Editor: **Robert Blumen**
Symphony Commerce
robert@robertblumen.com

The Modern Cloud-Based Platform

Stefan Tilkov

THIS MONTH'S EXCERPT from Software Engineering Radio (www.se-radio.net) features the third in a recent series of podcasts touching microservices, beginning with *210: Stefan Tilkov on Microservices* and *213: James Lewis on Microservices*. Docker was the subject of the fourth podcast, episode 217, in which Charles Anderson tracked down that project's founder for an in-depth discussion.

Netflix established itself as a company by disrupting the video rental industry with monthly pricing, a deep back catalog, and machine-learning-based recommendations. But it's barely in the DVD business now. The second Netflix revolution has been an aggressive move into streaming content from the Amazon cloud. Netflix and its US customers are one of the heaviest users of the US Internet, by some estimates accounting for over one-third of all traffic during peak movie-watching hours. Its movie-streaming business has been the source of technological innovations as it has innovated to meet consumers' ever-rising standards.

Much of what Netflix has learned is now public information through the Netflix open-source stack and the tireless efforts of Adrian Cockroft (formerly of Netflix), one of the major cloud architects during his years there. Cockroft has been a regular speaker at tech conferences, many of which are online.

In SE Radio episode 216, Cockroft and our newest host, Stefan Tilkov, converse about Netflix's move to the cloud, development speed as the critical competitive factor, how the monolith gave way to microservices, microservices at scale, microservices and DevOps, the Netflix service discovery infrastructure, distributed debugging in a deep microservices stack, geographic redundancy on the Amazon cloud, being always on, availability over consistency, the strategic plan behind open source, how open source helps with hiring in a competitive market, and what Adrian is doing post-Netflix in the venture-capital field.

We would enjoy hearing from readers of this column and listeners to the podcast. We accept incoming emails at se-radio@computer.org and tweets and direct messages to @seradio. You can also visit our Facebook page, Google+ group, and LinkedIn group. To hear this interview in its entirety, visit www.se-radio.net. —Robert Blumen

It's hard to read any sort of article these days that doesn't somehow mention the cloud or cloud computing. Despite that, how do you define those things? What is the cloud, and what does cloud computing mean to you?

The biggest change is when someone working at a company thinks, "I need some machines to do something." You have to file a ticket and wait for somebody else to get around to sorting out that ticket. In some big companies, it

Continued on p. 113

0740-7459/15/$31.00 © 2015 IEEE

Continued from p. 116

takes a month or two to get just a single machine. The real thing that makes it cloud computing is self-service. You make an API call, and a few minutes later a bunch of machines turn up. That is the most fundamental difference. It speeds up the whole procurement cycle. It makes everything much more dynamic.

You can use the cloud as a faster way to do the things you used to do in datacenters. But the really interesting things come when you start realizing what happens when you use it in a much more dynamic way by using machines as ephemeral resources. You can turn them on, turn them off, use a machine for a couple of hours, and then give it back. That is the essence of cloud. It comes down to putting self-service tools in the hands of the developers to do things themselves, rather than making "what operations used to do" into an API.

Developers don't care whether it's a public cloud like AWS [Amazon Web Services] or a private cloud inside the company, as long there's enough capacity for whatever you need to do. It's just there.

How did Netflix end up moving to the cloud?

Netflix started off as a DVD shipping company. It wasn't even seen as a very big technology company at the time. When I joined, its personalization algorithms were considered its primary interesting technology, not its scale.

Netflix had around 6,000,000 customers when I joined in 2007. Typically, every weekend the customers would visit the website, decide what DVDs they wanted to have shipped in the next week, and prioritize by shuffling their queue. Every time they sent a disc back, we would send them another one. The interaction with Netflix was sending a disc in through the postal service. That required a few tens of Web servers, a few back-end machines, and a big database running on a monolithic centralized app.

That year, we launched streaming with a very small catalog, but it started to take off quite quickly. When you interact with a streaming service, every click goes to the website. You browse around the website. When you decide that you want to watch something, you click Play. Then traffic goes back and forth figuring out what to do: finding the machine, finding the movie, doing authorization, giving you a security key to decode that movie, and then logging the activity so that we can make sure that there's good quality of service (rebuffers, calculating the speed to run at, and determining which content delivery network to use).

There are enormously many transactions to the back end. Streaming generates around a thousand times more Web requests than the DVD service. That was fine when we were just starting out with a small number of machines, a small number of movies, and not many customers using it. But the usage rate started to increase very rapidly because there was nothing stopping customers from watching a lot of movies. They no longer had to send a DVD back and wait for another one. So, we started getting a lot of people binge-watching.

The number of interactions people had with Netflix, the number of things they watched, and the number of interactions with the website per view all went up by orders of magnitude. Our datacenter consisted of a couple of small machines in the corner that were put in to initially launch streaming. And they were running out of capacity incredibly quickly. In 2008, there was a big outage when the central monolithic app broke due to a storage problem (a corruption in the storage area network corrupted Oracle). It was a big mess.

As a result, we decided we were not very good at running stuff in the datacenter. We started thinking about how to scale for this incredible future workload, where we didn't know how fast it would grow. And that really comes down to the core of why this is interesting: because we could not predict how much capacity we would need.

In 2009, we moved some of the back-end batch workloads like encoding movies to the cloud. In 2010, we moved the front-end website to the cloud. In 2011, we moved the database back end so that the master copies of all the data were in the cloud, and in 2012, we started open-sourcing the tooling we'd built to do that. Those are the main history points.

You've hinted that software architecture changes when you move to the cloud. What changes?

You can use cloud as nothing more than a faster way to do datacenter stuff, but that misses most of the benefit. The real benefit comes when you start doing things that you couldn't have done in your datacenter. You can trivially do hardware experiments that last a few days on a huge scale, scattered all over the world, something that you wouldn't even think of doing if you had to tell your ops guys, "Hey, I need a liberally distributed database with a hundred nodes in it and a couple hundred terabytes of solid state disc. And I'd like it this afternoon, in six different datacenters. And whatever."

We did this and we did it without asking permission. It took about

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 217—Charles Anderson talks with James Turnbull, the creator of Docker, a popular lightweight Linux deployment tool. Somewhere between a process and a virtual machine, Docker is emerging as a container for isolating microservices.
- 218—Robert Blumen discusses the Command-Query-Responsibility Segregation (CQRS) architectural pattern with Udi Dahan, one of the pattern's cocreators. CQRS formally separates a distributed system into a write master and one or more read models.
- 219—Jeff Meyerson interviews Apache Kafka committer Jun Rao on the high-throughput distributed event bus that combines features of messaging and publish–subscribe.

UPCOMING EPISODES

- 220—Robert Blumen sits down in person with Jon Gifford for a conversation about logging, APIs, log record formats, and how search engines are transforming the collection and interpretation of program messages.
- 221—Johannes Thönes and chief guru Jez Humble converse on the origins of continuous delivery (CD) in the lean movement, how to build a CD culture, and how to introduce CD in regulated environments.
- 222—Apache Storm Founder Nathan Marz chats with Jeff Meyerson about stream processing, "real-time Hadoop," the lambda architecture, and thinking about streaming in terms of spouts and bolts.

20 minutes to create. We built a very write-intensive globally distributed database to see what would happen. The decision to do it was made while we were walking out of a meeting. The guy who did it wandered by somebody else's cube and asked that person to set it up. That afternoon it was created. We put 18 Tbytes of data from backup on it. And then we hammered the thing as hard as we could with all kinds of error and failure injections to make sure it worked well. A few days later we removed it.

You couldn't do that if you had real infrastructure because it would take too long to get approval. In a datacenter, you would need a multimillion-dollar machine. I didn't know in advance what it would cost, but it worked out to a few hundred dollars per hour. The value we got out of it was much greater: at a meeting the following week, we said, "We just proved this works."

At the time, there was an internal argument going on about how we would build distributed systems and whether we could rely on high bandwidth in a global cloud. When you go to a meeting with working code or benchmarks, you win arguments. That short-circuited an enormous amount of what would have been debate and justification.

Can you explain the Chaos Monkey?

I will explain the principle, and then it will be obvious that it makes sense. There is an analogy of cattle versus pets. If you know your machines in production by name, and if one goes down everyone gets upset, then that's a pet. You have to take it to the vet if it gets sick. The other kinds of machines are cattle. When you have a herd of cattle in the field, they produce so many gallons of milk. If a few of them die, you get slightly less milk that day, and you buy some more cattle.

The principle that Netflix adopted was that everything in production is a herd of cattle. There are no pets, no individual machines that if one went down anybody would care about. Everything is on an autoscaler, even a single machine.

Once you have established that principle, then you have to test that compliance by killing individual machines chosen at random. That is what the Chaos Monkey does. It picks a random time to stop some machines. The autoscaler should automatically replace it. If somebody snuck an individual machine into production, the Chaos Monkey killed it, and they got upset, well, they should not have done that, right? It forces the developers to think in the new way. Everything you launch is on an autoscaler even if there is only one machine in the group. It must be a stateless, disposable machine that can be restarted.

They took it to the next level

with the data layer, which is a triple-replicated Cassandra back end. The Chaos Monkey kills those as well—including the discs that are inside the instances. You might lose a few hundred gigabytes of data when you delete the instance. It's not attached storage: the disks are inside the instance. But it's replaced as the data is resynchronized from the other two copies, proving that you can build an ephemeral data layer as well.

This is a different way of thinking from a sort of datacenter mentality where machines should always stay up. You would use perfect machines if you could. Instead, you create herds of machines that are extremely resilient because you can lose large numbers of them and everything still works.

Let's talk about the CAP (consistency, availability, partition tolerance) theorem. Which part of it do you apply? Which side of the triangle do you lean to?

You have to decide whether consistency or availability is most important to you. The basic principle of Netflix is that no partition or failure should take out the service. We lean very heavily to the availability side when things are partitioned, which means that if you slice Netflix up and drop all the networks between all the different parts of the system, the system continues to work. The isolated parts will just carry on working. And they'll gradually become inconsistent with the rest of the system. When you reconnect, "last writer wins" takes over. Cassandra kicks in. Whoever wrote a piece of data last ends up overwriting whatever was written in the meantime.

The system gradually gets back to being consistent, although you may lose a few updates if you modified something that was modified somewhere else. Generally it doesn't, and anyway, it's better to deal with inconsistency than to be down. For a service like Netflix, where 50 million people are trying to watch movies around the world, they have an expectation that when they turn on a TV set, it should just work. There should never be a message saying, "We're down right now."

It's hard to tell a three-year old that they can't watch their dinosaur cartoons because Netflix has failed.

Yeah, tell me about it.

STEFAN TILKOV is cofounder and principal consultant at innoQ, a technology consulting company with offices in Germany and Switzerland. Contact him at stefan.tilkov@innoq.com.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.