acmqueue Raw Networking

Relevance and repeatability

Dear KV,

The company I work for has decided to use a wireless network link to reduce latency, at least when the weather between the stations is good. It seems to me that for transmission over lossy wireless links we'll want our own transport protocol that sits directly on top of whatever the radio provides, instead of wasting bits on IP and TCP or UDP headers, which, for a point-to-point network, aren't really useful.

Raw Networking

Dear Raw,

I completely agree that the best way to roll out a new networking service is to ignore 30 years of research in the area. Good luck.

Second only to operating system developers—all of whom want to rewrite the scheduler (See “Bugs and Bragging Rights”, second letter <http://queue.acm.org/detail.cfm?id=2542663>)—are the networking engineers and developers who want to write their own protocol. “If only we could go at it with a clean sheet of paper, we could do so much better than the ARPANET, since that was designed for old, crappy hardware, and ours is shiny and new.” That statement is both true and false, and you had better be damned sure about which side of the Boolean logic your idea lies on before you write a single line of new code.

The Internet protocols are not the be-all and end-all of networking, but they have had more research and testing time applied to them than any other network protocols currently in existence. You say you're building a wireless network with—I'm sure—the highest quality gear you can buy. Wireless networks are notoriously lossy, at least in comparison to wired networks. And it turns out that there has been a lot of research done on TCP in lossy environments. So although you will pay an extra 40 bytes per packet to transport data over TCP, you might get some benefit from the work done—to tune the bandwidth and round-trip-time estimators—that will exist in the nodes sending and receiving the data.

Your network is point-to-point, which means you don't think you care about routing. But unless all the work is always going to be carried out at one or the other end of this link, you're eventually going to have to worry about addressing and routing. It turns out that someone thought about those problems, and they implemented their ideas in, yes, the Internet protocols.

The TCP/IP protocols aren't just a set of standard headers, they are an entire system of addressing, routing, congestion control, and error detection that has been built upon for 30 years and improved so that users can access the network from the poorest and most remote corners of the network, where bandwidth is still measured in kilobits and latencies exceed half a second. Unless you're building a system that will never grow and never be connected to anything else, you had better consider whether or not you need the features of TCP/IP.

I am all for clean-sheet research into networking protocols. There are many things that have not

been tried and some that have been, but didn't work at the time. Your letter implied not so much research, but rollout, and unless you've done your homework, this type of rollout will flatten you and your project.

KV

Dear KV,

You write about the importance of testing, but I haven't seen anything in your columns on how to test. It's fine to tell everyone that testing is good, but some specifics would be helpful.

How Not Why

Dear How,

The weasel's way out of this response would be to say that there are too many ways to test software to give an answer in a column. After all, many books have been written about software testing. Most of those books are dreadful, and for the most part, also theoretical. Anyone who disagrees can send me an email with their favorite book on software testing and I'll consider publishing the list or trashing the recommendation. What I will do here is describe how I have set up various test labs for my specific type of testing, and maybe this will be of some use.

There are two requirements for any testing regimen: *relevance* and *repeatability*. Test-driven development is a fine idea, but writing tests for the sake of writing tests is the same as measuring a software engineer's productivity in KLOC. To write tests that matter, test developers have to be familiar enough with the software domain to come up with tests that will confirm that the software works and that also attempt to break the software. Much has been written about this topic, so I'm going to switch gears to talk about repeatability.

Tests are considered repeatable when the executions of two different tests on the same system do not interfere with each other. A concrete example from my own work is the population of various software caches—such as routing and ARP tables—that might speed up the second test in a series of tests of packet forwarding. To achieve repeatability, the system or person running the test must have complete control over the environment in which the test runs. If the system being tested is completely encapsulated by a single program with no side effects, then running the program repeatedly on the same inputs is a sufficient level of control. But most systems are not so simple.

Working from the concrete example of testing a firewall: To test any piece of networking equipment that passes packets from one network to another, you need at least three systems, a source, sink, and the device under test (DUT, in test parlance). As I pointed out earlier, repeatability of tests requires a level of control over the systems being tested. In our network testing scenario, that means each system requires at least two interfaces and the DUT requires three. The source and sink need both a control interface and the interface on which packets will be either sent to or received from the DUT. "Why can't we just use the control interfaces to source and sink the packets?" I hear you cry. "Wiring all that stuff is complicated and we have three computers on the same switch, we can just test this now." The way it works is that the control and test interfaces must be distinct on all the systems to prevent interference during the test. No matter what you are testing, you must make sure that you reduce the amount of outside interference unless that is what you are intending to test. If you want to know how a system reacts with interference, then set up the test to introduce the interference, but don't let interference show up out of nowhere. In our specific networking case, we

want to retain control over all three nodes, no matter what happens when we blast packets across the firewall. Retaining control of a system under stress is non-trivial.

Another way to maintain control over the systems is to have access to a serial or video console. This requires even more specialized wiring than just a bunch more network ports, but it is well worth it. Often, bad things happen, and the only way to regain control over the systems is via a console login.

The ultimate fallback for control is the ability to remotely power-cycle the system being tested. Modern servers have an out-of-band management system, such as IPMI, that allows someone with a user name and password to remotely power-cycle a machine as well as do other low-level system management tasks including connecting to the console. Whenever someone wants me to test networked systems in the way I'm describing, I require them to have either out-of-band power management via a network connected power controller or IPMI on the systems in question. There is nothing more frustrating during testing than having a system wedge itself and having to either walk down to the data center to reset it or, worse, having your remote hands have to do it for you. The amount of time I've wasted in testing because someone was too cheap to get IPMI on their servers or put in a proper power controller could have been far better spent killing the brain cells that had absorbed the same company's poorly written code. It seems that inattention to detail is pervasive, and when I see a poor testing setup, I should be prepared to see poor code as well.

At this point, we know that we have to retain control over the systems—and we have several ways to do that via separate control interfaces—and ultimately, we have to have control over the system's power. The next place that most test labs fall down is in access to necessary files.

Once upon a time a workstation company figured out that they could sell lots of cheap workstations if they could concentrate file storage on a single, larger, and admittedly more expensive server. Thus was born the Network File System, the much-maligned, but still relevant, way of sharing files among a set of systems. If your tests can in any way destroy a system, or if upgrading a system with new software removes old files, then you need to be using some form of networked file system. Of late I've seen people try to handle this problem with distributed version control systems such as git, where the test code and configurations are checked out onto the systems in the test group. That might work if everyone were diligent about checking in and pushing changes from the test system. But in my experience, people are never that diligent, and inevitably someone upgrades a system that had crucial test results or configuration changes on it. Using a networked file system will save whatever hair you have left on your head. (I should have learned this lesson sooner.) Make sure that the networked file system traffic goes across the control interfaces and not the test interfaces. That should go without saying, but in test lab construction, much of what I think could go without saying needs to be said.

At this point we have fulfilled the most basic requirements of a networking test system: We have control over all the systems, and we have a way to make sure that all the systems can see the same configuration data without undue risk of data loss. From here it's time to write the automation that controls these systems. For most testing scenarios, I tend to just reboot all the systems on every test run, which clears all caches. That's not the right answer for all testing, but it definitely reduces interference from previous runs.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

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.

© 2015 ACM 1542-7730/15/0100 \$10.00