Lecture 35 — DevOps for P4P

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 10, 2017

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.

But a lot of the time we want to keep systems running over time.

That gets us into the notion of operations.

Keep it Rolling

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.

Disaster Girl Strikes Again

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.

DevOps - Good Plan?

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 lifecycle.

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.

On the Other Hand...

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.

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.

The first principle is to treat *configuration as code*.

Configuration as Code

- use version control on your configuration.
- test your configurations.
- aim for a suite of modular services that integrate together smoothly.
- refactor configuration files.
- use continuous builds (more on that later).

Autoconfig

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.

Rule of Ten

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

“Perf is a feature”.

— Jeff Atwood

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.

Naming Suggestions

- 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.

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

Pick something and be consistent.

Continuous Integration

- 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.

Canarying

Canarying

Deploy new software incrementally alongside production software, also known as “test in prod”.

- 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

Things to think about:

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

Dashboard

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.

No, for that we need some automatic monitoring.

Red Alert

- **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.

Action Stations! Set Condition One Throughout the Ship

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.”.

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.

Not the Kittens!!!

Alerts and tickets are a great way to make user pain into developer pain.

Some SUPER CRITICAL ticket OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson...

Devs will take steps that keep these things from happening in the future.

Clusters vs Laptops

The key idea: scaling to big data systems introduces substantial overhead.

Let’s just see how, say, a laptop compares, in absolute times, to 128-core big data systems.

Clusters vs Laptops

Big data systems haven't yet been shown to be obviously good; current evaluation is lacking.

The important metric is not just scalability; absolute performance matters a lot.

We don't want a situation where we are just scaling up to *n* systems to deal with the complexity of scaling up to *n* systems.

Or, as Oscar Wilde put it: "The bureaucracy is expanding to meet the needs of the expanding bureaucracy."

Methodology

We'll compare a competent single-threaded implementation to top big data systems.

The domain: graph processing algorithms, namely PageRank and graph connectivity (for which the bottleneck is label propagation).

The subjects: graphs with billions of edges, amounting to a few GB of data.

Results

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

Takeaways

- “If you are going to use a big data system for yourself, see if it is faster than your laptop.”
- “If you are going to build a big data system for others, see that it is faster than my laptop.”