Department of Electrical And Computer Engineering

University of Thessaly

MSc: "Science and Technology of ECE"

Diploma Thesis:

Enhancing the NITOS Testbed virtualization capabilities by using Docker Containers

by Panagiotis Theodosiou

Supervisor: Athanasios Korakis, Assistant Professor

October 2018

Ευχαριστίες

Η συγγραφή της πτυχιακής εργασίας σηματοδοτεί το τέλος του μεταπτυχιακού προγράμματος σπουδών και παράλληλα μιας διαδρομής που περιείχε τα πάντα: γνώσεις, εξετάσεις, αγωνίες και δημιουργικούς προβληματισμούς. Αρωγοί σε όλη αυτή τη μοναδική διαδρομή στάθηκαν πέρα από την οικογένειά μου και οι καθηγητές μου. Θα ήθελα όμως ξεχωριστά να ευχαριστήσω τον καθηγητή κ. Κοράκη Αθανάσιο για την βοήθεια και στήριξη που μου παρείχε κατά τη διάρκεια της παρούσας μεταπτυχιακής εργασίας. Τέλος, δεν θα ήθελα να λησμονήσω να αναφέρω τα ονόματα των Κατσαλή Κώστα, Ιωάννη Ηγούμενο, Χάρη Νιαβή και Άρη Δαδούκη που με τις γνώσεις και την θετική τους στάση με βοήθησαν να φέρω σε πέρας την υλοποίησή μου.

Στη μνήμη του πατέρα μου

Περίληψη

Οι εικονικές μηχανές (VMs), όπως έχουν εξελιχθεί, αποτελούν αναπόσπαστο και πολύτιμο κρίκο των σύγχρονων cluster, grid και cloud συστημάτων. Ευκολία διαχείρισης, ευελιξία και υψηλή χρησιμοποίηση πόρων είναι μερικά μόνο από τα αδιαμφισβήτητα πλεονεκτήματα που προσφέρει η εικονικοποίηση (virtualization), σήμερα. Σε ένα κόσμο πληροφοριακών συστημάτων που η τεχνολογία εικονοποίησης φαίνεται να κυριαρχεί, πόσος χώρος υπάρχει για εναλλακτικές λύσεις; Η αναδυόμενη τεχνολογία των Container δείχνει ότι θα μπορούσε να προσφέρει, μέχρι ένα βαθμό, παρόμοιες ή συμπληρωματικές υπηρεσίες. Στην παρούσα εργασία εξετάζουμε τις δυνατότητες και τους περιορισμούς των Container, ενσωματώνοντάς την νέα αυτή τεχνολογία στις υποδομές του Nitos Testbed. Παράλληλα επισημαίνονται τα πλεονεκτήματα και τα όρια, όχι για λόγους σύγκρισης αλλά περισσότερο για πληρέστερη κατανόηση αυτής της νέας πολλά υποσχόμενης τεχνολογίας.

Λέξεις κλειδιά: Virtual Machine, Linux container, Docker

Abstract

Recent reincarnation of virtual machines (VMs) presented a great opportunity for parallel, cluster, grid, cloud, and distributed computing. Whether the motivation is higher utilization, reduced management, or business agility, virtualization technology offers compelling possibilities. In an IT world dominated by virtualization computing, is there space for alternative concepts? Answer comes from the emerging Container technology that could provide similar or supplementary services, to the already established virtualization practice. In this paper, we explore container's potentials and boundaries by enhancing Nitos Testbed with Docker containers capabilities. Advantages and limitations are pinpointed, not for comparison purposes but for providing a clearer understanding of this new promising technology.

Keywords: Virtual Machine, Linux container, Docker

5

Πίνακας περιεχομένων

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

6

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

7

Intro

1   Introduction

1.1   Cloud computing

Cloud computing is a type of Internet-based computing that provides shared computer processing resources and data to computers and other devices on demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources (e.g., computer networks, servers, storage, applications and services), which can be rapidly provisioned and released with minimal management effort. Cloud computing relies on sharing of resources to achieve coherence and economy of scale, similar to a utility (like the electricity grid) over an electricity network.

Cloud computing exhibits the following key characteristics [1]:

- **Agility** for organizations may be improved, as cloud computing may increase users' flexibility with re-provisioning, adding, or expanding technological infrastructure resources.

- **Device and location independence** enables users to access systems regardless of their location or what device they use.

- **Maintenance** of cloud computing applications is easier, they do not need to be installed on each user's computer and can be accessed from different places

- **Multitenancy** enables sharing of resources and costs across a large pool of users thus

- **Performance** is continuously monitored, and consistent and loosely coupled architectures are constructed using web services as the system interface.

- **Increase of Productivity** allowing multiple access on the same data simultaneously.

- **Resource pooling:** a multi-tenant model with different physical and virtual resources dynamically assigned and reassigned according to demand.

- **Reliability** improves with the use of multiple redundant sites

- **Scalability** and elasticity via dynamic ("on-demand") provisioning of resources on a fine-grained, self-service basis in near real-time

- **Security** can improve due to centralization of data, increased security-focused resources.

1.2   Hypervisor Technologies

Hypervisor, or **virtual machine monitor** (**VMM**) as it is usually encountered in computer science, is a piece of computer software, firmware or hardware that creates, manages and runs virtual machines. A computer on which a hypervisor is running one or more virtual machines is defined as a *host* machine. Each virtual machine running on host is called *guest* machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of different operating systems may share the same virtualized hardware resources. The Hypervisor actually provides a virtual working platform that services and manages the requests for resources of the different operating systems running above it. Depending on the environment in which the hypervisor is installed hypervisors can be classified into two categories.

1.2.1   Type-1: native or bare-metal hypervisors

These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems. For this reason, they are sometimes called bare metal hypervisors. A guest operating system runs as a process on the host. The first hypervisors, which IBM developed in the 1960s, were native hypervisors, nowadays, software products such as Vmware vSphere, Citrix XenServer, Oracle VM Server and Microsoft Hyper V are representative of the specific category. It's inherent property to communicate directly with the hardware layers stands as the major advantage of the particular implementation.

1.2.2   Type-2 or hosted hypervisors

These hypervisors run on a conventional operating system just as other computer programs do. Type-2 hypervisors abstract guest operating systems from the host operating system. VMware Workstation, VMware Player, VirtualBox and QEMU are commercial software examples of type-2 hypervisors.
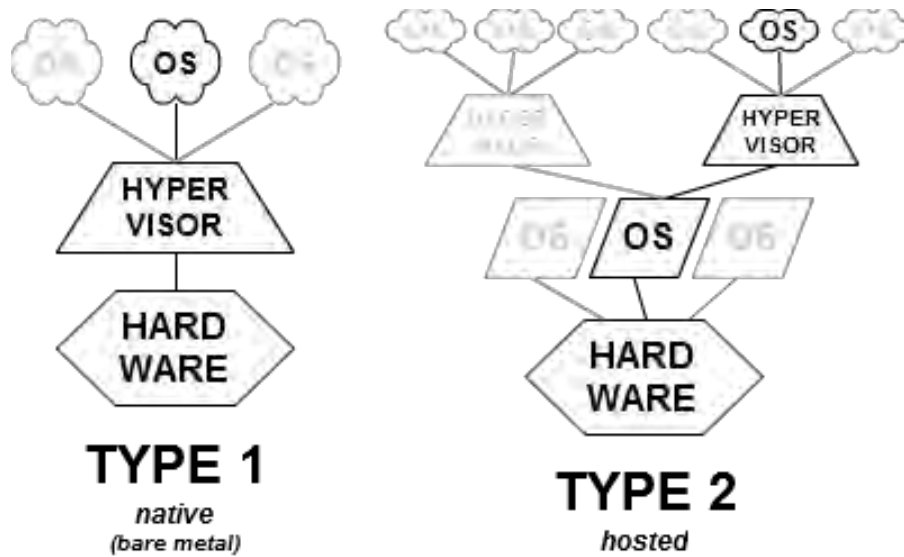
Figure 1 : Native vs Hosted Hypervisor

The figure above visualizes the concepts of the two technologies. The importance and the role of each category in the industry is underlined by the continuous research and development of leading software companies on both types of hypervisor implementation.

1.3 Linux Containers (Introduction and Comparison with Hypervisors)

Having explained the concept of hypervisors and virtual machines, allows us to introduce the idea of **Linux Containers**. Term Linux Container describes an environment similar to a virtual machine but without the payload of executing a new kernel. Originally, the idea of a container was born as a need to address the problem of developing, deploying and conveying different application with specific configurations between different targets of development environments such as production and staging servers, virtual machines or even shared hosting. Linux containers, using a high level approach, are commonly characterized as lightweight virtual machines. Typically, Linux Container is an operating-system-level virtualization environment for running multiple isolated Linux systems (Containers) on a single Linux control host. Linux containers make use of kernel's cgroups functionality and support for isolated namespaces to provide an isolated environment for applications.

Some of the great advantages of the specific concept lie on the fact that containers are executed directly on the kernel without the need of an intermediate software layer, resulting in astounding performance. Efficiency is guaranteed since there is almost any overhead in the individual parts of the architecture (CPU native performance, only small amount of memory is used for accounting and very small overhead to network performance). Creating and running containers may require few seconds or even milliseconds in some cases emphasizing the effectiveness of the technology [2]. The ability and flexibility to combine (contain) different applications or even whole systems

contributes to the growing popularity this concept was given. Comparison of linux containers and hypervisor technology is presented clearly in the picture below.
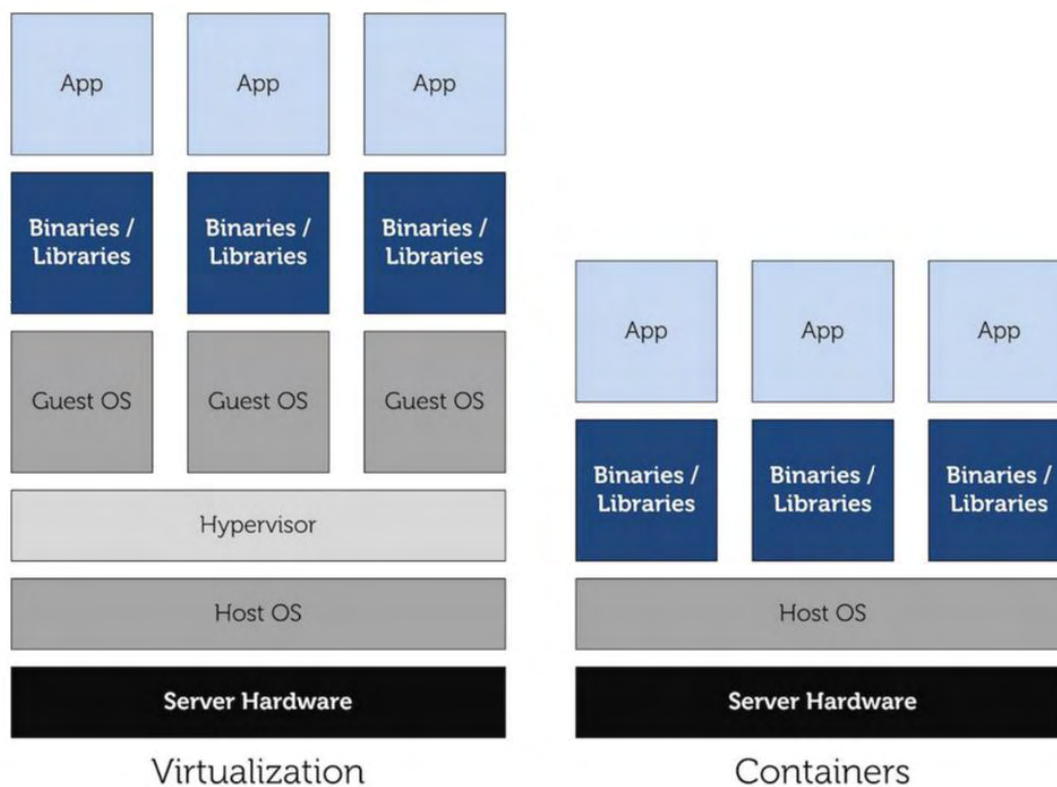
Figure 2 : Hypervisor vs Linux Container

Part I: Theory

2 Linux Containers

As already mentioned **Linux Containers (LXC)** allow the usage of Operating-system-level virtualization technology, by creating containers that resemble complete isolated Linux virtual machines on the physical Linux machine, sharing the kernel with the virtual portion of the system. A container is a virtual environment, with its own process and network space. LXC makes use of Linux kernel Control Groups and Namespaces to provide the isolation. Containers have their own view of the OS, the process ID space, the file system structure, and the network's interfaces. Since they use kernel features, and there's no emulation of hardware at all, the impact on performance is minimal. Figure 3 illustrates the concept of a linux container.
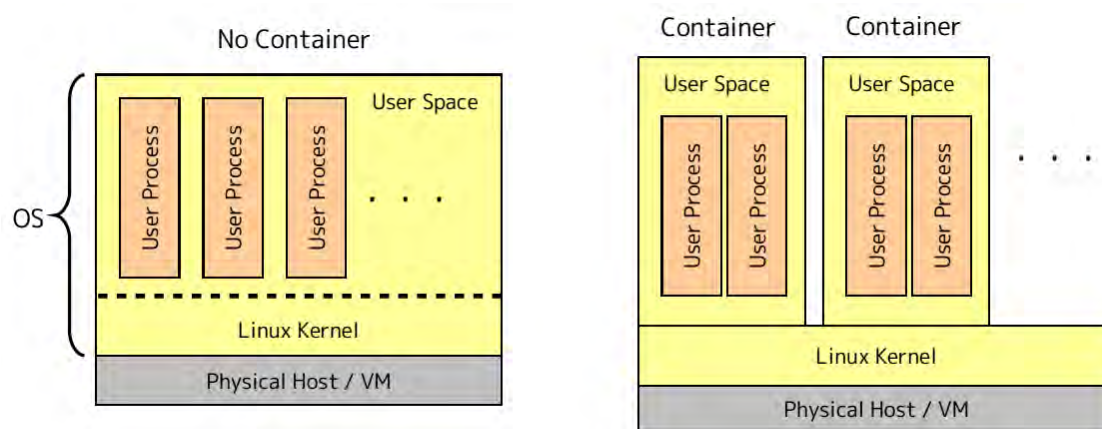
Figure 3 : Concept of Linux Container

2.1 Linux Containers Architecture

Linux Containers are implemented using three modern core technologies:

- **Control Groups (Cgroups)** for resource management
- **Namespaces** for process isolation and
- **Linux security modules** for security, enabling secure multi-tenancy and reducing the potential for security exploits.

There is also a management interface that forms a higher layer which interacts with the aforementioned kernel components and provides tools for the construction and

13

management of containers. As an example, the following scheme illustrates the architecture of Linux Containers in Red Hat Enterprise Linux 7 [3]:
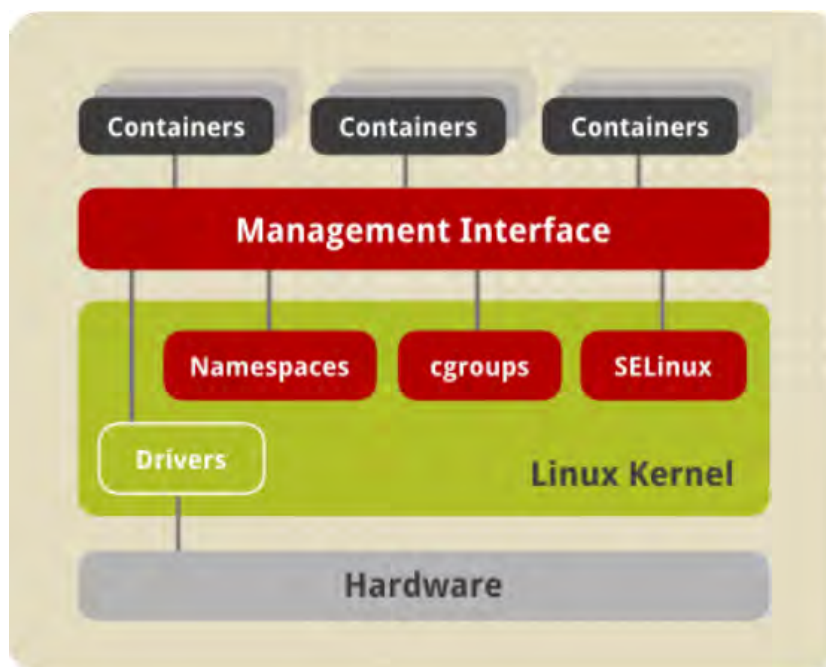
Figure 4 : Linux Containers in Red Hat Enterprise Linux 7

2.1.1 Namespaces

The kernel provides process isolation by creating separate namespaces for containers. **Namespaces**, a feature introduced between kernel 2.6.15 -2.6.26, enable creating an abstraction of a particular global system resource and make it appear as a separated instance to processes within a namespace [4]. Consequently, several containers can use the same resource simultaneously without creating a conflict.

Figure 5 : Linux namespaces conceptual view

There are **several types of namespaces:**

2.1.1.1 *Mount namespaces*

Mount namespaces isolate the set of file system mount points seen by a group of processes so that processes in different mount namespaces can have different views of the file system hierarchy. With mount namespaces, the `mount()` and `umount()` system calls cease to operate on a global set of mount points (visible to all processes) and instead perform operations that affect just the mount namespace associated with the container process. For example, each container can have its own `/tmp` or `/var` directory or even have an entirely different user space. This characteristic is typically used with `chroot()` or `pivot_root()` for effective file system isolation.

2.1.1.2 *UTS namespaces*

UTS namespaces isolate two system identifiers – nodename and domainname, returned by the `uname()` system call. This allows each container to have its own hostname and NIS domain name, which is useful for initialization and configuration scripts based on these names. Processes in a namespace can change UTS values but changes will only be reflected in the child namespace. Running hostname command on the container can confirm the above assertion.

2.1.1.3 *IPC namespaces*

IPC namespaces isolate certain interprocess communication (IPC) resources, such as System V IP objects and POSIX message queues. This means that two containers can create shared memory segments and semaphores with the same name, but are not able to interact with other containers memory segment or shared memory. Parent namespace connectivity is attained by the use of mechanisms like memory polling, signals, sockets (if there is no Network namespace), files / file descriptors (if there is no mount namespace) and events over pipe pair.

2.1.1.4 *PID namespaces*

PID namespaces introduced a new feature of Linux kernel that of multiple "nested" process trees. This enables each process tree to have an entirely isolated set of processes, ensuring that processes belonging to one process tree cannot inspect or kill or even know of the existence of processes in other sibling or parent process trees. So, from a container point of view you can only monitor processes running inside this container (limited visibility). In other words, the container is only aware of its native processes and cannot be aware of other processes running in different parts of the system.

On the other hand, the host operating system has a complete view of all processes running inside the container, but assigns them different PID numbers. This way, there is no PID conflict between namespaces which permits migrating namespace processes between hosts while keeping same PID. Namespaces allow processes in different containers to have the same PID, so each container can have its own init (PID1) process that manages various system initialization tasks as well as containers life cycle. Additionally, each container has its unique `/proc` directory.
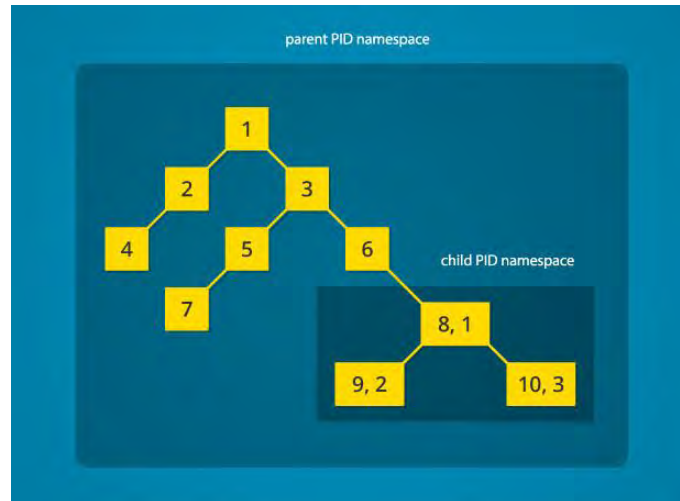
Figure 6 : Pid namespace visualization

2.1.1.5 *Network namespaces*

Network namespaces provide isolation of network controllers, system resources associated with networking, firewall and routing tables. This allows container to use separate virtual network stack, loopback device and process space, providing the capability of adding virtual or real devices to the container, assigning them their own IP addresses, and even full iptables rules. The different network settings can be revealed by executing the `ip addr` command on the host and inside the container. Connectivity between namespaces is sustained by the use of veth pairs, where each part is moved inside each namespace and configured accordingly, this scheme works like a pipe between the namespaces.
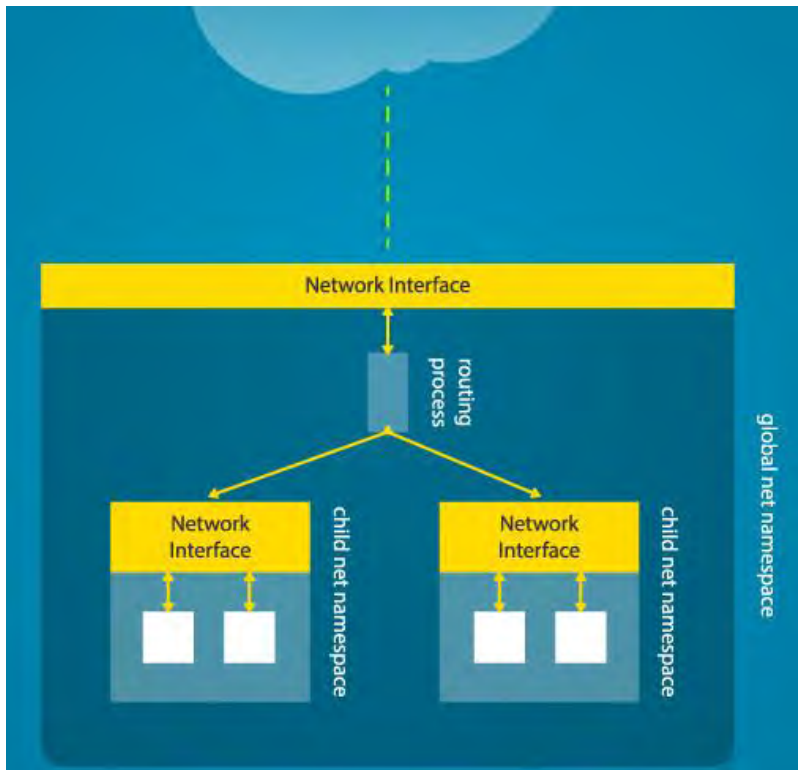
Figure 7 : Network namespace

2.1.1.6 *User namespaces*

User namespaces are similar to PID namespaces they allow mapping of UID / GID from outside the container to UID / GID inside the container. They also permit non-root users to launch Linux containers. There is a significant amount of work in progress for this specific feature, eventually user namespaces will mitigate many perceived Linux container security concerns.

Namespaces are materialized by pseudo-files in `/proc/<pid>/ns`. **Three system calls are used for namespaces**

- `clone():` creates a new process and a new namespace, the process is attached

  to the new namespace.
- `unshare():` does not create a new process; creates a new namespace and attaches the current process to it.
- `setns():` a new system call was added, for joining an existing namespace

When the last process of a namespace exits, it is destroyed but can also be preserved by bind-mounting the pseudo-file. Summing up, a set of namespaces is created for each

container and all processes are executed inside the specific namespace set. All processes in the container have an isolated view of resources. Linux container is realized with integrating all the above namespace features. There are multiple container management tools such as lxctools, libvirt and docker, discussed later in the current paper, that use different parts of these features and their main role is to simplify the tedious work needed to organize and coordinate all these parameters.
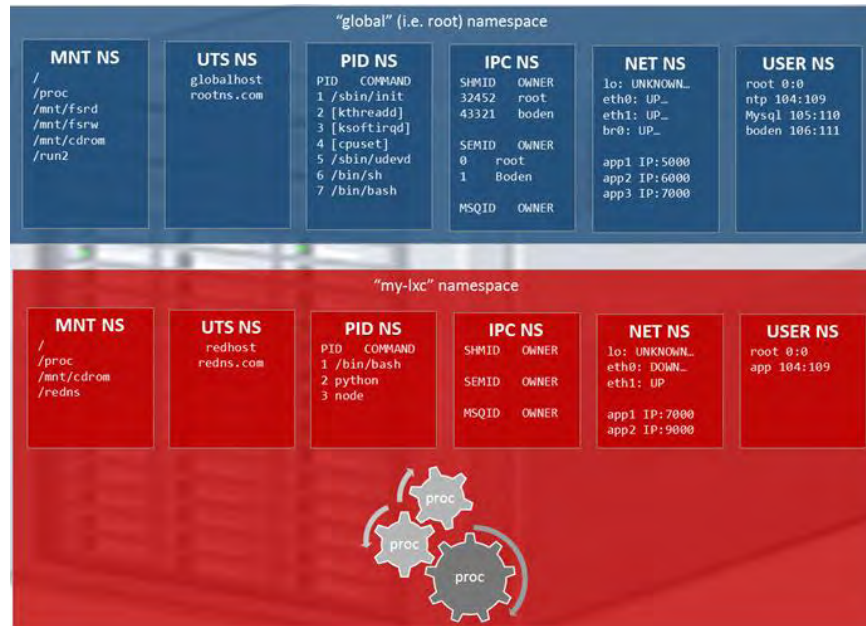
Figure 8 : Linux container namespace

2.1.2 Control Groups (cgroups)

Control Groups (**cgroups**) is a Linux kernel feature that provides a mechanism for easily managing and monitoring system resources, by partitioning resource usage such as CPU, memory, disk I/O, network bandwidth, etc into groups and then assigning tasks (processes) to those groups, thus providing guaranteed allocation of needful resources to an application. Cgroups can be monitored, reconfigured or even denied access to certain resources on a running system. All in all, cgroups provide fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources, thus increasing overall efficiency [5].

Cgroups main features can be summarized in the following points:

- **Access:** which devices can be used per cgroup
- **Resource limiting:** memory, CPU, device accessibility, block I/O, etc.
- **Prioritization:** assign different amount of CPU, memory, etc to some groups

- **Accounting:** measure resource usage per cgroup
- **Control:** freezing groups or check pointing and restarting
- **Injection:** packet tagging

Cgroups are organized hierarchically, like processes, and child cgroups inherit some of the attributes of their parents. However, there are differences between the two models.

2.1.2.1 *The Linux Process Model*

All processes on a Linux system are child processes of a common parent: the init process, which is executed by the kernel at boot time and starts other processes (which may in turn start child processes of their own). Because all processes descend from a single parent, the Linux process model is a single hierarchy, or tree. Additionally, every Linux process except init inherits the environment (such as the PATH variable) □and certain other attributes (such as open file descriptors) of its parent process.

2.1.2.2 *The Cgroup Model*

Cgroups are similar to processes in that, they are hierarchical, and child cgroups inherit certain attributes from their parent cgroup.

The fundamental difference is that many different hierarchies of cgroups can exist simultaneously on a system. If the Linux process model is a single tree of processes, then the cgroup model is one or more separate, unconnected trees of tasks.

Cgroup functionality is exposed by the term "resource controllers" (also called "subsystems"). A subsystem represents a single resource, such as CPU time or memory. Multiple separate hierarchies of cgroups are necessary because each hierarchy is attached to one or more subsystems. These subsystems are mounted on the File System with the root cgroup being the top-level subsystem mount and all other child directories per cgroup being mounted under this one. For clarification reasons, the available subsystems of the Red Hat Enterprise Linux, are listed below.

- **blkio:** this subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, or USB).
- **Cpu:** this subsystem uses the scheduler to provide cgroup tasks access to the CPU.
- **Cpuacct:** this subsystem generates automatic reports on CPU resources used by tasks in a cgroup.
- **Cpuset:** this subsystem assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
- **devices:** this subsystem allows or denies access to devices by tasks in a cgroup.
- **Freezer**: this subsystem suspends or resumes tasks in a cgroup.
- **Memory:** this subsystem sets limits on memory use by tasks in a cgroup and generates automatic reports on memory resources used by those tasks.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

20

- **net_cls:** this subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.
- **net_prio:** this subsystem provides a way to dynamically set the priority of network traffic per network interface.

One cgroup is created per container, Linux pseudo FS is the interface to cgroups, and various libraries have been developed for programmers to interface with this pseudo file system. Processes assigned to a cgroup are placed inside the tasks file. The following picture illustrates some of the cgroups main features described above and summarizes how these characteristics are combined to enforce resource limitations to Linux containers.
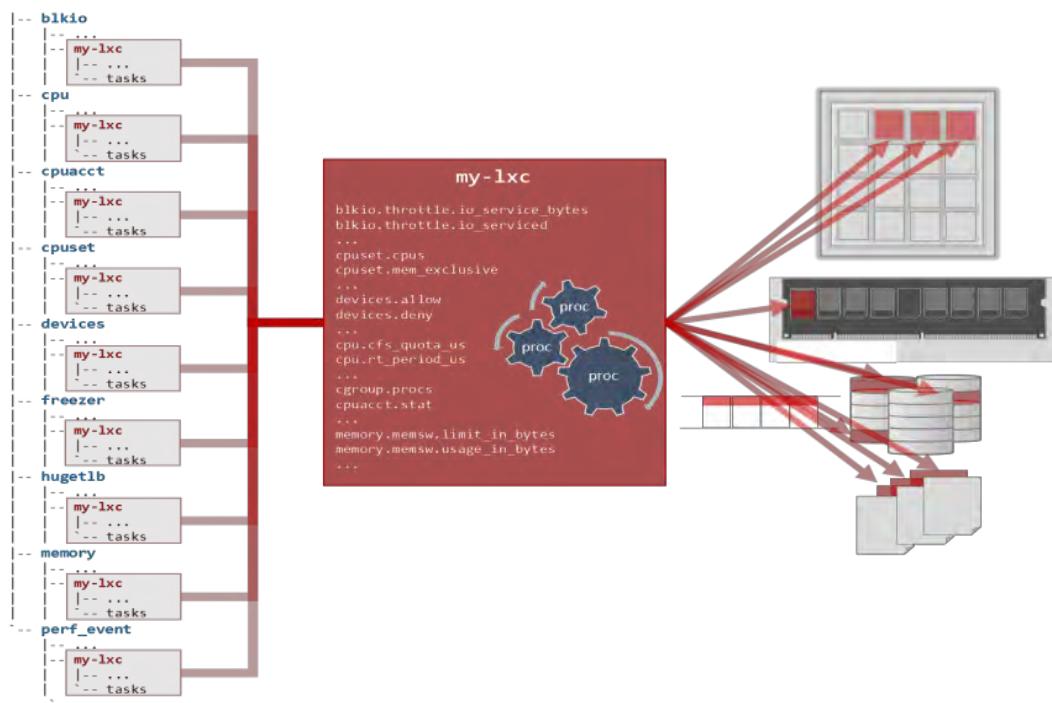
Figure 9 : Cgroup, Linux container realization

2.1.3 Linux Capabilities

One aspect of security is user privileges. UNIX-style user privileges come in two varieties, user and root. Regular users are relatively powerless; they cannot modify any process or file but their own. Access to hardware and most network specifications is also denied. Root, on the other hand, has full control of anything inside the operating system, from modifying all processes and files to having unrestricted network and hardware access. Very soon, the need for a middle ground solution aroused. It is a common occasion for a utility to need special privileges to perform its function, but unquestionably full root access cannot be granted to a user process. The most common example of this situation is ping

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

21

command which needs root access to send and receive ICMP messages. Fortunately, such a middle ground now exists, and it's called POSIX capabilities. **Capabilities divide system access into logical groups that may be individually granted to, or removed from, different processes**. Capabilities allow system administrators to fine-tune what a process is allowed to do, which may help them significantly reduce security risks to their system.

2.1.3.1 *POSIX capabilities*

A process has three sets of bitmaps called the inheritable (I), permitted (P), and effective (E) capabilities. Each capability is implemented as a bit in each of these bitmaps that is either set or unset. When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process is 0 as is normally done). The permitted set of the process indicates the capabilities the process can use. The process can have capabilities set in the permitted set that are not in the effective set. This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted exists so that processes can "bracket" operations that need privilege. The inheritable capabilities are the capabilities of the current process that should be inherited by a program executed by the current process. The permitted set of a process is masked against the inheritable set during exec(). Nothing special happens during fork() or clone(). Child processes and threads are given an exact copy of the capabilities of the parent process. The implementation in Linux stopped at this point, whereas POSIX Capabilities require the addition of capability sets to files too, to replace the SUID flag (at least for executables)

2.1.3.2 *Containers, capabilities and security*

Linux capabilities are thoroughly enforced inside containers, limiting root's privileges within a container and thus making an intruder that managed to grand root access inside a container, powerless and incapable to escalate to host or cause serious damage to the system. Container technologies (such as LXC and Docker) support the addition or removal of capabilities, this way adjusting the security policies of a container to the needs of the creator.

2.1.4 Chroot and pivot_root

Chroot is an operation that changes the apparent root directory for the current running process and their children. A program that is run in such a modified environment cannot access files and commands outside that environmental directory tree. This modified environment is called a `chroot` jail. Using chroot can be escaped given the proper capabilities, thus `pivot_root` is often used instead. There is a substantial difference between the two commands. The former points the processes root file system to the new

directory with the rest system continuing to run on the old root directory while the latter detaches the new root and attaches it to process root directory, removing dependencies on the old one, being able to unmount the original root directory   and proceed as it had never being used. In the case of Linux containers, they are used for bind mounting container root file system (image) and launching LXC init process in a new mount namespace [6].
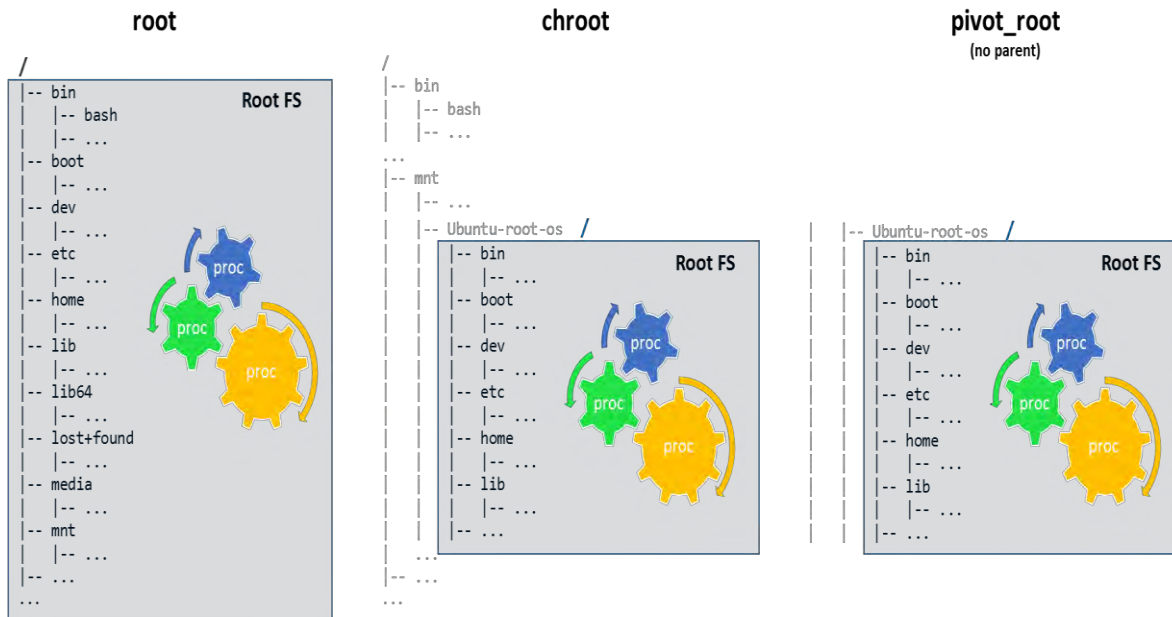
Figure 10 : chroot and pivot_root

2.1.5 Linux Security Modules

Linux was initially developed as a clone of the Unix operating system and thus inherited the core Unix security model which is a form of Discretionary Access Control (DAC). Briefly, Unix DAC allows the owner of an object (i.e a file) to set the security policy for that object, which is why it was given the name "discretionary". This policy is implemented as permission bits attached to the file's inode, set by the owner of the file. Permissions for accessing the file, such as read and write could be set separately for the owner, a specific group, and others (i.e. everyone else).This relatively simple control scheme resembles access control list (ACL) mechanism.

However, functional requirements for security evolved over time making this security design insufficient. Nowadays, new requirements for finer-grained policy and precise access control to resources aroused that were not adequately covered by this Unix DAC scheme. The need for new features to be retrofitted and compatible with the existing

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

23

design of the system became evident and **Linux Security Modules** (**LSM**), a security enhancement was introduced. Linux Security Modules (LSM) is a framework that allows the Linux kernel to support a variety of computer security models while avoiding favoritism toward any single security implementation [7]. The framework is licensed under the terms of the GNU General Public License and is standard part of the Linux kernel since Linux 2.6. Apparmor, SELinux, Smack and TOMOYO Linux are the currently accepted modules in the official kernel. LSM was designed to provide the specific needs of everything needed to successfully implement a mandatory access control module, while imposing the fewest possible changes to the Linux kernel.

The Linux Security Modules (LSM) API implements hooks at all security-critical points within the kernel. A user of the framework (an "LSM") can register with the API and receive callbacks from these hooks. All security-relevant information is safely passed to the LSM, avoiding race conditions, and the LSM decides whether to allow the operation, or deny it forcing an error code return, as explained in the diagram of figure 7. The LSM API allows different security models to be plugged into the kernel - typically access control frameworks. To ensure compatibility with existing applications, the LSM hooks are placed so that the Unix DAC checks are performed first, and only if they succeed, is LSM code invoked.

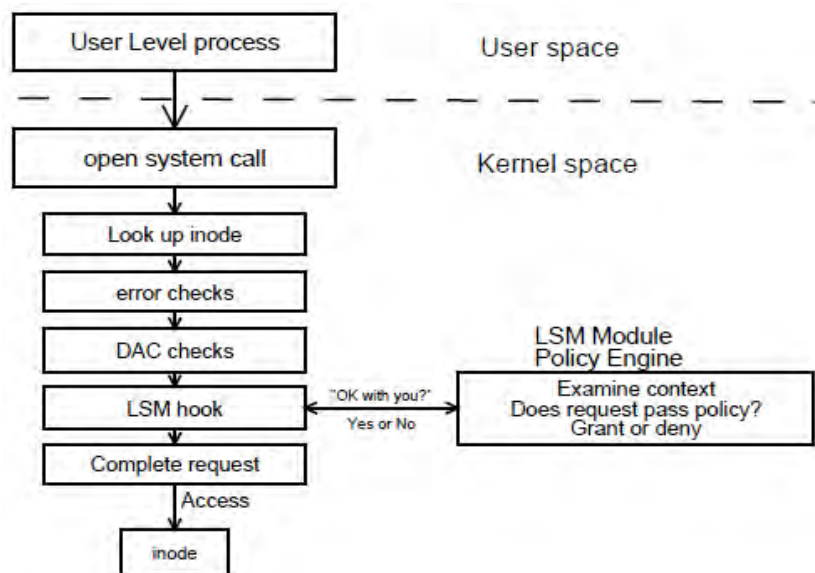

Figure 11 : LSM Hook Architecture

The basic abstraction of the LSM interface is to mediate access to internal kernel objects. LSM seeks to allow m**odules to answer the question "**May a subject S perform a kernel operation OP on an internal kernel object OBJ?**"

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

24

2.1.5.1 *SELinux*

Security Enhanced Linux (SELinux) is an implementation of fine-grained Mandatory Access Control (MAC) designed to meet a wide range of security requirements, from general purpose use, through to government and military systems which manage classified information. MAC security differs from DAC in that the security policy is administered centrally, and users do not administer policy for their own resources. This helps contain attacks which exploit user software bugs and misconfiguration.
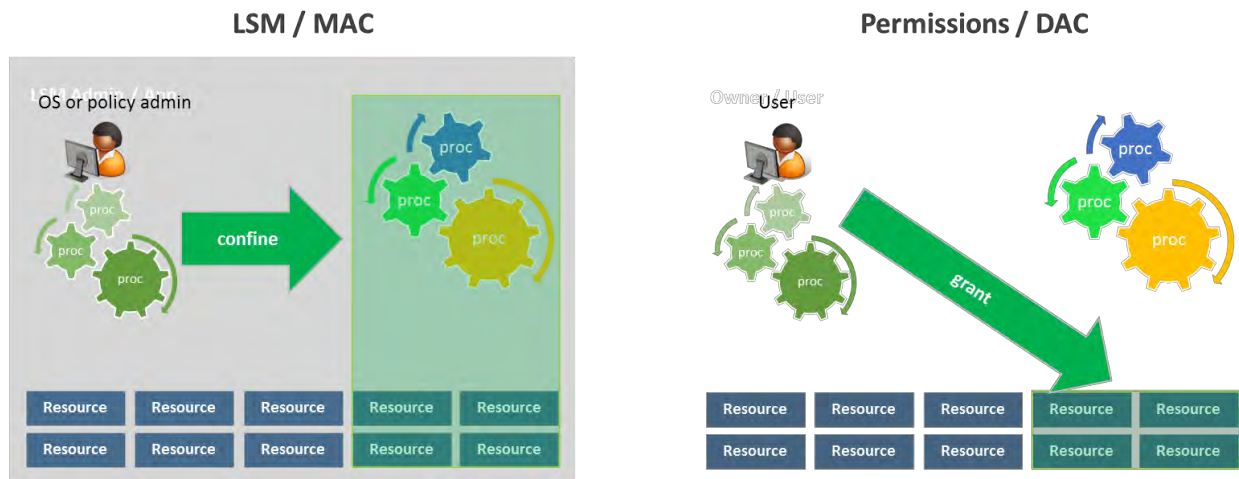
Figure 12 : MAC vs DAC

In SELinux, all objects on the system, such as files and processes, are assigned security labels. All security-relevant interactions between entities on the system are hooked by LSM and passed to the SELinux module, which consults its security policy to determine whether the operation should be allowed.

2.1.5.2 *Smack*

The Smack LSM was designed to provide a simple form of MAC security, in response to the relative complexity of SELinux. It's also implemented as a label-based scheme with a customizable policy. Smack is part of the Tizen security architecture and has seen adoption generally in the embedded space.

2.1.5.3 *AppArmor*

AppArmor is a MAC scheme for confining applications and was designed for simplicity in management. Policy is configured as application profiles using familiar Unix-style abstractions such as pathnames. It is fundamentally different to SELinux and Smack in that instead of direct labeling of objects, security policy is applied to pathnames. AppArmor also features a learning mode, where the security behavior of an application is observed and converted automatically into a security profile.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

25

2.1.5.4 *TOMOYO*

The TOMOYO module is another MAC scheme which implements path-based security rather than object labeling. It's also aimed at simplicity, by utilizing a learning mode similar to AppArmor's where the behavior of the system is observed for the purpose of generating security policy.

What's different about TOMOYO is that what's recorded are trees of process invocation, described as "domains". For example, when the system boots, from init, as series of tasks are invoked, this particular chain of tasks is recorded as a valid domain for the execution of that application, and other invocations which have not been recorded are denied.

In conclusion, LSM is a framework to enforce more advanced security politics in modern Linux operating systems than the initial obsolete Unix DAC scheme and its value lies on the fact that it is modular and appropriately designed to support a variety of security models.

3 LXC

LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it allows Linux users to create and manage system or application containers, effortless. LXC containers are often considered as something in the middle between a chroot and a full-fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel [8].

Current LXC uses the following kernel features to contain processes:

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Apparmor and SELinux profiles
- Seccomp policies
- Chroots (using pivot_root)
- Kernel capabilities
- CGroups (control groups)

3.1 Components

LXC is currently made of a few separate components:

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

26

- The liblxc library
- Several language bindings for the API:
- A set of standard tools to control the containers
- Distribution container templates

3.2   Security

LXC containers can be of two kinds:

- Privileged containers
- Unprivileged containers

3.2.1   Privileged Containers

Privileged containers are defined as any container where the container uid 0 is mapped to the host's uid 0. In such containers, protection of the host and prevention of escape is entirely applied through Mandatory Access Control (apparmor, selinux),seccomp filters, dropping of capabilities and namespaces. Those technologies combined will typically prevent any accidental damage of the host, where damage is defined as reconfiguring host hardware, the host kernel or accessing the host file system. LXC upstream's position is that those containers aren't and cannot be root-safe. They are still valuable in an environment where you are running trusted workloads or where no untrusted task is running as root in the container.

3.2.2   Unprivileged containers

Unprivileged containers are safe by design. The container uid 0 is mapped to an unprivileged user outside of the container and only has extra rights on resources that it owns itself.With such container, the use of SELinux, AppArmor, Seccomp and capabilities isn't necessary for security. LXC will still use those to add an extra layer of security which may be handy in the event of a kernel security issue but the security model isn't enforced by them. As a result, most security issues (container escape, resource abuse, etc) in those containers will apply just as well to a random unprivileged user and so would be a generic kernel security bug rather than a LXC issue.

3.3   LXD

LXD is a container "hypervisor" consisting of three components:

- A system-wide daemon (lxd)
- A command line client (lxc)
- An OpenStack Nova plugin (nova-compute-lxd)

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

27

The core of LXD is a daemon which offers a REST API to drive full system containers the same way as driving virtual machines. The command line tool provides a simple yet effective management of containers. It can handle connection to multiple container hosts and easily provide an overview of all the containers of the network. LXD supports effortless creation of containers and the ability of relocation, while running. Lastly, the OpenStack plugin allows the use of LXD hosts as compute nodes that run workloads on containers rather than virtual machines.

3.3.1 Features

Some of the biggest features of LXD are:

- Secure by design (unprivileged containers, resource restrictions and much more)
- Scalable (from containers on your laptop to thousands of compute nodes)
- Intuitive (simple, clear API and crisp command line experience)
- Image based (no more distribution templates, only good, trusted images)
- Live migration

3.3.2 Relationship with LXC

It is worth mentioning that LXD isn't a rewrite of LXC, in fact it's a set of tools built on top of LXC. It uses the stable LXC API to accomplish container management behind the scene, adding the REST API on top and providing a much simpler, more consistent user experience. It's basically an alternative to LXC's tools and distribution template system with the added features that come from being controllable over the network. The focus of LXD is on system containers, that is, containers which run a clean copy of a Linux distribution or a full appliance.

4 Docker

4.1 What is Docker?

A technology that packages software into standardized units for development, shipment and deployment. Docker container technology was launched in 2013 as an open source project that automates the deployment of applications inside software containers. A Docker container image is a lightweight, standalone, executable package of that contains everything needed to run correctly and seamlessly: code, runtime, system tools, system libraries and settings. This guarantees that the software will always run the same, regardless of its environment.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless

of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers characteristics:

- **Standard:** Docker created the industry standard for containers, so they could be portable anywhere
- **Lightweight:** Containers share the machine's OS system kernel and therefore do not require an OS per application avoiding the overhead of creating and maintaining virtual machines
- **Secure:** Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

4.2 Comparing Docker containers and Virtual machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently due to containers virtualizing the operating system instead of hardware. Containers' advantages include portability and efficiency.

Virtual machines' (VMs) architecture relies on an abstraction of physical hardware, allowing the hypervisor to run multiple VM instances on a single machine. Each VM includes a copy of an operating system, the application, all necessary binaries and libraries, resulting in considerable amount of storage. Each OS running demands its own resources thus impeding overall system performance.

Containers on the other hand, provide an abstraction of the application layer packaging code, dependencies and appropriate settings together. Multiple containers can run on the same machine, sharing the OS kernel with other containers, each running as isolated processes. Containers take up considerably smaller space (container images are typically tens of MBs in size), can handle more applications and require fewer system resources in comparison to VMs [9].

We have mentioned briefly the advantages of each technology but given the fact that are totally different in conception and as a result target different needs, it would be of no practical usage proceeding in point to point comparisons. It would be of much more interest if we could see them as complementary technologies which is the case of Docker Swarm we will examine later in this paper.

4.3 Solving portability problems

In our introductory part of Linux containers, we mentioned that the key concept of container technology was to solve efficiently the problem of migration between many different software environments and various hardware implementations. "Containerizing"

an application with all its dependencies, provides an autonomous software block capable of being moved among numerous environments.

Packaging an application in a container with its configurations and dependencies guarantees that the application will always work as designed in any environment, locally, or on another machine, leaving developers the space to focus only on the deployment of their application. Docker extends this idea and containers are not tied to any specific infrastructure, they run on any computer, on any infrastructure, and in any cloud, provided Docker engine is installed. Features like Docker images and Docker registries, assist in distributing and sharing content between developers and system administrators.

4.4 Docker Features

Docker introduced new features that facilitated the deployment and portability of application containers.

4.4.1 Docker Images

Docker containers are based on Docker images. A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities. It is actually the basic unit this packaging technology is built on. Docker containers have access only to resources defined in the image, unless you the container is given additional access while creating it.

Technically a Docker image is actually, a read-only template that contains all the ingredients of a container to run, including the operating system and the application with its dependencies. In detail, each Docker image references a list of read-only layers that represent file system differences. Layers are stacked on top of each other to form a base for a container's root file system. The diagram below shows the Ubuntu 15.04 image comprising 4 stacked image layers.

Figure 13 : Image consisting of layers

The Docker storage driver is responsible for stacking these layers and providing a single unified view.

4.4.2 Container and Layers

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted, while the underlying image remains unchanged. The diagram below shows a container based on the image presented earlier.
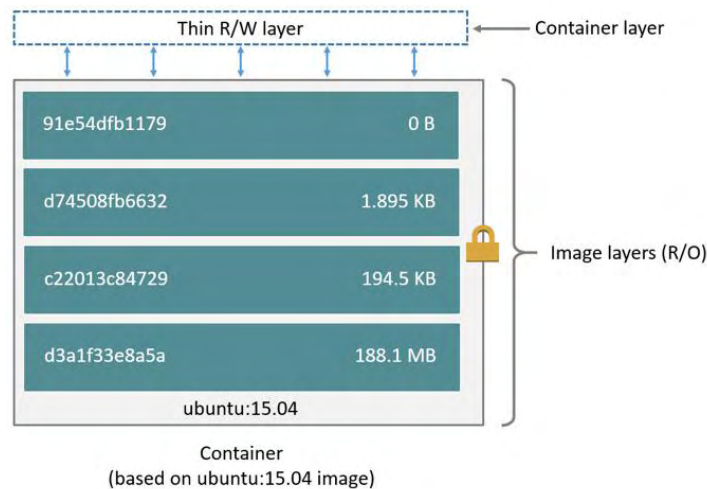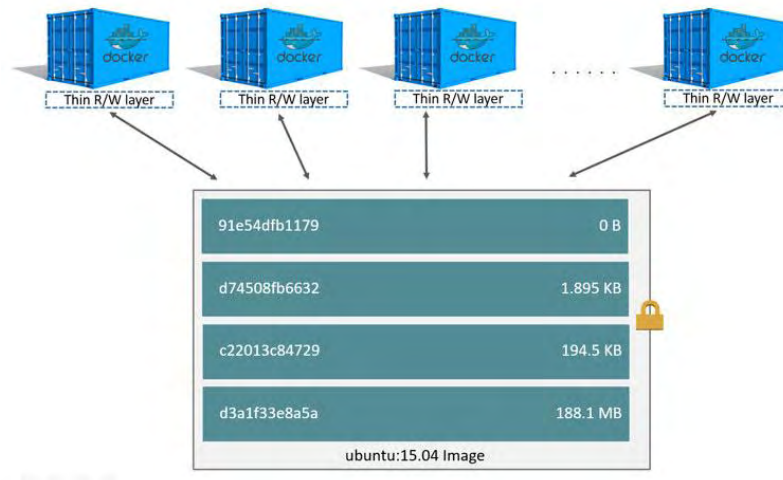

Figure 14 : Container Layer

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

31

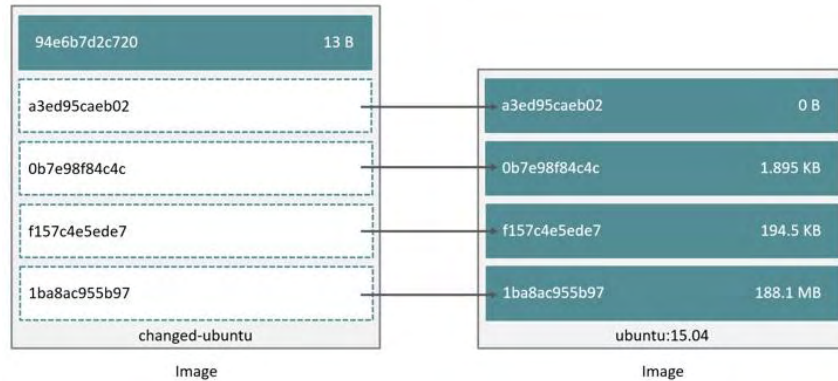**Figure 15 : Containers sharing the same image**

Docker 1.10 introduced a new content addressable storage model. This is a completely new way to address image and layer data on disk. Previously, image and layer data was referenced and stored using a randomly generated UUID. In the new model this is replaced by a secure content hash. The new model improves security, provides a built-in way to avoid ID collisions, and guarantees data integrity. It also enables better sharing of layers by allowing many images to freely share their layers even if they didn't come from the same build.

4.4.3 Docker storage driver

The Docker storage driver is responsible for enabling and managing both the image layers and the writable container layer. How a storage driver accomplishes these can vary between drivers. Two key technologies behind Docker image and container management are stackable image layers and copy-on-write (CoW).

Copy-on-write is a strategy that includes sharing and copying. In this strategy, system processes that need the same data share the same instance of that data rather than having their own copy. At some point, if one process needs to modify or write to the data, only then does the operating system make a copy of the data for that process to use. Only the process that needs to write has access to the data copy. All the other processes continue using the original data. Docker uses a copy-on-write technology with both images and containers. This CoW strategy optimizes both image disk space usage and the performance of container start times. In the diagram bellow we can see how sharing of image layers takes effect in the creation of a new image based on an existing one and how this technique promotes smaller images.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

32

Figure 16 : New image created from an existing one

This sharing of image layers is what makes Docker images and containers so space efficient. As, already explained all writes made to a container are stored in the thin writable container layer. The other layers are read-only (RO) image layers and can't be changed. This means that multiple containers can safely share a single underlying image.

When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the default aufs driver and the overlay and overlay2 drivers, the copy-on-write operation follows this rough sequence:

- Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.

- Perform a copy_up operation on the first copy of the file that is found, to copy the file to the container's writable layer.

- Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Btrfs, ZFS, and other drivers handle the copy-on-write differently.

This copy-on-write strategy not only reduces the amount of space consumed by containers, it also reduces the time required for a container to start. Docker only has to create the thin writable layer for each container, not needing to make an entire copy of the image stack, thus significantly reducing the startup time.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

33

4.4.4 Data Volume

A data volume is a directory or file in the Docker host's file system that is mounted directly into a container. Data volumes are not controlled by the storage driver. Reads and writes to data volumes bypass the storage driver and operate at native host speeds. Any number of data volumes can be mounted into a container. Multiple containers can also share one or more data volumes. The diagram below shows a single Docker host running two containers. Each container exists inside of its own address space within the Docker host's local storage area (`/var/lib/docker/...`). There is also a single shared data volume located at `/data` on the Docker host. This is mounted directly into both containers.
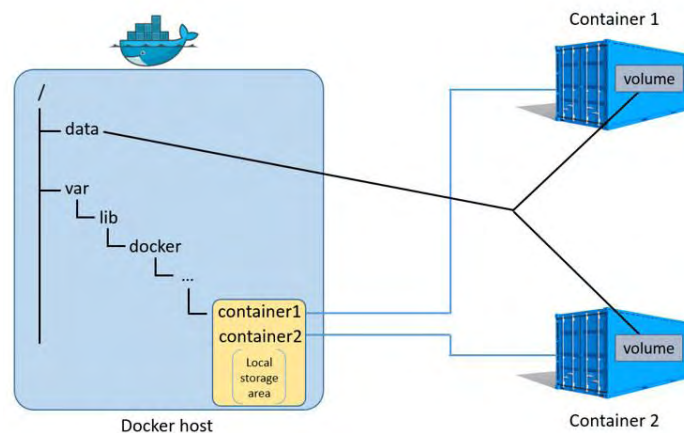

Figure 17 : Data volumes

Data volumes reside outside of the local storage area on the Docker host, further reinforcing their independence from the storage driver's control [10]. When a container is deleted, any data stored in data volumes persists on the Docker host.

4.4.5 Docker Hub

Docker moved one step further in the direction portability by introducing of the tool Docker Hub which leverages the sharing and reuse of Docker images. Docker Hub is a cloud-based registry service which allows linking to code repositories, building and storing of manually pushed images. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.
Its features can be summarized in the following list:

- Image Repositories: Find, manage, and push and pull images from community, official and private image libraries.
- Automated Builds: Automatically create new images when you make changes to a source code repository.
- Webhooks: A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository.
- Organizations: Create work groups to manage access to image repositories.

- GitHub and Bit bucket Integration: Add the Hub and your Docker Images to your current workflows.

4.5 Docker architecture

At its core, Docker provides a way to run almost any application securely isolated in a container. Isolation and security are the aspects that allow many containers to run simultaneously on the host. The lightweight nature of containers, which run without the extra load of a hypervisor, gives the opportunity to fully exploit hardware performance.

Docker platform and its tooling surrounds the container and is responsible for the following tasks

- Packing applications and supporting components into Docker containers
- Distribution and shipping of those containers
- Deployment of those applications to the production environment, whether it is in a local data center or the Cloud

It is critical, for understanding Docker architecture, to introduce **Docker Engine, a client-server application with three major components:**

- A server which is a type of long-running program called a daemon process.
- A REST API which specifies interfaces, used for programs to communicate with Docker daemon
- A command line interface (CLI) client.

The role of Docker daemon is to create and manage Docker objects. Docker objects include images, containers, networks and data volumes. The Docker client talks to the Docker daemon, which is responsible of building, running, and distributing your Docker containers. Both the Docker client and the daemon can run on the same system, or a Docker client can be connected to a remote Docker daemon and communicate via Linux sockets or through a RESTful API.

4.6 Inside Docker Network Configuration

Containers' default behavior is to promote application isolation from other containers and the underlying infrastructure, providing an overall layer of protection. However, networking connectivity and the way of delivering efficiently containerized microservice applications is an aspect Docker engineers took of great consideration.

Docker's networking architecture was designed to favor:

- Portability
- Service Discovery

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

35

- Load Balancing
- Security
- Performance
- Scalability

4.6.1 The Container Networking Model

The Docker networking architecture is built on a set of interfaces called the Container Networking Model (CNM) [11]. The philosophy of CNM is to provide application portability across diverse infrastructures. This model strikes a balance to achieve application portability and also takes advantage of special features and capabilities of the infrastructure. Figure below emphasizes the key points of this model.
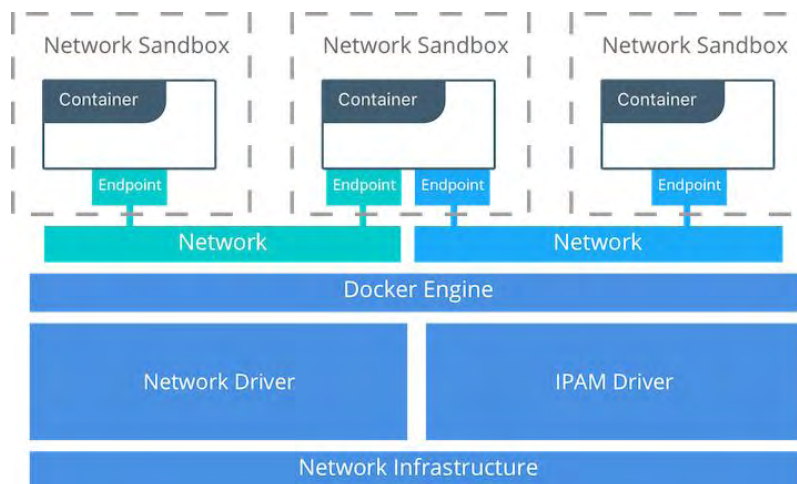

Figure 18 : Container Networking Model

- **Sandbox:** A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a Sandbox could be a Linux Network Namespace. A Sandbox may contain many endpoints from multiple networks.
- **Endpoint:** An Endpoint joins a Sandbox to a Network. The Endpoint construct exists so the actual connection to the network can be abstracted away from the application. This helps maintain portability so that a service can use different types of network drivers without being concerned with how it's connected to that network.
- **Network:** The CNM does not specify a Network in terms of the OSI model. An implementation of a Network could be a Linux bridge, a VLAN, etc. A Network is a collection of endpoints that have connectivity between them. Endpoints that are not connected to a network do not have connectivity on a network.

Network Drivers: Docker Network Drivers provide the actual implementation that makes networks work. They are pluggable so that different drivers can be used and interchanged easily to support different use cases. Multiple network drivers can be used on a given

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

36

Docker Engine, but each Docker network is only instantiated through a single network driver. There are two broad types of CNM network drivers:

- **Native Network Drivers:** Native Network Drivers are a native part of the Docker Engine and are provided by Docker. There are multiple drivers to choose from that support different capabilities like overlay networks or local bridges.
- **Remote Network Drivers:** Remote Network Drivers are network drivers created by the community and other vendors.

IPAM Drivers: Docker has a native IP Address Management Driver that provides default subnets or IP addresses for networks and endpoints if they are not specified. IP addressing can also be manually assigned through network, container, and service create commands. Remote IPAM drivers also exist and provide integration to existing IPAM tools.

Docker Native Network Drivers

The following native network drivers exist:

- **Host:** With the host driver, a container uses the networking stack of the host. There is no namespace separation, and all interfaces on the host can be used directly by the container
- **Bridge:** The bridge driver creates a Linux bridge on the host that is managed by Docker. By default containers on a bridge can communicate with each other. External access to containers can also be configured through the bridge driver, with limitations.
- **Overlay:** The overlay driver creates an overlay network that supports multi-host networks out of the box. It uses a combination of local Linux bridges and VXLAN to overlay container-to-container communications over physical network infrastructure
- **MacVlan:** The macvlan driver uses the MACVLAN bridge mode to establish a connection between container interfaces and a parent host interface (or sub-interfaces). It can be used to provide IP addresses to containers that are routable on the physical network. Additionally VLANs can be trunked to the macvlan driver to enforce Layer 2 container segmentation
- **None:** The none driver gives a container its own networking stack and network namespace but does not configure interfaces inside the container. Without additional configuration, the container is completely isolated from the host networking stack

4.6.2 Docker Host Network Driver

It's the same networking configuration that Linux uses. `--net=host` effectively turns Docker networking off and containers use the host (or default) networking stack of the host operating system.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

37

Typically with other networking drivers, each container is placed in its own network namespace (or sandbox) to provide complete network isolation from each other. With the host driver containers are all in the same host network namespace and use the network interfaces and IP stack of the host. All containers in the host network are able to communicate with each other on the host interfaces.

From a networking standpoint this is equivalent to multiple processes running on a host without containers. Because they are using the same host interfaces, no two containers are able to bind to the same TCP port. This may cause port contention if multiple containers are being scheduled on the same host.
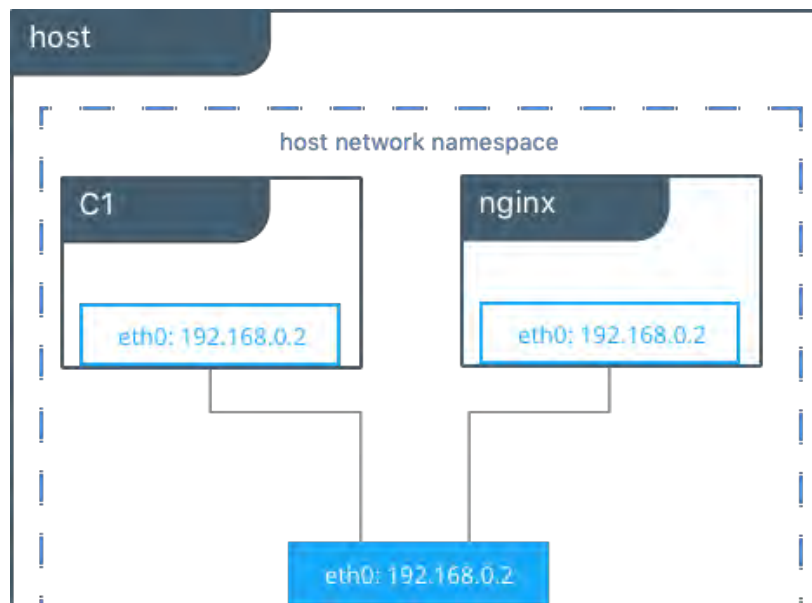

Figure 19 : Host Network Driver

4.6.3   Docker Bridge Network Driver

On any host running Docker Engine, there is, by default, a local Docker network named bridge. This network is created using a bridge network driver which instantiates a Linux bridge called `docker0`.

The bridge driver creates a private network internal to the host so containers on this network can communicate. External access is granted by exposing ports to containers. Docker secures the network by managing rules that block connectivity between different Docker networks. A container can have zero to many interfaces depending on how many networks it is connected to. Each Docker network can only have a single interface per container.
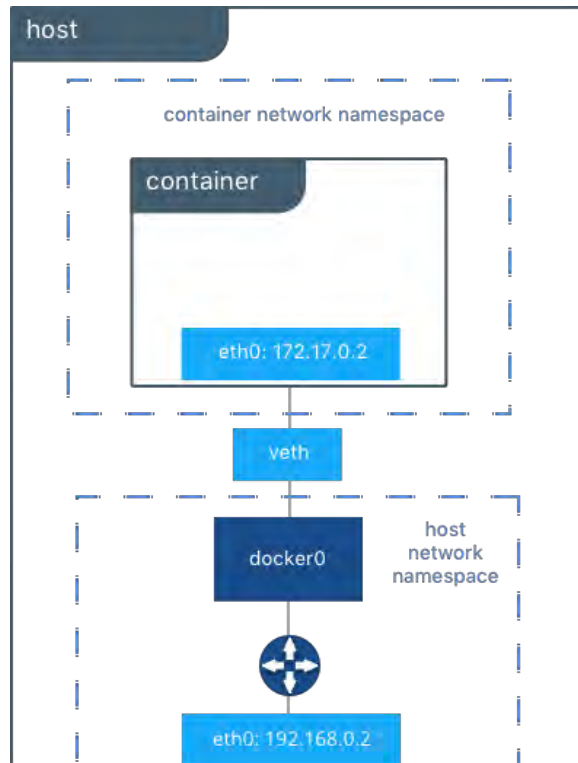
Figure 20 : Bridge Networking Driver

The host routing table provides connectivity between `docker0` and `eth0` on the external network, completing the path from inside the container to the external network. By default `bridge` is assigned one subnet from the ranges `172.[17-31].0.0/16` or `192.168.[0-240].0/20` which does not overlap with any existing host interface. The default `bridge` network can also be configured to use user-supplied address ranges. Also, an existing Linux bridge can be used for the `bridge` network rather than Docker creating one.

4.6.4 User-Defined Bridge Networks

In addition to the default networks, Docker provides the capability of creating user-defined networks of any network driver type. In the case of user-defined bridge networks, a new Linux bridge is setup on the host. Unlike the default bridge network, user-defined networks supports manual IP address and subnet assignment. If an assignment isn't given, then Docker's default IPAM driver assigns the next subnet available in the private IP space.

4.6.5 External Access for Standalone Containers

By default all containers on the same Docker network (multi-host swarm scope or local scope) have connectivity with each other on all ports. Communication between different Docker networks and container ingress traffic that originates from outside Docker is

firewalled. This is a fundamental security aspect that protects container applications from the outside world and from each other.

For most types of Docker networks (bridge and overlay included) external ingress access for applications must be explicitly granted. This is done through internal port mapping. Docker publishes ports exposed on host interfaces to internal container interfaces. The following diagram depicts ingress (bottom arrow) and egress (top arrow) traffic to container C2. Outbound (egress) container traffic is allowed by default. Egress connections initiated by containers are masqueraded/SNATed to an ephemeral port (typically in the range of 32768 to 60999). Return traffic on this connection is allowed, and thus the container uses the best routable IP address of the host on the ephemeral port
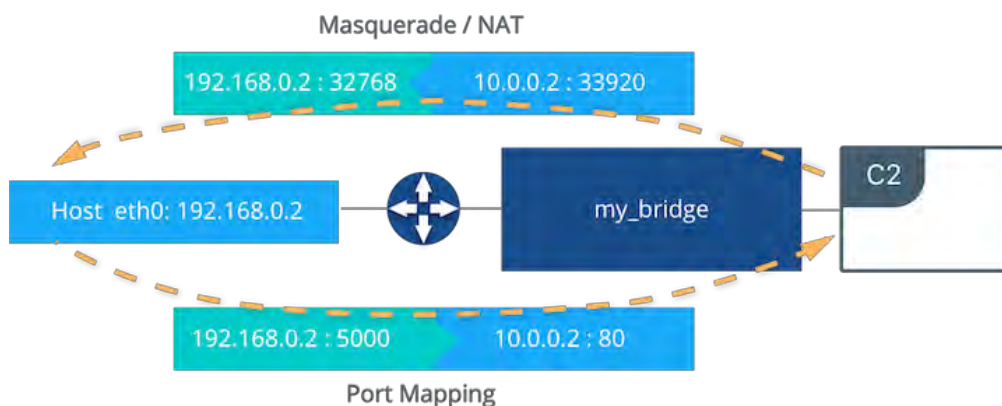

Figure 21 : Container Exposes Ports

Ingress access is provided through explicit port publishing. Port publishing is done by Docker Engine and can be controlled through UCP or the Engine CLI. A specific or randomly chosen port can be configured to expose a service or container. The port can be set to listen on a specific (or all) host interfaces, and all traffic is mapped from this port to a port and interface inside the container.

4.6.6 Overlay Driver Network Architecture

The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

The overlay driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the underlay). The Docker overlay network encapsulates container traffic in a VXLAN header which allows the traffic to traverse the physical Layer 2 or Layer 3 network. The overlay makes network segmentation dynamic and easy to control no matter what the underlying physical

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

40

topology. Use of the standard IETF VXLAN header promotes standard tooling to inspect and analyze network traffic.

IETF VXLAN (RFC 7348) is a data-layer encapsulation format that overlays Layer 2 segments over Layer 3 networks. VXLAN is designed to be used in standard IP networks and can support large-scale, multi-tenant designs on shared physical network infrastructure. Existing on-premises and cloud-based networks can support VXLAN transparently.

VXLAN is defined as a MAC-in-UDP encapsulation that places container Layer 2 frames inside an underlay IP/UDP header. The underlay IP/UDP header provides the transport between hosts on the underlay network. The overlay is the stateless VXLAN tunnel that exists as point-to-multipoint connections between each host participating in a given overlay network.
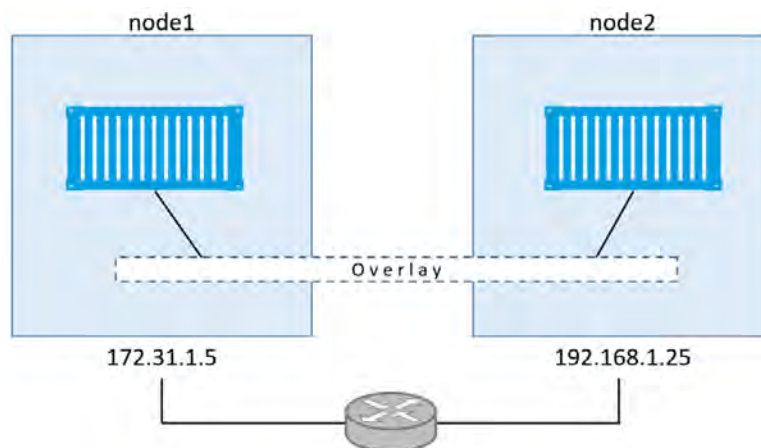

Figure 22 : Overlay driver perspective

Because the overlay is independent of the underlay topology, applications become more portable. Thus, network policy and connectivity can be transported with the application whether it is on-premises, on a developer desktop, or in a public cloud.

4.6.7   MACVLAN

The macvlan driver is a new implementation of the network virtualization technique. Linux implementations are extremely lightweight because rather than using a Linux bridge for

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

41

isolation, they are simply associated with a Linux Ethernet interface or sub-interface to enforce separation between networks and connectivity to the physical network.

MACVLAN offers a number of unique features and capabilities. It has positive performance implications by virtue of having a very simple and lightweight architecture. Rather than port mapping, the MACVLAN driver provides direct access between containers and the physical network. It also allows containers to receive routable IP addresses that are on the subnet of the physical network.

MACVLAN use-cases may include:

- Very low-latency applications
- Network design that requires containers be on the same subnet as and using IPs as the external host network

The macvlan driver uses the concept of a parent interface. This interface can be a physical interface such as `eth0`, a sub-interface for 802.1q VLAN tagging like `eth0.10` (.10 representing VLAN 10), or even a bonded host adaptor which bundles two Ethernet interfaces into a single logical interface.
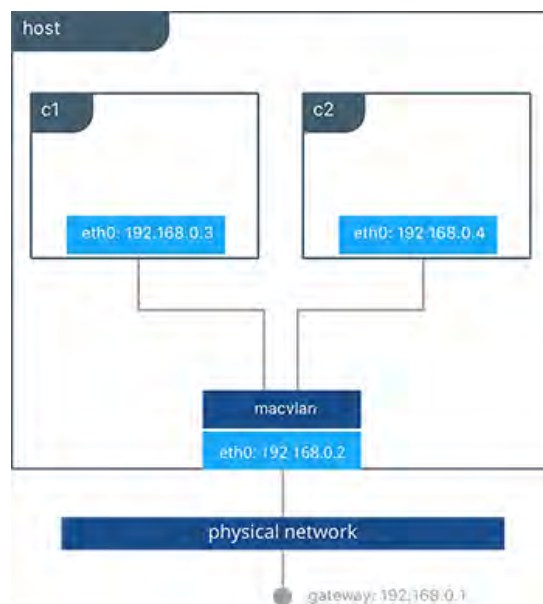

Figure 23 : Macvlan driver

A gateway address is required during MACVLAN network configuration. The gateway must be external to the host provided by the network infrastructure. MACVLAN networks allow access between containers on the same network. Access between different MACVLAN networks on routing outside the host.

4.6.8 VLAN Trunking with MACVLAN

The `macvlan` driver completely manages sub-interfaces and other components of the MACVLAN network through creation, destruction, and host reboots. When the `macvlan` driver is instantiated with sub-interfaces it allows VLAN trunking to the host and segments containers at L2. The `macvlan` driver automatically creates the sub-interfaces and connects them to the container interfaces. As a result each container is in a different VLAN, and communication is not possible between them unless traffic is routed in the physical network. Figure below presents an example of macvlan truncking implementation
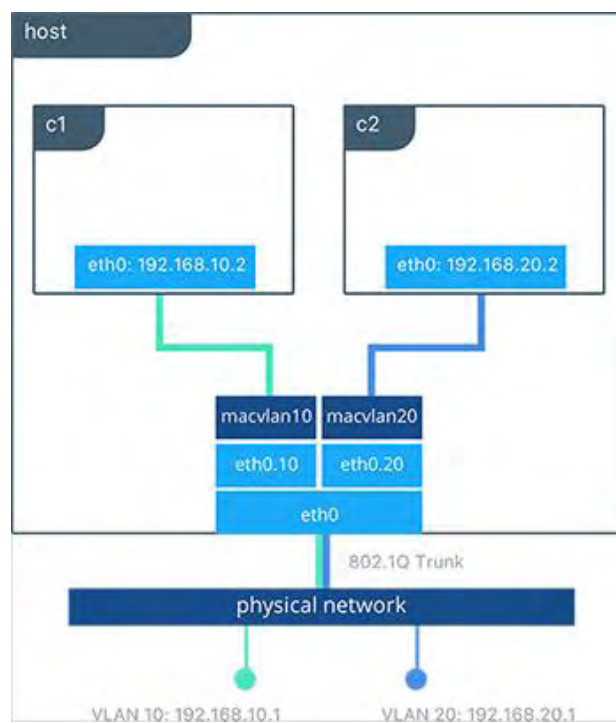

Figure 24 : Macvlan truncing

4.6.9 None (Isolated) Network Driver

Similar to the host network driver, none network driver is essentially an unmanaged networking option. Docker Engine does not create interfaces inside the container, establish port mapping, or install routes for connectivity. A container using --net=none is completely isolated from other containers and the host. The networking admin or external tools must be responsible for providing this plumbing. A container using none only has a loopback interface and no other interfaces. Unlike the host driver, the none driver creates

a separate namespace for each container. This guarantees container network isolation between any containers and the host.

4.7 Orchestration of Docker containers

Presentation of Docker, in this paper, focused mainly on the isolation of containers, which was part of the initial goal of this technology, the remaining part aims in composition and the challenges it introduces. Most valuable systems are composed of two or more components. Simple management of multiple components is more important than ever due to the rise of large-scale server software, service-oriented architectures, micro-services, and now the Internet-of-Things. Various Docker software tools and solutions exist to facilitate with these challenges and cover each of the following areas to a greater or lesser degree:

- Clustering: Grouping hosts (VMs or bare-metal) and networking them together. A cluster should feel like a single resource rather than a group of disparate machines.
- Orchestration: All components working together seamlessly. Starting containers on appropriate hosts and connecting them. An orchestration system may also include support for scaling, automatic failover and node rebalancing.
- Management: Providing oversight into the system and supporting various administrative tasks.

4.7.1 Docker Compose

Compose is a tool for defining, launching, and managing services, where a service is defined as one or more replicas of a Docker container. Services and systems of services are defined in YAML files and managed with the command-line program Docker-compose.

With compose simple commands are used to accomplish the following tasks:
- Build Docker images
- Launch containerized applications as services
- Launch full systems of services
- Manage the state of individual services in a system
- Scale services up or down
- View logs for the collection of containers making a service

Compose moves focus from individual containers to describing full environments and service component interactions. A Compose file might describe four or five unique services that are interrelated but should maintain isolated and may scale independently.

This level of interaction covers most of the everyday use cases for system management. For that reason, most interactions with Docker will be through Compose.

4.7.2 Docker Swarm

Swarm is the native clustering tool for Docker. Swarm uses the standard Docker API so, containers can be launched using normal Docker run commands and Swarm will take care of selecting an appropriate host to run the container on. This also means that other tools that use the Docker API, such as Compose, can use Swarm without any changes and take advantage of running on a cluster rather than a single host.

The basic architecture of Swarm is fairly straightforward: each host runs a Swarm Agent and one host runs a Swarm manager (on small test clusters both the agent and manager many be installed on the same node). The manager is responsible for the orchestration and scheduling of containers on the hosts.

4.7.3 Docker Machine

Docker Machine is a tool which facilitates the installation of Docker Engine on virtual hosts, and manages the hosts with Docker-machine commands. Machine can be used to create Docker hosts on local Mac or Windows boxes, on company networks, in data centers, or on cloud providers like AWS or Digital Ocean. Docker-machine can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon and configure properly a Docker client, with the convenience of simple and straightforward commands.

4.8 Differences between Docker and LXC

Docker technology is not a replacement for LXC. "LXC" refers to capabilities of the Linux kernel (specifically namespaces and control groups) which allow sandboxing processes from one another, and controlling their resource allocations. On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities further discussed below [12].

Portable deployment*: Portable deployment across machines. Docker defines a format for bundling an application and all its dependencies into a single object which can be transferred to any Docker-enabled machine, and executed there with the guarantee that the execution environment exposed to the application will be the same. LXC implements process sandboxing, which is an important pre-requisite for portable deployment, but that alone is not enough for portable deployment. An application installed in a custom LXC configuration, would almost certainly not run properly if installed in another node, because

it is tied to the machine's specific configuration: networking, storage, logging, distro, etc. Docker defines an abstraction for these machine-specific settings, so that the exact same Docker container can run unchanged, on many different machines, with many different configurations.

Application centric: Docker is optimized for the deployment of applications, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the LXC helper scripts focus on containers as lightweight machines - basically servers that boot faster and need less RAM.

Automatic build: Docker includes a tool for developers to automatically assemble a container from their source code, with full control over application dependencies, build tools, packaging etc. User is free to use make, maven, chef, puppet, salt, Debian packages, RPMs, source tarballs, or any combination of the above, regardless of the configuration of the machines.

Component re-use: Any container can be used as a base image to create more specialized components. This can be done manually or as part of an automated build.

Sharing: Docker has access to a public registry on Docker Hub, an open registry with pre-configured images, for sharing

Tool ecosystem: Docker defines an API for automating and customizing the creation and deployment of containers. There are a huge number of tools integrating with Docker to extend its capabilities

4.9   Linux Containers vs Virtual Machines

4.9.1   Performance comparison

Virtual machines are used extensively in cloud computing. In particular, the state-of-the-art in Infrastructure as a Service (IaaS) is largely synonymous with virtual machines. Recently though, popular commercial cloud platforms in their need of better performance and density, exploiting the inherit benefits of containers, began to adopt container technology. Their interest and the depth of specialization on this new technology, is emphasized not only by the fact that they use it internally along with virtual machines but they also offer simple and manageable container interfaces for the end users. Container's promising features arouse serious discuss on performance issues and instantaneously numerous researches that focus on containers and virtual machines emerged.

Initially, we will examine the various results of relatively recent papers and then elaborate on specific advantages of each technology. Details and graphs of each case study are

avoided and can be found in the reference part at the end of this paper, we emphasize on the quality results that become apparent after each survey. This paragraph is not intended to outbid on the usage of containers over virtual machines, but better aims to clarify the appropriate solution for a given set of preconditions.

Representative of the interest Linux containers have gathered is the fact that leading companies in the computer industry like IBM reported a research exactly on the performance comparison of virtual machines and containers [13]. In the specific paper, researchers of IBM compare overhead imposed by virtual machines (created by KVM) and containers (using Docker) relative to non-virtualized Linux. Benchmarks and workloads used are relevant to the cloud in attempt to collect results of practical usage. In general, Docker equals or exceeds KVM performance in every case of testing. Test results show that both KVM and Docker introduce negligible overhead for CPU and memory performance (except in extreme cases). For I/O intensive workloads, both forms of virtualization should be used carefully. Although KVM performance has improved considerably since its creation, and even using the fastest available forms of paravirtualization, KVM stills adds some overhead to every I/O operation. This overhead ranges from significant when performing small I/O operations to negligible when it is amortized over large ones. Although **containers have almost no overhead**, Docker is not without performance obstacles. Docker volumes have noticeable better performance than files stored in AUFS. Docker's NAT also introduces overhead for workloads [14] with high packet rates. These features represent a tradeoff between ease of management and performance and should be considered on a case-by-case basis.

Another, interesting and relatively recent study focuses on the performance comparison of a WebRTC server on Docker containers versus virtual machines (KVM) [15]. WebRTC is an API standard that supports voice and video chat and P2P file sharing, without the need to install external plugin on browsers. As lots of services will switch to WebRTC, a performance test between container and virtual machines for a real-time service seemed a very interesting scenario that would pinpoint the advantages and pitfalls of each mechanism. More details on the experiments and graphs can be found on the referenced paper but the overall picture of extensive experimenting on different testing scenarios is that **CPU usage found to be 5-10% lower for the Docker instances** where latency, a critical metric for real-time applications, is more stable and a bit lower compared to KVM machines.

Impressive results are drawn from the paper entitled "Performance comparison Analysis of Linux Container and Virtual Machine for building Cloud" [16]. Authors of this paper used Openstack, which is opensource software for building public and private cloud, for the construction of the environment and used Docker containers and KVM Hypervisors as virtualizations tools for comparison. Cloud comprised of Docker doesn't contain guest OS, so CPU resource and storage overhead is minimal. For this reason, **boot-time, time of**

generating and distributing images is little compared to the corresponding times of virtual machines.

Similar results on Cloud performance and deployment are underlined by the paper released from HP with the tile "Linux container performance on HPE ProLiant servers" [17] which container enhanced testing scenarios on distributed environments and web - based applications. Detailed diagrams and graphs of this paper explain every aspect of performance comparison and conclude to the fact that containers usually have a performance edge, as long as the mix of apps is suitable to containerization. This performance edge comes usually from the lack of hypervisor overhead, elimination of duplicated OS services inside each VM and less waste of available memory. Testing also showed that **containers become more efficient as the number of containers increases**. Interesting results occurred when memory requirements exceed available resources in the server. Both containers and VMs continued to work. However, container performance remained better. If resource needs increase beyond the available system resources, some VMs will hang while containers will continue to work, but could become unstable.

4.9.2 Deciding between containers and virtual machines

A common fact underlined by all papers examined above, is that containers' approach of virtualization results on **minimal performance overhead** which translates **into better exploitation of existing hardware and efficient use of system resources**. However, all papers imply that the choice of containers should be carefully considered and only applied if certain preconditions are met. Scott S. Lowe, a VMware engineering architect, in his article suggests that each time we should look at the "scope" of our work [18]. Containers offer a **more narrowed view of a system**, that of an application and should be used accordingly. In other words if we want to test and run multiple copies of a single application (MySQL for example), then it would be wise to use a container. If the flexibility of running multiple applications is needed then a virtual machine rises as a better and completed choice.

Examples of real-life applications that show better performance in containers include [19]:

- High-loaded, multi-component and multi-instance web and application servers
- Data analytic software (especially running in a SaaS model and using VM/container partitioning for tenancy)
- Applications built on a micro-services architecture
- Multiple concurrent batch-processing workloads that use all available resources

4.9.3 Advantages and limitations of Docker

Summing up, Docker is a container management solution that provides:

- A simple way to package and deliver applications and all their dependencies, one that enables seamless application portability and mobility.
- Relative ease of use and low administration requirements.
- A rich set of tools and utilities.

After a detailed presentation of Docker containers' features and a thorough comparison to virtual machines, follows a list of Dockers' advantages as long as the limitations it imposes. Some of these drawbacks will be addressed as containers continue to mature. Others reflect fundamental limitations of container architecture, emphasizing that their usage should be applied after careful thought of each case separately.

Docker benefits:

- Lightweight footprint and minimal overhead. Docker images are typically very small, which facilitates rapid delivery and reduces the time to deploy new application containers.
- Reduces a container to a single process which is then easily managed with Docker tools.
- Offers version control and component reuse, successive versions of a container can be tracked, providing the ability of rolling back to previous versions. Containers reuse components from the preceding layers, which makes them noticeably lightweight.
- Encapsulates application configuration and delivery complexity to dramatically simplify and eliminate the need to repeat these activities manually.
- Provides a strongly supportive user community for many aspects of using containers for significant implementations.
- Provides a highly efficient compute environment for applications that are stateless and micro-services based, as well as many stateful applications like databases, message bus, etc.
- Is used very successfully by many groups, particularly Dev and Test, as well as microservices-based production environments.

Docker limitations

- Treats containers differently from a standard host, such as sharing the host's IP address and providing access to the container via a selectable port. This approach can cause management issues when using traditional applications and management tools that require access to Linux utilities such as cron, ssh, daemons, and logging.

- Uses a layered storage system, which in some I/O intensive applications may decrease overall performance, usage of volumes is the proposed solution in such cases.
- Is not ideal for stateful applications due to limited volume management in case of container failover.
- Security and isolation of containers although inherit should be taken into great consideration from administrators
- Temporary nature of container can also work as a major disadvantage, being an obstacle in cases of troubleshooting.

Part II: Implementation

5 Software presentation

5.1 Overview of the initial problem

In this section we will provide a complete presentation of the software we developed, analyze in detail the main components it consists of, discuss the benefits it offers and conclude with future additions-improvements. Before, explaining the internal parts of our implementation, it is essential to present the initial problem that we tried to deal with. There are certain occasions where it is necessary to create and run simultaneously a considerable amount of virtual machines in certain and limited amount of resources. This might contradict with the ease of obtaining new machines from a cloud computing service, nowadays, but this is not always the case, university labs or small businesses have a narrow amount of available machines making a task like the above a real challenge. Particularly, in NITOS Testbed, usually during lab classes or on major experiments, there is a need of starting, concurrently, a considerable amount of virtual machines. This was the initial problem which at first seemed unbearable without the addition of auxiliary resources. On top of this, we had to take into account the time needed for the concurrent creation of all these machines.

Refining the problem and looking through the details, we understood that not always the virtual machines needed the complete functionality an operating system offers but where only partially used, especially for running certain applications. The latter point was the key that helped in exploring alternative solutions.

Linux container was the first candidate since this technology combined all preconditions needed to solve the existing problem. Containers are secure, astounding fast in creating and running and could easily handle the applications users needed for their experiments. Docker containers, an implementation of this technology, similar to LXC but using a different interface for interacting with the kernel, included additional useful characteristics that made it impossible to oversee, so our implementation is based on Docker solution. In the previous chapter we have included a discussion on Docker features that justifies our choice.

5.2 Describing basic parts

5.2.1 Docker

5.2.1.1 *Docker Api*

Docker's rest-ish API [20] was one of the most important factors that played a major role in the decision of the most appropriate solution, regarding linux containers. Docker offers a restful API (with the exception of few commands) that can be used for remote communicating with Docker Server [21]. Docker daemon offers the ability to listen to a specific TCP port on all network interfaces `tcp://0.0.0.0:2375`, other than the default linux socket which is `unix:///var/run/docker.sock` [22] providing the capability of controlling containers remotely. Docker's API was the structure element for building and designing our application.

5.2.1.2 *Docker Macvlan Driver*

Docker, as explained in previous section, offers a variety of networking driver solutions designed for the different needs of each network. We wanted to create a network of containers that resembles the topology and structure of a physical one, so using the Macvlan driver was the appropriate choice. Macvlan network driver provides the capability of assigning a MAC address to each container's virtual network interface, making it appear to be a physical network interface directly connected to the physical network [23].

Summing up, the advantages of macvlan driver that justify our choice:

* Positive performance implications of bypassing the Linux bridge (The bridge `docker0` that traditionally resides in between the Docker host NIC and container interface)
* Very simple setup consisting of container interfaces, attached directly to the Docker host interface
* Easy access for external facing services as there is no port mappings in these scenarios

5.2.2 RabbitMQ

As already mentioned, the environment we wanted to incorporate our implementation offers a limited amount of resources and despite the fact that containers add almost minimal overhead to the system, we wanted for administrative reasons, to have a detailed view of the amount of resources containers consume. While Docker's API has a corresponding command that returns, in streaming form, statistics of each running container, this solution could not be characterized efficient in terms of network usage. This can become clearer if we consider the overhead of continuous messages being transmitted between nodes of a laboratory where other probably more critical network

information is being exchanged. Nitos Lab has already solved the above problem using the technology of a message broker (*RabbitMQ*), so we had to adapt, including this messaging mechanism.

RabbitMQ is open source message broker or queue manager software (*sometimes called message-oriented middleware*) that implements the Advanced Message Queuing Protocol (*AMQP*) [24]. Its use lies on the fact that offers applications a common platform for sending and receiving messages, minimizing the need of re-planning and developing communication protocols between applications. Messaging in RabbitMQ is asynchronous, decoupling applications by separating sending and receiving data. It can be thought of as data delivery of non-blocking operations or push notification, as publish/subscribe asynchronous processing, or as work queues. The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. There are client libraries available to interface with the broker for all major programming languages, in our project we used the corresponding ruby client [25].

RabbitMQ's main features are highlighted below [26]:

- **Reliability:** Offers a variety of features allowing to trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability
- **Flexible Routing:** Messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic.
- **Clustering:** Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker
- **Federation:** For servers that need to be more loosely and unreliably connected than clustering allows, RabbitMQ offers a federation model
- **Highly Available Queues:** Queues can be mirrored across several machines in a cluster, ensuring that even in the event of hardware failure your messages are safe
- **Multi-protocol:** RabbitMQ supports messaging over a variety of messaging protocols.
- **Many Clients:** There are RabbitMQ clients for almost any language
- **Management UI:** RabbitMQ ships with an easy-to use management UI that allows monitoring and controlling of the message broker.
- **Tracing:** RabbitMQ offers tracing support for troubleshooting purposes
- **Plugin System:** RabbitMQ ships with a variety of plugins extending it in different ways.

5.2.2.1 *Topic Exchange*

The AMQP 0-9-1 Protocol Model, which RabbitMQ implements, uses exchanges for message management. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with help of header attributes, bindings, and routing keys. A binding acts as a link that binds a queue to an exchange, while routing key is an attribute of the message. The exchange, depending on exchange type, takes this key into consideration in the process of routing the message to queues. Then, broker either delivers messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand. Exchanges, connections, and queues can be configured with parameters such as durable, temporary, and auto delete upon creation. Durable exchanges survive server restarts and last until they are explicitly deleted. Temporary exchanges exist until RabbitMQ is shut down. Auto-deleted exchanges are removed once the last bound object unbound from the exchange.

For our implementation **Topic Exchange** type was chosen and specifically the default pre-declared `amq.topic` for storing and transmitting all collecting data. Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange.

Then, broker either delivers messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand.
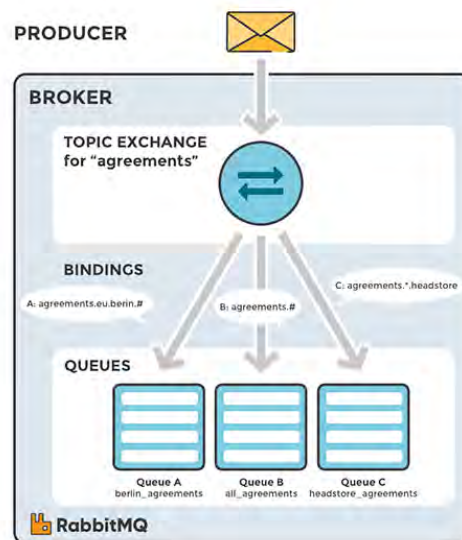
Figure 25 : Topic Exchange: Messages are routed to one or many queues based on a matching between a message routing key the routing pattern

The topic exchange type is often used to implement various publish/subscribe pattern variations, usually for multicast routing message scenarios.

5.2.2.2 *RabbitMQ Web STOMP Plugin*

For visualizing the statistics we collect from Docker Server we used Stomp technology and particularly its implementation in RabbitMQ [27]. The Web STOMP plugin is a simple bridge exposing the STOMP protocol over direct or emulated HTML5 WebSockets.

STOMP, Simple (or Streaming) Text Oriented Message Protocol, is a simple text-based protocol used for transmitting data across applications. It is much simpler and less complex protocol than AMQP, it is more similar to HTTP. STOMP clients can communicate with almost every available STOMP message broker, this provide easy and widespread messaging interoperability among many languages, platforms and brokers. STOMP does not deal with queues and topics, it uses a SEND semantic with a destination string. RabbitMQ maps the message to topics, queues or exchanges. Application consumers then subscribe to those destinations, in order to reach stored messages.

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. It is a different protocol from HTTP, although both protocols are located at layer 7 in the OSI model and, as such, depend on TCP at layer 4. The WebSocket protocol enables interaction between a web client (such as a browser) and a web server with lower overheads, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way (WebSocket protocol was standardized by the IETF as RFC 6455 in 2011 [28]) for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. The communications are done over TCP port number 80 (or 443 in the case of TLS-encrypted connections), which is of benefit for those environments which block non-web Internet connections using a firewall. The WebSocket protocol is currently supported in most major browsers including Google Chrome, Microsoft Edge, Internet Explorer, Firefox, Safari and Opera.

5.2.3 Chart JS Library

We wanted to enhance monitoring experience so we used an opensource Javascript library for the presentation of our data collection. The Chart JS library is an HTML5 based JavaScript library that employs the `<canvas>` element for creating animated, interactive and customizable charts and graphs [29].

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

55

5.2.4   Sinatra

Our ruby implementation used **Sinatra** as an elegant and effective way to invoke remote calls to the Docker Server. Sinatra is a free and open source software web application library and Domain Specific Language implemented in Ruby [30]. It is an alternative to other Ruby web application frameworks such as Ruby on Rails, Merb, Nitro, and Camping. Created by Blake Mizerany, Sinatra is Rack-based, which means it can fit into any Rack-based application stack, including Rails. Unlike Ruby on Rails, which is a Full Stack Web Development Framework that provides everything needed from front to back, Sinatra is designed to be lightweight and flexible. Sinatra is designed to provide with the bare minimum requirements and abstractions for building simple and dynamic Ruby web applications.

5.2.5   Bootstrap

Our application front-end was built with the use of one of the most popular frameworks, Bootstrap. Bootstrap is a free and open-source front-end framework for designing websites and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. Unlike many earlier web frameworks, it concerns itself with front-end development only [31].

5.3   Detailed description of the implementation

Having explained the basic elements our implementation consists of, we can proceed describing in detail our project.

5.3.1   The big picture
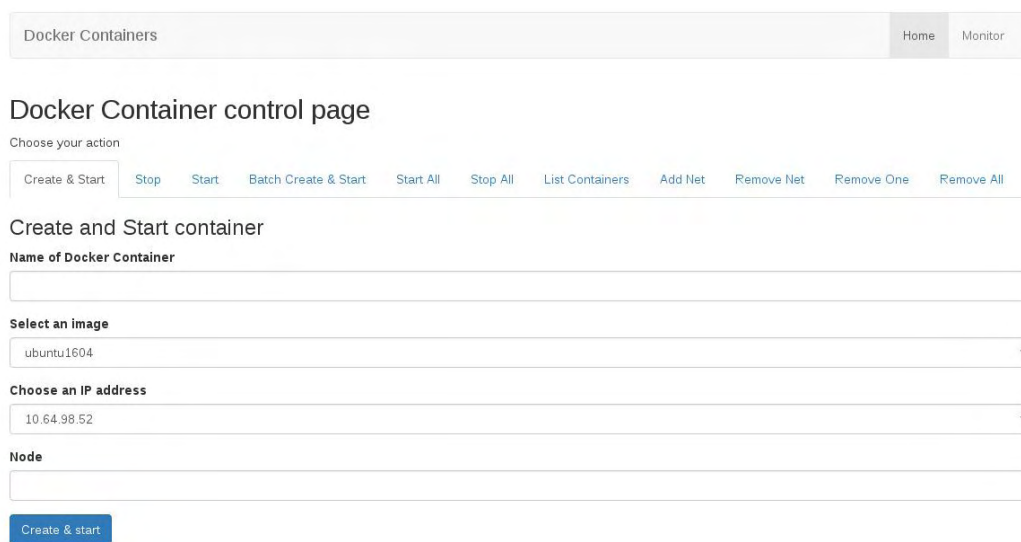
Our goal, as already explained, was to build a platform for creating and running rapidly a great amount of containers that offer the major advantages of a virtual machine but with minimal resource costs. Application residing in Nitos Server should provide us the capability of creating, running and stopping containers on a Testbed Node of our choice. Containers should have IP addresses on the range `10.64.98.52 -10.64.98.240` and the user should be able to use `ssh` to connect directly to the container. Adding a second network interface on a container was a feature also needed for specific scenarios.

5.3.2 Nitos Server Side

5.3.2.1 *Docker Control Page*

The main core of our application resides in the nitlab3.inf.uth.gr server. By calling the link `nitlab3.inf.uth.gr:4003/docker` we are introduced to our bootstrap designed Docker Control Page. This is the main page that gives us the capability to control container remotely by the use of a graphical, intuitive interface. Creating and running a single or a number of containers to the node we choose, is as simple as selecting values from the appropriate fields as seen in the main Docker Control Page below.

Figure 26 : Main control Page

Additional tools for stopping and removing containers from the node are also provided in the same interface. On creating containers, users can choose the name of the container, the operating system (ubuntu1604) wanting to use and also the IP address of the container from a predefined range. Batch Create and Run tab provides the very useful capability of creating and running a number of containers simultaneously. User provides a name and a base IP address and all containers created will have ascending IPs from the given range.

Figure 27 : Batch Creating & Running Containers

Interface also provides the ability of adding or removing a network to the container giving the opportunity for experimenting with networking. This procedure adds a new Ethernet Interface on the Docker container specified, giving it a "dummy" IP address in the range of `172.10.0.0/24`. User can then change the default address of the second interface (`eth1`) into one of the appropriate range `10.64.98.52 -10.64.98.240`. The tab which displays running or stopped containers on the node is a tool of great importance for administrator purposes, giving each time a correct view of containers states on the node.

5.3.2.2 *Monitor Page*

Front page of our application has also a link that transfers us to the monitor page. In this section, we are presented with a complete and detailed view of all Docker containers running on the node. A client utilizing the Websocket library Sock.js and the Stomp protocol connects to RabbitMQ server and presents collected data in a meaningful manner.
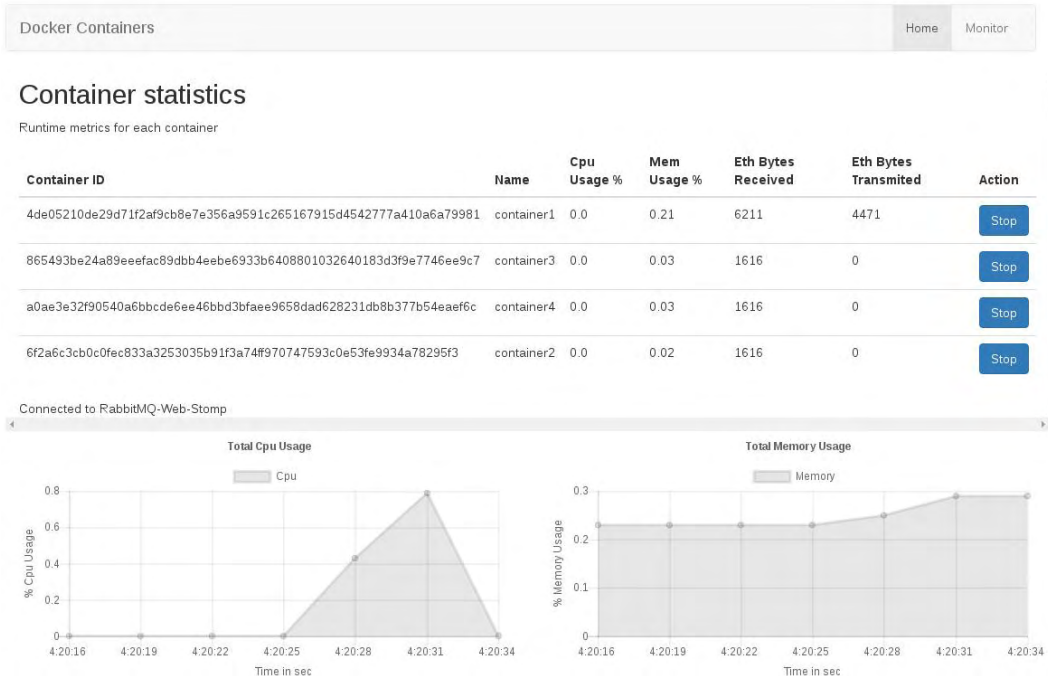
Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

58

Figure 28 : Monitoring Page

Each line on the table corresponds to a running container and information about its name, id, CPU, memory usage and eth I/O is displayed. At the end of each container line, a stop button exists, providing the extra feature of immediately stopping a container, for administrating purposes, without even leaving the page.

In order to have a complete view of the resources Docker containers acquire on the node, monitoring page has also the ability to provide graphic representations of the total CPU and memory usage of running containers. It should be noted that container statistics and graphical charts are dynamically updated in a 3 second interval. This supplementary view completes the overall picture an administrator should have on node running numerous containers.

5.3.3   Node side

Node is the side where Docker Engine configuration, networking and statistic collection takes place.

5.3.3.1   *Docker Configuration and networking*

By default, Docker Daemon will listen on socket `unix:///var/run/docker.sock` allowing only local connections by the root user. For accessing Docker Server remotely, though, TCP socket (`tcp://0.0.0.0:2375`) was enabled, allowing remote calls from our

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

59

application located at `nitlab3.inf.uth.gr` server. The default Unix domain socket was retained for debugging purposes. As mentioned in the previous paragraph, Macvlan driver was chosen for network configuration, as this model suited best the needs of our implementation. The second ethernet interface of the node (`eth1`) is configured with the IP address `10.64.98.51` and all containers created are assigned an address of the default network `10.64.98.0/23`, specifically in the range of `10.64.98.52 -10.64.98.240`. The following picture gives an overview of network configuration on the node.
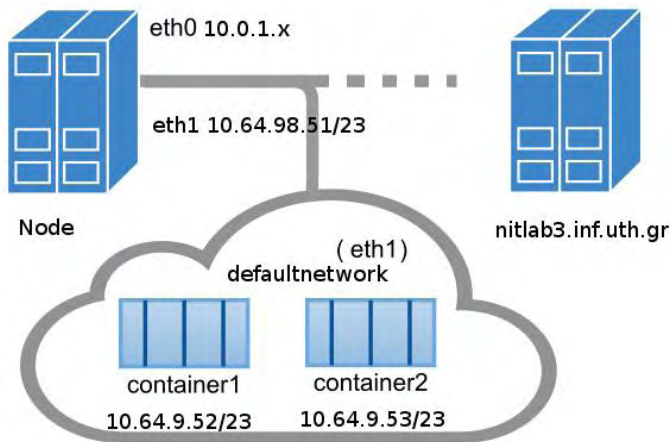
Figure 29

Figure 29: Network configuration on the Node

5.3.3.2 *Collecting Data*

On the node part of our implementation we needed a way to export the appropriate data (*CPU, memory usage, Eth I/O*) from the host and publish them to a RabbitMQ queue for collection and analysis. Pseudo files, the subsystem mechanism control groups' uses, was the way we could extract the information needed. For each cgroup subsystem (*memory, CPU, cpuacct, devices, etc.)* Docker creates a pseudo-file with the name (*full ID*) of the container containing corresponding information. For example in order to gather information about CPU usage the file to examine is `/sys/fs/cgroup/cpuacct/docker/container_id/cpuacct.usage`.

All remaining statistics were collected in a similar manner with the exception of network metrics that are not exposed directly by control groups. This is because network interfaces exist within the context of network namespaces. Metrics per interface is needed but since processes in a single cgroup can belong to multiple network namespaces, those metrics would be harder to interpret and not accurate information could be collected from cgroups. The most appropriate way was to examine the PID of the container and specifically the file `/proc/container_pid/net/dev`.

After collecting the needed data, we used the topic exchange type of RabbitMq server, described earlier in this document, for publishing stats. Statistics gathered, are then ready to be presented with the use of charts by the monitor client on the `nitlab3.inf.uth.gr` server.

5.4 Future Work

Admitting an application is completed, is definitely a characterization that has very little relevance with reality, especially in the case of being only in the first version. Procedure of debugging is non stopping and surely new bugs will be revealed in the future, as the application is being used. Besides this, new ideas for adding more features and tools will always be welcomed as there are a number of ways to expand and explore the possibilities of Docker Containers in the structure of Nitos Testbed.

Some features that could also be implemented in the future:

- Creating and adding new networks using applications' interface
- Extending container creation for multiple nodes
- Adjusting monitoring page for including multiple nodes
- Integrate with the existing Virtual Machine Creating Tool Platform for combining vm and container features

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

61

6   Table of images

7 References

[1] Wikimedia Foundation, Inc, "Cloud computing," [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing.

[2] A. Mouat, "Containers Vs Vm," in *Using Docker, Developing and Deploying Software With Containers*, O'Reilly Media, 2016, p. 355.

[3] Red Hat, "Overview of Containers in Red Hat Systems," [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/overview_of_containers_in_red_hat_systems/index.

[4] M. s. O. L. namespace. [Online]. Available: https://prefetch.net/blog/2018/02/22/making-sense-of-linux-namespaces/.

[5] J. Weissig, "Introduction to Linux Control Groups (Cgroups)," [Online]. Available: https://sysadmincasts.com/episodes/14-introduction-to-linux-control-groups-cgroups.

[6] m. s. k, "chroot, cgroups and namespaces," [Online]. Available: https://itnext.io/chroot-cgroups-and-namespaces-an-overview-37124d995e3d.

[7] T. L. Foundation, "Overview of Linux Kernel Security Features," 11 July 2013. [Online]. Available: https://www.linux.com/learn/overview-linux-kernel-security-features.

[8] Canonical Ltd., "Infrastructure for container projects," [Online]. Available: https://linuxcontainers.org/.

[9] L. L. C. P. Q. D. L. W. a. W. Z. Qi Zhang, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," p. 8, 5 Jul 2018.

Institutional Repository - Library & Information Centre - University of Thessaly
23/05/2024 20:13:02 EEST - 18.227.228.5

63

[10] J. Nickoloff, "4.1 Introducing Volumes," in *Docker in Action*, Manning Publications Co, 2016, p. 306.

[11] M. Church, "Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks," [Online]. Available: https://success.docker.com/article/networking.

[12] T. Vase, "Advantages Of Docker," [Online]. Available: https://jyx.jyu.fi/bitstream/handle/123456789/48029/URN:NBN:fi:jyu-201512093942.pdf?sequence=1.

[13] A. F. R. R. J. R. Wes Felter, "An Updated Performance Comparison of Virtual Machines," IBM Research Division, Austin, TX 78758 USA, July 21, 2014.

[14] N. Kratzke, "About Microservices, Containers," p. 5, 14 September 2017.

[15] A. C. C. O. T. C. F. Cristian Constantin Spoiala, "Performance comparison of a WebRTC server on," 13th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS, Suceava, Romania, May 19-21, 2016.

[16] H.-S. H. I.-Y. M. O.-Y. K. B.-J. K. Kyoung-Taek Seo, "Performance Comparison Analysis of Linux Container," Advanced Science and Technology Letters , 2014.

[17] HP Enterprise, "Linux container performance on HPE ProLiant servers," p. 22.

[18] S. Lowe, "Virtual Machines Vs. Containers: A Matter Of Scope," 28 May 2014. [Online]. Available: https://www.networkcomputing.com/cloud-infrastructure/virtual-machines-vs-containers-matter-scope/2039932943.

[19] "Performance Expectations of Container-Based Infrastructure," 24 June 2016. [Online]. Available: https://www.virtuozzo.com/connect/details/blog/view/performance-expectations-of-container-based-infrastructure.html.

[20] Docker, "Docker Engine API v1.24," 2017. [Online]. Available: https://docs.docker.com/engine/api/v1.24/.

[21] J. Turnbull, "Using the Docker API," in *The Docker Book*.

[22] Docker, "doackerd," 2017. [Online]. Available: https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-socket-option.

[23] Docker, "Get started with Macvlan network driver," 2017. [Online]. Available: https://docs.docker.com/v17.09/engine/userguide/networking/get-started-macvlan/.

[24] Wikipedia, "RabbitMQ," 28 September 2018. [Online]. Available: https://en.wikipedia.org/wiki/RabbitMQ.

[25] Ruby RabbitMQ Client Maintainers Team, "Bunny: all documentation guides," 2017. [Online]. Available: http://rubybunny.info/articles/guides.html.

[26] RabbitMQ, "What can RabbitMQ do for you?," [Online]. Available: https://www.rabbitmq.com/features.html.

[27] RabbitMq, "Introducing RabbitMQ-Web-Stomp," 2017. [Online]. Available: http://www.rabbitmq.com/blog/2012/05/14/introducing-rabbitmq-web-stomp/.

[28] Wikipedia, "WebSocket," 21 September 2018. [Online]. Available: https://en.wikipedia.org/wiki/WebSocket.

[29] Chartjs.org, "Chart.js," 2018. [Online]. Available: https://www.chartjs.org/.

[30] Wikimedia Foundation, Inc, "Sinatra (software)," 13 June 2018. [Online]. Available: https://en.wikipedia.org/wiki/Sinatra_(software).

[31] Wikimedia Foundation, Inc., "Bootstrap (front-end framework)," 19 October 2018. [Online]. Available: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework).