The Case of the Clumsy Kernel

BRENDAN GREGG

Brendan Gregg is the lead performance engineer at Joyent, where he analyzes performance and scalability at any level of the software stack. He is the author of the book *Systems Performance* (Prentice Hall, 2013) and is the recipient of the USENIX 2013 LISA Award for Outstanding Achievement in System Administration. He was previously a performance lead and kernel engineer at Sun Microsystems, where he developed the ZFS L2ARC, and later Oracle. He has invented and developed performance analysis tools, which are included in multiple operating systems, and has recently developed performance visualizations for illumos and Linux kernel analysis. brendan.gregg@joyent.com

All benchmarks are wrong until proven otherwise. Benchmarking is an amazingly error-prone activity, with results commonly misinterpreted and wrong conclusions drawn. However, every now and then, a benchmark takes me by surprise and not only is correct but also identifies a legitimate issue. This is a story of debugging a benchmark.

A Joyent customer had benchmarked Node.js connection rates and found a competitor had five times higher throughput than we did. Because we're supposed to be the "High Performance Cloud," as well as the stewards of Node.js, this was more than a little embarrassing. I was asked to troubleshoot and determine what was happening: were the results misleading, or was there a real performance issue? We hoped that the problem was something simple, like the benchmark system hitting a CPU limit.

Our support staff had already begun collecting a problem statement, which included checking which software versions were used. This process can solve some issues immediately, without any hands-on analysis. The benchmark was Apache Bench (ab) [1], measuring the rate of HTTP connections to a simple Node.js program from 100 simulated clients. This was about as simple as it gets.

Some factors can make these investigations very hard. The worst I deal with involve networking between multiple hardware-virtualized guests, which means trekking between guest and host kernels via a hypervisor, and where those kernels are entirely different (Linux and illumos [2]). In this case, it was on a single host via localhost, and in an OS-virtualized guest. These factors took the physical network components and lower-level network stack completely off the investigation table and left only one kernel to study (illumos). This should be easy, I thought.

The USE Method

I created a server instance and ran the same benchmark that the customer had. Benchmarks like ab run as fast as they can and are usually limited by some systemic bottleneck. The utilization, saturation, and errors (USE) method is a good way to identify these bottlenecks [3], and I performed it while the benchmark was executing. The USE method involves checking physical system resources: CPUs, memory, disks, network, as well as resource controls. I discovered that CPUs, which I expected to be the limiter for this test, were largely idle and also were not driven near the cloud-imposed limit. The USE method often gives me quick and early wins, but not in this case.

By this point, I had run ab a few times and noticed that its results matched what the customer had seen: the connection rate averaged around 1000 per second. Because I don't trust anything any benchmark software tells me, ever, I looked for a second way to verify the result and to get more information about it. Running on SmartOS [4], I used "netstat -s 1" to print per-second summaries, which also shows per-second variation (on Linux, I would have used "sar -n TCP 1").

```
$ netstat -s 1 | grep ActiveOpen
    tcpActiveOpens  =728004  tcpPassiveOpens  =726547
    tcpActiveOpens  =   4939  tcpPassiveOpens  =   4939
    tcpActiveOpens  =   5849  tcpPassiveOpens  =   5798
    tcpActiveOpens  =   1341  tcpPassiveOpens  =   1292
    tcpActiveOpens  =   1006  tcpPassiveOpens  =   1008
    tcpActiveOpens  =    872  tcpPassiveOpens  =    870
    tcpActiveOpens  =    932  tcpPassiveOpens  =    932
    tcpActiveOpens  =    879  tcpPassiveOpens  =    879
[…]
```

The first line of output is the summary since boot, followed by per-second summaries. What caught my eye was the change in connection rate: starting around 5000 per second, and then slowing down after two seconds. This was not only a great lead, it also rang a bell. I remembered that this type of benchmarking can hit a problem involving TCP TIME_WAIT. This state occurs when the SYN packet heralding a new connection is misidentified as belonging to an old connection and so is dropped by the kernel. My test for this issue is to see how many connections are stuck in TIME_WAIT, and whether the client's ephemeral port range is exhausted—which causes every new connection to clash with an old one. I used netstat and saw that there were only around 11,000 connections from a possible range of about 33,000. So much for that theory.

What else might be happening after two seconds? I drew a blank.

Thread State Analysis

My other go-to methodology is the thread state analysis (TSA) method [5], where thread run time is divided into states, and then you investigate the largest states. On Linux, I'd perform this using tools including pidstat. On SmartOS, I used prstat [6]. When ab was running at 5000 connections per second, this showed that a single thread in node (the runtime process for Node.js) was on-CPU 100% of the time. This was the kind of CPU limit I had expected to hit. When ab slowed down, prstat showed:

```
$ prstat -mLc 1
  PID USERNAME  USR SYS  TRP  TFL  DFL  LCK  SLP  LAT  VCX  ICX   SCL SIG  PROCESS/LWPID
63037 root       15 3.6  0.0  0.0  0.0  0.0  81   0.2  268  26    8K   0   node/1
12927 root      2.4 8.3  0.0  0.0  0.0  0.0  89   0.7   1K  42   16K   0   ab/1
[...]
```

The node thread was now spending 81% of its time in the sleep (SLP) state, meaning the thread is blocked waiting for some event to complete, typically I/O. Were this performed between two remote hosts on a network, I would guess that it was waiting for network packet latency. But this was a localhost test!

One way to investigate the sleep state is to trace system calls and their latency. I may find that the sleep time is during read(), or accept(), or recv(), and I can investigate each accordingly. On Linux, I'd use one of the (in-development) tracing tools, which include ktap, SystemTap, dtrace4linux, and perf, or, if I didn't mind the overhead, strace. Because this was SmartOS, I used DTrace and quickly found that the sleep time was in the portfs() system call. The user-level stacks that led to portfs() told me little: the threads were polling for events.

portfs() is part of the event ports implementation, which has a similar role to epoll on Linux: an efficient way to wait on multiple file descriptors. Being blocked on portfs() meant we were blocked on something else, but we didn't know what, and it would be a bit of work just to dig out the right file descriptors.

This was looking like a dead end. Imagine you have a thread blocked polling on portfs(), or on Linux, epoll_wait(). What do you investigate next?

Tracers can lead you to think in a thread-oriented manner. If the thread time between A and B is of interest, then you look for events that happened between A and B for that thread, and measure their relative times. But what if the thread does nothing between A and B, as was the case here? Time has been spent on something else in the kernel—something mysterious and likely involving other threads. There is no easy way to correlate this activity, let alone know what activity or threads to trace to in the first place.

Walking the Wakeups

There is a way, however, and it's one that I've been using more and more of late. My approach is to trace the kernel as it performs wakeups: where one thread wakes up another sleeping thread. This provides correlation and context: the stack trace of the waker.

I used cv_wakeup_slow.d [7], modified to trace node processes. This is a DTrace script I wrote earlier, which shows the stack trace of the threads that woke up a specified target (the cv is for conditional variable, which is how the sleep and wake up is implemented by the kernel). I ran it with a 10 ms threshold and caught:

```
# ./cv_wakeup_slow.d 10
[…]
 23  12326    sleepq_wakeone_chan:wakeup 63037 1 0 0 sched… 46 ms
           genunix`cv_signal+0xa0
           genunix`port_send_event+0x131
           genunix`pollwakeup+0x86
           sockfs`so_notify_newconn+0x81
           sockfs`so_newconn+0x159
           ip`tcp_newconn_notify+0x198
           ip`tcp_input_data+0x1b4a
           ip`squeue_drain+0x2fa
           ip`squeue_enter+0x28e
           ip`tcp_input_listener+0x1197
           ip`squeue_drain+0x2fa
           ip`squeue_enter+0x28e
           ip`ip_fanout_v4+0xc7c
           ip`ire_send_local_v4+0x1d1
           ip`conn_ip_output+0x190
           ip`tcp_send_data+0x59
           ip`tcp_timer+0x6b2
           ip`tcp_timer_handler+0x3e
           ip`squeue_drain+0x2fa
           ip`squeue_worker+0xc0
```

This stack trace shows a thread slept for 46 ms and was woken up by a TCP packet. Interpreting latency depends on application needs and expectations for the target. In this case, I was expecting the benchmark to stay on-CPU as much as possible, so any non-zero sleep time was worth investigating. My choice of a 10-ms threshold was intended to filter out noise from occasional systemic perturbations, such as interrupts preempting the benchmark. These perturbations should be fast (sub-millisecond), and unlikely to cause the 81% sleep time I saw earlier. But if a 10-ms threshold came up empty-handed, I'd reduce that to 1 ms, and, if need be, to 0 ms so I could see all events.

Looking down the stack shows tcp_timer() calling tcp_send_data(). Huh? I took a quick look at the tcp_timer() code, which largely handles TCP retransmission. Retransmits?

I checked the retransmission rate compared to the connection rate using "netstat -s 1" (on Linux, use "sar -n TCP -n ETCP 1"). When the connection rate from ab was high, the retransmit rate was zero. But, when retransmits began to occur, the connection rate slowed down. This correlation matched what I'd found with the wakeup tracing: the benchmark was getting blocked waiting on retransmits.

But…retransmits? Over localhost? How is this possible?

Retransmits can be a sign of a poor physical network, including bad wiring, cables not plugged in properly, an overloaded network, TCP incast, and other reasons. But this was localhost, where the kernel is passing packets to itself, with no networks (reliable or otherwise) involved. I mentioned this to a colleague, Robert, and we were amused by the mental image of a clumsy kernel, dropping packets as it passed them from one hand to the other.

We did remember some legitimate reasons why a kernel might drop packets (firewalls, out of memory, etc.), which could lead to retransmits. And there was always the possibility of bugs. It wouldn't be the first localhost bug I've seen, and I shuddered at the thought of finding another.

I noticed something else about the retransmit rates: they seemed to hit a ceiling of 100 per second. ab was simulating 100 clients, and the TCP retransmit interval was one second. This fit: each client could do at most one retransmit per second, because it would then spend an entire second blocked on the retransmit. As an experiment, I set the ab client count to 333, and, sure enough, the retransmits moved to a ceiling of 333 per second.

I used another DTrace script I had written earlier [8] to trace retransmit packets and show their TCP state. This script quickly tells me whether the retransmitted packets were from an established connection, or from a different stage of a TCP session. Such kernel state information is not visible on the wire (or "wire" in quotes, as this is localhost), so it cannot be observed directly using packet sniffers. I hesitate to use packet sniffers for this kind of investigation anyway, because their overheads can change the performance of the issue I'm trying to debug.

```
# ./tcpretranssnoop_sdc6.d
TIME                  TCP_STATE        SRC        DST        PORT
2014 Jan  4 01:31:31  TCPS_SYN_SENT    127.0.0.1  127.0.0.1  3000
2014 Jan  4 01:31:31  TCPS_SYN_SENT    127.0.0.1  127.0.0.1  3000
2014 Jan  4 01:31:31  TCPS_SYN_SENT    127.0.0.1  127.0.0.1  3000
[…]
```

The output showed that the sessions were in the SYN_SENT state, so the packets were likely SYNs for establishing new connections. I've seen this before, when the TCP backlog queue is full due to saturation, and the kernel starts dropping SYN packets. This can be identified from "netstat -s" and the tcpListenDrop and tcpListenDropQ0 counters on SmartOS (on Linux, "SYNs to LISTEN sockets dropped" and "times the listen queue of a socket overflowed"). I was kicking myself for not checking these sooner—I should have suspected this problem for this type of benchmark.

However, these drop counters were zero. Another dead end.

Given a TCP-related issue, I looked at the remaining counters from the "netstat -s" output to study TCP more carefully, and I saw that the rate of tcpOutRsts was consistent with tcpRetransSegs. tcpOutRsts indicates TCP RST (reset) packets. Now I had a new factor to investigate: RSTs.

TCP Resets

I was curious to see packet-by-packet sequences, to see if there was a direct relationship between the RSTs and retransmits. This may also reveal other packet types that are involved. To do this, I could trace all packets or use a packet-capturing tool. I decided to try the latter to begin with, despite the higher overheads, because these tools typically do a good job of presenting packet and protocol details, which can help reveal patterns across multiple packets. I could do the same with a tracing tool, but in that case I'd need to code that presentation myself, which takes time. I tried snoop (on Linux, tcpdump) to check how the RSTs occurred, and I saw that they were happening in response to the SYNs. Why would we RST a SYN? The port was open. Was this TIME_WAIT?

Another DTrace script from my toolkit [9] showed whether packets were arriving during TIME_WAIT:

```
# ./tcptimewait.d
TIME                 TCP_STATE       SRC-IP    PORT  DST-IP    PORT FLAGS
2014 Jan  4 01:56:16 TCPS_TIME_WAIT  127.0.0.1 54170 127.0.0.1 3000  2
2014 Jan  4 01:56:16 TCPS_TIME_WAIT  127.0.0.1 50427 127.0.0.1 3000  2
2014 Jan  4 01:56:16 TCPS_TIME_WAIT  127.0.0.1 37854 127.0.0.1 3000  2
[...]
```

The output showed hundreds of packets per second. This was the TIME_WAIT issue I had thought of at the very beginning, although manifesting in a different way. Checking the ephemeral ports from the snoop output and revisiting rate counters from netstat, I could see that each of the 100 ab clients would average two successful connections per second and then block on the third. This left two connections per client in TIME_WAIT for the default of 60 seconds. So, for each second, there would be about 2 x 100 x 60 = 12,000 connections still around in TIME_WAIT, similar to the 11,000 connections I had seen earlier, which I had thought was too few to matter. Picking an ephemeral port from a range of about 33,000 when 11,000 were in use was also consistent with encountering one clash out of every three attempts. A final detail also fell into place: the "fast" rate of 5000 connections per second, seen in the earlier prstat output, lasted about two seconds. That was the time it took to reach approximately 11,000 connections in TIME_WAIT.

To find the kernel code that was causing this problem, I could follow the stack traces that led to the RSTs. I remembered that I could do this using the DTrace TCP provider I had developed while at Sun, although I couldn't remember my own syntax! A quick Internet search found my documentation, and I quickly had the stack trace responsible.

Unfortunately, the stack trace didn't look that special, with tcp_send_data() called by tcp_xmit_ctl(), which can happen for many different reasons. Fortunately, I found a gift from the kernel engineer who wrote tcp_xmit_ctl(): its first argument was a character pointer to a string explanation. Such strings are trivial to trace, and I found that it contained the text "TCPS_SYN_SENT-Bad_seq". This took me straight to the problem code, which was...familiar.

Too familiar. I started remembering more about the last time I had debugged this: we had laughed about how silly it seemed to have 60 seconds of TIME_WAIT for localhost connections and had said that that should be fixed. In fact, it had been fixed (thank you, Jerry!), but the customer had benchmarked on a system with an older illumos kernel. Linux has a different way to recycle sessions in TIME_WAIT and didn't suffer this issue in the first place. This was the reason that the competitor, on Linux, was always running five times faster, without any slowdown from retransmits.

The actual problem originates from the TCP specification: 16-bit port numbers and a lengthy TIME_WAIT. Sessions are identified by a four-tuple: client-IP:client-port:server-IP:server-port (or a three-tuple, if the server IP is not included). Because this benchmark only has one client IP, one server IP, and one server port, the only variable to uniquely identify connections is the 16-bit client ephemeral port (which by default is restricted to 32,768–65,535, so only 15-bits). After (only) thousands of connections, the chances of colliding with an old session in TIME_WAIT become great.

So the final verdict for the customer benchmark: there was a real performance issue, *and* the results were misleading. It was thought that our competitor was five times faster, but this wasn't the case for real production workloads. Node.js typically handles thousands of clients making new connections every second, not one client making thousands of new connections every second. The workaround for the benchmark was to use multiple real clients (not simulated ab clients), which brought the connection rate to around 5000 per second and steady, the same as our competitor, for this workload. The use of HTTP keep-alives was another workaround, as it avoided creating new connections altogether.

In the end, I had amazed myself: moving directly from thread time in the sleep state to TCP retransmits, by tracing which thread woke up our sleeping thread. What if I had stopped at pollsys() and not drilled down this far? That's what had happened last time I investigated: I had eventually run "netstat -s" and studied all counters, hoping for a clue, and found it. Having solved the same problem twice, using two very different approaches, gives me a rare opportunity to compare my own debugging techniques. I much prefer the direct approach that I used here—drilling down on latency and walking the wakeups.

References

[1] ab - Apache HTTP server benchmarking tool: http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] illumos: http://illumos.org.

[3] USE Method: http://www.brendangregg.com/usemethod.html.

[4] SmartOS: http://smartos.org.

[5] B. Gregg, *Systems Performance: Enterprise and the Cloud* (Prentice Hall, 2013).

[6] prstat: http://illumos.org/man/1m/prstat.

[7] cv_wakeup_slow.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/system/cv_wakeup_slow.d.

[8] tcpretranssnoop_sdc6.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/net/tcpretranssnoop_sdc6.d.

[9] tcptimewait.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/net/tcptimewait.d.