Cold, Hard Cache

ON THE IMPLEMENTATION AND MAINTENANCE OF CACHES

who is
KV?

click for video

Dear KV,

Our latest project at work requires a large number of slightly different software stacks to deploy within our cloud infrastructure. With modern hardware, I can test this deployment on a laptop. The problem I keep running up against is that our deployment system seems to secretly cache some of my files and settings and not clear them, even when I repeatedly issue the command to do so. I've resorted to repeatedly using the find command so that I can blow away the offending files. What I've found is that the system caches data in many places—not in one single, easy-to-find place—so I've started a list. All of which brings me to my question: Who writes this stuff?!

Out of Cache

Dear OOC,

I'm not quite sure who writes this stuff, but I am pretty sure they do not know what they're doing. I'm also relatively sure that they're on very strong drugs, and I wish they would share.

A wise guy once said, "There are only two hard things in computer science: cache invalidation and naming things," and that wise guy was supposedly Phil Karlton, but, no matter who said it, they are partially correct. Let's skip the naming problem for now and go right into caching and the concerns one might have while maintaining a cache.

You'll note I said, "maintaining," because plenty of folks can build up a cache—that's just making copies of stuff you might want later and putting it in a place that's easier for you to find or faster for you to access. Maintaining a cache actually has several distinct operations, all of which must be carried out correctly for the cache to be of any real use.

The only reason to create a cache in the first place is that there is some piece of information that is frequently used and that can be stored in a place where it is quicker to access. For example, a system's main memory can be thought of as a cache for files on disk, because it's perfectly possible to read a file into memory, execute the instructions read from the file, and then immediately free the memory. No one builds systems like that because it's expensive to keep copying the data from disk for each execution, so we leave the data we've copied in memory until we need the memory for some other purpose. Your particular case seems like a cache of either settings or files, rather than a memory, CPU, or chip-level cache, so I'll narrow my remarks a bit more to cover just this case, but the principles apply to all sorts of caches.

Adding to a cache is relatively straightforward. As long as there is room in the cache, an entry is added, hopefully, in a place that is quicker to access the second time around. Accessing an entry in the cache requires a lookup operation, which can fail. Failing to find an entry in the cache is called a "miss" and is the cache's way of telling you to go back where you came from and look up the data in the old, slow way.

The problems you're having with your system come from its inability to invalidate stale data. If you change

Nothing
stinks
quite like
stale data—
it smells
like dead lizards.

something in files that are being deployed and your deployment system doesn't notice that change, then it's going to think that the data it has cached is fresh when, in fact, it is stale. Nothing stinks quite like stale data—it smells like dead lizards. Since I have no idea how your system is judging the freshness of your data, let's just assume for a moment that it's using the file modification timestamp, and then let's further assume that it's not looking at seconds, but instead, minutes. That means if you are doing a file save and then an immediate "deploy," your system will not, in fact, think that the data is stale. That is, if you do deploy-change-deploy in the same minute, which is quite possible for a touch typist, the system will not think that the old data is out of date. Another possibility is that the settings you're changing are in a file that's not being watched by the system, and that the thing that it cares about is some file that points to your file.

A correctly implemented cache tracks all the things being cached, not just the thing that the programmer assumes other people will modify. The problem usually comes as more types of data are added to the cache. A file here, a database entry there, and eventually what you have isn't really a well-organized cache, but instead, the data-storage equivalent of spaghetti code, where files point to files, and the file with the dragon has the pellet with the poison, but the directory with the dragon has the file that is true.

One way for you, the user, to deal with this is to be a tad brutal and flush the cache. Whereas invalidation is the careful, even surgical, removal of stale data, a flush is just

what it sounds like: it sends all the crap right down the drain, fresh *and* stale. It sounds like whoever implemented your system has made this difficult for you by scattering the requisite files all over the system, but once you have your complete list, it's probably time to write a `flush.sh` shell script to clear out all the cruft you can think of.

Coming back up for air, let's think about how a proper cache is managed, so that the next system any of us come across doesn't require the use of system-call tracing to find out why the system is misbehaving. Caches make sense only if they speed up access to frequently used data; otherwise, let the operating system or your database handle things, because they're pretty smart at managing data and keeping the fresh data near at hand.

When you build a cache, pick a key that's easy to search for. Here's a hint: strings are not easy to search for, but hashes of strings are. When you add new object types to the cache, don't do it by adding a pointer from one object to another, unless those two objects must be treated as one thing. Trying to track down whether or not a sub-object has changed by walking all the parent objects is inefficient. When using timestamps to indicate freshness, look at the seconds—seriously, computers are fast; many things happen in a computer minute. If the cache contains files, put them all under a common root directory, as this will make flushing them easier for the user.

Much of what we've talked about is local caching, and KV has left out the distributed case, for reasons of personal sanity. The advice here is a good start to building a distributed cache, because if you get this wrong in the local case, your chances of getting it right in the

distributed case are nil. If KV gets a good letter about the distributed case, he might take leave of his senses long enough to try to answer it, or he might burn the letter—and since sending letters is also a distributed system, there is no guarantee that you'll know if the message was delivered, lost, or is in a stale cache.

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 © 2017 held by owner/author. Publication rights licensed to ACM.