Part Eight

Advanced Topics

Virtualization permeates all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware. This environment behaves toward them as native hardware would but also protects, manages, and limits them.

A *distributed system* is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through a local-area or wide-area computer network. Computer networks allow disparate computing devices to communicate by adopting standard communication protocols. Distributed systems offer several benefits: they give users access to more of the resources maintained by the system, boost computation speed, and improve data availability and reliability.

CHAPTER 18

Virtual Machines

The term ***virtualization*** has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

CHAPTER OBJECTIVES

- Explore the history and benefits of virtual machines.
- Discuss the various virtual machine technologies.
- Describe the methods used to implement virtualization.
- Identify the most common hardware features that support virtualization and explain how they are used by operating-system modules.
- Discuss current virtualization research areas.

18.1 Overview

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. This concept may seem similar to the layered approach of operating system implementation (see Section 2.8.2), and in some ways it is. In the case of

virtualization, there is a layer that creates a virtual system on which operating systems or applications can run.

Virtual machine implementations involve several components. At the base is the **host**, the underlying hardware system that runs the virtual machines. The **virtual machine manager** (**VMM**) (also known as a **hypervisor**) creates and runs virtual machines by providing an interface that is ***identical*** to the host (except in the case of paravirtualization, discussed later). Each **guest** process is provided with a virtual copy of the host (Figure 18.1). Usually, the guest process is in fact an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its own virtual machine.

Take a moment to note that with virtualization, the definition of "operating system" once again blurs. For example, consider VMM software such as VMware ESX. This virtualization software is installed on the hardware, runs when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are, in fact, guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly it acts like an operating system. For clarity, however, we call the component that provides virtual environments a VMM.

The implementation of VMMs varies greatly. Options include the following:

- Hardware-based solutions that provide support for virtual machine creation and management via firmware. These VMMs, which are commonly found in mainframe and large to midsized servers, are generally known as **type 0 hypervisors**. IBM LPARs and Oracle LDOMs are examples.

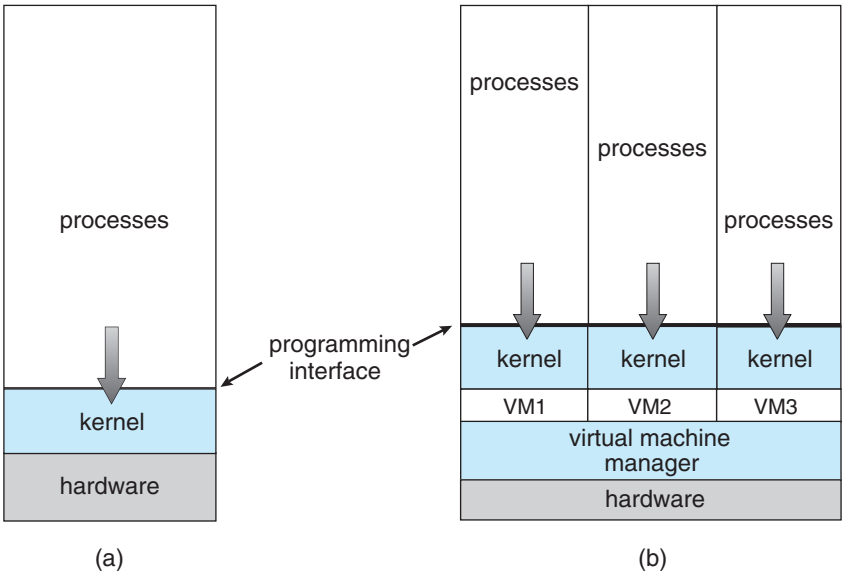

Figure 18.1 System models. (a) Nonvirtual machine. (b) Virtual machine.

INDIRECTION

"All problems in computer science can be solved by another level of indirection"—David Wheeler
". . . except for the problem of too many layers of indirection."—Kevlin Henney

- Operating-system-like software built to provide virtualization, including VMware ESX (mentioned above), Joyent SmartOS, and Citrix XenServer. These VMMs are known as **type 1 hypervisors**.
- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server with HyperV and Red Hat Linux with the KVM feature. Because such systems have a feature set similar to type 1 hypervisors, they are also known as type 1.
- Applications that run on standard operating systems but provide VMM features to guest operating systems. These applications, which include VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox, are **type 2 hypervisors**.
- **Paravirtualization**, a technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance.
- **Programming-environment virtualization**, in which VMMs do not virtualize real hardware but instead create an optimized virtual system. This technique is used by Oracle Java and Microsoft.Net.
- **Emulators** that allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU.
- **Application containment**, which is not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system. Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs "contain" applications, making them more secure and manageable.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

18.2 History

Virtual machines first appeared commercially on IBM mainframes in 1972. Virtualization was provided by the IBM VM operating system. This system has evolved and is still available. In addition, many of its original concepts are found in other systems, making it worth exploring.

IBM VM/370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine. The solution was to provide virtual disks—termed **minidisks** in IBM's VM operating system. The minidisks were identical to the system's hard disks in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once the virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

For many years after IBM introduced this technology, virtualization remained in its domain. Most systems could not support virtualization. However, a formal definition of virtualization helped to establish system requirements and a target for functionality. The virtualization requirements called for:

- **Fidelity**. A VMM provides an environment for programs that is essentially identical to the original machine.
- **Performance**. Programs running within that environment show only minor performance decreases.
- **Safety**. The VMM is in complete control of system resources.

These requirements still guide virtualization efforts today.

By the late 1990s, Intel 80x86 CPUs had become common, fast, and rich in features. Accordingly, developers launched multiple efforts to implement virtualization on that platform. Both **Xen** and **VMware** created technologies, still used today, to allow guest operating systems to run on the 80x86. Since that time, virtualization has expanded to include all common CPUs, many commercial and open-source tools, and many operating systems. For example, the open-source ***VirtualBox*** project (http://www.virtualbox.org) provides a program that runs on Intel x86 and AMD 64 CPUs and on Windows, Linux, macOS, and Solaris host operating systems. Possible guest operating systems include many versions of Windows, Linux, Solaris, and BSD, including even MS-DOS and IBM OS/2.

18.3 Benefits and Features

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because

each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.

A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to providing sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to freeze, or **suspend**, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and **snapshots** to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then **resume** where it was, as if on its original machine, creating a **clone**. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For example, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Of course, the operating system runs on and controls the entire machine, so the system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is **templating**, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a **live migration** feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and without interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The "Open Virtual Machine Format" is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. **Cloud computing**, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, that others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in

remote data centers and access their applications as if they were local. This practice can increase security, because no data are stored on local disks at the user's site. The cost of the user's computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as its runs remotely (via a protocol such as RDP). Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

18.4 Building Blocks

Although the virtual machine concept is useful, it is difficult to implement. Much work is required to provide an ***exact*** duplicate of the underlying machine. This is especially a challenge on dual-mode systems, where the underlying machine has only user mode and kernel mode. In this section, we examine the building blocks that are needed for efficient virtualization. Note that these building blocks are not required by type 0 hypervisors, as discussed in Section 18.5.2.

The ability to virtualize depends on the features provided by the CPU. If the features are sufficient, then it is possible to write a VMM that provides a guest environment. Otherwise, virtualization is impossible. VMMs use several techniques to implement virtualization, including trap-and-emulate and binary translation. We discuss each of these techniques in this section, along with the hardware support needed to support virtualization.

As you read the section, keep in mind that an important concept found in most virtualization options is the implementation of a **virtual CPU** (**VCPU**). The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing that guest's current CPU state. When the guest is context-switched onto a CPU by the VMM, information from the VCPU is used to load the right context, much as a general-purpose operating system would use the PCB.

18.4.1 Trap-and-Emulate

On a typical dual-mode system, the virtual machine guest can execute only in user mode (unless extra hardware support is provided). The kernel, of course, runs in kernel mode, and it is not safe to allow user-level code to run in kernel mode. Just as the physical machine has two modes, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call, an interrupt, or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode in the virtual machine.

How can such a transfer be accomplished? The procedure is as follows: When the kernel in the guest attempts to execute a privileged instruction, that is an error (because the system is in user mode) and causes a trap to the VMM in the real machine. The VMM gains control and executes (or "emulates") the action that was attempted by the guest kernel on the part of the guest. It

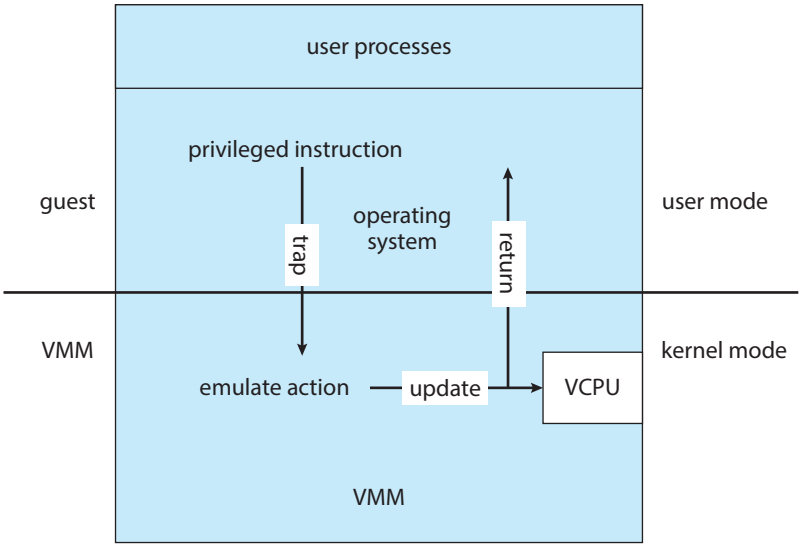

Figure 18.2 Trap-and-emulate virtualization implementation.

then returns control to the virtual machine. This is called the **trap-and-emulate** method and is shown in Figure 18.2.

With privileged instructions, time becomes an issue. All nonprivileged instructions run natively on the hardware, providing the same performance for guests as native applications. Privileged instructions create extra overhead, however, causing the guest to run more slowly than it would natively. In addition, the CPU is being multiprogrammed among many virtual machines, which can further slow down the virtual machines in unpredictable ways.

This problem has been approached in various ways. IBM VM, for example, allows normal instructions for the virtual machines to execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be emulated and hence execute more slowly. In general, with the evolution of hardware, the performance of trap-and-emulate functionality has been improved, and cases in which it is needed have been reduced. For example, many CPUs now have extra modes added to their standard dual-mode operation. The VCPU need not keep track of what mode the guest operating system is in, because the physical CPU performs that function. In fact, some CPUs provide guest CPU state management in hardware, so the VMM need not supply that functionality, removing the extra overhead.

18.4.2 Binary Translation

Some CPUs do not have a clean separation of privileged and nonprivileged instructions. Unfortunately for virtualization implementers, the Intel x86 CPU line is one of them. No thought was given to running virtualization on the x86 when it was designed. (In fact, the first CPU in the family—the Intel 4004, released in 1971—was designed to be the core of a calculator.) The chip has maintained backward compatibility throughout its lifetime, preventing changes that would have made virtualization easier through many generations.

Let's consider an example of the problem. The command `popf` loads the flag register from the contents of the stack. If the CPU is in privileged mode, all of the flags are replaced from the stack. If the CPU is in user mode, then only some flags are replaced, and others are ignored. Because no trap is generated if `popf` is executed in user mode, the trap-and-emulate procedure is rendered useless. Other x86 instructions cause similar problems. For the purposes of this discussion, we will call this set of instructions ***special instructions.*** As recently as 1998, using the trap-and-emulate method to implement virtualization on the x86 was considered impossible because of these special instructions.

This previously insurmountable problem was solved with the implementation of the **binary translation** technique. Binary translation is fairly simple in concept but complex in implementation. The basic steps are as follows:

1. If the guest VCPU is in user mode, the guest can run its instructions natively on a physical CPU.
2. If the guest VCPU is in kernel mode, then the guest believes that it is running in kernel mode. The VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on the guest's program counter. Instructions other than special instructions are run natively. Special instructions are translated into a new set of instructions that perform the equivalent task—for example, changing the flags in the VCPU.

Binary translation is shown in Figure 18.3. It is implemented by translation code within the VMM. The code reads native binary instructions dynamically from the guest, on demand, and generates native binary code that executes in place of the original code.

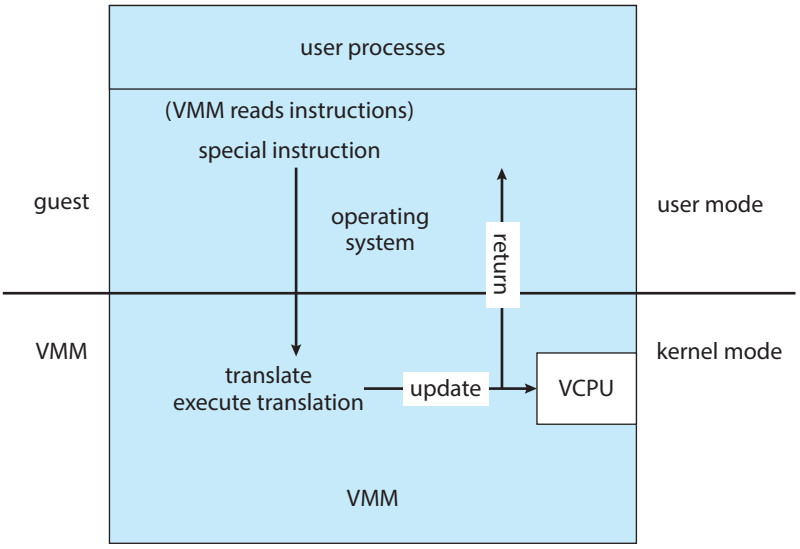

Figure 18.3 Binary translation virtualization implementation.

The basic method of binary translation just described would execute correctly but perform poorly. Fortunately, the vast majority of instructions would execute natively. But how could performance be improved for the other instructions? We can turn to a specific implementation of binary translation, the VMware method, to see one way of improving performance. Here, caching provides the solution. The replacement code for each instruction that needs to be translated is cached. All later executions of that instruction run from the translation cache and need not be translated again. If the cache is large enough, this method can greatly improve performance.

Let's consider another issue in virtualization: memory management, specifically the page tables. How can the VMM keep page-table state both for guests that believe they are managing the page tables and for the VMM itself? A common method, used with both trap-and-emulate and binary translation, is to use **nested page tables** (**NPTs**). Each guest operating system maintains one or more page tables to translate from virtual to physical memory. The VMM maintains NPTs to represent the guest's page-table state, just as it creates a VCPU to represent the guest's CPU state. The VMM knows when the guest tries to change its page table, and it makes the equivalent change in the NPT. When the guest is on the CPU, the VMM puts the pointer to the appropriate NPT into the appropriate CPU register to make that table the active page table. If the guest needs to modify the page table (for example, fulfilling a page fault), then that operation must be intercepted by the VMM and appropriate changes made to the nested and system page tables. Unfortunately, the use of NPTs can cause TLB misses to increase, and many other complexities need to be addressed to achieve reasonable performance.

Although it might seem that the binary translation method creates large amounts of overhead, it performed well enough to launch a new industry aimed at virtualizing Intel x86-based systems. VMware tested the performance impact of binary translation by booting one such system, Windows XP, and immediately shutting it down while monitoring the elapsed time and the number of translations produced by the binary translation method. The result was 950,000 translations, taking 3 microseconds each, for a total increase of 3 seconds (about 5 percent) over native execution of Windows XP. To achieve that result, developers used many performance improvements that we do not discuss here. For more information, consult the bibliographical notes at the end of this chapter.

18.4.3 Hardware Assistance

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature-rich and stable the virtual machines can be and the better they can perform. In the Intel x86 CPU family, Intel added new virtualization support (the **VT-x** instructions) in successive generations beginning in 2005. Now, binary translation is no longer needed.

In fact, all major general-purpose CPUs now provide extended hardware support for virtualization. For example, AMD virtualization technology (**AMD-V**) has appeared in several AMD processors starting in 2006. It defines two new modes of operation—host and guest—thus moving from a dual-mode to a

multimode processor. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to a guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees whatever devices are included in the host's definition of the guest. If the guest tries to access a virtualized resource, then control is passed to the VMM to manage that interaction. The functionality in Intel VT-x is similar, providing root and nonroot modes, equivalent to host and guest modes. Both provide guest VCPU state data structures to load and save guest CPU state automatically during guest context switches. In addition, **virtual machine control structures** (**VMCSs**) are provided to manage guest and host state, as well as various guest execution controls, exit controls, and information about why guests exit back to the host. In the latter case, for example, a nested page-table violation caused by an attempt to access unavailable memory can result in the guest's exit.

AMD and Intel have also addressed memory management in the virtual environment. With AMD's RVI and Intel's EPT memory-management enhancements, VMMs no longer need to implement software NPTs. In essence, these CPUs implement nested page tables in hardware to allow the VMM to fully control paging while the CPUs accelerate the translation from virtual to physical addresses. The NPTs add a new layer, one representing the guest's view of logical-to-physical address translation. The CPU page-table walking function (traversing the data structure to find the desired data) includes this new layer as necessary, walking through the guest table to the VMM table to find the physical address desired. A TLB miss results in a performance penalty, because more tables (the guest and host page tables) must be traversed to complete the lookup. Figure 18.4 shows the extra translation work performed by the hardware to translate from a guest virtual address to a final physical address.

I/O is another area improved by hardware assistance. Consider that the standard direct-memory-access (DMA) controller accepts a target memory address and a source I/O device and transfers data between the two without operating-system action. Without hardware assistance, a guest might try to set up a DMA transfer that affects the memory of the VMM or other guests. In CPUs that provide hardware-assisted DMA (such as Intel CPUs with VT-d), even DMA has a level of indirection. First, the VMM sets up **protection domains** to tell the CPU which physical memory belongs to each guest. Next, it assigns the I/O devices to the protection domains, allowing them direct access to those memory regions and only those regions. The hardware then transforms the address in a DMA request issued by an I/O device to the host physical memory address associated with the I/O. In this manner, DMA transfers are passed through between a guest and a device without VMM interference.

Similarly, interrupts must be delivered to the appropriate guest and must not be visible to other guests. By providing an interrupt remapping feature, CPUs with virtualization hardware assistance automatically deliver an interrupt destined for a guest to a core that is currently running a thread of that guest. That way, the guest receives interrupts without any need for the VMM to intercede in their delivery. Without interrupt remapping, malicious guests could generate interrupts that could be used to gain control of the host system. (See the bibliographical notes at the end of this chapter for more details.)

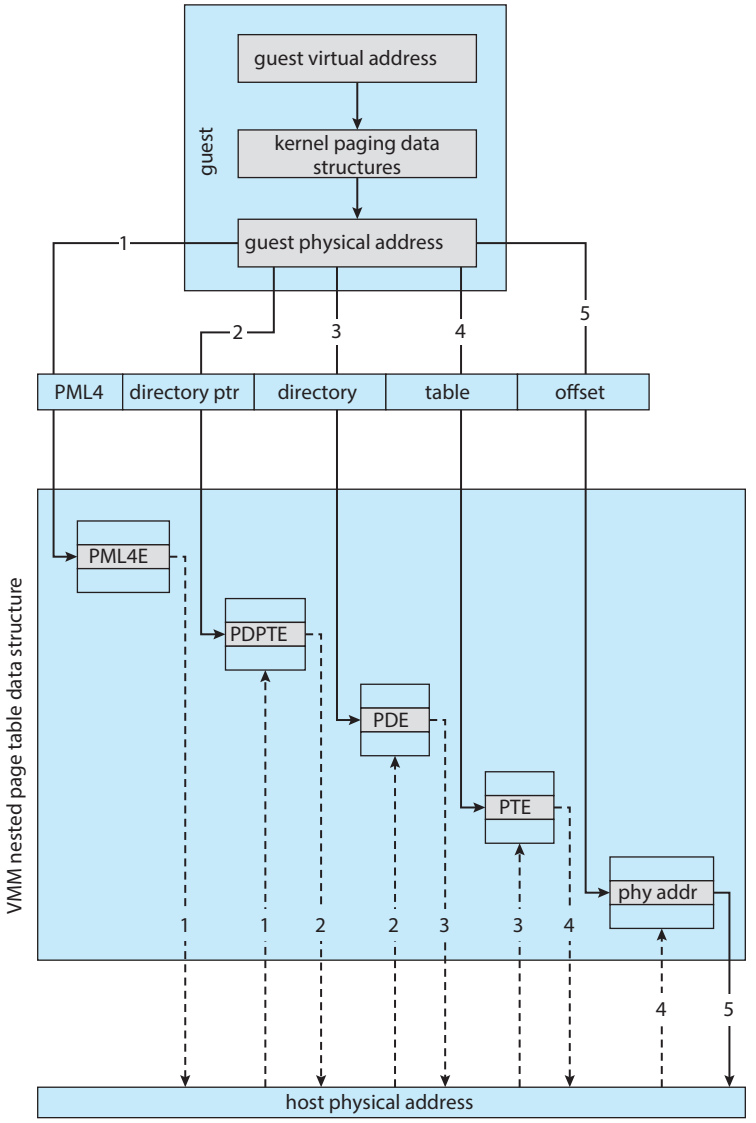

Figure 18.4 Nested page tables.

ARM architectures, specifically ARM v8 (64-bit) take a slightly different approach to hardware support of virtualization. They provide an entire exception level—EL2—which is even more privileged than that of the kernel (EL1). This allows the running of a secluded hypervisor, with its own MMU access and interrupt trapping. To allow for paravirtualization, a special instruction (HVC) is added. It allows the hypervisor to be called from guest kernels. This instruction can only be called from within kernel mode (EL1).

An interesting side effect of hardware-assisted virtualization is that it allows for the creation of thin hypervisors. A good example is macOS's hypervisor framework ("HyperVisor.framework"), which is an operating-system-supplied library that allows the creation of virtual machines in a few lines of

code. The actual work is done via system calls, which have the kernel call the privileged virtualization CPU instructions on behalf of the hypervisor process, allowing management of virtual machines without the hypervisor needing to load a kernel module of its own to execute those calls.

18.5 Types of VMs and Their Implementations

We've now looked at some of the techniques used to implement virtualization. Next, we consider the major types of virtual machines, their implementation, their functionality, and how they use the building blocks just described to create a virtual environment. Of course, the hardware on which the virtual machines are running can cause great variation in implementation methods. Here, we discuss the implementations in general, with the understanding that VMMs take advantage of hardware assistance where it is available.

18.5.1 The Virtual Machine Life Cycle

Let's begin with the virtual machine life cycle. Whatever the hypervisor type, at the time a virtual machine is created, its creator gives the VMM certain parameters. These parameters usually include the number of CPUs, amount of memory, networking details, and storage details that the VMM will take into account when creating the guest. For example, a user might want to create a new guest with two virtual CPUs, 4 GB of memory, 10 GB of disk space, one network interface that gets its IP address via DHCP, and access to the DVD drive.

The VMM then creates the virtual machine with those parameters. In the case of a type 0 hypervisor, the resources are usually dedicated. In this situation, if there are not two virtual CPUs available and unallocated, the creation request in our example will fail. For other hypervisor types, the resources are dedicated or virtualized, depending on the type. Certainly, an IP address cannot be shared, but the virtual CPUs are usually multiplexed on the physical CPUs as discussed in Section 18.6.1. Similarly, memory management usually involves allocating more memory to guests than actually exists in physical memory. This is more complicated and is described in Section 18.6.2.

Finally, when the virtual machine is no longer needed, it can be deleted. When this happens, the VMM first frees up any used disk space and then removes the configuration associated with the virtual machine, essentially forgetting the virtual machine.

These steps are quite simple compared with building, configuring, running, and removing physical machines. Creating a virtual machine from an existing one can be as easy as clicking the "clone" button and providing a new name and IP address. This ease of creation can lead to **virtual machine sprawl**, which occurs when there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track.

18.5.2 Type 0 Hypervisor

Type 0 hypervisors have existed for many years under many names, including "partitions" and "domains." They are a hardware feature, and that brings its own positives and negatives. Operating systems need do nothing special to take advantage of their features. The VMM itself is encoded in the firmware and

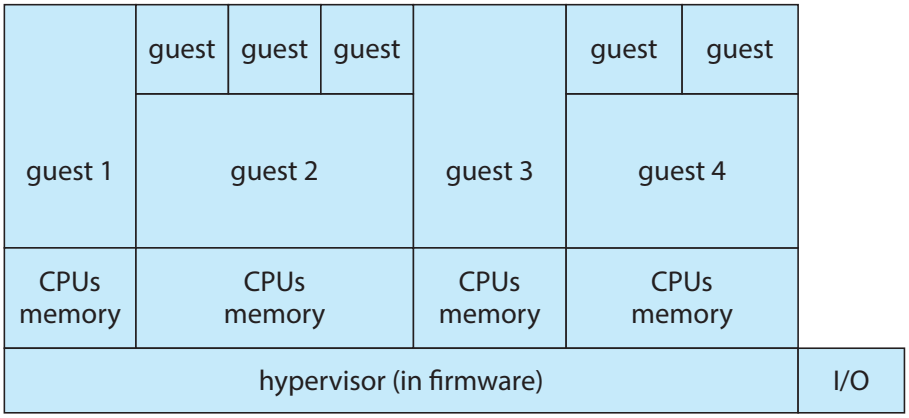

Figure 18.5 Type 0 hypervisor.

loaded at boot time. In turn, it loads the guest images to run in each partition. The feature set of a type 0 hypervisor tends to be smaller than those of the other types because it is implemented in hardware. For example, a system might be split into four virtual systems, each with dedicated CPUs, memory, and I/O devices. Each guest believes that it has dedicated hardware because it does, simplifying many implementation details.

I/O presents some difficulty, because it is not easy to dedicate I/O devices to guests if there are not enough. What if a system has two Ethernet ports and more than two guests, for example? Either all guests must get their own I/O devices, or the system must provided I/O device sharing. In these cases, the hypervisor manages shared access or grants all devices to a **control partition**. In the control partition, a guest operating system provides services (such as networking) via daemons to other guests, and the hypervisor routes I/O requests appropriately. Some type 0 hypervisors are even more sophisticated and can move physical CPUs and memory between running guests. In these cases, the guests are paravirtualized, aware of the virtualization and assisting in its execution. For example, a guest must watch for signals from the hardware or VMM that a hardware change has occurred, probe its hardware devices to detect the change, and add or subtract CPUs or memory from its available resources.

Because type 0 virtualization is very close to raw hardware execution, it should be considered separately from the other methods discussed here. A type 0 hypervisor can run multiple guest operating systems (one in each hardware partition). All of those guests, because they are running on raw hardware, can in turn be VMMs. Essentially, each guest operating system in a type 0 hypervisor is a native operating system with a subset of hardware made available to it. Because of that, each can have its own guest operating systems (Figure 18.5). Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

18.5.3 Type 1 Hypervisor

Type 1 hypervisors are commonly found in company data centers and are, in a sense, becoming "the data-center operating system." They are special-purpose operating systems that run natively on the hardware, but rather than providing

system calls and other interfaces for running programs, they create, run, and manage guest operating systems. In addition to running on standard hardware, they can run on type 0 hypervisors, but not on other type 1 hypervisors. Whatever the platform, guests generally do not know they are running on anything but the native hardware.

Type 1 hypervisors run in kernel mode, taking advantage of hardware protection. Where the host CPU allows, they use multiple modes to give guest operating systems their own control and improved performance. They implement device drivers for the hardware they run on, since no other component could do so. Because they are operating systems, they must also provide CPU scheduling, memory management, I/O management, protection, and even security. Frequently, they provide APIs, but those APIs support applications in guests or external applications that supply features like backups, monitoring, and security. Many type 1 hypervisors are closed-source commercial offerings, such as VMware ESX, while some are open source or hybrids of open and closed source, such as Citrix XenServer and its open Xen counterpart.

By using type 1 hypervisors, data-center managers can control and manage the operating systems and applications in new and sophisticated ways. An important benefit is the ability to consolidate more operating systems and applications onto fewer systems. For example, rather than having ten systems running at 10 percent utilization each, a data center might have one server manage the entire load. If utilization increases, guests and their applications can be moved to less-loaded systems live, without interruption of service. Using snapshots and cloning, the system can save the states of guests and duplicate those states—a much easier task than restoring from backups or installing manually or via scripts and tools. The price of this increased manageability is the cost of the VMM (if it is a commercial product), the need to learn new management tools and methods, and the increased complexity.

Another type of type 1 hypervisor includes various general-purpose operating systems with VMM functionality. Here, an operating system such as Red-Hat Enterprise Linux, Windows, or Oracle Solaris performs its normal duties as well as providing a VMM allowing other operating systems to run as guests. Because of their extra duties, these hypervisors typically provide fewer virtualization features than other type 1 hypervisors. In many ways, they treat a guest operating system as just another process, but they provide special handling when the guest tries to execute special instructions.

18.5.4 Type 2 Hypervisor

Type 2 hypervisors are less interesting to us as operating-system explorers, because there is very little operating-system involvement in these application-level virtual machine managers. This type of VMM is simply another process run and managed by the host, and even the host does not know that virtualization is happening within the VMM.

Type 2 hypervisors have limits not associated with some of the other types. For example, a user needs administrative privileges to access many of the hardware assistance features of modern CPUs. If the VMM is being run by a standard user without additional privileges, the VMM cannot take advantage of these features. Due to this limitation, as well as the extra overhead of running

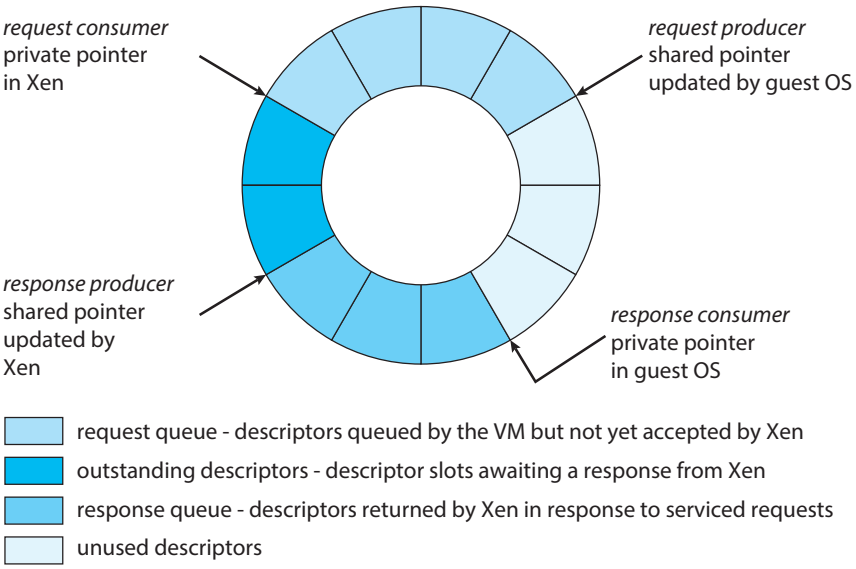

Figure 18.6 Xen I/O via shared circular buffer.[1]

a general-purpose operating system as well as guest operating systems, type 2 hypervisors tend to have poorer overall performance than type 0 or type 1.

As is often the case, the limitations of type 2 hypervisors also provide some benefits. They run on a variety of general-purpose operating systems, and running them requires no changes to the host operating system. A student can use a type 2 hypervisor, for example, to test a non-native operating system without replacing the native operating system. In fact, on an Apple laptop, a student could have versions of Windows, Linux, Unix, and less common operating systems all available for learning and experimentation.

18.5.5 Paravirtualization

As we've seen, paravirtualization works differently than the other types of virtualization. Rather than try to trick a guest operating system into believing it has a system to itself, paravirtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized virtual hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

The Xen VMM became the leader in paravirtulization by implementing several techniques to optimize the performance of guests as well as of the host system. For example, as mentioned earlier, some VMMs present virtual devices to guests that appear to be real devices. Instead of taking that approach, the Xen VMM presented clean and simple device abstractions that allow efficient I/O as well as good I/O-related communication between the guest and the VMM. For

[1]Barham, Paul. "Xen and the Art of Virtualization". SOSP '03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, p 164-177. ©2003 Association for Computing Machinery, Inc

each device used by each guest, there was a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer, as shown in Figure 18.6.

For memory management, Xen did not implement nested page tables. Rather, each guest had its own set of page tables, set to read-only. Xen required the guest to use a specific mechanism, a **hypercall** from the guest to the hypervisor VMM, when a page-table change was needed. This meant that the guest operating system's kernel code must have been changed from the default code to these Xen-specific methods. To optimize performance, Xen allowed the guest to queue up multiple page-table changes asynchronously via hypercalls and then checked to ensure that the changes were complete before continuing operation.

Xen allowed virtualization of x86 CPUs without the use of binary translation, instead requiring modifications in the guest operating systems like the one described above. Over time, Xen has taken advantage of hardware features supporting virtualization. As a result, it no longer requires modified guests and essentially does not need the paravirtualization method. Paravirtualization is still used in other solutions, however, such as type 0 hypervisors.

18.5.6 Programming-Environment Virtualization

Another kind of virtualization, based on a different execution model, is the virtualization of programming *environments*. Here, a programming language is designed to run within a custom-built virtualized environment. For example, Oracle's Java has many features that depend on its running in the **Java virtual machine** (**JVM**), including specific methods for security and memory management.

If we define virtualization as including only duplication of hardware, this is not really virtualization at all. But we need not limit ourselves to that definition. Instead, we can define a virtual environment, based on APIs, that provides a set of features we want to have available for a particular language and programs written in that language. Java programs run within the JVM environment, and the JVM is compiled to be a native program on systems on which it runs. This arrangement means that Java programs are written once and then can run on any system (including all of the major operating systems) on which a JVM is available. The same can be said of **interpreted languages**, which run inside programs that read each instruction and interpret it into native operations.

18.5.7 Emulation

Virtualization is probably the most common method for running applications designed for one operating system on a different operating system, but on the same CPU. This method works relatively efficiently because the applications were compiled for the instruction set that the target system uses.

But what if an application or operating system needs to run on a different CPU? Here, it is necessary to translate all of the source CPU's instructions so that they are turned into the equivalent instructions of the target CPU. Such an environment is no longer virtualized but rather is fully emulated.

Emulation is useful when the host system has one system architecture and the guest system was compiled for a different architecture. For example,

suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine.

As may be expected, the major challenge of emulation is performance. Instruction-set emulation may run an order of magnitude slower than native instructions, because it may take ten instructions on the new system to read, parse, and simulate an instruction from the old system. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge for emulator writers is that it is difficult to create a correct emulator because, in essence, this task involves writing an entire CPU in software.

In spite of these challenges, emulation is very popular, particularly in gaming circles. Many popular video games were written for platforms that are no longer in production. Users who want to run those games frequently can find an emulator of such a platform and then run the game unmodified within the emulator. Modern systems are so much faster than old game consoles that even the Apple iPhone has game emulators and games available to run within them.

18.5.8 Application Containment

The goal of virtualization in some instances is to provide a method to segregate applications, manage their performance and resource use, and create an easy way to start, stop, move, and manage them. In such cases, perhaps full-fledged virtualization is not needed. If the applications are all compiled for the same operating system, then we do not need complete virtualization to provide these features. We can instead use application containment.

Consider one example of application containment. Starting with version 10, Oracle Solaris has included **containers**, or **zones**, that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a zone with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU and memory resources can be divided among the zones and the system-wide processes. Each zone, in fact, can run its own scheduler to optimize the performance of its applications on the allotted resources. Figure 18.7 shows a Solaris 10 system with two containers and the standard "global" user space.

Containers are much lighter weight than other virtualization methods. That is, they use fewer system resources and are faster to instantiate and destroy, more similar to processes than virtual machines. For this reason, they are becoming more commonly used, especially in cloud computing. FreeBSD was perhaps the first operating system to include a container-like feature (called "jails"), and AIX has a similar feature. Linux added the **LXC** container feature in 2014. It is now included in the common Linux distributions via

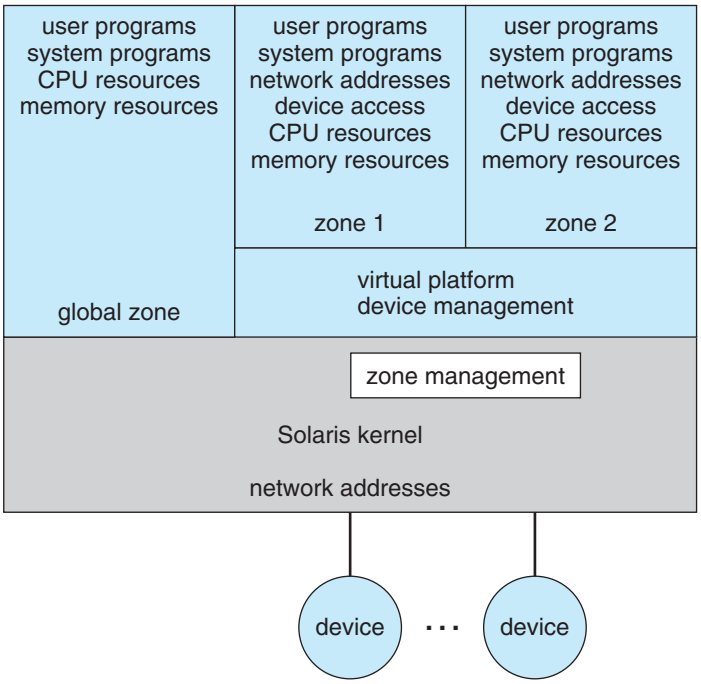

Figure 18.7 Solaris 10 with two zones.

a flag in the `clone()` system call. (The source code for LXCis available at https://linuxcontainers.org/lxc/downloads.)

Containers are also easy to automate and manage, leading to orchestration tools like **docker** and **Kubernetes**. Orchestration tools are means of automating and coordinating systems and services. Their aim is to make it simple to run entire suites of distributed applications, just as operating systems make it simple to run a single program. These tools offer rapid deployment of full applications, consisting of many processes within containers, and also offer monitoring and other administration features. For more on docker, see https://www.docker.com/what-docker. Information about Kubernetes can be found at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes.

18.6 Virtualization and Operating-System Components

Thus far, we have explored the building blocks of virtualization and the various types of virtualization. In this section, we take a deeper dive into the operating-system aspects of virtualization, including how the VMM provides core operating-system functions like scheduling, I/O, and memory management. Here, we answer questions such as these: How do VMMs schedule CPU use when guest operating systems believe they have dedicated CPUs? How can memory management work when many guests require large amounts of memory?

18.6.1 CPU Scheduling

A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system. The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.

The significant variations among virtualization technologies make it difficult to summarize the effect of virtualization on scheduling. First, let's consider the general case of VMM scheduling. The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs. In this situation, the guests act much like native operating systems running on native CPUs.

Of course, in other situations, there may not be enough CPUs to go around. The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests by scheduling its threads across all of the system CPUs, but the impact of this action is relatively minor. More difficult is the case of **overcommitment**, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms. For example, if there are six hardware CPUs and twelve guest-allocated CPUs, the VMM can allocate CPU resources proportionally, giving each guest half of the CPU resources it believes it has. The VMM can still present all twelve virtual CPUs to the guests, but in mapping them onto physical CPUs, the VMM can use its scheduler to distribute them appropriately.

Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will most likely be negatively affected by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system receives only what CPU resources the virtualization system gives it. A 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system can be even more serious.

The net outcome of such scheduling is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed are scheduling all of the cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the scheduling-algorithm efforts of the operating systems within virtual machines.

To correct for this, the VMM makes an application available for each type of operating system that the system administrator can install into the guests. This application corrects clock drift and can have other functions, such as virtual device management.

18.6.2 Memory Management

Efficient memory use in general-purpose operating systems is a major key to performance. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use. Further adding to this pressure is the fact that VMMs typically overcommit memory, so that the total memory allocated to guests exceeds the amount that physically exists in the system. The extra need for efficient memory use is not lost on the implementers of VMMs, who take extensive measures to ensure the optimal use of memory.

For example, VMware ESX uses several methods of memory management. Before memory optimization can occur, the VMM must establish how much real memory each guest should use. To do that, the VMM first evaluates each guest's maximum memory size. General-purpose operating systems do not expect the amount of memory in the system to change, so VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real-memory allocation for each guest based on the configured memory for that guest and other factors, such as overcommitment and system load. It then uses the three low-level mechanisms listed below to reclaim memory from the guests

1. Recall that a guest believes it controls memory allocation via its page-table management, whereas in reality the VMM maintains a nested page table that translates the guest page table to the real page table. The VMM can use this extra level of indirection to optimize the guest's use of memory without the guest's knowledge or help. One approach is to provide double paging. Here, the VMM has its own page-replacement algorithms and loads pages into a backing store that the guest believes is physical memory. Of course, the VMM knows less about the guest's memory access patterns than the guest does, so its paging is less efficient, creating performance problems. VMMs do use this method when other methods are not available or are not providing enough free memory. However, it is not the preferred approach.

2. A common solution is for the VMM to install in each guest a pseudo–device driver or kernel module that the VMM controls. (A **pseudo–device driver** uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This ***balloon memory manager*** communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory. Recall that pinning locks a page into physical memory so that it cannot be moved or paged out. To the guest, these pinned pages appear to decrease the amount of physical memory it has available, creating memory pressure. The guest then may free up other physical memory to be sure it has enough free memory. Meanwhile, the VMM, knowing that the pages pinned by the balloon process will never be used, removes those physical pages from the guest and allocates them to another guest. At the same time, the guest is using its own memory-management and paging algorithms to manage the available memory, which is the most

efficient option. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.

3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled. That hash value is a "thumbprint" of the page. The hash of every page examined is compared with other hashes stored in a hash table. If there is a match, the pages are compared byte by byte to see if they really are identical. If they are, one page is freed, and its logical address is mapped to the other's physical address. This technique might seem at first to be ineffective, but consider that guests run operating systems. If multiple guests run the same operating system, then only one copy of the active operating-system pages need be in memory. Similarly, multiple guests could be running the same set of applications, again a likely source of memory sharing.

The overall effect of these mechanisms is to enable guests to behave and perform as if they had the full amount of memory requested, although in reality they have less.

18.6.3 I/O

In the area of I/O, hypervisors have some leeway and can be less concerned with how they represent the underlying hardware to their guests. Because of the wide variation in I/O devices, operating systems are used to dealing with varying and flexible I/O mechanisms. For example, an operating system's device-driver mechanism provides a uniform interface to the operating system whatever the I/O device. Device-driver interfaces are designed to allow third-party hardware manufacturers to provide device drivers connecting their devices to the operating system. Usually, device drivers can be dynamically loaded and unloaded. Virtualization takes advantage of this built-in flexibility by providing specific virtualized devices to guest operating systems.

As described in Section 18.5, VMMs vary greatly in how they provide I/O to their guests. I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM, which sends the requests to a more complicated real device through a more complex real device driver. I/O in virtual environments is complicated and requires careful VMM design and implementation.

Consider the case of a hypervisor and hardware combination that allows devices to be dedicated to a guest and allows the guest to access those devices directly. Of course, a device dedicated to one guest is not available to any other guests, but this direct access can still be useful in some circumstances. The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With type 0 hypervisors that provide direct device access, guests can often

run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance may suffer.

With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present. The hardware needs to provide DMA pass-through with facilities like VT-d, as well as direct interrupt delivery (interrupts going directly to the guests). Given how frequently interrupts occur, it should be no surprise that the guests on hardware without these features have worse performance than if they were running natively.

In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while the device is being shared, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.

In the area of networking, VMMs also have work to do. General-purpose operating systems typically have one Internet protocol (IP) address, although they sometimes have more than one—for example, to connect to a management network, backup network, and production network. With virtualization, each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guests.

The guests can be "directly" connected to the network by an IP address that is seen by the broader network (this is known as **bridging**). Alternatively, the VMM can provide a **network address translation** (**NAT**) address. The NAT address is local to the server on which the guest is running, and the VMM provides routing between the broader network and the guest. The VMM also provides firewalling to guard connections between guests within the system and between guests and external systems.

18.6.4 Storage Management

An important question in determining how virtualization works is this: If multiple operating systems have been installed, what and where is the boot disk? Clearly, virtualized environments need to approach storage management differently than do native operating systems. Even the standard multiboot method of slicing the boot disk into partitions, installing a boot manager in one partition, and installing each other operating system in another partition is not sufficient, because partitioning has limits that would prevent it from working for tens or hundreds of virtual machines.

Once again, the solution to this problem depends on the type of hypervisor. Type 0 hypervisors often allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, a disk manager may be part of the control partition, and that disk manager may provide disk space (including boot disks) to the other partitions.

Type 1 hypervisors store the guest root disk (and configuration information) in one or more files in the file systems provided by the VMM. Type 2 hypervisors store the same information in the host operating system's file systems. In essence, a **disk image**, containing all of the contents of the root disk

of the guest, is contained in one file in the VMM. Aside from the potential performance problems that causes, this is a clever solution, because it simplifies copying and moving guests. If the administrator wants a duplicate of the guest (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting the new virtual machine brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.

Guests sometimes need more disk space than is available in their root disk image. For example, a nonvirtualized database server might use several file systems spread across many disks to store various parts of the database. Virtualizing such a database usually involves creating several files and having the VMM present those to the guest as disks. The guest then executes as usual, with the VMM translating the disk I/O requests coming from the guest into file I/O commands to the correct files.

Frequently, VMMs provide a mechanism to capture a physical system as it is currently configured and convert it to a guest that the VMM can manage and run. This **physical-to-virtual** (**P-to-V**) conversion reads the disk blocks of the physical system's disks and stores them in files on the VMM's system or on shared storage that the VMM can access. VMMs also provide a **virtual-to-physical** (**V-to-P**) procedure for converting a guest to a physical system. This procedure is sometimes needed for debugging: a problem could be caused by the VMM or associated components, and the administrator could attempt to solve the problem by removing virtualization from the problem variables. V-to-P conversion can take the files containing all of the guest data and generate disk blocks on a physical disk, recreating the guest as a native operating system and applications. Once the testing is concluded, the original system can be reused for other purposes when the virtual machine returns to service, or the virtual machine can be deleted and the original system can continue to run.

18.6.5 Live Migration

One feature not found in general-purpose operating systems but found in type 0 and type 1 hypervisors is the live migration of a running guest from one system to another. We mentioned this capability earlier. Here, we explore the details of how live migration works and why VMMs can implement it relatively easily while general-purpose operating systems, in spite of some research attempts, cannot.

First, let's consider how live migration works. A running guest on one system is copied to another system running the same VMM. The copy occurs with so little interruption of service that users logged in to the guest, as well as network connections to the guest, continue without noticeable impact. This rather astonishing ability is very powerful in resource management and hardware administration. After all, compare it with the steps necessary without virtualization: we must warn users, shut down the processes, possibly move the binaries, and restart the processes on the new system. Only then can users access the services again. With live migration, we can decrease the load on an overloaded system or make hardware or system changes with no discernable disruption for users.

Live migration is made possible by the well-defined interface between each guest and the VMM and the limited state the VMM maintains for the guest. The VMM migrates a guest via the following steps:

1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read–write pages to the target, marking them as clean.
5. The source repeats step 4, because during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, other state details, and the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.

This sequence is shown in Figure 18.8.

We conclude this discussion with a few interesting details and limitations concerning live migration. First, for network connections to continue uninterrupted, the network infrastructure needs to understand that a MAC address—the hardware networking address—can move between systems. Before virtualization, this did not happen, as the MAC address was tied to physical hardware. With virtualization, the MAC must be movable for existing networking connections to continue without resetting. Modern network switches understand this and route traffic wherever the MAC address is, even accommodating a move.

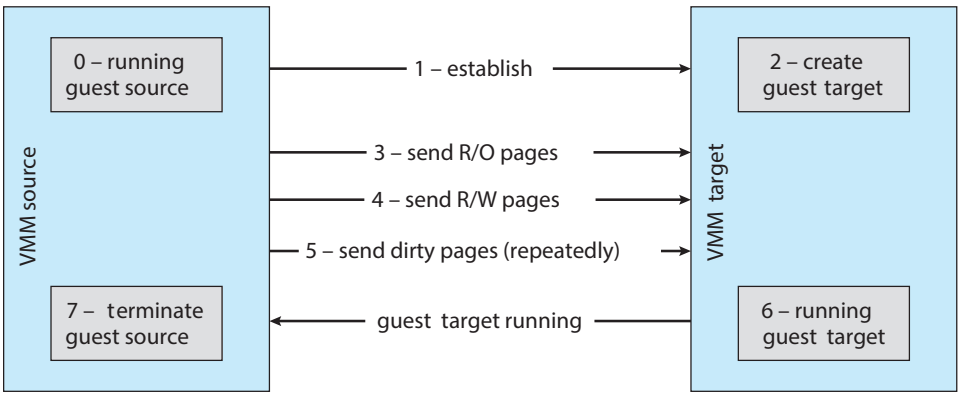

Figure 18.8 Live migration of a guest between two servers.

A limitation of live migration is that no disk state is transferred. One reason live migration is possible is that most of the guest's state is maintained within the guest—for example, open file tables, system-call state, kernel state, and so on. Because disk I/O is much slower than memory access, however, and used disk space is usually much larger than used memory, disks associated with the guest cannot be moved as part of a live migration. Rather, the disk must be remote to the guest, accessed over the network. In that case, disk access state is maintained within the guest, and network connections are all that matter to the VMM. The network connections are maintained during the migration, so remote disk access continues. Typically, NFS, CIFS, or iSCSI is used to store virtual machine images and any other storage a guest needs access to. These network-based storage accesses simply continue when the network connections are continued once the guest has been migrated.

Live migration makes it possible to manage data centers in entirely new ways. For example, virtualization management tools can monitor all the VMMs in an environment and automatically balance resource use by moving guests between the VMMs. These tools can also optimize the use of electricity and cooling by migrating all guests off selected servers if other servers can handle the load and powering down the selected servers entirely. If the load increases, the tools can power up the servers and migrate guests back to them.

18.7 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into greater use as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation and the Java virtual machine. These virtual machines can typically run on top of operating systems of any of the design types discussed in earlier chapters.

18.7.1 VMware

VMware Workstation is a popular commercial application that abstracts Intel x86 and compatible hardware into isolated virtual machines. VMware Workstation is a prime example of a Type 2 hypervisor. It runs as an application on a host operating system such as Windows or Linux and allows this host system to run several different guest operating systems concurrently as independent virtual machines.

The architecture of such a system is shown in Figure 18.9. In this scenario, Linux is running as the host operating system, and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. At the heart of VMware is the virtualization layer, which abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The physical disk that the guest owns and manages is really just a file within the file system of the host operating system. To create an identical guest, we can simply copy the file. Copying the file to another location protects the guest against a disaster at the original site. Moving the file to another location

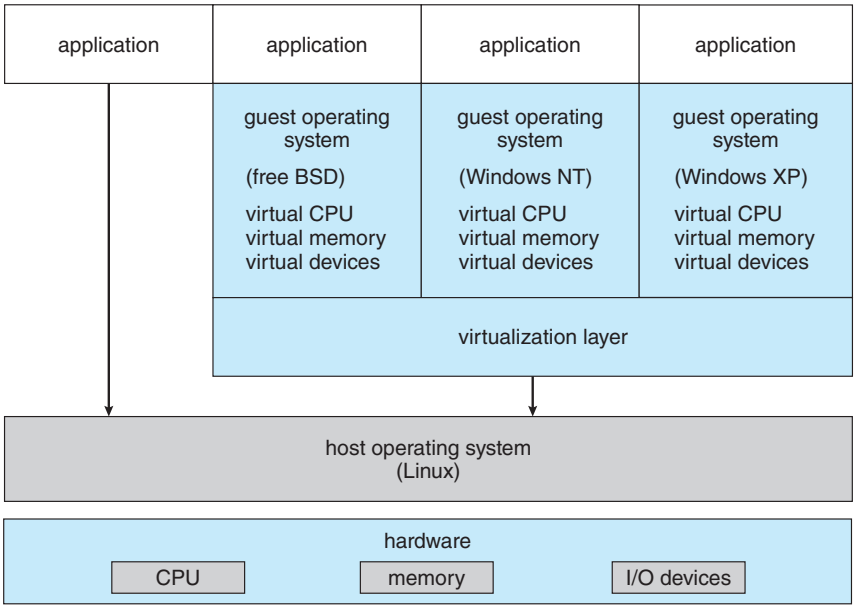

Figure 18.9 VMware Workstation architecture.

moves the guest system. Such capabilities, as explained earlier, can improve the efficiency of system administration as well as system resource use.

18.7.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java provides a specification for a Java virtual machine, or JVM. Java therefore is an example of programming-environment virtualization, as discussed in Section 18.5.6.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 18.10. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and that it does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

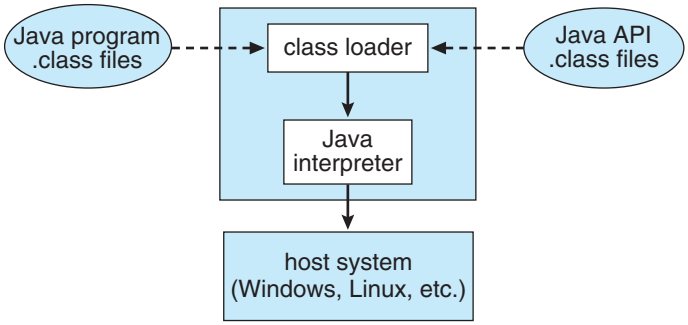

Figure 18.10 The Java virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or macOS, or as part of a web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time** (JIT) compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions, and the bytecode operations need not be interpreted all over again. Running the JVM in hardware is potentially even faster. Here, a special Java chip executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

18.8 Virtualization Research

As mentioned earlier, machine virtualization has enjoyed growing popularity in recent years as a means of solving system compatibility problems. Research has expanded to cover many other uses of machine virtualization, including support for microservices running on library operating systems and secure partitioning of resources in embedded systems. Consequently, quite a lot of interesting, active research is underway.

Frequently, in the context of cloud computing, the same application is run on thousands of systems. To better manage those deployments, they can be virtualized. But consider the execution stack in that case—the application on top of a service-rich general-purpose operating system within a virtual machine managed by a hypervisor. Projects like **unikernels**, built on **library operating systems**, aim to improve efficiency and security in these environments. Unikernels are specialized machine images, using one address space, that shrink the attack surface and resource footprint of deployed applications. In essence, they compile the application, the system libraries it calls, and the kernel services it uses into a single binary that runs within a virtual environment (or even on bare metal). While research into changing how operating system kernels, hardware, and applications interact is not new (see https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf,

for example), cloud computing and virtualization have created renewed interest in the area. See http://unikernel.org for more details.

The virtualization instructions in modern CPUs have given rise to a new branch of virtualization research focusing not on more efficient use of hardware but rather on better control of processes. Partitioning hypervisors partition the existing machine physical resources amongst guests, thereby fully committing rather than overcommitting machine resources. Partitioning hypervisors can securely extend the features of an existing operating system via functionality in another operating system (run in a separate guest VM domain), running on a subset of machine physical resources. This avoids the tedium of writing an entire operating system from scratch. For example, a Linux system that lacks real-time capabilities for safety- and security-critical tasks can be extended with a lightweight real-time operating system running in its own virtual machine. Traditional hypervisors have higher overhead than running native tasks, so a new type of hypervisor is needed.

Each task runs within a virtual machine, but the hypervisor only initializes the system and starts the tasks and is not involved with continuing operation. Each virtual machine has its own allocated hardware and is free to manage that hardware without interference from the hypervisor. Because the hypervisor does not interrupt task operations and is not called by the tasks, the tasks can have real-time aspects and can be much more secure.

Within the class of partitioning hypervisors are the **Quest-V**, **eVM**, **Xtratum** and **Siemens Jailhouse** projects. These are **separation hypervisors** (see http://www.csl.sri.com/users/rushby/papers/sosp81.pdf) that use virtualization to partition separate system components into a chip-level distributed system. Secure shared memory channels are then implemented using hardware extended page tables so that separate sandboxed guests can communicate with one another. The targets of these projects are areas such as robotics, self-driving cars, and the Internet of Things. See https://www.cs.bu.edu/richwest/papers/west-tocs16.pdf for more details.

18.9 Summary

- Virtualization is a method for providing a guest with a duplicate of a system's underlying hardware. Multiple guests can run on a given system, each believing that it is the native operating system and is in full control.
- Virtualization started as a method to allow IBM to segregate users and provide them with their own execution environments on IBM mainframes. Since then, thanks to improvements in system and CPU performance and innovative software techniques, virtualization has become a common feature in data centers and even on personal computers. Because of its popularity, CPU designers have added features to support virtualization. This snowball effect is likely to continue, with virtualization and its hardware support increasing over time.
- The virtual machine manager, or hypervisor, creates and runs the virtual machine. Type 0 hypervisors are implemented in the hardware and require modifications to the operating system to ensure proper operation. Some

type 0 hypervisors offer an example of paravirtualization, in which the operating system is aware of virtualization and assists in its execution.

- Type 1 hypervisors provide the environment and features needed to create, run, and manage guest virtual machines. Each guest includes all of the software typically associated with a full native system, including the operating system, device drivers, applications, user accounts, and so on.
- Type 2 hypervisors are simply applications that run on other operating systems, which do not know that virtualization is taking place. These hypervisors do not have hardware or host support so must perform all virtualization activities in the context of a process.
- Programming-environment virtualization is part of the design of a programming language. The language specifies a containing application in which programs run, and this application provides services to the programs.
- Emulation is used when a host system has one architecture and the guest was compiled for a different architecture. Every instruction the guest wants to execute must be translated from its instruction set to that of the native hardware. Although this method involves some performance penalty, it is balanced by the usefulness of being able to run old programs on newer, incompatible hardware or run games designed for old consoles on modern hardware.
- Implementing virtualization is challenging, especially when hardware support is minimal. The more features provided by the system, the easier virtualization is to implement and the better the performance of the guests.
- VMMs take advantage of whatever hardware support is available when optimizing CPU scheduling, memory management, and I/O modules to provide guests with optimum resource use while protecting the VMM from the guests and the guests from one another.
- Current research is extending the uses of virtualization. Unikernels aim to increase efficiency and decrease security attack surface by compiling an application, its libraries, and the kernel resources the application needs into one binary with one address space that runs within a virtual machine. Partitioning hypervisors provide secure execution, real-time operation, and other features traditionally only available to applications running on dedicated hardware.

Further Reading

The original IBM virtual machine is described in [Meyer and Seawright (1970)]. [Popek and Goldberg (1974)] established the characteristics that help define VMMs. Methods of implementing virtual machines are discussed in [Agesen et al. (2010)].

Intel x86 hardware virtualization support is described in [Neiger et al. (2006)]. AMD hardware virtualization support is described in a white paper available at http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf.

Memory management in VMware is described in [Waldspurger (2002)]. [Gordon et al. (2012)] propose a solution to the problem of I/O overhead in virtualized environments. Some protection challenges and attacks in virtual environments are discussed in [Wojtczuk and Ruthkowska (2011)].

For early work on alternative kernel designs, see https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf. For more on unikernels, see [West et al. (2016)] and http://unikernel.org. Partitioning hypervisors are discussed in http://ethdocs.org/en/latest/introduction/what-is-ethereum.html, and https://lwn.net/Articles/578295 and [Madhavapeddy et al. (2013)]. Quest-V, a separation hypervisor, is detailed in http://www.csl.sri.com/users/rushby/papers/sosp81.pdf and https://www.cs.bu.edu/ richwest/papers/west-tocs16.pdf.

The open-source ***VirtualBox*** project is available from http://www.virtualbox.org. The source code for LXC is available at https://linuxcontainers.org/lxc/downloads.

For more on docker, see https://www.docker.com/what-docker. Information about Kubernetes can be found at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes.

Bibliography

[Agesen et al. (2010)] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The Evolution of an x86 Virtual Machine Monitor", *Proceedings of the ACM Symposium on Operating Systems Principles* (2010), pages 3–18.

[Gordon et al. (2012)] A. Gordon, N. A. N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal Performance for I/O Virtualization", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pages 411–422.

[Madhavapeddy et al. (2013)] A. Madhavapeddy, R. Mirtier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library Operating Systems for the Cloud" (2013).

[Meyer and Seawright (1970)] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System", *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.

[Neiger et al. (2006)] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization", *Intel Technology Journal*, Volume 10, (2006).

[Popek and Goldberg (1974)] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures", *Communications of the ACM*, Volume 17, Number 7 (1974), pages 412–421.

[Waldspurger (2002)] C. Waldspurger, "Memory Resource Management in VMware ESX Server", *Operating Systems Review*, Volume 36, Number 4 (2002), pages 181–194.

[West et al. (2016)] R. West, Y. Li, E. Missimer, and M. Danish, "A Virtualized Separation Kernel for Mixed Criticality Systems", Volume 34, (2016).

[Wojtczuk and Ruthkowska (2011)] R. Wojtczuk and J. Ruthkowska, "Following the White Rabbit: Software Attacks Against Intel VT-d Technology", *The Invisible Things Lab's blog* (2011).

Chapter 18 Exercises

18.1 Describe the three types of traditional hypervisors.

18.2 Describe four virtualization-like execution environments, and explain how they differ from "true" virtualization.

18.3 Describe four benefits of virtualization.

18.4 Why are VMMs unable to implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap and emulate, what method can a VMM use to implement virtualization?

18.5 What hardware assistance for virtualization can be provided by modern CPUs?

18.6 Why is live migration possible in virtual environments but much less possible for a native operating system?

Networks and Distributed Systems

Updated by Sarah Diesburg

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate with one another through various networks, such as high-speed buses. Distributed systems are more relevant than ever, and you have almost certainly used some sort of distributed service. Applications of distributed systems range from providing transparent access to files inside an organization, to large-scale cloud file and photo storage services, to business analysis of trends on large data sets, to parallel processing of scientific data, and more. In fact, the most basic example of a distributed system is one we are all likely very familiar with—the Internet.

In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We also contrast the main differences in the types and roles of current distributed system designs. Finally, we investigate some of the basic designs and design challenges of distributed file systems.

CHAPTER OBJECTIVES

- Explain the advantages of networked and distributed systems.
- Provide a high-level overview of the networks that interconnect distributed systems.
- Define the roles and types of distributed systems in use today.
- Discuss issues concerning the design of distributed file systems.

19.1 Advantages of Distributed Systems

A **distributed system** is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.

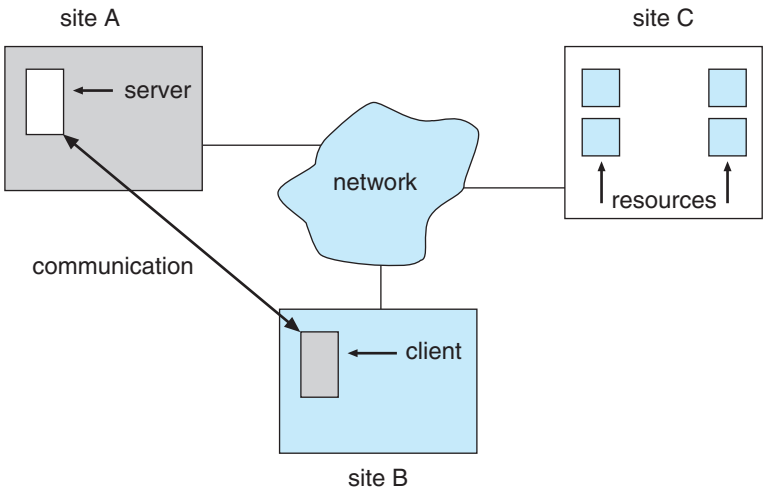

Figure 19.1 A client-server distributed system.

The nodes in a distributed system may vary in size and function. They may include small microprocessors, personal computers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *processors*, *sites*, *machines*, and *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine and *node* to refer to a specific system at a site. Nodes can exist in a *client–server* configuration, a *peer-to-peer* configuration, or a hybrid of these. In the common client-server configuration, one node at one site, the *server*, has a resource that another node, the *client* (or user), would like to use. A general structure of a client–server distributed system is shown in Figure 19.1. In a peer-to-peer configuration, there are no servers or clients. Instead, the nodes share equal responsibilities and can act as both clients and servers.

When several sites are connected to one another by a communication network, users at the various sites have the opportunity to exchange information. At a low level, **messages** are passed between systems, much as messages are passed between processes in the single-computer message system discussed in Section 3.4. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file storage, execution of applications, and remote procedure calls (RPCs).

There are three major reasons for building distributed systems: resource sharing, computational speedup, and reliability. In this section, we briefly discuss each of them.

19.1.1 Resource Sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may query a database located at site B. Meanwhile, a user at site B may access a file that resides at site A. In general, **resource sharing** in a distributed system provides mechanisms for

sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices such as a supercomputer or a **graphics processing unit** (GPU), and performing other operations.

19.1.2 Computation Speedup

If a particular computation can be partitioned into subcomputations that can run concurrently, then a distributed system allows us to distribute the subcomputations among the various sites. The subcomputations can be run concurrently and thus provide **computation speedup**. This is especially relevant when doing large-scale processing of big data sets (such as analyzing large amounts of customer data for trends). In addition, if a particular site is currently overloaded with requests, some of them can be moved or rerouted to other, more lightly loaded sites. This movement of jobs is called **load balancing** and is common among distributed system nodes and other services provided on the Internet.

19.1.3 Reliability

If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, however, the system is composed of diversified machines, each of which is responsible for some crucial system function (such as the web server or the file system), then a single failure may halt the operation of the whole system. In general, with enough redundancy (in both hardware and data), the system can continue operation even if some of its nodes have failed.

The failure of a node or site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly.

19.2 Network Structure

To completely understand the roles and types of distributed systems in use today, we need to understand the networks that interconnect them. This section serves as a network primer to introduce basic networking concepts and challenges as they relate to distributed systems. The rest of the chapter specifically discusses distributed systems.

There are basically two types of networks: **local-area networks** (LAN) and **wide-area networks** (WAN). The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of hosts distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of systems distributed over a large area (such as the United States). These differences

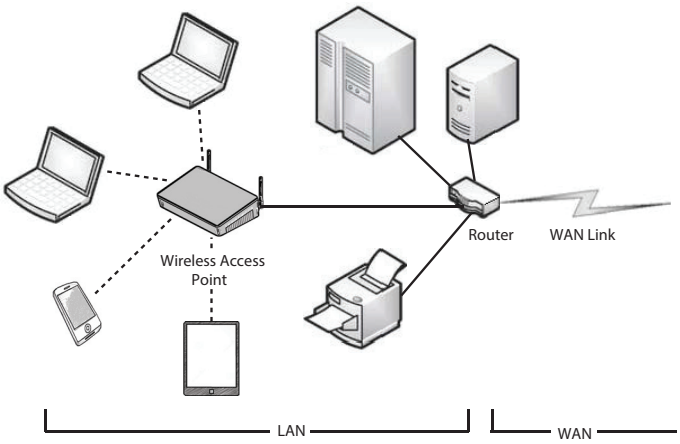

Figure 19.2 Local-area network.

imply major variations in the speed and reliability of the communications networks, and they are reflected in the distributed system design.

19.2.1 Local-Area Networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area, and they are generally used in an office or home environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than their counterparts in wide-area networks.

A typical LAN may consist of a number of different computers (including workstations, servers, laptops, tablets, and smartphones), various shared peripheral devices (such as printers and storage arrays), and one or more **routers** (specialized network communication processors) that provide access to other networks (Figure 19.2). Ethernet and **WiFi** are commonly used to construct LANs. *Wireless access points* connect devices to the LAN wirelessly, and they may or may not be routers themselves.

Ethernet networks are generally found in businesses and organizations in which computers and peripherals tend to be nonmobile. These networks use *coaxial, twisted pair,* and/or *fiber optic* cables to send signals. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard. Typical Ethernet speeds using common twisted-pair cabling

can vary from 10 Mbps to over 10 Gbps, with other types of cabling reaching speeds of 100 Gbps.

WiFi is now ubiquitous and either supplements traditional Ethernet networks or exists by itself. Specifically, WiFi allows us to construct a network without using physical cables. Each host has a wireless transmitter and receiver that it uses to participate in the network. WiFi is defined by the IEEE 802.11 standard. Wireless networks are popular in homes and businesses, as well as public areas such as libraries, Internet cafes, sports arenas, and even buses and airplanes. WiFi speeds can vary from 11 Mbps to over 400 Mbps.

Both the IEEE 802.3 and 802.11 standards are constantly evolving. For the latest information about various standards and speeds, see the references at the end of the chapter.

19.2.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the ARPANET. Begun in 1968, the ARPANET has grown from a four-site experimental network to a worldwide network of networks, the **Internet** (also known as the **World Wide Web**), comprising millions of computer systems.

Sites in a WAN are physically distributed over a large geographical area. Typical links are telephone lines, leased (dedicated data) lines, optical cable, microwave links, radio waves, and satellite channels. These communication links are controlled by routers (Figure 19.3) that are responsible for directing traffic to other routers and networks and transferring information among the various sites.

For example, the Internet WAN enables hosts at geographically separate sites to communicate with one another. The host computers typically differ from one another in speed, CPU type, operating system, and so on. Hosts are

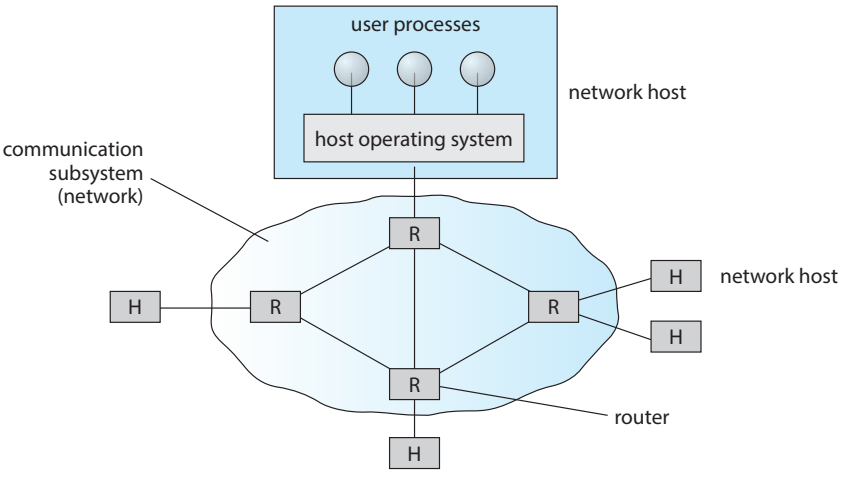

Figure 19.3 Communication processors in a wide-area network.

generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks are interlinked with routers to form the worldwide network. Residences can connect to the Internet by either telephone, cable, or specialized Internet service providers that install routers to connect the residences to central services. Of course, there are other WANs besides the Internet. A company, for example, might create its own private WAN for increased security, performance, or reliability.

WANs are generally slower than LANs, although backbone WAN connections that link major cities may have very fast transfer rates through fiber optic cables. In fact, many backbone providers have fiber optic speeds of 40 Gbps or 100 Gbps. (It is generally the links from local **Internet Service Providers** (**ISPs**) to homes or businesses that slow things down.) However, WAN links are being constantly updated to faster technologies as the demand for more speed continues to grow.

Frequently, WANs and LANs interconnect, and it is difficult to tell where one ends and the other starts. Consider the cellular phone data network. Cell phones are used for both voice and data communications. Cell phones in a given area connect via radio waves to a cell tower that contains receivers and transmitters. This part of the network is similar to a LAN except that the cell phones do not communicate with each other (unless two people talking or exchanging data happen to be connected to the same tower). Rather, the towers are connected to other towers and to hubs that connect the tower communications to land lines or other communication media and route the packets toward their destinations. This part of the network is more WAN-like. Once the appropriate tower receives the packets, it uses its transmitters to send them to the correct recipient.

19.3 Communication Structure

Now that we have discussed the physical aspects of networking, we turn to the internal workings.

19.3.1 Naming and Name Resolution

The first issue in network communication involves the naming of the systems in the network. For a process at site A to exchange information with a process at site B, each must be able to specify the other. Within a computer system, each process has a process identifier, and messages may be addressed with the process identifier. Because networked systems share no memory, however, a host within the system initially has no knowledge about the processes on other hosts.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where **host name** is a name unique within the network and **identifie** is a process identifier or other unique number within that host. A host name is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named *program*, *student*, *faculty*, and *cs*. The host name *program* is certainly easier to remember than the numeric host address *128.148.31.100*.

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to **resolve** the host name into a **host-id** that describes the destination system to the networking hardware. This mechanism is similar to the name-to-address binding that occurs during program compilation, linking, loading, and execution (Chapter 9). In the case of host names, two possibilities exist. First, every host may have a data file containing the names and numeric addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. In fact, in the early days of the ARPANET there was a canonical host file that was copied to every system periodically. As the network grew, however, this method became untenable.

The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The Internet uses a **domain-name system** (**DNS**) for host-name resolution.

DNS specifies the naming structure of the hosts, as well as name-to-address resolution. Hosts on the Internet are logically addressed with multipart names known as IP addresses. The parts of an IP address progress from the most specific to the most general, with periods separating the fields. For instance, *eric.cs.yale.edu* refers to host *eric* in the Department of Computer Science at Yale University within the top-level domain *edu*. (Other top-level domains include *com* for commercial sites and *org* for organizations, as well as a domain for each country connected to the network for systems specified by country rather than organization type.) Generally, the system resolves addresses by examining the host-name components in reverse order. Each component has a **name server**—simply a process on a system—that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted, and a host-id is returned. For example, a request made by a process on system A to communicate with *eric.cs.yale.edu* would result in the following steps:

1. The system library or the kernel on system A issues a request to the name server for the `edu` domain, asking for the address of the name server for `yale.edu`. The name server for the *edu* domain must be at a known address, so that it can be queried.
2. The `edu` name server returns the address of the host on which the `yale.edu` name server resides.
3. System A then queries the name server at this address and asks about `cs.yale.edu`.
4. An address is returned. Now, finally, a request to that address for `eric.cs.yale.edu` returns an Internet address host-id for that host (for example, `128.148.31.100`).

This protocol may seem inefficient, but individual hosts cache the IP addresses they have already resolved to speed the process. (Of course, the contents of these caches must be refreshed over time in case the name server is moved

```
/**
 * Usage: java DNSLookUp <IP name>
 * i.e. java DNSLookUp www.wiley.com
 */
public class DNSLookUp {
   public static void main(String[] args) {
      InetAddress hostAddress;

      try {
         hostAddress = InetAddress.getByName(args[0]);
         System.out.println(hostAddress.getHostAddress());
      }
      catch (UnknownHostException uhe) {
         System.err.println("Unknown host: " + args[0]);
      }
    }
}
```

Figure 19.4 Java program illustrating a DNS lookup.

or its address changes.) In fact, the protocol is so important that it has been optimized many times and has had many safeguards added. Consider what would happen if the primary edu name server crashed. It is possible that no edu hosts would be able to have their addresses resolved, making them all unreachable! The solution is to use secondary, backup name servers that duplicate the contents of the primary servers.

Before the domain-name service was introduced, all hosts on the Internet needed to have copies of a file (mentioned above) that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses had changed. Under the domain-name service, each name-server site is responsible for updating the host information for that domain. For instance, any host changes at Yale University are the responsibility of the name server for *yale.edu* and need not be reported anywhere else. DNS lookups will automatically retrieve the updated information because they will contact *yale.edu* directly. Domains may contain autonomous subdomains to further distribute the responsibility for host-name and host-id changes.

Java provides the necessary API to design a program that maps IP names to IP addresses. The program shown in Figure 19.4 is passed an IP name (such as *eric.cs.yale.edu*) on the command line and either outputs the IP address of the host or returns a message indicating that the host name could not be resolved. An `InetAddress` is a Java class representing an IP name or address. The static method `getByName()` belonging to the `InetAddress` class is passed a string representation of an IP name, and it returns the corresponding `InetAddress`. The program then invokes the `getHostAddress()` method, which internally uses DNS to look up the IP address of the designated host.

Generally, the operating system is responsible for accepting from its processes a message destined for <host name, identifier> and for transferring that message to the appropriate host. The kernel on the destination host is then responsible for transferring the message to the process named by the identifier. This process is described in Section 19.3.4.

19.3.2 Communication Protocols

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 19.5 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware.

The International Standards Organization created the Open Systems Interconnection (OSI) model for describing the various layers of networking. While these layers are not implemented in practice, they are useful for understanding how networking logically works, and we describe them below:

- **Layer 1: Physical layer**. The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data

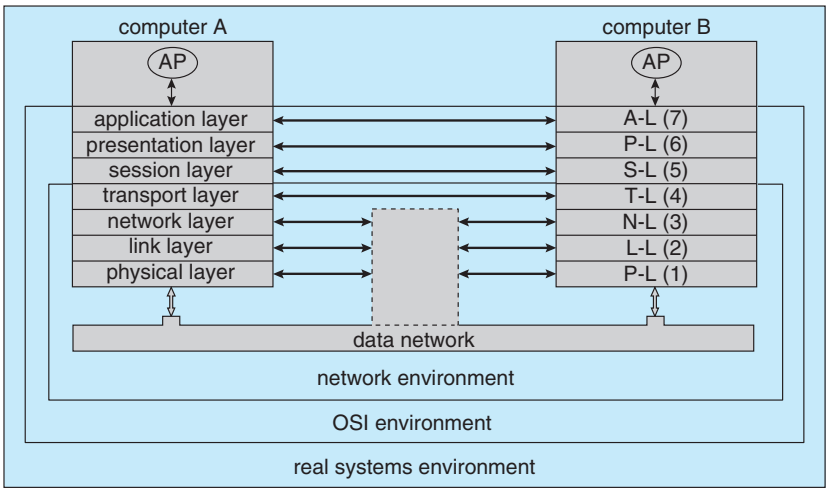

Figure 19.5 Two computers communicating via the OSI network model.

properly as binary data. This layer is implemented in the hardware of the networking device. It is responsible for delivering bits.

- **Layer 2: Data-link layer**. The data-link layer is responsible for handling *frames,* or fixed-length parts of packets, including any error detection and recovery that occur in the physical layer. It sends frames between physical addresses.
- **Layer 3: Network layer**. The network layer is responsible for breaking messages into packets, providing connections between logical addresses, and routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
- **Layer 4: Transport layer**. The transport layer is responsible for transfer of messages between nodes, maintaining packet order, and controlling flow to avoid congestion.
- **Layer 5: Session layer**. The session layer is responsible for implementing sessions, or process-to-process communication protocols.
- **Layer 6: Presentation layer**. The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex–full duplex modes (character echoing).
- **Layer 7: Application layer**. The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 19.6 summarizes the **OSI protocol stack**—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred as one or more packets (Figure 19.7). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack. It is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The OSI model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model (sometimes called the *Internet model*), which has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than the OSI model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than OSI networking. The relationship between the OSI and TCP/IP models is shown in Figure 19.8.

The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, SSH, DNS, and SMTP. The transport layer

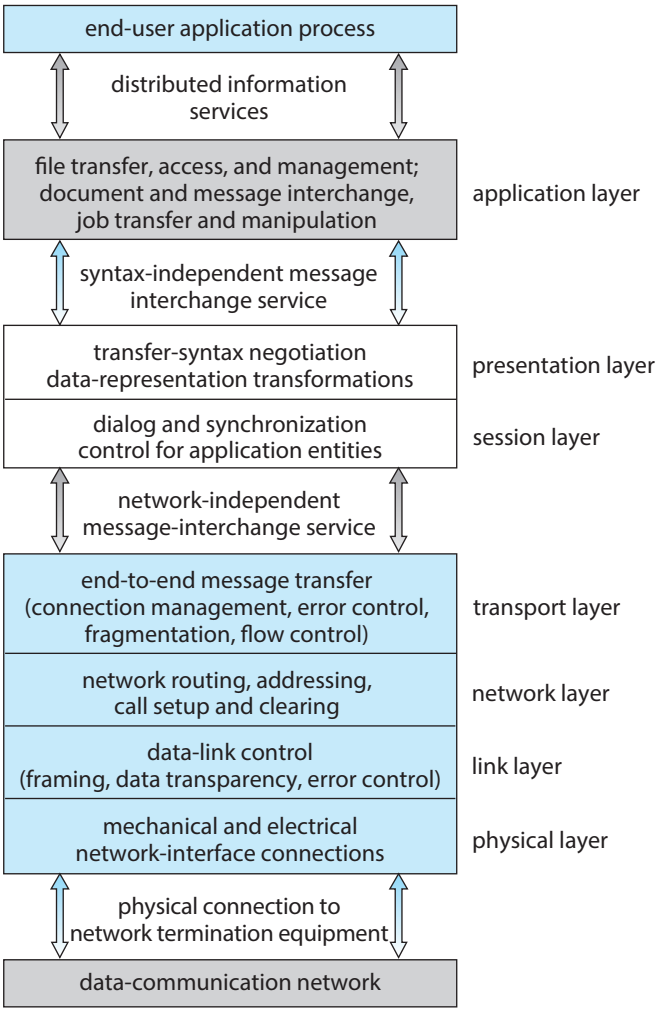

Figure 19.6 The OSI protocol stack.

identifies the unreliable, connectionless **user datagram protocol** (UDP) and the reliable, connection-oriented **transmission control protocol** (TCP). The **Internet protocol** (IP) is responsible for routing IP **datagrams**, or packets, through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section 19.3.3, we consider the TCP/IP model running over an Ethernet network.

Security should be a concern in the design and implementation of any modern communication protocol. Both strong authentication and encryption are needed for secure communication. Strong authentication ensures that the sender and receiver of a communication are who or what they are supposed to be. Encryption protects the contents of the communication from eavesdropping. Weak authentication and clear-text communication are still very common, however, for a variety of reasons. When most of the common protocols were designed, security was frequently less important than performance, sim-

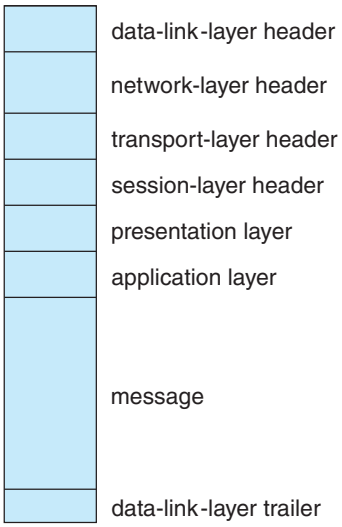

Figure 19.7 An OSI network message.

plicity, and efficiency. This legacy is still showing itself today, as adding security to existing infrastructure is proving to be difficult and complex.

Strong authentication requires a multistep handshake protocol or authentication devices, adding complexity to a protocol. As to the encryption requirement, modern CPUs can efficiently perform encryption, frequently including cryptographic acceleration instructions so system performance is not compromised. Long-distance communication can be made secure by authenticating

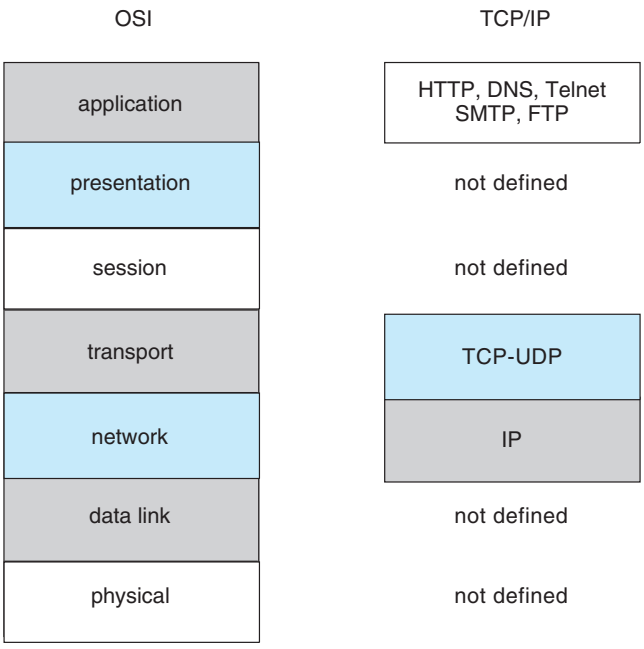

Figure 19.8 The OSI and TCP/IP protocol stacks.

the endpoints and encrypting the stream of packets in a virtual private network, as discussed in Section 16.4.2. LAN communication remains unencrypted at most sites, but protocols such as NFS Version 4, which includes strong native authentication and encryption, should help improve even LAN security.

19.3.3 TCP/IP Example

Next, we address name resolution and examine its operation with respect to the TCP/IP protocol stack on the Internet. Then we consider the processing needed to transfer a packet between hosts on different Ethernet networks. We base our description on the IPV4 protocols, which are the type most commonly used today.

In a TCP/IP network, every host has a name and an associated IP address (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. As described earlier, the name is hierarchical, describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the frame on its way. This routing table is either configured manually by the system administrator or is populated by one of several routing protocols, such as the **Border Gateway Protocol** (**BGP**). The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP (transport) layer for transmission to the destination process.

Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the **medium access control** (**MAC**) **address**, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the networking software generates an **address resolution protocol** (**ARP**) packet containing the IP address of the destination system. This packet is **broadcast** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by routers in between different networks, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP–MAC address pair in an internal table. The cache entries are aged, so that an entry is eventually removed from the cache if an access to that system is not required within a given time. In this way, hosts that are removed from a network are eventually forgotten. For added performance, ARP entries for heavily used hosts may be pinned in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. Networking software takes that name and determines the IP address of the target, using a DNS lookup or an entry in a local `hosts` file

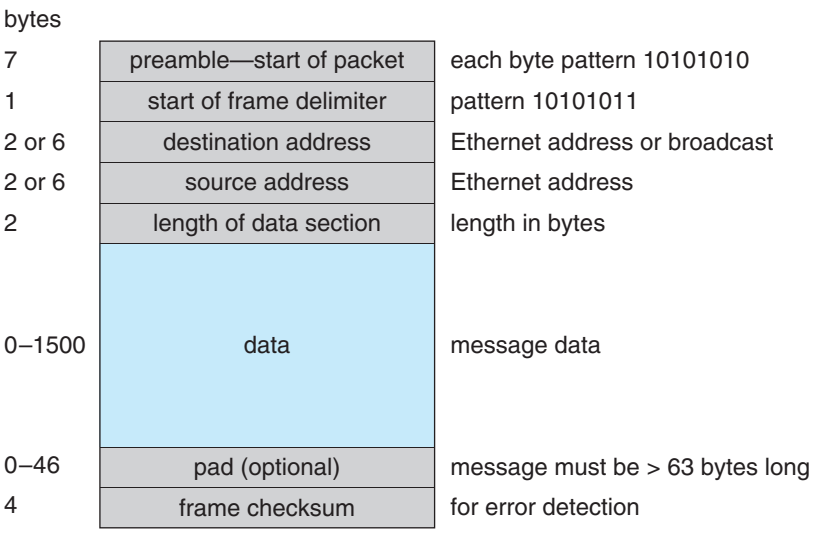

Figure 19.9 An Ethernet packet.

where translations can be manually stored. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet has the Ethernet address at its start; a trailer indicates the end of the packet and contains a **checksum** for detection of packet damage (Figure 19.9). The packet is placed on the network by the Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

19.3.4 Transport Protocols UDP and TCP

Once a host with a specific IP address receives a packet, it must somehow pass it to the correct waiting process. The transport protocols TCP and UDP identify the receiving (and sending) processes through the use of a **port number**. Thus,

a host with a single IP address can have multiple server processes running and waiting for packets as long as each server process specifies a different port number. By default, many common services use *well-known* port numbers. Some examples include FTP (21), SSH (22), SMTP (25), and HTTP (80). For example, if you wish to connect to an "http" website through your web browser, your browser will automatically attempt to connect to port 80 on the server by using the number 80 as the port number in the TCP transport header. For an extensive list of well-known ports, log into your favorite Linux or UNIX machine and take a look at the file `/etc/services`.

The transport layer can accomplish more than just connecting a network packet to a running process. It can also, if desired, add reliability to a network packet stream. To explain how, we next outline some general behavior of the transport protocols UDP and TCP.

19.3.4.1 User Datagram Protocol

The transport protocol UDP is *unreliable* in that it is a bare-bones extension to IP with the addition of a port number. In fact, the UDP header is very simple and contains only four fields: source port number, destination port number, length, and checksum. Packets may be sent quickly to a destination using UDP. However, since there are no guarantees of delivery in the lower layers of the network stack, packets may become lost. Packets can also arrive at the receiver out of order. It is up to the application to figure out these error cases and to adjust (or not adjust).

Figure 19.10 illustrates a common scenario involving loss of a packet between a client and a server using the UDP protocol. Note that this protocol is known as a *connectionless* protocol because there is no connection setup at the beginning of the transmission to set up state—the client just starts sending data. Similarly, there is no connection teardown.

The client begins by sending some sort of request for information to the server. The server then responds by sending four datagrams, or packets, to the client. Unfortunately, one of the packets is dropped by an overwhelmed router. The client must either make do with only three packets or use logic programmed into the application to request the missing packet. Thus, we

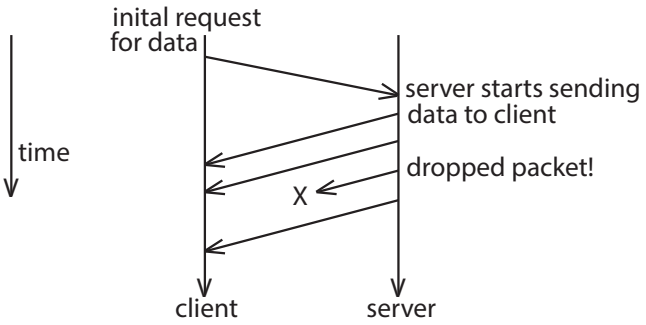

Figure 19.10 Example of a UDP data transfer with dropped packet.

need to use a different transport protocol if we want any additional reliability guarantees to be handled by the network.

19.3.4.2 Transmission Control Protocol

TCP is a transport protocol that is both *reliable* and *connection-oriented*. In addition to specifying port numbers to identify sending and receiving processes on different hosts, TCP provides an abstraction that allows a sending process on one host to send an in-order, uninterrupted *byte stream* across the network to a receiving process on another host. It accomplishes these things through the following mechanisms:

- Whenever a host sends a packet, the receiver must send an **acknowledgment packet**, or ACK, to notify the sender that the packet was received. If the ACK is not received before a timer expires, the sender will send that packet again.
- TCP introduces **sequence numbers** into the TCP header of every packet. These numbers allow the receiver to (1) put packets in order before sending data up to the requesting process and (2) be aware of packets missing from the byte stream.
- TCP connections are initiated with a series of control packets between the sender and the receiver (often called a *three-way handshake*) and closed gracefully with control packets responsible for tearing down the connection. These control packets allow both the sender and the receiver to set up and remove state.

Figure 19.11 demonstrates a possible exchange using TCP (with connection setup and tear-down omitted). After the connection has been established, the client sends a request packet to the server with the sequence number 904. Unlike the server in the UDP example, the server must then send an ACK packet back to the client. Next, the server starts sending its own stream of data packets starting with a different sequence number. The client sends an ACK packet for each data packet it receives. Unfortunately, the data packet with the sequence number 127 is lost, and no ACK packet is sent by the client. The sender times out waiting for the ACK packet, so it must resend data packet 127. Later in the connection, the server sends the data packet with the sequence number 128, but the ACK is lost. Since the server does not receive the ACK it must resend data packet 128. The client then receives a duplicate packet. Because the client knows that it previously received a packet with that sequence number, it throws the duplicate away. However, it must send another ACK back to the server to allow the server to continue.

In the actual TCP specification, an ACK isn't required for each and every packet. Instead, the receiver can send a ***cumulative ACK*** to ACK a series of packets. The server can also send numerous data packets sequentially before waiting for ACKs, to take advantage of network throughput.

TCP also helps regulate the flow of packets through mechanisms called *flow control* and *congestion control*. **Flow control** involves preventing the sender from overrunning the capacity of the receiver. For example, the receiver may

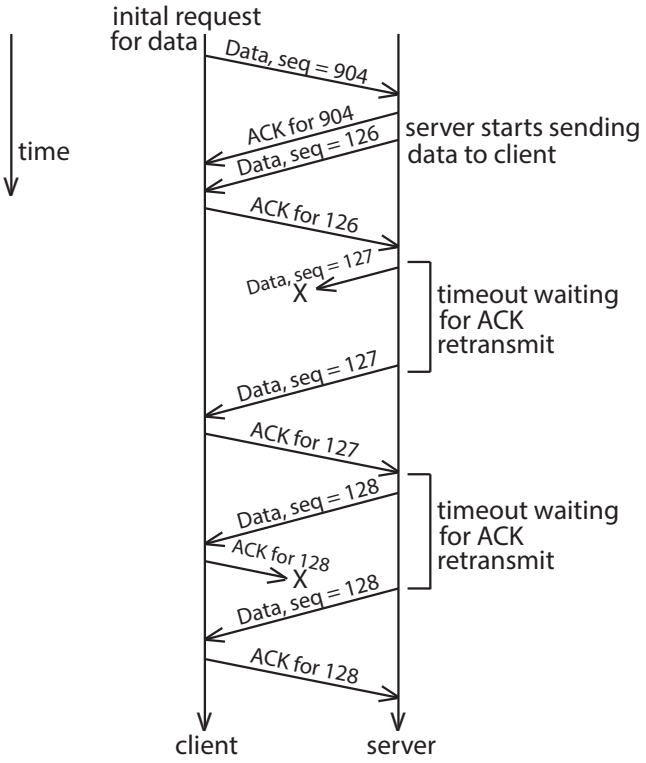

Figure 19.11 Example of a TCP data transfer with dropped packets.

have a slower connection or may have slower hardware components (like a slower network card or processor). Flow-control state can be returned in the ACK packets of the receiver to alert the sender to slow down or speed up. **Congestion control** attempts to approximate the state of the networks (and generally the routers) between the sender and the receiver. If a router becomes overwhelmed with packets, it will tend to drop them. Dropping packets results in ACK timeouts, which results in more packets saturating the network. To prevent this condition, the sender monitors the connection for dropped packets by noticing how many packets are not acknowledged. If there are too many dropped packets, the sender will slow down the rate at which it sends them. This helps ensure that the TCP connection is being fair to other connections happening at the same time.

By utilizing a reliable transport protocol like TCP, a distributed system does not need extra logic to deal with lost or out-of-order packets. However, TCP is slower than UDP.

19.4 Network and Distributed Operating Systems

In this section, we describe the two general categories of network-oriented operating systems: network operating systems and distributed operating sys-

tems. Network operating systems are simpler to implement but generally more difficult for users to access and use than are distributed operating systems, which provide more features.

19.4.1 Network Operating Systems

A **network operating system** provides an environment in which users can access remote resources (implementing resource sharing) by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines. Currently, all general-purpose operating systems, and even embedded operating systems such as Android and iOS, are network operating systems.

19.4.1.1 Remote Login

An important function of a network operating system is to allow users to log in remotely. The Internet provides the `ssh` facility for this purpose. To illustrate, suppose that a user at Westminster College wishes to compute on `kristen.cs.yale.edu`, a computer located at Yale University. To do so, the user must have a valid account on that machine. To log in remotely, the user issues the command

```
ssh kristen.cs.yale.edu
```

This command results in the formation of an encrypted socket connection between the local machine at Westminster College and the `kristen.cs.yale.edu` computer. After this connection has been established, the networking software creates a transparent, bidirectional link so that all characters entered by the user are sent to a process on `kristen.cs.yale.edu` and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

19.4.1.2 Remote File Transfer

Another major function of a network operating system is to provide a mechanism for **remote fil transfer** from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, Kurt at `albion.edu`) wants to access a file owned by Becca located on another computer (say, at `colby.edu`), then the file must be copied explicitly from the computer at Colby in Maine to the computer at Albion in Michigan. The communication is one-directional and individual, such that other users at those sites wishing to transfer a file, say Sean at `colby.edu` to Karen at `albion.edu`, must likewise issue a set of commands.

The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) and the more private secure file transfer protocol (SFTP). Suppose that user Carla at `wesleyan.edu` wants to copy a file that is owned by Owen at `kzoo.edu`. The user must first invoke the `sftp` program by executing

```
sftp owen@kzoo.edu
```

The program then asks the user for a login name and a password. Once the correct information has been received, the user can use a series of commands to upload files, download files, and navigate the remote file system structure. Some of these commands are:

- `get`—Transfer a file from the remote machine to the local machine.
- `put`—Transfer a file from the local machine to the remote machine.
- `ls` or `dir`—List files in the current directory on the remote machine.
- `cd`—Change the current directory on the remote machine.

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

19.4.1.3 Cloud Storage

Basic cloud-based storage applications allow users to transfer files much as with FTP. Users can upload files to a cloud server, download files to the local computer, and share files with other cloud-service users via a web link or other sharing mechanism through a graphical interface. Common examples include Dropbox and Google Drive.

An important point about SSH, FTP, and cloud-based storage applications is that they require the user to change paradigms. FTP, for example, requires the user to know a command set entirely different from the normal operating-system commands. With SSH, the user must know appropriate commands on the remote system. For instance, a user on a Windows machine who connects remotely to a UNIX machine must switch to UNIX commands for the duration of the SSH session. (In networking, a **session** is a complete round of communication, frequently beginning with a login to authenticate and ending with a logoff to terminate the communication.) With cloud-based storage applications, users may have to log into the cloud service (usually through a web browser) or native application and then use a series of graphical commands to upload, download, or share files. Obviously, users would find it more convenient not to be required to use a different set of commands. Distributed operating systems are designed to address this problem.

19.4.2 Distributed Operating Systems

In a distributed operating system, users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system. Depending on the goals of the system, it can implement data migration, computation migration, process migration, or any combination thereof.

19.4.2.1 Data Migration

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to **data migration** is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a

modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) Most modern distributed systems use this approach.

Whichever method is used, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

19.4.2.2 Computation Migration

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this process is called **computation migration**. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses network protocols to execute a routine on a remote system (Section 3.8.2). Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P.

Alternatively, process P can send a message to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q. In fact, it may have several processes running concurrently on several sites.

Either method could be used to access several files (or chunks of files) residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

19.4.2.3 Process Migration

A logical extension of computation migration is **process migration**. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing**. The processes (or subprocesses) may be distributed across the sites to even the workload.
- **Computation speedup**. If a single process can be divided into a number of subprocesses that can run concurrently on different sites or nodes, then the total process turnaround time can be reduced.
- **Hardware preference**. The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on a GPU) than on a microprocessor.
- **Software preference**. The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access**. Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely (say, on a server that hosts a large database) than to transfer all the data and run the process locally.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. The client then need not code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the World Wide Web has many aspects of a distributed computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, Javascript, and similar languages, it provides a form of process migration: Java applets and Javascript scripts are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility —one of the reasons for the huge growth of the World Wide Web.

19.5 Design Issues in Distributed Systems

The designers of a distributed system must take a number of design challenges into account. The system should be robust so that it can withstand failures. The system should also be transparent to users in terms of both file location and user mobility. Finally, the system should be scalable to allow the addition of more computation power, more storage, or more users. We briefly introduce these issues here. In the next section, we put them in context when we describe the designs of specific distributed file systems.

19.5.1 Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, a host, or a site and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when the failure is repaired.

A system can be **fault tolerant** in that it can tolerate a certain level of failure and continue to function normally. The degree of fault tolerance depends on the design of the distributed system and the specific fault. Obviously, more fault tolerance is better.

We use the term ***fault tolerance*** in a broad sense. Communication faults, certain machine failures, storage-device crashes, and decays of storage media should all be tolerated to some extent. A **fault-tolerant system** should continue to function, perhaps in a degraded form, when faced with such failures. The degradation can affect performance, functionality, or both. It should be proportional, however, to the failures that caused it. A system that grinds to a halt when only one of its components fails is certainly not fault tolerant.

Unfortunately, fault tolerance can be difficult and expensive to implement. At the network layer, multiple redundant communication paths and network devices such as switches and routers are needed to avoid a communication failure. A storage failure can cause loss of the operating system, applications, or data. Storage units can include redundant hardware components that automatically take over from each other in case of failure. In addition, RAID systems can ensure continued access to the data even in the event of one or more storage device failures (Section 11.8).

19.5.1.1 Failure Detection

In an environment with no shared memory, we generally cannot differentiate among link failure, site failure, host failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

To detect link and site failure, we use a **heartbeat** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a time-out scheme. At the time A sends the *Are-you-up?*

message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

- Site B is down.
- The direct link (if one exists) from A to B is down.
- The alternative path from A to B is down.
- The message has been lost. (Although the use of a reliable transport protocol such as TCP should eliminate this concern.)

Site A cannot, however, determine which of these events has occurred.

19.5.1.2 Reconfiguratio

Suppose that site A has discovered, through the mechanism just described, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can no longer be reached), then all sites in the system must be notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation in which two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation in which two processes are executing simultaneously in their critical sections.

19.5.1.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the heartbeat procedure described in Section 19.5.1.1.
- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables. For example, it may need routing-table information, a list of sites that are down, undelivered messages, a

transaction log of unexecuted transactions, and mail. If the site has not failed but simply cannot be reached, then it still needs this information.

19.5.2 Transparency

Making the multiple processors and storage devices in a distributed system **transparent** to the users has been a key challenge to many designers. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over a user's environment (for example, home directory) to wherever he logs in. Protocols like LDAP provide an authentication system for local, remote, and mobile users. Once the authentication is complete, facilities like desktop virtualization allow users to see their desktop sessions at remote facilities.

19.5.3 Scalability

Still another issue is **scalability**—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, with respect to a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks' I/O requests overwhelm the I/O subsystem. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design however cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding a network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and allow simple integration of added resources.

Scalability is related to fault tolerance, discussed earlier. A heavily loaded component can become paralyzed and behave like a faulty component. In addition, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. Thus, the multiple resources in a distributed system represent an inherent advantage, giving the system a greater potential for fault tolerance and scalability. However,

inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Scalability can also be related to efficient storage schemes. For example, many cloud storage providers use **compression** or **deduplication** to cut down on the amount of storage used. *Compression* reduces the size of a file. For example, a zip archive file can be generated out of a file (or files) by executing a `zip` command, which runs a lossless compression algorithm over the data specified. (***Lossless compression*** allows original data to be perfectly reconstructed from compressed data.) The result is a file archive that is smaller than the uncompressed file. To restore the file to its original state, a user runs some sort of `unzip` command over the zip archive file. *Deduplication* seeks to lower data storage requirements by removing redundant data. With this technology, only one instance of data is stored across an entire system (even across data owned by multiple users). Both compression and deduplication can be performed at the file level or the block level, and they can be used together. These techniques can be automatically built into a distributed system to compress information without users explicitly issuing commands, thereby saving storage space and possibly cutting down on network communication costs without adding user complexity.

19.6 Distributed File Systems

Although the World Wide Web is the predominant distributed system in use today, it is not the only one. Another important and popular use of distributed computing is the **distributed fil system**, or **DFS**.

To explain the structure of a DFS, we need to define the terms *service, server,* and *client* in the DFS context. A **service** is a software entity running on one or more machines and providing a particular type of function to clients. A **server** is the service software running on a single machine. A **client** is a process that can invoke a service using a set of operations that form its **client interface**. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the **intermachine interface**.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, hard disks or solid-state drives) on which files are stored and from which they are retrieved according to the clients' requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines. In others, a machine can be both a server and a client.

The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system. Ideally, though, a DFS should appear to its clients to be a conventional, centralized file system. That is, the client interface

of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A *transparent* DFS—like the transparent distributed systems mentioned earlier—facilitates user mobility by bringing a user's environment (for example, the user's home directory) to wherever the user logs in.

The most important performance measure of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of storage-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead associated with the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of the DFS's transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

The basic architecture of a DFS depends on its ultimate goals. Two widely used architectural models we discuss here are the **client–server model** and the **cluster-based model**. The main goal of a client–server architecture is to allow transparent file sharing among one or more clients as if the files were stored locally on the individual client machines. The distributed file systems NFS and OpenAFS are prime examples. NFS is the most common UNIX-based DFS. It has several versions, and here we refer to NFS Version 3 unless otherwise noted.

If many applications need to be run in parallel on large data sets with high availability and scalability, the cluster-based model is more appropriate than the client–server model. Two well-known examples are the Google file system and the open-source HDFS, which runs as part of the Hadoop framework.

19.6.1 The Client–Server DFS Model

Figure 19.12 illustrates a simple DFS **client–server model**. The server stores both files and metadata on attached storage. In some systems, more than one server can be used to store different files. Clients are connected to the server through a network and can request access to files in the DFS by contacting the server through a well-known protocol such as NFS Version 3. The server

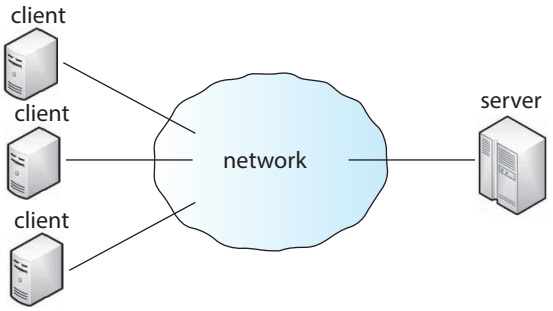

Figure 19.12 Client–server DFS model.

is responsible for carrying out authentication, checking the requested file permissions, and, if warranted, delivering the file to the requesting client. When a client makes changes to the file, the client must somehow deliver those changes to the server (which holds the master copy of the file). The client's and the server's versions of the file should be kept consistent in a way that minimizes network traffic and the server's workload to the extent possible.

The **network file system** (**NFS**) protocol was originally developed by Sun Microsystems as an open protocol, which encouraged early adoption across different architectures and systems. From the beginning, the focus of NFS was simple and fast crash recovery in the face of server failure. To implement this goal, the NFS server was designed to be stateless; it does not keep track of which client is accessing which file or of things such as open file descriptors and file pointers. This means that, whenever a client issues a file operation (say, to read a file), that operation has to be idempotent in the face of server crashes. **Idempotent** describes an operation that can be issued more than once yet return the same result. In the case of a read operation, the client keeps track of the state (such as the file pointer) and can simply reissue the operation if the server has crashed and come back online. You can read more about the NFS implementation in Section 15.8.

The **Andrew fil system** (**OpenAFS**) was created at Carnegie Mellon University with a focus on scalability. Specifically, the researchers wanted to design a protocol that would allow the server to support as many clients as possible. This meant minimizing requests and traffic to the server. When a client requests a file, the file's contents are downloaded from the server and stored on the client's local storage. Updates to the file are sent to the server when the file is closed, and new versions of the file are sent to the client when the file is opened. In comparison, NFS is quite chatty and will send block read and write requests to the server as the file is being used by a client.

Both OpenAFS and NFS are meant to be used in addition to local file systems. In other words, you would not format a hard drive partition with the NFS file system. Instead, on the server, you would format the partition with a local file system of your choosing, such as ext4, and export the shared directories via the DFS. In the client, you would simply attach the exported directories to your file-system tree. In this way, the DFS can be separated from responsibility for the local file system and can concentrate on distributed tasks.

The DFS client–server model, by design, may suffer from a single point of failure if the server crashes. Computer clustering can help resolve this problem by using redundant components and clustering methods such that failures are detected and failing over to working components continues server operations. In addition, the server presents a bottleneck for all requests for both data and metadata, which results in problems of scalability and bandwidth.

19.6.2 The Cluster-Based DFS Model

As the amount of data, I/O workload, and processing expands, so does the need for a DFS to be fault-tolerant and scalable. Large bottlenecks cannot be tolerated, and system component failures must be expected. Cluster-based architecture was developed in part to meet these needs.

Figure 19.13 illustrates a sample cluster-based DFS model. This is the basic model presented by the **Google file system** (**GFS**) and the **Hadoop distributed**

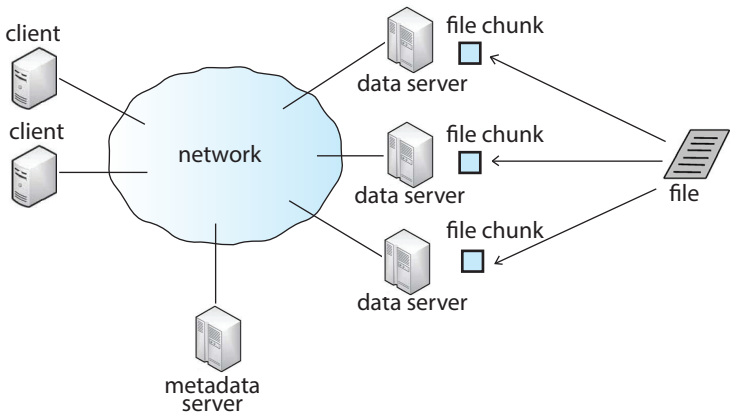

Figure 19.13 An example of a cluster-based DFS model

fil system (**HDFS**). One or more clients are connected via a network to a master metadata server and several data servers that house "chunks" (or portions) of files. The metadata server keeps a mapping of which data servers hold chunks of which files, as well as a traditional hierarchical mapping of directories and files. Each file chunk is stored on a data server and is replicated a certain number of times (for example, three times) to protect against component failure and for faster access to the data (servers containing the replicated chunks have fast access to those chunks).

To obtain access to a file, a client must first contact the metadata server. The metadata server then returns to the client the identities of the data servers that hold the requested file chunks. The client can then contact the closest data server (or servers) to receive the file information. Different chunks of the file can be read or written to in parallel if they are stored on different data servers, and the metadata server may need to be contacted only once in the entire process. This makes the metadata server less likely to be a performance bottleneck. The metadata server is also responsible for redistributing and balancing the file chunks among the data servers.

GFS was released in 2003 to support large distributed data-intensive applications. The design of GFS was influenced by four main observations:

- Hardware component failures are the norm rather than the exception and should be routinely expected.
- Files stored on such a system are very large.
- Most files are changed by appending new data to the end of the file rather than overwriting existing data.
- Redesigning the applications and file system API increases the system's flexibility.

Consistent with the fourth observation, GFS exports its own API and requires applications to be programmed with this API.

Shortly after developing GFS, Google developed a modularized software layer called **MapReduce** to sit on top of GFS. MapReduce allows developers to carry out large-scale parallel computations more easily and utilizes the benefits of the lower-layer file system. Later, HDFS and the Hadoop framework (which includes stackable modules like MapReduce on top of HDFS) were created based on Google's work. Like GFS and MapReduce, Hadoop supports the processing of large data sets in distributed computing environments. As suggested earlier, the drive for such a framework occurred because traditional systems could not scale to the capacity and performance needed by "big data" projects (at least not at reasonable prices). Examples of big data projects include crawling and analyzing social media, customer data, and large amounts of scientific data points for trends.

19.7 DFS Naming and Transparency

Naming is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of **fil replication**. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

19.7.1 Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. **Location transparency**. The name of a file does not reveal any hint of the file's physical storage location.
2. **Location independence**. The name of a file need not be changed when the file's physical storage location changes.

Both definitions relate to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. Some support **fil migration**—that is, changing the location of a file automatically, providing location independence. OpenAFS

supports location independence and file mobility, for example. HDFS includes file migration but does so without following POSIX standards, providing more flexibility in implementation and interface. HDFS keeps track of the location of data but hides this information from clients. This dynamic location transparency allows the underlying mechanism to self-tune. In another example, Amazon's S3 cloud storage facility provides blocks of storage on demand via APIs, placing the storage where it sees fit and moving the data as necessary to meet performance, reliability, and capacity requirements.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.
- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Dropbox and other cloud-based storage solutions work this way. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit is the ability to balance the utilization of storage across the system.
- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be **diskless** and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified).

The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

19.7.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name*, where *local-name* is a UNIX-like path. The Internet URL system also uses this approach. This naming scheme is neither location transparent nor location independent. The DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. Component units remain isolated, although means are provided to refer to remote files. We do not consider this scheme any further here.

The second approach was popularized by NFS. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. The advent of the **automount** feature allowed mounts to be done on demand based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, but this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using a third approach. Here, a single global name structure spans all the files in the system. OpenAFS provides a single global namespace for the files and directories it exports, allowing a similar user experience across different client machines. Ideally, the composed file-system structure is the same as the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their administrative complexity. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere on the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

19.7.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time. Hence, replicating the mapping makes a simple yet consistent update of this information impossible. To overcome this obstacle, we can introduce low-level, ***location-independent file identifiers***. (OpenAFS uses this approach.) Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component-unit migration. The only aspect that does change is the component-unit location mapping.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still in use, by adding sufficiently more bits (this method is used in OpenAFS), or by using a timestamp as one part of the name (as was done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

19.8 Remote File Access

Next, let's consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a **remote-service mechanism**, whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the RPC paradigm, which we discussed in Chapter 3. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

19.8.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to

the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, the least-recently-used algorithm) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the **cache-consistency problem**, which we discuss in Section 19.8.4. DFS caching could just as easily be called **network virtual memory**. It acts similarly to demand-paged virtual memory, except that the backing store usually is a remote server rather than a local disk. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, though with a resulting performance penalty.

The granularity of the cached data in a DFS can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 14.6.2). OpenAFS caches files in large chunks (64 KB). The other systems discussed here support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

19.8.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: they are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
- Technology is moving toward larger and less expensive memory. The resulting performance speedup is predicted to outweigh the advantages of disk caches.

- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Thus, to evaluate the two methods, we must evaluate the degree to which either method is emphasized. The NFS protocol and most implementations do not provide disk caching (but OpenAFS does).

19.8.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a **write-through policy** is reliability: little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the **delayed-write policy**, also known as **write-back caching**, where we delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans its local cache. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

Yet another variation on delayed write is to write data back to the server when the file is closed. This **write-on-close policy** is used in OpenAFS. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through,

which reduces the performance advantages of delayed writes. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

19.8.4 Consistency

A client machine is sometimes faced with the problem of deciding whether a locally cached copy of data is consistent with the master copy (and hence can be used). If the client machine determines that its cached data are out of date, it must cache an up-to-date copy of the data before allowing further accesses. There are two approaches to verifying the validity of cached data:

1. **Client-initiated approach**. The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.
2. **Server-initiated approach**. The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 15.7) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that a file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

In a cluster-based DFS, the cache-consistency issue is made more complicated by the presence of a metadata server and several replicated file data chunks across several data servers. Using our earlier examples of HDFS and GFS, we can compare some differences. HDFS allows append-only write operations (no random writes) and a single file writer, while GFS does allow random writes with concurrent writers. This greatly complicates write consistency guarantees for GFS while simplifying them for HDFS.

19.9 Final Thoughts on Distributed File Systems

The line between DFS client–server and cluster-based architectures is blurring. The NFS Version 4.1 specification includes a protocol for a parallel version of NFS called pNFS, but as of this writing, adoption is slow.

GFS, HDFS, and other large-scale DFSs export a non-POSIX API, so they cannot transparently map directories to regular user machines as NFS and OpenAFS do. Rather, for systems to access these DFSs, they need client code installed. However, other software layers are rapidly being developed to allow NFS to be mounted on top of such DFSs. This is attractive, as it would take advantage of the scalability and other advantages of cluster-based DFSs while still allowing native operating-system utilities and users to access files directly on the DFS.

As of this writing, the open-source HDFS NFS Gateway supports NFS Version 3 and works as a proxy between HDFS and the NFS server software. Since HDFS currently does not support random writes, the HDFS NFS Gateway also does not support this capability. That means a file must be deleted and recreated from scratch even if only one byte is changed. Commercial organizations and researchers are addressing this problem and building stackable frameworks that allow stacking of a DFS, parallel computing modules (such as MapReduce), distributed databases, and exported file volumes through NFS.

One other type of file system, less complex than a cluster-based DFS but more complex than a client–server DFS, is a **clustered file system** (**CFS**) or **parallel file system** (**PFS**). A CFS typically runs over a LAN. These systems are important and widely used and thus deserve mention here, though we do not cover them in detail. Common CFSs include **Lustre** and **GPFS**, although there are many others. A CFS essentially treats N systems storing data and Y systems accessing that data as a single client–server instance. Whereas NFS, for example, has per-server naming, and two separate NFS servers generally provide two different naming schemes, a CFS knits various storage contents on various storage devices on various servers into a uniform, transparent name space. GPFS has its own file-system structure, but Lustre uses existing file systems such as ZFS for file storage and management. To learn more, see http://lustre.org.

Distributed file systems are in common use today, providing file sharing within LANs, within cluster environments, and across WANs. The complexity of implementing such a system should not be underestimated, especially considering that the DFS must be operating-system independent for widespread adoption and must provide availability and good performance in the presence of long distances, commodity hardware failures, sometimes frail networking, and ever-increasing users and workloads.

19.10 Summary

- A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and the Internet. The processors in a distributed system vary in size and function.
- A distributed system provides the user with access to all system resources. Access to a shared resource can be provided by data migration, computation migration, or process migration. The access can be specified by the user or implicitly supplied by the operating system and applications.

- Protocol stacks, as specified by network layering models, add information to a message to ensure that it reaches its destination.
- A naming system (such as DNS) must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance).
- If systems are located on separate networks, routers are needed to pass packets from source network to destination network.
- The transport protocols UDP and TCP direct packets to waiting processes through the use of unique system-wide port numbers. In addition, the TCP protocol allows the flow of packets to become a reliable, connection-oriented byte stream.
- There are many challenges to overcome for a distributed system to work correctly. Issues include naming of nodes and processes in the system, fault tolerance, error recovery, and scalability. Scalability issues include handling increased load, being fault tolerant, and using efficient storage schemes, including the possibility of compression and/or deduplication.
- A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.
- There are two main types of DFS models: the client–server model and the cluster-based model. The client-server model allows transparent file sharing among one or more clients. The cluster-based model distributes the files among one or more data servers and is built for large-scale parallel data processing.
- Ideally, a DFS should look to its clients like a conventional, centralized file system (although it may not conform exactly to traditional file-system interfaces such as POSIX). The multiplicity and dispersion of its servers and storage devices should be transparent. A transparent DFS facilitates client mobility by bringing the client's environment to the site where the client logs in.
- There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. Another approach, popularized by NFS, provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.
- Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and the results are forwarded back to the client. With caching, if the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client. Accesses are performed on the cached copy. The problem of keeping the cached copies consistent with the master file is the cache-consistency problem.

Practice Exercises

19.1 Why would it be a bad idea for routers to pass broadcast packets between networks? What would be the advantages of doing so?

19.2 Discuss the advantages and disadvantages of caching name translations for computers located in remote domains.

19.3 What are two formidable problems that designers must solve to implement a network system that has the quality of transparency?

19.4 To build a robust distributed system, you must know what kinds of failures can occur.

a. List three possible types of failure in a distributed system.

b. Specify which of the entries in your list also are applicable to a centralized system.

19.5 Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is "yes," explain why. If your answer is "no," give appropriate examples.

19.6 A distributed system has two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.

b. The link between A and B goes down.

c. B is extremely overloaded, and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Further Reading

[Peterson and Davie (2012)] and [Kurose and Ross (2017)] provide general overviews of computer networks. The Internet and its protocols are described in [Comer (2000)]. Coverage of TCP/IP can be found in [Fall and Stevens (2011)] and [Stevens (1995)]. UNIX network programming is described thoroughly in [Steven et al. (2003)].

Ethernet and WiFi standards and speeds are evolving quickly. Current IEEE 802.3 Ethernet standards can be found at http://standards.ieee.org/about/get/802/802.3.html. Current IEEE 802.11 Wireless LAN standards can be found at http://standards.ieee.org/about/get/802/802.11.html.

Sun's network file system (NFS) is described by [Callaghan (2000)]. Information about OpenAFS is available from http://www.openafs.org.

Information on the Google file system can be found in [Ghemawat et al. (2003)]. The Google MapReduce method is described in http://research.google.com/archive/mapreduce.html. The Hadoop distributed file system is discussed in [K. Shvachko and Chansler (2010)], and the Hadoop framework is discussed in http://hadoop.apache.org/.

To learn more about Lustre, see http://lustre.org.

Bibliography

[Callaghan (2000)] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[Comer (2000)] D. Comer, *Internetworking with TCP/IP, Volume I,* Fourth Edition, Prentice Hall (2000).

[Fall and Stevens (2011)] K. Fall and R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Second Edition, John Wiley and Sons (2011).

[Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

[K. Shvachko and Chansler (2010)] S. R. K. Shvachko, H. Kuang and R. Chansler, "The Hadoop Distributed File System" (2010).

[Kurose and Ross (2017)] J. Kurose and K. Ross, *Computer Networking—A Top–Down Approach,* Seventh Edition, Addison-Wesley (2017).

[Peterson and Davie (2012)] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approacm,* Fifth Edition, Morgan Kaufmann (2012).

[Steven et al. (2003)] R. Steven, B. Fenner, and A. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API,* Third Edition, John Wiley and Sons (2003).

[Stevens (1995)] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley (1995).

Chapter 19 Exercises

19.7 What is the difference between computation migration and process migration? Which is easier to implement, and why?

19.8 Even though the OSI model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

19.9 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance when the UDP transport protocol is used. What changes could help solve this problem?

19.10 What are the advantages of using dedicated hardware devices for routers? What are the disadvantages of using these devices compared with using general-purpose computers?

19.11 In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

19.12 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

19.13 The lower layers of the OSI network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.

19.14 Run the program shown in Figure 19.4 and determine the IP addresses of the following host names:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

19.15 A DNS name can map to multiple servers, such as www.google.com. However, if we run the program shown in Figure 19.4, we get only one IP address. Modify the program to display all the server IP addresses instead of just one.

19.16 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

19.17 What are the advantages and the disadvantages of making the computer network transparent to the user?

19.18 What are the benefits of a DFS compared with a file system in a centralized system?

19.19 For each of the following workloads, identify whether a cluster-based or a client–server DFS model would handle the workload best. Explain your answers.

- Hosting student files in a university lab.
- Processing data sent by the Hubble telescope.
- Sharing data with multiple devices from a home server.

19.20 Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.

19.21 Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

19.22 What aspects of a distributed system would you select for a system running on a totally reliable network?

19.23 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

19.24 Which scheme would likely result in a greater space saving on a multiuser DFS: file-level deduplication or block-level deduplication? Explain your answer.

19.25 What types of extra metadata information would need to be stored in a DFS that uses deduplication?