Zagreb, Cloud analysis

Container technologies

Davor Popović · Marko Ratkaj

sartura

June 15, 2019

About us

- Delivering solutions based on Linux, OpenWrt and Yocto
  - Focused on software in network edge and CPEs
- Continuous participation in Open Source projects
- www.sartura.hr

Introduction to GNU/Linux

- Written by Linus Torvalds
- First Linux prototypes publicly released in 1991
  - https://github.com/zavg/linux-0.01
- Version 1.0 release on March 14, 1994
- Written from the ground up

- Linux = operating system kernel
- GNU/Linux distribution = kernel + userspace (***Ubuntu***, ***Arch Linux***, ***Gentoo***, ***Debian***, ***OpenWrt***, ***Mint***, ...)
- Userspace = set of libraries + system software
  - Libraries = resources for other software
  - System software = platform/environment for other software to run (operating system)

- Each distribution is specific (depending on the purpose), but all follow the same principles:
  - Kernel = all are based on the Linux kernel
  - Folder hierarchy = all follow similar folder hierarchy (minor differences between distros)
  - Init systems = all have init systems (most use ***systemd***, some use ***SystemV*** (sysV) or other (openRC)
  - Package managers = all have package managers, but they differ between distributions (***apt***, ***pacman***, ***emerge***, ***yum***, etc.)

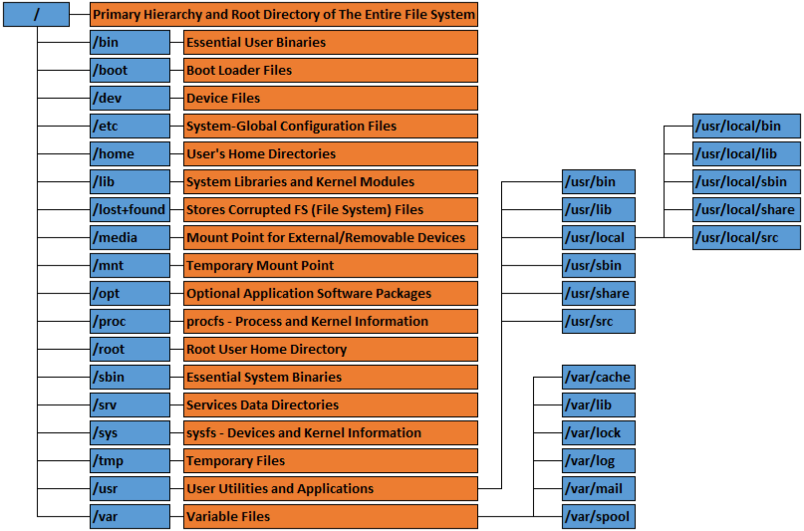

FIGURE 1 Linux file system structure

(GNU/Linux) kernel

- Core part of an operating system which manages system resources
  - CPU
  - RAM
  - Input/Output devices (keyboards, mouse, disk drives, printers, USB devices, network adapters, and display devices)
- The first program loaded into RAM by the bootloader (BIOS, U-Boot...)
- Acts as a layer between software/applications (userspace) and hardware

Linux kernel

- Operating systems have two spaces of operation:
  - *Kernel space* - protected memory space and full access to the device's hardware
  - *Userspace* - space in which all other application run
    - Has limited access to hardware resources
    - Accesses hardware resources via kernel
    - Userspace applications invoke kernel services with system calls

Example: opening a file

- Application wants to open a file → invokes library call for open → invokes system call for open → kernel opens a file from disk signaling the success result back to application in the same ladder
- A small programming detail for C language:
  - For opening files `fopen(const char *path, const char *mode)` from `stdio.h` is used (glibc)
  - Programs invoke this call from the source code with necessary parameters
  - `fopen` contains a system call for opening a file which is a function from `open` family of system calls

Example: opening a file

- Example taken from https://github.com/freebsd/freebsd/blob/master/lib/libc/stdio/fopen.c

```
FILE *
fopen(const char * __restrict file, const char * __restrict mode)
{
...
  if ((f = _open(file, oflags, DEFFILEMODE)) < 0) {
    fp->_flags = 0;^^I^^I^^I/* release */
    return (NULL);
  }
...
```

- system call `open` returns a pointer to structure `FILE` back to the caller

- Every other interaction follows the same principle but with a different system call

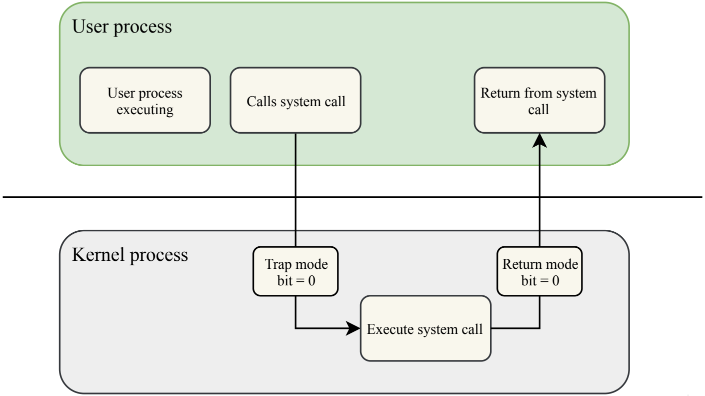

FIGURE 2 Linux kernel system calls

Linux kernel

- System calls provide the following services from the kernel to applications:
  1. Process creation and management
  2. Main memory management
  3. File Access, directory and file system management
  4. Device handling (I/O)
  5. Protection
  6. Networking, etc.

Linux kernel

- System calls can be grouped in the following types:
  1. Process control: end, abort, create, terminate, allocate and free memory.
  2. File management: create, open, close, delete, read file etc.
  3. Device management
  4. Information maintenance
  5. Communication

	User applications	E.g. bash, LibreOffice, GIMP, Blender, Mozilla Firefox, etc.				
User mode	**Low-level system components**	System daemons: systemd, runit, logind, networkd, PulseAudio, …	Windowing system: X11, Wayland, SurfaceFlinger (Android)	Other libraries: GTK+, Qt, EFL, SDL, SFML, FLTK, GNUstep, etc.	Graphics: Mesa, AMD Catalyst, …	
	C standard library	Up to 2000 subroutines depending on C library (glibc, musl, uClibc, bionic) (`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, …)				
Kernel mode	**Linux Kernel**	About 380 system calls (`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc.)				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: SELinux, TOMOYO, AppArmor, Smack				
		Hardware (CPU, main memory, data storage devices, etc.)				

TABLE 1 Layers within Linux

Virtualization

- Virtualization = creating a virtual version of hardware/software
- Term coined in 1960s with mainframe computers
- What can be virtualized? - basically everything
  - Hardware (with emulation)
  - Storage devices (with pooling of physical storage from multiple storage devices into a single logical storage device)
    - Can be applied on HDD, RAM
  - Networking, processes (using various kernel features that allow resource separation)
    - These are operating system features rather than applications
  - Software/applications
    - Running applications in separate machines/containers and making them portable
    - Running different services in separate machines/containers (cloud-based applications)

Virtualization Concepts

Two virtualization concepts:

- *Hardware virtualization* (full/para virtualization)
  - Emulation of complete hardware (virtual machines - VMs)
  - VirtualBox, QEMU, etc.
- *Operating system level virtualization*
  - Utilizing kernel features for running more than one userspace instance
  - Each instance is isolated from the rest of the system and other instances
  - Method for running isolated processes is called a ***container***
  - *Docker*, *LXC*, *Solaris Containers*, *Microsoft Containers*, ***rkt***, etc.

Virtual machines use hypervisors (virtual machine managers – *VMM*)

- Allows multiple guest operating systems (OS) to run on a single host system at the same time
- Responsible for resource allocation – each VM uses the real hardware of the host machine but presents system components (e.g. CPU, memory, HDD, etc.) as their own
- Two types of hypervisors (VMMs)
  - Type 1
    - Native or bare-metal hypervisors running directly on host's hardware and controlling the resources for Vms (Microsoft Hyper-V, VMware ESXi, Citrix XenServer)
  - Type 2
    - Hosted hypervisors running within a formal operating system environment.
    - Host OS acts as a layer between hypervisor and hardware.

- Containers do not use hypervisors
- Containers sometimes come with ***container managers***
  - Used for managing containers rather than resource allocation
- Containers use direct system calls to the kernel to perform actions
  - Kernel is shared with the host

- Idea behind containers is to pack the applications with all their dependencies and run them in an environment that is isolated from the host
- Two types of containers:
  - Full OS containers – contain full root file system of the operating system
    - Meant to run multiple applications at once
    - Provide full userspace isolation
    - *LXC*, *BSD jails*, *OpenVZ*, *Linux VServer*, *Solaris Zones*
  - Application containers – contain an application which is isolated from the rest of the system (*sandboxing*)
    - Application behaves at runtime like it is directly interfacing with the original operating system and all the resources managed by it
    - *Docker*, *rkt*

Parameter	VMs	Containers
Size	Few GBs	Few MBs
Structure	Full contained environment	Rely on underlying OS
Resources	Contains full OS with no dependencies on the underlying OS (e.g. Windows running on Linux and vice-versa)	Rely on underlying OS
Boot time	Few second overhead	Millisecond overhead

TABLE 2 VMs vs containers - Differences

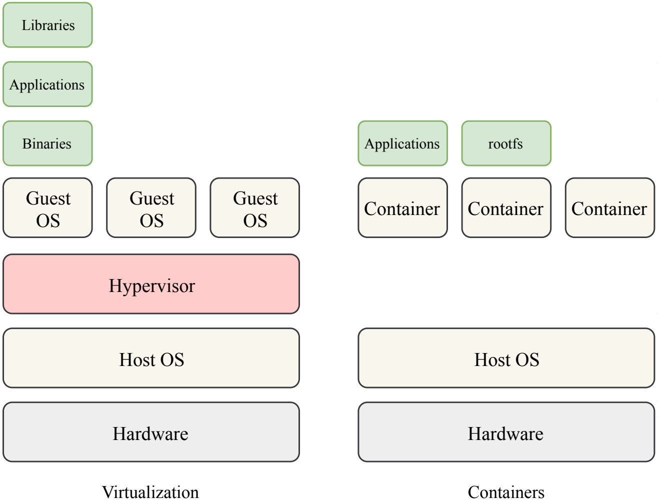

FIGURE 3 Virtualization vs Containers

Why use virtualization?

- Cost effective, resource savings
  - Multiple machines can be virtualized on a single machine
- Management
  - Everything can be managed from a single point, usually using a management software for virtual machines and/or containers
- Maintenance
  - Once deployed, machines can be easily switched for new machines if needed in the future

Linux Containers (LXC)

- LXC is a userspace interface for the GNU/Linux kernel containment features
  - Allows operating system level virtualization on GNU/Linux systems
- In-between ***chroot*** and complete VM
  - Sometimes referred to as ***chroot-on-steroids***
- Does not depend on hardware support for virtualization
  - Ideal for containerization/virtualization on embedded devices
- Configurable as a full feature file system (rootfs) or minimized for running single applications
- Relies ***heavily*** on kernel features

Kernel features

- LXC uses the following kernel features:
  - *Namespaces*
    - ipc, uts, mount, pid, network and user
- *Apparmor* and *SELinux* profiles
- *Seccomp* policies
- Kernel *capabilities*
- *CGroups*
- *Chroots (using pivot_root)*

Namespaces

- Lightweight process virtualization
- Namespaces = way of grouping items under the same designation
  - Kernel feature which organizes resources for processes
    - One process sees one set of resources
    - Another process sees another set of resources
    - One process cannot see other processes' set of resources
  - Each process has its own namespace – a set of resources uniquely allocated for that process
  - Namespaces allow processes to see the same parts of the system differently

Namespaces

- GNU/Linux kernel supports the following types of namespaces:
  - network
  - uts
  - PID
  - mount
  - user

Namespaces – Network

- Logical copy of the network stack containing its own network features (routes, firewall rules and network devices)
- Initial network namespace:
  - Initial network namespace = loopback device + physical devices + networking tables + …
- Every other newly created namespace contains only a loopback device
  - New network namespace instance = loopback device
- Every other device must be added manually
- To have network devices every container must be configured via configuration files

Namespaces - Network

- Network namespaces are completely isolated from each other
  - Host network namespace and container namespaces do not see each other
  - To get to the network, network namespaces need to be connected
  - E.g. the inside interface in the container must be connected to a physical interface on the host to go online

Namespaces – Network

- Connecting network namespaces
  - Done with a virtual interface called `veth`
  - `veth` is actually a ”virtual cable” handled by the kernel – one end is ”plugged” into a container, the other end into the host interface
  - Done with a virtual interface called `macvlan`
  - `macvlan` is a virtual interface which has its own MAC address and can be directly associated with a physical interface of the host

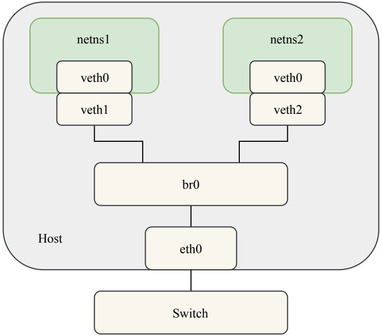

FIGURE 4 Network namespaces - veth

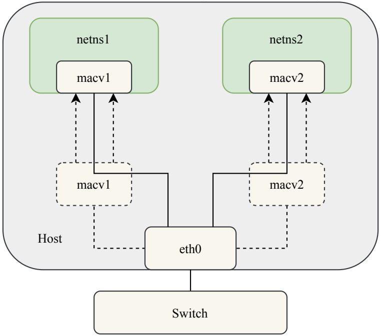

FIGURE 5 Network namespaces - macvlan

Namespaces – PID

- Process = current program in execution (set of instructions that are currently occupying CPU cycles)
- PID = way of organizing processes inside of the kernel by a unique number – each process is assigned a number
  - Special case regarding process with PID 1 – the first process started when OS is booted up (called init system) which starts all other processes
  - GNU/Linux supports multiple init systems – ***systemd***, ***systemV***, ***openRC***, etc.
  - Processes communicate with signals (e.g. `kill` signal)
- PID namespaces kernel feature enables isolating PID namespaces
  - Processes in different PID namespaces can have the same PID

Namespaces - PID

- Kernel creates namespace one – NS1
- Kernel creates namespace two – NS2
- NS1 contains PID 1 and all other PIDs
- NS2 contains PID1 and all other PIDs
- NS1 can see PID 1 of NS2 but with some other PID (e.g. PID 10001) and all other PIDs of NS2
- NS2 can not see PIDs from NS1
- This way, isolation is achieved from these namespaces – processes inside NS2 are only functional if NS2 is isolated from NS1
    - In NS2 a process cannot send signal (e.g. `kill`) to a host machine

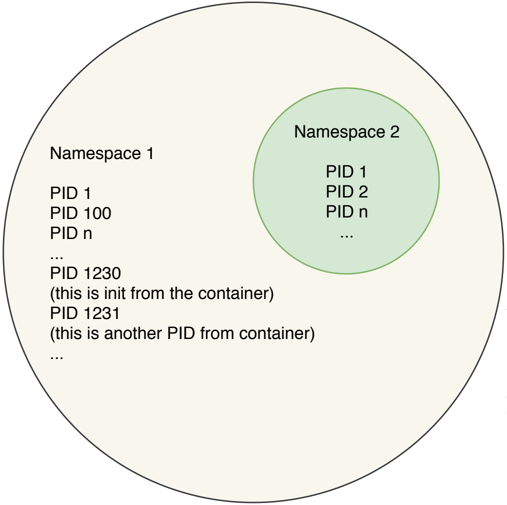

FIGURE 6 Namespaces - PID

CGroups

- PID namespace allows processes to be grouped together in isolated environments
- Group of processes (or a single process) needs access to certain hardware components
  - E.g. RAM, CPU, ...
- Kernel provides the control groups (CGroups) feature for limiting how process groups access and use these resources

CGroups

- 4 main purposes
  - Limiting resources = groups can be set to not exceed a pre-configured memory limit (i.e. this group of processes can access X MB of RAM)
  - Prioritization = some groups may get a larger share of CPU utilization (i.e. this group of processes can utilize 43% if CPU 1, while another group only 5%)
  - Accounting = measures a group's resource usage (i.e. this group of processes has been using 5% of CPU)
    - Useful for statistics
  - Control = used for freezing, snapshoting/checkpointing and restarting

CGroups

- CGroups uses a Virtual File System
  - `/sys/cgroups`
  - All CGroups actions are performed via file system actions

Working with LXC

- Install LXC package with your distribution's package manager
- Kernel must have all the required features enabled
  - Some distributions may come with a complete required feature set (e.g. Ubuntu)
  - Some distributions do not come with a complete required feature set
  - Some distributions require manual kernel configurations (e.g. Gentoo)
  - After installing LXC, it is required to check that the kernel has everything it needs with `lxc-checkconfig` command

Working with LXC

- Each container needs its own configuration file
- Each container needs its own root file system which will run in side of the container
  - Root file system contains all the necessary libraries, applications and environment settings
  - Root file system needs to be either manually prepared or downloaded from remote online repositories
- Place configuration file and root file system on the same location:
  - `/var/lib/lxc/<container_name>/`

Configuring the container

- Container configuration defines the following components:
  - Capabilities – what the container is allowed to do from an administrative perspective
  - Cgroups – which resources of the host are allowed for the container (e.g. configuring which devices can the container use)
  - Mount namespaces – which of the host folders/virtual file systems will be allowed for mounting inside the container (virtual file systems such as `proc` or `sys`)
  - Network namespaces – which devices will be created inside the container and how they connect to the outside network

- First part of the file handles capabilities
- A list of all the capabilites dropped (not allowed) for the container
- Usually best to consult with `man` pages http://man7.org/linux/man-pages/man7/capabilities.7.html

```
lxc.cap.drop = mac_admin
lxc.cap.drop = mac_override
lxc.cap.drop = sys_admin
lxc.cap.drop = sys_boot
lxc.cap.drop = sys_module
lxc.cap.drop = sys_nice
lxc.cap.drop = sys_pacct
lxc.cap.drop = sys_ptrace
lxc.cap.drop = sys_rawio
lxc.cap.drop = sys_resource
lxc.cap.drop = sys_time
lxc.cap.drop = sys_tty_config
lxc.cap.drop = syslog
lxc.cap.drop = wake_alarm
```

- The second part is CGroup – what is allowed for this container (as mentioned before, a container can be seen as a set of processes grouped together, meaning this shows what these processes can access)
- It uses traditional Linux desigantions for devices (try running `ls -l /dev` which will list all the devices with corresponding major:minor numbers)
- E.g. bold entry is for console on PC

```
lxc.cgroup.devices.deny = a
lxc.cgroup.devices.allow = c 1:1 rwm
lxc.cgroup.devices.allow = c 1:3 rwm
lxc.cgroup.devices.allow = c 1:5 rwm
lxc.cgroup.devices.allow = c 5:1 rwm
lxc.cgroup.devices.allow = c 5:0 rwm
lxc.cgroup.devices.allow = c 4:0 rwm
lxc.cgroup.devices.allow = c 4:1 rwm
lxc.cgroup.devices.allow = c 1:9 rwm
lxc.cgroup.devices.allow = c 1:8 rwm
lxc.cgroup.devices.allow = c 1:11 rwm
lxc.cgroup.devices.allow = c 136:* rwm
lxc.cgroup.devices.allow = c 5:2 rwm
lxc.cgroup.devices.allow = c 254:0 rwm
lxc.cgroup.devices.allow = c 10:200 rwm
```

- Some metadata about the container
- Where is the rootfs located
- Hostname of the container
- What to mount from the host
- `/proc` and `/sys`

```
# Distribution configuration
lxc.arch = x86_64
# Container specific configuration
lxc.rootfs.path = dir:/var/lib/lxc/openwrt/rootfs
lxc.uts.name = openwrt
# Mount entries
lxc.mount.entry = /proc proc /proc nodev,noexec,nouid 0 0
lxc.mount.entry = sysfs sys sysfs default 0 0
```

- Network namespace configuration
- We can read this as follows:
  - Create eth0 device inside a container
  - Use a veth (virtual cable) to connect this eth0 from container to lxcbr0 interface (a bridge interface) on the host
- It can be configured in many different ways – depends on the use case

```
# Network configuration
lxc.net.0.type = veth
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
lxc.net.0.name = eth0
```

Working with LXC

- Once configuration and root file system are ready issue `lxc-ls`
  - If the container is configured properly and its root file system is valid, the container should appear on the list
- Start the container with

```
lxc-start -n <container_name>
```

- There will be no output, but the container should start
- Check that the container is running

```
lxc-info -n <container_name>
```

- Container is running in the background, and we can now run applications inside of the container

Working with LXC

- Entering the shell of the container (attaching inside of the container)

```
lxc-attach -n <container_name>
```

- From the shell, it is possible to do everything as on host GNU/Linux system
- To exit, type `exit`
- To stop the container

```
lxc-stop -n <container_name>
```

- This demonstration is a simple case of a single container created by root user and with no particular functionalities – so what can be done with the container?

Working with LXC

- Containers generally have two purposes

  1. *Development* – when developing, containers allow developers to share the same environment – once a container is set, it can be deployed on many machines with exactly the same libraries and user space

  2. *Deployment* – containers allow quick deployment. It is possible to run applications in the cloud inside of the container allowing external access to them.

     - E.g. running a container with web servers allowing access from the outside of the network to each web server. If the security is compromised, in theory the attacker would be inside of the container isolated from the rest of the cloud. This method allows running multiple web servers on a single physical machine all listening on the same port. The additional effort is in creating firewall rules between the outside network and containers.

Working with LXC

- Cloud machines allow running tens of thousands containers at once
  - All hell breaks loose when all these containers need to be administered and supported in the future
- LXD was created to answer challenges with managing huge numbers of containers

LXD

- LXD = container manager, useful when running and configuring huge amounts of Linux containers
- Concept
  - Server + client side (communicating over REST API)
  - Accessible locally and remotely over network
  - Command line tool for working with containers
- Supports the full LXC feature set
  - By default, LXD creates unprivileged containers (what we demonstrated is the creation of privileged containers by the root user which might have some security issues)

- Scalable – allows creating containers over thousands of hardware nodes (PC, servers, embed-ded)
- Intuitive – simple API and CLI
- Image-based – huge variety of distributions available over networkd
- Allows live migrations
- Resources control (CGroups – CPU, memory, I/O, …)
- Device pass-through (allowing USB, GPU, character devices, …)
- Networking
- Storage (storage pools and volumes)

LXD - Prerequisites

- Initialized LXD daemon
- Root file system and metadata
  - Metadata = data about the container
- Container image
  - Image from which the container will be created
  - Image = rootfs + metadata
- Container profile
  - Basic container configuration

LXD init

- Configuring the LXD daemon

```
lxd init
```

Prepare rootfs

- Create `gentoo` directory inside the home directory

```
cd ~
mkdir gentoo
```

- Copy compressed root file system on that location

```
cp gentoo-rootfs.tar.gz ~/gentoo
```

- In the same directory, create metadata file for the container
    - Metadata describes basic information about the container
    - Can be written in YAML format or JSON (examples below use YAML)

- Minimal metadata template file

```
vim metadata.yaml
```

```yaml
architecture: "aarch64"
creation_date: 1554382805      # mandatory and must be valid. Each container must have unique. Take this value: date +%s
properties:
        architecture: "aarch64"
        description: "Example of Gentoo virtual router"
        os: "Gentoo Linux"
        release: "0.1"
        variant: "Custom"
```

Import rootfs and metadata as image

- Compress both image and metadata

```
tar cf gentoo-matadata.tar metadata.yaml
```

- Import compressed root file system and metadata into LXD

```
lxc image import gentoo-metadata.tar.gz gentoo-rootfs.tar.gz --alias GentooImage
```

- If everything went well, the image should appear on LXD image list

```
lxc image list
```

LXD

- At this point, the container is still not created - only the image for the container is prepared (root file system + metadata)
- Containers do have default profiles but in practice each container needs a profile with different configuration
- Profile = initial configuration for the container (networking, storage, hostname, …)
- The example below creates a simple profile with one network interface connected to the host physical interface with macvlan virtual interface (see Figure 5)

Prepare container profile

- Create minimal YAML file to define the container profile: `vim gentoo-profile.yaml`

```
config: {}
description: Gentoo LXD profile
devices:
  eth0:
    name: eth0
    nictype: macvlan
    parent: eth1^^I^^I^^I^^I#this can vary, depending on how is the interface named on host ( enpXXX, ethXXX, enoXXX...)
    type: nic
  root:
    path: /
    pool: workstation-pool
    type: disk
name: default
```

LXD

- At this point the profile is not attached to any container, and is actually just a file
- First step is to create a profile for LXD and apply the YAML file with the profile

```
lxd profile create Gentoo-profile
lxd profile edit Gentoo-profile < gentoo-profile.yaml
```

- This profile can be used over n number of containers

LXD

- Next step is to apply the profile to a container
- First, container must be created from the image

```
lxc init GentooImage GentooContainer
```

- Then, apply the profile to the initialized container

```
lxc profile apply Gentoo-Profile GentooContainer
```

Verifying the process

- To verify what has been done (and if it went OK)
  - Checking images

    ```
    lxc image list
    ```
  - Checking containers

    ```
    lxc ls
    ```
  - Checking available profiles

    ```
    lxc profile list
    ```

Verifying the process

- If necessary, profiles can be modified on the fly and all changes applied in real time
  - Checking a specific profile

```
lxc profile show Gentoo-profile
```

  - Modifying a specific profile

```
lxc profile edit Gentoo-profile
```

Starting the container

- The container is ready to start at this point

```
lxc start GentooContainer
```

- How does this all fit together?
  - Inspect `htop`
  - Network namespace

Starting the container - next steps

- Run any application inside the container
  - To attach inside the container, execute `bash`

    > `lxc exec GentooContainer -- /bin/bash`

  - From this shell we can do everything as regular Linux users
  - Any other application can be run in the same way

    > `lxc exec GentooContainer -- /bin/bash`

- With this principle different servers and applications can be run inside the container to isolate them from the rest of the host system
- Once a container is stopped (or destroyed), applications have no power

SDN – practical application of Containers

- The core network component = virtual router
- Virtual router = LXC container running Linux with networking set for router performance
  - What is a router?
    - Forwards packets between two different networks
    - One side is LAN one side is WAN – with NAT in-between

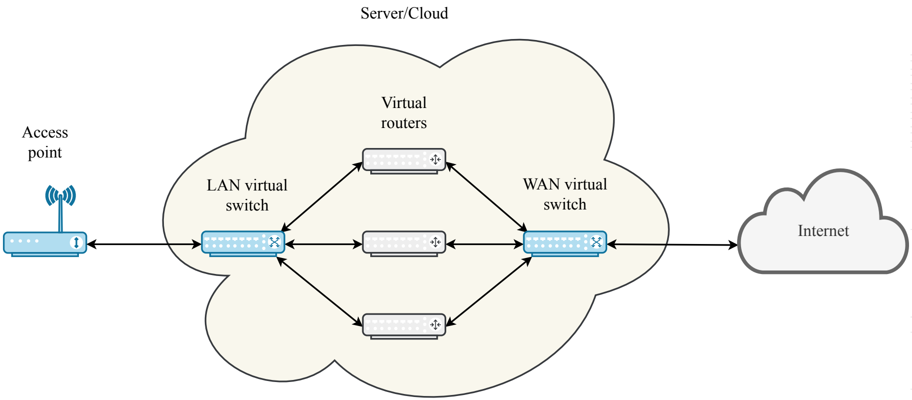

FIGURE 7 Real life example - Software defined networking (SDN)

- Each router needs to have profiles defined to support two sides of the network (LAN and WAN)
- How can this be done?

```
eth0:
  name: eth-wan
  nictype: macvlan
  parent: eth2
  type: nic
eth1:
  name: eth-lan
  nictype: macvlan
  parent: tap1
  type: nic
```

- How to configure network inside the container? Remember systemd?
- LAN should behave as a DHCP server
- WAN should behave as a DHCP client
- Network Addres Translation (NAT) in between
  - Keeps track who is going where on the Internet
  - Achieved with Linux firewall (`iptables`)

Container technologies

davor.popovic@sartura.hr · marko.ratkaj@sartura.hr