ECE459: Programming for Performance Winter 2016

Lecture 35 — DevOps for P4P

Patrick Lam

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.

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. 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).

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

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 computer science. There are a lot of ways to name things. We'll talk about systems/VMs[1], 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.

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;

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

- build;
- run tests;
- report results.

What's also key is a social convention to not break the build.

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

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. 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.

Clusters versus Laptops

There is a paper about this:

Frank McSherry, Michael Isard, Derek G. Murray. "Scalability! But at what COST?" HotOS XV.

This part of the lecture is based on the companion blog post[2].

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.

Conclusion. 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 too.

[2] `http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html`

Methodology. We'll compare a competent single-threaded implementation to top big data systems, as described in an OSDI 2014 (top OS conference) paper on GraphX[3]. 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. 128 cores don't consistently beat a laptop at PageRank: e.g. 249–857s on the twitter_rv dataset for the big data system vs 300s for the laptop, and they are 2× slower for label propagation, at 251–1784s for the big data system vs 153s on twitter_rv. (See the blog post for the full results).

Wait, there's more. I keep on saying that we can improve algorithms for additional performance boosts too. But that doesn't generalize, so it's hard to teach. In this case, two improvements are: using Hilbert curves for data layout, improving memory locality, which helps a lot for PageRank; and using a union-find algorithm (which is also parallelizable). "10× faster, 100× less embarrassing". We observe an overall $2\times$ speedup for PageRank and $10\times$ speedup for label propagation.

Takeaways. Some thoughts to keep in mind, from the authors:

- "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."

[3] `https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-gonzalez.pdf`