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

Docker

Charles Anderson

THE SOFTWARE ENGINEERING Radio podcast welcomes two new hosts this year: Josh Long and Sven Johann, whose interview with *Software Architecture for Developers* author Simon Brown is coming soon. Later this year, we'll publish an episode in which I interview all the podcast hosts and some of the editorial staff. This will give our listeners the opportunity to find out why busy software engineers are volunteering their time to this podcast.

In episode 217, host Charles Anderson talks with James Turnbull, a software developer and security specialist who's vice president of services at Docker. Lightweight Docker containers are rapidly becoming a tool for deploying microservice-based architectures, a topic we've covered in several shows and in last issue's column.

Portions of the interview that aren't featured in this column owing to space include networking between containers, how Docker images are built, the DockerHub repository for sharing images, developer use cases for containers, the role of containers in a microservices architecture, and Docker's importance for DevOps. You can download the full episode at www.se-radio.net. —Robert Blumen

What is Docker?

Docker is a container virtualization technology. So, it's like a very lightweight virtual machine [VM]. In addition to building containers, we provide what we call a developer workflow, which is really about helping people build containers and applications inside containers and then share those among their teammates.

What problems does it address?

There are a couple of problems we're looking at specifically. The first one is aimed at the fact that a VM is a fairly large-weight compute resource. Your average VM is a copy of an operating system running on top of a hypervisor running on top of physical hardware, which your application is then on top of. That presents some challenges for speed and performance, and some challenges in an agile sort of environment.

So, we're aiming to solve the problem of producing a more lightweight, more agile compute resource. Docker containers launch in a subsecond, and you can then have a hypervisor that sits directly on top of the operating system. So, you can pack a lot of them onto a physical or virtual machine. You get quite a lot of scalability.

For most people, the most important IT asset they own is the code they're developing, and that code lives on a developer's workstation or laptop or in a dev test environment. It's not really valuable to the company until it actually gets in front of the customer. The process by which it gets in front of a customer, that workflow of dev test, staging, and deployment to production, is one of the most [tension-fraught] in IT.

0740-7459/15/$31.00 © 2015 IEEE

The DevOps movement, for example, emerged from one of the classic stumbling blocks in a lot of organizations. Developers build code and applications and ship them to the operations people, only to discover that the code and applications don't run in production. This is the classic "it works on my machine; it's operations' problem now."

We were aiming to build a lightweight computing technology that helped people put code and applications inside that resource, have them be portable all the way through the dev test, and then be able to be instantiated in production. We made the assumption that what you build and run in dev test looks the same as what you build and run in production.

What are some typical use cases in which a developer or admin might want to use Docker?

We have two really hot use cases right now. The first one is continuous integration and continuous deployment. With Docker being so lightweight, developers can build stacks of Docker containers on their laptops that replicate some production environments—for example, a LAMP stack or a multitier application. They can build and run their application against that stack.

You can then move these containers around—they're very portable. Let's say you have a Jenkins continuous-integration environment. Instead of relying on VMs, we have to spin up a new VM, install all the software, install your application source code, run the tests, and then probably tear it all down again because you may have destroyed the VM as part of the test process.

Let's say it would take 10 minutes to build those VMs. In the Docker world, you can build those VMs or the containers that replace them in a matter of seconds, which means if you've cut 10 minutes out of your build–test run, that's an amazing cost saving. It allows you to get much more bang out of your buck from the continuous-deployment and continuous-testing model.

The other area where we're seeing a lot of interest is what we call high capacity. Traditional VMs have a hypervisor, which probably occupies about 10 to 15 percent of the capacity of a host. We have a lot of customers for whom that 10 to 15 percent is quite an expensive 10 to 15 percent. They want to say, "Okay, let's root that out, replace it with a Docker host, and then we can run a lot more containers." We can run hyperscale numbers of containers on a host container because without a hypervisor, they sit right on top of the operating system and are very, very fast.

IBM released some research last year that suggested that on a per-transaction basis, the average container is about 26 times faster than a VM, which is pretty amazing.

Docker is based on containers, which provide an isolated environment for user-mode code. Some of

SOFTWARE ENGINEERING RADIO

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

RECENT EPISODES

- 219—Jeff Meyerson talks with Apache Kafka project committer Jun Rao about the popular streaming and messaging framework, and the challenges of building a reliable distributed messaging system.
- 220—Robert Blumen is on location with Jon Gifford to learn about using logging to provide insights into programs' run-time behavior, modern logging infrastructure based on a search engine, and logging as a service in the cloud.
- 221—Jez Humble joins host Johannes Thönes to explore the benefits and challenges of implementing continuous delivery, from both a technological and cultural standpoint.

UPCOMING EPISODES

- Sven Johann interviews Simon Brown, author of *Software Architecture for Developers*, about Brown's approach to developing, documenting, and communicating software architecture via a set of simple drawings.
- Joshua Suereth and Matthew Farwell appear with host Tobias Kaatz to discuss software builds, the SBT (a scala build tool), and the guests' new book.
- Stefan Tilkov sits down with Mark Nottingham to learn about the game-changing new version of the venerable HTTP protocol and its impact on Web application development.

our listeners might be familiar with earlier container systems such as Solaris Zones or FreeBSD jails, or even going back to the chroot system call from Unix Version 7. How does the container have its own copy of the file system without duplicating the space between identical containers, especially when you're talking about hyperscale?
One of the other interesting technologies Docker relies on is a concept we call copy-on-write. Many file systems, such as Btrfs, Device Mapper, and AuFS, all support this copy-on-write model, which is what the kernel developers call a union file system. Essentially, what happens is that you build layers of file systems. So, every Docker container is built on what we call an image. The Docker image is like a prebaked file system that contains a very thin layer of libraries and binaries that are required to make your application work, and perhaps your application code and maybe some supporting packages.

For example, you might have a LAMP stack that might have a container that has Apache in it, and libc, and a small number of very thin shims that fake out an operating system. That image is saved in what we call a file system layer.

If I was to then make a change to that image—for example, if I wanted to install another package—I'd say, "I want to have PHP as well." On an Ubuntu system I'd say, "apt-get install PHP," and Docker says, "You want to create a new thing. I'm going to create a new layer on top of our existing layer, and I'm only going to add in the things that I have changed." So, for example, "I'm going to add in the new package, and that's it." That is a layered construction. I end up with a read-only file system with multiple layers. You can think about this layer a bit like a git commit or a version control commit. As a result, I end up with a very lightweight system that only has the things on it that I want.

Docker understands that I can cache things. So it says, "You've already installed PHP. I'm not going to make any changes to the environment; I'm not going to have to write anything. So, I'm going to just reuse that existing layer, and I'll drag that in as the PHP layer." For example, if you're changing your source code, instead of a VM you may be rebuilding the DM. With Docker, you say, "Here's the new commit of my source code. I'm going to add it to my Docker image and maybe that's 10 Kbytes worth of code change." Docker says, "That's the only thing you want to change; therefore, that's the only thing I'll write to the file system." As a result, it's very lightweight, and with the cache, extraordinarily fast to rebuild.

Explain how a process in a container can only see other processes in the container.
Docker relies heavily on two pieces of Linux kernel technology. The first one is called *namespaces*. If you run a new process on the Linux kernel, then you're making a system call to the namespaces: "I want to create a new process." If you want to create a new network interface, you're making a call to the network namespace. The kernel assigns you a namespace that has a process and whatever other resources you want. For example, it might have some access to the network. It might have access to some parts of the file system. It might have access just to memory or CPU.

When we create a Docker container, we're basically making a bunch of calls to the Linux kernel to say, "Can you build me a box? The box should have access to this particular file system, access to CPU and memory, and access to the network, and it should be inside this process namespace." And from inside that process namespace, you can't see any other process namespaces outside it.

The second piece of technology we use is called control groups, or *cgroups*. These are designed around managing the resources available to a container. It allows us to do things such as "This container only gets 128 Mbytes of RAM" or "This container doesn't have access to the network." You can add and drop capabilities as needed, and that makes it fairly powerful to be able to granularly control a container in much the same way you would with the point-and-shoot VM interface to say, "Get this network interface," "Get this CPU core," "Take this bit of memory access to this file system," or "Create a virtual CD-ROM drive." It's a similar level of technology.

Besides files and processes, containers provide an isolated environment for network addresses and ports. For example, I can have Web servers running in multiple containers all using port 80. By default, the container doesn't expose network ports to the outside. However, they can be exposed manually or intentionally, which is a bit like opening up ports on a firewall such as iptables, right?
That's correct. Each container has its own network interface, which is a virtual network interface. You can run processes that use network ports inside those containers. For example, I can run 10 Apache containers inside port 80, inside the container. And then, outside the container, I

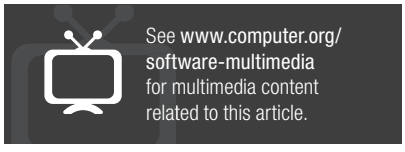
can say, "Expose this port." And if I want to actually expose port 80 to port 80 I can, but obviously I can only do that once, because you can only map one port inside the container to the one port outside the host. By default, Docker chooses a random port and says, "I'm going to do a network address translation between this port 80 inside the container and, say, port 49154 on the host."

For example, I could have multiple Apache processes running on different ports. Then I put a service discovery tool or a load balancer or some sort of proxy in front of it. HAProxy is very commonly used. We can use things such as nginx or service discovery tools such as ZooKeeper, etcd, and Consul that allow you to either proxy the connections or provide a way to say, "I want this application. Query that particular service discovery tool," or "That application belongs to these 10 containers, and you can choose one of these 10 ports," which will connect to the Apache process. So it's a very flexible, scalable model. It's designed to run complicated applications.

Those are going to be aimed at running containers on multiple hosts as well, because so far we've been talking about Docker on a single host, right?

We launched a prototype called *libswarm*, like a swarm of wasps. That tool was designed to prototype how you would get Docker hosts to talk to one another, because currently containers on one Docker host have to use the network to talk to one another. But they should have a back channel of communication. You should essentially be able to have Docker hosts communicating together. We look at this like a way to say, "I'm a Docker host and I run Apache Web services," and another Docker host is saying, "I have a database back end that needs an Apache front end. Can you link one of your containers with one of my containers?" This starts to create some really awesome stories around scalability, autoscaling, and redundancy. And, you start to be able to build really complex applications, which up until now has been very challenging for a lot of organizations.

CHARLES ANDERSON is a software developer with more than 30 years of experience in operating systems, networking, databases, software engineering, and testing. His experience includes embedded systems, Unix kernel internals, client-server desktop applications, Web applications, and Hadoop applications. He also taught many of these subjects at the university level. Contact him at cander@cander.org.



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.