Distributed Systems: A Modern Approach

Prashant Shenoy

University of Massachusetts Amherst

4. Virtualization and Virtual Machines

Virtualization is a key technology that is used in data centers employed by cloud computing platforms and enterprises. Virtualization comes in many flavors and its use in computer and distributed systems goes well beyond cloud computing. In this chapter, we first discuss the broad concept of Virtualization and its many uses in system design. We then delve into virtualization technologies that are specific to data centers and cloud computing.

What is virtualization?

Intuitively, virtualization is a layering abstraction provided by a computer system that makes the system easier to use and reduces complexity for applications and users. To better understand the concept of virtualization, let us consider several examples that may be familiar to the reader.

Let us consider operating systems, which make extensive use of virtualization by providing a set of logical abstractions on the underlying hardware resources such as CPU, memory and storage. These OS abstractions make it easier for a user to use the machine without having to understand or deal with low-level details of the hardware. In case of the CPU, the OS provides the abstraction of processes and threads. Each process is provided an illusion that is the only entity executing on the processor and the OS multiplexes various processes on the CPU using scheduling techniques. A user can start any number of applications on their computer (subject to hardware resources limits) and all of these applications execute concurrently by sharing hardware resources. In case of memory, the OS provides abstractions of addresses spaces and virtual memory. A process can use its memory address space transparently and without regard to the presence of other processes resident in RAM. The OS uses virtual memory to give an illusion of infinite RAM by using a backing disk store as swap space and through the use of on-demand paging. Finally, in case of disk, the OS provide the abstraction of a file and folder that can be used by users to store their data without having to deal with low-level details of where to store file data on disk. These are common features that should be familiar to any computer user but the user may not have realized they are dealing with virtualization! In these examples, the OS virtualizes a hardware resources in order to share that resource across multiple processes or to provide logical abstractions that are easier to use for the user. This type of virtualization is also referred to as resource virtualization.

Next consider the Java programming language. Java programs needs a java virtual machine (JVM) to execute them. The JVM provides an abstraction of a logical machine that is used to execute the java program. The advantage of this approach is portability, since programs run inside the JVM and the details of the underlying hardware platform and the CPU architecture are hidden from it. Hence, the same java program will run on any hardware and OS platform so long as a JVM is available for it. %% In contrast, a C++ binary that is complied for the x86 architecture will not run on Arm machines without being recompiled. This is an example of application-level or language-level virtualization.

Finally consider a infrastructure-as-a-service cloud platform. The cloud platform provides virtual servers to its customer. Each virtual server abstraction is identical to a physical server. The OS and applications that run on the virtual server are not aware that they are running on a virtual host, rather than a physical one. Further, each physical server can run multiple virtual servers, with physical resources being partitioned across resident virtual servers. The virtualization layer multiplexes the various virtual servers onto the physical server. This is an example of hardware virtualization.

These examples illustrate the virtualization is a broad concept that is used by operating systems, programming languages and distributed systems for many different purposes. We now provide two definitions for virtualization.

Virtualization is a layering abstraction that extends or replaces an existing interface in order to mimic the behavior of another system [Tannenbaum].

Figure XX depicts this concept where the virtualization layer uses interface A on system A to mimic interface B of another system B. In case of resource virtualization, the virtualization layer is the operating system, which uses the hardware interfaces of the physical machine (interface A) to expose abstractions such as processes, address spaces, files (interface B). In case of Java, the virtualization layer is the JVM, which uses the OS interface and libraries (interface A) to provide the logical machine (java virtual machine or interface B) that runs java programs. Finally, in case of cloud servers, the virtualization layer is called the hypervisor, which we will discuss in Sec XXX. The hypervisor uses the hardware interface of the physical server (interface A) to expose virtual servers that resemble physical servers (interface B).

An equivalent and more formal definition is provided by [Kaashok] and elaborated further in [Neih] which states:

Virtualization is a technique that uses the layering principle through enforced modularity to implement a virtual resource using the underlying physical resource.

The layering princiрe refers to the layered approach for software design that uses resources and interfaces below that layer to create an abstraction exposed to applications above that layer. This, in effect, adds a layer of indirection where applications have to use the interface of the abstraction to access the resources below that later. Enforced modularity means that applications using the layer can not bypass it in order to access the resources underneath the layer. The resources below the layer are also not directly visible to the applications using the virtualization layer.

Both of these definitions define the concept of virtualization broadly to capture its use in operating systems, programming languages, and distributed systems. While they capture virtualization technologies used in they cloud context, they are not specific to those technologies alone. We now discuss three general principles for implementing virtualization.

Virtualization principles

There are three high level approaches for implementing virtualization in a computing system [Kashaook, Neih]: multiplexing, aggregation, and emulation. These approaches are depicted in Figure XX

Multiplexing. Multiplexing is used when the virtualization layer exposes multiple copies of a virtual resource, all of which are mapped onto the same physical resource. In this case, the virtual resources are mutiplexed onto physical resource. As an example of multiplexing, consider multiple logical network inferences that are mapped onto single physical network interface card (e.g., a physical ethernet card). The interfaces are temporally multiplexed onto the physical interface, which enables them to use a certain fraction of the physical network bandwidth. As another example, consider the OS memory manager that allocates physical memory pages to various processes to implement the abstraction of address spaces. This is an example of spatial multiplexing where address spaces belonging to processes share different pages of physical memory.

Aggregation. The aggregation method is used when the virtual resource is constructed by aggregating multiple physical resources. For example, RAID array provide the abstraction of single larger logical disk that are constructed using multiple physical disks. The logical disk exposed by the RAID array aggregates the storage space of the underlying physical disks and the presence of physical disks is hidden from users in line with notion of enforced modularity.

Intuitively, aggregation and multiplexing are opposites of one another. In aggregation, multiple physical resources are used to construct a single virtual resource, whereas in multiplexing, multiple virtual resources are constructed using a single physical resource.

Emulation. In emulation, the virtual resource mimics ("emulates") a physical resource that is different from the actual physical resource used to implement it. A full machine simulator used by computer architecture researchers implements a new machine architecture (e.g., a new CPU instruction set) using the underlying machine resources. The Bochs open source simulator emulates an x86 machine on many non-Intel machines. Apple's Rosetta2 technology emulates an x86 architecture on its ARM-based Apple Silicon processors to enable Intel-based MacOS applications to seamlessly run on its newer Arm-based processors. Virtual memory is implemented by an OS by using disk to emulate a slower, but larger, main memory.

Virtual Machines and Taxonomy of virtual machines

A virtual machine is a software implementation of a physical machine and its computing environment. From an application's standpoint, the virtual machine resembles a physical machine in all respects and an application can execute on a virtual machine as if it were running on a physical machine. Since virtual machines come in many flavors, we provide a broad definition of virtual machines from [Nieh]

A virtual machine is a full-fledged computing environment with its own isolated processing, memory, and networking capabilities.

A virtual machine is capable of running one or more application processes in its computing environment. While some virtual machine (e.g., java virtual machine) run a single application processes, others can run multiple concurrent processes and may even run their own operating system that is independent of the OS running on the physical machine. Regardless of its capabilities, all virtual machines hide the underlying physical machine from the applications and the OS that run inside them. The processes and OS see a virtual machine that resembles a physical machine and execute normally like they were running on actual physical machine.

We now provide two taxonomies to understand different types of virtual machines. The first taxonomy [Smith and Nair] is based on the interface on the host machine (interface A in Fig X) that is used to implement the implement the exposed interface B. Figure XX shows three different interfaces.

- CPU instruction set. The machine instructions exposed by the CPU represents the interface between the hardware and software and is the lowest level interface that can be used by the virtualization layer. The instruction set supported by a CPU can be grouped into two types. The first group consists of all machine instructions that can be executed by any user-level process. The second group consists of privileged instructions that can only be executed by the operating systems. The virtualization layer can use one or both group of machine instructions to implement a new CPU and its corresponding instruction set. For example, a virtual machine with an ARM CPU can be implemented using the x86 instruction set. Since the virtualization layer uses hardware interfaces, this type of virtualization is also referred to as hardware-level virtualization.

- System call interface: In this case, the system call interface exposed by the OS (interface A) is used to implement the virtualized interface B. This type of virtualization can be viewed as OS-level virtualization. Container technologies, such as docker, that uses OS interfaces to implement virtual machines are an example of OS-level virtualization. Some operating systems, such as Oracle's Solaris, have the ability to emulate the system call interface of an older version of the OS (interface B) using the current interface, which allows the system to be backward compatible to applications that depended on an older OS version. This functionality is referred to as legacy containers by Solaris. Finally, Wine is an open source software tool to run Windows win32 and win64 applications on POSIX-supported operating systems such as Linux, BSD, and macOS. It does do by implementing the Window's win32 and win64 interface using POSIX system and library calls.
- Application-level libraries. In this case, the virtualization layer using application-level libraries and application programming interfaces (APIs) to implement the virtual interface B. This type of virtualization can be viewed as application-level virtualization. Application-level libraries, such as the C and C++ libraries, are higher level interfaces that are themselves implemented using the underlying system call interface, which they hide from the application developer. Application-level virtualization uses these libraries and their APIs to implement the virtual interface exposed by the virtual machine. The Java virtual machine is an example of this type of virtualization since the JVM is often implemented in C or C++ and exposes the abstract java virtual machine interface to run java programs.

Our second taxonomy [Nieh] focuses on the functionality of the virtual machine itself rather than the interfaces used to implement it. When viewed from this perspective, virtual machines can be classified into three types.

- Language-based virtual machines. Language-based VMs provide the run-time environment to run applications written in a specific programming language. Java virtual machine, javascript engines in web browsers, Microsoft's common language runtime, and the Python run-time environment are all examples of language-based VMs. Language-based VMs run a single application process, and its threads, in the VM and are also referred to a process virtual machines.
- Lightweight virtual machines. Lightweight VMs use operating system and hardware isolation mechanisms to provide a sandboxed environment that can run one or more processes. Lightweight VMs typically present the same OS interface as the underlying OS to applications and are used for isolation and resource allocation. That is, they are used to isolate a process or groups of processes from other applications running on the physical host and to allocate a specified amount of the

host's hardware and software resources to applications. Each physical host can run multiple lightweight virtual machines, each of which is isolated from the rest and provides an isolated environment to its processes. Examples of lightweight virtual machines include Docker-based linux containers, Solaris Resource Containers, and BSD jails. Research prototypes such as the Denali isolation kernel are also an example of lightweight VMs.

- System-level virtual machines. System-level VMs are computing environments that mimic an entire physical machine abstraction, including all its hardware such as the CPU, memory, disk and network interface. Since system virtual machine expose a bare metal machine abstraction, they can run an operating system and applications on top of them. Like lightweight virtual machines, a system-level virtual machine provides isolation from other virtual machines or applications running on the physical host. Multiple system-level VMs can run on a single host and the virtualization layer multiplexes them onto the physical host and isolates them from one another.

There are two broad categories of platforms that implement system-level virtual machines. The first type of platform is called a hypervisor, which typically exposes virtual versions of the same hardware as the underlying physical machine. Since the exposed hardware (e.g., the CPU) is the same as the physical hardware, the hypervisor uses direct execution to execute its virtual machines. Direct execution means that the machine instructions of application processes and the operating systems running inside the VM are directly executed on the physical processor. While normal instructions can be directly executed and provides highly efficient execution, privileged instructions can not be executed directly for security reasons and are instead implemented using a trap and emulate framework, as discussed in Sec XXX.

Full emulation is the other method for implementing system-level virtual machine. In full emulation, the emulation layer can expose a different set of hardware than the physical hardware (e.g., a virtual CPU with a different instruction set from the underlying physical CPU),. In this case, the goal is to accurately mimic the emulated machine and requires each virtual instruction to be implemented using an equivalent set of physical instructions. As a result, full emulation can be orders of magnitude slower than using hypervisors, but it allows any machine architectures to be implemented using the emulation layer. Full emulation, also knows as full machine simulation, is popular in computer architecture research, where researchers an experiment with new hardware designs or new architectural features. The Bochs open source tool is another example of full emulation, where a full x86 PC is emulated using native hardware.

We note that some emulation techniques are not based on system-level virtual machines. For example, Apple Rosetta and Rossetta 2 are processors emulation techniques that perform dynamic binary translation of one instruction set to another and enable MacOS applications designed for an older Mac hardware platform to run on a newer plarform. Rosetta allows PowerPC Mac applications to run unmodified on Intel-based Macs, while Rosetta 2 enables Intel-based Mac applications to run umodified on Arm-based Macs (e.g., on the M1 arm processor). Both emulators should be viewed as application-level emulation frameworks—neither exposes a full machine to applications and each is a process virtual machine.

We can derive a rough mapping between the two taxonomies. Language-level virtual machines are implemented using application layer virtualization. Lightweight virtual machines are implemented using OS-layer virtualization, although some lightweight

VMs may also reply on isolation features at the hardware layer. Within system-level virtual machines, hypervisors are implemented using hardware-layer virtualization, while full emulation frameworks are user-level programs that implement virtualization using application-layer libraries and APIs.

In the rest of this chapter, we focus on hypervisors and containers since these virtualization technologies are predominantly used in cloud and data centers.

Hypervisors

A hypervisor is a type of system software that runs and manages virtual machines. It provides the virtualization functionality that exposes virtual machines to users and executes them by multiplexing VMs onto the resources of the physical machine.

Hypervisors are also referred to as virtual machine monitors (VMMs) and these terms are used interchangeably in the literature. However some researchers prefer to use VMMs to refer to the subsystem of the hypervisor responsible for virtualizing processors and memory, and use hypervisors to refer to the full system for providing VM functionality [Nieh]. In this book, we will not distinguish between the terms and assume they are synonymous. The concept of hypervisors was introduced in a seminal paper by [Popeck and Goldberg] where they specified three important properties of hypervisors. - Efficiency: A hypervisor should be capable of executing a large fraction of CPU instructions directly on hardware with no intervention. This property ensures that process running inside virtual machines see only a small performance slowdown. It also distinguishes hypervisors from full system emulators, which need to translate every instruction and see a significant performance impact. - Safety: This property, also known as the resource control property, states that the hypervisor should remain in full control of the virtualized resources. Hypervisors should allow arbitrary operating systems and processes to execute inside virtual machines. Safety, which is enforced through isolation, should ensure that it is not possible for a malicious process from impacting another VM or the hypervisor itself. - Equivalence: A process executing under the VMM should execute a behavior that is indistinguishable from when it runs directly on the physical machine. The only exceptions to this property are that the virtual machine may be slower than the physical machine it mimics and may also have fewer resources (e.g., less memory) than the underlying physical machine.

Hypervisors can be classified into two basic types. In this thesis, Goldberg [Goldberg thesis] named them as Type 1 and Type 2 hypervisors, and these names, while not very descriptive, are used to this day.

Type 1: Type 1 hypervisors run on bare machine, which means they do not require a separate operating system to run and take on the role of an operating system that is designed specifically to run virtual machines. Since they run on a bare machine, type 1 hypervisors are responsible for managing virtual machines as well for performing the OS task of managing and allocating resources to each VM.

Each virtual machine running on a type 1 hypervisor runs its own operating system, referred to as a guest operating system. Since each VM is independent and isolated machine, different VMs can run different operating systems on the same physical machine.

Type 1 hypervisors rely on direct execution, where virtual machine instructions are executed directly on the CPU. As we will see shortly, direct execution only applies to CPU instructions that are unprivilged, while privileged instructions needed to be handled by the hypervisors. As we will see shortly, type 1 hypervisors require hardware support from the underlying CPU and not all CPUs are capable to running Type 1 hypervisors. Such type

1 hypervisors are also referred to as hardware virtualization, and can run any unmoidified operating systems and applications that are capable of running on the physical machine.

Since many CPUs, including early Intel x86 processors, lacked support for running Type 1 hypervisors, researchers designed a new virtualization technique called paravirtualization. Paravirtualized type 1 hypervisors no longer require hardware support from the underlying CPU and can run on any CPU. To do so, they require the operating system kernel to be modified to support virtualization, and as a result, are no longer able to run an arbitrary unmodified OS. Paravirtualization represents a tradeoff between the ability to run hypervisors on any processor and the ability of run an unmodified OS. The notion of paravirtualization was introduced in the Denali kernel, a research operating sysyem designed to run a large number of VMs without any hardware support from the CPU. Since then, the approach has been employed by many other hypervisors.

Examples of type 1 hypervisors include VMWare ESX server, Windows Hyper-V, and Xen. Of these ESX and Hyper-V use hardware virtualization, while Xen uses paravirtualization.

Type 2: A type 2 hypervisor runs on top on an operating system, which is called the host operating system (host OS). From the perspective of the host OS, a type 2 hypervisor is simply a user process (i.e., an application) that runs alongside other active processes. The hypervisor provides an environment to run virtual machines, much like a type 1 hypervisors. However type 2 hypervisors only need to manage virtual machines and they leave the task of allocating and managing resources to the host OS (unlike type 1 hypervisors, which perform both of these functions).

Like before, each VM runs a guest operating system and arbitrary applications on top of the guest OS. Type 2 hypervisors do not require any special hardware support, since they rely on the host OS to perform all privileged operations on behalf of the VMs. Hence, they can run on any processor and can run arbitrary unmodified operating systems and applications.

Examples of type 2 hypervisors include VMware workstation, VMware Fusion, Oracle VirtualBox, Microsoft VirtualPC, and Linux KVM. This type of virtualization is also referred to as full or software virtualization.

Next we discuss each type of hypervisor in detail.

Type 1 hypervisors: Hardware Virtualization

Goldberg theorem for type 1

Type 2 Hypervisor: Software Virtualization

Paravirtualization and Xen

virtualizing memory, disk, network