT Laboratory for Computer Science
Computer Systems and Communications Group
July, 1982

1. Introduction

This document describes implementation strategies to deal with two mechanisms in TCP, the window and the acknowledgement. These mechanisms are described in the specification document, but it is possible, while complying with the specification, to produce implementations which yield very bad performance. Happily, the pitfalls possible in window and acknowledgement strategies are very easy to avoid.

It is a much more difficult exercise to verify the performance of a specification than the correctness. Certainly, we have less experience in this area, and we certainly lack any useful formal technique. Nonetheless, it is important to attempt a specification in this area, because different implementors might otherwise choose superficially reasonable algorithms which interact poorly with each other. This document presents a particular set of algorithms which have received testing in the field, and which appear to work properly with each other. With more experience, these algorithms may become part of the formal specification: until such time their use is recommended.

2. The Mechanisms

The acknowledgement mechanism is at the heart of TCP. Very simply, when data arrives at the recipient, the protocol requires that it send back an acknowledgement of this data. The protocol specifies that the bytes of data are sequentially numbered, so that the recipient can acknowledge data by naming the highest numbered byte of data it has received, which also acknowledges the previous bytes (actually, it identifies the first byte of data which it has not yet received, but this is a small detail). The protocol contains only a general assertion that data should be acknowledged promptly, but gives no more specific indication as to how quickly an acknowledgement must be sent, or how much data should be acknowledged in each separate acknowledgement.

The window mechanism is a flow control tool. Whenever appropriate, the recipient of data returns to the sender a number, which is (more or less) the size of the buffer which the receiver currently has available for additional data. This number of bytes, called the window, is the maximum which the sender is permitted to transmit until the receiver returns some additional window. Sometimes, the receiver will have no buffer space available, and will return a window value of zero. Under these circumstances,the protocol requires the sender to send a small segment to the receiver now and then, to see if more data is accepted. If the window remains closed at zero for some substantial period, and the sender can obtain no response from the receiver, the protocol requires the sender to conclude that the receiver has failed, and to close the connection. Again, there is very little performance

information in the specification, describing under what circumstances the window should be increased, and how the sender should respond to such revised information.

A bad implementation of the window algorithm can lead to extremely poor performance overall. The degradations which occur in throughput and CPU utilizations can easily be several factors of ten, not just a fractional increase. This particular phenomenon is specific enough that it has been given the name of Silly Window Syndrome, or SWS. Happily SWS is easy to avoid if a few simple rules are observed. The most important function of this memo is to describe SWS, so that implementors will understand the general nature of the problem, and to describe algorithms which will prevent its occurrence. This document also describes performance enhancing algorithms which relate to acknowledgement, and discusses the way acknowledgement and window algorithms interact as part of SWS.

3. SILLY WINDOW SYNDROME

In order to understand SWS, we must first define two new terms. Superficially, the window mechanism is very simple: there is a number, called "the window", which is returned from the receiver to the sender. However, we must have a more detailed way of talking about the meaning of this number. The receiver of data computes a value which we will call the "offered window". In a simple case, the offered window corresponds to the amount of buffer space available in the receiver. This correspondence is not necessarily exact, but is a suitable model for the discussion to follow. It is the offered window which is

actually transmitted back from the receiver to the sender. The sender uses the offered window to compute a different value, the "usable window", which is the offered window minus the amount of outstanding unacknowledged data. The usable window is less than or equal to the offered window, and can be much smaller.

Consider the following simple example. The receiver initially provides an offered window of 1,000. The sender uses up this window by sending five segments of 200 bytes each. The receiver, on processing the first of these segments, returns an acknowledgement which also contains an updated window value. Let us assume that the receiver of the data has removed the first 200 bytes from the buffer, so that the receiver once again has 1,000 bytes of available buffer. Therefore, the receiver would return, as before, an offered window of 1,000 bytes. The sender, on receipt of this first acknowledgement, now computes the additional number of bytes which may be sent. In fact, of the 1,000 bytes which the recipient is prepared to receive at this time, 800 are already in transit, having been sent in response to the previous offered window. In this case, the usable window is only 200 bytes.

Let us now consider how SWS arises. To continue the previous example, assume that at some point, when the sender computes a useable window of 200 bytes, it has only 50 bytes to send until it reaches a "push" point. It thus sends 50 bytes in one segment, and 150 bytes in the next segment. Sometime later, this 50-byte segment will arrive at the recipient, which will process and remove the 50 bytes and once again return an offered window of 1,000 bytes. However, the sender will now

compute that there are 950 bytes in transit in the network, so that the useable window is now only 50 bytes. Thus, the sender will once again send a 50 byte segment, even though there is no longer a natural boundary to force it.

In fact, whenever the acknowledgement of a small segment comes back, the useable window associated with that acknowledgement will cause another segment of the same small size to be sent, until some abnormality breaks the pattern. It is easy to see how small segments arise, because natural boundaries in the data occasionally cause the sender to take a computed useable window and divide it up between two segments. Once that division has occurred, there is no natural way for those useable window allocations to be recombined; thus the breaking up of the useable window into small pieces will persist.

Thus, SWS is a degeneration in the throughput which develops over time, during a long data transfer. If the sender ever stops, as for example when it runs out of data to send, the receiver will eventually acknowledge all the outstanding data, so that the useable window computed by the sender will equal the full offered window of the receiver. At this point the situation will have healed, and further data transmission over the link will occur efficiently. However, in large file transfers, which occur without interruption, SWS can cause appalling performance. The network between the sender and the receiver becomes clogged with many small segments, and an equal number of acknowledgements, which in turn causes lost segments, which triggers massive retransmission. Bad cases of SWS have been seen in which the

average segment size was one-tenth of the size the sender and receiver were prepared to deal with, and the average number of retransmission per successful segments sent was five.

Happily, SWS is trivial to avoid. The following sections describe two algorithms, one executed by the sender, and one by the receiver, which appear to eliminate SWS completely. Actually, either algorithm by itself is sufficient to prevent SWS, and thus protect a host from a foreign implementation which has failed to deal properly with this problem. The two algorithms taken together produce an additional reduction in CPU consumption, observed in practice to be as high as a factor of four.

4. Improved Window Algorithms

The receiver of data can take a very simple step to eliminate SWS. When it disposes of a small amount of data, it can artificially reduce the offered window in subsequent acknowledgements, so that the useable window computed by the sender does not permit the sending of any further data. At some later time, when the receiver has processed a substantially larger amount of incoming data, the artificial limitation on the offered window can be removed all at once, so that the sender computes a sudden large jump rather than a sequence of small jumps in the useable window.

At this level, the algorithm is quite simple, but in order to determine exactly when the window should be opened up again, it is necessary to look at some of the other details of the implementation.

Depending on whether the window is held artificially closed for a short or long time, two problems will develop. The one we have already discussed -- never closing the window artificially -- will lead to SWS. On the other hand, if the window is only opened infrequently, the pipeline of data in the network between the sender and the receiver may have emptied out while the sender was being held off, so that a delay is introduced before additional data arrives from the sender. This delay does reduce throughput, but it does not consume network resources or CPU resources in the process, as does SWS. Thus, it is in this direction that one ought to overcompensate. For a simple implementation, a rule of thumb that seems to work in practice is to artificially reduce the offered window until the reduction constitutes one half of the available space, at which point increase the window to advertise the entire space again. In any event, one ought to make the chunk by which the window is opened at least permit one reasonably large segment. (If the receiver is so short of buffers that it can never advertise a large enough buffer to permit at least one large segment, it is hopeless to expect any sort of high throughput.)

There is an algorithm that the sender can use to achieve the same effect described above: a very simple and elegant rule first described by Michael Greenwald at MIT. The sender of the data uses the offered window to compute a useable window, and then compares the useable window to the offered window, and refrains from sending anything if the ratio of useable to offered is less than a certain fraction. Clearly, if the computed useable window is small compared to the offered window, this means that a substantial amount of previously sent information is still

in the pipeline from the sender to the receiver, which in turn means that the sender can count on being granted a larger useable window in the future. Until the useable window reaches a certain amount, the sender should simply refuse to send anything.

Simple experiments suggest that the exact value of the ratio is not very important, but that a value of about 25 percent is sufficient to avoid SWS and achieve reasonable throughput, even for machines with a small offered window. An additional enhancement which might help throughput would be to attempt to hold off sending until one can send a maximum size segment. Another enhancement would be to send anyway, even if the ratio is small, if the useable window is sufficient to hold the data available up to the next "push point".

This algorithm at the sender end is very simple. Notice that it is not necessary to set a timer to protect against protocol lockup when postponing the send operation. Further acknowledgements, as they arrive, will inevitably change the ratio of offered to useable window. (To see this, note that when all the data in the catanet pipeline has arrived at the receiver, the resulting acknowledgement must yield an offered window and useable window that equal each other.) If the expected acknowledgements do not arrive, the retransmission mechanism will come into play to assure that something finally happens. Thus, to add this algorithm to an existing TCP implementation usually requires one line of code. As part of the send algorithm it is already necessary to compute the useable window from the offered window. It is a simple matter to add a line of code which, if the ratio is less than a certain

percent, sets the useable window to zero. The results of SWS are so devastating that no sender should be without this simple piece of insurance.

5. Improved Acknowledgement Algorithms

In the beginning of this paper, an overly simplistic implementation of TCP was described, which led to SWS. One of the characteristics of this implementation was that the recipient of data sent a separate acknowledgement for every segment that it received. This compulsive acknowledgement was one of the causes of SWS, because each acknowledgement provided some new useable window, but even if one of the algorithms described above is used to eliminate SWS, overly frequent acknowledgement still has a substantial problem, which is that it greatly increases the processing time at the sender's end. Measurement of TCP implementations, especially on large operating systems, indicate that most of the overhead of dealing with a segment is not in the processing at the TCP or IP level, but simply in the scheduling of the handler which is required to deal with the segment. A steady dribble of acknowledgements causes a high overhead in scheduling, with very little to show for it. This waste is to be avoided if possible.

There are two reasons for prompt acknowledgement. One is to prevent retransmission. We will discuss later how to determine whether unnecessary retransmission is occurring. The other reason one acknowledges promptly is to permit further data to be sent. However, the previous section makes quite clear that it is not always desirable to send a little bit of data, even though the receiver may have room for

it. Therefore, one can state a general rule that under normal operation, the receiver of data need not, and for efficiency reasons should not, acknowledge the data unless either the acknowledgement is intended to produce an increased useable window, is necessary in order to prevent retransmission or is being sent as part of a reverse direction segment being sent for some other reason. We will consider an algorithm to achieve these goals.

Only the recipient of the data can control the generation of acknowledgements. Once an acknowledgement has been sent from the receiver back to the sender, the sender must process it. Although the extra overhead is incurred at the sender's end, it is entirely under the receiver's control. Therefore, we must now describe an algorithm which occurs at the receiver's end. Obviously, the algorithm must have the following general form; sometimes the receiver of data, upon processing a segment, decides not to send an acknowledgement now, but to postpone the acknowledgement until some time in the future, perhaps by setting a timer. The peril of this approach is that on many large operating systems it is extremely costly to respond to a timer event, almost as costly as to respond to an incoming segment. Clearly, if the receiver of the data, in order to avoid extra overhead at the sender end, spends a great deal of time responding to timer interrupts, no overall benefit has been achieved, for efficiency at the sender end is achieved by great thrashing at the receiver end. We must find an algorithm that avoids both of these perils.

The following scheme seems a good compromise. The receiver of data

will refrain from sending an acknowledgement under certain circumstances, in which case it must set a timer which will cause the acknowledgement to be sent later. However, the receiver should do this only where it is a reasonable guess that some other event will intervene and prevent the necessity of the timer interrupt. The most obvious event on which to depend is the arrival of another segment. So, if a segment arrives, postpone sending an acknowledgement if both of the following conditions hold. First, the push bit is not set in the segment, since it is a reasonable assumption that there is more data coming in a subsequent segment. Second, there is no revised window information to be sent back.

This algorithm will insure that the timer, although set, is seldom used. The interval of the timer is related to the expected inter-segment delay, which is in turn a function of the particular network through which the data is flowing. For the Arpanet, a reasonable interval seems to be 200 to 300 milliseconds. Appendix A describes an adaptive algorithm for measuring this delay.

The section on improved window algorithms described both a receiver algorithm and a sender algorithm, and suggested that both should be used. The reason for this is now clear. While the sender algorithm is extremely simple, and useful as insurance, the receiver algorithm is required in order that this improved acknowledgement strategy work. If the receipt of every segment causes a new window value to be returned, then of necessity an acknowledgement will be sent for every data segment. When, according to the strategy of the previous section, the

receiver determines to artificially reduce the offered window, that is precisely the circumstance under which an acknowledgement need not be sent. When the receiver window algorithm and the receiver acknowledgement algorithm are used together, it will be seen that sending an acknowledgement will be triggered by one of the following events. First, a push bit has been received. Second, a temporary pause in the data stream is detected. Third, the offered window has been artificially reduced to one-half its actual value.

In the beginning of this section, it was pointed out that there are two reasons why one must acknowledge data. Our consideration at this point has been concerned only with the first, that an acknowledgement must be returned as part of triggering the sending of new data. It is also necessary to acknowledge whenever the failure to do so would trigger retransmission by the sender. Since the retransmission interval is selected by the sender, the receiver of the data cannot make a precise determination of when the acknowledgement must be sent. However, there is a rough rule the sender can use to avoid retransmission, provided that the receiver is reasonably well behaved.

We will assume that sender of the data uses the optional algorithm described in the TCP specification, in which the roundtrip delay is measured using an exponential decay smoothing algorithm. Retransmission of a segment occurs if the measured delay for that segment exceeds the smoothed average by some factor. To see how retransmission might be triggered, one must consider the pattern of segment arrivals at the receiver. The goal of our strategy was that the sender should send off

a number of segments in close sequence, and receive one acknowledgement for the whole burst. The acknowledgement will be generated by the receiver at the time that the last segment in the burst arrives at the receiver. (To ensure the prompt return of the acknowledgement, the sender could turn on the "push" bit in the last segment of the burst.) The delay observed at the sender between the initial transmission of a segment and the receipt of the acknowledgement will include both the network transit time, plus the holding time at the receiver. The holding time will be greatest for the first segments in the burst, and smallest for the last segments in the burst. Thus, the smoothing algorithm will measure a delay which is roughly proportional to the average roundtrip delay for all the segments in the burst. Problems will arise if the average delay is substantially smaller than the maximum delay and the smoothing algorithm used has a very small threshold for triggering retransmission. The widest variation between average and maximum delay will occur when network transit time is negligible, and all delay is processing time. In this case, the maximum will be twice the average (by simple algebra) so the threshold that controls retransmission should be somewhat more than a factor of two.

In practice, retransmission of the first segments of a burst has not been a problem because the delay measured consists of the network roundtrip delay, as well as the delay due to withholding the acknowledgement, and the roundtrip tends to dominate except in very low roundtrip time situations (such as when sending to one’s self for test purposes). This low roundtrip situation can be covered very simply by including a minimum value below which the roundtrip estimate is not permitted to drop.

In our experiments with this algorithm, retransmission due to faulty calculation of the roundtrip delay occurred only once, when the parameters of the exponential smoothing algorithm had been misadjusted so that they were only taking into account the last two or three segments sent. Clearly, this will cause trouble since the last two or three segments of any burst are the ones whose holding time at the receiver is minimal, so the resulting total estimate was much lower than appropriate. Once the parameters of the algorithm had been adjusted so that the number of segments taken into account was approximately twice the number of segments in a burst of average size, with a threshold factor of 1.5, no further retransmission has ever been identified due to this problem, including when sending to ourself and when sending over high delay nets.

6. Conservative Vs. Optimistic Windows

According to the TCP specification, the offered window is presumed to have some relationship to the amount of data which the receiver is actually prepared to receive. However, it is not necessarily an exact correspondence. We will use the term "conservative window" to describe the case where the offered window is precisely no larger than the actual buffering available. The drawback to conservative window algorithms is that they can produce very low throughput in long delay situations. It is easy to see that the maximum input of a conservative window algorithm is one bufferfull every roundtrip delay in the net, since the next bufferfull cannot be launched until the updated window/acknowledgement information from the previous transmission has made the roundtrip.

In certain cases, it may be possible to increase the overall throughput of the transmission by increasing the offered window over the actual buffer available at the receiver. Such a strategy we will call an "optimistic window" strategy. The optimistic strategy works if the network delivers the data to the recipient sufficiently slowly that it can process the data fast enough to keep the buffer from overflowing. If the receiver is faster than the sender, one could, with luck, permit an infinitely optimistic window, in which the sender is simply permitted to send full-speed. If the sender is faster than the receiver, however, and the window is too optimistic, then some segments will cause a buffer overflow, and will be discarded. Therefore, the correct strategy to implement an optimistic window is to increase the window size until segments start to be lost. This only works if it is possible to detect that the segment has been lost. In some cases, it is easy to do, because the segment is partially processed inside the receiving host before it is thrown away. In other cases, overflows may actually cause the network interface to be clogged, which will cause the segments to be lost elsewhere in the net. It is inadvisable to attempt an optimistic window strategy unless one is certain that the algorithm can detect the resulting lost segments. However, the increase in throughput which is possible from optimistic windows is quite substantial. Any systems with small buffer space should seriously consider the merit of optimistic windows.

The selection of an appropriate window algorithm is actually more complicated than even the above discussion suggests. The following considerations are not presented with the intention that they be

incorporated in current implementations of TCP, but as background for the sophisticated designer who is attempting to understand how his TCP will respond to a variety of networks, with different speed and delay characteristics. The particular pattern of windows and acknowledgements sent from receiver to sender influences two characteristics of the data being sent. First, they control the average data rate. Clearly, the average rate of the sender cannot exceed the average rate of the receiver, or long-term buffer overflow will occur. Second, they influence the burstiness of the data coming from the sender. Burstiness has both advantages and disadvantages. The advantage of burstiness is that it reduces the CPU processing necessary to send the data. This follows from the observed fact, especially on large machines, that most of the cost of sending a segment is not the TCP or IP processing, but the scheduling overhead of getting started.

On the other hand, the disadvantage of burstiness is that it may cause buffers to overflow, either in the eventual recipient, which was discussed above, or in an intermediate gateway, a problem ignored in this paper. The algorithms described above attempts to strike a balance between excessive burstiness, which in the extreme cases can cause delays because a burst is not requested soon enough, and excessive fragmentation of the data stream into small segments, which we identified as Silly Window Syndrome.

Under conditions of extreme delay in the network, none of the algorithms described above will achieve adequate throughput. Conservative window algorithms have a predictable throughput limit,

which is one windowfull per roundtrip delay. Attempts to solve this by optimistic window strategies may cause buffer overflows due to the bursty nature of the arriving data. A very sophisticated way to solve this is for the receiver, having measured by some means the roundtrip delay and intersegment arrival rate of the actual connection, to open his window, not in one optimistic increment of gigantic proportion, but in a number of smaller optimistic increments, which have been carefully spaced using a timer so that the resulting smaller bursts which arrive are each sufficiently small to fit into the existing buffers. One could visualize this as a number of requests flowing backwards through the net which trigger in return a number of bursts which flow back spaced evenly from the sender to the receiver. The overall result is that the receiver uses the window mechanism to control the burstiness of the arrivals, and the average rate.

To my knowledge, no such strategy has been implemented in any TCP. First, we do not normally have delays high enough to require this kind of treatment. Second, the strategy described above is probably not stable unless it is very carefully balanced. Just as buses on a single bus route tend to bunch up, bursts which start out equally spaced could well end up piling into each other, and forming the single large burst which the receiver was hoping to avoid. It is important to understand this extreme case, however, in order to understand the limits beyond which TCP, as normally implemented, with either conservative or simple optimistic windows can be expected to deliver throughput which is a reasonable percentage of the actual network capacity.

7. Conclusions

This paper describes three simple algorithms for performance enhancement in TCP, one at the sender end and two at the receiver. The sender algorithm is to refrain from sending if the useable window is smaller than 25 percent of the offered window. The receiver algorithms are first, to artificially reduce the offered window when processing new data if the resulting reduction does not represent more than some fraction, say 50 percent, of the actual space available, and second, to refrain from sending an acknowledgment at all if two simple conditions hold.

Either of these algorithms will prevent the worst aspects of Silly Window Syndrome, and when these algorithms are used together, they will produce substantial improvement in CPU utilization, by eliminating the process of excess acknowledgements.

Preliminary experiments with these algorithms suggest that they work, and work very well. Both the sender and receiver algorithms have been shown to eliminate SWS, even when talking to fairly silly algorithms at the other end. The Multics mailer, in particular, had suffered substantial attacks of SWS while sending large mail to a number of hosts. We believe that implementation of the sender side algorithm has eliminated every known case of SWS detected in our mailer. Implementation of the receiver side algorithm produced substantial improvements of CPU time when Multics was the sending system. Multics is a typical large operating system, with scheduling costs which are large compared to the actual processing time for protocol handlers.

Tests were done sending from Multics to a host which implemented the SWS suppression algorithm, and which could either refrain or not from sending acknowledgements on each segment. As predicted, suppressing the return acknowledgements did not influence the throughput for large data transfer at all, since the throttling effect was elsewhere. However, the CPU time required to process the data at the Multics end was cut by a factor of four (In this experiment, the bursts of data which were being sent were approximately eight segments. Thus, the number of acknowledgements in the two experiments differed by a factor of eight.)

An important consideration in evaluating these algorithms is that they must not cause the protocol implementations to deadlock. All of the recommendations in this document have the characteristic that they suggest one refrain from doing something even though the protocol specification permits one to do it. The possibility exists that if one refrains from doing something now one may never get to do it later, and both ends will halt, even though it would appear superficially that the transaction can continue.

Formally, the idea that things continue to work is referred to as "liveness". One of the defects of ad hoc solutions to performance problems is the possibility that two different approaches will interact to prevent liveness. It is believed that the algorithms described in this paper are always live, and that is one of the reasons why there is a strong advantage in uniform use of this particular proposal, except in cases where it is explicitly demonstrated not to work.

The argument for liveness in these solutions proceeds as follows.

First, the sender algorithm can only be stopped by one thing, a refusal of the receiver to acknowledge sent data. As long as the receiver continues to acknowledge data, the ratio of useable window to offered window will approach one, and eventually the sender must continue to send. However, notice that the receiver algorithm we have advocated involves refraining from acknowledging. Therefore, we certainly do have a situation where improper operation of this algorithm can prevent liveness.

What we must show is that the receiver of the data, if it chooses to refrain from acknowledging, will do so only for a short time, and not forever. The design of the algorithm described above was intended to achieve precisely this goal: whenever the receiver of data refrained from sending an acknowledgement it was required to set a timer. The only event that was permitted to clear that timer was the receipt of another segment, which essentially reset the timer, and started it going again. Thus, an acknowledgement will be sent as soon as no data has been received. This has precisely the effect desired: if the data flow appears to be disrupted for any reason, the receiver responds by sending an up-to-date acknowledgement. In fact, the receiver algorithm is designed to be more robust than this, for transmission of an acknowledgment is triggered by two events, either a cessation of data or a reduction in the amount of offered window to 50 percent of the actual value. This is the condition which will normally trigger the transmission of this acknowledgement.

APPENDIX A

Dynamic Calculation of Acknowledgement Delay

The text suggested that when setting a timer to postpone the sending of an acknowledgement, a fixed interval of 200 to 300 milliseconds would work properly in practice. This has not been verified over a wide variety of network delays, and clearly if there is a very slow net which stretches out the intersegment arrival time, a fixed interval will fail. In a sophisticated TCP, which is expected to adjust dynamically (rather than manually) to changing network conditions, it would be appropriate to measure this interval and respond dynamically. The following algorithm, which has been relegated to an Appendix, because it has not been tested, seems sensible. Whenever a segment arrives which does not have the push bit on in it, start a timer, which runs until the next segment arrives. Average these interarrival intervals, using an exponential decay smoothing function tuned to take into account perhaps the last ten or twenty segments that have come in. Occasionally, there will be a long interarrival period, even for a segment which is does not terminate a piece of data being pushed, perhaps because a window has gone to zero or some glitch in the sender or the network has held up the data. Therefore, examine each interarrival interval, and discard it from the smoothing algorithm if it exceeds the current estimate by some amount, perhaps a ratio of two or four times. By rejecting the larger intersegment arrival intervals, one should obtain a smoothed estimate of the interarrival of segments inside

a burst. The number need not be exact, since the timer which triggers acknowledgement can add a fairly generous fudge factor to this without causing trouble with the sender’s estimate of the retransmission interval, so long as the fudge factor is constant.