Lecture 22: November 27

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

22.0.1 Server State and Replication

When designing server software for a distributed system of any sort, the programmer must decide whether the server should be stateful or stateless. A **stateful** server is one which maintains information about the clients that are connected with it. This provides better performance because clients do not need to repeatedly inform the server of their identity or their past requests, reducing overhead. However, a stateful server can be less resilient to failures—if the server crashes it loses all of its state leading to errors or forcing clients to somehow send information to rebuild the state. A **stateless** server is one that does not maintain any state at all about client connections. In this case, every request made by a client must provide sufficient information for the server to authenticate the client and return a proper result. Stateless servers can be more fault tolerant because if a server crashes it can simply be rebooted—since it does not have any state to lose it will have no impact on the system other than a delay while the machine or process is restarted.

The distributed systems we have described can involve a large number of clients accessing a server. As a result, the server can easily become a bottleneck, reducing performance. **Replication** involves running multiple servers that each contain the data or services provided by the distributed system. This can improve both performance and fault tolerance since client requests can be distributed across multiple servers, and if one server fails the others can continue working. However, replicating servers adds difficulties similar to the cache consistency ones listed previously, since now the distributed system must ensure that all servers maintain consistent data.

22.1 Sun's Network File System

Sun's Network File System, or **NFS**, is an example of a distributed file system. NFS has been around for more than twenty years and is the standard for UNIX-based distributed file access. It was originally designed to be used over LANs (using UDP) but more recent versions are also used over WANs (by using TCP). NFS had stateless servers until the most recent version (NFSv4). Now that NFS servers are stateful they can do things like locking files.

NFS is implemented by a set of RPC operations for file accesses. These RPC include: directory search, accessing file attributes, reading/writing files, etc. Nodes do not need to have a common operating system; all nodes just have to support the defined protocol using RPCs. RPCs were originally created for the first version of NFS.

22.2 Data Centers

A data center is a large collection of servers and storage devices, typically used by enterprises to run large server applications. Internet companies such as Google and Facebook have data centers with thousands of servers. These data centers require vast amounts of electricity (for example a single location can use more

energy than a small city). Electricity is used not only to power the servers but also a cooling infrastructure. Backup generators (typically diesel) must also be available in the event of a power failure. Data centers have been traditionally built in large warehouses but a more modern approach is to use shipping containers. A modular data center uses shipping containers that can be "plugged in" to expand capacity. Each container is filled with thousands of servers. In a traditional data center, each physical server typically runs a single application, e.g. a web or database server. In modern data centers, virtualization is increasingly being used to provide a way to cleanly subdivide a single physical server into multiple virtual machines. This allows for greater utilization of server resources without worrying that one application crashing will impact others.

One of the benefits of virtualization is that the resources (CPU, memory, and network bandwidth) allocated to a virtual machine can be adjusted dynamically. This means that a virtual data center can more efficiently allocate resources and respond to changing resource demands, such as when a website may have different traffic loads at different times of the day. **Consolidation** is also easier with virtualization. Consolidation is the process of removing some number of old servers and replacing them with a smaller number of newer/faster processes (or even just one new server).

22.3 Virtualization

Virtualization was developed in the 1970s as a way to run legacy software on newer mainframe hardware. The new systems being developed did not have an identical architecture to older ones, so they could not run applications from older systems without modification. Virtualization tried to solve this problem by creating an interface within the system that would mimic the behavior of the legacy system being reproduced. Virtualization can also be used to provide isolated containers within which to run an application, or even a full "virtual machine" composed of an operating system and all of its applications. Running these within a virtual environment means that applications can be kept completely isolated from one another, leading to more flexible control over how resources are allocated and better reliability in the face of crashes.

Virtualization can be performed in different ways. **Emulation** attempts to completely simulate the hardware which is being mimicked. This means that completely unmodified applications and operating systems can be run within the emulator because they are completely "tricked" into thinking they are running on the original hardware. However, emulators typically incur a high overhead and thus may only be able to run the software at reduced performance. Examples: Bochs, QEMU, "video game" emulators.

Full or Native Virtualization is a step below emulation in that it may not completely emulate the original hardware. A **virtual machine** is the software being run within the virtual environment, and it is essentially a full virtual computer composed of an operating system and software packages. In general, the virtual machine is completely unaware that it is running within a virtualization environment, and thinks that it has complete access to the system's hardware. In practice, the virtualization environment must mediate access between the virtual machine (or machines) and the real hardware. Example: VMware ESX

Para-virtualization is a system which requires a small number of modifications to be made to the operating system inside each virtual machine in order to be run. By making the operating system aware that it is within a virtual environment, it can be possible to improve the performance of the system. Example: Xen.

OS-level Virtualization is a simpler technique which allows multiple copies of a single operating system to run simultaneously. This provides some protection between each instance of the OS, but it is typically not as strong as the boundary between virtual machines in para or full virtualization. Examples: BSD jails, Linux Vserver.

Application Level Virtualization provides a virtual environment for only a single application within an operating system. This is the technique used to run java applications—the JVM gives each java application its own set of resources and prevents conflicts between them.

22.4 Hypervisors

A hypervisor is the piece of software which provides the virtualization abstraction. In many cases, the hypervisor is very similar to an operating system—it must manage how resources are allocated to each virtual machine and it must provide protection and security between them; each VM can be thought of as an application. A **Type 1** hypervisor is one that runs on "bare metal". This means that the hypervisor itself is the lowest level operating system, and that it runs directly over the hardware. In contrast, a **Type 2** hypervisor runs within a host OS. This means that you could, for example, run Windows on your computer and a hypervisor within it. Then, within the hypervisor you could start up a virtual machine, perhaps running Linux. In both cases, the hypervisor pretends to be giving each virtual machine exclusive access to the hardware. In the Type 1 case, the hypervisor does have direct control over the hardware, but for Type 2, the hypervisor must request access to hardware devices from the host OS on behalf of its virtual machines.

The primary challenge when building a virtualization system is to be able to deal with special **privileged instructions**. These are the assembly instructions that are only allowed to be run within kernel mode in order to ensure the proper protection within the system. In order to run a virtual machine, the hypervisor needs a way of detecting when privileged instructions are about to be run, and find a way to modify how they are run.

In a Type 1 hypervisor, the virtual machine is always running in "user mode", even though its operating system may think it is in kernel mode. Instead, when the VM makes a call to a privileged instruction (one that truly requires being in kernel mode), the hypervisor intercepts this using an instruction trap. They hypervisor then takes over to run the necessary instructions for the VM.

In a Type 2 hypervisor, it is not possible to have the hypervisor run in kernel mode because it is simply an ordinary process within the host OS. Instead, the hypervisor must scan through all of the code that a VM is about to run. When it detects that the VM will run a privileged instruction, the hypervisor uses binary translation to replace the call with one that goes through the hypervisor. The hypervisor then makes the call into the host OS using ordinary system calls.