RACK AND ALTERNATE TCP STACKS FOR FREEBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

In 2017 changes were made to the TCP stack in FreeBSD, allowing the coexistence of multiple TCP stacks. This way, the existing TCP stack could be left untouched and allow innovation at the cost of a limited number of function calls. Some functionality is still shared between all TCP stacks: the implementation of the SYN-Cache including the handling of SYN-Cookies and the initial steps of the handling of incoming TCP segments like checksum verification and looking up the TCP endpoint based on the port numbers and IP addresses. At any given time, a TCP connection is handled by exactly one TCP stack, but this TCP stack can be changed during the lifetime of the TCP connection.

This is where the TCP RACK stack began as a complete rewrite of the original TCP stack from the call to the `tcp_do_segment()` function and many other modularized sub-functions. The initial goal was to add support for a loss detection method called Recent Acknowledgement (RACK). RACK was described in an Internet draft, which became RFC 8985 in 2021. This is where the name of this TCP stack—RACK—comes from. But the TCP RACK stack has grown far beyond just the addition of support for RFC 8985. Part of the rewrite includes a completely different way of handling selective acknowledgement (SACK) information. In the TCP RACK stack, a complete map of all user data sent is maintained that allows an improved handling of retransmissions of user data as well as the addition of the RACK loss detection described in RFC 8985. Many additional features have grown out of this rewrite and are described in this article.

The RACK stack is available in both FreeBSD CURRENT and FreeBSD 14.0.

How to Use the TCP RACK Stack

The RACK stack is available in both FreeBSD CURRENT and FreeBSD 14.0. How to make it available depends on the FreeBSD version.

For FreeBSD 14.0, one needs to add the following two lines to the kernel configuration file

```
option TCPHPTS
makeoptions WITH_EXTRA_TCP_STACKS=1
```

and rebuild the kernel. The first line results in compiling the TCP high precision timer system (HPTS) into the kernel. The second line results in generating a kernel loadable module for the TCP RACK stack (`tcp_rack.ko`). To use the TCP RACK stack, the kernel module must be loaded. This can be done on every reboot by adding the line

```
tcp_rack_load="YES"
```

to the file `/boot/loader.conf`.

In FreeBSD CURRENT, both TCP RACK and HPTS are built as kernel modules by default. Since **tcphpts.ko** is loaded automatically as a dependency of **tcp_rack.ko**, only the latter must be loaded using **kldload**. To load the TCP RACK stack on every reboot, the following two lines need to be added to the file **/boot/loader.conf**:

```
tcphpts_load="YES"
tcp_rack_load="YES"
```

Compiling the TCP RACK stack statically into the kernel of FreeBSD CURRENT is also possible by adding the following two lines to the kernel configuration file

```
option TCPHPTS
option TCP_RACK
```

and rebuilding the kernel.

Note that TCP Blackbox Logging (**option TCP_BLACKBOX**) is now built by default in FreeBSD 14.0 and higher and also in FreeBSD CURRENT for all 64-bit platforms, since it is the standard way that TCP transport developers are both instrumenting as well as debugging the various TCP stacks.

The above describes how to make the TCP RACK stack available on a FreeBSD system. A list of all available TCP stacks is shown by running

```
sysctl net.inet.tcp.functions_available
```

in a shell.

In the upcoming versions—FreeBSD 14.1 and higher—the usage of the TCP RACK stack will be the same as the one described above for FreeBSD CURRENT.

There are different ways of actually using the TCP RACK stack, some involving source code changes of applications, some only involving configuration changes.

The **sysctl**-variable **net.inet.tcp.functions_default** is used to specify the default TCP stack that is used for new TCP endpoints created using the **socket(2)** system call. Executing

```
sysctl net.inet.tcp.functions_default=rack
```

sets the default stack to the TCP RACK stack. By adding the line

```
net.inet.tcp.functions_default=rack
```

to the file **/etc/sysctl.conf** the TCP RACK stack will be the default TCP stack after rebooting the system. When a TCP endpoint is created via a listener, the TCP stack is either inherited from the listener or based on the default TCP stack depending on the **sysctl**-variable **net.inet.tcp.functions_inherit_listen_socket_stack** being non-zero or zero. The default value of this variable is one.

It is also possible to change the TCP stack of individual TCP connections by using the **tcpsso(8)** command line tool as described in the man-page of the tool.

If the source code can be changed, the **IPPROTO_TCP**-level socket option with the name **TCP_FUNCTION_BLK** can be used to switch the TCP stack being used for the socket to the TCP RACK stack. The option value has the type **struct tcp_function_set**. For example, the following code performs this:

```
struct tcp_function_set tfs;

strncpy(tfs.function_set_name, "rack", TCP_FUNCTION_NAME_LEN_MAX);
tfs.pcbcnt = 0;
setsockopt(fd, IPPROTO_TCP, TCP_FUNCTION_BLK, &tfs, sizeof(tfs));
```

Using the TCP RACK stack allows the use of a number of features that the default TCP stack does not currently support. A lot of these features can be controlled via **`IPPROTO_TCP`**-level socket options or **`sysctl`**-variables under **`net.inet.tcp.rack`**.

Features of the TCP RACK Stack

The following sections describe the most important features provided by the TCP RACK stack.

RACK/TLP

Recent Acknowledgement (RACK) and Tail Loss Probe (TLP) are two integrated features within the TCP RACK stack. Recent acknowledgement changes the way that packet loss is detected and retransmissions are triggered. The loss detection implemented in the FreeBSD base stack and specified in RFC 5681 takes three duplicate acknowledgments or acknowledgement arrivals with SACK to get the TCP stack to send out a retransmission. In some cases, where, for example, less than four packets have been sent, this will cause the TCP stack to send the retransmission only after a retransmission timeout has occurred. RACK changes this so that when a SACK arrives, if enough time has elapsed since the sending of the lost packets, a retransmission happens immediately. If not enough time has occurred (usually the time is a bit larger than the current RTT), then a small RACK timer is started, and when this expires the retransmission is sent. This then will fix many—but not all—of the cases where a retransmission timeout would have to force out data. The last case is solved by the TLP. This is where whenever the TCP RACK stack has sent data, it starts a TLP timer instead of a retransmission timer. If the TLP timer expires, the TCP RACK stack sends either a new segment or the last segment sent. The hope of this TLP-sent segment is that the sender would either receive an acknowledgement back indicating all data has been received (a case where the last acknowledgement was lost) or the TLP would elicit a SACK, which would allow the normal fast recovery mechanisms to take over without hitting a retransmission timeout and thus collapsing the congestion window to 1 MSS.

A user of the TCP RACK stack, just by enabling the stack, gets the benefits of both RACK and TLP automatically. No socket option or configuration is required by the upper layer.

Proportional Rate Reduction (PRR)

Proportional Rate Reduction (PRR) is another automatic built-in feature of the TCP RACK stack, specified in RFC 6937 and currently being updated by the IETF. PRR improves the way data is sent during fast recovery. When using the TCP congestion control as specified in RFC 5681, the congestion window is reduced in half on entering fast recovery. This then causes a stall in sending new data during fast recovery. Basically, the sender must wait for one-half of the outstanding data to be acknowledged, and then the sender can start sending new data (along with our retransmissions). This causes a "stall" in the data flow between the sender and receiver. PRR is designed to improve that, such that during fast recovery, a new data segment can be sent roughly every other acknowledgment. This then prevents

the data "stall" and keeps data continually moving thus keeping the RTT and other transport metrics active and updating.

RACK Rapid Recovery (RRR)

RACK Rapid Recovery (RRR) is an interesting feature that started as a bug. In the initial development, the TCP RACK stack inadvertently allowed a case where when a SACK arrived that declared more than a single segment missing and the RACK timer expired for all of the data, the TCP RACK stack would send one segment and start a RACK timer. When the RACK timer expired (which was set to the RACK minimum timeout value of 1 ms), the TCP RACK stack would send another one of the missing segments. This would repeat until all of the missing segments were sent. This effectively ignores PRR during the initial recovery with a cost of sending further PRR segments much later. So, for example, if RRR sent 3 segments, the first retransmission and two extras, it would take the arrival of roughly 6 more acknowledgements before PRR would send out a new segment.

When this bug was discovered and "fixed," the quality of experience (QoE) for the users degraded. This is because those early segment losses often hold up the delivery of quite a few segments of data. This led to adding this as a feature that can be turned off and also has programmability into the amount of time since the time in question effectively makes RRR paced at 12Mbps in its default setup. By default, this feature is on with the RRR recovery rate set for one segment every millisecond. This results in a rate of 12 Mbps assuming a maximum transmission unit (MTU) of 1500 bytes.

SACK Attack Detection

One of the downsides of keeping a complete map of what is being sent is that this map can grow quite large in some circumstances. The TCP RACK stack attempts at all times to collapse the map to as small as possible yet still keep track of all stages of outstanding data. There is, however, an introduced possibility that a malicious peer can be designed to attack the memory and CPU resources used by the TCP RACK stack for a TCP connection by constantly splitting the sendmap into smaller and smaller pieces so that TCP RACK stack uses large amounts of memory and spends excessive amounts of time searching through that memory. An example might be where an attacker sends SACKs for every other byte. This can pose a serious threat and can impact a machine in undesired ways.

The TCP RACK stack includes the optional compiled in feature called TCP_SAD_DETECTION. The SAD stands for SACK Attack Detection (SAD). One can enable it for the TCP RACK stack by adding the line

```
option TCP_SAD_DETECTION
```

to the kernel configuration file and rebuilding the kernel.

Once added, it is on by default. It montors for a malicious peer and if detected, it disables the processing of SACKs from the peer. This degrades that single peer's performance but does not prevent the connection from making progress. It, in effect, becomes a connection that responds as if no SACK was ever enabled. This penalizes loss recovery, but still allows the connection to continue.

Burst Mitigation

Built into the TCP RACK stack, and on without any user intervention, is burst mitigation. To mitigate bursts, the stack will only send out a set size (the max burst size) at a send

opportunity and start either a small timer (to send out more) or depend on the returning acknowledgement stream to prompt the sending of more data. This helps mitigate large bursts that can cause excessive loss.

Support for TCP Blackbox Logging (BBLog)

One of the interesting aspects of the TCP RACK stack is the extensive support of TCP Blackbox Logging for both debugging and just general statistical analysis and instrumentation. This makes it much easier to track down problems and to acquire analysis of connection behavior.

Large Receive Offload (LRO) Integration for Burst Mitigation

TCP Large Receive Offload (LRO) is a feature to reduce the CPU resources needed for a receiver by coalescing multiple received TCP segments into a single one before passing them into the TCP stack. Often this results in a loss of information about the individual received segments but reduces the CPU resources needed, since fewer TCP segments need to be processed by the TCP stack.

An interesting feature interaction is a set of changes that have been made to the LRO code for better support of pacing in the TCP RACK stack. When a TCP connection is doing burst mitigation, it tends to walk through the send path more often, sending smaller bursts. Due to this, changes were made to the LRO code allowing all of the timing data on packet arrival information to be carried through to the TCP RACK stack without loss. Basically, during packet processing, the LRO code looks up to see if the packet is associated with a connection that allows it to queue packets directly to the TCP RACK stack. If so, the packets are enqueued directly to the connection and, depending upon connection state, the connection may be woken up. In cases where the TCP RACK stack is doing burst mitigation or pacing, that wake up is deferred until a timer expires and something can be done with the inbound acknowledgments. These steps also bypass IP stack processing and thus provide an additional mild reduction of needed CPU resources.

A Host of Alternate Features

Many other features are available in the TCP RACK stack via various socket options and `sysctl`-variables. Currently, TCP RACK stack supports 58 socket options that enable various features including pacing, burst mitigation options and recovery response modifications. Besides the socket options, around 150 `sysctl`-variables exist to either make a socket option apply to all connections or to modify various TCP RACK stack default configurations. All of these features and configuration are available to help adjust the TCP RACK stack to better conform to your network conditions and requirements.

How Netflix Evolves the TCP RACK Stack

Netflix currently uses only the TCP RACK stack, the FreeBSD default stack is present, but not in use. The way Netflix uses the TCP RACK stack is a bit novel and worth noting. Netflix actually keeps several generations of the TCP RACK stack named for its release numbers. At all times, it keeps the "latest" TCP RACK stack with all of the leading-edge features under development by its transport group.

Periodically, when a release is cut, the latest TCP RACK stack under development is copied and supported based on a release number. This TCP stack is then evaluated based upon QoE and CPU performance in comparison to the previously released TCP stack which is the

default in use. When the newest TCP RACK stack is at least as good or better than the old TCP RACK stack, the default is switched to the newer TCP RACK stack in the next release. The old TCP RACK stack is maintained for several releases and eventually removed.

New features on TCP RACK stacks are also tested this way so that it can be determined if a feature adds value or not. Reducing network impact with no degradation of QoE for Netflix's users is one of the transport team's main goals, so that Netflix is both a better network citizen and at the same time providing a good overall QoE.

Conclusion and Outlook

The TCP RACK stack provides a strong alternative to the FreeBSD base stack. It adds more features and options that provide a richer set of alternatives for the application developer to better tailor the TCP experience for users.

The TCP RACK stack was extensively tested using the Netflix setup and workload. But it is important to also test it in other setups and workloads. Therefore, it would be great if users could test the TCP RACK stack on their hardware, using their setup, and under their workload. Please report any issues found during testing to net@freebsd.org or to the authors of this article. Depending on the feedback and further testing, the TCP RACK stack might become the default stack for FreeBSD in the future.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer for over 10 years. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He currently works at Netflix in its transport team, supporting the TCP stack while innovating to constantly improve user QoE.

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.