Adventures in TCP/IP

Pacing in the FreeBSD TCP Stack

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP sending and receiving behavior has evolved over the more than 40 years that TCP has been used. Many of the advances have helped TCP to be able to transmit a reliable stream of data at very high speeds. However, some of the enhancements (both in the stack and in the network) come with downsides. Originally, a TCP stack sends a TCP segment in response to a received TCP segment (an acknowledgment), or in response to the upper layer providing new data to send, or due to a timer running off. The TCP sender also implements a congestion control to protect the network against sending too fast. These features combined can cause a TCP endpoint to be driven most of the time by an "ACK clock" as depicted in the following figure.

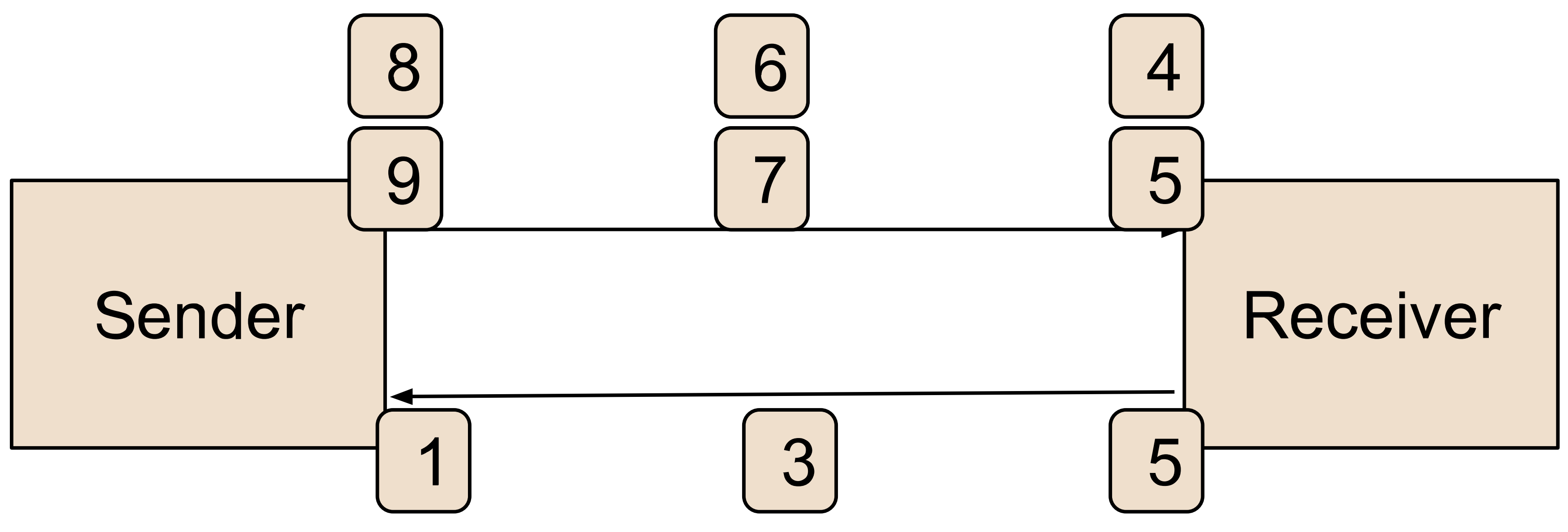

Figure 1: An example of ACK-clocking

For simplicity it is assumed that packets are numbered and acknowledged and that the receiver acknowledges every other packet to minimize the overhead. The arrival of the acknowledgment labeled "1" on the bottom arrow acknowledges packets 0 and 1 that were sent earlier (not shown in the figure), removes two of the outstanding packets, and allows two more to be sent aka packets 8 and 9. At that point the sender is blocked (due to its congestion control) waiting for the arrival of the ACK labeled "3" which will acknowledge packet 2 and 3 and again send out the next two packets. This ACK-clocking was prevalent in a large number of flows in the early Internet and can still be seen today in some circumstances. ACK-clocking forms a natural pacing of data through the Internet allowing packets to be sent through a bottleneck, and oftentimes, by the time the next packets from the sender arrives at the bottleneck, the previous packets are transmitted.

However, over the years, both the network and TCP optimizations have changed this behavior. One example of a change in behavior is often seen in cable networks. In such networks the down link bandwidth is large, but the uplink bandwidth is small (often only a fraction of the downlink bandwidth).

Due to this scarcity of the return path bandwidth, the cable modems will often keep only the last acknowledgment sent (assuming it is seeing the acknowledgments in sequence) until it decides to transmit. So, in the above example, instead of allowing ACK-1, ACK-3 and ACK-5 to be transmitted by the cable modem, it might only send ACK-5. This would then cause the sender to send one larger burst of 6 packets, instead of spacing out 3 separate bursts of two packets.

Another example of a modifying behavior can be seen in other slotted technologies (they hold off sending until their time slot is reached, and then they send out all queued packets at that time) where the acknowledgments are queued up and then all sent at once in one burst of 3 ACK packets. This type of technology would then interplay with TCP-LRO (talked about in our last column) and thus either collapse the acknowledgments into one single acknowledgment (if the old methods are being used) or queue up for simultaneous processing of all of the acknowledgments before the sending function is called. In either case, a large burst is again sent instead of a series of small two packet bursts, separated by a small increment of time (closely approximating the bottleneck bandwidth plus some propagation delay).

Another example of a modifying behavior can be seen in other slotted technologies.

Our example shows only six packets but several dozen packets can burst all in one `tcp_output()` call. This is good for CPU optimization but can cause packet loss in the network since router buffers are limited and large bursts are more likely to cause a tail drop. This loss would then reduce the congestion window and hinder overall performance.

In addition to the two examples given above, there are other reasons that TCP can become bursty (some of which have always been inherent in TCP) such as application limited periods. This is where a sending application stalls for some reason and delays the sending for some number of milliseconds. During that time, the acknowledgments arrive but there is no more data to send. But before all the data from the network is drained, which might cause a congestion control reduction due to idleness, the application sends down another large block of data to be sent. In such a case, the congestion window is open and a large sending burst can be generated.

Yet another source of sources of burstiness may come from the peer TCP implementations that might decide to acknowledge every eighth or sixteenth packet instead of following the TCP standard and acknowledging every other packet. Such large stretch acknowledgments will again cause corresponding bursts. TCP pacing, described in the next section, is a way to improve the sending of TCP segments to smooth out these bursts.

TCP Pacing

The following example illustrates TCP pacing. Let's assume that a TCP connection is used to transfer data to its peer at 12 megabits per second including the IP and TCP header. Assuming a maximum IP packet size of 1500 bytes (which is 12,000 bits), this results in sending 1000 TCP segments per second. When using pacing, sending a TCP segment every millisecond would be used. A timer that runs off every millisecond could be used to achieve this. The following figure illustrates such a sending behavior.

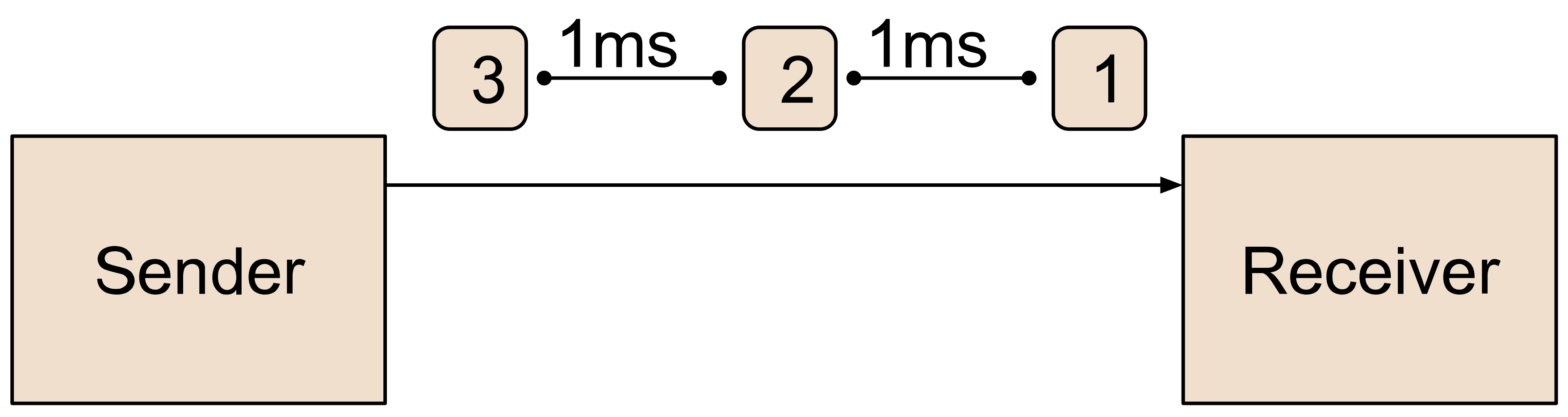

Figure 2: A TCP connection paced at 12Mbps

Doing pacing in such a manner is possible but not desirable due to the high CPU cost this would take. Instead, for efficiency reasons, when pacing, TCP sends small bursts of packets with some amount of time between them. The size of the burst is generally correlated to the speed that the stack wishes to pace at. If pacing at a higher rate, larger bursts are in order. If pacing at a lower rate, smaller bursts are used.

When designing a methodology for pacing a TCP stack, there are a number of approaches that can be taken. A common one is to have the TCP stack set a rate in a lower layer and then hand off large bursts of data to be sent to that layer. The lower layer then just multiplexes packets from various TCP connections with appropriate timers to space out the data. This is not the approach taken in the FreeBSD stack, specifically because it limits the control the TCP stack has over the sending. If a connection needs to send a retransmission, that retransmission ends up falling in behind all the packets that are in queue to be sent.

In FreeBSD, a different approach was taken in letting the TCP stack control the sending and creating of a timing system that is dedicated to calling the TCP stack to send data on a connection when the pacing interval has ended. This leaves complete control of what to send in the hands of the TCP stack, but can have some performance implications that need to be compensated for. This new subsystem created for FreeBSD is described in the next section.

High Precision Timing System

Conceptual Overview

The High Precision Timing System (HPTS) is a loadable kernel module and provides a simple interface to any TCP stack wishing to use it. Basically, there are two main functions that a TCP stack would call to get service from HPTS:

- `tcp_hpts_insert()` inserts a TCP connection into HPTS to have either `tcp_output()` or, in some cases, TCP's inbound packet processing call `tfb_do_queued_segments()` at a specified interval.
- `tcp_hpts_remove()` asks HPTS to remove a connection from HPTS. This is often used when a connection is closed or otherwise no longer going to send data.

There are some other ancillary helping functions that are available in HPTS to help with timing and other housekeeping functions, but the two functions listed above are the basic building blocks that a TCP stack uses to implement pacing.

Details

Internally, each CPU has an HPTS wheel, which is an array of lists of connections wanting service at various time points. Each slot in the wheel represents 10 microseconds. When a TCP connection is inserted, it is given the number of slots from now (i.e., 10 microsecond intervals) that need to elapse before the `tcp_output()` function is called. The wheel is managed by a combination of both a system timer (i.e., FreeBSD's callout system) and a soft timer as proposed in [1]. Basically every time a system call returns, before the return to user space, HPTS can potentially be called to see if an HPTS wheel needs to be serviced.

The HTPS system also auto tunes its FreeBSD system timer having first a minimum (defaulting to 250 microseconds) and a maximum that it can tune up. If an HPTS wheel has more connections and is getting called more often, the small amount of processing during FreeBSD system timeout will raise the length of the system timer. There is also a low connection threshold where if the number of connections drops below, then only the system timer based approach is used. This helps avoid starving out connections by keeping them on the wheel too long. HPTS attempts to yield a precision of the timer minimum aka 250 microseconds, but this is not guaranteed.

A TCP stack using HPTS to pace has some distinct responsibilities in order to collaborate with HPTS to achieve its desired pace rate including:

- Once a pacing timer has been started, the stack must not allow a send or other call to `tcp_output()` to perform any output until the pacing timer expires. The stack can look at the `TF2_HPTS_CALLS` flag in the `t_flags2` field. This flag is or'ed onto the `t_flags2` as HPTS calls the `tcp_output()` function and should be noted and cleared by the stack inside its `tcp_output()` function.
- At the expiration of a pacing timer, in the call from HPTS, the stack needs to verify the time that it has been idle. It is possible that HPTS will call the stack later than expected, and it is even possible that HPTS will call the stack early (though this is quite rare). The amount of time that the stack is late or early needs to be included in the TCP stack's next pacing timeout calculation after it has sent data.
- If the stack decides to use the FreeBSD timer system, it must also prevent timer callouts from sending data. The RACK and BBR stacks do not use the FreeBSD timer system for timeouts, and, instead, just use HPTS as well.
- If the stack is queuing packets from LRO, then HPTS may call the input function instead of `tcp_output()`. If this occurs, no other call to `tcp_output()` will be made, since it is assumed that the stack will call its output function if it is needed.

There are also a number of utilities that HTPS offers to assist a TCP stack including:

- `tcp_in_hpts()` tells the stack if it is in the HPTS system.
- `tcp_set_hpts()` sets up the CPU a connection will use, and is optional to call, since the HPTS will do this for the connection if the stack does not call this function.
- `tcp_tv_to_hptstick()` converts a `struct timeval` into the number of HPTS slots the time is.
- `tcp_tv_to_usectick()` converts a `struct timeval` into a 32-bit unsigned integer.
- `tcp_tv_to_lusectick()` converts a `struct timeval` into a 64-bit unsigned integer.
- `tcp_tv_to_msectick()` converts a `struct timeval` into a 32-bit unsigned millisecond tick.
- `get_hpts_min_sleep_time()` returns the minimum sleep time that HPTS is enforcing.
- `tcp_gethptstick()` optionally fills in a `struct timeval` and returns the current monolithic time as a 32-bit unsigned integer.
- `tcp_get_u64_usecs()` optionally fills in a `struct timeval` and returns the current monolithic time as a 64-bit unsigned integer.

`sysctl`-Variables

The HPTS system can be configured using `sysctl`-variables to change its performance characteristics. These values come defaulted to a set of "reasonable" values, but depending on the application, they might need to be changed. The values are settable under the **`net.inet.tcp.hpts`** system control node.

The following tunables are available:

Name	Default	Description
`no_wake_over_thresh`	1	When inserting a connection into HPTS, if this boolean value is true and the number of connections is larger than **`cnt_thresh`**, do not allow scheduling of a HPTS run. If the value is 0 (false), then when inserting a connection into HPTS, it may cause the HPTS system to run connections aka call **`tcp_output()`** for connections due to be scheduled.
`less_sleep`	1000	When HPTS finishes running, it knows how many slots it ran over. If the number of slots is over this value then the dynamic timer needs to be decreased.
`more_sleep`	100	When HPTS finishes running, if the number of slots run is less than this value then the dynamic sleep is increased.
`min_sleep`	250	This is the absolute minimum value that the HPTS sleep timer will lower to. Decreasing this will cause HPTS to run more using more CPU. Increasing it will cause HPTS to run less using less CPU, but it will affect precision negatively.
`max_sleep`	51200	This is the maximum sleep value (in HPTS ticks) that the timer can reach. It is typically used only when no connections are being serviced i.e., HPTS will wake up every 51200 x 10 microseconds (approximately half a second).
`loop_max`	10	This value represents how many times HPTS will loop when trying to service all its connections needing service. When HPTS starts, it pulls together a list of connections to be serviced and then starts to call **`tcp_output()`** on each connection. If it takes too long to do this, then it's possible more connections need service, so it will loop back around to again service connections. This value represents the maximum HPTS will do that loop, before being forced to sleep. Note that being called on return from a function call never causes any looping to occur; only the FreeBSD timer call is affected by this parameter.
`dyn_maxsleep`	5000	This is the maximum value that the dynamic timer can be raised to when adjusting the callout time upwards is being performed and seeing the need for **`more_sleep`**.
`dyn_minsleep`	250	This is the minimum value that the dynamic timer can lower the timeout to when adjusting the callout time down after seeing **`less_sleep`**.
`cnt_thresh`	100	This is the number of connections on the wheel that are needed to start relying more on system call returns. Above this threshold, both system call return and timeouts cause HPTS to run, below this threshold, we rely more heavily on the callout system to run HPTS.

Optimizations for Pacing in the RACK Stack

When pacing using the HPTS system, there is some performance loss as compared to a pacing system that runs below a TCP stack. This is because when you call `tcp_output()` a lot of decisions are made as to what to send. These decisions usually reference many cache lines and cover a lot of code. For example, the default TCP stack has over 1500 lines of code in the `tcp_output()` path and it includes no code to deal with pacing or burst mitigation. For the default stack without pacing, going through such a large number of lines of code and lots of cache misses is compensated easily by the fact that it might output several dozen segments in one send. Now, when you implement a pacing system that is below the TCP stack, it can readily optimize the sending of the various packets it has to do by keeping track of what and how much it needs to send next. This makes a lower layer pacing system have many fewer cache misses.

In order to obtain similar performance with a higher layer system like HPTS, it becomes up to the TCP stack to find ways to optimize the sending paths (both transmissions and retransmissions). A stack can do this by creating a "fast path" sending track. The RACK stack has implemented these fast paths so that pacing does not cost quite so much. The BBR stack currently does pace, as required by the BBRv1 specification that was implemented, but it does not (as yet) have the fast paths described below.

Retransmissions in RACK also have a fast path. This is made possible by RACK's sendmap.

Fast Path Transmissions

When RACK is pacing the first time, a send call falls through its `tcp_output()` path and it will derive the number of bytes that can be sent. This is then lowered to conform to the size of the pacing microburst that has been established, but during that reduction, a "fast send block" is set up with the amount that is left to send and pointer to where in the socket send buffer that data is. A flag is also set so that RACK will remember next time that the fast path is active. Note that if a timeout occurs, the fast path flag is cleared so that proper decisions will be made as to which retransmission needs to be sent.

At the entry to RACK's `tcp_output()` routine, the fast path flag is checked after validating that, pacing wise, it is ok to send. If the flag is set, it proceeds to use the previously saved information to send new data without all of the typical checks that the output path would normally do. This brings the cost of pacing down considerably, since much of the code and cache misses are eliminated from this fast output path.

Fast Path Retransmissions

Retransmissions in RACK also have a fast path. This is made possible by RACK's sendmap which tracks all data that has been sent. When a piece of data needs retransmission, the sendmap entry tells the fast path precisely where and how much data needs to be sent. This bypasses typical socket buffer hunting and other overhead and provides a level of efficiency even when sending retransmissions.

Conclusion

HPTS provides a novel service TCP stacks can make use of to implement pacing. In order to achieve efficiencies more equivalent to competing design approaches, both the TCP stack and the HPTS need to cooperate to minimize overhead and provide for efficient sending of packet bursts. This column only discusses the need for pacing and the infrastructure provided to do so in FreeBSD. Future columns will look at another key question when a TCP stack paces, i.e., what rate to pace at.

Reference

1. Mohit Aron, Peter Druschel: *Soft Timers: Efficient Microsecond Software Timer Support for Network Processing*. In: ACM Transactions on Computer Systems, Vol. 18, No. 3, August 2000, pp 197-228. https://dl.acm.org/doi/pdf/10.1145/319344.319167.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.