Zagreb, NKOSL, FER

Container technologies

Marko Golec · Juraj Vijtiuk · Jakov Petrina

sartura

April 11, 2020

About us

- Embedded Linux development and integration
- 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

sartura

- Linux = operating system kernel
- GNU/Linux distribution = kernel + userspace (***Ubuntu***, ***Arch Linux***, ***Gentoo***, ***Debian***, ***OpenWrt***, ***Mint***, …)
- Userspace = set of libraries + system software

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

User mode	User applications	E.g. bash, LibreOffice, GIMP, Blender, Mozilla Firefox, etc.				
	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 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*, *systemd-nspawn*, *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

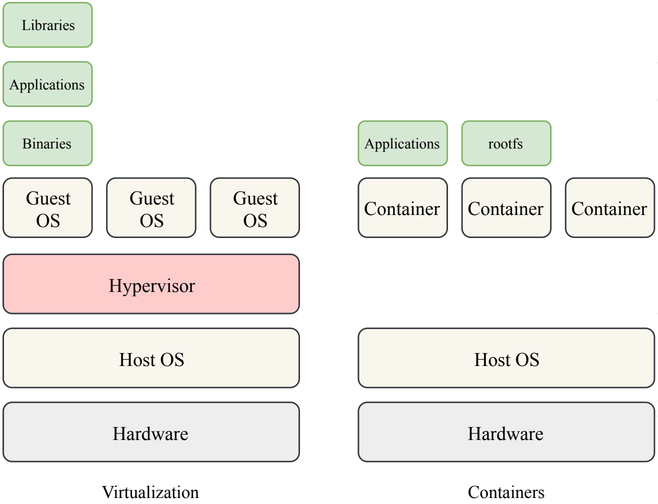

FIGURE 1 Virtualization vs Containers

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 Features

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
  - time

Namespaces - PID

- PID namespaces kernel feature enables isolating PID namespaces, so that different namespaces can have the same PID
- Kernel creates two namespaces - NS1 and 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

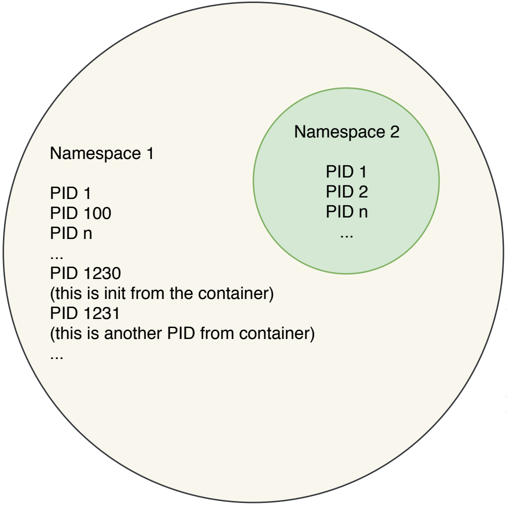

FIGURE 2 Namespaces - PID

Namespaces — Linux configuration

- Namespace feature requires build-time kernel configuration
- Example configuration from a system with Docker and LXC

```
...
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
...
```

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 — Linux configuration

- CGroups feature requires build-time kernel configuration
- Example configuration from a system with Docker and LXC

```
...
CONFIG_CGROUPS=y
CONFIG_BLK_CGROUP=y
CONFIG_CGROUP_WRITEBACK=y
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_RDMA=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_HUGETLB=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_CGROUP_PERF=y
CONFIG_CGROUP_BPF=y
# CONFIG_CGROUP_DEBUG is not set
CONFIG_SOCK_CGROUP_DATA=y
CONFIG_BLK_CGROUP_RWSTAT=y
CONFIG_NET_CLS_CGROUP=y
...
```

Linux Containers (LXC)

sartura

- 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

Container security

- LXC uses the following Linux features to improve security:
  - *Namespaces*
    - ipc, uts, mount, pid, network and user
    - user namespaces, privileged and unprivileged containers
  - *Apparmor* and *SELinux* profiles
  - *Seccomp* policies
  - Kernel *capabilities*
  - *CGroups*
  - *Chroots (using pivot_root)*

Working with LXC

- Each container needs its own configuration file
- Each container needs its own root file system
  - The root file system contains all the necessary libraries, applications and environment settings
  - Needs to be manually prepared or downloaded from remote online repositories
- Place the configuration file and root file system in the same location
  - `/var/lib/lxc/<container_name>/`

Configuring the container

- `/etc/lxc/default.conf`, `$HOME/.config/lxc/default.conf` or container specific in container directory
- 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 designations 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 runs in the background and we can now run applications inside it

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?

LXD Container Manager

sartura

LXD

- Container manager, useful when running and configuring large numbers 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)

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
```

```
architecture: "aarch64"
creation_date: 1554382805      # Mandatory, must be unique for each container. 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
```

Prepare container profile

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

```yaml
config: {}
description: Gentoo LXD profile
devices:
  eth0:
    name: eth0
    nictype: macvlan
    parent: eth1    #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, the 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

systemd-nspawn

systemd-nspawn

- Part of the ***systemd*** project — a ***chroot*** alternative
  - ”Fully virtualizes the file system hierarchy, as well as the process tree, the various IPC subsystems and the host and domain name” — `SYSTEMD-NSPAWN(1)`
- Concept
  - Command line tool for working with containers
  - Integrates with ***systemd*** on the host via the ***systemd-machined*** virtual machine and container registration manager

systemd-nspawn — Prerequisites

- *systemd-nspawn* does not require special metadata to boot containers
- Usually, container root filesystem directory is placed in the `/var/lib/machines` directory, which can also be a symlink to a directory
- Container filesystems for some distributions can be created via appropriate utilities e.g. `pacstrap` for Arch Linux, and `debootstrap` for Debian

Launching containers using the CLI

- Launching a container is straightforward:

```
systemd-nspawn --boot --directory=/var/lib/machine/gentoo
```

- CLI tool allows specifying configuration options such as capabilities and networking

```
systemd-nspawn --boot --directory=/var/lib/machine/gentoo
    --capability=CAP_NET_ADMIN \
    --network-macvlan=eth0
```

Starting containers as a service

- When using the CLI tool containers are foreground processes
- Containers can be run in the background with the `systemd-nspawn@.service` unit
  - Enable and start the `machines.target`

    ```
    systemctl enable machines.target
    ```

  - Enable and start the `systemd-nspawn@<machine>.service`, where `<machine>` specifies an nspawn container in `/var/lib/machines`

    ```
    systemctl enable --now systemd-nspawn@gentoo.service
    ```

- Optionally configure container in the `/etc/systemd/nspawn/<machine>.nspawn` file

Example: setting up networking

Setting up network inside the container

- Practical example – setting up the network inside the container
- As defined in the container profile, the interface from the container is connected directly to a physical interface on the host machine with `macvlan` interface
- `macvlan` creates a new interface with a different MAC address than the host one and allows traffic to go directly through (as opposed to a bridge where it has to hit the bridge first)

- In theory, there is nothing wrong with this configuration
- In practice, network has to be configured inside the container as well
- The user can either set up static IP on the inside interface or set dynamic IP (meaning that the IP on the container interface will be offered by someone else – DHCP server running somewhere in the network)

- How?
  - `systemd`
  - Daemon in role of PID 1 – master process, initial process from which all other processes are spawned
  - One of the domains directly under `systemd` control is networking
  - As any other program, `systemd` and its components are configured with different configuration files located under `/etc/system/(network)`

- Listing out `/etc/systemd/network` might show that it is empty so a new file containing network configuration must be created
- A good practice is to name the file `<file_name>.network`
- This file will define the following:
    - Match the given interface
    - Assign it with IPv4 address from a DHCP server

- Create a file

```
vim /etc/systemd/network/eth0.network
```

- Add the following:

```
[Match]              # Match this interface
Name=eth0

[Network]            # Assign IPv4 address from a DHCP server
DHCP=ipv4

[DHCP]
RouteMetric=10
```

- Restart systemd networking service

```
systemctl restart systemd-networkd
```

- At this point, on the `eth0` interface an IP address should appear and it should be from the same subnet as the IP address offered on the physical interface of the host
- Try pinging the Internet

```
ping 8.8.8.8
```

Container technologies

marko.golec@sartura.hr · juraj.vijtiuk@sartura.hr · jakov.petrina@sartura.hr

Feedback form: forms.gle/WKPBDoS69gssafAE9

info@sartura.hr · www.sartura.hr