SIMPLIFY CONTINUOUS OPERATION TESTS WITH A PERIODIC REBOOT

By **Michael Stahl** - February 11, 2019
https://www.stickyminds.com/article/simplify-continuous-operation-tests-periodic-reboot

Continuous operation tests find important bugs, partly as a result of their long operation and partly by increasing the probability of finding statistical bugs. However, CO tests have their own downsides. Mandating a periodic reset or reboot can work around these issues, as well as save time and cost for testing, reproduction, debugging, and fix verification.

The 1982 movie *Blade Runner* describes a world where human-like androids are built to serve as slaves on faraway space colonies. The manufacturer found that memories the androids collect during their lifetime generate emotional reactions, which limit the ability to control and exploit the androids. The solution to the problem is simple if cruel: a built-in four-year life span.

Any software that operates a long time collects "memories"—this may be data in log files, updates to a database, leftover bits in freed memory on the RAM, etc. One of the risks associated with this phenomenon is that under certain circumstances, it can cause a failure.

Sometimes it is a bug in the product itself: The developer forgot to release resources properly, as in file handles. As a result, the system runs out of resources when operating for a very long time. In other cases, it can be an interaction with the operating system: The log file is never cleared and eventually grows above the maximum file size supported by the OS, or opening this file becomes longer and longer, eventually triggering a watchdog timer. There are many other types of failures that show up only after running the system for days or weeks.

One of the ways to deal with this issue is to run very long tests. In teams I worked with, these were called continuous operation (CO) tests. CO tests run the system for hours or days and check that the system behaves. The tests usually monitor the memory usage, to catch memory leaks; monitor other resources, such as file handles or the stack status; and periodically attempt some functional action to ensure the system is still alive. Performance speed, CPU, and power usage are other commonly monitored parameters.

CO also targets errors caused by accumulation errors, such as a rounding error in an algorithm that starts to make a difference on the program's output only after millions of calculations, or a slight hardware-related inaccuracy in frequency generators that eventually causes a meaningful difference between internal clocks of two separate modules that interact with each other.

Apart from bugs that would show up only after very long execution time, CO also targets statistical bugs. These bugs appear when a rare combination of events takes place, such as a crash that occurs when a certain process of the operating system starts exactly when the program under test writes to a log file. If, for example, the OS process runs once a minute for ten milliseconds and the program writes to the log file about once every hour for one millisecond, you need pretty bad luck to have these two functions occur at exactly the same time. But if you run the program for 230 hours, you have a 90 percent chance that the situation will happen at least once.

CO testing poses two main challenges. The first is that it's hard to decide what would be a long enough test time. Let's say you decided on forty-eight hours. Why this number? What's the logic behind it? Is it necessarily better than twenty-four hours? Is there a business need to test more than ten hours? Without a clear and justified requirement for a certain continuous operation time, the people who define the requirement are the testers, or maybe the product manager, by deciding how long to run CO tests. Often this decision is made based on logistics considerations, a guess, or just picking a nice, round number like twenty-four hours or a week without any real justification.

The other challenge is the cost of CO testing. I am not talking about the direct cost—it's not really expensive to get a PC and let it run the code for a week. Much more significant is the indirect cost. First, it's a long test. So if we find a bug after fifteen hours, it may mean that reproducing the bug calls for a fifteen-hour trial—if we are lucky and the bug is reproduced on the first attempt. Once the fix is in place, we need to run the CO to its full length again, hoping the fix did not introduce another bug.

The long test duration may also prove a problem when thinking of exit criteria. If releasing the product is conditioned on successful completion of the CO tests, it is very possible that CO testing will be the one test that defines the minimum time it takes to complete a test cycle.

So we established that CO testing is a pain. Is there anything we can do about it? I think we can.

Instead of investing energy ensuring that the system won't fail during prolonged execution, maybe we can take a different route. The idea is as simple as the *Blade Runner* solution: force the software to reboot, restart, or reset periodically, whatever fits your system. (This idea, by the way, is not mine. I heard it in a conference ten years ago and I know that there are systems that do just that.)

For the sake of discussion, let's say we'll restart every twenty-four hours, although this is a totally arbitrary number. In a real project we would need to take into account the product, its uses, and its users, and define the correct restart period. I believe strongly that every product designer should consider this option, especially for embedded software products where resources are scarce.

To start, we need to ask why our product needs to run continuously for long periods of time. Sometimes it's very simple: If the system controls or monitors a critical activity where a bad event can happen within seconds, such as a pacemaker, obviously we don't want to periodically reset it. But there are many systems where a quick reboot while the system is idle can save a lot of headaches.

My kids, for example, know that I will not answer a distress call of "Dad, there is no internet" before they turn the router off and back on. There are a few articles explaining how to do an auto-reboot on your router as a remedy for the router getting slower over time. Even complex systems like a CT scanner can, from an operations point of view, be reset once every twenty-four hours. Surely we need to make sure the machine is not scanning anyone, but after a scan is over, we can restart the machine before seeing a new patient.

Assuming the system meets the conditions for a periodic reset, and that the code needed to enforce this regime was added to the product and tested to work correctly, we gain a number of benefits:

- If we previously set the CO test time to a certain value by a combination of intuition, baseless optimism, and project constraints, we now have a strong factual basis to decide on a twenty-four hour CO test: Because we made sure the system will never operate longer than this time before it is forced to reboot
- It saves time—test time, bug reproduction time, and debugging and fix verification time
- We can prevent certain bugs from occurring, simply because these bugs need more than twenty-four operation hours to develop, such as cumulative errors

A note on the last point, which seems a bit immoral: We ignore the risk that there is a bug in the system by claiming, "We made sure that even if a bug does exist, it won't be activated." Is this a problem?

I think not. We regularly release products to the field knowing that we didn't remove all the bugs—we just can't. We take the risk that our users will experience some of them, and we think it's acceptable. With bugs that show only after a long operation, our position is actually better: We define an operation envelope that prevents these bugs from materializing at all. This is an acceptable approach in other aspects of our lives, like when road safety engineers construct a concrete separator between lanes. This does not solve the problem of drivers veering off course due to DUI or excessive speed, but it does practically eliminate the risk of frontal collision.

But we neglected one thing. There was another type of bug that CO test was targeting: the statistical bugs. If we now reduce CO time to twenty-four hours, we also reduce the probability of catching these bugs! But this issue can be dealt with rather simply. Instead of running one system for, say, seventy-two hours, we can run three systems in parallel for twenty-four hours. We get more or less the same probability of finding statistical bugs as before. Indeed, some statistical bugs may be more prone to happen after a longer operation time, but then they fall under the previous consideration: By resetting every twenty-four hours, we prevent the higher chance that these bugs will occur.

If we are considering adopting a periodic reset, there is still some analysis to do before moving forward. We need to check how deep our reset or reboot actually is. Are any settings or values from the previous operation, such as database records, values committed to non-volatile memory or to files on the disk, or undeleted log files left intact, even on reboot? We may conclude that we need to add some code to ensure a real clean start after reset, such as changing the name of the log file so that when a new execution period starts, logs are written to a new file.

I think it's fair to assume that there are almost no cases, except in very simple systems, that just turning the system off and on brings it back to a pristine state, with no leftovers from the previous run. Actually, I assume that in most cases we *don't want* the system to forget everything. So we need to put some deep thought into deciding what to keep, what will impact the operation of the system going forward, and what effort is worthwhile to put into clearing certain things.

Finally, here are two real-life examples of when a reboot helped:

- The Patriot missile deployed during the Persian Gulf War was an anti-missile defense. An accumulative error in one of the calculations caused a mismatch between different clocks in the system, causing the algorithm that controlled the launch to make incorrect decisions. The Israeli Defense Force identified this problem and found a simple workaround until a proper fix was deployed: reset the system every two days. This workaround was not done by the US Army, with devastating results: The system wrongly decided not to shoot at an incoming Scud missile, which killed twenty-eight US soldiers and wounded about a hundred others.

- The architecture for a product I worked on many years ago included a watchdog timer that checked if a certain part of the system was inactive for more than two milliseconds. If the timer expired, regardless of the reason, the system did an internal reset of some of the circuits. The reset action was very fast and the user could not notice that anything out of the ordinary just took place. It did not fix the problem that caused the system to stall, although we did follow the logs and fixed the problems one by one. But even before those fixes, we had a system that almost never hung.

Continuous operation tests find important bugs, partly as a result of their long operation and partly by increasing the probability of finding statistical bugs. In many products it is feasible to work around bugs caused by long operation by mandating a periodic reset or reboot as part of the product's requirements. To ensure statistical bugs are still caught even when the maximum operation time is lowered, we can run a number of systems in parallel.

The benefits of specifying a periodic reboot are that the decision about the periodic interval for a reboot is part of the product's architecture, taking into account the use cases. It is therefore well thought-out and justified, and the cost of CO testing, reproduction, debugging, and fix verification are reduced significantly.

This article benefitted from a discussion with the following colleagues: Yonit Elbaz, Dani Almog, Kobi Halperin, Yan Baron, Nir Gallner, Jan Van-Moll, Dvir Yoskovitz, Noam Kfir, Tal Peer, Yaron Tsubery, and Yonatan Klein.