ECE459: Programming for Performance Winter 2020

Lecture 34 — DevOps for P4P

Patrick Lam & Jeff Zarnett *2019-12-09*

Two topics today: 1) DevOps considerations (think big); 2) the cost of scalability (think small).

DevOps for P4P

So far, we've talked almost exclusively about one-off computations: you want to figure out the answer to a question, and you write code to do that. Our assignments have been like that, for instance. But a lot of the time we want to keep systems running over time. That gets us into the notion of operations.

The theme today will be using software development skills in operations (e.g., system administration, database management, etc).

Even when we've talked about multi-computer tools like MPI and cloud computing, it still has not been in the context of keeping your systems operational over longer timescales. The trend today is away from strict separation between a development team, which writes the software, and an operations team, which runs the software.

This is not the case at startup companies. There isn't the money to pay for separate developers and operations teams. And in the beginning there's probably not that many servers, just a few demo systems, test systems, etc... but it spirals out from there. You're not really going to ask the sales guys to manage these servers, are you? So, there's DevOps.

Is DevOps a good idea? Like most ideas it can be used for both good and evil. There's a lot to be said for letting the developers be involved in all the parts of the software from development to deployment to management to training the customers. Developers can learn a lot by having to do these kinds of things, and be motivated to make proper management and maintenance tools and procedures.

There's also something to be said for never letting the developers out of their cubes and keeping them far, far away from the customers. They will be scared. Both parties.

Thanks to Chris Jones and Niall Murphy for the following points.

Configuration as code

Systems have long come with complicated configuration options. Sendmail is particularly notorious, but apache and nginx aren't super easy to configure either.[1] The first principle is to treat *configuration as code*. Therefore:

- use version control on your configuration.
- test your configurations: that means that you check that they generate expected files, or that they spawn expected services. (Behaviours, or outcomes.) Also, configurations should "converge". Unlike code, they might not terminate; we're talking indefinitely-running services, after all. But the CPU usage should go down after a while, for instance.
- aim for a suite of modular services that integrate together smoothly.
- refactor configuration files (Puppet manifests, Chef recipes, etc);
- use continuous builds (more on that later).

Furthermore, it's an excellent idea to have tools for configuration. It's not enough to just have a wiki page or github document titled "How to Install AwesomeApp" (fill in name of program here). There are tons of great tools like the Red Hat Package Manager (RPM) which will allow you to make the installation and update process automatic and simple. Complicated means mistakes... people forget steps. They are human.

Servers as cattle, not pets

By servers, I mean servers, or virtual machines, or containers. At a certain scale (and it's smaller than you think), it's useful to mass-produce tools for dealing with servers, rather than doing tasks manually. At a minimum, you need to be able to set up these servers without manual intervention. They should be able to be spun up programmatically.

Common infrastructure

Use APIs to access your infrastructure. Some examples:

- storage: some sort of access layer to MongoDB or Amazon S3 or whatever;
- naming and discovery infrastructure (more below);
- monitoring infrastructure.

Try to avoid one-offs by using, for instance, open-source tools when applicable. Be prepared to build your own tools if needed.

Design for 10× growth, redesign before 100×

[original credit: Jeff Dean at Google] This discussion is based on Martin Fowler's piece on sacrificial architecture: `http://martinfowler.com/bliki/SacrificialArchitecture.html`.

Consider eBay: in 1995, perl scripts; in 1997, C++/Windows; in 2002, Java. Each of these architectures was appropriate at the time, but not as the requirements change. The more sophisticated successor architectures, however, would have been overkill at an earlier time. And it's hard to predict what would be needed in the future.

> "Perf is a feature".
> — Jeff Atwood

[1] If anyone should be foolish enough to want to look into procmail, well, good luck to you...

That is, you apply developer time to perf, and you make engineering tradeoffs to get it. Some thoughts:

- design with the eventual replacement in mind;
- don't abandon internal quality (e.g. modularity);
- sacrifice individual modules at a time, not the whole system;
- you can also implement new features with a rough draft and deploy to a test audience.

Naming

Naming is one of the hard problems in computing. There is a saying that there are only two hard things in computers: cache invalidation, naming things, and off by one errors.

There are a lot of ways to name things. We'll talk about systems/VMs[2], but naming is necessary for resources of all kinds.

In brief:

- use canonical one-word names for servers;
- but, use aliases to specify functions, e.g. 1) geography (nyc); 2) environment (dev/tst/stg/prod); 3) purpose (app/sql/etc); and 4) serial number.

This enables you to have a way of referring to each machine in an absolute sense, but also allows you to use functional names when creating dependencies between systems.

There's also the Java package approach of infinite dots: live.application.customer.webdomain.com or however you want to call it. But pick something and be consistent.

Other Topics

Beyond the five principles above, there are a couple more techniques that particularly apply to DevOps:

Continuous Integration. This is now a best practice. It's enabled by the use of version control, good tests, and scripted deployments. It works like this:

- pull code from version control;
- build;
- run tests;
- report results.

What's also key is a social convention to not break the build. These things get done automatically on every commit and the results are sent to people by e-mail or instant messenger (because e-mail is for old people, right?).[3]

CI is good for all code, but it's especially good for configuration-as-code, which is especially likely to break in different environments.

[2] `http://mnx.io/blog/a-proper-naming-scheme`

[3] I did work at a company where the person who broke the build got a sign outside his cubicle that said IOTD - Idiot of the Day. I'm not too proud to admit that I won this award on my last day of the co-op term.

Canarying. Deploy new software incrementally alongside production software, also known as "test in prod". Sometimes you just don't know how code is really going to work until you try it. After, of course, you use your best efforts to make sure the code is good. Steps:

- stage for deployment;
- remove canary servers from service;
- upgrade canary servers;
- run automatic tests on upgraded canaries;
- reintroduce canary servers into service;
- see how it goes!

Of course, you should implement your system so that rollback is possible.

Monitoring. Monitoring is surprisingly difficult. There are a lot of recommendations about what to monitor and what to do about it. We care about performance so here are a few things to think about:

- CPU Load
- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Application Response Times

With multiple systems, you will want some sort of dashboard that gives an overview of all the multiple systems in a summary. The summary needs to be sufficiently detailed that you can detect if anything is wrong, but not an overwhelming wall of data. Then you do not necessarily want to pay someone to stare at the dashboard and press the "Red Alert!" button if anything goes out of some preset range of what is okay. No, for that we need some automatic monitoring.

Here's one way to think about it.

- **Alerts**: a human must take action now;
- **Tickets**: a human must take action soon (hours or days);
- **Logging**: no need to look at this except for forensic/diagnostic purposes.

A common bad situation is logs-as-tickets: you should never be in the situation where you routinely have to look through logs to find errors. Write code to scan logs.

It is very important to be judicious about the use of alerts. If your alerts are too common, they get ignored. When you hear the fire alarm in a building, chances are your thought is not "the building is on fire; I should leave it immediately in an orderly fashion.". More likely your reaction is "great, some jerk[4] has pulled the fire alarm for a stupid prank or to get out of failing a midterm." This is because we have been trained by far too many false

[4]This is the PG-13 version of what I actually think.

alarms to think that any alarm is a false one. It's a good heuristic; you'll be correct most of the time. But if there is an actual fire, you will not only be wrong, you might also be dead.

Still, alerts and tickets are a great way to make user pain into developer pain. Being woken up in the middle of the night (... day? A lot of programmers are nocturnal, now that I think of it) because of some SUPER CRITICAL ticket OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson that production code needs to be written carefully, reviewed, QA'd, and perhaps run by a customer or two before it gets deployed to everyone. Developers, being human (... grant me some leeway here), will probably take steps to avoid their pain[5]. and they will take steps that keep these things from happening in the future: good processes and monitoring and all that goes with it.

[5]There is a great quotation to this effect by Frédéric Bastiat about how men will avoid pain and work is pain.