Notes on Burst Mitigation for Transport Protocols*

Mark Allman
ICSI / ICIR
mallman@icir.org

Ethan Blanton
Purdue University
eblanton@cs.purdue.edu

ABSTRACT

In this note we explore the various causes of micro-bursting in TCP connections and also the behavior of several mitigations that have been suggested in the literature along with extensions we develop herein. This note methodically sketches the behavior of the mitigations and presents the tradeoffs of various schemes as a data point in the ongoing discussion about preventing bursting in transport protocols.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Algorithms, Performance

Keywords

Bursting, TCP

1. INTRODUCTION

In this note we investigate the Transmission Control Protocol's (TCP) [20] bursting behavior.[1] TCP's bursting behavior can be problematic in several ways (as sketched in more detail in the next section). First, bursting can stress queues in the network and lead to packet loss, which can in turn negatively impact both the connection doing the bursting and traffic sharing the stressed router. Second, bursting can cause scaling on short timescales as well as increase queuing delays in routers. Over the years, several researchers have suggested mitigations to TCP's burstiness. We investigate and show the behavior of both these previously proposed mitigations and newly extended techniques.

TCP naturally sends two distinct kinds of segment bursts into the network. First, each TCP acknowledgment (ACK) covering new data that arrives at the TCP sender slides a sending window and liberates a certain number of segments which are transmitted at the line-rate of the connected network. We will call bursts of segments sent in response to a single incoming ACK *micro-bursts*. TCP's congestion control algorithms cause the second kind of bursting behavior. While using the slow start algorithm, a TCP sender increases the amount of data transmitted into the network by a factor of 1.5–2 during each subsequent round-trip time (RTT) (the exact factor depends on whether the receiver generates delayed ACKs [7, 4], whether the sender uses byte counting [2], and the network dynamics of the path the ACKs traverse). An additional cause of macro-bursting is ACK compression [23]. This phenomenon occurs when ACK packets get "bunched up" behind larger data packets in router queues and end up arriving at the end host more rapidly than they were transmitted by the peer. This "bunching up" can cause bursting. We term the bursts caused by slow start and ACK compression *macro-bursts* since they occur over longer time periods than the micro-bursts sketched above.

TCP's macro-burstiness has been the topic of several papers in the literature. [16] analyzes the impact of TCP's macro-burstiness on queueing requirements. [2] proposes an increase in the macro-burstiness of TCP in an attempt to mitigate some of the performance hit caused by delayed ACKs during slow start. [17] proposes using rate-based pacing during slow start to reduce the queueing requirement TCP's macro-bursts place on routers in a network path. Additionally, [1] investigates a general pacing scheme for TCP.

We do not consider macro-bursts further here. Rather, we outline the behavior of schemes to reduce micro-burstiness. TCP's "normal" level of micro-burstiness is to transmit 1–3 segments in response to each ACK (assuming no ACK loss and nothing else anomalous on the connection) [4]. Each ACK that acknowledges new data causes TCP's transmission window to slide. In the normal case, with delayed ACKs, the window slides by 2 segments for each ACK. When TCP is increasing the size of the window, an additional third segment may be sent due to this increase (which happens during slow start and congestion avoidance, at different intervals). As outlined in § 4 bursts of more than 3 segments can happen naturally in TCP connections. We consider bursts of 3 segments or less to be acceptable (per the standards) and bursts of over 3 segments in length to be anomalous[2]. That is not to say that the bursts are caused by problematic TCP implementations[3]. Here we concentrate on bursts caused by an interaction between TCP's congestion control algorithms and specific network dynamics.

*ACM Computer Communication Review, April 2005.

[1]This note is cased in terms of TCP, but should naturally apply to transport protocols that use similar window-based congestion control algorithms (e.g., SCTP [21], DCCP [15] using CCID2 [9]).

[2]TCP specifically allows bursts of 4 segments at the beginning of a connection or after a lengthy idle period [3]. However, we do not consider this one-time allowance to indicate that the TCP congestion control specification tolerates such micro-bursts. Also, [2] allows for micro-bursts up to 4 segments in length as a regular occurrence during TCP slow start. However, [2] is an experimental document and not standard.

[3]Bursts *are* caused by buggy TCP implementations, as well, of course. For instance, stretch ACKs outlined in [18, 19] cause micro-bursts. [6] shows that such stretch ACKs are not uncommon in today's Internet.

The goal of this note is to illustrate the causes of bursting and the behavior of several mitigations that have been proposed, as well as extensions developed within. The purpose is not to make hard conclusions, but rather to offer a data point in the ongoing discussion about micro-bursting within transport protocols[4]. This note is organized as follows. § 2 details related work. § 3 discusses a number of mechanisms that reduce the size of bursts. § 4 presents simulations illustrating TCP's bursting behavior and the behavior of the burst mitigation strategies. We conclude and sketch future work in § 5.

2. RELATED WORK

The literature contains several studies dealing with the impact of micro-bursts and potential methods for mitigation of large burst sizes. [22] considers bursts caused by re-using HTTP connections after an idle period, and shows that rate-based pacing is useful in reducing burstiness and increasing performance. We discuss the topic of using rate-based pacing as a general micro-burst mitigation strategy in § 3 and § 4.

[12] discusses micro-bursts in a more general way, considering techniques for both detecting and reducing micro-bursts. It introduces the Use It or Lose It algorithm discussed in § 3.

[11] discusses the behavior and performance impact of micro-bursts that occur after loss recovery in satellite networks across a range of TCP variants. [8] also illustrates micro-bursts in the context of loss recovery, and introduces the MaxBurst algorithm discussed in § 3.

[14] investigates the causes of bursts (both micro- and macro-bursts) in the network and their impact on aggregate network traffic. It finds that bursts at sources create scaling on shortb timescales and can cause increased queuing delays in intermediate nodes along a network path.

[6] attempts to quantify micro-bursting in the Internet, and correlate micro-bursts with performance impact on individual TCP connections, and suggests that while micro-bursts of moderate size are well-tolerated (in the context of individual TCP connections, in contrast to the findings in [14] about aggregate traffic), larger bursts greatly increase the probability of packet drops.

We incorporate the techniques previously defined in the literature and make several extensions to them in our analysis.

```
def MaxBurst ():
    if ackno > highack:
        count = 0
        while (ownd < cwnd) && \
                  (count < MB_SIZE):
            send_packet ()
            count += 1
            ownd += 1


def AggressiveMaxBurst ():
    count = 0
    while (ownd < cwnd) && \
              (count < MB_SIZE):
        send_packet ()
        count += 1
        ownd += 1


def UseItOrLoseIt ():
    if cwnd > (ownd + MB_SIZE):
        cwnd = ownd + MB_SIZE
    while ownd < cwnd:
        send_packet ()
        ownd += 1


def CongestionWindowLimiting ():
    if cwnd > (ownd + MB_SIZE):
        if ssthresh < cwnd:
            ssthresh = cwnd
        cwnd = ownd + MB_SIZE
    while ownd < cwnd:
        send_packet ()
        ownd += 1
```

Figure 1: Burst mitigation pseudo-code.

3. BURST MITIGATION ALGORITHMS

Several mitigation strategies have been proposed by various researchers to control micro-bursts. The two fundamental methods that have been proposed to deal with micro-bursts are to (i) limit the number of segments sent in response to a given ACK or to (ii) spread the transmission of the burst of segments out using rate-based pacing. With regards to the former, the two basic ways that have been proposed for limiting the size of the bursts are: (a) placing a simple limit, called *maxburst*, on the transmission of new segments in response to any given ACK, and (b), scaling TCP's congestion window (*cwnd*) back to prevent a line-rate burst from being transmitted. Both of these controls are enforced on a per ACK basis. In this note we explore two variants of each basic strategy. In addition, we discuss the application of a rate-based pacing scheme to the bursting scenarios studied.

MaxBurst (MB). This mechanism, introduced in [8], is a simple limit on the number of data segments that can be transmitted in response to each incoming ACK. The `MaxBurst()` function in figure 1 provides the pseudo-code for the MB strategy. The code includes a check to ensure that the most recent ACK that arrived is valid (i.e., the cumulative acknowledgment number in the arriving segment is not below the current cumulative ACK point). After passing this check the sender's transmission of data is constrained by (i), ensuring that the amount of outstanding data (*ownd*) does not exceed *cwnd*[5], and (ii), by ensuring that at most a constant number of segments (`MB_SIZE`) are transmitted. This method lends itself to controlling the acceptable bursting by offering a direct control (`MB_SIZE`) of the allowable burst size on each ACK.

Aggressive Maxburst (AMB). We introduce this mechanism, given in the `AggressiveMaxBurst()` function in figure 1 and is similar to MB. The AMB scheme calls for the removal of the validity check on the incoming ACK used in the MB scheme. This may seem odd as [20] declares ACKs with acknowledgment numbers less than the connection's current cumulative ACK point to be "invalid". However, as shown in the next section these ACKs can be useful (at least in some cases) for clocking out new segments

[4]E.g., as discussed on the IETF's transport area working group mailing list in June, 2003. Archive at: http://www1.ietf.org/mail-archive/web/tsvwg/current/.

[5]We ignore the limit imposed by the receiver's advertised window from our discussions (and code) in this section for simplicity since the advertised window limit applies to all of the burst mitigation strategies.

during a burst mitigation phase (which was not considered in [20]).

Use It or Lose It (UI/LI). This mechanism, introduced in [12], calls for the TCP sender to monitor the size of the burst that will be transmitted in the response to an ACK arrival and reduce *cwnd* accordingly if a large line-rate burst will be transmitted. The pseudocode for the UI/LI scheme is given in the `UseItOrLoseIt()` function of figure 1. This function first compares the *ownd* and the *cwnd* to gauge whether a burst of more than a certain size (`MB_SIZE`) will be transmitted. If so, the *cwnd* is scaled back to limit the burst to no more than `MB_SIZE` segments. Under this scheme the actual sending of data is only constrained by the *cwnd*, in contrast with the two controls used in MB and AMB (in which transmission is controlled by both the `MB_SIZE` and the *cwnd*).

Congestion Window & Slow Start Threshold Limiting (CWL). We extend UI/LI with this mechanism in order to mitigate the performance penalty imposed by a potentially large decrease in the *cwnd* when using UI/LI. The `CongestionWindowLimiting()` function in figure 1 shows the pseudo-code for CWL. Like UI/LI the CWL technique compares *cwnd* to the *ownd* to detect bursts. If a burst would otherwise be sent and the value of the slow start threshold (*ssthresh*) is less than the current *cwnd* then *ssthresh* is set to the *cwnd* before the *cwnd* is reduced to mitigate the burst. This causes TCP to use slow start (exponential *cwnd* increase), as opposed to congestion avoidance (linear *cwnd* increase) to build the *cwnd* back to the point it was at when the burst was detected. In effect, CWL uses *ssthresh* as a history mechanism. This contrasts with the UI/LI scheme which leaves the method for *cwnd* increase to chance by leaving *ssthresh* untouched.

Rate-Based Pacing (RBP). While the above schemes take efforts to limit the number of segments transmitted in response to an ACK, using RBP limits the rate the segments are emitted from the sender. The benefit of RBP is that — unlike the above schemes — there is no reduction in the amount of data sent. However, as will be observed in the next section, sometimes there are natural gaps in the connection after a burst that a TCP could fill with a rate-based smoothing of a burst. However, at other times there is not a natural gap in which to send a rate-based volley of segments. If no natural pause in the transmission occurs then TCP either has to use something different from RBP or discontinue the use of traditional ACK clocking (even if temporarily) or RBP will not offer any burst mitigation.

4. SIMULATIONS

In this section we use the *ns* simulator to illustrate four different situations that cause bursts to naturally occur in TCP connections. We then illustrate how the four mitigations outlined in figure 1 cope with the burstiness. Our simulations involve a simple 4 node network with two endpoints connected by two intermediate routers. The endpoints connect to the routers using 10 Mbps links with a one-way delays of 1 ms. The routers are connected to each other using a 1.5 Mbps link with a one-way delay of 25 ms (except for the simulations involving ACK reordering given in § 4.4 which use a one-way delay of 75 ms on the link between the routers). The router employs drop-tail queueing with a maximum queue depth of 20 packets. Each endpoint uses the *sack1* variant [8] of TCP included in *ns* and delayed acknowledgments [7, 4]. Unless otherwise noted the advertised window used in these simulations is 500 segments (enough to never be a limit on sending). All simulations involve a single TCP connection.

In addition, each simulation involves some manipulation to ensure that bursting occurs (as will be described in the subsequent subsections). We note that the exact setup of the simulations and the manipulations performed are not terribly important to the results presented in this note. As will be shown, all the situations discussed in this section can occur naturally. The simulations are presented to show the stock TCP *behavior* and that of the mitigations in theoretical terms and are not a complete study of how well the mitigations work (which will be highly dependent on specific network dynamics and their prevalence).

4.1 ACK Loss

First, we explore bursts caused by ACK loss. During these simulations, all ACKs between 3.3 and 3.4 seconds of simulated time are dropped. Figure 2(a) shows a time-sequence plot of the behavior of stock TCP in the face of ACK loss. As shown in the figure, each of the missed ACKs represents a missed opportunity for the sender to transmit new data. When an ACK (finally) arrives at nearly 3.45 seconds the sender transmits a burst of roughly 20 segments.

The second two plots in figure 2 ((b) and (c)) show the behavior of MB and AMB for allowable burst sizes (`MB_SIZE`) of 3 and 5 segments respectively. In this simulation there is no difference in the behavior of MB and AMB since no ACKs arrive out-of-order. As shown the *maxburst* limit on sending reduces the size of the burst just before 3.45 seconds to 3 (or 5) segments and continues to limit the sending of segments to no more than 3 (or 5) segments per ACK until the *ownd* reaches the size of *cwnd*. The time required to build *ownd* to *cwnd* is directly related to the choice of the `MB_SIZE` parameter used. When using an `MB_SIZE` of 3 segments the *ownd* increase takes roughly 150 ms longer than when `MB_SIZE` is 5 segments in the sample case.

Figure 2(d) shows the behavior of UI/LI (with an `MB_SIZE` of 3) in the face of ACK loss. When the burst is detected just before 3.45 seconds *cwnd* is reduced, followed by roughly 1 second of slow start. The amount of slow start (if any) used after the burst mitigation under UI/LI is arbitrary and depends on the value of *ssthresh* before the burst is detected. If the *cwnd* is less than *ssthresh* after UI/LI, then slow start will be used until *cwnd* reaches *ssthresh*. However, if *cwnd* is not reduced to below *ssthresh*, then the linear increase of congestion avoidance will be used.

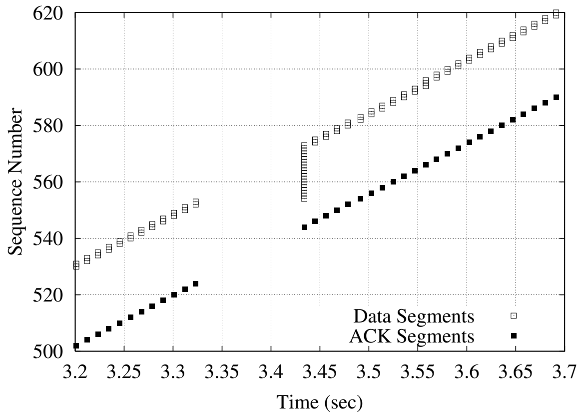
Figure 2(e) shows the behavior of CWL when faced with ACK loss. As with UI/LI, *cwnd* is reduced just before 3.45 seconds. However, as discussed in § 3, CWL sets *ssthresh* to *cwnd* (assuming the connection is using congestion avoidance) before scaling back *cwnd* to prevent the burst. This provides a sort-of history that helps the connection return to its pre-burst state and provides more determinism in the *cwnd* growth after burst mitigation than UI/LI. As shown, CWL utilizes slow start to increase the *cwnd* for almost 2 seconds yielding a larger *cwnd* when compared to UI/LI.

We also note that subfigures (b) and (e) are exactly the same in this situation. That is, MB, AMB and CWL provide the same effective response to the burst in terms of data sent into the network when `MB_SIZE` is 3 segments, even though the methodology is different.

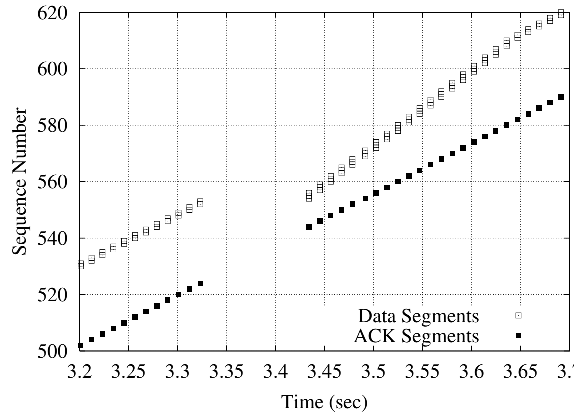
Finally, we note that in the case of the burst depicted in this situation (and sent in figure 2(a)), a rate-based pacing scheme would have no natural lull over which to spread the burst of segments.

4.2 Limited Advertised Window

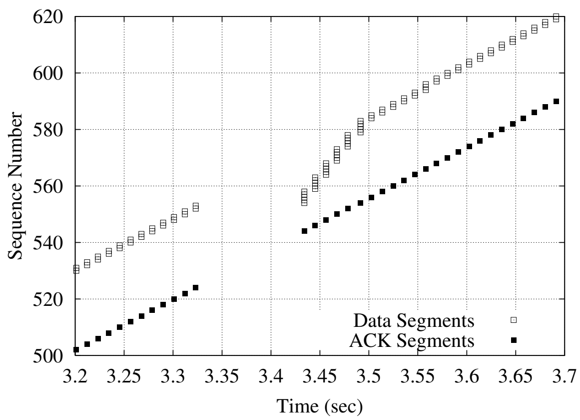
In the next scenario we explore bursting as caused by the advertised window during loss recovery. For this set of simulations we set the advertised window to 32 segments. Figure 3(a) shows the time-sequence plot of standard TCP's behavior. The TCP sender is able to fill the advertised window and then takes a single loss. Fast retransmit [13, 4] is used to retransmit the segment. Fast recovery should then take over and clock out new segments during the sec-



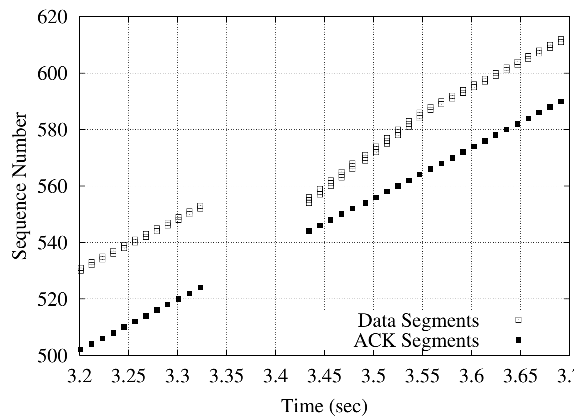
(a) Stock TCP



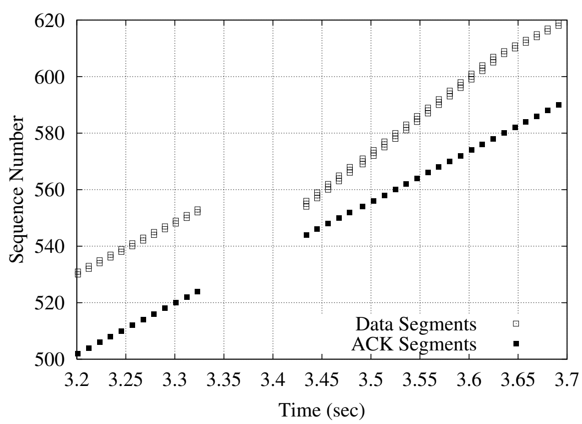
(b) MB 3 (and AMB 3)



(c) MB 5 (and AMB 5)



(d) UI/LI 3



(e) CWL 3

Figure 2: ACK loss induced bursting behavior.

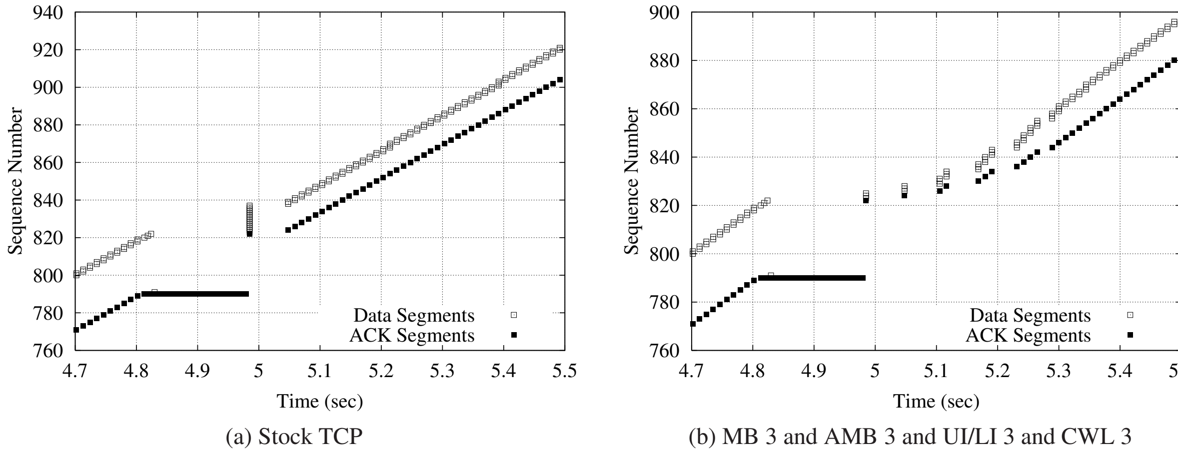


Figure 3: Advertised window induced bursting behavior.

ond half of the loss recovery period. However, because the sender has filled the advertised window no new data can be sent. Therefore, when the retransmitted segment is ACKed (along with the rest of the outstanding data – in this case) a burst of data is transmitted into the network. In the situation shown in figure 3(a) the burst is nearly 20 segments. This phenomena has been observed and detailed elsewhere for different strategies of TCP loss recovery and different network environments [8, 11].

While algorithmically different, all the burst mitigation strategies perform identically in this situation (with a common `MB_SIZE` of 3), as shown in figure 3(b). Since the *ownd* is collapsed to zero by the large cumulative ACK that arrives just before 5 seconds, all of the schemes start from the same place. Further, with an `MB_SIZE` of 3 segments the *maxburst*-based schemes exactly mimic slow start with delayed ACKs and no ACK loss (where each ACK clocks out 3 data segments). Finally, the non-determinism shown in UI/LI in the last subsection is not present because *ssthresh* is set to a known point at the time of the fast retransmit.

Unlike the ACK loss case explored in the last section, the advertised window limit shown in this section offers a reasonably straightforward situation in which to use rate-based pacing. In this case, all the data has drained from the network and so there is a lull in activity after the burst is transmitted that lasts roughly one RTT (from just before 5 seconds to around 5.05 seconds). Therefore, an RBP scheme could easily space out the segments evenly over the course of the RTT following the burst detection.

4.3 Application Limiting

The next case of bursting we examine is caused by the application layer protocol's sending pattern. Figure 4(a) shows the time-sequence plot of a TCP transfer where the application does not send data from just after 0.65 seconds until 0.8 seconds. The plot shows that no data is sent in this interval even as ACKs arrive. However, when the application begins sending again at 0.8 seconds the underlying TCP transmits a burst of roughly 20 segments. The burst caused by such an idle period can be mitigated by using an idle timer (as introduced in [13] and discussed in [12]). After the idle timer fires the TCP connection must start sending with a small *cwnd* (per RFC 3390 [3]) and use slow start to increase *cwnd*. In addition, Congestion Window Validation (CWV) [10] can come into play in this scenario. CWV calls for TCP to use only "valid" window sizes — i.e., windows that have been fully utilized and therefore are known to be reasonable, but windows that are not fully used are not known to be appropriate for the current network conditions and therefore the window will be reduced.

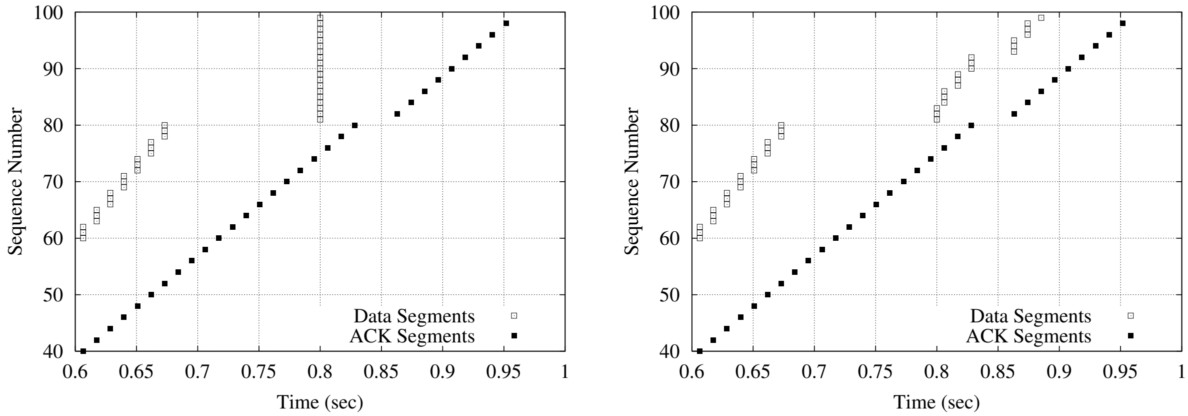
Figure 4(b) shows the behavior of all mitigations given in figure 1 in the face of the application sending pattern sketched above. As in the last subsection MB, AMB, UI/LI and CWL perform the same in this simulation (with a common `MB_SIZE` of 3 segments). MB and AMB perform the same because there are not out-of-order ACKs in this simulation. The *maxburst* schemes perform the same as the strategies based on limiting the *cwnd* because both *maxburst* and slow start call for transmitting 3 segments on each ACK received. In this simulation UI/LI and CWL perform the same. However, as discussed above the non-determinism of UI/LI does not guarantee that these two schemes will behave the same. In particular, CWL could use slow start longer than UI/LI (as shown in § 4.1), yielding a larger *cwnd*.

Application limited situations sometimes present a straightforward opportunity for using RBP, while at other times offering a more muddled situation. For instance, if all data on a connection has drained from the network and is acknowledged and then the application produces more data the TCP sender can easily pace out the congestion window over its estimate of the RTT. The flip side is shown in figure 4(a), whereby there is a period where no data is available for transmission, but not all the data drains from the network. Therefore, when the burst occurs there is still an ACK clock and there is not a natural gap in the data transmission over which the burst can be smoothed.

4.4 ACK Reordering

The last bursting situation we examine in detail involves ACK reordering[6]. [5] finds that packet reordering is not a rare occurrence over the MAE-East exchange, suggesting that ACK reordering may not be an uncommon phenomenon on at least some network paths. Figure 5(a) shows the behavior of stock TCP in the face of ACK reordering. In the simulation we changed the delay imposed on

[6]This is not to say that bursting does not occur in additional situations. However, we believe the four we sketch in this note cover the space of general types of bursting scenarios.



(a) Stock TCP

(b) MB 3 and AMB 3 and UI/LI 3 and CWL 3

Figure 4: Bursting caused by application layer sending patterns.

the link between the routers that carries the ACKs from 75 ms to 1 ms at 6.32 seconds and then back to 75 ms at 6.33 seconds. This caused a single ACK to "pass" a number of previously sent ACKs in the trip from the receiver to the sender. When this ACK arrives the TCP window slides and a burst of segments is sent, as shown around 6.33 seconds in figure 5(a). Following the burst, a number of ACKs arrive that are not used to clock out data segments because (a) the ACKs convey no new information and (b) the *cwnd* is full.

Figure 5(b) shows the behavior of the MB technique (with an `MB_SIZE` of 3 segments). The burst limit does not allow the full use of *cwnd* until just after 6.5 seconds. Figure 5(c) shows the behavior of AMB, which uses each "invalid" ACK to clock out an `MB_SIZE` burst of segments. While these ACKs convey no new information for the connection, from a reliability standpoint, they can be used to clock out new segments because, unlike stock TCP, the TCP is not utilizing the entire window due to the burst mitigation. As shown in the figure, the last two ACKs are, in fact, not used to clock new data into the network. This is explained by the TCP sending 3 segments on each of the previous invalid ACKs, rather than 2 segments as TCP would normally transmit during congestion avoidance. Therefore, the *cwnd* is filled using less ACKs than normal and so the last two "invalid" ACKs are ignored.

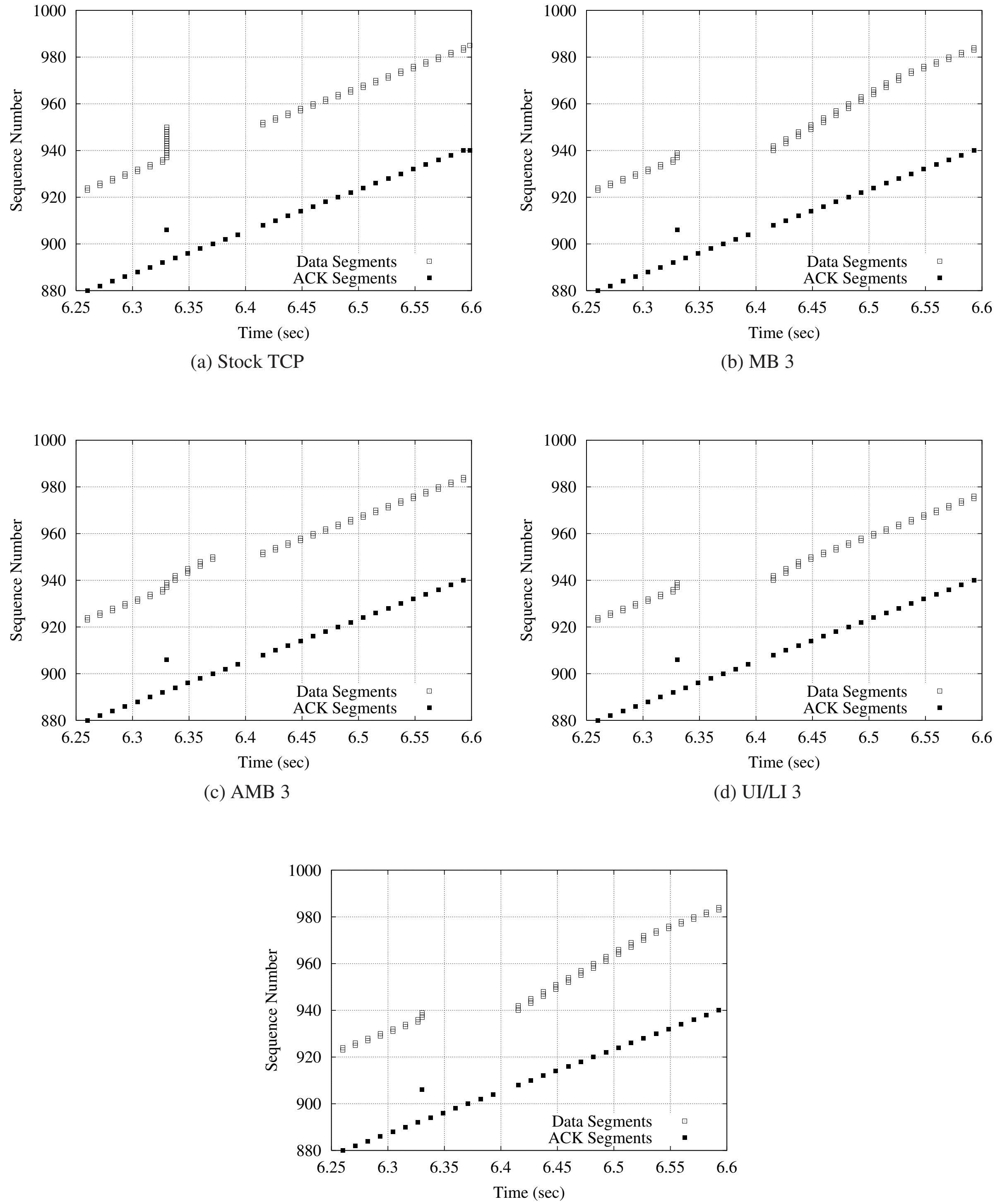
Figure 5(d) shows the behavior of the UI/LI technique. This figure shows that when the burst is detected (just after 6.4 seconds) the *cwnd* is clamped to mitigate the burst and congestion avoidance (linear *cwnd* increase) ensues. Finally, figure 5(e) shows the behavior of CWL. In contrast to the UI/LI scheme, CWL utilizes slow start to increase *cwnd* to the value it had prior to the burst detection. As in the previous sections, MB and CWL show identical on-the-wire behavior in our simulations, even though the two schemes use different methods for obtaining their behavior.

ACK reordering presents a tricky situation for RBP. As shown in figure 5(a), there is a natural lull in the connection after the burst is transmitted. At first glance, it may seem natural to attempt to smooth the burst over this pause. However, the reception of the ACK that causes the burst could indicate either ACK reordering (as is the case in figure 5) or simply a case of dropped ACKs (as discussed previously). If the sender could know that ACK reordering was the root cause then conceivably RBP could be used over an interval that depends on the length of the reordering. On the other hand, if the root cause was known to be dropped ACKs then there is no clear way to utilize RBP. Without knowledge about the cause of a larger than expected cumulative ACK it is difficult to make sound decisions as to what course of action to take.

5. CONCLUSIONS

This note's contribution is in (i) the methodical analysis of the behavior of several burst mitigation schemes and (ii) the extension of several previously defined burst mitigation strategies. In doing so, several high-level points have surfaced:

- The behavior and performance of UI/LI is dependent on the congestion control state when UI/LI is invoked. We introduced the notion of using *ssthresh* as a history mechanism to avoid this non-determinism in CWL.
- If faster than slow-start transmission rate increase is desired after a burst is detected then MB or AMB are needed because *cwnd*-based schemes can increase the transmission rate no faster than slow start. The flip side of this issue is the question of whether it is safe to increase faster than slow start would. We suspect that the answer is that it is indeed safe, given that the connection is increasing only to a previously (and recently) known appropriate operating point.
- CWL provides a *single control* for the amount of data a TCP connection can transmit into the network at any given point. This is arguably a clean approach to controlling the load imposed on the network. On the other hand, MB provides for *separation of concerns*. In other words, limiting the sizes of micro-bursts is, in some sense, a different task than limiting the overall transmission rate to control network congestion. Therefore, using two different mechanisms may make sense. As noted above, the MB scheme is more flexible than the CWL scheme. However, an additional drawback is that MB adds a second control and brings with it the possibility of the two transmission controllers interacting poorly and causing problems.
- The simulations in § 4.4 shows that there are times when traditionally discarded "invalid" ACKs could be useful in keeping the ACK clock going. Of course, these ACKs have been



(a) Stock TCP

(b) MB 3

(c) AMB 3

(d) UI/LI 3

(e) CWL 3

Figure 5: Bursting caused by ACK reordering.

traditionally disregarded for a reason and these ACKs could be bogus for any number of reasons (network duplicates, old segments from previous connections, etc.). Therefore, careful thought is required before using such ACKs to trigger further data transmission.

There are pros and cons to all of the strategies studied in this note. Therefore, we do not concretely find any one "best" mechanism. Rather, we hope that this note provides useful information for researchers and implementers to use when reasoning about the various possibilities.

Acknowledgments

Armando Caro, Sally Floyd, Tom Henderson, Kacheong Poon, Scott Shenker and Randall Stewart provided helpful discussions about the topics covered in this note. The first author's work was funded by NSF grant number 0205519. Our thanks to all!

6. REFERENCES

[1] Amit Aggarwal, Stefan Savage, and Tom Anderson. Understanding the Performance of TCP Pacing. In *IEEE INFOCOM*, March 2000.

[2] Mark Allman. TCP Congestion Control with Appropriate Byte Counting (ABC), February 2003. RFC 3465.

[3] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, October 2002. RFC 3390.

[4] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[5] Jon Bennett, Craig Partridge, and Nicholas Sheetman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, December 1999.

[6] Ethan Blanton and Mark Allman. On the Impact of Bursting on TCP Performance. In *Passive and Active Measurement Workshop*, March 2005.

[7] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[8] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[9] Sally Floyd and Eddie Kohler. Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control, March 2005. Internet-Draft draft-ietf-dccp-ccid2-10.txt (work in progress).

[10] Mark Handley, Jitendra Padhye, and Sally Floyd. TCP Congestion Window Validation, June 2000. RFC 2861.

[11] Chris Hayes. Analyzing the Performance of New TCP Extensions Over Satellite Links. Master's thesis, Ohio University, August 1997.

[12] Amy Hughes, Joe Touch, and John Heidemann. Issues in TCP Slow-Start Restart After Idle, December 2001. Internet-Draft draft-hughes-restart-00.txt (work in progress).

[13] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[14] Hao Jiang and Constantinos Dovrolis. Source-Level IP Packet Bursts: Causes and Effects. In *ACM SIGCOMM/Usenix Internet Measurement Conference*, October 2003.

[15] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram Congestion Control Protocol (DCCP), March 2005. Internet-Draft draft-ietf-dccp-spec-11.txt (work in progress).

[16] Craig Partridge. ACK Spacing for High Delay-Bandwidth Paths with Insufficient Buffering, September 1998. Internet-Draft draft-rfced-info-partridge-01.txt (work in progress).

[17] Craig Partridge, Dennis Rockwell, Mark Allman, Rajesh Krishnan, and James P.G. Sterbenz. A Swifter Start for TCP. Technical Report TR-8339, BBN Technologies, March 2002.

[18] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[19] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, March 1999. RFC 2525.

[20] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[21] Randall Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Juergen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. Stream Control Transmission Protocol, October 2000. RFC 2960.

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

[23] Lixia Zhang, Scott Shenker, and David Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two- Way Traffic. In *ACM SIGCOMM*, September 1991.