Containers

JAMES BOTTOMLEY AND PAVEL EMELYANOV

James Bottomley is CTO of server virtualization at Parallels where he works on container technology and is Linux kernel maintainer of the SCSI subsystem. He is currently a director on the Board of the Linux Foundation and chair of its Technical Advisory Board. He went to university at Cambridge for both his undergraduate and doctoral degrees after which he joined AT&T Bell Labs to work on distributed lock manager technology for clustering. In 2000 he helped found SteelEye Technology becoming vice president and CTO. He joined Novell in 2008 as a Distinguished Engineer at Novell's SUSE Labs and Parallels in 2011. jbottomley@parallels.com

Pavel Emelyanov is a principal engineer at Parallels working on the company's cloud server projects. He holds a PhD in applied mathematics from the Moscow Institute of Physics and Technology. His speaking experience includes talks about container virtualization at LinuxCon 2009, at the joint memory management, storage and file-system summit in 2011, and about checkpoint-restore on LinuxCon Europe 2012 and Linux Conf AU 2013. xemul@parallels.com

Today, thanks to a variety of converging trends, there is huge interest in container technology, but there is also widespread confusion about just what containers are and how they work. In this article, we cover the history of containers, compare their features to hypervisor-based virtualization, and explain how containers, by virtue of their granular and specific application of virtualization, can provide a superior solution in a variety of situations where traditional virtualization is deployed today.

Since everyone knows what hypervisor-based virtualization is, it would seem that comparisons with hypervisors are the place to begin.

Hypervisors and Containers

A hypervisor, in essence, is an environment virtualized at the hardware level.

In this familiar scenario, the hypervisor kernel, which is effectively a full operating system, called the host operating system, emulates a set of virtual hardware for each guest by trapping the usual operating system hardware access primitives. Since hardware descriptions are well known and well defined, emulating them is quite easy. Plus, in the modern world, CPUs now contain special virtualization instruction extensions for helping virtualize hard-to-emulate things like paging hardware and speeding up common operations. On top of this emulated hardware, another operating system, complete with unmodified kernel (we're ignoring paravirtual operating systems here for the sake of didactic simplicity), is brought up. Over the past decade, remarkable strides have been made in expanding virtualization instructions within CPUs so that most of the operations that hardware-based virtualization requires can be done quite efficiently in spite of the huge overhead of running through two operating systems to get to real hardware.

Containers, on the other hand, began life under the assumption that the operating system itself could be virtualized in such a way that, instead of starting with virtual hardware, one could start instead with virtualizing the operating system kernel API (see Figure 2).

In this view of the world, the separation of the virtual operating systems begins at the init system. Historically, the idea was to match the capabilities of hypervisor-based virtualization (full isolation, running complete operating systems) just using shared operating system virtualization techniques instead.

In simplistic terms, OS virtualization means separating static resources (like memory or network interfaces) into pools, and dynamic resources (like I/O bandwidth or CPU time) into shares that are allotted to the virtual system.

A Comparison of Approaches

The big disadvantage of the container approach is that because you have to share the kernel, you can never bring up two operating systems on the same physical box that are different at the kernel level (like Windows and Linux). However, the great advantage is that, because a single kernel sees everything that goes on inside the multiple containers, resource sharing

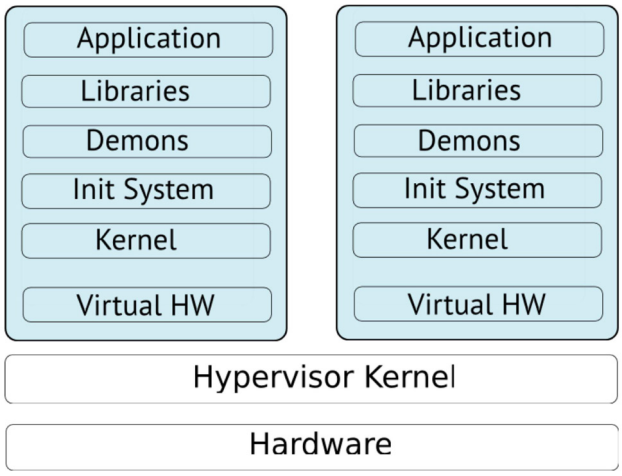

Figure 1: Hypervisor diagram

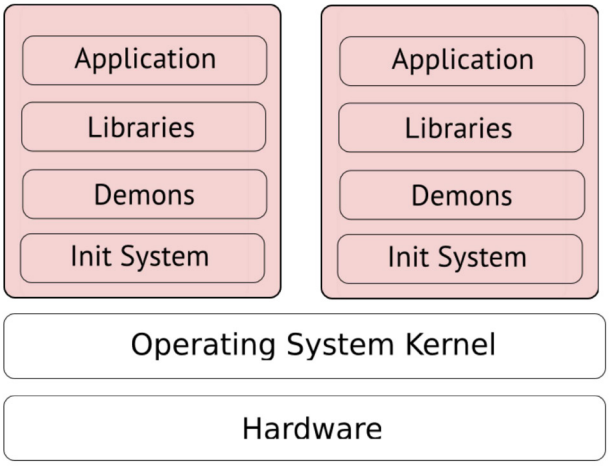

Figure 2: Container diagram

and efficiency is greatly enhanced. Indeed, although the container stack is much thinner than the hypervisor stack by virtue of not having to run two kernels, most of the container improvements in density in fact come from the greater resource efficiency (in particular, sharing the page cache of the single kernel). The big benefit, of course, is that the image of what's running in the container (even when it's a full operating system) is much smaller. This means that containers are much more elastic (faster to start, stop, migrate, and add and remove resources, like memory and CPU) than their hypervisor cousins. In many ways, this makes container technology highly suited to the cloud, where homogeneity is the norm (no running different operating systems on the same physical platform) and where elasticity is supposed to be king.

Another great improvement containers have over hypervisors is that the control systems can operate at the kernel (hence API) level instead of at the hardware level as you have to do with hypervisors. This means, for instance, that the host operating system can simply reach inside any container guest to perform any operation it desires. Conversely, achieving this within a hypervisor usually requires some type of hardware console emulating plus a special driver running inside the guest operating system. To take memory away from a container, you simply tune its memory limit down and the shared kernel will instantly act on the instruction. For a hypervisor, you have to get the cooperation of a guest driver to inflate a memory balloon inside the guest, and then you can remove the memory from within this balloon. Again, this leads to greatly increased elasticity for containers because vertical scaling (the ability of a virtual environment to take over or be scaled back from the system physical resources) is far faster in the container situation than in the hypervisor one.

The History of Containers

In many ways, the initial idea of containers goes back to Multics (the original precursor to UNIX) and the idea of a multi-user time-sharing operating system. In all time-sharing systems, the underlying operating system is supposed to pretend to every user that they're the sole owner of the resources of the machine, and even impose limits and resource sharing such that two users of a time-sharing system should not be able materially to impact one another.

The first real advance was around 1982 with the BSD `chroot()` system call leading to the Jail concept, which was founded in the idea of logically disconnecting the Jail from the rest of the system by isolating its file-system tree such that you could not get back out from the containerized file system into the host (although the host could poke about in the Jailed directory to its heart's content).

In 1999, SWsoft began the first attempts at shared operating system virtualization, culminating with the production release of Virtuozzo containers in 2001. Also in 2001, Solaris released Zones. Both Virtuozzo and Zones were fully isolating container technology based on capabilities and resource controls.

In 2005, an open source version of Virtuozzo (called OpenVZ) was released, and in 2006 an entirely new system called process containers (now CGroups) was developed for the Linux kernel. In 2007, Google saw the value of containers, hired the CGroups developers, and set about entirely containerizing the Googleplex (and making unreleased additions to their container system in the meantime), and in 2008, the first release of LXC (LinuX Containers) based wholly on upstream was made. Although OpenVZ was fully open source, it was never integrated into the Linux mainstream (meaning you always had to apply additional

patches to the Linux kernel to get these container systems), which by 2011 led to the situation in which there were three separate Linux container technologies (OpenVZ, CGroups/namespaces, and the Google enhancements). However, at the fringes of the 2011 Kernel Summit, all the container parties came together for a large meeting, which decided that every technology would integrate upstream, and every out-of-Linux source tree container provider would use it. This meant selecting the best from all the out-of-tree technologies and integrating them upstream. As of writing this article, that entire program is complete except for one missing CGroups addition: the kernel memory accounting system, which is expected to be in Linux by kernel version 3.17.

The VPS Market and the Enterprise

In Web hosting parlance, VPS stands for Virtual Private Server and means a virtual instance, sold cheaply to a customer, inside of which they can run anything. If you've ever bought hosting services, the chances are what you bought was a VPS. Most people buying a VPS tend to think they have bought a hypervisor-based virtual machine, but in more than 50% of the cases the truth is that they've actually bought a container pretending to look like a hypervisor-based virtual machine. The reason is very simple: density. The VPS business is a race to the bottom and very price sensitive (the cheapest VPSes currently go for around $10 US a month) and thus has a very low margin. The ability to pack three times as many virtual container environments on a single physical system is often the difference between profit and loss for hosters, which explains the widespread uptake of containers in this market.

Enterprises, by contrast, took to virtualization as a neat way of repurposing the excess capacity they had within datacenters as a result of mismatches between application requirements and hardware, while freeing then from the usual hardware management tasks. Indeed, this view of virtualization meant that the enterprise was never interested in density (because they could always afford more machines) and, because it built orchestration systems on varied virtual images, the container disadvantage of being unable to run operating systems that didn't share the same kernel on the same physical system looked like a killer disadvantage.

Because of this bifurcation, container technology has been quietly developing for the past decade but completely hidden from the enterprise view (which leads to a lot of misinformation in the enterprise space about what containers can and cannot do). However, in the decade where hypervisors have become the standard way of freeing the enterprise datacenter from hardware dependence, several significant problems like image sprawl (exactly how many different versions of operating systems do you have hidden away in all your running and saved hypervisor images) and the patching problem (how do you identify and add all the security fixes to all the hypervisor images in your entire organization) have lead to significant headaches and expensive tooling to solve hypervisor-image lifecycle management.

Container Security and the Root Problem

One of the fairly ingrained enterprise perceptions is that containers are insecure. This is fed by the LXC technology, which, up until very recently, was not really secure, because the Linux container security mechanisms (agreed upon in 2011) were just being implemented. However, if you think about the requirements for the VPS market, you can see that because hosting providers have to give root access to most VPS systems they sell, coping with hostile root running within a container was a bread-and-butter requirement even back in 2001.

One of the essential tenets of container security is that root (UID 0 in UNIX terms) may not exist within the container, because if it broke out, it would cause enormous damage within the host. This is analogous to the principle of privilege separation in daemon services and functions in a similar fashion. In upstream Linux, the mechanism for achieving this (called the user namespace) was not really functional until 2012 and is today only just being turned on by the Linux distributions, which means that anyone running a distribution based on a kernel older than 3.10 likely doesn't have it enabled and thus cannot benefit from root separation within the container.

Containers in Linux: Namespaces and CGroups

In this section, we delve into the Linux specifics of what we use to implement containers. In essence, though, they are extensions of existing APIs: CGroups are essentially an extension of Resource Limits (POSIX RLIMITs) applied to groups of processes instead of to single processes. Namespaces are likewise sophisticated extensions of the `chroot()` separation system applied to a set of different subsystems. The object of this section is to explain the principles of operation rather than give practical examples (which would be a whole article in its own right).

Please also bear in mind as you read this section that it was written when the 3.15 kernel was released. The information in this section, being very Linux specific, may have changed since then.

CGroups

CGroups can be thought of as resource controllers (or limiters) on particular types of resources. The thing about most CGroups is that the control applies to a group of processes (hence the interior of the container becomes the group) that it's inherited across forks, and the CGroups can actually be set up hierarchically. The current CGroups are:

- blkio—controls block devices
- cpu and cpuacct—controls CPU resources

- cpuset—controls CPU affinity for a group of processes
- devices—controls device visibility, effectively by gating the `mknod()` and `open()` calls within the container
- freezer—allows arbitrary suspend and resume of groups of processes
- hugetlb—controls access to huge pages, something very Linux specific
- memory—currently controls user memory allocation but soon will control both user and kernel memory allocations
- net_cls and net_prio—controls packet classification and prioritization
- perf_event—controls access to performance events

As you can see from the brief descriptions, they're much more extensive than the old RLIMIT controls. With all of these controllers, you can effectively isolate one container from another in such a way that whatever the group of processes within the container do, they cannot have any external influence on a different container (provided they've been configured not to, of course).

Namespaces

Although, simplistically, we've described namespaces as being huge extensions of `chroot()`, in practice, they're much more subtle and sophisticated. In Linux there are six namespaces:

- Network—tags a network interface
- PID—does a subtree from the fork, remapping the visible PID to 1 so that init can work
- UTS—allows specifying new host and NIS names in the kernel
- IPC—separates the system V IPC namespace on a per-container basis
- Mount—allows each container to have a separate file-system root
- User—does a prescribed remapping between UIDs in the host and container

The namespace separation is applied as part of the `clone()` flags and is inherited across forks. The big difference from `chroot()` is that namespaces tag resources and any tagged resources may disappear from the parent namespace altogether (although some namespaces, like PID and user are simply remappings of resources in the parent namespace).

Container security guarantees are provided by the user namespace, which maps UID 0 within the container (the root user and up, including well known UIDs like bin) to unused UIDs in the host, meaning that if the apparent root user in the container ever breaks out of the container, it is completely unprivileged in the host.

Containers as the New Virtualization Paradigm

One of the ironies of container technology is that, although it has spent the last decade trying to look like a denser hypervisor (mostly for the VPS market), it is actually the qualities that set it apart from hypervisors that are starting to make container technology look interesting.

Green Comes to the Enterprise

Although the enterprise still isn't entirely interested in density for its own sake, other considerations besides hardware cost are starting to be felt. In particular, green computing (power reduction) and simply the limits imposed by a datacenter sited in a modern city—the finite capacity of a metropolitan location to supply power and cooling—dictate that some of the original container differentiators now look appealing. After all, although the hosting providers primarily demand density for cost reasons, the same three times density rationale can also be used to justify running three times as many applications for the same power and cooling requirements as a traditional hypervisor and, thus, might just provide the edge to space-constrained datacenters in downtown Manhattan, for example.

Just Enough Virtualization

The cost of the past decade of hypervisor-based virtualization has been that although virtual machine images mostly perform a specific task or run a particular application, most of the management software for hypervisor-based virtualization is concerned with managing the guest operating system stack, which is entirely superfluous to the running application. One of the interesting aspects of containers is that instead of being all or nothing, virtualization can be applied on a per-subsystem basis. In particular, because of the granularity of the virtualization, the amount of sharing between the guest and the host is adjustable on a continuous scale. The promise, therefore, is that container-based virtualization can be applied only to the application, as shown in Figure 3 where a traditional operating system container is shown on the left-hand side and a new pure-application

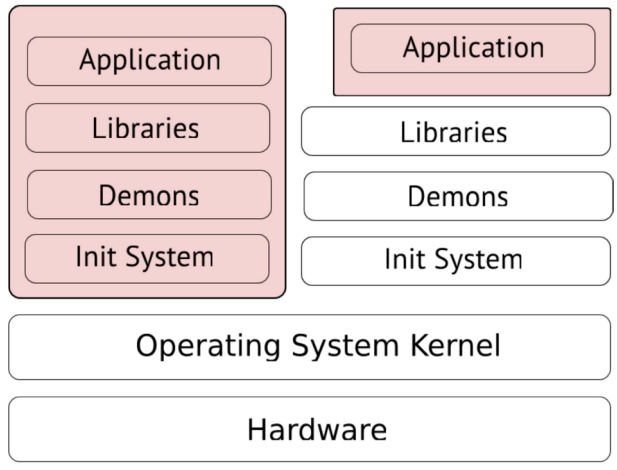

Figure 3: Containerizing just the application

container is shown on the right. If done correctly, this type of application virtualization can make management of the support operating system a property of the host platform instead of being, as it is today with hypervisors, a property of every virtual machine image.

This new "just enough virtualization" world promises to greatly reduce the image sprawl problem by making sure that the virtualized image contains only enough elements to support the application or task itself instead of being a full-fledged operating system image in its own right.

Solving Current Problems with Containers

As an illustration of the way containerization can solve existing problems in a new way, consider the problem of tenancy in the cloud: Standard enterprise applications are designed to serve a single tenant. What this means in practice is that one overall administrator for the enterprise application administers the application for all users. If this application is transferred to the cloud, in its enterprise incarnation, then each consumer (or tenant) wants to designate an administrator who can only administer users belonging to the tenant. The tenancy problem can be solved by running the application inside a virtual machine with one VM per tenant, but it can be solved much more elegantly by adding a small amount of containerization to the application. A simple recipe to take a single tenant application and make it multi-tenant is to fork the application once for each tenant; to each fork, add a new network namespace so that it can have its own IP address, and a new mount namespace so that it can have a private datastore. Because we added no other containerization, each fork of the application shares resources with the host (although we could add additional containerization if this becomes a concern), so the multi-tenant application we have created is now very similar to a fleet of simple single tenant applications. In addition, because containers are migratable, we can even scale this newly created multi-tenant application horizontally using container migration techniques.

Enabling a Containerized Future

The multi-tenant example above shows that there might be a need for even applications to manipulate container properties themselves. Thus, to expand the availability and utility of container technologies a consortium of companies has come together to create a library for manipulating basic container properties. The current C version of this library exists on GitHub (https://github.com/xemul/libct), but it will shortly be combined with a GO-based libcontainer to provide bindings for C, C++, Python, and Go. Although designed around the Linux container API, the library nevertheless has flexibility to be used as a backend to any container system (including Solaris Zones or Parallels Containers for Windows). This would mean, provided the portability works, that the direct benefits of containerizing applications would be exported to platforms beyond Linux.

Conclusions

Hopefully, you now have at least a flavor of what containers are, where they came from, and, most importantly, how their differences from hypervisors are being exploited today to advance virtualization to the next level of usability and manageability. The bottom line is that containers have a new and interesting contribution to make; they've gone from being an expense-reducing curiosity for Web applications to the enterprise mainstream, and they hold the possibility of enabling us to tailor container virtualization to the needs of the application, and thus give applications interesting properties that they haven't been able to possess before.

Resources

The subject of varied uses of containers is very new, so there are few articles to refer to. However, here are some useful Web references on the individual technologies that have been used to create containers on Linux.

Michael Kerrisk of *Linux Weekly News* did a good online seven-part write-up of what namespaces are and how they work: http://lwn.net/Articles/531114/.

Neil Brown as a guest author for *Linux Weekly News* has done a good summary of CGroups: http://lwn.net/Articles/604609/.

This blog post on network namespaces is a useful introduction to using the separated capabilities of namespaces to do interesting things in tiny semi-virtualized environments: http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/.