Cloud Calipers

NAMING THE NEXT GENERATION AND REMEMBERING THAT THE CLOUD IS JUST OTHER PEOPLE'S COMPUTERS

Dear KV,

Why do so many programmers insist on numbering APIs when they version them? Is there really no better way to upgrade an API than adding a number on the end? And why are so many systems named “NG” when they’re clearly just upgraded versions?

API2NG

Dear API2NG,

While software versioning has come a long way since the days when source code control was implemented by taping file names to hacky sacks in a bowl in the manager’s office, and file locking was carried out by digging through said bowl looking for the file to edit, programmers’ inventiveness with API names has not advanced very much. There are languages such as C++ that can handle multiple functions—wait, methods with the same names but different arguments—but these present their own problems, because now instead of a descriptive name, programmers have to look at the function arguments to know which API they’re calling.

Perhaps the largest sources of numbered APIs are the base systems to which everyone programs, such as operating systems and their libraries. These are written in C, a lovely, fancy assembler that has no truck with such fancy notions as variant function signatures. Because of

who is
KV?

click for video

this limitation of the language that actually does most of the work on all of our collective behalves, C programmers add whole new APIs when they only want to create a library function or system call with different arguments.

Take, for example, the creation of a pipe, a very common operation. Once upon a time, pipes were simple and returned a new pipe to the program, but then someone wanted new features in pipes, such as making them nonblocking and making the pipe close when a new sub-program is executed. Since `pipe()` is a system call defined both by the operating system and in the Posix standard, the meaning of `pipe()` was already set in stone. In order to add a flags argument, a new pipe-like API was required, and so we got `pipe2()`. I would say something like “Ta-da!” but it’s more like the sad trombone sound. Given that the system call interface is written in C, there was nothing to do but add a new call so that we could have some flags. The utter lack of naming creativity is shocking. So now there are two system calls, `pipe()` and `pipe2()`, but it could have been worse: we could have had `pipeng()`.

Perhaps the worst thing that Paramount ever did was name its *Star Trek* reboot *The Next Generation*, as this seems to have encouraged a generation of developers to name their shiny new thing, no matter what that thing is, ThingNG. Somehow, no one thinks about what the next, next version might be. Will the third version of something be ThingNGNG? If your software lasts a decade, will it eventually be a string of NGs preceded by a name? The use of “next generation” is probably the only thing more

Programmers are a notoriously lazy lot and making them type an extra character annoys them, which is another reason that versioned APIs often end in a single digit to save typing time.

aggravating than numeric indicators of versioned APIs.

The right answer to these versioning dilemmas is to create a descriptive name for the newer interface. After all, you created the new version for a good reason, didn't you? Instead of `pipe2()`, perhaps it might have made sense to name it `pipef()` for "pipe with a flags argument." Programmers are a notoriously lazy lot and making them type an extra character annoys them, which is another reason that versioned APIs often end in a single digit to save typing time.

For the time being, we are likely to continue to have programmers who version their functions as a result of the limitations of their languages, but let's hope we can stop them naming their next generations after the next generation.

KV

Dear KV,

My team has been given the responsibility of moving some of our systems into a cloud service as a way of reducing costs. While the cloud looks cheaper, it has also turned out to be more difficult to manage and measure because many of our former performance-measuring systems depended on having more knowledge about how the hardware was performing as well as the operating system and other components. Now that all of our devices are virtual, we find that we're not quite sure we're getting what we paid for.

Cloudy with a Chance

Dear Cloudy,

Remember the cloud is just other people's computers. Virtualized systems have existed for quite a while now and are deployed for an assortment of reasons, most of which have to do with lower costs and ease of management. Of course, the question is whose management is easier. For services that are not performance-critical, it often makes good sense to move them off dedicated hardware to virtualized systems, since such systems can be easily paused and restarted without the applications knowing that they have been moved within or between data centers.

The problems with virtualized architectures appear when the applications have high demands in terms of storage or network. A virtualized disk might try to report the number of IOS (I/O operations per second), but since the underlying hardware is shared, it is difficult to determine if that number is real, consistent, and will be the same from day to day. Sizing a system for a virtualized environment runs the risk of the underlying system changing performance from day to day. While it's possible to select a virtual system of a particular size and power, there is always the risk that the underlying system will change its performance characteristics if other virtualized systems are added or if nascent services suddenly spin up in other containers. The best one can do in many of these situations is to measure operations in a more abstract way that can hopefully be measured with wall-clock time. Timestamping operations in log files ought to give some reasonable set of measures, but even here, virtualized systems can trip you up because virtual systems are pretty poor at tracking the time of day.

Working backward toward the beginning, if you want to know about performance in a virtualized system, you'll have to establish a reliable time base, probably using NTP (Network Time Protocol) or the like, and on top of that, you'll have to establish the performance of your system via logging the time that your operations require. Other tools may be available on various virtualized environments, but would you trust them? How much do you trust other people's computers?

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 © 2016 held by owner/author. Publication rights licensed to ACM.