MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Virtualization using Docker Platform

MASTER'S THESIS

Vladimír Jurenka

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vladimír Jurenka

Advisor: Filip Nguyen, RNDr.

Acknowledgement

I would like to thank my supervisor RNDr. Filip Nguyen, for his advices, encouragement and insightful feedback.

Abstract

The aim of the diploma work is to provide an overview of virtualization methods, while focusing on Docker. This new virtualization platform, which greatly differs from the traditional, virtual machine based approach to virtualization, has sparked massive interest in Linux containers. As a result, many Docker based projects have emerged ranging from simple command line tools to entire operating systems. To gain a deeper understanding of Docker's internal mechanisms, the practical part of this thesis demonstrates working with Docker's Remote API and further enhances Docker with a new command.

Keywords

Docker, virtualization, virtual machines, containers, vmware, virtualbox, linuc, lxc, namespaces, cgroups

Contents

1 Introduction

"Virtualization refers to the creation of virtual machines which have an independent Operating Systems but the execution of software running on the virtual machine is separated from the underlying hardware resources. Also it is possible that multiple virtual machines can share the same underlying hardware."[1] It has been around for many years, throughout which the range of offered features widened and its performance gradually increased. However, there is still some space for improvement and one particular project - Docker is trying exactly that. This thesis therefore takes a look at the core virtualization technologies and tools and compares them to Docker, which is also described in detail along with other tools built on top of it.

The first chapter starts with the presentation of the most frequent use cases for virtualization and later explains the two major approaches to it. Next, the focus shifts to the analysis of virtual machine based virtualization techniques and tools. A more lightweight method of creating an isolated environment, container based virtualization, is described afterwards, presenting both its benefits and shortcomings as well as several available tools.

The third chapter introduces Docker, the new popular virtualization platform and describes its history and inner structures, while taking a look at its security and performance. Furthermore an overview of several large Docker-based virtualization projects is provided. As to how to use Docker, the following chapter explains its core principles, high-level tools and ends with a comparison to the other container based virtualization tools.

Finally, in the implementation part, Docker Remote API is used to exchange a file between two running containers. This simple test of Docker-java library serves as an example of interoperability of Docker with custom programs or scripts. Last but not least, Docket itself is modified and extended with a new command - *docker update*.

2 Virtualization

2.1 Virtualization use cases

On a daily bases, virtualization is used all in information systems all over the world. Large companies like Google, Facebook, Amazon or Microsoft use it on several layers of their infrastructure, but this technology is not employed only in server clusters, but millions of people use it on their personal computers as well. Its wide range of use cases could be split into these categories:

Consider running an instance of resource intensive application, for example Matlab. Using the benefits provided by executing it inside a virtualized environment, it could be easily restricted from exhausting the system's computational power and memory. Such use case is relatively rare when using a personal computer, but common in both academic and enterprise environments.

Testing is an important part of every software's development lifecycle. Especially for big, multiplatform projects, testing usually includes running the product on several system configurations. A different configuration may mean another version of operating system and used libraries or even an entirely different operating system. Available system resources, hardware and installed drivers are additional aspects that contribute the application's behaviour, so swapping or limiting them could also be the focus of testing. Having these different configurations on physical computers would prove impractical from both the timely and economical points of view.

Virtualization technologies are often closely related to emulation, currently mostly visible when employing virtual machines. Thanks to them, it is possible to run software written for entirely different architectures such as running ARM applications on x86 systems, or Linux software on Windows and vice versa.

Security experts frequently need to run an application in a sandboxed environment, when static analysis doesn't lead to any significant conclusions. Moreover, even ordinary users can become worried about how an untrusted piece of software might affect their primary system. The benefits offered by the ability to run the application inside an isolated system are what makes virtualization technologies extremely valuable in such scenarios.

Many IT companies do not have their own cloud infrastructure, but rather rely on using external solution. On the other hand a company having a large computational capacity may decide to rent some of it. There are many products from the cloud computing business on the market, but they can be separated into the following categories:

- **IaaS** - Infrastructure as a Service refers to offering access to part of the clouds infrastructure, such as servers, data storages or network components. The provider is responsible for the hardware's maintenance, while it's up to the customer to take care of the software he or she wants to run. Examples include Google Compute Engine, Microsoft Azure or Amazon Web Services, where the latter two may be also used as PaaS.

- **PaaS** - When deploying an application to cloud, it is often enough to use parts of the existing environment, rather than starting from zero. Platform as a Service is built on top of IaaS, and refers to environment which is largely set up and prepared to host the customer's services. In the contrast to IaaS, which is centred around managing the entire infrastructure, PaaS focuses on the individual applications. Commonly used PaaS products are OpenShift, Google App Engine and Heroku.

- **SaaS** - Built on top PaaS, Software as a Service are applications, which are already deployed in the cloud, are offered to customers. In this model, the provider is the one responsible for the applications' updates and availability. Office 365, Gmail or other Google Apps are all examples of SaaS

The virtualization takes place in all three layers, even in IaaS, as it often doesn't represent the hardware directly but rather a set of resources isolated from the rest of the cloud by using the means of virtualization. The operating system environment in the case of PaaS or the application instances in the case of SaaS are also isolated.

2.2 Virtual machine based virtualization

2.2.1 Virtual machine monitors

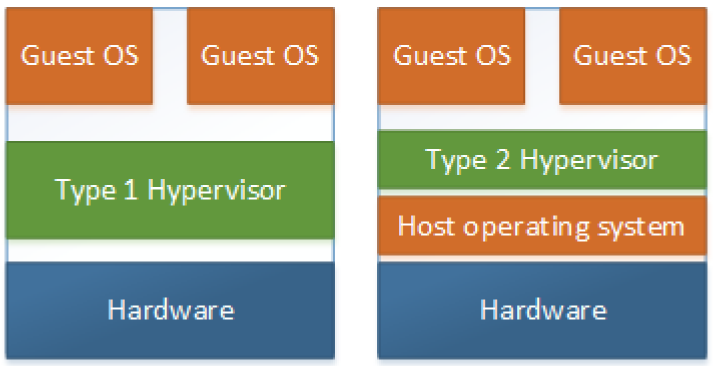
Virtual machine monitors, sometimes referred to as hypervisors are software or hardware solutions which enable creation and running of virtual machines. Popek and Goldberg have stated the following characteristics of virtual machine monitors [3]:

- The efficiency property. All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program.

- The resource control property. It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.

- The equivalence property. Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.

Two types of hypervisors are distinguished. The first type runs directly on the the hardware, while hypervisors of the second type act as applications inside another operating system. The operating system with direct access to hardware is called host, whereas the systems in virtualized environments are called guests.



Comparison between hypervisors of Type 1 and Type 2

Examples of type 1 hypervisors are Oracle VM Server, XenServer, Microsoft Hyper-V or VMware ESXi. Type 2 hypervisors include VMware Workstation or VirtualBox.

2.2.2 Software based virtualization

Since the guesting operating systems always run on top of a hypervisor, they do not access the hardware directly. Furthermore, from a security viewpoint, the shouldn't even be able to, since otherwise they would be able to corrupt the host or take control of it. In the following subsections, the detailed causes of the two most notable performance penalties of virtual machines are explained.

CPU

Modern CPUs utilise a resource protection scheme where several privilege rings are created and each of them represents a different level of trust. An x86 architecture offers up to four rings (numbered 0 to 3) and usually a lower ring number represents more trusted code. Sometimes, the ring numbers are referred to as current privilege level - CPL. The operating system's part, which needs the most privileges runs in ring 0, device drivers run in ring 1 or more and for most other applications, the highest ring number is enough. Some instructions, for example direct I/O require the highest privileges. When a user application needs such instructions, it calls a function provided by the operating system, which triggers execution of operating system's kernel code, running in ring 0. Once the function completes, the function returns the appropriate result and the execution of the application's code continues, with privileges falling back. Such mechanism is called a system call, the lower privileged layer(ring), where applications run marks the user space, and the kernel's layer is labelled as kernelspace.

From the virtualization perspective, this provides a security mechanism, which helps to satisfy the virtual machines monitor's security property. By not running the guest code in ring 0, the host operating system is the only one with full control of the computer's resources. However, when the guest operating tries to execute a system a call, it would fail, since it doesn't run in ring 0. To overcome this issue, one can employ paravirtualization, a compile-time technique, in which the instructions that are impossible to execute in virtualized environment are replaced statically. Furthermore, additional drivers may be installed, which allow the host to communicate with the hypervisor. Another option is to replace the instructions at runtime, by scanning the executing code and patching it where necessary.

Memory

Modern processors use a concept of virtual memory, where every application gets the illusion, that it can use the entire system memory for itself. In order for this technique to work, the operating system has to manage the mapping of virtual memory to physical memory. If the total memory consumed by applications is larger than the physical memory, parts of the virtual memory are temporary moved to a secondary storage (disc).

Since the guest operating system doesn't know that the host is already managing the virtual memory, this introduces a heavy performance penalty for virtual machines, because every memory access from the guest has to be mapped firstly in the guest's address space and then again in the host's address space. The solution is to shadow the page table, (the structure which keeps track of memory mappings), where every time the guest maps a memory, the hypervisor performs a direct mapping to host's memory in the shadow table. Then, when the guest attempts to access the memory, the shadow page table is used instead of the one managed by the guest.

2.2.3 Hardware assisted virtualization

Historically, there wasn't much support for virtualization in the computer's hardware. However in 2005 and 2006, two new technologies were released, AMD-V[5] for AMD processors and Intel VT[4] for Intel processors. While both technologies are very similar in their functionality, the terminology slightly differs. I have decided to use the AMD-V terminology in the rest of the chapter.

CPU

A new structure was introduced - Virtual machine control block (VMCB), which represents a virtual machine inside the CPU. When a VMCB is run (VMRUN instruction), CPU executes the following steps:

- The hosts state is saved in a memory area specified by VCMB
- The guests state is loaded from a memory area specified by VCMB
- The guest code begins to run

The VCMB structure also contains the desired CPL level, allowing the virtual machine to run even at ring 0. However, some operations, such as direct device I/O are still prohibited and cause the virtual machine to exit the guest state. After the exit, virtual machine monitor may decide to change several fields in the VCMB, thus effectively emulating the I/O and execute VMRUN again.

Memory

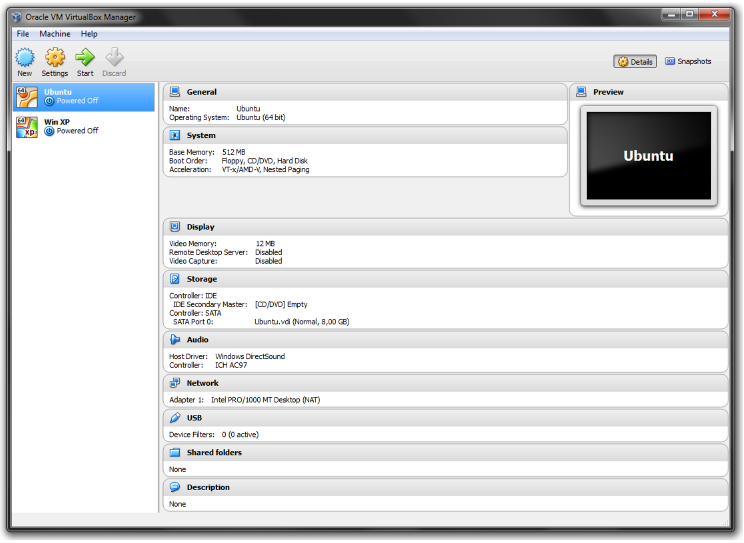
Intel and AMD have also come with a mechanism to eliminate the need for shadowing page tables. Extended Page Tables (Intel) and Rapid Virtualization Indexing (AMD) are examples of secondary level address translation, where the nested page tables are created and maintained by the hardware. In this model, the hardware performs both the translations from guest's virtual address space to guest's physical address space and from guest's physical address space to host's physical address space.

2.2.4 VirtualBox

VirtualBox is an open source, type 2 hypervisor. It was initially developed by Innotek GmbH, a German company, which was later bought by Sun Microsystems. Today, VirtualBox is branded as Oracle product, since in 2010, Oracle acquired Sun Microsystems. The full product name is thus Oracle VM VirtualBox.

VirtualBox supports all major operating systems as either hosts or guests and with the exception of a guest OS X on non-Apple hardware (although this is possible using a cracked image of the OS X installation). Both software based and hardware assisted virtualization are implemented. 64-bit guests are supported as long as either the host is 64-bit or 64-bit hardware virtualization is supported on the CPU.

Other notable features are shared folders, shared clipboard or cloning of virtual machines. A complete state of any running virtual machine (Snapshot) can also be saved to a file and restored later. VirtualBox offers a rich GUI interface, a command line interface (VBoxManage), an interactive Python Shell and web API. A portable version that doesn't require installation is available too, under the name Portable-VirtualBox.



General
Name: Ubuntu
Operating System: Ubuntu (64 bit)

System
Base Memory: 512 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: Ubuntu.vdi (Normal, 8,00 GB)

Audio
Host Driver: Windows DirectSound
Controller: ICH AC97

Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)

USB
Device Filters: 0 (0 active)

Shared folders
None

Description
None

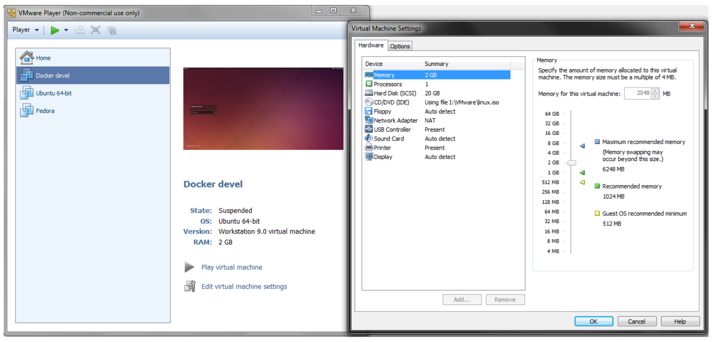
VirtualBox's GUI

2.2.5 VMWare Player

VMWare player is a virtualization software created by the company VMWare. While VMWare offers a big range of virtualization products, I will focus on VMWare player and only mention the other ones briefly, since VMWare Player is the only one that can be used for free, (and thus can be a better comparison for VirtualBox and Docker, which are both free) although the licence restricts this to be a non-commercial use.

VMWare Player's features differ very little from what is offered by VirtualBox. In the terms of supported operating systems, the most notable missing feature in VMWare Player is the inability to install it on OS X (VMWare offers a commercial product VMWare Fusion). On the other hand, it is possible to install retail copies of OS X as a guest system even on non-Apple hardware. Another missing feature is taking snapshots, which is only available in the commercial VMWare products. For some features, such as sharing clipboard and folders, VMWare Player requires VMWare tools to be installed on in the guest system.

Similar to VirtualBox's API, VMWare's VIX API gives the ability to control virtual machines programmatically. VIX also contains a command line utility vmrun, which enables using VMWare Player without GUI.



Docker devel

State: Suspended
OS: Ubuntu 64-bit
Version: Workstation 9.0 virtual machine
RAM: 2 GB

Device	Summary
Memory	2 GB
Processors	1
Hard Disk (SCSI)	20 GB
CD/DVD (IDE)	Using file I:\VMware\linux.iso
Floppy	Auto detect
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

Memory

Specify the amount of memory allocated to this virtual machine. The memory size must be a multiple of 4 MB.

Memory for this virtual machine: 2048 MB

Maximum recommended memory (Memory swapping may occur beyond this size.) 6248 MB

Recommended memory 1024 MB

Guest OS recommended minimum 512 MB

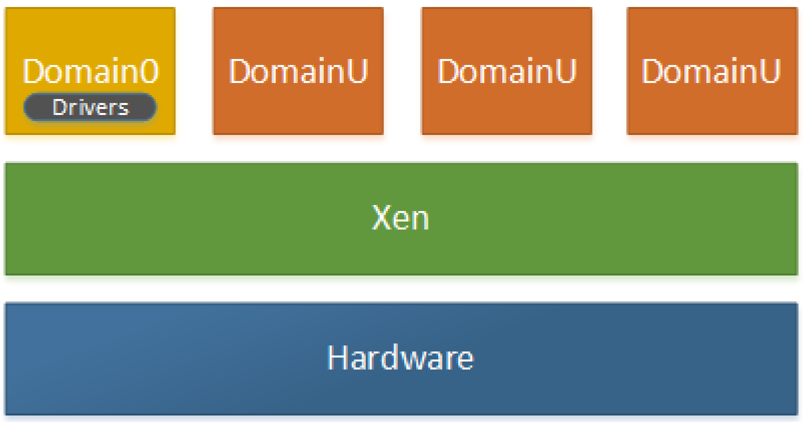
VMWare Player in use

2.2.6 Other tools

Quick Emulator (**qemu**) is an open source virtual machine monitor. It runs on Linux, WIndows and OS X. As the name suggests Qemu can perform an emulation of a hardware and thus enable the host to run software written for different architecture. One can utilise this to run only one application or an entire operating system. Performance wise, emulation of the cpu instruction set is a costly process so qemu is often used in conjunction with another virtualization software which handles the execution.

KVM - kernel-based virtual machine, an extension to the Linux kernel, giving it type 2 hypervisor capabilities. KVM works by exposing an API, which can be used to interact with some kernel and hardware features. This API is then used by a client such as Qemu. This allows the software hosted by Qemu to run much faster. Hardware-assisted virtualization is used, so a CPU supporting it is required.

Xen, initially released by the University of Cambridge in 2003 is an example of type 1 Hypervisor. It has grown into "#1 Open Source hypervisor according to analysts such as Gartner. Conservative estimates show that Xen has an active user base of 10+ million: these are users, not merely hypervisor installations which are an order of magnitude higher. Amazon Web Services alone runs ½ million virtualized Xen Project instances according to a recent study and other cloud providers such as Rackspace and hosting companies use the hypervisor at extremely large scale. Companies such as Google and Yahoo use the hypervisor at scale for their internal infrastructure." [6] In Xen's terminology, the installed hosts are called domains, out of which exactly one is privileged domain0. Domain0 is used for management and control of the other unprivileged domains (domainU guests) and contains network and disc drivers for paravirtualized guests.



Xen architecture

Parallels Desktop for Mac Parallels Desktop for Mac is a commercial virtualization tool for OS X, which is popular mainly due it Windows-virtualization capabilities, although Linux, OS X and a few other operating systems are supported as well. It relies on hardware-assisted virtualization technology from Intel. While the company Parallels offered virtualization products for several platforms, a lot their products such as Parallels Workstation became discontinued in 2013.

2.3 Container based virtualization

Providing an isolated environment inside the hosting operating system is commonly known as operating-system level virtualization and such an isolated enviroment can be defined as container: “A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.“ [11] Commonly used technologies are Solaris Containers or Zones, OpenVZ, FreeBSD jails and LXC.

2.3.1 Chroot and jail

As Linux developed, the idea to isolate a process from the host filesystem arose and a chroot command was created for this purpose. Chroot being short for *change root* is both a utility and a system call and it allows to specify a new root directory other than /. The process and its children then can't access files above the new root directory while programs from elsewhere can still see inside the the new root. It a crucial that none of the processes inside can obtain root privileges as that can potentially allow them to break out of the specified directory. Such a procedure is often called *jailbreak* and could be easily performed by issuing chroot again while leaving open a file descriptor pointing to a file outside of the newly selected root directory.

Nowadays chroot is used to provide basic isolated environments for testing unknown and unstable applications or for discovering unwanted dependencies. Package building tools like Pbuilder for Debian or Mock for Fedora also utilise chroot to provide isolation and enable testing in different Linux distributions.

An advanced mechanism built on top of chroot is jail. It is available in FreeBSD since 2000 and adds isolation of process lists, sets of users and networking. Jail can therefore define a new root user, which has full control inside it, but cannot reach anything outside. The limitations are in the form of inability to mount or unmount filesystems or modifying the network configuration. Jails can be started, stopped or restarted and with the use of ezjails utility, even archiving and later restoring a jail is possible.

2.3.2 Namespaces

Namespaces are one of the key features of the Linux kernel for supporting lightweight virtualization. "The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource." [13] To get a better understanding, I will focus at the individual types. Six namespaces are currently available in Linux.

Mount namespace was the first one implemented and its behaviour is quite intuitive. All the mounts/unmounts from the global namespaces are visible in it. Mounts/unmounts which happen in the namespace remain invisible to all other namespaces including the global one. However setting a master-slave relationship is also possible, to allow propagation of mounted devices.

UTS(Unix timestamp sharing) allows to isolate gethostname(), getdomainname() identifiers as well as corresponding members of uname(). Any changes made by calling sethostname() or setdomainname() are only visible inside the caller's namespace.

IPC(Interprocess communication) namespace similarly provides isolation for System V IPC (shared memory, semaphores) and different filesystems for POSIX message queues.

PID(process ID) namespace isolates the list of process ids. It allows processes from different namespaces to have the same PID and nesting of PID namespaces is allowed. The principle is that a process from a particular namespace can see (and send signals) only to processes in its own namespace or in namespaces nested below it. The first created process in PID namespace has PID = 1, and when any process in the current namespace dies, all of its orphan processes become children of the process with PID = 1. It is not possible to send SIGKILL to any process with PID = 1.

Network namespaces allows each namespace to have it's own network stack, including but not limited to: IP addresses, port numbers, routing tables, firewall rules, network devices. When a new network namespace is created, it only contains the loopback device(lo), however network devices may be moved across the network namespaces. The rule is, that a network device other, than lo may only belong to one network namespace. Furthermore, physical devices cannot be moved from the default namespace. Thus if one wants to give networking capabilities to a non-default network namespace, the usual approach is to create a virtual ethernet device (veth) in the default namespace, bridge one end with physical devices and move the other end to namespace, which needs networking. Nesting of network namespaces is also possible.

User namespaces, the last namespace to be implemented, give means to processes to have different user and group ID inside the user namespace. Files /$proc_id/uid_map and /proc/$proc_id/gid_map contain the actual mapping of user(group) ID ranges from child to parent user namespaces. The primary benefit is, that even a non-privileged user can start a process with root privileges, inside a particular namespace. Nesting of user namespaces is supported.

The implementation of namespaces added two new system calls, setns() for joining an existing namespace and unshare(), which allows the calling processes to continue in a newly created namespace.

The clone() system call has added 6 new flags, one for each of the namespaces, to allow the child process resulting from the call to start in a newly created namespace.

2.3.3 Control groups

With the use of namespaces, processes could be isolated from each other, but that's still far from what standard virtualization offers. As the term suggests, virtual machine monitors provide also ways to manage the virtual machines, not just create and run them. Here's where control groups, frequently abbreviated to cgroups, come in the picture.

Control groups is a kernel feature originally started in 2006 by Google engineers that enables administrators to restrict and/or limit the usage of system resources for groups of processes. The groups can be nested in a forest-like structure, where the roots of trees are the default groups of each system.

Subsystems are the controllers referring to individual system resource types. Configurations for every subsystem's group are stored in filesystem (usually /cgroups/subsystem/), where the default groups settings reside. Settings for child groups are also stored here, recursively stored in additional directories. The most commonly used subsystems are the following:

- blkio - Control of access to block io devices. Can be used to limit read / write speed, or set read / write priority of a group, either globally or in per each device. Speeds can be set in io operations or bytes per second.
- cpu - Assign the overall CPU power, either setting the CPU priority or the absolute time from a set period the processes can run.
- cpusets - Specify individual CPUs the group can use.
- devices - Allows to create a whitelist containing access rights to individual devices.
- memory - Impose limits on the memory used by all processes in the groups. It's possible to include swap memory.

- net_prio - Can override SO_PRIORITY option of processes' sockets, which is used as the priority level of packets in the Linux's networking queues.

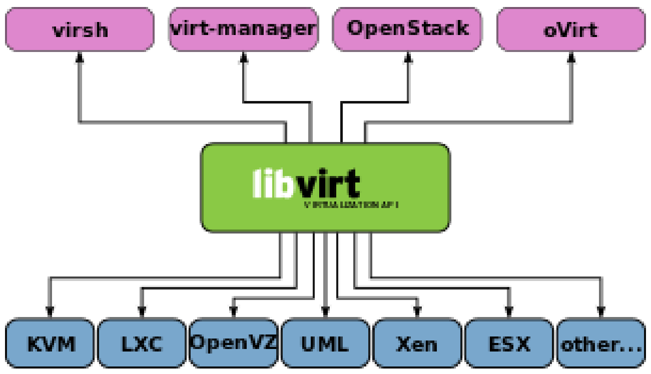
Moreover, cgroups provide a freezer subsystem, which can be used to freeze a non-root group of processes. Cgroups can be used also for monitoring, where each relevant subsystem keeps stat files, tracking every groups resource consumption. A notification mechanism can be used to capture various events, such as reaching the allocated resource limit.

2.3.4 Container based virtualization tools

LXC (Linux containers) combines the features offered by namespace and control groups to create a fully isolated environment, which is can be easily managed - a container. Since containers are the unit LXC deals with, the basic operations are all provided: creating, starting, stopping, listing and removing containers. An empty container won't be of much use to anyone, therefore it is possible to create containers based on a template, which sets up the container's initial filesystem and configuration. LXC comes bundled with templates for several major Linux distributions. Additionally LXC can freeze and unfreeze containers, giving the ability to suspend and later resume their execution. It is also possible to create a checkpoint, an information about the state of a container. The container can be later restored to its previous state, captured by the checkpoint. Cloning of containers is also possible. Containers can be also interacted with programmatically, either using containers lifecycle management hooks, or using an API - liblxc, which has binging for several language, including Java, C, Python, Ruby and Go.

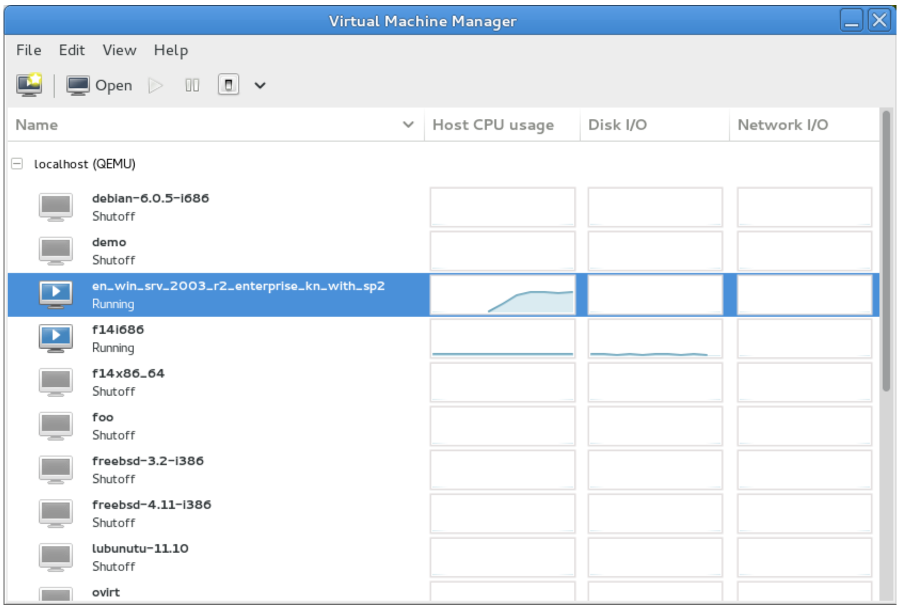
OpenVZ from Parallels is another container based virtualization technology, and is somehow an ancestor to the LXC project, as lot of LXC's codebase is either derived from OpenVZ or contributed by OpenVZ team members. The core distinction is, that OpenVZ project uses its own kernel, originally derived from Linux's kernel version 2.6. However, most of the kernel changes required by OpenVZ were recently ported to the official kernel and thus it is possible to run a feature-limited version OpenVZ on the official kernel. Even if OpenVZ may seem like an outdated software, many system admins prefer it over LXC, since it has been used successfully for long time and thus proved itself in production environments. LXC also doesn't have all the features offered by OpenVZ, such as live migration - suspension of a running container on one host and resuming it on another. OpenVZ is often used to provide virtual private servers (VPS).

I've mentioned several virtualization tools, and there are many more available. Since it is quite possible for a company to have multiple virtual machines and / or containers, each created by a different virtualization provider, their management could become a complex task. To overcome this problem, **libvirt** provides a unified API, supporting every hypervisor that was mentioned and several others, even from the container-based virtualization world. While it is written in C, bindings for other popular programming languages are available as well. Libvirt can be used on Linux, Windows as well as on OS X.



Libvirt provides unified interface for many hypervisors[7]

Virt-manager (Virtual Machine Manager) is virtual machine management tool built by Red Hat on top of libvirt, which comes with a rich GUI. It is currently only available on Linux.



Name	Host CPU usage	Disk I/O	Network I/O
localhost (QEMU)			
debian-6.0.5-i686 Shutoff			
demo Shutoff			
en_win_srv_2003_r2_enterprise_kn_with_sp2 Running			
f14i686 Running			
f14x86_64 Shutoff			
foo Shutoff			
freebsd-3.2-i386 Shutoff			
freebsd-4.11-i386 Shutoff			
lubunutu-11.10 Shutoff			
ovirt			

Virt-manager's GUI [8]

2.3.5 Comparison with virtual machines

Virtual machines create a new instance of an operating system for every virtual machine run. This gives several benefits such as the ability to run entirely different guest system, compared to host it also comes with many drawbacks. The first thing that comes to mind is the virtual machine execution overhead, caused either by the virtual machine monitor's instruction patching and translation, the paravirtualization drivers or in case of hardware assisted-virtualization, the source of the overhead is the CPU context swapping. Secondly the virtual machines take much more disc space and are more difficult to maintain.

When compared to virtual machines, it's the additional overhead that containers takes away. Containers only require the application and it's dependencies, while the kernel is shared among them. Because the operating system is already running, starting a container tends to be much quicker than starting a virtual machine. Shared kernel may not always be a benefit, as for example running Windows applications in containers on Linux is not possible.

3 Docker

The idea behind Docker project is well expressed by the following goal set by its development team: "To build the 'button' that enables any application to be built and deployed on any server, anywhere." [14] At it's core, Docker is an open-source platform which allows applications to be deployed inside software containers. This start-up from the Silicon Valley has quickly caught attention of IT-world leading companies. Amazon, Google, Microsoft and Red Hat added support for Docker to their platforms and continuously contribute to the project.

But Docker is more than just a virtualization library, it abstracts away the differences between operating systems distributions and creates a standardized environment for developing applications. A software developer can create a standardized application which becomes portable and can run everywhere where the Docker Engine is installed. This saves a lot work for the author as it is no longer necessary to support many different platforms and operating system distributions. System administrators need to spend less time configuring the application as it comes packed with all its dependencies. The fact that each application runs in it's own container solves many common problems like completely uninstalling or replacing it or when two applications require two different versions of the same dependency.

Docker is a relatively new piece of technology, so it still comes with some limitations. Firstly, it only supports application which can be run on Linux, at least for now, although the recent partnership with Microsoft may change things in the future. Docker also runs natively only on Linux, and while it's possible to use additional software to run it on Windows or OSX, it still requires a virtual machine to do so.

3.1 History of Docker

The initially internal project in dotCloud was released as open source in March 2013. Two months later, the public Docker registry was launched. In the second half of the year Google, Yandex and Baidu

(Russian and Chinese most used search engines) have integrated Docker into their cloud services.

Docker entered 2014 with completing a $15 million fund[15], allowing it to heavily invest both in the open source project and planned enterprise support as well as in expanding the community platform. In April, LXC was dropped as the default execution environment in favour of Docker's own libcontainer. Next month, Ubuntu 14.04 became the first enterprise grade Linux distribution to ship with Docker natively packaged, bringing millions of Ubuntu servers no more than three command away from using Docker containers. The version 1.0 was finally released in June at the first Docker-centric conference - DockerCon. September brought the announcement that another major fund of $40 million was raised[16], valuing the project at roughly $400 million[17]. One month later, Docker and Microsoft declared partnership with the goal of creating Docker Engine for Windows Server and multi-Docker container model, including support for applications consisting of both Linux and Windows Docker containers. In December the first official Docker conference in Europe took place in Amsterdam, announcing several new Docker related projects as well as Docker Hub Enterprise. Docker finished the year 2014 with the release of 1.4, being the 24th most starred project on GitHub.

In February 2015, version 1.5 was released, bringing IPv6 support, read only containers and support of multiple Dockerfiles per project. Shortly afterwards, a trio of orchestration tools was announced : Docker Machine, Docker Swarm and Docker compose. The current stable version(1.6) was released in April and came with the long expected Windows client and the ability to apply custom labels to images and containers.

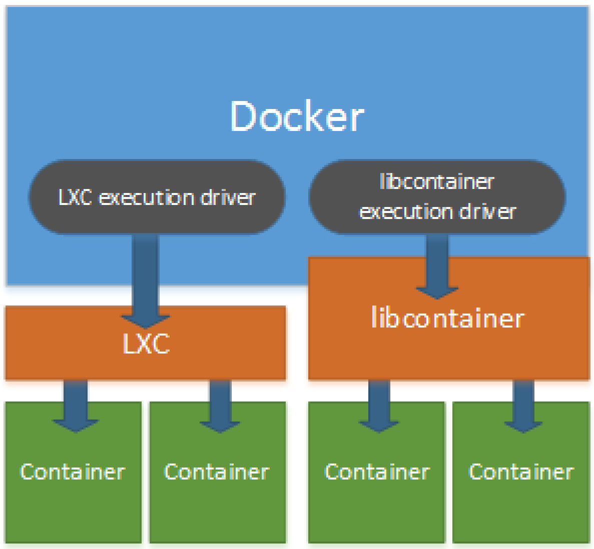
3.2 Docker daemon

Internally Docker uses a client-server model, where the server is a daemon which may run on an entirely different system than the client. The daemon may be either started by using the docker command while passing -d flag, or starting it a service with *systemctl start docker* or *service docker start*. Privileged account is required to run docker in daemon mode, although significant effort is made to remove this drawback, so even regular users would be able to run containers.

By default, the server only listens on a Unix socket, making itself unreachable over the network. This is a security feature, since anyone who can access the daemon could easily take control of the entire host. A trivial attack would be to run a container with mounted host's / directory. The container would then be able to rewrite any host's file. Therefore it is critical to only run Docker on public IP with TLS, where each client is authenticated by a certificate from a trusted certificate authority.

3.3 Libcontainer

In Docker vs 0.9 a concept of execution drivers was introduced and two drivers became supported: the LXC driver, which used the in past required liblxc and the native driver utilising Dockers own library, libcontainer. It's written purely in Go and handles the management of containers, using the previously mentioned kernel capabilities such a namespaces and control groups. Libcontainer is expected to be ported to other other languages and to support operating systems, removing the need for additional tools such as Boot2docker when Docker is being run on Windows.

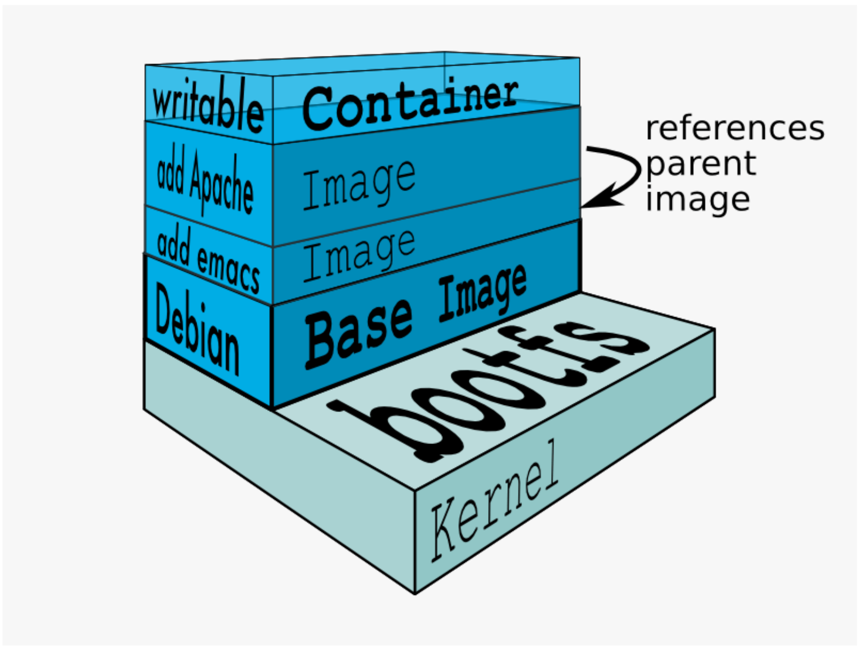


Execution drivers in Docker

3.4 Layering filesystem in Docker

When launching a container, Docker uses a mechanism called union mount - filesystems are not mounted at different places but on top of each other, thus a directory content may be composed of files a directories from different filesystems.

In Docker, applications usually specify a parent image. For example, an web application could depend on a specific web server, which in contrast would depend only on an operating system. This means, every image adds a new read only layer in the filesystem on top of its parent's layer. Once the application is started as a container an additional writable layer is put on top. When a file from a read-only layer needs to be changed, this file is copied to the writeable layer and the change is made there. It is important that the changes to the upper layer persist even after the container exits and thus are still in effect during its next run.



Docker's filesystem[12]

Data volumes are an exception to the union file existing. They give way to sharing data between containers, making direct changes to the filesystem and these changes are not included when committing updates to the used image. One can create an empty data volume or mount a host file or directory. A data volume may be either added to a specific container or a dedicated data volume container may be created, which can be not only be shared among containers, but advances functions such as migrations, backups and restores are also provided by Docker.

3.5 Security

Docker currently requires root privileges, therefore if it gets compromised, the host will be exposed as well. I have showed an example of such situation in the chapter about the Docker daemon. Mainly because of this security threat, one of goals for Docker is the ability for non-root users to run containers[2, p. 110]. Docker has already announced that it's working on it, changing the architecture to two daemons. The current daemon will run in user space, while the privileged operations will be forwarded to a new service in kernel space.

In December 2014, Docker promoted a new feature called image signing. It was long requested for images to contain a cryptographic signature, so that they will be verified prior to running. However, a detailed inspection of the implementation revealed that "Docker's report that a downloaded image is *verified* is based solely on the presence of a signed manifest, and Docker never verifies the image checksum from the manifest. An attacker could provide any image alongside a signed manifest. " [23] Even additional problems were found such as badly constructed tarsum used for the image verification, processing the manifest after the image was extracted, or the fact that if the manifest is incorrect only a warning is issued and the image is still run. In response to the mentioned discoveries, the Docker team initiated a security audit a promised to revise Docker's security.

The other potential threat to container based virtualization comes with security issues in the Linux namespaces implementation. Namely, bugs are being discovered in the user namespace, which hasn't yet been thoroughly tested in production as it was implemented quite recently. An example of a recently discovered vulnerability is, that process could potentially gain access to a filesystem entry, to which even the user running it was denied access. This could be accomplished by the possibility of dropping supplementary groups from within a user namespace. It could be achieved by calling setgroups, which was possible prior to the existence of guid mapping (thus having root privileges only in namespace). Several other vulnerabilities have also been reported and fixed in the recent months. Recently, during a security fix in December 2014, one the kernel's developer has warned, that "..while it seems possible to contain privilege within a user namespace, there is always the possibility of surprises like this one hiding in the corners of the system. It may be some time yet before we can be truly confident that all of those surprises have been found and that the unprivileged creation of user namespaces is truly a safe thing to allow." [24]

3.6 Performance

A detailed performance testing of Docker was done by IBM in January 2014. The tests comparing the Docker to virtual machines covered memory access, block I/O, networking and benchmarking Redis and MySQL instances. Considering the differences between the two virtualization technologies, the results confirmed what was expected: "Both VMs and containers are mature technology that have benefited from a decade of incremental hardware and software optimizations. In general, Docker equals or exceeds KVM performance in every case we tested. Our results show that both KVM and Docker introduce negligible overhead for CPU and memory performance (except in extreme cases)." [25] The tests have also revealed that the performance of Docker can significantly differ depending on whether it's using the host's network or a NAT bridge. Similarly, the overhead is slower when data are stored on a shared volume rather than in the union filesystem.

3.7 Future

Docker has inspired the creation of many projects, which take advantage of it's functionality. While these projects are under development, new ones are still emerging, as everyone wants to fill a gap in the market as soon as possible. It yet remains to be seen, which ones will emerge as winners from this competitive environment, but I will mention the ones that seem to show the greatest premises.

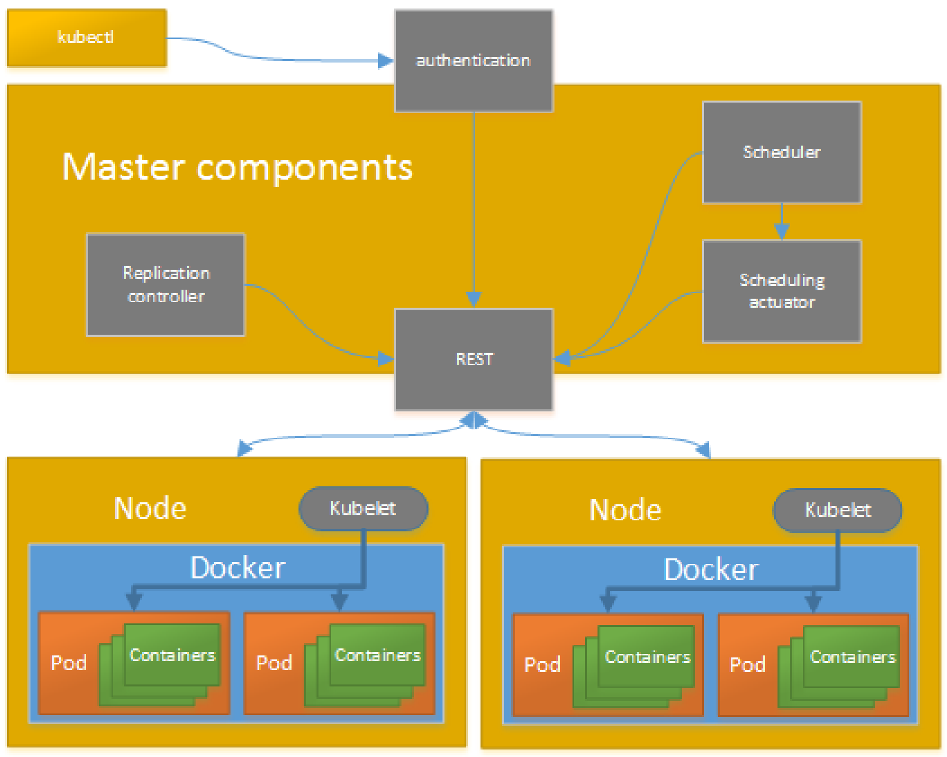
Kubernetes

Google, undoubtedly one of the largest data centre operator, has also admitted that every single one of their services runs inside a Linux container [26]. While they have not yet shared their main internal task scheduler - Omega, they have released another container manager Kubernetes, as an open-source project. Currently, the containers are run inside Docker, although support of rkt, which will be covered in a later chapter, is planned as well.

Kubernetes introduces the concept of Pods - groups of containers that are relatively tightly coupled and are treated as the smallest deployable unit. A common use case would be a main container running a web application utilising several other containers with its helper services. It makes sense for these containers to start/stop at the same time and to run on the same host. Pods have usually attached labels (key,value pairs) to them, which enables querying the cluster nodes running a particular group of pods.

The next abstraction is Service - a label-defined groups of ports, exposing the same port and running the same application. A pod may stop running on a node and be replicated on another, so with the use of a service, other pods doesn't need to keep track, which pod is running of which node, instead, they can use a virtual IP of the service.

To support scaling, Kubernetes introduces replication controllers. A replication controller uses a template, according to which the pods are created and allows to set the numbers of pod replicas to be running. This can be used to easily increase or decrease the number of the running instances of an application. Another use case would the update of an application, when using one replication controller for the old version and one for the new version, continuous uptime could be achieved.



Kubernetes architecture

Mesos

“If a Docker application is a Lego brick, Kubernetes would be like a kit for building the Millennium Falcon and the Mesos cluster would be like a whole Star Wars universe made of Legos.” [27]. Apache Mesos, launched in 2013, abstracts the resources available in a cluster, presenting it as a single gigantic computer. Its architecture defines the concept of a framework - an application written for Mesos, which contains a scheduler an an executor for the application’s tasks. Kubernetes may be used as one of the frameworks so a possible task may be the launching of a pod.

The interaction cycle between the cluster, Mesos and a framework can be described as follows:

- A Mesos slave daemon runs on every node in the cluster

- One Mesos master daemon is used to control the slaves
- Slaves present resources available on the node to the master
- Master node selects a framework and offers it part of the available resources along with node identifiers - the policy for deciding which framework gets which resources is configurable.
- Framework's scheduler decides which tasks to run on which nodes and specifies the resource allocation
- Master node invokes frameworks executor on the selected nodes, with the requested resources

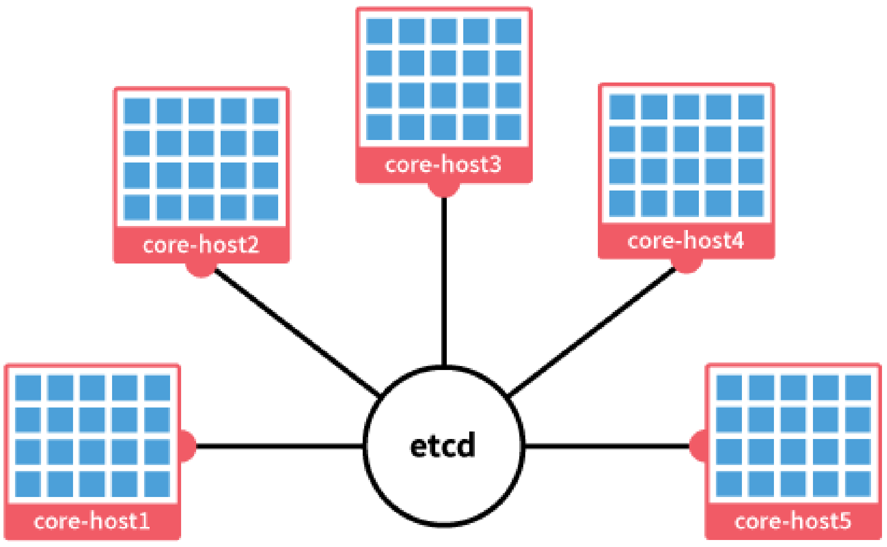
Mesos quickly gained popularity in companies, which employ huge clusters. Namely PayPal, Vimeo, Twitter, Airbnb, Netflix are all using Mesos. Even Apple has announced, that Siri, the personal assistant in iOS uses Mesos in its backend [28].

A datacenter operating system called Mesosphere is currently being built on top of Mesos, which focuses on scaling, self-healing and fault tolerance. It is expected to be released in 2015.

CoreOS

CoreOS is a small open source operating system based on Google's Chrome OS. With its first release in the fall of 2013 it started to push its vision of a container-centric operating system. This means that CoreOS doesn't include a package manager, but the applications are instead provided in the means of containers. Docker is currently used as the container manager.

CoreOS tries to target cloud infrastructure with it's two key utilities, etcd and fleet. Etcd is shared cluster configuration management daemon, providing an API for propagation of configuration changes across an entire cluster of etcd instances. On the other hand, fleet is a cluster level systemd control daemon. It allows to deploy containers either globally(across all machines in the cluster) or on a single machine with support for failover.



a cluster of CoreOS hosts with containers[10]

Snappy Ubuntu Core and LXD

Another operating system, targeted to running containerized applications is Canonical's Snappy. This project is trying to provide the minimal operating system required for running container based applications, although very few applications have been ported. Multiple container providers are supported (including Docker), which are called *frameworks*.

Snappy comes with atomic update system, which applies to both installed applications and Snappy itself. Updates happen in the form of transactions, so it is always possible to rollback a failed / undesired update. Internally Snappy keeps the base versions of all packages, and updates only send the difference from the previous version.

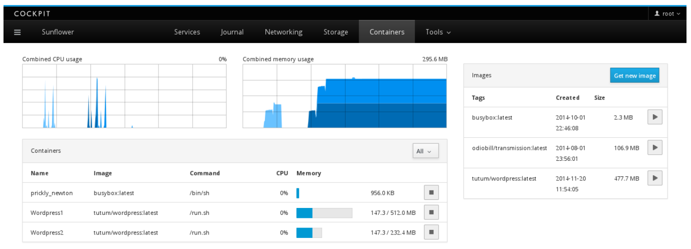
Ubuntu is also developing LXD (Linux Contained Daemon), which is aiming to "take all the speed and efficiency of docker, and turn it into a full virtualisation experience" [29]. LXD should be an extension to LXC, providing a REST API and most notably live migration support and checkpoint-resume abilities for containers.

Project Atomic

Project Atomic is a set of components which provide solutions for deploying containerized applications. The main result of Project Atomic are Project Atomic hosts, which are lightweight operating systems based on either Red Hat Enterprise Linux, Fedora or CentOS. The components included are most notably Docker, Kubernetes, SELinux, rpm-ostree and Project Cockpit.

Rpm-ostree is yet another tool for providing atomic updates to the operating system. A previous version is also stored for rollbacking purposes, since it works by placing the updated version in a newly created filesystem root. The system then boots from the new filesystem, keeping the previous version intact.

Project Cockpit is remote Linux server manager, which comes with also contains a web based GUI. Its main benefit is providing clean visualization of the server's status, which is a great help for new system administrators. Project Cockpit is expected to ship with Fedora Server 21 as well.



Name	Image	Command	CPU	Memory
prickly_newton	busybox:latest	/bin/sh	0%	956.0 KB
Wordpress1	tutum/wordpress:latest	/run.sh	0%	147.3 / 512.0 MB
Wordpress2	tutum/wordpress:latest	/run.sh	0%	147.3 / 232.4 MB

Tags	Created	Size
busybox:latest	2014-10-01 22:46:08	2.3 MB
odiobill/transmission:latest	2014-08-01 23:56:01	106.9 MB
tutum/wordpress:latest	2014-11-20 11:54:05	477.7 MB

Project Cockpit's GUI [9]

Rkt

In December 2014, the CoreOS team has expressed its concern about the direction the Docker project has taken and released a blog post, which immediately caught attention. They expressed their disagreement with how wide the scope of Docker has grown:

“When Docker was first introduced to us in early 2013, the idea of a *standard container* was striking and immediately attractive: a simple component, a composable unit, that could be used in a variety of systems. The Docker repository included a manifesto of what a standard container should be.“ [30] [31]. "We thought Docker would become a simple unit that we can all agree on." “The standard container manifesto was removed. We should stop talking about Docker containers, and start talking about the Docker Platform. It is not becoming the simple composable building block we had envisioned.”

This criticism was probably targeted at the trio of projects, Docker Machine, Docker Swarm and Docker Compose, which were released in 2015. They also claim that the architectural model of Docker, where everything runs through a central daemon, is "fundamentally flawed" from the security’s perspective. Such arguments immediately spawned discussions whether or not is Docker trying to do too much and becoming a large monolithic platform, without providing reasonable modularity, or has only stepped into the business area of other project/startups and this blog is an example of an attempted defense.

Furthermore, the post contained an announcement of Rocket - a new container runtime and a direct competitor to Docker. CoreOS vision is to center it around a specification for containers and the “App Container executor,” which they included and encouraged developers to submit feedback for [32]. Later, it was renamed to rkt - “rock it”. Rkt is currently under heavy development, with very little documentation or examples, but it already allows running of native Docker images.

They team has also expressed, that it is possible for Docker to implement their App Container specification, making the two projects interoperable. A statement, that CentOS will continue to ship Docker was included too.

4 Using Docker

4.1 Installing Docker

Installation of Docker on any major Linux distribution is nowadays a very straightforward process, which means simply using the distribution's native package manager.

Ubuntu

Assuming the version 14.04, the installation command is

```
sudo apt-get install docker-io
```

alternatively, using wget

```
wget -qO- https://get.docker.com/ | sh
```

CentOS, Fedora

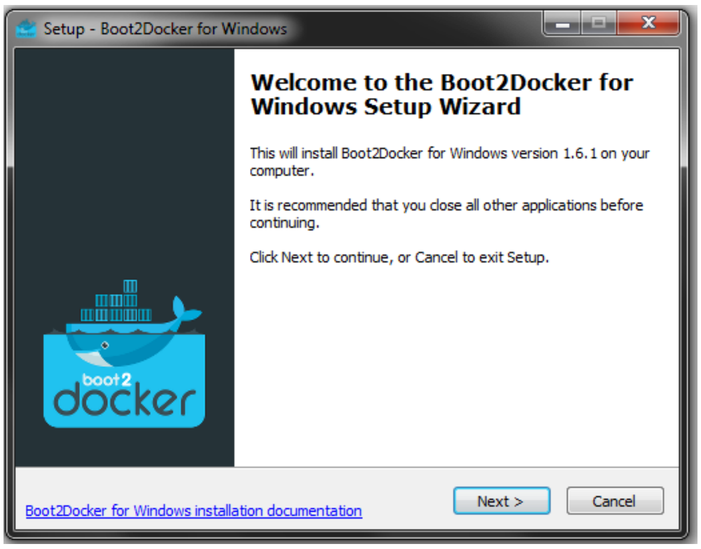
On these distributions, yum gets to act

```
sudo yum install docker-io
```

Sometimes, the package name mentioned is just *docker*, however there may be a package name conflict with the a system tray application (for example on CentOS 6.5), thus it's safer to use *docker-io*.

Boot2Docker

Boot2Docker is lightweight Linux distribution, based on Tiny Core Linux - which is another very small (12 MB) distribution. One of its most frequent use cases is running Docker on Windows or OS X. For Windows and OS X an installer is provided, which installs VirtualBox and creates a virtual machine with the Boot2Docker system and adds an application and shell commands for it's management.


Setup - Boot2Docker for Windows

Welcome to the Boot2Docker for Windows Setup Wizard

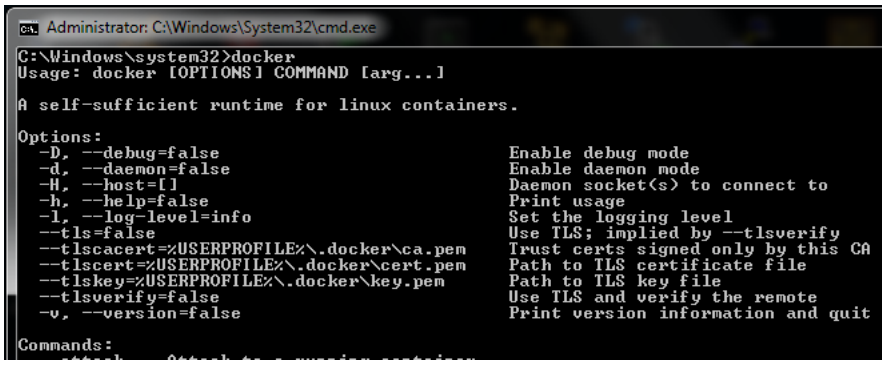
This will install Boot2Docker for Windows version 1.6.1 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Windows

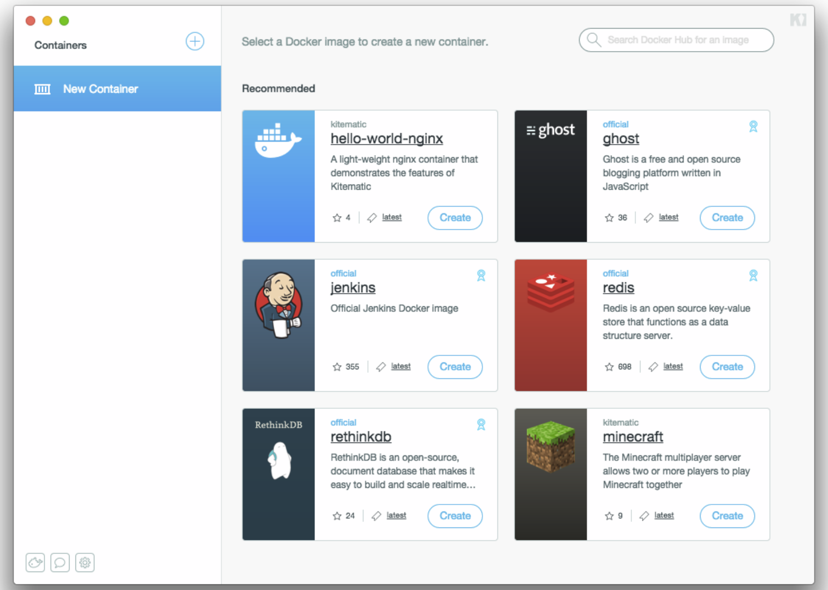
Even though Docker requires Linux's kernel, it's possible to install it on Windows via Boot2Docker, which we will explore later. However, it is possible to use a client only version, which was released with Docker 1.6. The package is currently available only through chocolatey package manager [18].


Administrator: C:\Windows\System32\cmd.exe

```
C:\Windows\system32>docker
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.

Options:
  -D, --debug=false                             Enable debug mode
  -d, --daemon=false                            Enable daemon mode
  -H, --host=[]                                 Daemon socket(s) to connect to
  -h, --help=false                              Print usage
  -l, --log-level=info                          Set the logging level
  --tls=false                                   Use TLS; implied by --tlsverify
  --tlscacert=%USERPROFILE%\.docker\ca.pem      Trust certs signed only by this CA
  --tlscert=%USERPROFILE%\.docker\cert.pem      Path to TLS certificate file
  --tlskey=%USERPROFILE%\.docker\key.pem        Path to TLS key file
  --tlsverify=false                             Use TLS and verify the remote
  -v, --version=false                           Print version information and quit

Commands:
```

OS X

With OS X, the situation is similar to Windows, meaning that one can either use Boot2Docker or Kitematic. Kitematic works similarly to Boot2Docker, employing VirtualBox to create a virtual machine with Docker, but provides a rich GUI for Docker's management.



Select a Docker image to create a new container.

kitematic
hello-world-nginx
A light-weight nginx container that demonstrates the features of Kitematic
4 | latest

official
ghost
Ghost is a free and open source blogging platform written in JavaScript
36 | latest

official
jenkins
Official Jenkins Docker image
355 | latest

official
redis
Redis is an open source key-value store that functions as a data structure server.
698 | latest

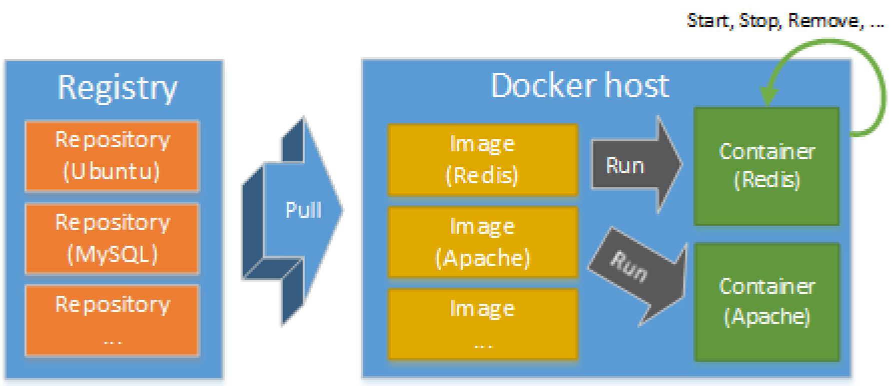
official
rethinkdb
RethinkDB is an open-source, document database that makes it easy to build and scale realtime…
24 | latest

kitematic
minecraft
The Minecraft multiplayer server allows two or more players to play Minecraft together
9 | latest

Kitematic GUI

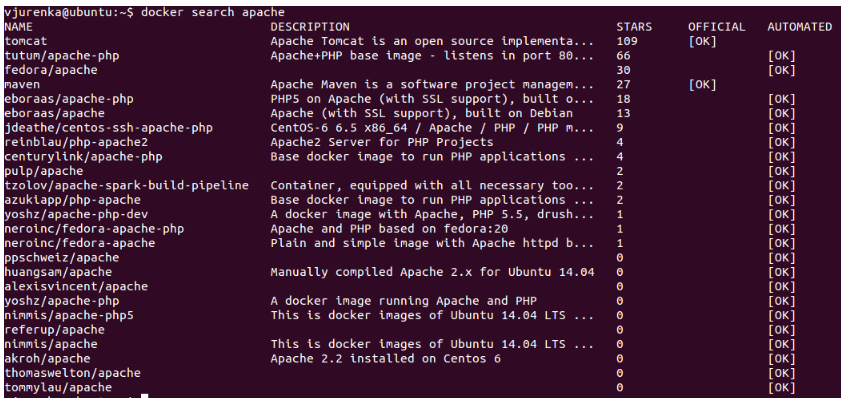
4.2 Docker basics

In this chapter, I will explain the basic workflow required to get an application running inside Docker.



Basic workflow in Docker

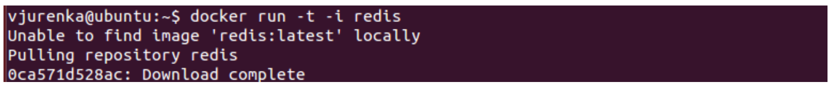
As with any other platform, the first step to run an application is to somehow get its binaries/scripts/data. For Docker, the entire application is packaged in what is called an *image* and these images are stored in *registries*. Currently, there is a default registry [20], which can be explored using a browser or alternatively use Docker's search command to list the available images.



```
vjurenka@ubuntu:~$ docker search apache
NAME                                 DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
tomcat                               Apache Tomcat is an open source implementa...   109       [OK]
tutum/apache-php                     Apache+PHP base image - listens in port 80...   66                   [OK]
fedora/apache                                                                        30                   [OK]
maven                                Apache Maven is a software project managem...   27        [OK]
eboraas/apache-php                   PHP5 on Apache (with SSL support), built o...   18                   [OK]
eboraas/apache                       Apache (with SSL support), built on Debian      13                   [OK]
jdeathe/centos-ssh-apache-php        CentOS-6 6.5 x86_64 / Apache / PHP / PHP m...   9                    [OK]
reinblau/php-apache2                 Apache2 Server for PHP Projects                 4                    [OK]
centurylink/apache-php               Base docker image to run PHP applications ...   4                    [OK]
pulp/apache                                                                          2                    [OK]
tzolov/apache-spark-build-pipeline   Container, equipped with all necessary too...   2                    [OK]
azukiapp/php-apache                  Base docker image to run PHP applications ...   2                    [OK]
yoshz/apache-php-dev                 A docker image with Apache, PHP 5.5, drush...   1                    [OK]
neroinc/fedora-apache-php            Apache and PHP based on fedora:20               1                    [OK]
neroinc/fedora-apache                Plain and simple image with Apache httpd b...   1                    [OK]
ppschweiz/apache                                                                     0                    [OK]
huangsam/apache                      Manually compiled Apache 2.x for Ubuntu 14.04   0                    [OK]
alexisvincent/apache                                                                 0                    [OK]
yoshz/apache-php                     A docker image running Apache and PHP           0                    [OK]
nimmis/apache-php5                   This is docker images of Ubuntu 14.04 LTS ...   0                    [OK]
referup/apache                                                                       0                    [OK]
nimmis/apache                        This is docker images of Ubuntu 14.04 LTS ...   0                    [OK]
akroh/apache                         Apache 2.2 installed on Centos 6                0                    [OK]
thomaswelton/apache                                                                  0                    [OK]
tommylau/apache                                                                      0                    [OK]
```

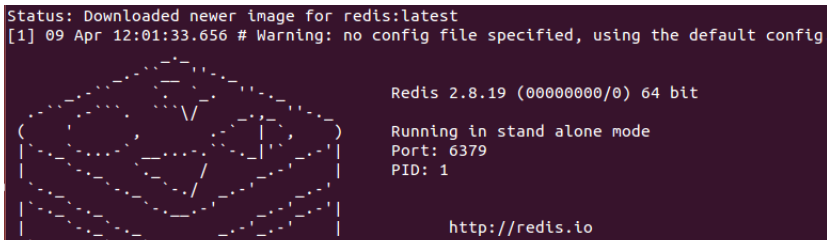
The search command actually returns *repositories*, which can contain several versions of a particular image, each with a different version *tag*, but exactly one is tagged as "latest". A docker image is also uniquely identified by an ID. It is important to notice that only the image's ID is a truly unique identifier, since even when two images may come from the same repository and have the same version tag, they may still be different, for example in the case of the tag “latest”.

The whole point of using Docker is of course, the virtualization it brings, so when an image is run, Docker creates a container and places the application from the image inside it. To run an image, one can specify either the image’s ID, repository name with tag, or only the repository name in which case, the default tag "latest" is used.



```
vjurenka@ubuntu:~$ docker run -t -i redis
Unable to find image 'redis:latest' locally
Pulling repository redis
0ca571d528ac: Download complete
```

The parameters -t -i cause Docker to attach a pseudo TTY to the container, so after all the layers are pulled, the console output from the container can be seen.



```
Status: Downloaded newer image for redis:latest
[1] 09 Apr 12:01:33.656 # Warning: no config file specified, using the default config
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 2.8.19 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in stand alone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 1
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
```

A number of other optional parameters can be passed as well, including the options to name the container, specify port mapping from the container to the host, or use custom DNS servers for the container.

The console output also says, that the image was automatically downloaded from the registry, prior to running it. If we wanted just to download the image, we could have used the command *docker pull*. One more option that may be the desired, is just to create a container from an image, but not to run it. For this purpose, there’s a *docker create* command.

4.3 Docker containers

For containers on the host, Docker provides all the basic features, such as listing the running containers, stopping or removing them. The more interesting capabilities are examining logs(*docker logs*), viewing processes inside containers(*docker top*) or connecting to a running container(*docker attach*). One of the important options is the ability run the container in the background using the daemon mode with -d switch.

When starting a container, it is possible to impose restrictions on the number of resources it can use. To force a maximum memory limit for a given container, -m parameter is used, and -c parameter to give it a CPU priority level. These parameters are actually passed to the cgroups module, that was described earlier. Mapping the container's ports to the host's ports allows the possibility of having multiple containers, in which processes run on the same port.

4.4 Docker images

Inside the Docker environment, applications are provided in the form of read-only application images. Such an image contains the application together with a reference to its parent image, so a layered architecture is created. If an image has no parent image, it is called base or root image. Common examples of root images are those of Linux distributions. The process of creating an application image for Docker is often referred to as *dockerizing*.

Building a custom image

Currently, there are two possible ways to create an application image. One can run an existing image as a container and make changes. Issuing the command *docker commit* saves the changes and it can be followed by a *docker push* command, storing the changes in a repository. The other way is to take advantage of automated building of images. This is handled by a command *docker build*, which requires the instructions to be provided in a Dockerfile. Supported instructions include copying and downloading files, executing scripts, specifying the parent image and data volumes, exposing ports and several other options. It is also possible to create a base image, either by providing a tar archive of an existing filesystem to docker import command, or specifying a special image called *scratch* (it contains an empty filesystem) as parent image.

A sequence of Dockerfile instructions to dockerize an application may look as following:

FROM repository:image *#parent image*
MAINTAINER name email *#maintainer*
ENV key value *#set environment variable*
RUN/ADD/COPY commands *#install application, copy and download data*
EXPOSE port1, port2 *#expose ports*
ENTRYPOINT /path/to/app *#path to binary*
VOLUME /path/to/data/volume *#data volume*

Here is a real world example, the Dockerfile of open-jdk8 container, with stripped comments [19]:

```
FROM buildpack-deps:sid-scm
RUN apt-get update && apt-get install -y unzip &&
rm -rf /var/lib/apt/lists/
ENV JAVA_VERSION 8u40
ENV JAVA_DEBIAN_VERSION 8u40-b27-1
ENV CA_CERTIFICATES_JAVA_VERSION 20140324
RUN apt-get update && apt-get install -y openjdk-8-jdk="
$JAVA_DEBIAN_VERSION" ca-certificates-java="
$CA_CERTIFICATES_JAVA_VERSION" && rm -rf /var/lib/apt/lists/ ×

RUN /var/lib/dpkg/info/ca-certificates-java.postinst configure
```

4.4.1 Docker registry

If a developer wants to shade his images, the easiest way is to upload them to a Docker registry. He can choose either a public registry, available at Docker Hub [20] or use a private registry. Inside the registry, the various images of an application reside in a repository.

Working with the repository is similar to using git version control system. It means commands such as push (uploading a new version) and pull (downloading the latest version) are used to for interaction with the registry.

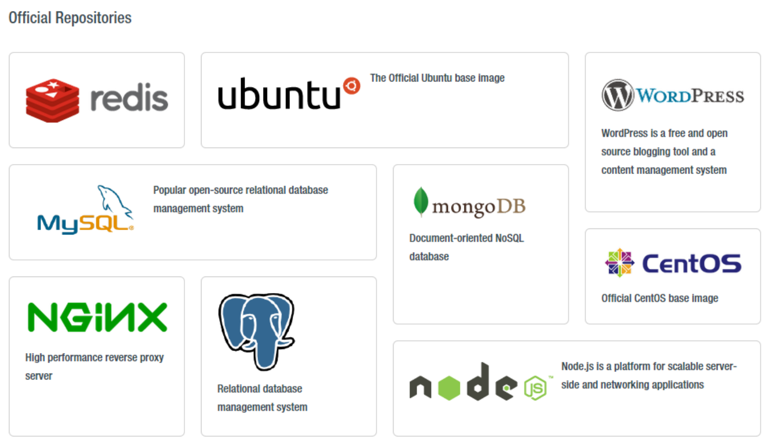
Registries also comes with several features which help in the process of creating new versions of images. Automatic builds provide a way of automatically building an application image in a repository from a github or bitbucket source and a Dockerfile, while Webhooks and Webhook chains allow to send one or more HTTP requests with specific JSON payloads. This can be used as a way to send notifications about a new update of the application image.

There are currently two versions of Docker registry project, both of them being open source. The older one(v1), written in Python, was used in Docker up to v1.6, until the stable release of the current version, Distribution (v2), which is written in Go. Distribution claims to offer faster push and pull requests, while also having more efficient implementation. Even though the versions try to maintain backwards compatibility as much as possible, some registry endpoints still vary slightly.

4.4.2 Docker Hub

The major differences between Docker Hub and a registry are, that there is exactly one instance of Docker Hub (managed by Docker Inc.) and it only handles user authentication and authorization plus it contains the checksums of images, while there may be multiple registries which store Docker images. Docker Hub currently also hosts the largest public registry, so sometimes the terms Docker Hub and public registry are interchanged.

Docker Hub's registry contains three kinds of repositories: official repositories, which contain images from vendors and Docker contributors; private repositories, where non-public images can be kept; public repositories for sharing public images. As of April 2015, Docker Hub's public registry offered over 45 000 images, which means that basically all major Linux software is already dockerized.


Official Repositories

redis

ubuntu The Official Ubuntu base image

WordPress WordPress is a free and open source blogging tool and a content management system

MySQL Popular open-source relational database management system

mongoDB Document-oriented NoSQL database

CentOS Official CentOS base image

NGINX High performance reverse proxy server

Relational database management system

node Node.js is a platform for scalable server-side and networking applications

Public registry on Docker Hub

4.5 Docker Orchestration Tools

4.5.1 Docker Swarm

Docker Swarm is a clustering tool for Docker. In other words, it takes several Docker Engines and exposes them as a single instance. All commands from the standard Docker Engine are available, but a few more were added, to setup the swarm and specify the placement of containers. There are several options how to add individual machines to the cluster. The most dynamic solution is to generate a cluster id and then pass it to *swarm join* command on each node. On the other hand, one can attach a list of nodes' IP addresses either explicitly or passing a filename to load the list from. Other options include etcd, zookeeper and consul.

Once the swarm is formed, a *swarm manage* command will start the swarm manager and regular Docker commands can now be issued for the entire swarm. For creation of new containers, a strategy needs to be picked, to decide how to assign them to nodes. The default strategy, BinPacking, tries to avoid fragmentation and thus places the new container into to node with the highest resource usage, which still has enough resources left to run it. The second currently implemented strategy is to simply pick a random node.

When using Docker Swarm, additional parameters can be passed to the *docker run* command, which can add further restrictions for picking the node to run the image. Such are restrictions are called filters and may force or exclude the selection of a specific node by name, available ports, operating system, kernel version, the node's storage type or place the container on the same node as some other, already placed container.

4.5.2 Docker Machine

The tool Docker Machine is a an example how classic and container based virtualization can coexist and complement each other. The idea is to use a dedicated virtual machine whose primary function is to run docker daemons. It is also integrates well with Docker Swarm, so a cluster can be created too.

Docker Machine then offers all the basic commands for management so one can not only create, start, restart, stop, remove and list the virtual machines, but also upgrade the docker daemon or SSH into the virtual machine. This approach offers several benefits over using only the docker daemon. First, Docker Machine is not only available on Linux, but support Windows and OSX as well. Second it can work with both local virtual machines, such as those created by VirtualBox or with cloud solutions (Microsoft Azure). Last but not least, Docker Machine aims to be a *from zero to Docker* tool, which means the tool itself takes care of common tasks such as generating SSH keys for the new machines or installing Boot2Docker as an operating system on a newly created machine.

There are currently twelve implemented drivers for virtual machine providers, offering support for example for VirtualBox, VMware vSphere, Microsoft Azure and Amazon EC2 and six more drivers are at the pull request stage. It should be noted, that at the present time, Docker machine is under active development and it's usage in production is thus not recommended. However a peek on the upcoming features can be found in the projects roadmap [**?**, DockerRoadMap]

4.5.3 Docker Compose

It is often the case, that complex software is composed of several components, for example a webserver and a database. It would be a repetitive and possibly error prone task, to install every single component manually, so a project called Fig was created. In February 2015, Fig was deprecated in favour of Docker Compose, which is based on Fig's code base.

The main idea is the ability to link several containers together, allowing to compose a complex application from several images. It uses a yaml configuration, which also allows to specify the settings for exposing ports, passing environment variables and mounting data volumes. A configuration may also extend an existing one, so for example an application can have a simple common configuration plus two specialized ones, one for development and the other one for production. Docker compose works well with Docker Swam, where linked containers are scheduled on the same host.

4.6 Family of Docker APIs

Docker platform comes with three separate APIs, to allow interaction with third party applications: Docker Hub API, Docker Registry API and Docker Remote API. All of the APIs work over HTTP and conform to the REST style.

Docker Hub API provides services for users, such as registering, updating and logging in and allows to create and delete repositories, There is currently no client library for this API. Docker Registry API handles putting and getting images and also exposes a search function. An AngularJS client library is currently the only one available [22]. Docker Remote API is covered separately in the next chapter.

4.6.1 Docker Remote API

For a company managing hundreds or thousands of virtualized environments an automated way of management is always helpful. Docker thus comes with an API for remote interaction with containers - Docker Remote API. Docker Swarm also uses the same API, but exposes additional information, such as IP or name of the node which runs a specific container. However, not all endpoints are implemented yet. The API could be used as simply as any REST API by utilising curl, or with the help of a client libraries. These are available in many languages: C++, C#, Java, Python, Ruby, PHP, Go and several more.

By default, the Docker daemon listens on a Unix socket and is thus available only locally, however a TCP port may also be specified, by using the option -H when starting docker daemon or creating an environment variable with name DOCKER_HOST. Once the Docker runs on a public TCP port, Docker Remote API may be used to manage containers and images.

4.7 Docker vs other container technologies

After exploring the major features offered by Docker a comparison with LXC and OpenVZ can take place. It is clear, that Docker is much higher level tool, which is intended to simplify the containers management while adding additional features, such as its API.

From the architectural point of view, the probably most significant difference is between Docker images and LXC / OpenVZ templates. The images are reusable, thanks to the parent-child relationship, both versionable and easily shareable, because of Docker Hub and registries. Images are also easily buildable and deployable.

Several new projects built on top of Docker will be introduced in the next section. All of them currently support Docker, some even require it to be the containers manager. This put Docker far ahead of other container technologies ,as the range of available tools is much larger, while still steadily growing.

5 Implementation

The second part of this thesis focusing on practical experience with Docker. Firstly, I will use Docker Remote API with a corresponding client library to transfer a file between two Docker containers, each running on a different host. The other task is to enhance Docker with a new command, *docker update* which simplifies detection of outdated images.

5.1 Using Docker Remote API

Docker-java

Currently, there are three Java client libraries for Docker's remote API. All these are open source projects, with their code available on github. For the purposes of our thesis I have picked docker-java [33], simply because it is the most starred project from those three.

On the positive side, the library is easy to setup, supports SSL, tries to stay up to date with Docker API and the code has very good test coverage. The configuration is especially well done, since it's possible to provide it either programmatically, via property file or using environment variables.

What I think could be improved, is the sparse documentation. Completing the project's wiki or using more comments in sources, would help in cases where the usage is not intuitive and the only remaining option is to study tests' sources. It is mentioned that only a subset of the API is provided, but it is not clear what exactly is missing, although all the important features seem to be supported.

Setup

I have decided to demonstrate the API's power by programmatically transferring a file over SCP using Docker-java. My setup consisted of the following machines:

- Ubuntu 14.10, virtual machine created using VMWare Player
  - Running Docker with Ubuntu 14.04 container (A)

    * Destination for file

- Fedora 21, virtual machine created using VMWare Player
  - Running Docker with Ubuntu 14.04 container (B)
    * File to copy (source file)
- Windows 7, physical machine (C)
  - Running VMWare Player with Linux guests (listed above)
  - Installed JDK 1.7

Results

The code required to provide the transfer is very short, once a connection to the container is established, the SCP command which does the actual transfer is sent over to be executed. The file transfer proved to be successful and happened swiftly, as was expected. Using the cat command for printing the file contents it was immediately clean that the new file on (A) is indeed a copy of the original one from (B).

5.2 Docker update

In this thesis, I have implemented a new Docker command, *update*. The commands checks (and pulls) newer versions of images installed on the host. Such command was request several times on Docker's github [34] and since I found it not only to be a great way to get a deeper understanding of Docker's core, I have decided to come up with an implementation.

The update command can be run from the command line, by issuing *docker run*, or from Docker's remote API's endpoint "images/update", by making a POST request. One additional flag is supported, *–dry-run* or *-d* which when present, only causes the command to return whether the images on the host are up to date or not, without pulling the newer versions.

Developing Docker

Docker contribution workflow, which is in detail explained in the official guide [35] , specifies the necessary tools, such as git, make and docker. The guide covers every basic step from creating/claiming an issue, though installing, compiling and testing up to making pull requests. As a great communication tool and help for new Docker developers an IRC channel #docker-dev is publicly available.

Except for the compilation time, which is mediocre (about half a minute), development is quite fast, since it simply involves copying the new binary and restarting the docker service, which can be all handled by an utility script. What is more complicated is testing, since running the entire test suite can take several hours.

Docker's code itself is mostly well structured and formatted, but lacks enough comments. This is something that I feel should be improved, since for example, it is often hard to tell a variable's type, considering Go's dynamic type inference. What gives a very good impression is Docker's heavy error checking and frequent debug logging. Another positive is the emphasis placed on project's modularity and loose coupling. I have also mentioned the tests, which are present .

Our IDE choice was LiteIDE [36], a lightweight open source IDE for Go. It is easy and intuitive to install and use, plus it offers a good search mechanism. The not so outstanding features are code navigation, which seems to be working about half the time and code completion, which didn't work in our installation at all.

Writing a custom Docker command

Implementing a custom docker command usually requires the following steps:

- Implement the server side function, which does the core part
- Create a REST endpoint which runs the server-side function on the docker daemon and bind it to particular URL and HTTP method
- Add the command itself, which parses the console input, creates a request on the REST endpoint and parses and prints the

output

- Register the command in the command line interface
- Write the command help, specifying the usage and arguments
- Write unit tests for added functions, and an integration test for the command

Summary

This command could be very beneficial for Docker to have as system administrators will have a quick way of checking whether all of their images are up to date. It would be also possible to build a notification service, utilising the command, which could periodically check and possibly pull updates for the images present in the system.

6 Conclusion

In my thesis, I have described both approaches to virtualization - virtual machines and containers. In the case of virtual machines, the principles were explored and various tools were described. It can be stated that virtual machines have a long tradition, the technology is widely used and mature and furthermore, the years of production use have caused them to be thoroughly tested. Nevertheless, virtual machines suffer from several drawbacks. They are often difficult to set up and maintain, can take a lot of space on disc and even though they are directly supported by the hardware, a performance hit still occurs.

Container based virtualization works by sharing the operating system's kernel, eliminating these kind of problems. However it important to note, that containers are not able to fully replace virtual machines yet. Running an entirely different guest operating system is a frequently used feature of virtual machines, which the containers don't offer. Additionally, as the usage of containers spreads, security bugs are being discovered in the underlying technologies, and even if they are being fixed quickly, it will take some time before containers will become as much trusted and production tested as virtual machines. Still, large companies have decided that containers are indeed what could dramatically improve performance and accessibility of their clouds and started adopting them.

Docker is a project which brings the containers closer to developers and system administrators. It comes with an easy to use interface for container management and provides a healthy ecosystem for sharing the containerized applications - images. Docker is on its way to become a standard for handling containers, while introducing higher level tools for both applications and users. Rapidly evolving, Docker is quickly adding features, while trying to stay in contact with its user base, in order to make sure containers will once earn the same reputation as virtual machines. Moreover, dozens of projects are being created on top of Docker, ranging from small extensions to entire cloud operating systems. The following months and possibly years will show, which ones will dominate the market and which ones will fade into history.

The practical part of this thesis starts with demonstration of Docker's API and its public libraries, showing that interacting with Docker is almost as easy for programs as it is for users. Writing a simple Java application, which transferred a file using SCP was indeed a task, which I've successfully managed to complete.

A new command, Docker update was the final focus of this thesis. This commands allows the Docker client to check whether the local images are all up to date, while possibly updating them. Its implementation served as an introduction to Docker's internals and its development process. The final code was then submitted in the form of pull request directly to Docker's github repository, where it passed all the automated tests and is currently waiting for a decision whether to be merged in Docker's official codebase.

7 Appendix

7.1 Attached files

Files containing source code for both tasks from the implementation part are attached to the electronic version of this thesis. The Docker Remote API demonstration can be found in the directory *file-copy_demo*, which includes a pom.xml, containing the dependency on docker-java and a src folder with the actual source code. The implementation of Docker update command is located in *docker_udpate*, which contains the modified source code of Docker 1.7. A readme_jurenka.txt contains the relevant information, regarding the compilation and/or running of this modified docker binary.

Bibliography

[1] Navin Sabharwal, Bibin W. *Hands on Docker*, 2015.

[2] Shrikrishna Holla. *Orchestrating Docker*, 2015.

[3] Gerald J. Popek and Robert P. Goldberg. *Formal Requirements for Virtualizable Third Generation Architectures*, 1974.

[4] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B:System Programming Guide, Part 2*, 2011.

[5] *Secure Virtual Machine Architecture Reference Manual*, 2005.

[6] *http://www.xenproject.org/users/why-the-xen-project.html*, cit. 23.4.2015.

[7] *http://en.wikipedia.org/wiki/Libvirt#/media/File:Libvirt_support.svg*, cit. 12.5.2015

[8] *http://virt-manager.org/wp-content/uploads/2013/04/virt-manager-vm-list.png*, cit. 12.5.2015

[9] *http://cockpit-project.org/images/screenshot-docker.png*, cit. 12.5.2015

[10] *http://infoslack.com/images/etcd-cluster.png*, cit. 12.5.2015

[11] *https://github.com/docker/libcontainer*, cit. 17.3.2015.

[12] *https://docs.docker.com/terms/images/docker-filesystems-multilayer.png*, cit. 10.5.2015

[13] *http://lwn.net/Articles/531114/*, cit. 10.1.2015.

[14] *http://blog.docker.com/2014/10/docker-microsoft-partner-distributed-applications/*, cit. 11.1.2015

[15] *http://techcrunch.com/2014/01/21/docker-raises-15m-for-popular-open-source-platform-designed-for-developers-to-build-apps-in-the-cloud/*, cit. 12.1.2015.

[16] *http://venturebeat.com/2014/09/16/docker-funding/*, cit. 12.1.2015.

[17] *https://gigaom.com/2014/08/06/the-400-million-container-company-docker-closes-in-on-funding-round-of-over-40-million/*, cit. 12.1.2015

[18] *https://chocolatey.org/packages/docker*

[19] *https://github.com/docker-library/java/blob/master/openjdk-8-jdk/Dockerfile*, cit. 6.4.2015.

[20] *https://registry.hub.docker.com/*

[21] *https://github.com/docker/machine/blob/master/ROADMAP.md*

[22] *https://github.com/kwk/docker-registry-frontend*

[23] *https://titanous.com/posts/docker-insecurity*, cit. 21.4.2015.

[24] *http://lwn.net/Articles/626665/*, cit. 21.4.2015.

[25] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio. *An Updated Performance Comparison of Virtual Machines and Linux Containers*, 2014.

[26] *http://googlecloudplatform.blogspot.sk/2014/06/an-update-on-container-support-on-google-cloud-platform.html*, 10.6.2014

[27] *https://gigaom.com/2014/08/18/google-wants-to-show-the-world-how-sexy-cluster-management-really-is/*, cit 24.4.2015

[28] *https://mesosphere.com/blog/2015/04/23/apple-details-j-a-r-v-i-s-the-mesos-framework-that-runs-siri/*, cit 24.4.2015

[29] *http://www.ubuntu.com/cloud/tools/lxd*, cit 26.4.2015

[30] *https://coreos.com/blog/rocket/*, cit 26.4.2015

[31] `https://github.com/docker/docker/commit/0db56e6c519b19ec16c6fbd12e3cee7dfa6018c5`, cit 26.4.2015

[32] *https://github.com/appc/spec/blob/master/SPEC.md,* cit 26.4.2015

[33] *https://github.com/docker-java/docker-java*

[34] *https://github.com/docker/docker/issues/4239*

[35] https://docs.docker.com/project/who-written-for/

[36] *https://github.com/visualfc/liteide*