Static Pacing in FreeBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

Previous articles described FreeBSD's underlying support for pacing via its High Precision Timing System (HPTS) as well as a software pacing methodology within the RACK stack called Dynamic Goodput Pacing (DGP) which will dynamically pace at an optimal rate for the current network condition. There is one more pacing methodology that is provided by the RACK stack that has yet not been described: static pacing..

Static pacing is much simpler and was one of the first pacing methods added to the RACK stack mainly to test out and improve the pacing facilities (i.e. HPTS). Even though it was initially intended as a testing methodology it can, in some instances, be used by an application in specific situations. When using static pacing, the pacing rate is not computed by the RACK stack. Instead, the application must provide the pacing rate by using socket options. Therefore, code in the application is needed to make use of static pacing.

When using static pacing,the pacing rate is not computed by the RACK stack.

Sending of TCP Segments

Three events trigger the sending of TCP segments:

1. The application provides user data to be sent to the TCP stack via `send()` system calls.
2. A TCP timer (for example, the retransmission or delayed acknowledgment timer) runs off.
3. A TCP segment is received.

The number of segments, which can be sent in such an event is mainly controlled by two mechanisms:

1. The flow control protecting a slow receiver against a fast sender. The receiver drives this by advertising the number of bytes the sender is allowed to send in the `SEG.WND` field in the TCP segments sent by the receiver.
2. The congestion control protecting a slow network against a fast sender. The sender drives this by computing the number of bytes, which are allowed to be sent. This number is called the congestion window `CWND`.

The sender only sends what is allowed by the flow control and by the congestion control.

There are various algorithms to perform congestion control. One congestion control, which was the default congestion control in FreeBSD for a long time, is New Reno. New Reno operates in one of two phases:

1. Slow start: this is the initial phase and the one after a timer-based retransmission. In this phase, the `CWND` grows exponentially.
2. Congestion avoidance: this is the phase in which the TCP connection should operate most of the time. In this phase, the `CWND` grows linearly.

At the beginning of a TCP connection, `CWND` is set to the initial congestion window which is controlled by the `sysctl`-variable `net.inet.tcp.initcwnd_segments`. The default is 10 TCP segments.

There are two methods used by a TCP endpoint to retransmit user data, which it considers to be lost. One is after the retransmission timer expires. In the other case, the TCP endpoint operates in recovery, because a loss of TCP segments has been detected.

There are various algorithms to perform congestion control.

The above description shows that a TCP endpoint might send bursts of TCP segments when the flow and congestion control allow this.

The following `packetdrill` script illustrates the behavior of FreeBSD when the application provides user data for 10 TCP segments right after the establishment of the TCP connection. `packetdrill` is a testing tool for TCP stacks and its scripts contain the system calls provided by the application and the TCP segments sent and received by the TCP stack. The lines start with a timestamp given in seconds. If the time information starts with a `+`, it is relative to the one before. So, `+0.100` means that the event happens 100 ms after the one before.

```
--ip_version=ipv4

 0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                      pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
+0.000 < S      0:0(0)                  win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)        ack     1 win 65535 <mss 1460,sackOK,eol,eol>
+0.050 <  .     1:1(0)        ack     1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 >  .     1:1461(1460)  ack     1 win 65535
```

```
+0.000 >  .  1461:2921(1460)  ack     1 win 65535
+0.000 >  .  2921:4381(1460)  ack     1 win 65535
+0.000 >  .  4381:5841(1460)  ack     1 win 65535
+0.000 >  .  5841:7301(1460)  ack     1 win 65535
+0.000 >  .  7301:8761(1460)  ack     1 win 65535
+0.000 >  .  8761:10221(1460) ack     1 win 65535
+0.000 >  . 10221:11681(1460) ack     1 win 65535
+0.000 >  . 11681:13141(1460) ack     1 win 65535
+0.000 > P. 13141:14601(1460) ack     1 win 65535
+0.050 <  .     1:1(0)        ack  2921 win 65535
+0.000 <  .     1:1(0)        ack  5841 win 65535
+0.000 <  .     1:1(0)        ack  8761 win 65535
+0.000 <  .     1:1(0)        ack 11681 win 65535
+0.000 <  .     1:1(0)        ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)    ack     1 win 65535
+0.050 < F.     1:1(0)        ack 14602 win 65535
+0.000 >  . 14602:14602(0)    ack     2
```

It is assumed that the round-trip time (RTT) is 50 ms. The script shows that the `send()`-call triggers a burst of 10 TCP segments to be sent at about the same time.

Static Pacing

Static pacing is a method for mitigating the burstiness of traffic. It replaces a large burst with a sequence of smaller bursts. The size of the smaller bursts is called the pacing burst size. The time between these smaller bursts is specified by giving a pacing rate.

Consider a sending rate of 12,000,000 bits/sec, which corresponds to 1,500,000 bytes/sec. When packets with a size of 1500 bytes are sent, this means sending one packet every millisecond. Or sending two packets every two milliseconds, and so on.

By specifying a pacing rate and the pacing burst size, the time between the pacing bursts can be computed such that the sending rate is the given pacing rate.

The size of the smaller bursts is called the pacing burst size.

For static pacing in the RACK stack, the application provides individual pacing rates used for slow start, congestion avoidance, and recovery and the pacing burst size. For the packet size, the size of the IP header, the size of the TCP header, and the TCP payload are considered. The size of link-layer headers and trailers is not considered.

Static pacing is controlled by using `IPPROTO_TCP`-level socket options in the source code of an application. Three of these socket options are used to control the pacing rate in different states of the congestion control algorithm. There is one specific socket option to set the pacing burst size and one to enable and disable the static pacing.

These socket options are shown in the following table.

Socket option name	Type of value	Description
`TCP_RACK_PACE_RATE_CA`	`uint64_t`	This option will set the static pacing rate in bytes per second when the congestion control algorithm is in congestion avoidance.
`TCP_RACK_PACE_RATE_SS`	`uint64_t`	This option will set the static pacing rate in bytes per second when the congestion control algorithm is in slow start.
`TCP_RACK_PACE_RATE_REC`	`uint64_t`	This option will set the static pacing rate in bytes per second when the congestion control algorithm has entered recovery and is working to recover lost packets.
`TCP_RACK_PACE_ALWAYS`	`int`	This option enables or disables pacing.
`TCP_RACK_PACE_MAX_SEG`	`int`	Sets the pacing burst size.

There are four important notes to make:

1. Setting a socket option (as any other system call) can fail. The application must check for it. One reason that the setting of any of the above socket options fails is that the TCP stack currently used on the socket is not RACK. The default FreeBSD does not support static pacing. In addition to that, enabling static pacing can fail if the overall system limit regarding the number of TCP connections using pacing is reached. This limit is controlled by the **`sysctl`**-variable **`net.inet.tcp.pacing_limit`**.
2. When setting the first pacing rate, the rate is not only set for the mode specified by the socket option but for all three modes (congestion avoidance, slow start, recovery).
3. Setting the pacing rate is not enough to enable static pacing. Static pacing must explicitly be enabled by using the **`TCP_RACK_PACE_ALWAYS`** socket option.
4. When not setting the pacing burst size, the default value of 40 will be used.

The following **`packetdrill`** script illustrates the pacing of the RACK stack with a pacing rate of 12,000,000 bit/sec in slow start and a pacing burst size of 1.

```
--ip_version=ipv4

 0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                     pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
+0.000 < S      0:0(0)                  win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)        ack     1 win 65535 <mss 1460,sackOK,eol,eol>
```

```
+0.050 <  .     1:1(0)        ack     1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_RATE_SS, [1500000], 8) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_MAX_SEG, [1], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_ALWAYS, [1], 4) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 >  .     1:1461(1460)  ack     1 win 65535
+0.001 >  .  1461:2921(1460)  ack     1 win 65535
+0.001 >  .  2921:4381(1460)  ack     1 win 65535
+0.001 >  .  4381:5841(1460)  ack     1 win 65535
+0.001 >  .  5841:7301(1460)  ack     1 win 65535
+0.001 >  .  7301:8761(1460)  ack     1 win 65535
+0.001 >  .  8761:10221(1460) ack     1 win 65535
+0.001 >  . 10221:11681(1460) ack     1 win 65535
+0.001 >  . 11681:13141(1460) ack     1 win 65535
+0.001 > P. 13141:14601(1460) ack     1 win 65535
+0.042 <  .     1:1(0)        ack  2921 win 65535
+0.002 <  .     1:1(0)        ack  5841 win 65535
+0.002 <  .     1:1(0)        ack  8761 win 65535
+0.002 <  .     1:1(0)        ack 11681 win 65535
+0.002 <  .     1:1(0)        ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)    ack     1 win 65535
+0.050 < F.     1:1(0)        ack 14602 win 65535
+0.000 >  . 14602:14602(0)    ack     2
```

The required code changes to enable static pacing are highlighted in yellow and the resulting changes in behavior on the wire are in grey. It should be mentioned that only the timing of the TCP segments has changed, not the TCP segments themself. There is now a delay of one millisecond between the sending of the outgoing TCP segments containing the user data.

Using the same pacing rate in slow start but a pacing burst size of 4 is illustrated by the following **packetdrill** script:

```
--ip_version=ipv4

 0.000 `kldload -n tcp_rack`
+0.000 `kldload -n cc_newreno`
+0.000 `sysctl kern.timecounter.alloweddeviation=0`

+0.000 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0.000 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_FUNCTION_BLK, {function_set_name="rack",
                                                     pcbcnt=0}, 36) = 0
+0.000 setsockopt(3, IPPROTO_TCP, TCP_CONGESTION, "newreno", 8) = 0
+0.000 bind(3, ..., ...) = 0
+0.000 listen(3, 1) = 0
```

```
+0.000 < S      0:0(0)                  win 65535 <mss 1460,sackOK,eol,eol>
+0.000 > S.     0:0(0)        ack     1 win 65535 <mss 1460,sackOK,eol,eol>
+0.050 <  .     1:1(0)        ack     1 win 65535
+0.000 accept(3, ..., ...) = 4
+0.000 close(3) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_RATE_SS, [1500000], 8) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_MAX_SEG, [4], 4) = 0
+0.000 setsockopt(4, IPPROTO_TCP, TCP_RACK_PACE_ALWAYS, [1], 4) = 0
+0.100 send(4, ..., 14600, 0) = 14600
+0.000 >  .     1:1461(1460)  ack     1 win 65535
+0.000 >  .  1461:2921(1460)  ack     1 win 65535
+0.000 >  .  2921:4381(1460)  ack     1 win 65535
+0.000 >  .  4381:5841(1460)  ack     1 win 65535
+0.004 >  .  5841:7301(1460)  ack     1 win 65535
+0.000 >  .  7301:8761(1460)  ack     1 win 65535
+0.000 >  .  8761:10221(1460) ack     1 win 65535
+0.000 >  . 10221:11681(1460) ack     1 win 65535
+0.004 >  . 11681:13141(1460) ack     1 win 65535
+0.000 > P. 13141:14601(1460) ack     1 win 65535
+0.042 <  .     1:1(0)        ack  2921 win 65535
+0.000 <  .     1:1(0)        ack  5841 win 65535
+0.004 <  .     1:1(0)        ack  8761 win 65535
+0.000 <  .     1:1(0)        ack 11681 win 65535
+0.004 <  .     1:1(0)        ack 14601 win 65535
+0.000 close(4) = 0
+0.000 > F. 14601:14601(0)    ack     1 win 65535
+0.050 < F.     1:1(0)        ack 14602 win 65535
+0.000 >  . 14602:14602(0)    ack     2
```

As expected, four TCP segments are now sent every four milliseconds. Therefore, the timing of the TCP segments highlighted in grey is affected.

Additional Considerations

The last example would thus put in the 12Mbps pacing rate for all congestion control states with a burst size restriction of four segments and then finally turn on pacing. This means that the RACK stack would send four segments, wait for four milliseconds, and then send four more packets and continue repeating that sequence until all packets were sent.

It is important to note that the RACK stack will not wait at a send opportunity for a full four segments, so if there were less than four segments available in the socket buffer, the stack would send out what it could at that time and start an adjusted pacing timer that would space that burst at precisely the pace rate that was requested. Another important consideration is congestion control and flow control; pacing can be slower than the set rate if the stack hits the congestion control or flow control limit. So, the RACK stack may send only one, two, or three TCP segments not due to the lack of user data, but due to the congestion control or flow control restricting how much can be sent. Static pacing may yield reduced throughput due to the inability of it to “make up” for lost bandwidth beyond its set pacing rate after the competing traffic ceases.

There are at least four other interactions to be considered by a developer using static pacing. One is the interaction with Proportional Rate Reduction (PRR). PRR is invoked by the RACK stack whenever the stack is in recovery. In effect, it limits the sending to roughly one segment for every other inbound acknowledgment. This means that the data is sent both subject to a pacing time and subject to whether acknowledgments from the peer are received. In most cases with static pacing, this interaction between static pacing and PRR is not wanted. Therefore, there is an `IPPROTO_TCP`-level socket option `TCP_NO_PRR` to disable PRR. The socket option value is of type `int`.

Yet another recovery action that the RACK stack takes is rapid recovery; this is a feature that allows the RACK stack to recover packets faster based on loss and transmission time and not just three duplicate acknowledgments. However, this may change, during recovery, how fast data is sent and thus modify your expected pacing behavior. The `IPPROTO_TCP`-level socket option `TCP_RACK_RR_CONF` can be used to modify this behavior. Allowable values of type `int` for this socket option are 0, 1, 2, and 3. The default value is 0, which means that the RACK stack has full control to use its rapid recovery feature. The values provide subtle control over recovery by the caller. However, for static pacing, where only the rates provided should be used, the value should be set to 3.

Yet another recovery action that the RACK stack takes is rapid recovery.

Setting the pacing burst size has a CPU impact when TCP segment offloading (TSO) is enabled. Using smaller pacing burst sizes increases the CPU load. For example, sending 40 TCP segments with a pacing burst size of 40 will only require one TSO operation, whereas a pacing burst size of 4 will require 10 TSO operations.

Another thing to keep in mind when setting the pacing burst size is the delay of acknowledgments. Most TCP stacks enable this feature. This means that a TCP stack waits for two (or more) segments or a timeout of the delayed acknowledgment timer before sending out an acknowledgment. The timeout is normally set somewhere between 40 to 200 milliseconds (the specification of delayed acknowledgment calls for 200 but many operating systems have shortened it). So, if the pacing burst size is set to 1 instead of 4, in the above example, an interaction between the delayed ack timer and the pacing timer can happen resulting in pacing at a substantially lower rate than anticipated. Therefore, the pacing burst size should not be set below 2 to avoid such an interaction.

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.