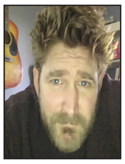
# iVoyeur
## eBPF Tools: What's in a Name?

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Fastly. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

There is a story in ancient Egyptian folklore that the goddess Isis created a serpent to poison the sun god Ra. Isis withheld the antidote from the withering sun god in exchange for his true name, which he eventually surrendered. This—the true name of Ra—gave Isis complete power over him and enabled her to elevate her son Horus to the Egyptian throne.

Horus died with Ancient Egypt, but Isis lived on into Greek mythology, along with many of her Egyptian counterparts. In fact the Romans were still building temples to her ~1000 years later. I find this early story about her gleaning Ra's true name kind of fascinating because she also happens to be one of the very few gods who was never renamed in the whole of human history. Ra, of course, became Apollo, who in turn became Phoebus Apollo to the Romans.

So all the while the Egyptian gods were being given Greek names, the Sumerian gods were being given Akkadian equivalents, and throughout the infamous Roman divinity-rebranding pivot from Greek mythos, Isis remained Isis. It's almost as if her nearly prehistoric cognizance of the power inherent in names somehow rendered her immune from the incessant attempts of mortals to relabel the divine.

Today, our god situation is comparatively simple (in cardinality at least), but our complicated relationship with names lives on. There is, for example, a Sunni Hadith (https://sunnah.com/bukhari/80/105) that asserts God has 99 names, and to know them is the path to paradise. The power of the "true name of God" is a central theme in Kabbalism, Sufism, Judaism, and in Christianity where we're reminded not to use it in vain, and where we find Jacob wrestling with an angel who refuses to reveal his true name.

Richard Feynman famously doubted the significance of names when he wrote about the difference between naming a thing and knowing it (https://fs.blog/2015/01/richard-feynman-knowing-something/). "See that bird?" he said. "It's a brown-throated thrush, but in Germany it's called a halzenfugel, and in Chinese they call it a chung ling and even if you know all those names for it, you still know nothing about the bird. You only know something about people; what they call the bird."

If naming something corporeal like a bird provides us no useful insight, what then are we to make of our propensity for foisting names upon the divine and ethereal? This is a question Socrates ponders in Cratylus; are names arbitrary labels? Or might they carry within them some innate, visceral power beyond our ability to comprehend? Are names the random vocalizations of apes or priceless gifts from some immortal creator?

There's a joke in our industry that goes: "There are two hard problems in computer science: cache invalidation, naming things, and off-by-one errors," and I personally would reorder that list such that naming things came first. There is, you know, a positively terrifying undercurrent to the act of giving something a name. A nagging suspicion that what I'm doing is not naming a thing at all but, rather, foisting upon future generations of engineers the banal and loathsome historical context of the present.

```
Device:  rrqm/s   wrqm/s     r/s       w/s     rkB/s      wkB/s  avgrq-sz  avgqu-sz  await   r_await  w_await  svctm  %util
sda        0.00    10.00    0.00     22.00      0.00     134.40     12.22      0.00   0.00      0.00     0.00   0.00   0.00
sdb        0.00     0.00    0.00      0.00      0.00       0.00      0.00      0.00   0.00      0.00     0.00   0.00   0.00
sdd        0.00  4275.40   13.60   7349.00     54.40   47689.60     12.97     21.10   2.87      0.53     2.87   0.08  57.76
md0        0.00     0.00  279.60  63568.20   1118.40  259052.00      8.15      0.00   0.00      0.00     0.00   0.00   0.00
sdc        0.00  4266.40   22.80   7343.80     91.20   47625.60     12.95     27.58   3.70      0.07     3.71   0.08  60.88
sde        0.00  4190.40   36.20   5611.60    144.80   39660.80     14.10      4.78   0.85      0.15     0.85   0.07  38.72
sdf        0.00  4189.20   20.80   5612.80     83.20   39660.80     14.11      4.34   0.77      0.23     0.77   0.06  34.56
sdo        0.00  4261.60   27.00   7508.40    108.00   48224.00     12.83     28.31   3.76      0.33     3.77   0.08  58.64
```

Extended disk report, trimmed to a reasonable length

Consider sed, a shell tool that derives its name from a still-older tool, ed (https://en.wikipedia.org/wiki/Ed_(text_editor)), developed in August 1969 when memory was so dear a commodity that every computer program had two- and three-letter names. Or Kubernetes, a tool with so unwieldy a name that the community has resorted to numeronyms (https://en.wikipedia.org/wiki/Numeronym) to deal with it on a daily basis.

## eBPF

Despite the considerable buzz surrounding eBPF these days, it's completely understandable if you're not exactly sure at first blush just what the heck it actually *is*. For one, it carries an understated—some would even say misleading—name, which like many things named by engineers, has more to say about its origins than its identity. I say it "carries" its name, but really it drags its name behind it like an iron ship-anchor. A name that makes it impossible to introduce to newcomers without delving into the history of its origins.

Upon hearing that the acronym eBPF stands for "Extended Berkeley Packet Filter," you might come to the conclusion that it's a packet-filtering program, which is either mostly wrong or completely wrong, depending on what you expect to get out of a name. If you think a name should imply what a thing is, you're completely wrong. eBPF is not a packet-filtering program; it's a register-based Virtual Machine running inside the Linux Kernel.

If you think a name should imply what a thing is *good for*, then you're only mostly wrong. eBPF can, in fact, filter packets for you. But it can also do many, many other things for you that have nothing whatsoever to do with the network stack.

Just the other day, in fact, I used an eBPF program to identify a failing drive in an mdraid array by asking it for a histogram of block-I/O latency as a function of device. One drive in the array had actually already failed and had been replaced with a new drive. But having added the new drive and rebuilt the array, disk I/O was still noticeably slow.

This left me in the unenviable position of having an array of 12 disks, one (or some) of which were not performing as well as they should. I don't know about you, but when I've encountered problems like this in the past, I've turned to iostat.

## iostat -dx5

Sometimes I wonder how many hours of my life I've spent staring at the output from this little command, which shows an extended disk report similar to the one above every five seconds.

The input data for this report comes from /proc/diskstats and is documented in the kernel docs https://www.kernel.org/doc/Documentation/iostats.txt. If you skim it, you'll probably notice that the report format and other details depend on the kernel version you're running, which is annoying. If you put your engineer hat on and read in a bit deeper, you'll start to come across some weird details related to—of course—*naming*.

The avgqu-sz field, for example, is misleading in that it isn't really an average of the queue size, because it doesn't show how many operations are queued waiting for service. Rather it shows how many I/O ops were either in the queue waiting *or being serviced*. Similarly await is not an in-queue wait time but actually measures end-to-end latency. Oh, and the disk report's last column %util? It tells you how much of the time during the measurement interval the device was in use (many people would understandably interpret something called "%util" as a measure of whether a device is reaching its limit of throughput, but nope).

If you know these things (and more) about iostat, and you are practiced at staring at this output, and you have something of a baseline understanding of what a healthy I/O load looks like for your system, and you have fewer than 50 disks, iostat will probably get you where you need to be. It probably would have gotten me to the finish line with my latency problem eventually, but I'd been reading about eBPF on Brendan's blog (http://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html), and I found myself staring at iostat and wondering whether there was a BPF tools script that could show me a breakdown of how much latency each individual disk was experiencing. Check this out:

```
usecs              : count    distribution
    0 -> 1         : 0        |                                        |
    2 -> 3         : 0        |                                        |
    4 -> 7         : 0        |                                        |
    8 -> 15        : 0        |                                        |
   16 -> 31        : 6870     |                                        |
   32 -> 63        : 516091   |****************                        |
   64 -> 127       : 838139   |*****************************           |
  128 -> 255       : 963522   |*********************************       |
  256 -> 511       : 318996   |*************                           |
  512 -> 1023      : 146827   |******                                  |
 1024 -> 2047      : 74222    |***                                     |
 2048 -> 4095      : 66658    |**                                      |
 4096 -> 8191      : 33339    |*                                       |
 8192 -> 16383     : 25817    |*                                       |
16384 -> 32767     : 13587    |                                        |
32768 -> 65535     : 8990     |                                        |
65536 -> 131071    : 425      |                                        |
```

This output, a histogram of I/O latency, came from the `biolatency` tool in the BCC tools suite (https://github.com/iovisor/bcc). Biolatency, read: block I/O latency, even has a name I can get behind. Passing a "-D" gleans a histogram breakdown of latency per disk. Where does this awesome biolatency tool get its data, you ask? Well from the enhanced Berkeley Packet Filter obviously!

Wait, what?

eBPF works like an embedded `lua` interpreter or the spidermonkey VM that executes JavaScript inside the Mozilla web browser. It resides in kernel space, ready to execute bytecode supplied from userspace. Its original intent was to filter packets without having to resort to context-switches, but it has grown to become a fully fledged kernel tracing system comparable to DTrace in long-lost Solaris.

A userspace BPF script sends a bytecode to the kernel together with a program type which determines what kernel areas the program can access. If you look at the source code for `biolatency` (https://github.com/iovisor/bcc/blob/master/tools/biolatency.py), you'll notice it is a Python program which contains a small C program inside it as a string (starting on line 56).

The Python code takes care of compiling and loading that block of C code into the kernel and then stays resident in memory, collecting data from its own kernel probe, and eventually presenting it to us, the user. I'm intentionally glossing over *a lot* of detail here, including a pre-load code verifier which guarantees your probe payload won't crash the system. There are a lot of moving parts, but the result is high-resolution, low-cost visibility into the inner-workings of the system and everything running on it. Unprecedented observability.

I'd like to spend the next few articles together digging into eBPF more deeply. My plan is to use our new friend, the biolatency tool, as a laboratory frog we can dissect together. We'll start light, talking about the various endpoints eBPF gives us to get our hooks into the kernel, and finish up with hopefully a solid place to get started crafting your own eBPF programs. Who knows, maybe we'll even filter some packets.

Take it easy.