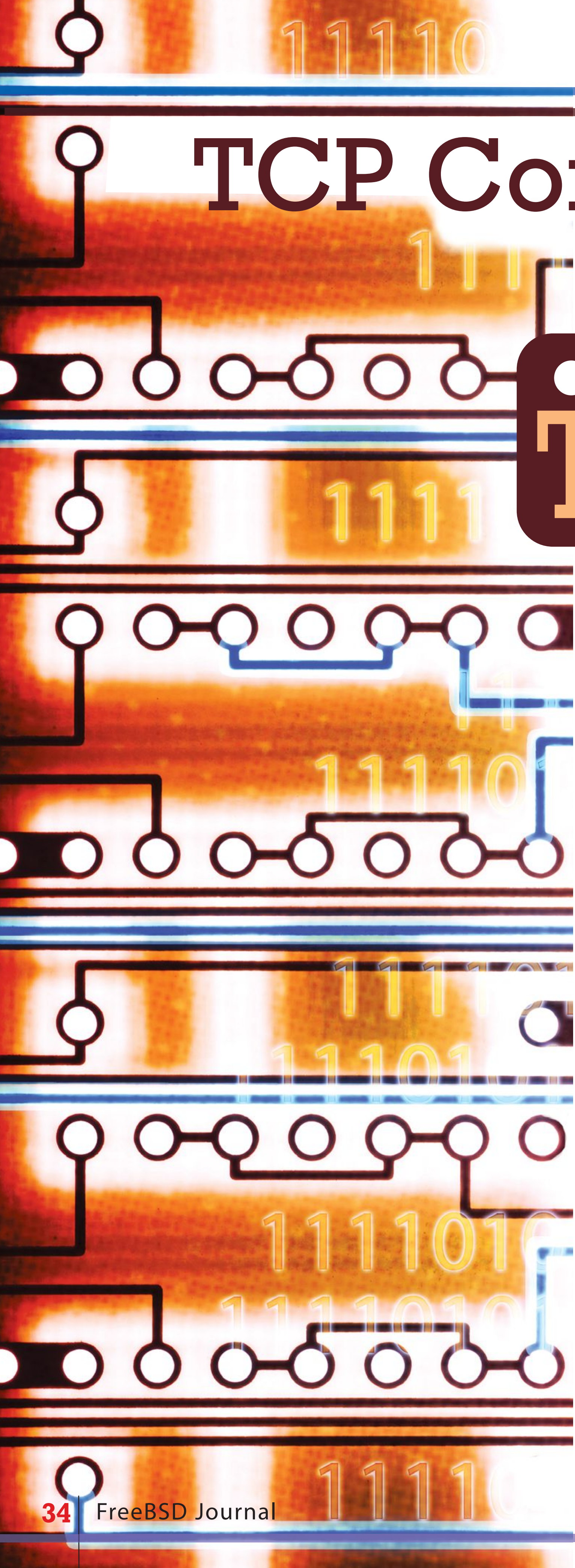

TCP Connection Rat

By Michael Bentkofsky and Julien Charbon

Today's commodity servers, with bandwidth of 10+ Gigabits per Network Interface Card (NIC) port and dozens of processor cores, have sufficient network and processing capacity to host the most demanding network services on a small server footprint.

With the popularity of web-based services, significant attention has gone into scaling these types of services to address issues such as the C10K problem (http://en.wikipedia.org/wiki/C10k_problem) of the previous decade that posed the challenge of handling 10,000 simultaneous connections on a single server.

Modern server software that handles tens of thousands of simultaneous connections is implemented using non-blocking I/O and event notification such as kqueue(). Today's new challenge, however, is aimed at servicing up to a million connections concurrently on a single server. Current generation NIC hardware can support this, but in order to keep scaling connection counts higher, one of the remaining challenges is to scale the TCP connection rate that can be serviced by a single server (http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/). This article considers several types of TCP-based services where the primary scaling problem is handling the highest rate of TCP connection establishment on modern server hardware. This is a different challenge from serving content to millions of established TCP connections. Examples of such services include:

- DNS services over TCP (http://tools.ietf.org/html/rfc5966),
- HTTP services with a single request and response, such as an Online Certificate Status Protocol (OCSP) service,
- Whois services for domain name registries.

e Scaling in FreeBSD

Of the examples above, DNS over TCP is the type of traffic we should expect to increase on the Internet as the adoption of DNSSEC causes larger DNS responses (https://www.dns-oarc.net/node/199) and as service providers use TCP to reduce attack vectors that are associated with connectionless protocols such as User Datagram Protocol (UDP). To delve into this scaling challenge, Figure 1 shows the packets associated with establishing a TCP connection, exchanging the request and response data, and tearing-down the TCP connection.

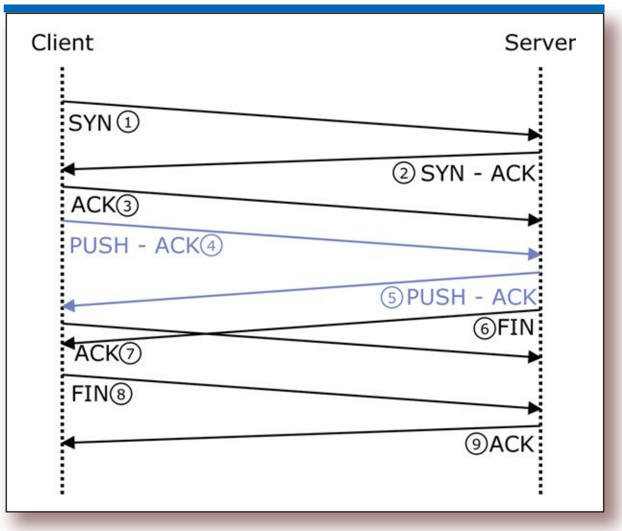

Figure 1. Typical packets, small TCP requests

In this use case, a client is initiating a connection SYN (1), the server responds with the SYN/ACK (2), and the client establishes the connection with an ACK (3). The request data packet from the client to the server is sent in (4) with a response data packet from the server in (5). This represents the exchange of a request to the server and a response sent back to the client. The server initiates the connection teardown (6) while the response is acknowledged (7). The client tears down its side of the connection (8) and the server finally acknowledges the client's FIN (9). Either end can initiate the connection teardown, so packets (6), (7), and (8) could be slightly differently sequenced. In this packet flow we can make several observations:

- There are a total of nine packets exchanged with five sent from the client to the server and four sent from the server to the client.
- There is a fair bit of overhead in establishment and teardown of connections with only two packets of data exchanged between the machines in (4) and (5). Although there can be more data exchanged on the same connection, this workflow considers small requests and responses.
- This kind of communication will typically have primarily small packets exchanged, depending on the size of the request and response. For this article, we'll consider generally small requests and responses, to analyze the worst-case scenarios to scale these types of exchanges.

This packet exchange is considered the canonical form of the small request and response problem for TCP. While the packet flow can be slightly different, such as the client and server both simultaneously initiating connection teardown, or the FIN being sent along with the data packet from either end, those differences do not significantly alter the nature of the scaling.

For this packet flow, we'll consider what it would take to scale to the NIC bandwidth capacity of currently available cards. For each packet, there will be at least 78 bytes:

- 12-byte inter-frame gap
- 8-byte preamble + the start of frame delimiter
- 18-byte Ethernet (source and destination MAC, type, checksum)
- 20-byte IP header (IPv4 without options)
- 20-byte TCP header at a minimum

If we assume that bandwidth will be limited primarily by egress (response) packets, for the canonical packet flow, we can calculate per request egress bandwidth consumption:

Packet Type	Octets	Notes
(2) SYN - ACK	98	78 + MSS, SACK, wscale, timestamps
(5) PSH - ACK	90+	78 + timestamps + response payload
(6) FIN	90	78 + timestamps
(9) ACK	90	78 + timestamps
Total Egress	368 + response payload	

```
$ top -SHIPz

CPU 0:   0.0% user,  0.0% nice,  0.4% system, 96.1% interrupt,  3.5% idle [irq core]
…
CPU 4:   2.4% user,  0.0% nice, 20.0% system,  0.0% interrupt, 77.6% idle [accept thread]
CPU 5:  11.8% user,  0.0% nice, 63.5% system,  0.0% interrupt, 24.7% idle [worker thread}
…

  PID  USERNAME   PRI  NICE    SIZE     RES  STATE   C    TIME      WCPU   COMMAND
   12  root       -92     -      0K    560K  CPU0    0    8:39    75.78%   [intr{irq263: ix0}]
 2636  ...         85    -5   4927M   4886M  CPU5    5    9:16    58.06%   tcp_server
 2636  ...        -21   r31   4927M   4886M  CPU4    4    3:00    20.26%   tcp_server
```

Figure 2. CPU Utilization, Single Queue

Additional payload, VLAN-tagging (802.1Q), slightly different packet counts, and additional TCP options could increase the cumulative request size, so if we assume aggregate inbound or outbound packet sizes between 500 and 1,000 bytes, a single gigabit NIC port should strive to serve between about 125,000 and 250,000 of these requests cumulative per second, representing between 625,000 and 1.25 million inbound packets per second and between 500,000 and 1 million outbound packets per second.

We measure how a multi-core server handles this workload without any overhead introduced in the application. The application is a simple TCP server running under FreeBSD 10.0, with a user space application listening on a socket in one thread, assigning file descriptors for new connections to multiple other threads, termed "worker threads." All threads are cpu_set to separate cores and there are not more user threads than total cores. The worker threads service multiple connections with kqueue(), read the request data with a single receive, generate the response with a single send, and close the connection.

For the client side, multiple requests are generated from a private network using a wide range of client source IP addresses and ports. While the client capacity is out-of-scope for this article, the client software has been benchmarked to create hundreds of thousands of connections per second.

At approximately 56,000 connections per second, or between 225 – 450 Mbits/sec of bandwidth, we see the CPU utilization shown in Figure 2. The bottleneck on core 0 is due to interrupt

```
CPU 0: 0.0%   user,  0.0%   nice,  30.5%   system, 64.5%   interrupt,   5.0%  idle  [irq core]
CPU 1: 0.0%   user,  0.0%   nice,  29.9%   system, 64.3%   interrupt,   5.7%  idle  [irq core]
CPU 2: 0.0%   user,  0.0%   nice,  30.2%   system, 66.1%   interrupt,   3.6%  idle  [irq core]
CPU 3: 0.0%   user,  0.0%   nice,  30.2%   system, 66.4%   interrupt,   3.4%  idle  [irq core]
CPU 4: 5.7%   user,  0.0%   nice,  84.4%   system,  0.0%   interrupt,   9.9%  idle  [accept() thread]
...
CPU 6: 8.3% user,    0.0%   nice,  59.1%   system,  0.0%   interrupt,  32.6%  idle  [worker thread]
CPU 7: 8.9% user,    0.0%   nice,  58.3%   system,  0.0%   interrupt,  32.8%  idle  [worker thread]
...

    PID  USERNAME   PRI  NICE    SIZE     RES    STATE  C     TIME     WCPU  COMMAND
   3110  ...        -21   r31   4955M   4911M    CPU4   4     5:12   70.26%  tcp_server
     12  root       -92     -      0K    688K    CPU1   1     7:20   66.46%  intr{irq264: ix0:que }
     12  root       -92     -      0K    688K    CPU2   2     7:21   66.26%  intr{irq265: ix0:que }
     12  root       -92     -      0K    688K    CPU3   3     7:21   66.16%  intr{irq266: ix0:que }
     12  root       -92     -      0K    688K    CPU0   0     7:20   65.87%  intr{irq263: ix0:que }
   3110  ...         52    -5   4955M   4911M    RUN    7     4:50   65.38%  tcp_server
   3110  ...         86    -5   4955M   4911M    CPU6   6     4:47   60.50%  tcp_server
      0  root       -92     0      0K    304K    RUN    1     0:50    8.06%  kernel{ix0 que}
      0  root       -92     0      0K    304K    RUN    0     0:49    8.06%  kernel{ix0 que}
      0  root       -92     0      0K    304K    RUN    3     0:48    8.06%  kernel{ix0 que}
      0  root       -92     0      0K    304K    RUN    2     0:49    7.96%  kernel{ix0 que}
```

Figure 3. CPU Utilization, Multiple Queues

```
# pmcstat -c 0 -S unhalted-cycles -O sample.pmc
# pmcstat -R sample.pmc -G call.graph

... [397427 samples]

64.82%       [257597]        __rw_wlock_hard @ /boot/kernel/kernel
99.01%       [255052]          tcp_input
100.0%       [255052]            ip_input
100.0%       [255052]              netisr_dispatch_src
00.91%       [2346]          in_pcblookup_hash
100.0%       [2346]            tcp_input
100.0%       [2346]              ip_input
00.07%       [180]           tcp_usr_attach
100.0%       [180]             sonewconn
100.0%       [180]               syncache_expand
00.01%       [19]            syncache_expand
100.0%       [19]                tcp_input
100.0%       [19]                  ip_input

02.48%       [9851]          __rw_rlock @ /boot/kernel/kernel
26.34%       [2595]            vlan_input
...
```

Figure 4. PMC Profile, Interrupt CPU

processing for packets to and from the NIC on CPU 0. Fortunately this bottleneck can be overcome by using the NIC Receive-Side Scaling (RSS) feature which distributes the traffic across multiple queues and dispatches interrupt handling to multiple CPUs via Message Signaled Interrupts (MSI or MSI-X).

With RSS, a NIC is configured to have multiple receive hardware queues, and MSI allows interrupts from the NIC to be directed to particular CPU cores that are lightly loaded. The NIC directs packets to receive queues based upon a hash function of packets, often a hash of the source and destination IPs and ports. In FreeBSD, this configuration is set partially in the device driver initialization, and then interrupt processing can be pinned to particular cores using cpuset. With a properly configured driver and smart selection of CPUs to handle interrupts, we should expect to see the processing workload shift more to the lightly loaded cores.

If all packet processing could be done in parallel, we would expect to see the capacity scale linearly with the number of receive queues and CPUs dedicated to receive processing, until the next bottleneck was reached. The results from such a test however are inconsistent with that, as shown in Figure 3, with 62,000 connections per second, or 250 to 500 Mbits/sec of bandwidth. In this configuration there are four NIC queues with affinity to four CPUs. An additional thread is added in the user space application to handle requests since Figure 2 suggested that CPU 5 was highly loaded. We note that interrupt processing has consumed all of the four dedicated CPUs and there are indications that both the threads accepting new connections and the two threads handling requests consume significant system time.

To analyze why distributing the TCP input processing workload on multiple MSI receiving queues did not scale as expected, we used Performance Monitoring Counter (PMC) profiling on one of the CPUs handling interrupt processing, as shown in Figure 4.

According to this profile, more than 50 percent of the CPU time of this core was spent in __rw_wlock_hard() and this kernel function was predominantly called from tcp_input().

__rw_wlock_hard() is part of the reader/writer kernel lock implementation [rwlock(9)], and more precisely this function aims for an exclusive access on the lock. Details about contended locks have been provided by running kernel lock profiling [LOCK_LOCK_PROFILING(9)] and sorting results by the total accumulated wait time for each lock (wait_total) in microseconds, shown in Figure 5.

The main contention point for this workload is on the rw:tcp lock. If we look at the top call points for this lock, we see in rank order:

1. sys/netinet/tcp_input.c:1013: tcp_input()
2. sys/netinet/tcp_input.c:778: tcp_input()
3. sys/netinet/tcp_usrreq.c:635: tcp_usr_accept()
4. sys/netinet/tcp_usrreq.c:984: tcp_usr_close()
5. sys/netinet/tcp_usrreq.c:728: tcp_usr_shutdown()

The first two locks in tcp_input() are called by the kernel on each TCP packet whereas the latter three calls correspond to socket system calls from the user-space TCP server. The per-packet lock is acquired in tcp_input() under these conditions:

```
# sysctl debug.lock.prof.stats | head -2; sysctl debug.lock.prof.stats | sort -n -k 4 -r | head -6

max  wait_max     total   wait_total     count   avg wait_avg cnt_hold cnt_lock name

210       660   2889429      7854725    568650     5       13      0   562770  …/tcp_input.c:1013 (rw:tcp)
 79       294   3346826      7309124    642543     5       11      0   541026  …/tcp_input.c:778 (rw:tcp)
  9       281    109754      4907003    321270     0       15      0   284203  …/tcp_usrreq.c:635 (rw:tcp)
  6       204    174398      1484774    321267     0        4      0   284754  …/tcp_usrreq.c:984 (rw:tcp)
 17       207   1252721      1195551    321268     3        3      0   241000  …/tcp_usrreq.c:728 (rw:tcp)
210       197   3828100       315757   1606362     2        0      0    90016  …/in_pcb.c:1802 (rw:tcpinp)
```

Figure 5. Lock Profiling the TCP Workload

- Any of the SYN, FIN, or RST TCP flags are set
- The TCP connection state is any state other than ESTABLISHED

Looking at the packets exchanged and corresponding TCP connection states (shown in the table below), we see that four of the five packets received cause the exclusive write lock rw:tcp to be acquired. As this lock is global to all TCP sockets, this contention appears to be a large factor in limited scaling through RSS.

#	Packet In	Packet Out	State, Before	State, After	rw:tcp locked in tcp_input()?
1	**SYN** (1)	SYN + ACK (2)	None	SYN-RECEIVED	**Yes, WLOCK**
2	ACK (3)		**SYN-RECEIVED**	ESTABLISHED	**Yes, WLOCK**
3	PSH (4)	PSH (5):	ESTABLISHED	ESTABLISHED	No
4		FIN (6)	ESTABLISHED	FIN-WAIT-1	
5	ACK (7)		**FIN-WAIT-1**	FIN-WAIT-2	**Yes, WLOCK**
6	**FIN** (8)	ACK (9)	**FIN-WAIT-2**	TIME-WAIT	**Yes, WLOCK**

In Figure 6, we see that the contention points on rw:tcp include tasks originated from interrupts driven from the NIC and packet processing, from user space system calls, and from other periodic timer tasks such as the TCP TIME-WAIT timer.

The rw:tcp lock protects the global data structures defined for the TCP state including:

The hash table (struct inpcbinfo.ipi_hashbase) to search among the structures (struct inpcb)

The global list to scan the structures (struct inpcbinfo.ipi_listhead)

Layer 4 specific additional structures including the list of sockets in the TIME-WAIT state for TCP.

The lock is defined as a readers-writer lock so multiple read tasks may be simultaneously searching, but only one task may be updating while also blocking all readers. When an inpcb structure is added or removed, the writers lock is held. In the context of TCP this occurs when a connection is established with ACK(3) that completes the connection and when the connection is entirely torn down and TIME-WAIT has completed.

In addition to the global rw:tcp lock, each inpcb has a lock named rw:tcpinp. This lock is held when per-connection information is updated. With long-running connections, each rw:tcpinp lock may be held and released several times, although this only causes per-connection contention, not across all connections. While rarely causing contention for short-lived connections, multiple locks necessitate a well-established lock order to avoid deadlock. This well-established order is rw:tcp (the global lock) must be locked before a rw:tcpinp (the per-connection lock) is locked when both locks are to be held. As previously noted, the TCP state transitions on four of the five received packets lead to both locks being held.

There are cases where the rw:tcp lock is held other than normal packet processing considered here and the user-space system calls. Other examples include inbound packets with the TCP RST flag set, conditions where resources can't be allocated or system configured limits are reached, and even unusual cases such as when the TCP congestion control algorithm is reconfigured. Careful attention must also be given to these other processing paths to avoid possible deadlock or global structure corruption whenever considering changes to the locking strategy to scale connection rate. While complex, there appear to be several opportunities to reduce contention on the global rw:tcp lock. Analyzing the paths where the lock is acquired shows sev-

Keep FreeBSD Free!
(Donate Today...)

Last year your donations helped FreeBSD by:

- Funding development projects to improve FreeBSD, including: Native iSCSI kernel stack, Updated Intel graphics chipset support, Integration of Newcons, UTF-8 console support, Superpages for ARM architecture, and Layer 2 networking modernization and performance improvements.
- Hiring two more staff members to help with FreeBSD development projects, security, and release engineering.
- Educating the public and promoting FreeBSD. Providing funding and leadership in publishing the FreeBSD Journal. We create high-quality brochures to teach people about FreeBSD. We also visit companies to help facilitate collaboration efforts with the Project.
- Sponsor BSD conferences and summits in Europe, Japan, Canada, and the US.
- Protecting FreeBSD intellectual property and providing legal support to the Project.
- Purchasing hardware to build and improve FreeBSD project infrastructure.

What will your donation help the Foundation accomplish in 2014?

Support FreeBSD by donating to the FreeBSD Foundation

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system. Our mission is to continue and increase our support and funding to keep FreeBSD at the forefront of operating system technology.

Thanks to people like you, the FreeBSD Foundation has been proudly supporting the FreeBSD Project and community for over 13 years now. We are incredibly grateful for all the support we receive from you and so many individuals and organizations that value FreeBSD.

Make a gift to support our work in 2014. Help us continue and increase our support of the FreeBSD Project and community worldwide!

Making a donation is quick and easy. Go to:
www.freebsdfoundation.org/donate

The FreeBSD FOUNDATION

www.freebsdfoundation.org

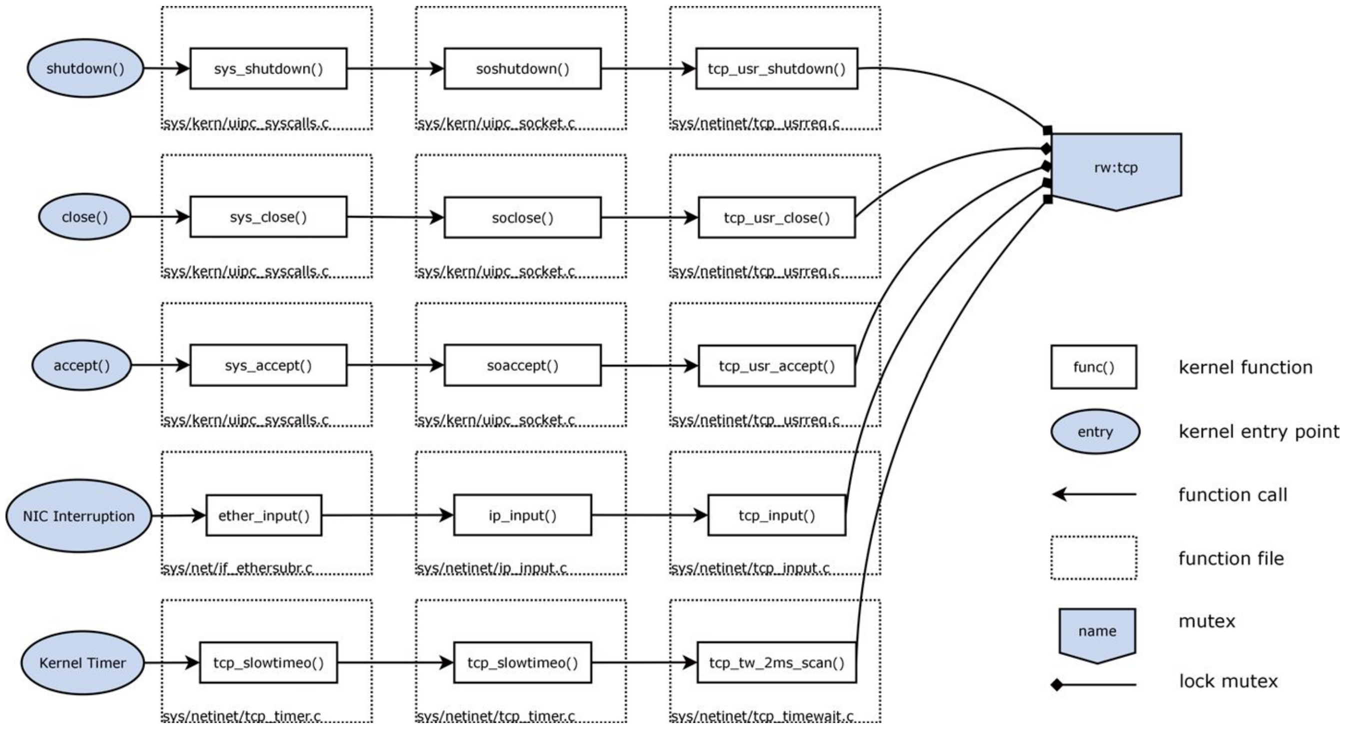

Figure 6. Lock Contention Points for rw:tcp

eral promising ways to:
1. Avoid the rw:tcp lock altogether, particularly driven from user-space system calls,
2. Add new finer-grained locks where the rw:tcp lock is currently used,
3. Switch to rw:tcp read locks to permit concurrency in critical code paths.

As an example of the first contention mitigation approaches, an implementation that avoids locking rw:tcp lock during the accept() system call has been adopted for future versions of FreeBSD (http://www.freebsd.org/cgi/query-pr.cgi?pr=183659). There appear to be other similar opportunities.

As an example of the second mitigation approach, an alternative implementation is also being adopted for expiring connections in the TIME-WAIT state (http://svnweb.freebsd.org/base?view=revision&revision=264321). Instead of locking the rw:tcp lock to manage the global TIME-WAIT list, a new lock rw:tcptw is created. While the rw:tcp lock is still used to finally destroy the inpcb structures, it is only held briefly and not while iterating the expiration list.

Early performance results using the two cited patches show that the techniques help scale connection rate. With accept() system call avoidance and a separate fine-grained lock for TIME-WAIT, we see the connection-rate increase from 62,000 connections per second to 69,000 connections per second. While modest, there appear to be further applications of these two techniques.

This work will continue to explore the possibility of allowing more parallelism in packet processing. Although in early development, the third technique looks promising. Early performance results suggest this technique could exceed 120,000 connections per second utilizing 480 – 960 Mbits / sec of bandwidth.

While high connection rates may be atypical for many networked services including streaming media or services with long-standing connections, such applications are present. The challenge of handling a high rate of connections that could utilize a single Gigabit NIC is within reach, although more work needs to be done to achieve that goal. ●

Julien Charbon is a Software Development Engineer at Verisign, Inc. Julien has worked on the company's high-scale network service ATLAS platform and related high-scale network services. Julien has worked with FreeBSD to perform tasks including porting software, developing fixes and patches, and network performance studies.

Michael Bentkofsky is a Principal Software Engineer at Verisign, Inc. and leads the development of their ATLAS platform. He has been part of teams implementing high-scale, always-available TCP services including DNS and Whois services for the .COM and .NET top-level domains, and the Online Certificate Status Protocol (OCSP) for certificate validation.