Reducing Kernel Queuing Delays with TCP Window Space Events

David Goulet
The Tor Project
dgoulet@torproject.org

Rob Jansen
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Abstract

The combination of TCP auto-tuning and asynchronous I/O event notifications (e.g., `epoll`) allows the Linux kernel to generally sustain high-volume TCP connections—even for connections with high bandwidth-delay products (high link bandwidth and/or high path latency). However, bufferbloat can quickly become an issue when multiple such connections are in use. In particular, high outbound kernel queuing delays have been observed in the Tor anonymity network, a large distributed system whose relays often manage thousands of sockets—many of which are simultaneously-active, high-volume TCP connections.

In this work, we propose a new notification event that triggers when *TCP is ready to send data* on a socket while seeking to better understand how it can help applications to better manage network I/O and improve performance. The new event supplements and extends the current write event that triggers when *a socket buffer has free space*, and the difference in semantics allows more precise control over queuing to the application. We describe the problem, detail a proposal for extending `epoll` to support the new semantics (including a code patch), and show the effect that such a change could have on performance through a small scale simulation.

1 Introduction

High performance networking applications require facilities that allow for precise interactions between user space and kernel space. In order for a network application to reach high throughput, it needs to ensure that the kernel buffers associated with a network socket have enough data so that the kernel can package and send packets on demand.

The state of a socket can generally be managed using an I/O event notification facility such as `epoll`: the application creates an `epoll` descriptor and uses it to "listen" for when the socket becomes readable (the `EPOLLIN` event) and writable (the `EPOLLOUT` event). The application uses the "socket is writable" event notification to trigger the continued writing of data from user-space into the kernel such that the kernel is not starved of data (the socket buffer has enough data available when the kernel wants to send). The kernel then attempts to maximize the TCP sending rate by using TCP auto-tuning to monotonically increase the size of the write buffer over time, ensuring that TCP can keep a full bandwidth-delay product worth of bytes in flight.

Bufferbloat: Although the `epoll` facility works well for maintaining high sending rates (there is always enough data for the kernel to send when it can), it has also been shown to result in bufferbloat and high kernel queuing times [3]. This is a particular problem for nodes in large distributed systems like the Tor anonymity network [1] that may have thousands of sockets open at any time. Large kernel outbound queuing delays were observed at Tor relays as they would attempt to keep all of the socket buffers full whenever `EPOLLOUT` events were triggered, but the link rates of the relays were not high enough to send all of the data that Tor was writing to the kernel socket write buffers [4]. This bufferbloat problem was identified, measured, and shown to harm Tor's ability to maintain control over traffic priority [4].

TCP Window Space: We would like to provide more control over when the socket write event is delivered by the kernel in order to allow applications with many sockets to reduce and control kernel bufferbloat. We propose to add a new feature to the `epoll` event notification facility that causes `EPOLLOUT` events to be triggered when *TCP is ready to send data* rather than when *a socket buffer has free space*. This is done by calculating the number of bytes that are needed to fill the TCP congestion window; if the number of such bytes are greater than a configurable threshold, we trigger an `EPOLLOUT` event on the socket. The application can then proceed to write enough bytes to fill the window, thus reducing the kernel outbound buffer lengths.

Outline: We next describe some previous work that employed the TCP window space concept in §2. We then specify the kernel changes necessary to implement our proposal in §3. Finally, we describe the results from a small-scale simulation of the concept in §4 before concluding in §5.

2 Related Work

Bufferbloat in Tor: Researchers from the Tor community designed [3], developed [4], and deployed [6] an algorithm called KIST (Kernel-Informed Socket Transport) to mitigate the congestion caused by overfull buffers. The main idea of KIST is to limit the number of bytes written to a socket buffer to only the number of bytes that TCP is willing to send out onto the network, reducing kernel queuing and increasing Tor's control over data priority.

The KIST algorithm works by continuously (every 10 ms) tracking the TCP state on the active sockets opened by Tor using calls to `getsockopt` on level `TCP_SOL` for option `TCP_INFO`. Then, KIST computes a write limit for each socket:

$$tcp_space \leftarrow (cwnd - una) \cdot mss \quad (1)$$

$$write_limit \leftarrow tcp_space - notsent \quad (2)$$

where *mss* is the maximum segment size, *cwnd* is the number of packets in the congestion window, *una* is the number of unacked packets, and *notsent* is the number of bytes written to the socket buffer that have not yet been sent. The write limit ensures that the kernel will be able to immediately send any data the application is writing, and it has been shown to significantly reduce outbound kernel buffering time by reducing write buffer lengths [4].

The KIST idea is well-reasoned: there may be no need to send large amounts of data to kernel buffers if TCP will prevent it from being sent out anyway. However, the method that KIST uses to achieve its result is not ideal for the following reasons:

- continuously polling sockets for *TCP_INFO* is a nonoptimal solution that does not scale (requires many syscalls);
- the information that the application obtains may quickly become stale as new packets arrive in the kernel; and
- it takes up to a full polling interval (10 ms) before TCP state changes are recognized by the application.

Therefore, the application has to write more than the *tcp_space* to ensure that the kernel has data to send when previously sent data becomes acknowledged by the receiver.

We strive for a more elegant solution (fewer syscalls) that is implemented in kernel space so that any application that wants more control over bloated outbound buffers can benefit.

Related Kernel Functionality: Since the development of KIST, a related feature has been developed and deployed in the Linux kernel that allows an application to adjust when it receives `EPOLLOUT` events from the kernel. The feature is set with the TCP option `TCP_NOTSENT_LOWAT` and will cause the kernel to report `EPOLLOUT` event only if the number of *notsent* bytes (bytes written to the socket but not yet sent to the network) is below the lowater setting: i.e., it reports `EPOLLOUT` when the buffer is almost empty (ignoring TCP). This is distinct from the KIST approach in two significant ways:

- `TCP_NOTSENT_LOWAT` considers the number of notsent bytes in the socket buffer, whereas KIST considers the number of bytes in the socket buffer *in addition to* the TCP congestion window. In particular, KIST considers the TCP congestion window (*cwnd*) and the number of packets that have been sent but not yet acknowledged (*una*), and only report `EPOLLOUT` when TCP would actually be capable of sending packets. This is important in the case when the number of *notsent* bytes is low, but TCP will still prevent sending packets because *cwnd* is closed or because it is waiting for ACKs.
- `TCP_NOTSENT_LOWAT` will *always* report `EPOLLOUT` if *notsent* bytes is below the lowater threshold. Our proposal in §3 is to report `EPOLLOUT` only if TCP can send data (see above) *and* we will be able to write *at least* a configurable minimum number of bytes.

Tor relays have thousands of open TCP connections, many of which are long-lived and whose congestion state is highly dynamic over time. We believe that our proposal will better address this use-case.

3 Design

We propose to add a new feature to the `epoll` event notification facility that causes `EPOLLOUT` events to be triggered when the number of bytes required to fill the TCP window is greater than some threshold value w.

To each TCP socket in `struct tcp_sock` we add a new state variable called `pollout_window_min_len`; this variable simultaneously indicates if the socket is in our new TCP window polling mode and also stores the value of the threshold w. The user enables TCP window polling mode through a call to `setsockopt` with a positive value for the new option `TCP_POLLOUT_WIN_LEN`, which the kernel will store in the new `pollout_window_min_len` variable.

When the kernel polls the I/O status of sockets as usual in `tcp_poll` and the value of `pollout_window_min_len` is positive, it computes the space available in the TCP window following Equation 1:

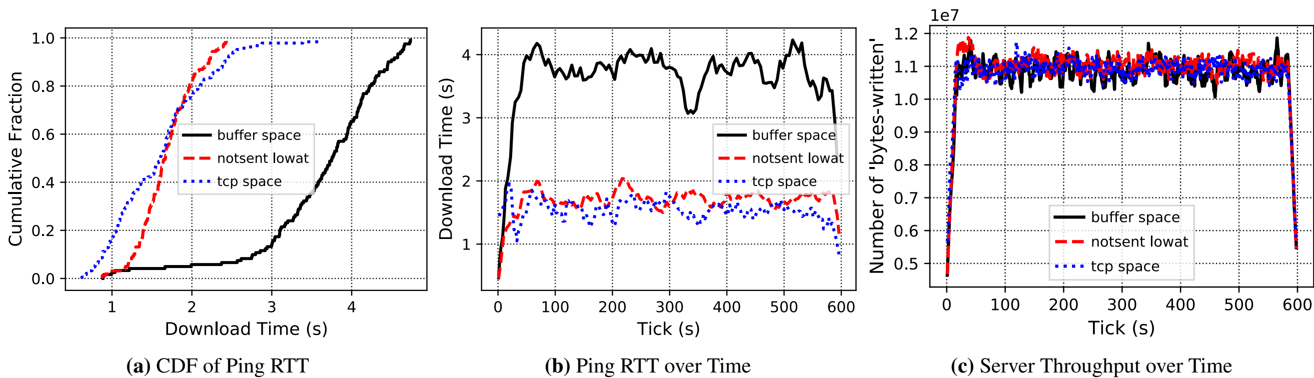
$$space \leftarrow (snd_cwnd - sk_ack_backlog) \cdot mss_cache$$

and an EPOLLOUT event will only be emitted if the computed space is at least `pollout_window_min_len`.

Please refer to Appendix A for a full code listing of a relatively small kernel patch required to implement our design as described above.

4 Experiment

To test our proposed modification to the epoll facility, we ran a small-scale simulation using the Shadow simulator [2] and its TGen traffic generator [5]. Shadow is a hybrid discrete-event network simulator that directly executes applications in a network simulation environment. We configured our Shadow experiment to run a single TGen server with 100 Mbit/s symmetric bandwidth. We also configured 25 TGen clients that



(a) CDF of Ping RTT **(b)** Ping RTT over Time **(c)** Server Throughput over Time

Figure 1: Round trip times through the server as measured from the TGen ping clients, and server throughput over time.

repeatedly download 10 MiB files, pausing after each download completes for a time selected uniformly at random from the range [1,3] seconds. Finally, we configured a single TGen ping client that attempts to measure RTT through the server using a TCP connection, pausing for 1 second after completing each measurement. All clients were placed in random cities in Shadow's global Internet map (which models Internet latency according to RIPE Atlas measurements [5]), and the server was placed at a random city in the US. The experiment was configured to run for 10 simulated minutes.

We ran the experiment 3 times, once using the traditional `epoll` semantics where an `EPOLLOUT` event is emitted whenever there is positive *buffer space*, once with the related *notsent lowat* kernel feature, and once using our proposed design that emits `EPOLLOUT` whenever the *TCP space* is greater than a threshold as described in Section 3. We set `tcp_notsent lowat` to 1 byte and `pollout_window_min_len` to the maximum segment size in our experiments. The primary results of the experiments are shown in Figure 1.

Figure 1a shows that the round trip time through the server is significantly reduced—by more than 2 seconds in the median—when the server is configured to emit `EPOLLOUT` based on the TCP window space rather than the buffer space. The same data is again shown in Figure 1b, showing how the round trip time measurements vary over time throughout the 10 minute (600 second) experiments. Finally, Figure 1c shows that the server maintained near full link utilization (100 Mbit/s) throughout both experiments.

5 Conclusion

Our results show that significant improvements in responsiveness may be possible for some applications, confirming indications from previous work on the Tor anonymity network [4]. We observed that our proposed design works as well as the existing *notsent lowat* feature in this small-scale experiment. Although our experiments are based on simulation, we believe that our proposed modification to the `epoll` event notification facility could be beneficial to many applications who want more control over the buffering behavior in the kernel. Further testing in a deployed Linux kernel is required to better understand the performance trade-offs. In the meantime, we hope to collect feedback on our current design and code patch and better understand the path forward for getting the code successfully merged.

Acknowledgments

This work was supported by the Office of Naval Research.

References

[1] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[2] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[3] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium*, 2014.

[4] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. KIST: Kernel-Informed Socket Transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):3:1–3:37, December 2018.

[5] R. Jansen, M. Traudt, and N. Hopper. Privacy-preserving dynamic learning of Tor network traffic. In *25th ACM Conference on Computer and Communications Security (CCS)*, 2018. See also `https://tmodel-ccs2018.github.io`.

[6] The Tor Project. KIST and Tell: Tor's New Traffic Scheduling Feature. `https://blog.torproject.org/kist-and-tell-tors-new-traffic-scheduling-feature`, October 2017. Blog Post.

A Kernel Patch Code Listing

DISTRIBUTION STATEMENT: Approved for public release: distribution unlimited.
Redistributions of source and binary forms, with or without modification, are permitted if redistributions retain the above distribution statement and the following disclaimer.
DISCLAIMER: The software is supplied "as is" without warranty of any kind.
As the owner of the software, the United States, the United States Department of Defense, and their employees:

1. disclaim any warranties, express or implied, including but not limited to any implied warranties of merchantability, fitness for a particular purpose, title or non-infringement,
2. do not assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the software,
3. do not represent that use of the software would not infringe privately owned rights,
4. do not warrant that the software will function uninterrupted, that it is error-free or that any errors will be corrected.

Portions of the software resulted from work developed by or for the U.S. Government subject to the following license: the Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in this computer software to reproduce, prepare derivative works, to perform or display any portion of that work, and to permit others to do so for Government purposes.

```
diff --git a/include/linux/tcp.h b/include/linux/tcp.h
index a9b0280687d5..54f2dbb9730f 100644
--- a/include/linux/tcp.h
+++ b/include/linux/tcp.h
@@ -401,6 +401,8 @@ struct tcp_sock {
          */
        struct request_sock *fastopen_rsk;
        u32     *saved_syn;
+
+       u32 pollout_window_min_len;
 };

 enum tsq_enum {
diff --git a/include/uapi/linux/tcp.h b/include/uapi/linux/tcp.h
index e02d31986ff9..a68e662bf109 100644
--- a/include/uapi/linux/tcp.h
+++ b/include/uapi/linux/tcp.h
@@ -124,8 +124,9 @@ enum {
 #define TCP_FASTOPEN_NO_COOKIE 34 /* Enable TFO without a TFO cookie */
 #define TCP_ZEROCOPY_RECEIVE   35
 #define TCP_INQ                36 /* Notify bytes available to read as a cmsg on read */
+#define TCP_POLLOUT_WIN_LEN    37 /* POLLOUT event is based on TCP out queue. */

-#define TCP_CM_INQ             TCP_INQ
+#define TCP_CM_INQ             TCP_POLLOUT_WIN_LEN

 #define TCP_REPAIR_ON          1
 #define TCP_REPAIR_OFF         0
diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
index 40cbe5609663..1e36091c1562 100644
--- a/net/ipv4/tcp.c
+++ b/net/ipv4/tcp.c
@@ -493,6 +493,19 @@ static inline bool tcp_stream_is_readable(const struct tcp_sock *tp,
                sk->sk_prot->stream_memory_read(sk) : false);
 }

+static inline bool tcp_stream_window_has_space(const struct tcp_sock *tp,
+                                            struct sock *sk)
+{
+       /* This computes how much room we have before we hit the limit of the
+        * congestion window. The idea for this is to take the congestion window
```

```
+        * size minus the unacked packet times the segment window size.
+        *
+        * It results in how much more we can put in the window before reaching
+        * its limit. */
+       u64 window_space = (tp->snd_cwnd - sk->sk_ack_backlog) * tp->mss_cache;
+       return (window_space >= tp->pollout_window_min_len);
+}
+
 /*
  *      Wait for a TCP event.
  *
@@ -566,7 +579,12 @@ __poll_t tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
                        mask |= EPOLLIN | EPOLLRDNORM;

                if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
-                       if (sk_stream_is_writeable(sk)) {
+                       /* If the socket has been set to wakeup if the TCP out queue has
+                        * enough room for the user defined length. */
+                       if (tp->pollout_window_min_len > 0 &&
+                                       tcp_stream_window_has_space(tp, sk) &&
+                                       sk_stream_is_writeable(sk)) {
+                               mask |= EPOLLOUT | EPOLLWRNORM;
+                       } else if (tp->pollout_window_min_len == 0 &&
+                                       sk_stream_is_writeable(sk)) {
                                mask |= EPOLLOUT | EPOLLWRNORM;
                        } else {  /* send SIGIO later */
                                sk_set_bit(SOCKWQ_ASYNC_NOSPACE, sk);
@@ -3054,6 +3072,12 @@ static int do_tcp_setsockopt(struct sock *sk, int level,
                else
                        tp->recvmsg_inq = val;
                break;
+       case TCP_POLLOUT_WIN_LEN:
+               if (val < 0)
+                       err = -EINVAL;
+               else
+                       tp->pollout_window_min_len = val;
+               break;
        default:
                err = -ENOPROTOOPT;
                break;
```