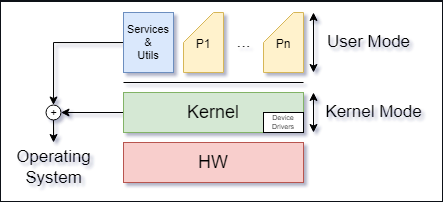
**System Shared Environment**



**IPC - Inter-Process Communication**

**-** Shared memory;

- Data exchange via Kernel

**1.Named Pipes (FIFO)**

**2.Unix Domain Sockets.**

**Named Pipes**

**- FIFO**: First In First Out.

- Extension to the traditional pipe concept on Unix.

- Created by the ***mkfifo*** function.

**- Unidirectional** communication.

**Unix Domain Sockets**

- AF\_UNIX: Address Family Unix.

**- Bidirectional** communication.

- Functions:

1. socket() -> Creates a socket.

2. bind() -> bind a name to a socket.

3. listen() -> listens for a connection on a socket.

4. accept() -> accepts a connection on a socket.

5. connect() -> initiates a connection on a socket.

-----------

**Systemd**

Systemd is a suite of **basic building blocks for a Linux system**. It provides a system and service manager that runs a **PID 1** and starts the rest of the system. It also provides a D-Bus-activated system bus, which other processes can use to talk to the system manager and interact with it.

**- PID 1**: Process ID 1 is the first process started by the kernel when the system boots.

- Systemd is a **system and service manager**.

**- SysVinit**: System V initialization is an old system for starting and stopping services on Unix-like operating systems.

**1. Run levels**: Run levels are a way of categorizing the state of the system; they’re used to determine which services should be started and stopped; In Systemd, run levels are called **targets**.

In linux, services are also called **daemons**.

**Daemons** have a few characteristics:

1. Is a process that runs in the background and is not interactive aka the user can’t access it.

2. They have no controlling terminal on their own from the user’s perspective from the desktop.

3. They continue to exist and operate regardless of any user being logged into the server if the computer is on.

**Unit block**

A unit block is a configuration file that describes a system resource such as a device, a mount point, a swap file, a service or a target.

Is the basic building block of **Systemd**.

**-- Unit types**

**- Service**: A service unit describes how to manage a service or application on the server. This will include how to start or stop a service, under which circumstances it should be automatically started, and the dependency and ordering information for related software.

1. .service

**- Socket**: A socket unit file describes a network or IPC socket, or a FIFO buffer that Systemd uses for socket-based activation. This always has an associated .service file that will be started when activity is seen on the socket that this unit defines. When a socket is inactive, it is not listening for connections therefore the service cannot be started.

1. .socket

**- Target**: A synchronization point for other units. Usually used to start enabled services on boot. Targets are used for grouping and ordering units. They are somewhat of a rough equivalent to runlevels at different targets, different services, sockets and other units are started. Unlike runlevels, they are much more free-form and can easily make their own targets for ordering units. Targets also have dependencies among themselves.

1. .target

**-- Unit status**

**- Loaded**: indicates whether the unit is loaded or not. It can be loaded or not found. Has the enabled status; if the unit is not loaded, it cannot be enabled.

**- Active**: indicates whether the unit is active or not. It can be active or inactive.

**- Docs**: contains the documentation of the unit.

**- Main PID**: contains the Process ID of the main process of the unit.

**- CGROUP**: contains the control group of the unit. Control group is a mechanism for **controlling the resources** used by a set of processes.

**- TriggeredBy**: contains the units that triggers the unit. Usually, a **service is triggered by a socket**.

**- Triggers**: contains the units that the unit triggers. Usually, a **socket triggers a service**.

**Note** - Enabled vs Active vs Loaded

- Enable and disabled indicate whether the unit is configured to start automatically when the system boots.

- Loaded and not found indicate whether the unit is loaded in the system.

- Active and inactive indicates whether the unit is currently running.

**-- Managing Units**

*systemctl* - is a command-line tool for controlling the systemd system and service manager;

To use the *systemctl* command, you need to install the systemd package (which is already installed on most Linux distributions): *sudo apt install libsystemd-dev*;

Here are some of the most common commands:

*- systemctl list-units*- lists all units

*- systemctl status <unit>*- shows the status of a unit

*- systemctl start <unit>*- starts a unit

*- systemctl stop <unit>*- stops a unit

*- systemctl restart <unit>*- restarts a unit

*- systemctl enable <unit>*- enables a unit

*- systemctl disable <unit>*- disables a unit

*- systemctl list-dependencies <unit*> - lists dependencies of a unit.

[**Service Templates**](https://www.freedesktop.org/software/systemd/man/systemd.service.html#Service%20Templates)

A service template is a unit that is used to create multiple instances of a service. It is a template that is instantiated for each instance of the service. The template is used to define the common properties of the service, and the instances are used to define the properties that are specific to each instance.

- It is possible to create a service template by adding the @ character to the end of the service unit file name: service@.service;

- To create a service instance, you need to start the service template with the @ character and the argument that will be passed to the template: sudo systemctl start service@argument;

[**Installing a Service**](https://linuxhandbook.com/create-systemd-services/)

- To install a service, you need to create a service unit file in the /etc/systemd/system directory;

- The service unit file must have the .service extension

Uma imagem com texto, captura de ecrã, ecrã

Descrição gerada automaticamente - The service unit file must have the following structure:

The [**Unit**] section contains the options that are **common to all types of units**:

- **Description** - contains the description of the service;

- **After** - contains the units that the service will be started after;

- **Before** - contains the units that the service will be started before;

The [**Service**] section contains the options that are **specific to services**:

- **ExecStart** - contains the command that will be executed when the service is started;

- **Type** - contains the type of the service;

- *simple* - the service will be started in the foreground;

- *forking* - the service will be started in the background;

- *oneshot* - the service will be started only once;

- *dbus* - the service will be started as a D-Bus service;

- *notify* - the service will be started as a notification service;

- *idle* - the service will be started as an idle service;

- **Restart** - contains the restart policy of the service;

- *no* - the service will not be restarted;

- *on-failure* - the service will be restarted if it fails;

- *on-abnormal* - the service will be restarted if it fails or is terminated abnormally;

- *on-watchdog* - the service will be restarted if it fails or is terminated by a watchdog timeout;

- *on-abort* - the service will be restarted if it fails or is terminated by an abort signal;

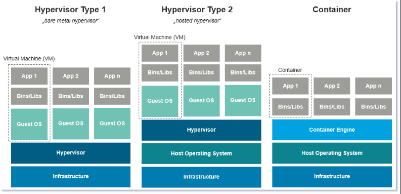
- *always* - the service will be restarted always.

- **etc**.

The [**Install**] section contains the options that are **common to all types of units** (this is the section that is used to enable the service):

- **WantedBy** - contains the targets that will be used to start the service; this is similar to the After option in the [Unit] section, but is used to **specify systemd targets**;

- **RequiredBy** - contains the targets that will be used to start the service; this is similar to the Before option in the [Unit] section, but is used to **specify systemd targets**;



**Hypervisor and containers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Soft.** | **Repr.** | **Ex.Env.** | **OS?** | **Arc?** | **Sup OS?** |
| Ja.Sc | .js | Node.js | N | N | Y |
| Java | .jar/.cl | JVM | N | N | Y |
| C/++/… | Exe&libs | OS | Y | Y | Y |
| Dockerfile | docker Image | Docker System | Y | Y | N |
| Assembly | binary | Machi. | N | Y | N |

**Virtual machines**

A virtual machine is a virtualization/emulation of a computer system. There are two types of VM’s:

**- Process VM’s** (eg: JVM, Node.js): they are not real VM’s, just processes that run in the background on the host’s OS.

- Not isolated from the host’s OS, they can access the host’s OS resources.

- The main purpose of these VM’s is to **translate the instruction set** of the guest’s OS to the **Instruction Set** of the host’s OS

- Also known as **Virtual Executions Environment**.

**- System VM’s** (eg: VirtualBox, etc): they are real VM’s, running on host’s OS yet isolated from host’s OS resources

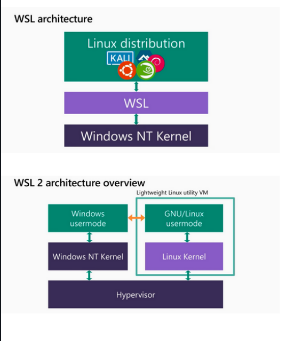
- Isolated from the host’s OS, they can’t access the host's OS resources.

- These are managed by a **Virtual Machine Monitor** (VMM) also known as **Hypervisor**.

**Note**: Emulator vs Simulator

- **Emulator** is a software that **emulates the hardware** of a device.

- **Simulator** is a software that **simulates the software** of a device.

**Hypervisor**

Hypervisor is a software, firmware or hardware that creates and runs virtual machines.

- A computer on which a hypervisor runs one or more virtual machines is called a **host machine** and each virtual machine is **called guest** machine.

- A hypervisor may allow multiple **guests machines** to run concurrently on a single host machine.

- A host machine may run **multiple hypervisors** simultaneously with each hypervisor supporting multiple guest machines.

- A hypervisor may also be referred to as a **virtual machine monitor (VMM)**.

There are **2 types** of hypervisors:

**- type 1**: native or bare-metal hypervisors

- VMM and VMs **run directly on the hardware**

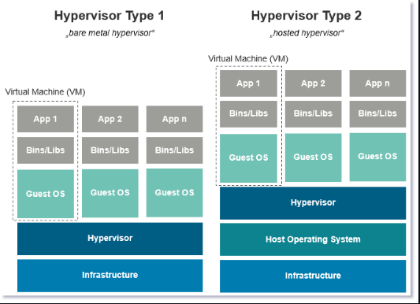
- Runs directly on the host’s hardware to control it and manage the guest’s operating systems

**- type 2**:

- VMM run on top of an OS and VM’s run on top of a VMM.

- Runs on top of an **OS** as a software application.

- The host’s OS is responsible for managing the hardware and the hypervisor is responsible for managing the guest OSes

 - Address translation occurs twice: once by the host’s OS and once by the hypervisor.

Note: **Paravirtualization**

Paravirtualization is a type of virtualization that allows Virtual Machines (VM’s) to run on host’s computer with **minimal modification to the host’s kernel**. This is achieved by creating a thin layer of software between the virtual machine and the host’s hardware which allows the **virtual machine to communicate with the host’s resources directly** rather than emulating them.

In contrast to full virtualization, paravirtualization **requires that the guest operating system be modified to work in the virtualized environment**, and it requires a paravirtualized hypervisor.

One of the main advantages of paravirtualization is that it can **improve the performance** of virtual machines by **reducing the overhead caused by hardware emulation**. It also allows for greater control over the virtualized environment, which can be useful in high-performance computing and other resource intensive applications.

However, paravirtualization also has some **disadvantages** such as the **requirement of modification of the guest’s OS** and the fact that it isn’t as **flexible as full virtualization**. It also typically only works with specific operating systems and hypervisors.

**Containers**

A container is a **standard unit of software** that **packages up code and all dependencies** so the application runs quickly and reliably from one computing environment to another - Docker

- Containers are **isolated** from each other and bundle their own software, libraries and configuration files.

- They **share the OS kernel** with other containers running as **lightweight processes**.

- Containers are **more portable than VMs** and can run on any OS that supports containers.

- The main purpose of containers is to **distribute and install software in an isolated execution context**, in a much **more efficient and scalable** way compared to the use of virtual machines.

- Containers provide isolated environments, with all resources (file systems, networks, etc) necessary for the execution of a given application or (micro-)service.

In linux, the support for construction of containers is performed through a set of kernel services that allow, in particular, the **creation of different namespaces for various resources** thus delimiting the resources available to processes and the definition of access quotas (**cgroups**) to certain physical resources.

**Main Differences**

**- Hypervisor**: Start-up time -> Slow | Disk space -> Large

**- Containers**: Start-up time -> Fast | Disk space -> Small

The isolation of VMs is achieved by the **virtualization** of the entire hardware, while the isolation of containers is achieved by the **virtualization** of individual processes, increasing portability and efficiency.

**WSL - Windows Subsystem for Linux**

The Windows Subsystem for Linux lets developers run a GNU/Linux environment – including most command-line tools, utilities and applications – directly on Windows, unmodified, without the overhead of a virtual machine. There are 2 versions of WSL:

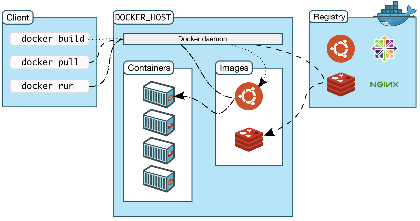
**- WSL 1**: It is a **compatibility layer** for running **Linux binary executables** (ELF format)natively on windows

**- WSL 2**: Includes a **full Linux kernel** with **system call** compatibility for Windows, running on a **lightweight virtual machine** (VM). Runs on top of **Hyper-V** – a **hypervisor** that allows **multiple VMs** to run on a single host machine (Windows).

**Docker**

Is a set of platform as a service (PaaS) products that use **OS-level virtualization to deliver software in packages** called containers.

**-- Architecture**

Docker uses a client-server architecture: the **Docker Client** talks to the **Docker Daemon**, which does the heavy lifting of building, running and distributing your Docker containers.

**- Docker Client** (docker): the primary way that many Docker users interact with Docker.

**- Docker Daemon** (dockerd): listens for Docker API requests and manages Docker objects such as images, containers, networks and volumes.

**- Docker Objects**: images, containers, networks and volumes.

**- Docker Registries**: stores Docker images (Docker Hub, Docker Cloud, etc).

**- Docker Desktop**: a GUI for the Docker Engine that runs on Windows, Mac and Linux.

**-- Images**

An image is a **read-only template** with instructions to create a Docker container.

- Contains everything needed to run an application – all dependencies, configuration, scripts, binaries, etc.

- Provides a filesystem and a set of parameters which can be used to create a container.

- Also contains other configuration such as **environmental variables** and **entrypoint**.

- To create a container, you need an image.

- To create an image, you need a Dockerfile: a text document that contains all the commands a user could call on the command line to assemble an image.

- The docker build command builds an image from a dockerfile and **context**.

**-- Container**

A **container** is a **sandboxed process** on your machine that is **isolated** (filesystem, network, process tree, etc) from all other processes on the machine.

- Is a runnable instance of an **image**.

- You can create, start, stop, delete or move a container using Docker API or CLI.

- Is portable.

- Is isolated from other containers, running its own software, binaries and configurations.

**-- Network**

A **network** is a set of **rules** that define how containers **communicate** between each other and external networks. Docker’s network subsystem is pluggable using drivers:

**- bridge**: the default network driver.

**- host**: removes network isolation between the container and the Docker’s host and uses the host’s network directly.

**- overlay**: for multi-hosting networking.

**- none**: removes all networking from a container.

**-- Volumes**

Volumes are the preferred mechanism for persisting data generated and used by Docker containers.

- Volumes are stored in a part of the host filesystem which is managed by Docker.

- Volumes are **not** tied to the lifecycle of the container.

**-- Layering**

Docker uses a **union file system** (overlayFS) to **layer** images and containers.

- A **union file system** is a file system that operates by **creating layers**, making them **seamlessly stackable**.

- A **Docker image is built up from a series of layers**. Each layer represents an instruction in the image’s Dockerfile. Each layer except the very last one is read-only.

- When you make a change to an image, a new layer is created that applies specifically to that change.

- The **union file system** combines the layers into a single-view (**upper**).

- The **union file system** uses a **copy-on-write** strategy, which means that it only copies a file the first time you modify it – after that, it keeps writing to the same copy.

**Compose**

Compose is a tool for defining and running multi-container Docker applications.

- Compose is a **declarative** way to define and run a multi-container Docker application, using an YAML file to configure the application’s services.

- With Compose, use an YAML file to configure services, networks and volumes.

**Proxying** - (Feito na parte do cliente) Bloco (proxy server) entre um cliente e um outro servidor. O cliente conecta-se ao proxy server que por sua vez faz pedidos ao servidor destino

pelo cliente. Pode ser usado como medida de segurança para esconder a identidade do cliente.

**Reverse Proxying** - (Feito na parte do Servidor) Bloco (reverse-proxy server) que serve como intermediário para pedidos que estejam a ser realizados a um ou mais servidores

backend, o reverse-proxy server recebe os pedidos e distribui estes entre os servidores backend. (Podendo fazer load balancing).

**Load Balancing** - Método para distribuir tráfego por múltiplos servidores para assegurar que nenhum servidor faça um bottleneck. E que todos os servidores são usados

eficientemente.

Como adicionar um ficheiro de configuração nginx para um servidor: Symbolic link em sites enabled, a referenciar o ficheiro de configuração em sites-available.

O que significa o código ser escalável? Quando um código é escalável, significa que ele é capaz de lidar com uma carga crescente de trabalho de maneira eficiente e sem falhar. Com o uso de **named pipe**s o código não é escalável pois os named pipes são unidirecionais e não permitem escrita e leitura ao mesmo tempo.

. 2 vantagens da organização em camadas da imagem Docker, suportadas por overlay type system files:

- **Space optimization**: Layers are stored in a compressed format and are shared between multiple images. This results in efficient storage utilization, as only the changes made in each layer are stored, reducing the overall size of the image.

- **Faster Deployment**: When an image is built in layers, it can be easily cached and reused. This results in faster deployment times, as only the changes made in the latest layer need to be pulled and deployed, rather than the entire image. The overlay type system files support this by allowing multiple layers to be stacked and merged into a single cohesive file system at runtime.

Desvantagem:

- **Complexity**: The process of creating and maintaining images with multiple layers can be complex, especially when multiple people are involved in the development process.

- **Increased size**: Although layers reduce the overall size of an image, they can still increase the size of the final image, especially if many layers are included or if large files are stored in the layers.

- **Debugging**: Debugging a multi-layered image can be challenging, as it can be difficult to identify the source of a problem in a complex layer stack.

- **Immutable**: Once an image layer is committed, it is immutable and cannot be modified. This can limit the ability to make changes to the image, especially in testing and development environments.

--------------

**Dockerfile**

**FROM node:alpine**

**WORKDIR /home/node**

**COPY package.json .**

**RUN npm install COPY app.js .**

**COPY README.md .**

**RUN chown -R node.node /home/node**

**USER node**

**EXPOSE 80**

**CMD ["node", "app,js"]**

**Overall explanation**

**- FROM node:alpine** sets the base image for the Docker image to be built, using the lightweight Alpine Linux distribution with Node.js pre-installed.

**- WORKDIR /home/node** sets the current working directory to **/home/node** within the container.

**- COPY package.json .** copies the **package.json** file from the host machine to the current working directory within the container.

**- RUN npm install** runs the **npm install** command, which installs the dependencies specified in the **package.json** file.

**- COPY app.js .** copies the **app.js** file from the host machine to the current working directory within the container.

**- COPY README.md .** copies the **README.md** file from the host machine to the current working directory within the container.

**- RUN chown -R node.node /home/node** runs the **chown** command, which changes the owner and group of the **/home/node** directory and its contents to **node**.

**- USER node** sets the default user for future commands to **node**.

**- EXPOSE 80** exposes port 80 on the container, making it accessible from the host machine.

**- CMD ["node", "app.js"]** sets the default command to run when the container is started, which is **node app.js**. This command runs the Node.js application.

**Overlays**

- The first overlay layer is created when you copy the package.json file from the host machine to the container. This layer contains the changes made by the COPY command.

- The second overlay layer is created when you run the npm install command. This layer contains the changes made by the installation of the npm packages specified in the package.json file.

- The third overlay layer is created when you copy the app.js and README.md files from the host machine to the container. This layer contains the changes made by the COPY command.

- The fourth overlay layer is created when you run the chown command. This layer contains the changes made by the chown command, which changes the owner and group of the /home/node directory and its contents to node.

These overlay layers are stacked on top of the node:alpine base image and combined at runtime to form the final file system of the Docker container.

**Nota**: WORKDIR e RUN também criam overlay

**Versão modificada (menos overlay)**

**FROM node:alpine**

**WORKDIR /home/node**

**COPY package.json app.js README.md .**

**RUN npm install && chown -R node.node /home/node**

**USER node**

**EXPOSE 80**

**CMD ["node", "app.js"]**

----------

In a linux system, have in consideration a service that attend client processes from one type stream unix socket. Is it possible to attend multiple clients simultaneously?

- Yes, it is possible to attend to multiple clients simultaneously in a Linux system using a Unix stream socket. Unix stream sockets are full-duplex connections that allow for bi-directional communication between the client and server. Multiple clients can connect to the server at the same time and the server can attend to each client in parallel, either by using multiple threads or by using asynchronous I/O.

- However, the exact number of clients that can be attended to simultaneously will depend on various factors such as the server's hardware and network resources, the complexity of the service being provided, and the number of concurrent connections that are allowed by the operating system. In general, Unix stream sockets are well suited for handling a large number of concurrent connections, as they provide a reliable and efficient mechanism for inter-process communication.

How to distinguish each client?

- In a Linux system, when a server is using a Unix stream socket to attend to multiple clients, each client can be distinguished by its unique file descriptor. When a client connects to the server, the server's accept system call returns a new file descriptor that is associated with the client socket. This file descriptor can then be used by the server to read and write data to and from the client.

- Additionally, the server can use the **getsockname** or **getpeername** system call to obtain the socket addresses associated with the client and server sockets. These addresses can then be used to identify the client and distinguish it from other clients that are connected to the server.

- In this way, the server can maintain separate data structures for each client and provide individualized service to each client, while still using a single Unix stream socket to communicate with all clients.

A docker-compose.yml file, for a solution compose with the name tp2, contains three services: svca, svcb, svcc, all on the same network, svcnet, which it bridge type. svca and svcc services have only one instance each, but svcb was launched with scale=4. Executing a shell (/bin/sh) in the svca container, what is the main difference between executing nslookup svcb and nslookup tp2-svcb-1?

- In a Docker Compose environment, each service defined in the **docker-compose.yml** file has a unique hostname, generated from the name of the service and a unique identifier. When a service is launched with multiple instances, as in the case of the **svcb** service with **scale=4**, each instance of the service is assigned a unique hostname that is based on the service name and an incremental index.

- The main difference between executing **nslookup svcb** and **nslookup tp2-svcb-1** in the **svca** container is the hostname that is being looked up.

**- nslookup svcb** will perform a DNS lookup for the hostname **svcb**, which corresponds to the **svcb** service in the **docker-compose.yml** file. This hostname resolves to the virtual IP address that is assigned to the **svcb** service by the Docker Compose network. When the service has multiple instances, the virtual IP address is a load-balancer that distributes incoming traffic to the individual instances of the service.

**- nslookup tp2-svcb-1** will perform a DNS lookup for the hostname **tp2-svcb-1**, which corresponds to a specific instance of the **svcb** service with an index of **1**. This hostname resolves to the IP address assigned to that specific instance of the service by the Docker Compose network.

- In general, using the hostname of the service (e.g. **svcb**) will allow you to connect to any instance of the service, while using the hostname of a specific instance (e.g. **tp2-svcb-1**) will allow you to connect only to that instance.