Watchdogs vs. Snowflakes

**TAKING
WILD-ASS
GUESSES**

who is
KV?

click for video

Dear KV,

I've been working with a distributed, job-control system for a large computing cluster for the past year. The system was developed in-house by one of the co-founders of the company, and he continues to work on it sporadically, while a small team of us adds new features and tries to fix bugs. The code isn't terrible, but it has one major defect—if the system doesn't have enough jobs in its queues, it tends to freeze up. I've been working with one other person on my team to diagnose the problem, but it has been assigned a very low priority by management because as long as we add dummy jobs when the system would otherwise be idle, the bug doesn't occur. I've never seen a system act like this, and I have to wonder, is this kind of problem common in distributed job-control systems?

Jobless

Dear Jobless,

Is the specific problem of a system freezing up because of starvation common in distributed job-control systems? It has been my experience that each distributed system is a precious snowflake, and KV hates snow!

Let's talk first about the high-level issue—the fact that no one cares if you fix the bug, because if you put in dummy jobs, the system “just works.” The phrase *just works* is one of the most overused in computing, and what it really

indicates, in this case, is that someone is intellectually lazy, or that his or her motivation lies elsewhere. “Why should we care that we’re running our systems at 100 percent power draw, when fixing the problem would cost time and money?” Apart from the fact that computing now consumes a significant percentage of the world’s electricity, leaving a bug like this unaddressed can have other deleterious side effects.

That a system can randomly jam doesn’t just indicate a serious bug in the system; it is also a major source of risk. You don’t say what your distributed job-control system controls, but let’s just say I hope it’s not something with significant, real-world side effects, like a power station, jet aircraft, or financial trading system. The risk, of course, is that the system will jam, not when it’s convenient for someone to add a dummy job to clear the jam, but during some operation that could cause data loss or return incorrect results. I rather suspect that having a system like this jam while coordinating, for example, the balancing of electrical power across a power grid would have spectacular and perhaps fatal results.

I’m not saying every bug must be fixed at the expense of doing otherwise productive work, but it is bugs like this one that, in my experience, tend to hit at the absolute worst possible time. If the team knew about the bug in advance, it just leads to embarrassment when they have to admit they knew about such a risk before it actually happened.

It’s hard to say much about the technical issue without looking into the system itself. (Remember KV’s earlier comment about snowflakes.) The most common way of handling this type of freezing is itself not completely

A very short watchdog timeout has the potential to make the system thrash, since each restart caused by the watchdog firing will require the system to do work to return to its running state.

satisfying, and that is to have a watchdog process that sees if the system is making progress and restarts it after a suitable timeout when it believes the system is stuck.

There are several problems with the watchdog approach. The first is what the watchdog will actually do. Some watchdogs operate by restarting a stuck process, and they do this bluntly, by killing the process and restarting it. If the computations undertaken by the system are all idempotent, then there is little risk because any operation that did not complete will be restarted from the beginning and should have no side effects. Most systems have side effects, which means that such restarts can cause a cascade of errors through the whole system. If the errors are obvious, then a human operator might be able to roll back the system to a good, known state and start the system again. But what if the errors are a type of silent corruption, returning incorrect answers (as I mentioned at the start of this piece)? In that case, the watchdog is likely to do more harm than good.

Even if a watchdog approach isn't otherwise harmful, there is a second problem of choosing an appropriate timeout duration. Since the system becomes jammed when it doesn't have enough work, some people will want to set the watchdog timer to be very fast so as to prevent these jams from reducing the overall efficiency of the system. A very short watchdog timeout has the potential to make the system thrash, since each restart caused by the watchdog firing will require the system to do work to return to its running state. All the work done by the system when a process is restarted is pure overhead; it does not help the system perform the work it was intended to do. Conversely,

setting a watchdog timeout to be too long risks having the system remain stuck for long periods, again reducing overall efficiency. Too often, the choice of these timeouts is accomplished by a form of black magic, referred to as “taking a wild-ass guess,” followed by a heuristic, which is

referred to as “taking another wild-ass guess,” to see if it’s better than the first.

Do not underestimate the number of production systems that use these approaches. I believe that if we truly knew how many of the systems we depend on every day used black magic under the hood, we would all be more likely to buy land in Wyoming, build bunkers, and live in them.

Unfortunately, as KV has discussed before, debugging distributed systems is hard, but it turns out that not

debugging them and having them fail catastrophically makes for even harder days.

Related articles

Poisonous Programmers

Kode Vicious

A koder with attitude, KV answers your questions. Miss Manners he ain’t.

<https://queue.acm.org/detail.cfm?id=1348585>

From the EDVAC to WEBVACs

Daniel C. Wang

Cloud computing for computer scientists

<https://queue.acm.org/detail.cfm?id=2756508>

Too Big NOT to Fail

Pat Helland, Simon Weaver, and Ed Harris

Embrace failure so it doesn’t embrace you.

<https://queue.acm.org/detail.cfm?id=3077383>

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. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System [second edition]. He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.