acmqueue

Scale Failure

Using a tool for the wrong job is OK until the day when it isn't.

Dear KV,

I have been digging into a network-based logging system at work because, from time to time, the system jams up, even when there seems to be no good reason for it to do so. What I found would be funny, if only it weren't my job to fix it: the central dispatcher for the entire logging system is a simple `for` loop around a pair of `read` and `write` calls; the `for` loop takes input from one of a set of file descriptors and sends output to one of another set of file descriptors. The system works fine so long as none of the remote readers or writers ever blocks, and normally that's not a problem. The problem has come about because what was once handling fewer than 10 machines is now handling 40, some of which are remote across a wide area network. The obvious fix is to make the code nonblocking, but what I'm surprised about is that anyone would write code this way. It's obvious from the first time you look at the code that it cannot scale.

Blocked and Loopy

Dear Loopy,

I would like to say that I'm sure the original author of the code you're looking at wasn't trying to torture you; but after seeing many similar pieces of code, it's hard for me to continue to accept this particular bit of make-believe. What you're probably looking at is "throw-away" or "prototype" code that got away. The schlimazel who wrote the code probably had a boss pop into his cube one day with a "great idea" to improve the logging system by using the network and a central dispatcher, and then asked the programmer to code up something simple to toss around. That something simple is what you now see. In my mind, I see the programmer getting the code running, and—since programmers are optimists—being excited when it ran and considering it done.

The next thing I see is that once the code was deployed, people found a use for it. Code that people don't find a use for rarely causes problems, because it rarely gets executed. From 10 clients, it went to 20, and then on to the point where it broke and someone asked you to look at it.

If I were you, I'd count your blessings. Taking a single, simple `read/write` loop and converting it into a reasonably robust, nonblocking piece of code, while not trivial, isn't a massive undertaking. Of course, while you're at it, you're going to add code to report when your clients are slow, or disconnect, or cause problems, right? Right! You could easily spend days hacking around and polishing a system like this, but I would suggest that you just add enough code and hooks so that when the system goes to 100 nodes you can split your dispatcher and run more than one of them simultaneously on separate nodes, because that's the next thing you'll have to do for scalability. If you don't do this correctly, then your successor will be writing me a letter—exactly like this one.

KV

Dear KV,

My employer recently deployed a system on its network that is very sensitive to variations in network traffic. Although our team let people know that the amount of load on our network might cause problems with this particular application, it was decided to deploy the software anyway and see what happened in production. As you can imagine, most of the time things work pretty well; but occasionally, often because of random misconfigurations or because another application abuses the network resources, our shiny software fails completely, resulting in angry e-mail threads and finger pointing. At this point, there is no way to turn back, and we now live in fear of the next time someone adds a new application in the network. There are ways to work around these issues, but people seem unwilling to do the necessary work and are only interested in our group “just fixing the code.” Of course we can patch and hack the code to work around temporary problems in the network, but that doesn’t really address the problem. Why is it so difficult for people to understand when they are using a tool the wrong way?

Wrong Way Round

Dear Wrong Way,

Whenever I see people taking one tool and using it—usually poorly—for the wrong job, I am always reminded of screwdrivers. You can use a screwdriver to drive screws, yes, but you can also turn the screwdriver around use the handle as a hammer to drive nails. Of course, doing this means that you’re at risk of poking your eye out, but, you say, “I only need to drive this one nail, I’m sure it will be OK.” And it is OK, until the day when it isn’t. Software, being far more malleable than a screwdriver, is subject to this extension problem far more often than physical tools.

There are a couple of ways to make your point in these situations. One is simply to let the code break and watch people suffer. I recommend against developing an evil laugh or learning to cackle, as that will give you away. While this is an enjoyable fantasy, it’s not very practical in a work environment. There is probably a good reason for your company to use the code you’re complaining about, and it behooves you to do what you can to help them use it correctly.

Instead of screaming, or cackling, or pulling your hair out, you can try to explain to one person, rather than to a group, how the software works and its limitations. If you can find one other person who understands the problem, that can help you in two ways. First, it will make you feel less crazy—there is nothing worse than being the only person who sees or understands a problem. Second, it will help convince others of the correctness of your position. If you can get momentum behind your idea, then maybe you can convince the powers that be to use the system correctly and within its design parameters. Failing that, at least you’ll have someone to commiserate with over a beer when the system collapses again.

Like so many problems in computing, the screwdriver problem is a human problem and not a technical one, and thus it requires a human solution.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor’s degree in computer science at Northeastern University in Boston,

Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

© 2012 ACM 1542-7730/11/0200 \$10.00