INTERNET-DRAFT Amy S. Hughes, Joe Touch, John Heidemann
draft-hughes-restart-00.txt ISI
Dec. 1, 2001
Expires: Jun. 1, 2002

# Issues in TCP Slow-Start Restart After Idle

## Status of this Memo



## Abstract

This draft discusses variations in the TCP 'slow-start restart' (SSR) algorithm, and the unintended failure of some variations to properly restart in some environments. SSR is intended to avoid line-rate bursts after idle periods, where TCP accumulates permission to send in the form of ACKs, but does not consume that permission immediately. SSR's original "restart after send is idle" is commonly implemented as "restart after receive is idle". The latter unintentionally fails to restart for bidirectional connections where the sender's burst is triggered by a reverse-path data packet, such as in persistent HTTP. Both the former and latter are shown to permit bursts in other circumstances. Several solutions are discussed, and their implementations evaluated.

This document updates draft-ietf-tcpimpl-restart-01.txt. It is a product of the LSAM, X-Bone, and DynaBone projects at ISI. Comments are solicited and should be addressed to the authors.

## Introduction

Slow-Start Restart (SSR) describes one TCP behavior to respond to long sending pauses in an open connection. When a sender becomes idle, the normal ACK-clocking mechanism which regulates traffic is no longer present and the sender may introduce a burst of packets into the network as large as the current congestion window (CWND). Such a burst may be too large for the intermediate routers to handle and may be too large for the receiver to handle at one time as well.

A send timer was first proposed [JK92] to detect idle sending periods; the recommended response is to close the congestion window and perform a new slow-start. However, a footnote to this first proposed solution noted that send/receive symmetry on the channel meant that a receive timer could be used instead to achieve the same results. As this second solution takes advantage of a timer that is already required (to detect packet loss) it was implemented by Jacobson and Karels. This solution has been repeated in implementations which derive from their work.

Bursty connections, such as the persistent connections required in HTTP/1.1 [FGMFB97] have been found to interact in meaningful ways with SSR [Hei97]. In fact, it was discovered that SSR never occurs with HTTP/1.1 [Poo97]. This is because a new request will reset the receive timer (as suggested in the footnote in [JK92]) and the sending pause will not be detected [Tou97].

Further, both timer solutions depend on the retransmit timeout (RTO) and cannot detect send pauses that are shorter than this duration. In such cases, the sender may transmit a burst as large as the full congestion window.

## Burst Detection.

There are several ways of determining whether a connection is at risk of sending a burst of packets into the channel. We will discuss each method below, from the least radical to the most radical.

### Receive Timer:

The use of a receive timer is the most common burst detection method. It is attractive because it is simple and makes use of an existing timer. However, a receive timer does not properly detect bursts in

HTTP/1.1 because the timer is cancelled when the request packet is received. Further, when the connection is idle for less than a full RTO, a burst cannot be detected. Such a burst can happen when the connection is "nearly idle" or when ACKs are lost or reordered.





Send Timer:

A send timer is the reciprocal solution to using a receive timer. While it requires a new timestamp field to be maintained, it clearly detects send pauses and corrects the problem presented by HTTP/1.1. However, as with the receive timer, it cannot detect bursts that could happen before a full RTO.

Packet Counting:

An alternative method examines the unused portion of the congestion window to determine if the capacity to burst exists. This method is simple, it uses existing information to make its decision, and it solves both the HTTP/1.1. problem as well as the RTO problem. In addition, it addresses the problem that needs to be solved (bursts) instead of a specific circumstance where the problem could happen (send pauses). However, where timer detection avoids defining a burst (it defines idle periods instead), here a burst must be defined before it can be detected. One possible definition is the situation where the available portion of the sending window is some proportion of the entire congestion window, say 50%. Another definition places a numerical limit on the available portion of the congestion window, say 4 or CWND-1 packets.

Burst Response

Once a burst is detected, there are several different ways to take action. The different possibilities are listed below, again from least to most radical.

Full Restart:

Reducing the congestion window to one packet and re-entering slow-start, the original slow-start restart, is one response. This was the solution proposed by Jacobson and Karels. This is a very conservative response and it defeats most of the speedup that

HTTP/1.1 provides [HOT97]. Current proposals [FAP97] have suggested increasing the initial window from 1 packet to 4 packets. Further, depending on the method of burst detection, Full Restart can be far more punitive than it should be. Coupled with a timer, full restart is most likely to respond to a completely empty congestion window. Coupled with Packet Counting, the response could close the window too far, even smaller than the amount of outstanding data.

Window Limiting:

This is a modified version of Full Restart which solves the problem created by using Packet Counting to detect bursts. With this type of





response, the congestion window is reduced to the amount of outstanding data plus the slow-start initial window (1, 2, or 4). It works exactly like Full Restart in the idle case, but is successful at controlling bursts in an active connection. Further, in an active connection, it effectively implements a leaky bucket of the initial window size for the accumulation of send opportunity based on the receipt of ACKs. This solution is fairly conservative, especially as it defaults to Full Restart, but more importantly, sending opportunity is simply lost if not used, and is not available for paced output. Also, it forces negative congestion feedback on the congestion window.

Burst Size Limitation:

When a burst is detected, its effects are limited, the sender may not send any more than a preset number of packets into the network. This is less conservative than the first two responses in that it does not affect the size of the congestion window, and it is simple to implement, simply count up the number of packets you can send and stop when you reach the limit. The burst count is reset according to some policy, such as when ACKs are received, each time send is called, or when some timer triggers. The behavior is determined by how these parameters are combined to reset the count.

Pacing:

When a burst is detected, packets are periodically sent into the network until the sender starts receiving acks and normal maintenance can be resumed [VH97,PK98]. This solution is very easy on the

network and scales well in cases of high bw/delay. However, it requires a new timer and additional research for parameter tuning.

## Implemented Solutions

Now we will examine combinations of the different detection and response methods presented above. Each of the solutions below have been implemented in some form.

### BSD Implementation (Jacobson and Karels)

The most common implementation uses a receive timer coupled with Full Restart. This is the implementation that causes the interaction problems with HTTP/1.1. The obvious alternative is to implement a send timer as originally intended and use Full Restart. There are several drawbacks to this solution. First, a send timer adds additional state and serves no purpose other than to correct the bursting behavior after send pauses. Second, forcing a slow-start in this situation is problematic for HTTP/1.1. A slow-start for each





new user request adds a delay burden to characteristically small HTTP responses. Further, the HTTP user request pattern is unpredictable. It is possible for the user to make a new request before the send timer expires, triggering a burst that would defeat such a timer.

### Maximum Burst Limitation (Floyd)

Floyd has proposed a coupling of Packet Counting with Burst Size Limitation. It was developed to avoid bursts caused by gaps in the ACK stream. Such gaps cause the send window to slide forward in jumps, rather than smoothly by 1-2 MTUs; subsequent send calls can thus generate bursts. This solution has been implemented in "ns" and it prevents the sender from transmitting a series of back-to-back packets larger than the user configured burst limit (suggested to be 5 packets) [NS97].

Maxburst defines a new function, send_much, with a parameter, maxburst, which is called every time an external event occurs, such as ACK is received or a timeout occurs. Each time send_much is called, at most maxburst packets are emitted. This forces interleaved processing of ACKs and sending of data. Sends initiated by the arrival of data in the buffer are not affected; as a result, if there

is less than RTO of delay, and the send buffer is filled by the application, Maxburst can still send a burst the size of an entire window.

Maxburst does not explicitly address the sending of bursts after an idle period; it relies on the existing mechanism to collapse CWND after an RTO, because send_much does not replace send as the application interface to transmitting data.

The limit of 5 packets is designed to allow non-sequental ACK losses, and allow the CWND to grow normally under slow-start. Here is a list of the possible burst limits, and the need for each:

1 packet per ACK or nothing gets done

2 packets per ACK if ACKs are compressed

3 packets during window growth; 2 for the compressed ACK, 1 for the growth

4 packets per ACK received if ACKs are lost (not back-to-back), and ACKs are compressed

5 packets per ACK received if ACKs are lost (not back-to-back), ACKs are compressed, and the window is growing



---



Use it or lose it (UI/LI) (Hughes, Touch, and Heidemann)

One proposed solution combines Packet Counting with Window Limiting, and the IETF community refers to it as "Use-It-or-Lose-It" (it was originally called Congestion Window Monitoring, CWM, but we will use the acronym UI/LI, pronounced 'wee-lee', here). Whenever (CWND - outstanding data > 4), we reduce CWND to (outstanding data + 4). The choice of 4 packets is discussed in with the implementation details below. UI/LI allows the congestion window to grow normally but shrinks the congestion window as the sender becomes idle. It also prevents the sender from transmitting any bursts larger than 4 packets in response to a new request. Because UI/LI is not dependent on any timers, the loss of an ACK or a nearly idle connection cannot cause any bursts. UI/LI is similar to Maxburst, but avoids the burst by reducing CWND, rather than by inhibiting the sends directly. As a

result, we avoid the potential problem of sequential calls to "tcp_output", which would cause bursts in Maxburst. UI/LI also causes TCP to use the feedback of 'not using the CWND fast enough', which results in a decrease in the CWND.

UI/LI effectively imposes a leaky bucket type limitation on the congestion window. The window is allowed to grow and be managed normally but the sender is not allowed to save up any sending opportunities. Any opportunity that is not used is lost. This property of UI/LI forces interleaved reception of ACKs and processing of sends.

Burst-or-lose (Touch, Floyd)

We have considered a modified version of Maxburst, in which all sends, including those initiated by data arrival from the application, limit their burst size each time they are called, and reset their flag at that time. We call this burst-or-lose (BOL). The result is very similar to UI/LI, but the bucket reflects only the permission to send created as the result of an ACK receipt, rather than the overall limit of the congestion window.

In this case, it is unncessary to collapse CWND after an RTO. The primary reason for the collapse is to avoid bursts; BOL avoids bursts completely, but recovers as quickly as the ACK clocking can be restored.

In the case when there is no data to send, multiple ACK arrivals do not increase the bucket available for sending. This avoids the bursts after idle periods. In the case when ACKs are lost, the remaining ACK generates permission to send only a fixed amount of data. The application cannot defeat this bucket by calling send multiple times.





Rate Based Pacing (Visweswaraiah and Heidemann)

Rate Based Pacing (RBP) combines the Pacing response with either a Send Timer or Packet Counting. It avoids slow-start when resuming after sending pauses and allows the normal clocking of packets to be gracefully restarted. When a burst potential is detected, the algorithm meters a small burst of packets into the channel [VH97]. RBP is the least conservative solution to the bursting problem

because it continues to make use of the pre-pause congestion window. If network conditions have changed significantly, maintaining the previous window could cause the paced connection to be overly aggressive as compared to other connections. (Although some work suggests congestion windows are stable over multi-minute timeframes [BSSK97].) More recently pacing has been suggested for use in wireless networking scenarios [BPK97], and for satellite connections.

## Solution Comparison

Below we present a comparison of the solutions presented above, plus the original Send Timer solution. The Send Timer solution is not in use, but we implemented it in FreeBSD for comparison purposes.

We contrast each solution on four criteria. A "yes" in the "HTTP Burst" column indicates that this solution will solve the bursting problem created by HTTP/1.1. A "yes" in the "Other Burst" column indicates that the solution will prevent bursts in other situations as well. A "yes" in the "Adj CWND" column indicates that the solution involves adjusting the value of CWND. The final column, "Code Size", gives the number of lines of code needed to correctly implement the solution.

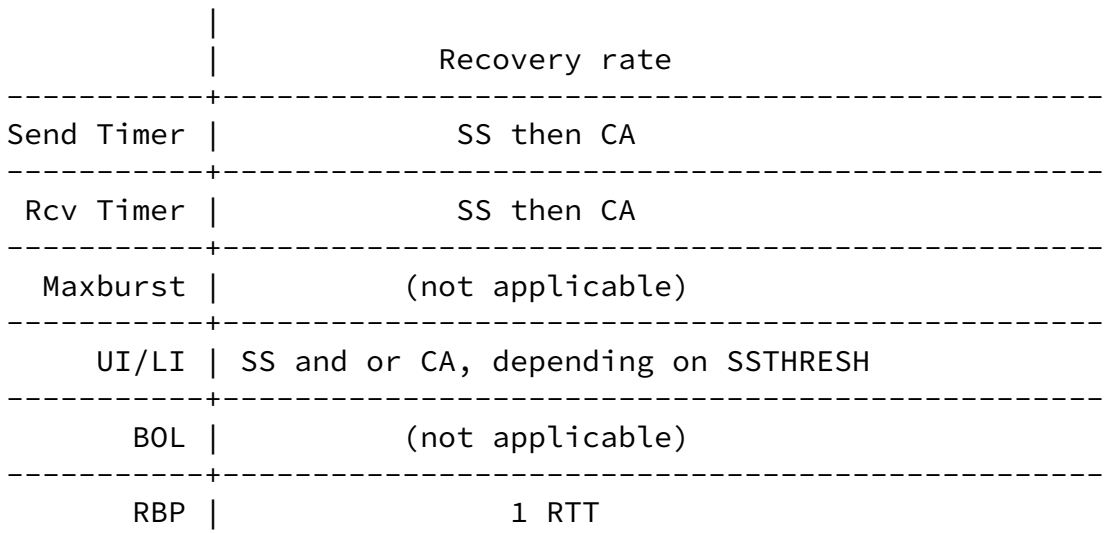
| | HTTP Burst | Other Burst | Adj CWND | Code Size |
|---|---|---|---|---|
| Send Timer | yes | no | yes | ~5 lines |
| Rcv Timer | no | no | yes | 3 lines |
| Maxburst | yes | yes | no | ~5 lines |
| UI/LI | yes | yes | yes | 3 lines |
| BOL | yes | yes | no | ~5 lines |
| RBP | yes | yes | no | ~30 lines |





There are other conditions under which to compare these algorithms,

including behavior after a single ACK loss, behavior after multiple ACK losses (sequential and non-sequential), and the window regrowth properties. These are outside the scope of this document.

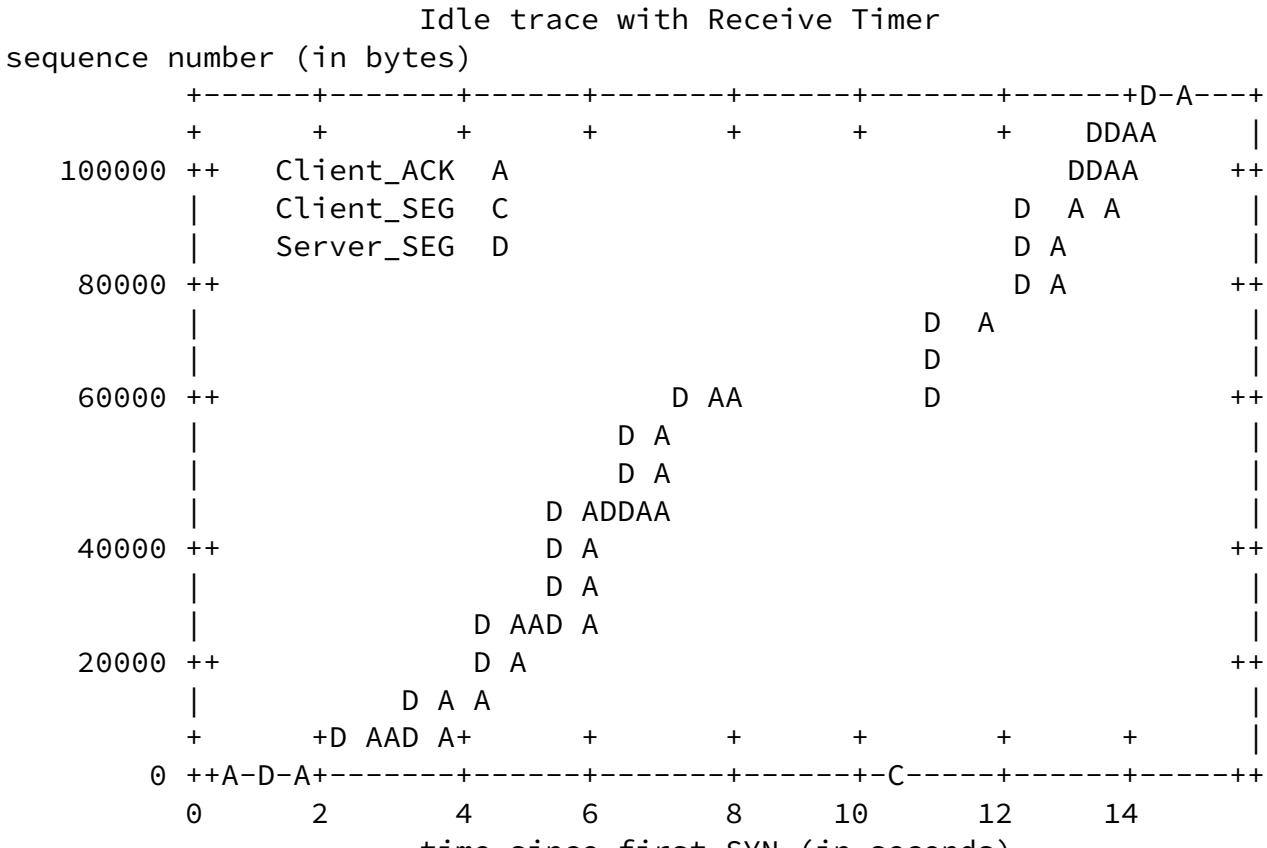



The only significant difference between UI/LI and BOL is whether CWND is affected, and thus the recovery period after the loss of an ACK. UI/LI interprets ACK loss as an event warranting traditional TCP window recovery, whereas BOL and Maxburst do not. Both UI/LI and BOL avoid the need for idle detection for the purpose of CWND adjustment, and implement a type of leaky bucket. Although UI/LI was intended as a type of leaky-bucket system, BOL is much more directly analogous.

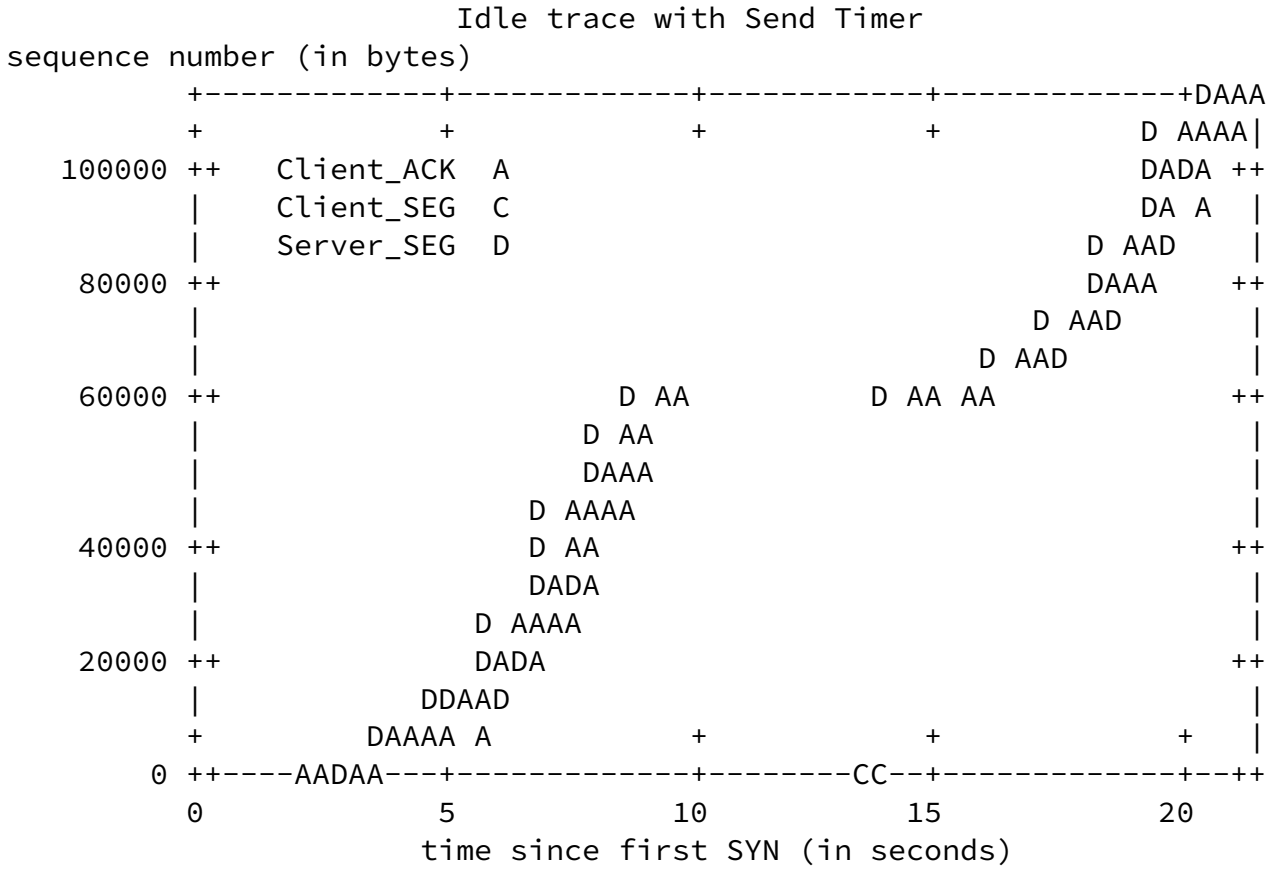
Experimental Comparisons

Packet traces of the current FreeBSD implementation of SSR (using the receive timer), of a modified version of FreeBSD using a send timer, and of UI/LI with HTTP/1.1 support the above observations. The graphs below show traces of two HTTP/1.1 requests. The first request opens the CWND. How the server responds to the packet containing the second request reveals the differences between implementations.

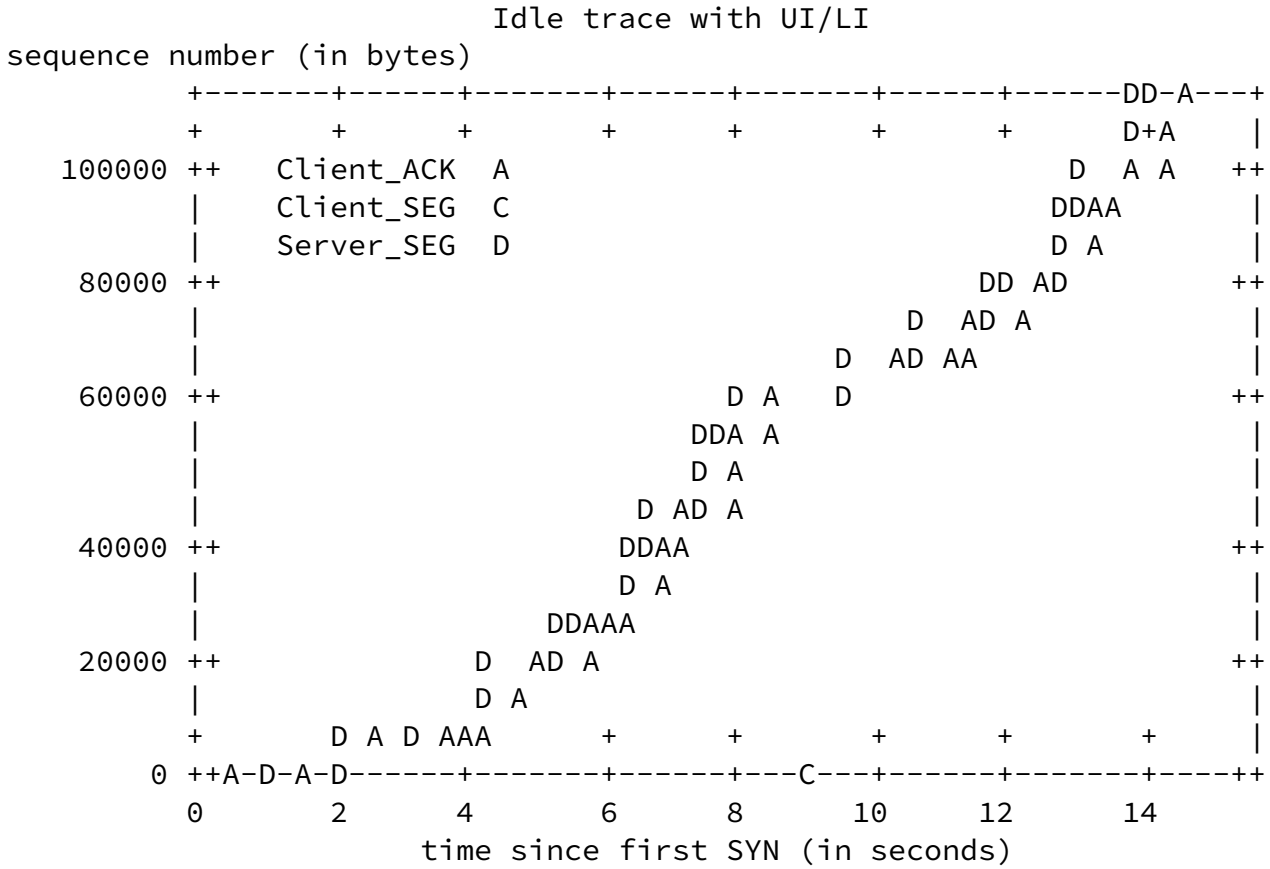
The first graph shows the trace with FreeBSD v2.2.6, but the relevant portions of the code have not been altered in more current releases. In response to the first request, the server performs slow-start normally. After 10 seconds, and after the retransmission timeout (RTO), the second request arrives. At this point, receipt of the second request resets the receive timer. The server, under the belief that communication has not gone idle, sends a CWND-sized burst of packets into the network.




The second graph shows the trace with FreeBSD v2.2.6 altered to use a send timer to detect idle communication. As expected, the server responds to the first request with normal slow-start. Now, the receipt of the second request comes in after the RTO and the idle period is correctly detected. In response, the server re-enters slow-start for the second response.




The third graph shows the trace with FreeBSD altered to use UI/LI to limit bursts. Again, the first request and response are normal. This shows that UI/LI allows the congestion window to grow normally. With the second request we see the main difference of UI/LI. Again, the second request comes after the RTO, and here we see the main difference with UI/LI. In response to the second request, the server does not send a burst the size of CWND or enter slow-start. Instead, it sends several small 4-packet bursts, with the CWND growing normally from this point.




Note, RTO is the usual timer limit, but any value would have the same results, depending on when the second request was presented in relation to the timer. If the second request arrives after the timer expires, the reponse will behave as presented in these graphs. If the second request arrives before the timer expires and after all outstanding packets have been acknowledged, systems with a send or receive timer will

send a CWND-sized burst. If the second request arrives before the last ACK but after the last packet sent, the server has the potential to send a burst no larger than CWND. UI/LI will limit any burst to 4 packets regardless of the timing of the second request.

Implementation of UI/LI

UI/LI requires a simple modification to existing TCP output routines. The changes required replace the current idle detection code. Below is the suggested pseudocode from Appendix C of [JK92]:

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

UI/LI would replace that code as below:





```
          int idle = (snd_max == snd_una);
          int maxwin = 4 + snd_nxt - snd_una;
          if (cwnd > maxwin)
               cwnd = maxwin;
```

Packet counting is implemented by line 2. Lines 3 and 4 implement Window Limitation. Note that the comparison in line 1 (defining "idle" [JK92]) is required for later Silly Window Syndrome avoidance and could now be moved there.

We have implemented UI/LI in FreeBSD. The affected code is in the file /sys/netinet/tcp_output.c. The lines:

```
          idle = (tp->snd_max == tp->snd_una);
          if (idle && tp->t_idle >= tp->t_rxtcur)
               tp->snd_cwnd = tp->t_maxseg;
```

Are replaced with:

```
          idle = (tp->snd_max == tp->snd_una);
          maxwin = (4 * tp->t_maxseg) + tp->snd_nxt - tp->snd_una;
          if (tp->snd_cwnd > maxwin)
               tp->snd_cwnd = maxwin;
```

The choice of limiting the available congestion window to 4 packets

is based on the normal operation of TCP. An ACK received by the sender may be in response to the receipt of 2 packets, allowing another 2 to be sent. Further, normal window growth may require the sending of a third packet. Lastly, in slow-start with delayed ACKs, the receipt of an ACK can trigger the sending of 4 packets. Thus, 4 packets is a reasonable burst to send into the network.

Increasing the initial window in slow-start to 4 packets has already been proposed [FAP97]. The effects of this change have been explored in simulation in [PN98] and in practice in [AHO98]. Such a modification to TCP would cause the same behavior as our solution in the cases where the pause timer has expired. It does not address the pre-timeout bursting situation we are concerned with.

Implementation of BOL

BOL can be implemented with a small amount of additional state, and a small number of lines in TCP's input, output, and timer routines. The state indicates the number of packets that can be emitted before the next timer or ACK receipt. The other lines increment the state when a notable event occurs, and decrement it when packets are sent.

Our implementation uses two state variables: bucket - the number of





packets that can be immediately sent, and ack_ratio - the number of packets per ACK (upper bound). ack_ratio is implied by RFC-2581, as at least one ACK "SHOULD" be sent for every 2 MSSs received, i.e., ACK compression [APS99]. This is our default; we parameterize it so that when others modify TCP to use less frequent ACKs, our portion of the code will operate correctly. It should be noted that less frequent ACKs will, by definition, increase the burstiness of the source as a result. We also assume 'initial_win' is defined, currently 2 MSS, but may be configurable as per Experimental RFC-2414 [FAP97].

There are two events that set the value of bucket. When an ACK is received, we set bucket to ack_ratio * 2 + 1. This is for the same reasons as Maxburst uses '5' - to allow for TCP to run at full rate in the presence of isolated ACK losses. On timeout events, the bucket is set to init_win.

When packets are emitted in the existing tcp_output routine, the bucket is decremented. The bucket does not become negative; if it

would, that data is not sent, and the routine returns. This 'refusal to send' behavior is identical to not having sufficient send window; the application will retry, typically due to some timer event.

The code is as follows:

```
/* define the state for BOL */
     int  ack_ratio,     /* max number of packets per ack */
     int  bucket,        /* max burst size */

#DEFINE init_win 2

/* set the bucket on ACK receipt */
     bucket = ack_ratio * 2 + 1;

/* set bucket on timeout */
     bucket = init_win;

/* decrement the bucket accordingly */
     can_send = min(bucket - want_send, 0);
     bucket -= can_send;
     do_send(can_send);
     return can_send;
```

Local connections

We recently discovered cases where SSR adversely affected local connections, i.e., connections within the same subnet. There is already code in the current BSD-derived TCP implementation that disables initializing the congestion window (CWND) to a small value, when both





ends of a connection share a subnet. This code, from FreeBSD tcp_input.c, is outlined below, where inp = tp->t_inpcb.

```
/*
 * Don't force slow-start on local network.
 */
if (!in_localaddr(inp->inp_faddr))
     tp->snd_cwnd = mss;
```

RFC-2414 observes that there are 3 types of window sizes used in BSD TCP:

1. intital window
2. restart after timeout
3. restart after loss

Recent work indicates that these windows should be treated differently; e.g., the restart-after-loss window should always be set to 1, and the initial window can be set to 4 or not used for local connections [FAP97]. We proposed that it is consistent to extend this local exception to the restart-after-timeout window, as shown below as a FreeBSD code segment:

```
idle = (tp->snd_max == tp->snd_una);
maxwin = (4 * tp->t_maxseg) + tp->snd_nxt - tp->snd_una;
if ((tp->snd_cwnd > maxwin) &&
   !in_localaddr(tcp->t_inpcb->inp_faddr))
     tp->snd_cwnd = maxwin;
```

This modification is considered reasonable because it corresponds to the initial window determination. There are other cases where TCP uses "local" rules, such as MSS determination, and piggybacking.

## Performance Issues

The purpose of these modifications is to avoid line-rate bursts, primarily as the result of the source being idle, but also due to ACK losses. A result of many of the proposed solutions is to limit the source, either by reducing the congestion window, or limiting the transmission of bursts. The performance effects of congestion window reduction are well-studied, and are not addressed here. BOL can limit the source without modifying the congestion window, and there are concerns that this may affect the performance of TCP.

In particular, some implementations of TCP send ACKs in a bursty fashion, either by reducing the send/receive context switching, or by reducing the overall number of ACKs. These modifications are used to increase the performance of TCP, but can also increase the burstiness





of the source. BOL, by reducing the burstiness of the source, is considered to potentially limit the effectiveness of these enhancements.

TCP currently recommends that an ACK "SHOULD" be sent for at least

every two MSS's received. We assume that this recommendation implies an interleaving factor for the source as well. If this factor is encoded in a TCB variable (ack_ratio), then BOL can allow for corresponding bursts consistent with reasonable (non-adjacent) ACK losses.

We assume that TCPs should not delay ACK processing to aggregate the receive-side timeslices, in order to reduce send/receive interleaving, unless this is also encoded in their ack_ratio variable. Our initial measurements indicate that even the recommended interleaving does not substantially affect the performance of TCP.

In addition, we recommend that any burst-avoidance modification be disabled for direct LAN connections. For such connections, the congestion window mechanism is already disabled in several TCP implementations. The resulting burstiness is handled by the link layer and receiver buffers.

## Conclusions

At this time, we propose BOL as a simple, minimal and effective fix to the 'bug' in current TCP implementations that is exploited by HTTP/1.1. Modifications can be made to TCP to solve the slow-start restart problem that are consistent with the original congestion avoidance specifications (i.e. a send timer). However, we feel that the original intended behavior is not appropriate to some current applications, specifically HTTP. Thus, we recommend BOL to prevent bursts into the network. Not only does this solution solve the current problem in a simple way, it will prevent bursting in any other situation that might arise. The packet bursts which we allow are consistent with congestion window growth algorithms and with Floyd's conclusion about increasing the initial window size.

BOL, as well as the other solutions listed, need to be re-evaluated within emerging TCP implementations, e.g., SACK [JB88]. In general, TCP has no rate pacing and uses congestion control to avoid bursts in current implementations. A more explicit mechanism, such as RBP or similar proposals may be desirable in the future.

## Security implications

There are security risks associated with these algorithms that can result in denial of service attacks. Intermediate routers can delay data or ACKs in ways that can cause the restarting party to restart more conservatively. They also result in less probability of the

restarting party generating a line-rate burst.

In Send-Timer and Maxburst algorithms, as well as in a no-restart system, an attacker (either an intermediate router or data receiver) can clump the delivery of ACKs, which can result in line-rate bursts.

The Receive-Timer, Rate-Based Pacing, UI/LI, and BOL algorithms use data or ACK arrival to moderate outgoing data and avoid bursts. If the attacker attempts to deliver data or ACKs so as to cause a burst, these algorithms successfully avoid creating a burst. In the process, their throughput is reduced.

We believe the latter algorithms should be considered more robust than the former, because slow-start restart focuses on avoiding bursts, rather than maximizing throughput. In the recommended algorithms (BOL now, RBP for the future), the system succesfully avoids generating a burst as the result of an attacker.

## Acknowledgements

We would like to acknowledge Kacheong Poon, Sally Floyd, members of the End-to-End-Interest Mailing List, and members of the IETF TCP Implementation Working Group for their feedback and discussions on this document. We would also like to thank Ruth Aydt, Steve Tuecke, and Carl Kesselman for their observation of the local connection issue.

## References

[AHO98] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP with Larger Initial Windows. ACM Computer Communications Review, 28(3), 41-52 , July 1998.

[APS99] M. Allman, V. Paxson, W. Stevens. TCP Congestion Control, April 1999. RFC-2581.

[BPK97] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz. The Effects of Asymmetry on TCP Performance. In Proceedings of the ACM/IEEE Mobicom, Budapest, Hungary, ACM. September, 1997.

[BSSK97] Hari Balakrishnan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz. Analyzing Stability in Wide-Area Network Performance. In Proceedings of the ACM SIGMETRICS, Seattle WA, USA, ACM. June, 1997.

[FGMFB97] R. Fielding, Jim Gettys, Jeffrey C. Mogul, H. Frystyk, and

Tim Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1,





January 1997. RFC-2068.

[FAP97] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC-2414.

[Hei97] John Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. ACM Computer Communications Review, 27(2), 65-73, April 1997.

[HOT97] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the Performance of HTTP Over Several Transport Protocols. ACM/IEEE Transactions on Networking 5(5), 616-630, October, 1997.

[JB88] Van Jacobson and R.T. Braden. TCP extensions for long-delay paths, October 1988. RFC 1072.

[Jac88] Van Jacobson. Congestion Avoidance and Control. Proc. SIGCOMM '88, in ACM Computer Communication Review, 18(4):314-329, August 1988. NOTE: This paper lacks the appendix that describes slow-start restart, which appears only in the unpublished [JK92].

[JK92] Van Jacobson and Mike Karels. Congestion Avoidance and Control. This is a revised version of [Jac88], which adds Karels as co-author and includes an additional appendix. The revised version has not been published, but is available at ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

[NS97] ns Network Simulator. http://www-mash.cs.berkeley.edu/ns/, 1997.

[PK98] Venkata N. Padmanabhan and Randy H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In Proceedings of the IEEE GLOBECOM Internet Mini-Conference, Sydney, Australia, November, 1998.

[PN98] K. Poduri and K. Nichols. Simulation Studies of Increased Initial TCP Window Size, September 1998. RFC-2415.

[Poo97] Kacheong Poon, Sun Microsystems, tcp-implementors mailing list, August, 1997.

[Tou97] Joe Touch, ISI, tcp-implementors mailing list, August 12, 1997.

[VH97] Vikram Visweswaraiah and John Heidemann. Improving Restart of Idle TCP Connections. Technical Report 97-661, University of Southern California, November 1997.





Authors/ Address

Amy S. Hughes, Joe Touch, John Hiedemann
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA
Phone: +1 310-822-1511
Fax: +1 310-823-6714
URLs: http://www.isi.edu/~ahughes
http://www.isi.edu/touch
http://www.isi.edu/~johnh
Email: ahughes@isi.edu
touch@isi.edu
johnh@isi.edu