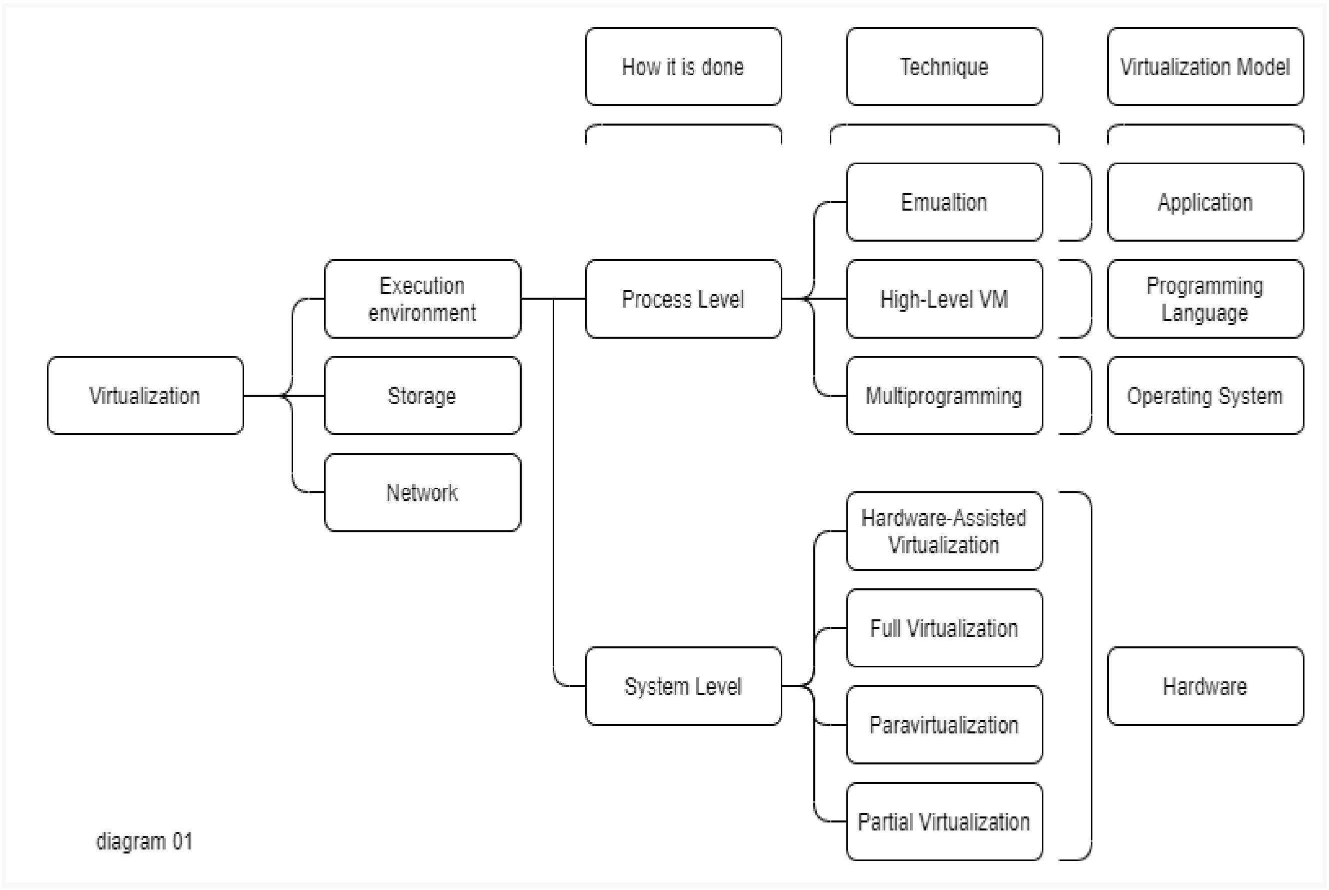
Introduction

The concept of virtualization is generally assumed to have originated in the mainframe era (late 1960s and early 1970s). The method logically divides the systems resources provided by the mainframe between different applications. Today, the term "Virtualization" refers to many technologies and concepts.

The main applications of virtualization are emulating execution environments (either at the process level implemented on top of an existing operating system or at the system level implemented directly on hardware), storage, and networks.

The taxonomy of virtualization



There are five virtualization levels that include the hardware level, Operating system level, Programming language level, Application levels, and the Instruction Set Architecture. which all fall under hardware, software or Networking virtualization type. The implementation, abstraction, as well as examples of technologies for each of these levels will be discussed.

Application level Virtualization

Virtualization at the application level allows applications to run on run-time environments that do not support all of the features that an application might require. It allows the running of applications on an operating system that does not support the application. This type of virtualization focuses primarily on the storage of data (partial file systems, libraries, and emulation of operating system components). It is possible to have unlimited storage without worrying about the user's location, the data is accessed by specifying its logical address or path.

Applications can also be virtualized through application sandboxing, application isolation, and application scalability. Wrapping the application in an isolated layer with the host OS and other applications is part of the process. Consequently, the application can be distributed and removed from user workstations with a lot less hassle. LANDesk application virtualization platforms deploy software applications as self-contained, executable files in an isolated environment without requiring installation, system modifications, or elevated security privileges.

There are several examples of this level of virtualization, including the .NET CLR and the Java Virtual Machine (JVM)

.NET Common Language Runtime(CLR)

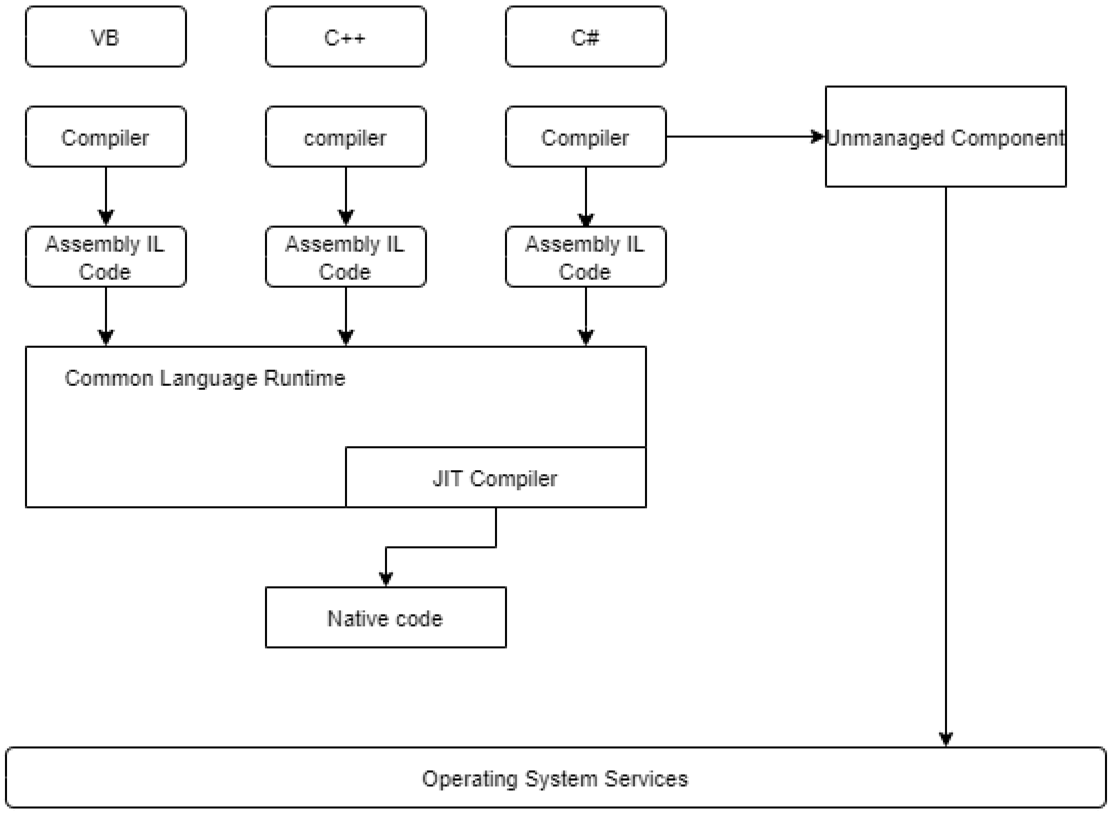
In the .NET Framework, CLR is a basic component and a Virtual Machine. .NET Framework's run-time environment runs code and provides various services to make the development process easier. Basically, it is responsible for managing the execution of *.NET programs* regardless of the language used to write them. Microsoft's CLI(Common Language Infrastructure) implements the Virtual Execution System(VES) which is implemented internally by CLR.
Managed code is the code that runs under the Common Language Runtime. In short, CLR provides a managed execution environment for .NET programs by providing improved security, including cross language integration and a rich set of class libraries. Every .NET framework version includes CLR. The following table illustrates the CLR version in the .NET framework.

What the CLR does during C# program execution

- Using the language-specific compiler, the source code is compiled into Microsoft Intermediate Language (MSIL) which is also called the Common Intermediate Language (CIL) or Intermediate Language (IL) along with it's metadata. The metadata includes the actual implementation of each function of a program.
- This is when CLR came into existence. MSIL code is provided with services and runtime environment by CLR. The CLR includes a JIT (Just-In-Time) compiler that converts MSIL code into machine code that can then be executed by the CPU. .NET Framework class libraries are also used by CLR. CLR makes use of metadata to provide information about a programming language, environment, version, and class library the MSIL code. As

CLR is common so it allows an instance of a class that is written in a different language to call a method of the class which written in another language.

CLR Execution Model



Benefits of CLR:

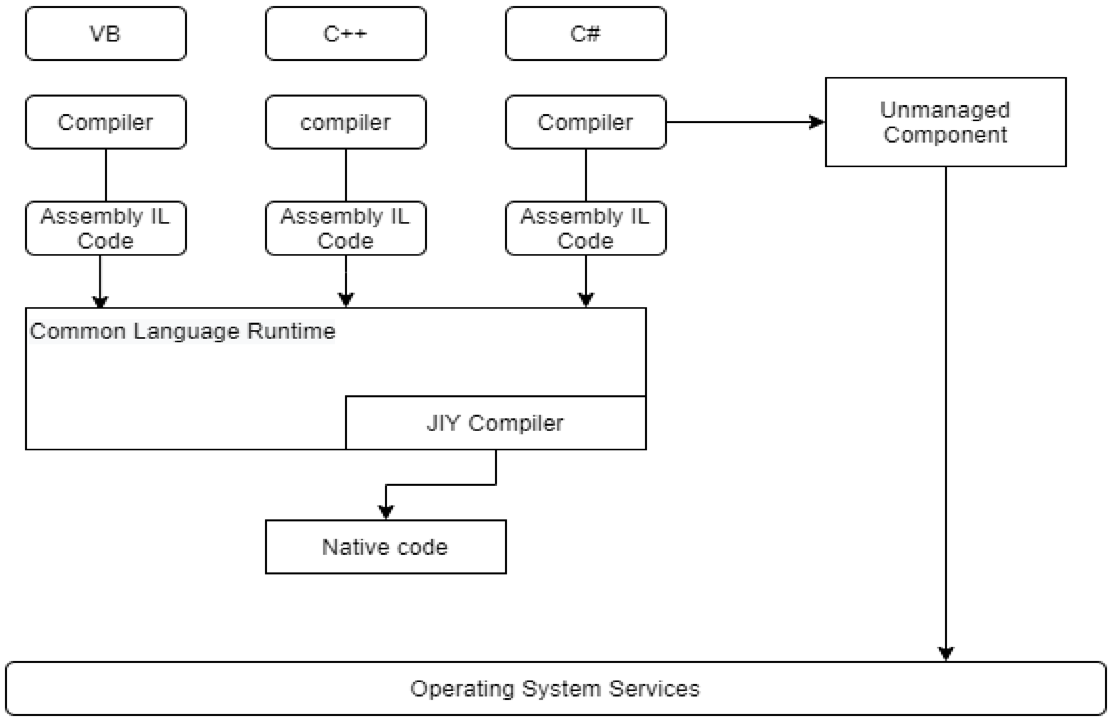
- The security of MSIL instructions is also improved as it analyzes whether they are safe or not. Delegations, rather than function pointers, improve the security and safety of types.
- Garbage Collector supports automatic memory management.
- Developers can easily create multi-threaded and scalable applications since memory management and security issues are automatically handled by the framework.
- Provides a way to use components that are developed by other .NET developers.
- Independent of language, platform, and architecture.
- CTS inside CLR allows the different languages to extend and share their libraries because it provides a common standard.
- Run-time interaction between programs enhances performance.
- Remove the need to recompile a program on any operating system that supports it, enhancing portability.

<u>JAVA Virtual Machine (JVM)</u>

Java Virtual Machine (JVM) is an engine that creates a runtime environment for Java Code or applications. Java bytecode is converted into machine language. Java Runtime Environment (JRE) comprises JVM. Other programming languages are compiled to produce machine code tailored to a particular system. However, the Java compiler produces code for a virtual machine called Java Virtual Machine.

In Java, applications are written once and run everywhere (WORA). A Java programmer can develop Java code on one system and expect it to run on any Java-enabled system without adjusting it.The JVM makes all of this possible.

Java Compilation:



JVM Memory

- The method area is where all of the class level information, including variables and methods, is stored. JVMs have only one method area, which is a shared resource.
- A heap area stores all object information. Per JVM, there is also a heap area. Additionally, it is a shared resource.
- There is one run-time stack created for every thread of the JVM. These blocks are called activation records or stack frames, and they store method calls. The local variables of that method are stored in their corresponding frames. A thread's run-time stack will be destroyed by JVM after it terminates. It is not a shared resource.

- PC Registers: Store the address of the current thread execution instruction. Each thread has its own PC Register.
- There is one native method stack per thread that stores native method information.

The Library level

The vast majority of applications use APIs exported by user-level libraries instead of lengthy OS system calls. A system's API is another candidate for virtualization, since they are most often well-documented. By controlling the communication link between applications and the rest of the system through API hooks, library interfaces can be virtualized. WINE is a software tool that supports Windows applications on top of UNIX hosts. There is also vCUDA, which allows applications running within VMs to take advantage of GPU hardware acceleration.

WINE ("Wine Is Not an Emulator")

Windows applications can be run on several POSIX-compliant operating systems, such as Linux, macOS, and BSD, with the help of Wine. Wine does not emulate Windows logic like a virtual machine or emulator, but instead translates Windows API calls into POSIX calls on-the-fly, eliminating performance and memory penalties associated with other methods and allowing you to integrate Windows applications into your desktop.

For its Windows programs, Wine allows the loading of both Windows DLLs and Unix shared objects. Since there are many functions in the host operating system to be used, NTDLL, KERNEL32, GDI32, and USER32 use the shared object method of implementation within it. Libraries with a higher level of abstraction, such as WineD3D, can use the DLL format. There are many situations in which users can choose not to use Wine's latest DLL, but a DLL from Windows. This can enable features not yet implemented by Wine, but may also result in malfunctions if those features rely on elements not in Wine. Through automated unit testing carried out at each git commit, Wine tracks the state of its implementation

Architecture

Dynamic-link libraries (DLLs) constitute a large part of the Windows programming interface. NTOS kernel-mode program (ntoskrnl.exe) contains a large number of wrapper routines for the system calls of the kernel. Typical Windows programs call some Windows DLLs, which in turn call the user-mode gdi/user32 libraries, which then use kernel32.dll (win32 subsystem) to interact with the kernel. Because the documentation for the system-call layer is not publicly available, and all published interfaces rely on system subsystems running on top of the kernel, the system-call layer is considered private to Microsoft programmers. Additional programming interfaces run individually as separate processes and are implemented as services. RPCs are the principal means of communication between applications and user-mode services.

The Windows application binary interface (ABI) is entirely implemented in user space, rather than as a kernel module. There is a hierarchy within the wine industry, with services provided by In Windows, the kernel is provided by a daemon called wineserver, whose task is to implement basic Windows functionality, integrate with X Window System, and translate signals into native Windows exceptions. Due to Wine's underlying architecture, it is not possible to use native Windows drivers with it, even though it implements some aspects of the Windows kernel. Some games and applications cannot run if virtual device drivers need to be installed. For example, games with StarForce copy-protection cannot run

Security

In light of Wine's ability to run Windows binary code, concerns have been raised over native Windows viruses and malware affecting Unix-like operating systems since Wine can run limited malware made for Windows. Five of thirty malware samples successfully ran through Wine according to a 2018 security analysis, a low number that nonetheless posed a security threat. Because of this, the developers of Wine recommend never running it as the superuser. ZeroWine, for example, runs Wine on Linux in a virtual machine to keep malware completely separated from hosts. Taking advantage of Anbox's Android software, which runs automatically in LXC containers by default, is an alternative method to achieve better security without violating performance.

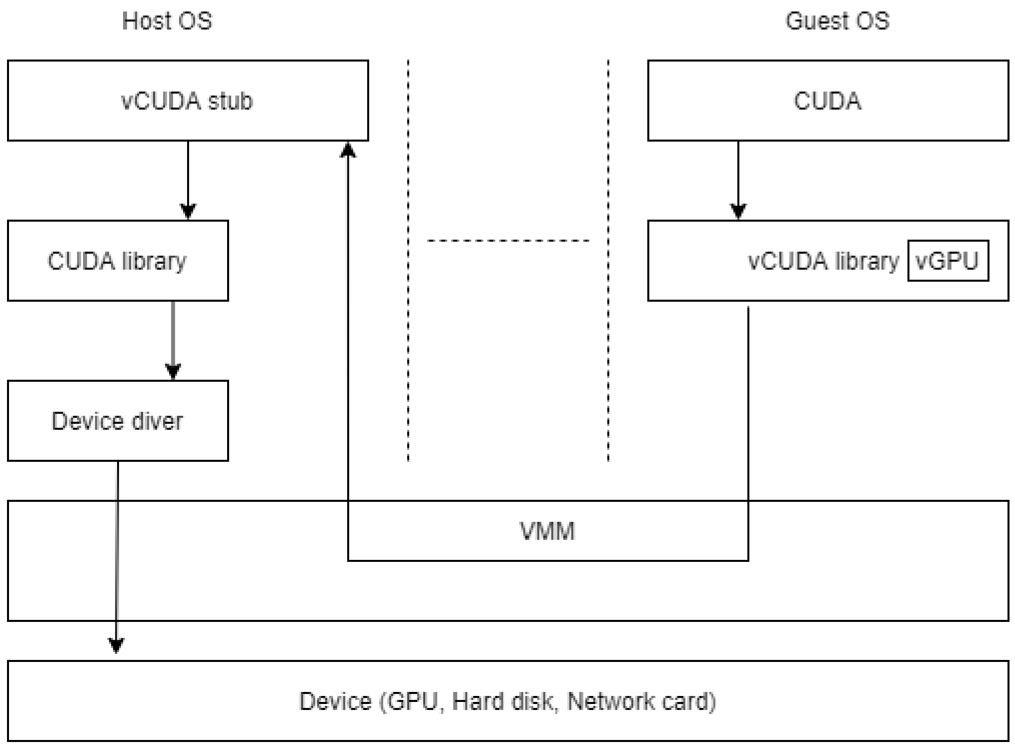
vCUDA

For general-purpose GPUs, CUDA is a programming model and library. It runs compute-intensive applications on host operating systems using the high performance of GPUs. Directly running CUDA applications on a hardware-level virtual machine is difficult, however. The CUDA library can be virtualized with vCUDA and installed on guest OSes. CUDA applications running on a guest OS call the CUDA API, but vCUDA intercepts the call and redirects it to the CUDA API running on the host OS.

Virtualization of CUDA is accomplished using the vCUDA client-server model. vCUDA consists of three user space components: the vCUDA library, the virtual GPU in the guest OS (which acts as a client), and the vCUDA stub in the host OS (which acts as a server). In place of the standard CUDA library, the vCUDA library resides in the guest OS. Stubs are responsible for intercepting API calls, and redirecting them to the client. Additionally, vCUDA manages and creates vGPUs.

A vGPU has three main functions: This abstracts the GPU structure and gives applications a uniform view of the underlying hardware; The vGPU can send a local virtual address to a CUDA application requesting device memory and alert the remote stub to allocate the real memory, and the vGPU stores the CUDA API flow.

vCUDA architecture



<u>Operating system level</u>

It refers to an abstraction layer between traditional operating systems and user applications. OS-level virtualization enables data centers to utilize both hardware and software using isolated virtualized containers. Containers behave like real servers. In virtual hosting environments, OS-level virtualization is commonly used to allocate hardware resources among a large number of mutually distrusting users. To a lesser extent, it can also be used to consolidate server hardware by moving services from separate hosts into containers or virtual machines on one server.

In operating system virtualization, a virtualization layer is inserted inside the operating system to partition a machine's physical resources. VMs can be isolated within a single kernel of an operating system. Containers are often referred to as virtual execution environments (VEs), virtual private systems (VPSs), or simply virtual machines. From the user's perspective, VEs are just like real clients. The result is VEs have their own processes, file system, user accounts, firewall rules, routing tables, and other personal settings. VEs can be customized for different people, but they share the same operating system kernel. Thus, OS-level virtualization is also known as single-OS image virtualization.

<u>Advantages</u>

OS extensions provide two advantages over hardware-level virtualization: (1) Operating system extensions have minimal startup/shutdown costs, low resource consumption, and high scalability; and (2) for an OS-level VM, it is possible for a VM and its host environment to synchronize state changes when necessary. OS-level virtualization can accomplish these benefits via two mechanisms: (1) all VMs on the same physical machine share the same kernel; and (2) the virtualization layer can be designed so that the VMs use as many resources of the host machine as possible, but not modify them.

Disadvantages

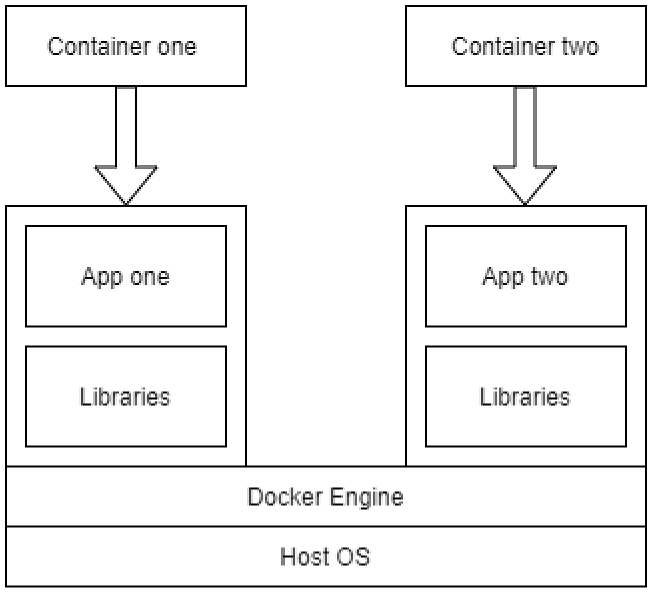
The main disadvantage of OS extensions is that every VM at operating system level in a container must be running the same type of guest OS. Although different OS-level VMs may run different operating systems, they must belong to the same operating system family. Windows XP, for example, cannot run on a Linux-based container. Cloud computing users, however, have different preferences. Some people prefer Windows, while others prefer Linux or another operating system. Thus, there is a challenge for OS-level virtualization in such cases.

The example that will be covered are Docker and Jail

Docker

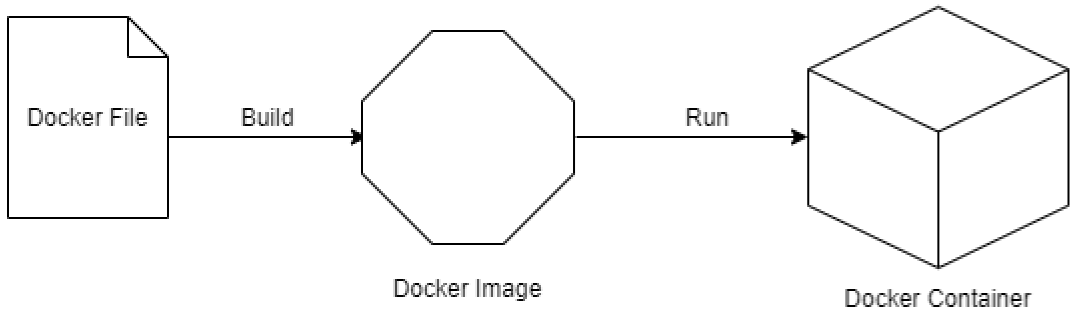
Docker is a platform which packages an application and all its dependencies together in the form of containers. This containerization aspect ensures that the application works in any environment.

Docker's Architecture



In the diagram, each application runs on its own container and has its own set of dependencies and libraries. By making sure that each application is independent of the others, developers can build applications that will not interfere with each other. Thus, a developer can build a container with different applications installed on it and give it to the QA team. The QA team would only need to run the container to replicate the developer's environment.

Dockerfile, Image and Container



Dockerfiles are text documents that contain the commands that a user can run on the command line to assemble an image. Thus, Docker can build images automatically by reading instructions from a Dockerfile. Using docker build, you can automate the execution of several command-line instructions in succession.

Docker Image: In layman's terms, a Docker Image is similar to a template used to create Docker Containers. Therefore, these templates are the building blocks of a Container. Docker run can be used to run the image and create a container.

The Docker Registry stores Docker images. A repository can either be a user's local repository or a public repository like a Docker Hub that allows multiple users to collaborate on an application.

The **Docker Container** is a running instance of a Docker Image that contains all the packages needed to run the application. In essence, these are ready applications created from Docker Images, which are Docker's ultimate benefit.

JAIL

With jail, you can create various virtual machines, each with its own set of utilities installed and configuration. This makes it a safe way to test out software. It is possible, for example, to run different versions or configurations of a web server package in different jails. Because the jail is confined to a very narrow scope, any misconfiguration or mistake (even if done by the in-jail superuser) will not affect the rest of the system. As nothing has actually been changed outside of the jail, "changes" can be discarded by deleting the jail's copy of the directory tree.

However, the FreeBSD jail does not achieve true virtualization; the virtual machines cannot run different kernel versions than the base system. Virtual servers share the same kernel, exposing the same bugs and security holes. There is no support for clustering or process migration, so the host kernel and the host computer remain a single point of failure for all virtual servers. Jails can be used to safely test new software, but not new kernels.

Security

Due to the separation between the jailed environment and the rest of the system (the other jails and the base system), FreeBSD jails can enhance the security of a server.

Limitations

Processes in a jail cannot communicate with processes in a different jail or on the main host. The ps command, for exa

mple, only displays the processes running inside the jail.

Direct access to the running kernel and loading of modules are prohibited. Most sysctls and the securelevel cannot be modified.

It is prohibited to modify the network configuration, including interfaces, IP addresses, and routing tables. Accessing divert and routing sockets is also prohibited. Furthermore, raw sockets are disabled by default. Jails are bound to specific IP addresses and firewall rules cannot be modified. With the introduction of VNET(virtual network stack), jails can modify their Network Configuration (including interfaces, IP addresses, etc.) as long as the jail has the vnet enabled.

It is forbidden to mount and unmount filesystems. The jail cannot access files above its root directory (i.e. the jail is chrooted).

Device nodes cannot be created by jailed processes.

Hardware Abstraction Layer (HAL)

The hardware virtualization occurs directly on top of the bare hardware. By using this technique, a virtual machine is generated in a virtual hardware environment. Virtualization, on the other hand, manages the underlying hardware. A computer's resources, such as its processor, memory, and I/O devices, can be virtualized. Several users will concurrently upgrade the hardware utilization rate. IBM implemented the idea in the 1960s with the VM/370. More recently, the Xen hypervisor has been used to virtualize x86-based machines to run Linux and other guest operating systems.

VMware