WHITE PAPER

For CTO's: the no-nonsense way to accelerate your business with containers

February 2017

Contents

Executive overview

Container technology has brought about a step-change in virtualisation technology. Organisations implementing containers see considerable opportunities to improve agility, efficiency, speed, and manageability within their IT environments. Containers promise to improve datacenter efficiency and performance without having to make additional investments in hardware or infrastructure.

This white paper explains the background and timeline to the development of containers, and the most recent technology that has led to the proliferation of containers on the Linux platform. It explains the differences between and advantages and disadvantages of process containers and machine containers. It examines the two main software tools (Docker and LXD) that are used to manipulate containers. Lastly, a glossary at the end of the paper provides a convenient reference point for the technical terms used within the paper.

Traditional hypervisors provide the most common form of virtualisation, and virtual machines running on such hypervisors are pervasive in nearly every datacenter.

Containers offer a new form of virtualisation, providing almost equivalent levels of resource isolation as a traditional hypervisor. However, containers are lower overhead both in terms of lower memory footprint and higher efficiency. This means higher density can be achieved – simply put, you can get more for the same hardware.

Containers come in two varieties: process containers and machine containers. Machine containers act very much like lightweight physical or virtual machines, having their own full operating system image, but sharing the same kernel as the host machine; they share familiar tooling and workflow and are simple to adopt. Process containers are still more lightweight, only containing the binaries or processes for the specific application they run. As such, they require changes in workflow, which may carry associated costs. However, these same changes in workflow provide other advantages, such as the ability to change easily to immutable infrastructure, which itself brings benefits in terms of creating large, manageable, scalable deployments.

Large deployments of containers bring challenges associated with network management and service discovery. If each container carries its own IP address (which is the simple route to avoiding service discovery problems), IP address management and management of overlay networking become significant issues in a large estate. Technologies such as fan networking, popularised by Canonical, can significantly alleviate such issues.

The attraction of containers is not just static efficiency. Containers are also quicker to spin up and take down. This facilitates a more scalable architecture where applications are divided up into microservices where each such service can be developed, deployed, and scaled independently. Such an architecture provides greater agility and business responsiveness, as well as lower total cost of ownership and greater resilience; however, it requires container orchestration software in order to take full advantage.

Containers thus present a new opportunity for the CTO to reduce cost, to increase agility, and to move to a more scalable and resilient architecture. However, CTOs must also recognize that some use cases are better suited to containers than others, that grasping this opportunity requires a recognition of transition costs, and that in some cases this change in infrastructure also brings some challenges. These concepts are more fully explained within the body of this white paper.

The background of containers

THE HISTORICAL CONTEXT

Virtualisation first arrived on the Linux operating system in the form of hypervisors such as Xen and KVM (kernel virtual machines) - referred to in this paper as 'traditional hypervisors'. Each segment of the host machine being known as a 'virtual machine', running its own operating system kernel.

Those who ran applications that were particularly cost sensitive (the first example being consumer web hosting) attempted to squeeze as much out of a given piece of hardware as possible. However, high densities were difficult to achieve with existing forms of virtualisation, especially when the application was small in size compared to the kernel, as much of the system's memory was taken up with multiple copies of kernel – often the same kernel. Hence in such high density applications, machines were instead divided using cruder technologies (for instance 'chroot jails') despite the fact that this provided imperfect workload isolation and carried security implications. In 2001, operating system virtualisation (in the form of Linux vServer) was introduced as a series of kernel patches.

This was an early form of container virtualisation. Rather than running one copy of the kernel per tenant, the kernel itself recognized separate groupings of processes as belonging to different tenants, each sharing the same kernel, but each being isolated from each other. Moreover, borrowing ideas from FreeBSD jails and Solaris zones, formed the basis of Google's Process Containers, which eventually became cgroups and Linux namespaces, the basis of modern Linux containers, with the user-space program 'LXC' was introduced by IBM and Canonical to manage them.

CONTAINERS AND APPLICATIONS TODAY

As containers do not contain their own kernel, they are typically quick to deploy and remove. They can take two forms: process containers (a container containing only a single application), or machine containers (a container with its own user-space operating system, but sharing the kernel of a host). In modern usage, the term 'container' refers not just to the runtime isolation mechanism, but also to the image deployed with it; these two forms of container differ in the type of image deployed.

In the meantime, applications themselves have changed. Rather than single monolithic processes, they are now composed of multiple components. Each component may have many instances running at once in order to scale, and each component needs to communicate with other components. This division of the application into many component elements is often referred to as 'microservices' (see glossary).

The need for rapid deployment, both to scale and to allow swift, simple upgrades to individual components in keeping with the trend towards ‘DevOps’ and agile development methodologies. Both favour the use of containers, as each application is small compared to the operating system (which would thus constitute a significant overhead in traditional hypervisor-based virtualisation) and upgrades can be performed without the overhead of an operating system boot.

These trends have also brought a growing need to track, and to reproducibly create and destroy, the increasing number of containers, and the container management tools address this need. Docker manages process containers on a single host and provides the de-facto standard for packaging files within a container image. Similarly, LXD provides a means of running machine containers on a single host. Technologies such as Docker Swarm and Kubernetes allow the management of containers across multiple hosts. These technologies represent the current status of container technology.

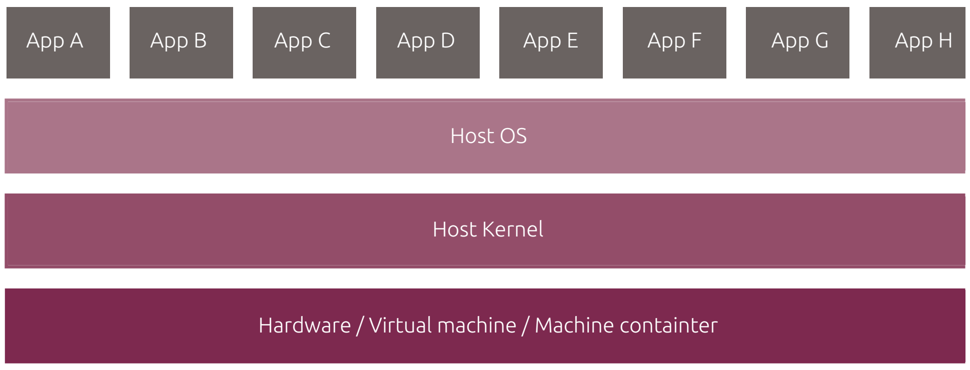
Date	Event
1982	Addition to BSD of chroot system call, permitting process level isolation
2000	Introduction of FreeBSD jails and SWSoft’s (later Parallels) Virtuozzo, file system isolation, memory isolation, network isolation and root privilege isolation
2001	Linux VServer patches bring rudimentary containers to Linux
2004	Sun Microsystems (later Oracle) introduces Solaris Zones, bringing containers to Solaris
2005	Parallels releases OpenVZ kernel patches for Linux, including some of its Virtuozzo container technology
2006	Google develops ‘Process Containers’ for Linux
2007	Control Groups and Namespaces merged into Linux kernel (Process Containers renamed)
2008	LXC userland tooling for Linux Containers released by IBM
2010	Ubuntu booting in an LXC machine container, Canonical assumes stewardship of LXC
2013	Docker released
2014	LXC 1.0 (stable version with long term support) released
2015	LXD released by Canonical
2016	Canonical’s Distribution of Kubernetes

WHAT EXACTLY IS A CONTAINER?

Containers are a technology that allows the user to divide up a machine so that it can run more than one application (in the case of process containers) or operating system instance (in the case of machine containers) on the same kernel and hardware, and in so doing maintain isolation between these workloads.

Before containers came to prime-time, two other techniques were used: multitasking and traditional hypervisor-based virtualisation.

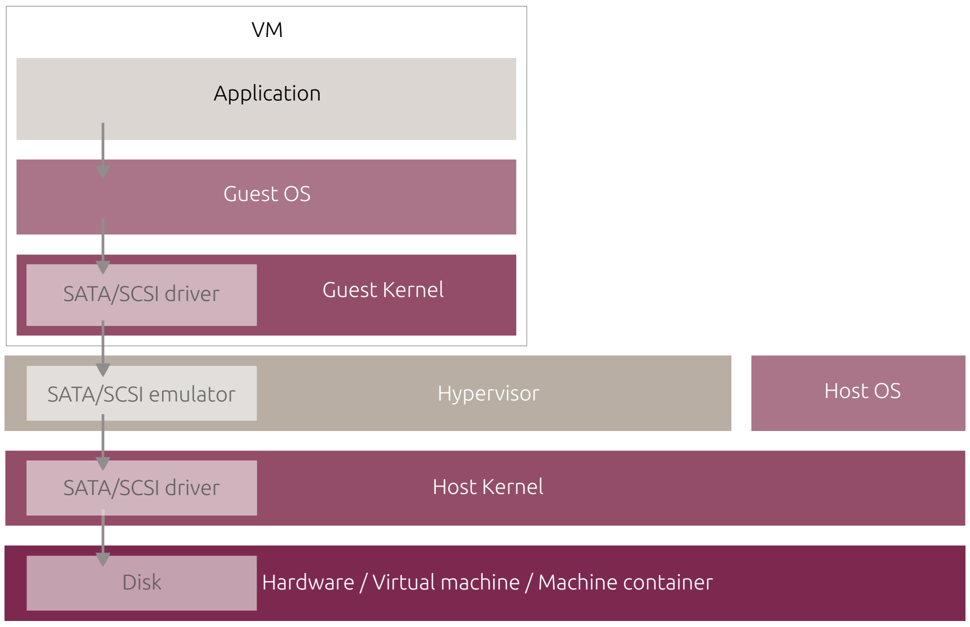
Multitasking allows multiple applications to run on the same operating system, kernel, and hardware; however, it provides little isolation between different applications. For instance a runaway application might exhaust memory, I/O capability or disk space. A malicious or buggy application might provide access to the operating system and hence the data of every application.



...
Without virtualisation or containers

Traditional hypervisor-based virtualisation allows guest kernels to run on top of (or alongside) a host kernel. Each of these guest kernels runs its own operating system instance. These traditional hypervisors provide optimum isolation as well as the ability to run a wide range of operating systems and kernels simultaneously on the same hardware. However, they come with a number of disadvantages:

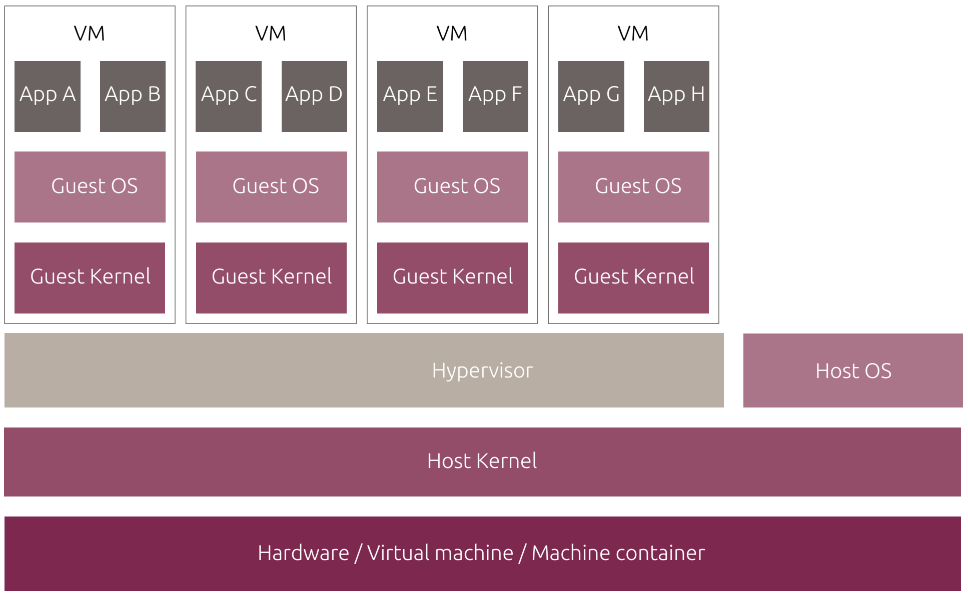
1. Each kernel and operating system takes a while to boot.
2. Each kernel takes up its own permanent memory footprint and carries a CPU overhead, so the overhead of virtualisation is quite large.
3. I/O is less efficient. In order for the guest application to perform I/O, it needs first to call the guest kernel, which makes a request to what it believes is the hardware. Which is in turn emulated by the hypervisor, and passed to the host operating system, and finally to the hardware. The response is then passed the same circuitous route; although paravirtualised drivers have been introduced to remove the emulation overhead, two kernels are still involved, and thus there is still performance degradation, both in terms of overhead and latency.



...
Hypervisor virtualisation overhead of disk write

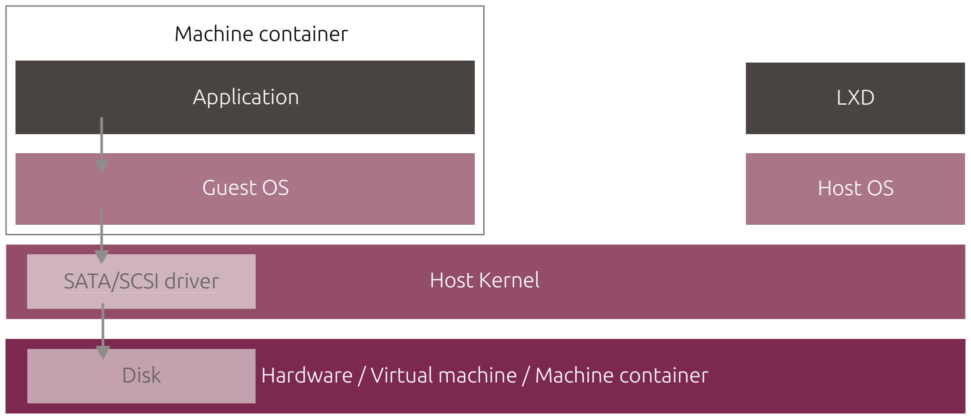
4. Resources are not allocated on a fine-grained basis. As a simple example, each virtual machine has a memory size specified on creation, so memory left idle by one virtual machine is not in general available to another. Technologies such as 'ballooning' can alleviate the problem, resource allocation is necessarily less efficient.

5. The maintenance load of keeping up to date one kernel per virtual machine (as is necessary under traditional hypervisor-based virtualisation) is significantly greater than one kernel per host (as is the case under container-based virtualisation), and the resultant downtime with traditional hypervisor-based virtualisation is also correspondingly greater.



...
Hypervisor virtualisation

Containers to some extent provide the best of both worlds. A single kernel handles logically separate instances of applications (process containers) or operating systems (machine containers). I/O thus passes directly from the application to the host kernel, and to the hardware, and therefore performance is the same as with native applications, and latency is minimized.



Machine containers overhead of disk write

The containers provide isolation that is almost as good as a traditional hypervisor; in fact the isolation is more flexible as containers can (for instance) share some resources but not others. Boot is faster as there is no kernel to start up, and (in the case of process containers) no operating system to start – in fact, even machine containers tend to carry lightweight operating systems with sub-second boot times. And resource allocation via cgroups is fine-grained, being handled by the host kernel, allowing the effective per-container quality of service (QoS) metrics and additional efficiency driven by more flexibility in job scheduling. For instance, in Google's Borg container orchestration system, long running compute-intensive batch jobs are typically run in containers alongside more latency-sensitive user-facing applications.

The latter typically reserve more resources than they need on a continuous basis to satisfy the latency requirements in the event of spikes in load or diminution of availability of some cluster members. The batch jobs are able to make use of this unused capacity, subject to preemption by the user-facing applications. cgroups also provides accurate measurement of consumption of these resources, allowing for development of tools such as Google's cAdvisor as well as intelligent autoscaling.

However, there are disadvantages of containers compared to traditional hypervisor-based virtualisation that include:

- Guests are limited to those that can use the same kernel: you cannot directly run a Windows OS within a Linux container

- There is arguably additional isolation provided by a traditional hypervisor that is not available to containers, meaning the 'noisy neighbour' problem is potentially more significant on containers than under a traditional hypervisor. However, it should be noted that this difference has been minimized in recent years by advances in kernel support for containers. For example, there are resources like level 3 processor cache and memory bandwidth which neither virtualisation technology can arbitrate. Additional isolation provided by a traditional hypervisor can be argued to make traditional hypervisor-based virtualisation more secure than containers, though this is in essence a question of whether the traditional hypervisor's defences are better than the kernel's defences. Fourthly, there are disadvantages of running a single kernel – an 'all eggs in one basket' argument. If that kernel crashes or is exploited, perhaps due to a vulnerability within an application which itself is insecure, the whole machine is compromised; and upgrading that kernel is thus arguably more problematic (as more depends on it) than upgrading kernels in a traditional hypervisor based environment

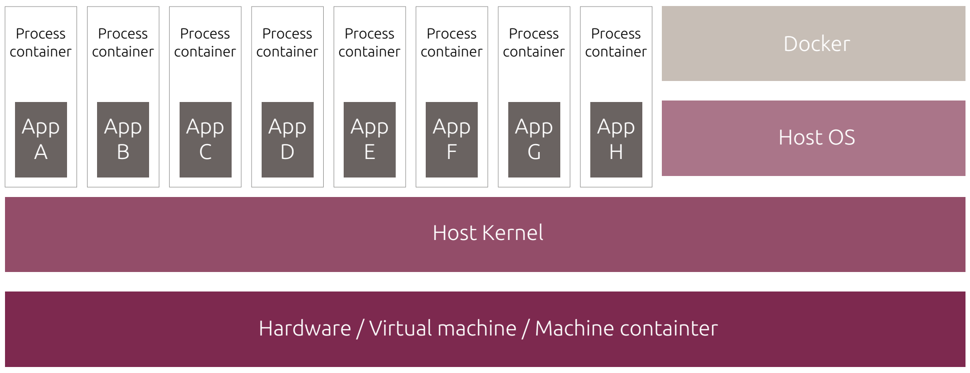
- Techniques such as live migration of containers are in their infancy compared to the equivalent on traditional hypervisors

It is worth noting that virtualisation techniques can be nested. For instance, traditional hypervisors can be run inside hypervisors or containers, and containers can be run inside traditional hypervisors or containers. This, at the expense of some complexity, provides further flexibility in the resource isolation and performance trade-off.

PROCESS VS. MACHINE CONTAINERS

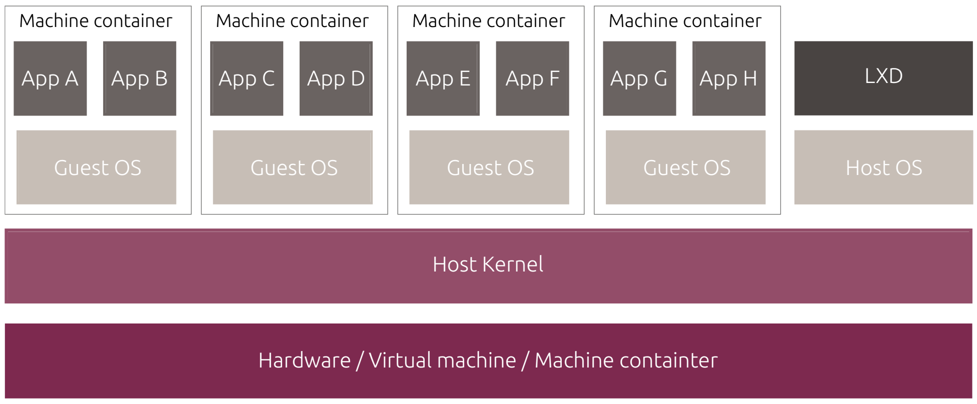
We have briefly distinguished above between process containers and machine containers, saying that a process container contains only a single application, but a machine container contains one or more applications, and its own operating system. In either case, they share a kernel with the host. To drill down a little more into the differences, we need to understand a little more about how applications work in a Linux environment, and what exactly we mean by 'Operating System'.

In a process container environment such as Docker, containers are designed to be very small. They contain only the binaries for a single application. Often these are statically linked, or contain a minimal subset of libraries necessary for that application alone. For instance, they need not contain a shell, or a traditional 'init' process. The disk space required for a process container can be very small indeed (perhaps as little as a couple of megabytes). Technologies such as Docker are often able to 'compose' such containers (overlay one image upon another) so an application can be built upon a base template. Read-write storage is normally separated from the image. As a result, these tend to be ideal for building small immutable services. Of course a real world deployment may contain tens or hundreds of such services.



Process containers

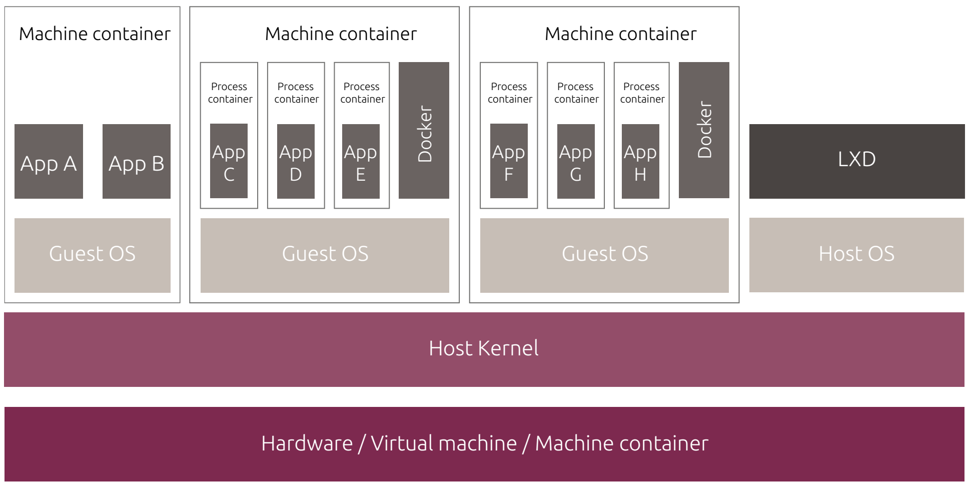
A machine container environment, such as one provided by LXD, looks far more similar to a virtual machine. The container will appear to have its own disk image, often containing a cut down version of an operating system. It will have its own init process, and may run a limited number of daemons. Normally it would also contain a shell. Programs are installed in the manner that the guest operating system would normally expect (for instance using 'apt-get' on an Ubuntu system). LXD thus functions similarly to a traditional hypervisor. The containers are normally stateful, i.e. they provide mutable configuration, though copy-on-write technologies allow them to be reset easily to their initial configuration. Some sources refer to machine containers as 'system containers'.



...
Machine contaiers

We can see from the above that we have used 'operating system' to mean the user-space programs that surround the kernel that do not form part of the application itself. The fact that machine containers each have their own 'operating system' does not mean they each have a full running copy of Linux or their own kernel. Rather they run a few lightweight daemons and have a number of files necessary to provide a separate 'OS within an OS'.

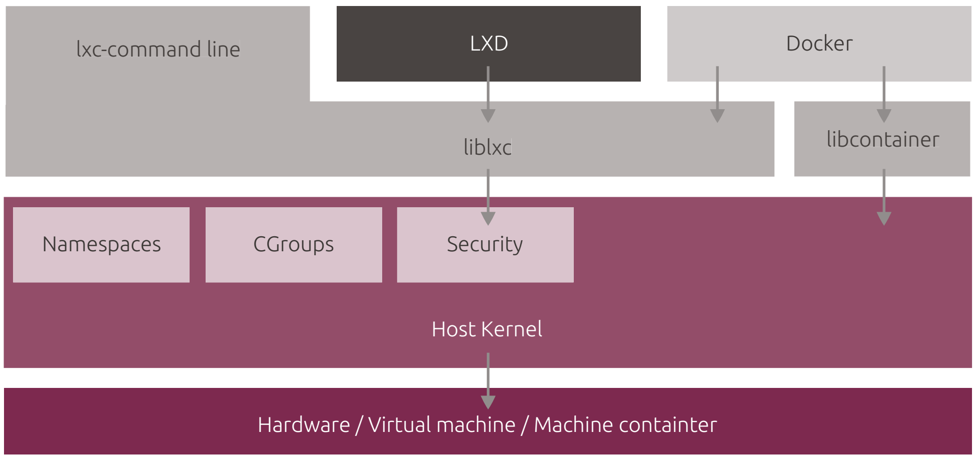
There is a third route for deployment of containers. This is to utilize the ability of containers to nest, and to run process containers within machine containers – for instance to run Docker containers within LXD containers. As containers are so lightweight, this mixed container environment is an eminently practical proposition, and provides an excellent way to manage Docker hosts.



...
Mixed containers

Containers on Linux

The terminology surrounding containers on Linux can be confusing, in part because the names of various components have changed, some projects have morphed into other projects, and occasionally the same name has been used for more than one component. This section sets out the current state of play.



Containers on Linux and LXC

At the bottom of the stack is the Linux kernel. Perhaps surprisingly, the Linux kernel does not itself have a concept of 'containers' per-se; rather the functionality of containers is provided by three kernel concepts: cgroups (control groups), Linux namespaces, and the kernel security infrastructure.

Control groups limit and account for different types of resource usage (CPU, memory, disk I/O, network I/O and so forth) across a collection of different processes; they also provide prioritization of resource usage, and control via checkpointing.

Linux namespaces provide ways to segregate various types of Linux resources into different groups, for instance network namespaces permit different groups of processes to have completely independent views of the Linux networking stack. The other resource types that can be segregated include the process ID space, the mounted filesystems, the filesystem attributes, the IPC space, and the System V semaphore space. If all spaces are segregated, the segregated processes have no operating system attributes in common with those that launched them.

So a container in kernel terms is a tree of processes that is segregated from the tree of processes that launched it, and normally that segregation applies to all the resources.

But, segregation is not sufficient for isolation. To fully isolate a container it must not only be unable to 'see' into other containers, but it must also have its resource usage controlled. For instance, it should not be able to hog memory or I/O bandwidth that other containers may need. To achieve this, the Linux kernel uses cgroups to protect one container from resource starvation or contention caused by another container (whether that originates from a runaway process or a denial of service attack), and thereby provide QoS guarantees.

Lastly, the kernel provides a level of security to containers via Apparmor, SELinux, kernel capabilities and seccomp. These prevent, amongst other things, a process running as root in a container having full access to the system (for instance to hardware), and aid in ensuring processes cannot escape the container in which they run. Containers thus offer two complementary forms of access control. Firstly, discretionary access control (DAC) mediates access to resources based on user-applied policies, so that individual containers cannot interfere with each other, and can be run by non-root users securely. Secondly mandatory access control (MAC) ensures that neither the container code itself nor the code run within the containers has a greater degree of access than the process itself requires, so the privileges granted to rogue or compromised process are minimised.

Above the kernel lies the user-space toolset for manipulating containers, the most popular of which is LXC. This itself is divided into a number of parts: a library called liblxc which does all the heavy lifting, language bindings for that library, a set of command line tools to manipulate containers at a low level, and a set of standard container templates.

It is possible to use LXC containers directly, but they are unlikely to have much practical application as the interface is very low level. Hence most users abstract them further, using programs such as Docker to build process containers, or LXD to build machine containers.

Docker and process containers

Docker is an open-source project that permits the user to automate deployment of applications inside process containers. It permits a small subset of a filing system (just those files necessary for an application to run) to be built into a template, and then allows this to be deployed, repeatedly, into a lightweight container. As such the application is packaged to be separate from the operating system on which it is run. By using only static libraries (or libraries built into the container image), a container built on one version of Ubuntu can run on a host running another; a container built on RHEL (for instance) could even be run on Ubuntu.

Docker follows the principle of immutable infrastructure. When launched, each container looks exactly like its template, so if a container is restarted all changes previously made are lost. This ensures that a given container launches the same each time, and avoids configuration drift where updates to a template and a running machine get out of sync. The recommended way to update a container is simply to destroy it and restart it with a new image. Of course, it is necessary to store state somewhere – for instance a database needs to store its data somewhere. Docker provides 'volumes' for this purpose. Volumes are external containers that have data and no code, which are bound to the template at runtime to create the running containerized application. This move to immutable infrastructure and configuration can represent a significant change of working practices (and thus an obstacle to adoption) but ultimately is a powerful weapon in simplifying management of large container estates, particularly when combined with container orchestration.

Docker uses a plain text configuration file (called a Dockerfile) in order to describe what goes into a container, and permits composing of existing templates and new files to produce new templates. In this way existing templates can be built upon and modified. Underlying this (and the immutability described above) is Docker's use of copy-on-write filesystems. Cut down versions of various distributions (including Ubuntu, Centos, and CoreOS) are provided as base templates, but need not necessarily be used. The built templates, in the form of images, can be uploaded and shared, and thus can be downloaded from a public or private image repository.

Docker has an unusual way of dealing with networking. Each container is not a full-blown independent operating system image, and thus does not necessarily have its own IP address. Rather it runs one or more processes, each of which typically will be listening on one port. The Dockerfile can therefore describe which of these ports need to be ‘exposed’, which in practice means mapped by port translation to a port on the host. For system administrators, this change in technique may require significant acclimatisation. Newer versions of Docker permit containers to have their own IP addresses, and have network interfaces mapped to bridges or overlay networks.

Docker previously used LXC to build its containers, but now by default uses libcontainer, which provides similar functionality.

Docker has a significant ecosystem of other tools into which it is integrated. For instance, it forms the basis of container management in Kubernetes and is integrated into the Cloud Foundry PaaS. It also has integrations with conventional configuration management tools such as Ansible, CFEngine, Chef, Puppet, and Salt, and with various cloud platforms such as AWS, GCE, Azure and OpenStack.

Alternatives to Docker exist, primarily rkt (pronounced ‘rocket’), an open-source rival to Docker whose development is led by CoreOS; it provides broadly similar functionality.

LXD and machine containers

LXD is an open-source tool that provides a means to manage machine containers. These act far more like virtual machines on a traditional hypervisor than the process containers managed by Docker, which has led to LXD being described by Canonical as a “pure container hypervisor”.

LXD is made up of three components. Firstly, a system-wide daemon (itself called ‘lxd’) performs all the heavy lifting. It provides a REST API for local usage, which can also be exported over the network if desired. This daemon communicates with liblxc to create the containers themselves. Secondly a command line tool (called ‘lxc’, but used to manipulate LXD containers not LXC containers) communicates with the daemon, either on the same host or over the network, via the REST API. Thirdly, a plugin to OpenStack (Nova LXD) allows OpenStack to use LXD as a hypervisor, so that OpenStack can create instances on LXD in the same way that it would normally create virtual machines running on a traditional hypervisor such as KVM.

Security has been a key principle of LXD from the design stage. It creates unprivileged containers by default (meaning non-root users can launch containers). Resource constraints for containers were built in from the start rather than as an afterthought. It thus provides a simple way of creating machine containers with excellent isolation from the rest of the system.

By providing operation over the network as well as locally, containers can be created on remote systems through the same command line prompt as is used to create containers on the developer’s own laptop. This allows for simple scalability and management of containers over multiple hosts.

LXD is designed to be an intuitive system, having a clean API and command line. It is thus easy to get going, and will be familiar to those who have come from a virtual machine environment.

LXD containers are image-based, meaning that each container has a full filing system, rather than a composition of templates as with process containers. This means existing trusted image sources can be used, and one can be share those image repositories used for traditional hypervisor-based virtual machines. This makes migration between hypervisor-based environments easy.

LXD also provides support for live-migration, so that if a host node needs maintenance, the container can be migrated elsewhere to ensure continuity of service. Again, in this way it is similar to traditional hypervisor-based virtual machines. Various other features familiar to traditional hypervisor users are also available, such as snapshots and device passthrough.

As LXD containers are fully isolated containers, they have networking that appears very similar to virtual machine based networking. Each container has its own IP address (or IP addresses), and the container's virtual Ethernet device can appear on a bridge in the host. This makes networking no different to that of virtual machines on a traditional hypervisor, and avoids the complexity of port mapping and exposure. It does however mean that IP address management becomes more complex, and that it is necessary to protect (via packet filters or otherwise) machine containers in the same manner that one would need to protect virtual machines.

As set out above, it is possible to nest containers within machine containers, so one can run Docker within a LXD container, or even LXD containers within a LXD container.

Containers and the need for orchestration software

The need for business agility has led to commercial pressure for more frequent deployment of software. In order to support this, new software development techniques (broadly classed as 'agile') and new operational cultures (such as 'DevOps') have taken hold, allowing for increased frequency of deployment changes.

In order to support such rapid change cycles, applications increasingly tend to be built from existing components, be they re-using in-house technology or (more commonly) utilizing open-source elements. Rather than a monolithic application, a modern application consists of multiple components, many being open source elements running unchanged bar configuration (for instance databases, web-servers, message queues and so forth), with a smaller number being written in-house (for instance elements of business logic). The logical conclusion of this is trend is a situation in which the application is composed entirely of microservices, small independently deployable services communicating locally over a network.

Containers provide an ideal vehicle for such components due to their low overhead and speed of deployment. They also permit efficient horizontal scaling by deployment of multiple identical containers of the relevant component each sharing the same image. Modern applications thus might be built from hundreds or even thousands of containers, potentially with complex interdependencies. How can such containers be deployed in a reliable and reproducible manner? If a given container ceases to function, how can it automatically be replaced? And if an application needs to scale, how can the correct component to scale be identified and have its array of containers expanded? These are issues addressed by container orchestration software.

How Canonical helps

Canonical's Ubuntu Linux distribution allows organisations direct access to all significant Linux container technology. Ubuntu brings variety, velocity and quality: variety meaning a wide selection of container technology from which your organisation can select the most suitable choice; velocity meaning timeliness of delivery with a release cadence that ensures the most up-to-date versions of container software are available to you through our predictable release cadence; and quality meaning a keen focus on usability, compatibility and interoperability.

In conjunction with Google, Canonical has released its own distribution of Kubernetes - The Canonical Distribution of Kubernetes (CDK). CDK provides a 'pure K8s' experience, tested across a wide range of clouds and integrated with modern metrics and monitoring. Further, CDK works across all major public clouds and private infrastructure, enabling teams to operate Kubernetes clusters on demand, anywhere.

Canonical is helping many companies to get the most out of containers. From creating distributed applications and Microservices for Platform as a Service (PaaS) environments, Batch and ETL (extract, transform, load) jobs within the financial services industry, and improving DevOps efficiency through continuous integration and deployment.

Next steps

Containers offer a new form of virtualisation, with lower overhead than traditional hypervisors both in terms of lower memory footprint and higher efficiency. This allows organisations to achieve higher density, and run the same compute load for less money. Machine containers provide a simple means to garner this cost advantage for Linux-on-Linux workloads without application redesign, whereas process containers allow additional opportunity to increase agility and to move to a more scalable and resilient architecture.

To discover how Canonical can help you take advantage of containers, we invite you to **test-drive the Canonical Distribution of Kubernetes** and LXD containers using Conjure-up. This pure upstream distribution of Kubernetes is designed to be easily deployable to public clouds, on-premise, bare metal, and developer laptops. During the installation, conjure-up will ask you what cloud you want to deploy on and prompt you for the proper credentials. If you're deploying to local containers (LXD) **see these instructions** for localhost-specific considerations.

For production grade deployments and cluster lifecycle management it is recommended to **read the full Canonical Distribution of Kubernetes documentation**.

Home page: **jujucharms.com/canonical-kubernetes/**
Source code: **github.com/juju-solutions/bundle-canonical-kubernetes**

If you want to learn more about how Canonical can help to develop and deploy your container strategy, please call +1 781 761 9427 (Americas), +44 207 093 5161 (Rest of World) or **contact us online**.

Glossary

Agile Software Development	A set of concepts, practices and principles for the development of software under which both requirements and the software that meets them evolve during the development life-cycle by processes of collaboration, as opposed to being defined at milestones within it.
Ansible	An open-source configuration management utility primarily developed by Red Hat.
AppArmor	A Linux kernel security module allowing restriction of the capabilities of a process based on a profile that is distributed with the application containing the process.
Application	A computer program or set of computer programs designed to perform a particular activity or set of activities.
Application container	Occasionally used as a synonym for ‘process container’.
ARP	Address Resolution Protocol, a protocol translating IP addresses (layer 3 addresses) into MAC addresses (layer 2 address).
AWS (or “Amazon Web Services”)	A suite of cloud computing services offered by Amazon, including EC2 (or “Elastic Cloud Compute”), an IaaS service.
Azure	A suite of cloud computing services offered by Microsoft.
Ballooning	A technique within hypervisor-based virtualisation where memory pages not used by the guest operating system can be returned to the host (or other guests).
Borg	A proprietary container orchestration system produced and used internally by Google, which is the predecessor to Kubernetes.
cAdvisor	An open-source tool produced by Google that allows users to introspect the performance and resource usage of their containers.
CentOS	An open source community-supported Linux distribution aiming for functional compatibility with RHEL.
Ceph	An open-source a distributed block store, object store and file system primarily developed by Red Hat.
cgroups (or Control Groups)	A facility provided by the Linux Kernel that allows limitation and accounting of usage of various resources (including CPU, disk I/O and network I/O) across groups of different processes, and prioritization between these groups of processes.
Chef	An open-source configuration management utility primarily developed by the eponymous company named Chef.
chroot	A system call in UNIX operating systems that changes the root directory to an alternate point in the filesystem hierarchy, thus providing a view of a subset of the filesystem as opposed to the whole.
Cloud Foundry	An open-source Platform-as-a-Service software suite overseen by the Cloud Foundry Foundation. The new scheduler (Diego) permits general container workloads. An enterprise edition is distributed by Pivotal and others.

Compose (as " Docker Compose")	An open-source tool for defining and running multi-container Docker applications.
Container	A virtualisation method whereby the host operating system permits multiple isolated guest environments ("containers") that share the same kernel (in vthe case of system containers) or operating system (in the case of application containers).
Copy-on-write	A technology whereby a copy of resource is made in a delayed manner. Rather than copying the resource immediately, the resource is marked as copied, and when either the copy or (on occasion) the original is written to, a copy of the written area is made, meaning that only the differences are recorded.
CoreOS	An open source lightweight Linux distribution optimised for container usage distributed by the eponymous company CoreOS.
Device passthrough	A technique allowing virtual machine guests direct access to hardware that would otherwise be managed by the host.
DevOps	A group of concepts and practices that emphasise integration of and collaboration between development and operation of IT systems, often incorporating practices such as continuous delivery, integration and testing together with automated deployment.
Diego	A scheduler for Cloud Foundry that permits general container workloads.
Discretionary access control (DAC)	A form of access control where each user can determine the access to particular resources owned by that user.
DNS (or "Domain Name System")	A directory system and the associated network protocol whereby the client can look up an address and be returned resultant entries in the directory. Most commonly used for translating a domain name into an IP address, it can also be used for service discovery.
Docker	An open-source project that permits the user to package and automate deployment of applications inside application containers.
Dockerfile	A text file that describes to Docker what goes into a container, permitting addition or modification of files from a base container and exposure of ports.
EBS (or "Elastic Block Store")	A block storage product forming part of Amazon Web Services.
etcd	An open-source distributed key-value store that provides shared configuration and service discovery, primarily developed by CoreOS.
Fan networking	A technique to provided a routed overlay network that provides simple IP address management and hence supports high container densities.
FreeBSD jail	An early form of inter-process isolation introduced in the FreeBSD operating system.
GCE (or "Google Compute Engine")	A suite of cloud computing services offered by Google.
Guest	Virtual environments run within a host. For instance guest virtual machines might be launched by a hypervisor running within a host physical (or virtual) machine; a guest operating system might run within a system container launched within a host operating system.

Host	The environment in which guests are run. For instance a host machine is a physical or virtual machine in which a hypervisor is run in order to launch guest virtual machines; a host operating system might launch system containers to run guest containers.
Hypervisor	Software that permits the creation of virtual machines within a physical machine. Occasionally used to describe only Type I or Type II hypervisors such as Xen or KVM where each virtual machine is required to have its own kernel; in this paper we use the term 'traditional hypervisor' for this meaning.
IaaS (or "Infrastructure as a Service")	A form of cloud computing service that offers the user the ability to purchase computing infrastructure (in the form of CPU, network and storage resources) on a subscription basis. The resources often take the form of virtual machines.
Isolation	Techniques employed to prevent visibility of or interference in one system by another.
Juju	An open source application and service modelling tool developed by Canonical that permits rapid modelling, configuration, and deployment of applications into the cloud.
Kernel	A central component of an operating system that manages the boot of the operating system and low-level functions such as access to hardware. The operating system normally contains in addition to the kernel a large number of system programs, some of which are launched by the kernel on start-up.
Kubernetes (or referred to as K8)	An open-source container cluster manager that allows the user to automate the deployment, scaling and operation of application containers.
KVM (or Kernel Virtual Machine)	An open-source hypervisor built into the Linux kernel.
libcontainer	A library used by Docker and others that provides similar functionality to liblxc and is used for low-level container management.
liblxc	A library forming part of LXC that acts as the interface to the Linux kernel to manage containers at a low level.
LXC	A low-level set of user-space tools for manipulating containers, formed from the liblxc library, bindings for that library, a set of command line tools that call the library, and a set of container templates.
LXD	An open-source tool that provides a means to manage system containers, primarily developed by Canonical.
MAC address	Media Access Control address, a unique identifier that is assigned to a hardware or virtual interface at layer 2 of the OSI stack.
Machine container	A form of container that shares the same kernel as the host, but contains an independent operating system image (save for the kernel). A system container thus acts similarly to a virtual machine in a hypervisor environment.
Mandatory Access Control (MAC)	A form of access control where a system-wide policy dictates which users and/or processes are permitted to access which resources in which manner, where such policy cannot be overridden by individual users.

Microservices	A means of deploying software systems and applications from a number of small self-contained services ("microservices") each of which performs an independent function with a well defined API and that can be composed to form a larger application.
Multitasking	The ability of an operating system to execute more than one process at once either by using more than one central processor unit, by time-slicing between processes, or by a combination of the two methods.
Namespaces	A facility of the Linux kernel that allows segregation of resources into different groups, so that different processes can have entirely separate views of the relevant set of resources as managed by the operating system.
NAT (or "Network Address Translation")	A method for mapping one range of IP address space and ports to another.
Nova	An open-source component of OpenStack that deals with the orchestration of compute resources.
Nova LXD	A plug-in to OpenStack that permits Nova to use LXD to create instances of virtual machines as if it were dealing with a conventional hypervisor.
OpenStack	An open-source suite of software that allows the user to create their own cloud platform, predominantly aimed at IaaS private clouds.
OpenVZ	An open-source version of the some of the container technology within Virtuozzo.
Operating System	A set of system software that manages the hardware and software resources of a computer, providing a common set of services for applications running on top of it. In this white paper we distinguish between operating system software (including system applications, system libraries and so forth) and the kernel.
PaaS (or "Platform as a Service")	A form of cloud computing service that offers a software platform (i.e. a combination of development environment, software components and APIs) that permits customers to deploy applications onto a common base. The PaaS service is either hosted by the PaaS vendor within a cloud platform or is available as software to be
Paravirtualisation	installed by the customer on its own compute infrastructure.
Process container	A virtualisation technique where the virtual environment is similar to a physical environment, but not identical. The principle difference is that the guest operating system does not access virtual hardware through emulation of that hardware by the host or the hypervisor, but rather through a special software interface. By extension, a paravirtualised driver is a driver for an operating system running as a hypervisor guest that uses a similar technique to achieve enhanced I/O speeds.
Process Containers (capitalised)	A form of container which shares the same operating system as the host. The container image is thus typically very small, holding only the binaries necessary to run a single application. See also "Process Containers (capitalised)".
Puppet	An early container initiative by Google that became the basis for cgroups and Linux Namespaces. See also "Process container" (above).
QoS (or "Quality of Service")	Measures of performance of various resources (particularly I/O) and levels those measures are meant to meet.

RHEL	Red Hat Enterprise Linux, a commercial Linux distribution sold by Red Hat.
rkt	An open-source project (pronounced “rocket”) similar to Docker that permits the user to package and automate deployment of applications inside application containers.
Salt	An open-source configuration management utility primarily developed by SaltStack.
seccomp	A facility within the Linux kernel for application sandboxing whereby a process can make a one way transition into a secure state in which the system calls it can make are restricted.
SELinux	A Linux kernel security module and associated user-space tools permitting mandatory access control policies that restrict applications’ access to various resources managed by the kernel.
Service discovery	Techniques by which clients of a particular network service can locate instances of that service that are available to it to use.
Swarm	A native clustering tool for Docker that transforms a collection of Docker hosts into a larger single Docker host.
SWSoft	An early innovator in the container space, later to become part of Parallels Inc.
System container	An alternative name for ‘Machine container’.
Traditional hypervisor	A Type I or Type II hypervisor such as Xen or KVM where each virtual machine is required to have its own kernel, as opposed to (for instance) LXD which provides hypervisor functionality without the necessity for a virtual machine to have its own kernel.
Ubuntu	An open source Debian-based Linux distribution published by Canonical, who offer commercial support.
Virtual Machine (or VM)	A simulated physical machine run within an actual physical machine that behaves like a separate self-contained physical machine. The simulation may be performed by an emulator or (for far better performance) by a hypervisor.
Virtualisation	Creation of a virtual (as opposed to physical) version of something, most often creation of a virtual machine within a physical machine that behaves like a virtual machine, or another form of virtual environment capable of running software within another environment.
Virtuozzo	An early (and current) commercial container offering produced by SWSoft, which later became part of Parallels. The company producing Virtuozzo is now itself called Virtuozzo. Many of the kernel components were released as OpenVZ and became part of the Linux kernel.
VServer	An early form of interprocess isolation available as a set of patches to the Linux operating system that were aimed at hosting providers.
Xen	An open-source hypervisor that uses a microkernel design, often but not exclusively using Linux to manage it.
Zones	A form of container available on Solaris.

CANONICAL