especially useful in designing applications that provide navigation—for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in ***augmented-reality*** applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers. We examine these two mobile operating systems in further detail in Chapter 2.

1.10.3 Client–Server Computing

Contemporary network architecture features arrangements in which **server systems** satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server

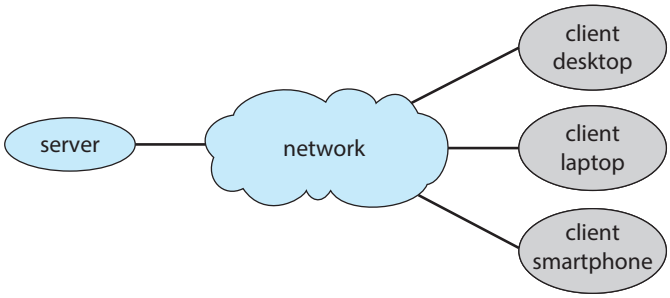

Figure 1.22 General structure of a client–server system.

running a database that responds to client requests for data is an example of such a system.

- The **file-serve system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

1.10.4 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems. In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a ***discovery protocol*** must be provided that allows peers to discover services provided by other peers in the network. Figure 1.23 illustrates such a scenario.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcast file requests to other nodes in the system, and nodes that could service the request responded directly to the client. Peer-to-peer networks can be used to exchange copyrighted materials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement, and its services were shut down in 2001. For this reason, the future of exchanging files remains uncertain.

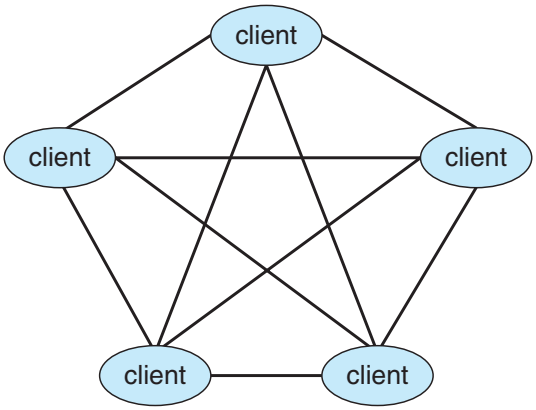

Figure 1.23 Peer-to-peer system with no centralized service.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP** (**VoIP**). Skype uses a hybrid peer-to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

1.10.5 Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service (**SaaS**)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (**PaaS**)—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (**IaaS**)—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

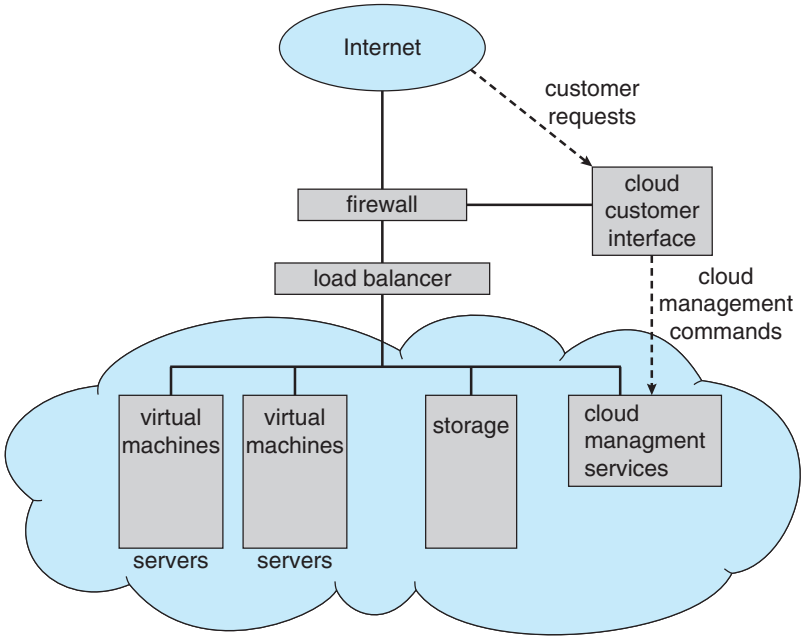

Figure 1.24 Cloud computing.

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as publicly available services.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

1.10.6 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to optical drives and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices

with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator will be able to notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

In Chapter 5, we consider the scheduling facility needed to implement real-time functionality in an operating system, and in Chapter 20 we describe the real-time components of Linux.

1.11 Free and Open-Source Operating Systems

The study of operating systems has been made easier by the availability of a vast number of free software and open-source releases. Both **free operating systems** and **open-source operating systems** are available in source-code format rather than as compiled binary code. Note, though, that free software and open-source software are two different ideas championed by different groups of people (see http://gnu.org/philosophy/open-source-misses-the-point.html/ for a discussion on the topic). Free software (sometimes referred to as ***free/libre software***) not only makes source code available but also is licensed to allow no-cost use, redistribution, and modification. Open-source software does not necessarily offer such licensing. Thus, although all free software is open source, some open-source software is not "free." GNU/Linux is the most famous open-source operating system, with some distributions free and others open source only (http://www.gnu.org/distros/). Microsoft Windows is a well-known example of the opposite **closed-source** approach. Windows is **proprietary** software—Microsoft owns it, restricts its use, and carefully protects its source code. Apple's macOS operating system comprises a hybrid

approach. It contains an open-source kernel named Darwin but includes proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—**reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating-system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to write it, debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

1.11.1 History

In the early days of modern computing (that is, the 1950s), software generally came with source code. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Company-specific user groups, such as Digital Equipment Corporation's DECUS, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members. In 1970, Digital's operating systems were distributed as source code with no restrictions or copyright notice.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Although the Homebrew user groups of the 1970s exchanged code during their meetings, the operating systems for hobbyist machines (such as CPM) were proprietary. By 1980, proprietary software was the usual case.

1.11.2 Free Operating Systems

To counter the move to limit software use and redistribution, Richard Stallman in 1984 started developing a free, UNIX-compatible operating system called GNU(which is a recursive acronym for "GNU's Not Unix!"). To Stallman, "free" refers to freedom of use, not price. The free-software movement does not object

to trading a copy for an amount of money but holds that users are entitled to four certain freedoms: (1) to freely run the program, (2) to study and change the source code, and to give or sell copies either (3) with or (4) without changes. In 1985, Stallman published the GNU Manifesto, which argues that all software should be free. He also formed the **Free Software Foundation** (**FSF**) with the goal of encouraging the use and development of free software.

The FSF uses the copyrights on its programs to implement "copyleft," a form of licensing invented by Stallman. Copylefting a work gives anyone that possesses a copy of the work the four essential freedoms that make the work free, with the condition that redistribution must preserve these freedoms. The **GNU General Public License** (**GPL**) is a common license under which free software is released. Fundamentally, the GPL requires that the source code be distributed with any binaries and that all copies (including modified versions) be released under the same GPL license. The Creative Commons "Attribution Sharealike" license is also a copyleft license; "sharealike" is another way of stating the idea of copyleft.

1.11.3 GNU/Linux

As an example of a free and open-source operating system, consider **GNU/Linux**. By 1991, the GNU operating system was nearly complete. The GNU Project had developed compilers, editors, utilities, libraries, and games — whatever parts it could not find elsewhere. However, the GNU kernel never became ready for prime time. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called "Linux" operating system to grow rapidly, enhanced by several thousand programmers. In 1991, Linux was not free software, as its license permitted only noncommercial redistribution. In 1992, however, Torvalds rereleased Linux under the GPL, making it free software (and also, to use a term coined later, "open source").

The resulting GNU/Linux operating system (with the kernel properly called Linux but the full operating system including GNU tools called GNU/Linux) has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, Red Hat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **live CD**—an operating system that can be booted and run from a CD-ROM without being installed on a system's boot disk. A variant of PCLinuxOS—called PCLinuxOS Supergamer DVD—is a **live DVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

You can run Linux on a Windows (or other) system using the following simple, free approach:

1. Download the free Virtualbox VMM tool from

   https://www.virtualbox.org/

   and install it on your system.
2. Choose to install an operating system from scratch, based on an installation image like a CD, or choose pre-built operating-system images that can be installed and run more quickly from a site like

   http://virtualboxes.org/images/

   These images are preinstalled with operating systems and applications and include many flavors of GNU/Linux.
3. Boot the virtual machine within Virtualbox.

An alternative to using Virtualbox is to use the free program Qemu (http://wiki.qemu.org/Download/), which includes the `qemu-img` command for converting Virtualbox images to Qemu images to easily import them.

With this text, we provide a virtual machine image of GNU/Linux running the Ubuntu release. This image contains the GNU/Linux source code as well as tools for software development. We cover examples involving the GNU/Linux image throughout this text, as well as in a detailed case study in Chapter 20.

1.11.4 BSD UNIX

BSD UNIX has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within Virtualbox, as described above for Linux. The source code comes with the distribution and is stored in `/usr/src/`. The kernel source code is in `/usr/src/sys`. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in `/usr/src/sys/vm`. Alternatively, you can simply view the source code online at https://svnweb.freebsd.org.

As with many open-source projects, this source code is contained in and controlled by a **version control system**—in this case, "subversion" (https://subversion.apache.org/source-code). Version control systems allow a user to "pull" an entire source code tree to his computer and "push" any changes back into the repository for others to then pull. These systems also provide other features, including an entire history of each file and a conflict resolution feature in case the same file is changed concurrently. Another

version control system is **git**, which is used for GNU/Linux, as well as other programs (http://www.git-scm.com).

Darwin, the core kernel component of macOS, is based on BSD UNIX and is open-sourced as well. That source code is available from http://www.opensource.apple.com/. Every macOS release has its open-source components posted at that site. The name of the package that contains the kernel begins with "xnu." Apple also provides extensive developer tools, documentation, and support at http://developer.apple.com.

THE STUDY OF OPERATING SYSTEMS

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/.

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (http://www.vmware.com) provides a free "player" for Windows on which hundreds of free "virtual appliances" can run. Virtualbox (http://www.virtualbox.com) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

In some cases, simulators of specific hardware are also available, allowing the operating system to run on "native" hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on macOS can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system, as well as the original manuals.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Not so many years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

1.11.5 Solaris

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

Several groups interested in using OpenSolaris have expanded its features, and their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at http://wiki.illumos.org.

1.11.6 Open-Source Systems as Learning Tools

The free-software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

Another advantage of working with open-source operating systems is their diversity. GNU/Linux and BSD UNIX are both open-source operating systems, for instance, but each has its own goals, utility, licensing, and purpose. Sometimes, licenses are not mutually exclusive and cross-pollination occurs, allowing rapid improvements in operating-system projects. For example, several major components of OpenSolaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.12 Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run.
- Interrupts are a key way in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the interrupt handler.
- For a computer to do its job of executing programs, the programs must be in main memory, which is the only large storage area that the processor can access directly.
- The main memory is usually a volatile storage device that loses its contents when power is turned off or lost.

to specific processes. The entire program is included in Figure 2.22 and Figure 2.23.

We begin by describing how to create a new entry in the /proc file system. The following program example (named hello.c and available with the source code for this text) creates a /proc entry named /proc/hello. If a user enters the command

```
cat /proc/hello
```

the infamous Hello World message is returned.

```
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
   size_t count, loff_t *pos)
{
   int rv = 0;
   char buffer[BUFFER_SIZE];
   static int completed = 0;

   if (completed) {
    completed = 0;
    return 0;
   }

   completed = 1;

   rv = sprintf(buffer, "Hello World\n");

   /* copies kernel space buffer to user space usr_buf */
   copy_to_user(usr_buf, buffer, rv);

   return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

Figure 2.23 The /proc file system kernel module, Part 2

In the module entry point proc_init(), we create the new /proc/hello entry using the proc_create() function. This function is passed proc_ops, which contains a reference to a struct file_operations. This struct initial-

izes the `.owner` and `.read` members. The value of `.read` is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read.

Examining this `proc_read()` function, we see that the string `"Hello World\n"` is written to the variable `buffer` where `buffer` exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of `buffer` to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory `buffer` to the variable `usr_buf`, which exists in user space.

Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string `"Hello World\n"`) that is to go into the corresponding `/proc/hello` file.

Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function `remove_proc_entry()`.

IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

   ```
   cat /proc/jiffies
   ```

   Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a `proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of `jiffies` as well as the `HZ` rate. When a user enters the command

   ```
   cat /proc/seconds
   ```

   your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

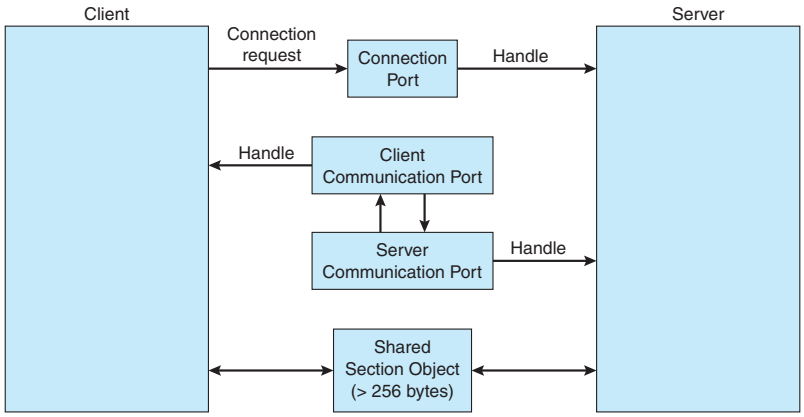

Figure 3.19 Advanced local procedure calls in Windows.

When an ALPC channel is created, one of three message-passing techniques is chosen:

1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

The client has to decide when it sets up the channel whether it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method listed above, but it avoids data copying. The structure of advanced local procedure calls in Windows is shown in Figure 3.19.

It is important to note that the ALPC facility in Windows is not part of the Windows API and hence is not visible to the application programmer. Rather, applications using the Windows API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is handled indirectly through an ALPC procedure call. Additionally, many kernel services use ALPC to communicate with client processes.

3.7.4 Pipes

A **pipe** acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication unidirectional?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
3. Must a relationship (such as ***parent–child***) exist between the communicating processes?
4. Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

In the following sections, we explore two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

3.7.4.1 Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer–consumer fashion: the producer writes to one end of the pipe (the **write end**) and the consumer reads from the other end (the **read end**). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. We next illustrate constructing ordinary pipes on both UNIX and Windows systems. In both program examples, one process writes the message `Greetings` to the pipe, while the other process reads this message from the pipe.

On UNIX systems, ordinary pipes are constructed using the function

```
pipe(int fd[])
```

This function creates a pipe that is accessed through the `int fd[]` file descriptors: `fd[0]` is the read end of the pipe, and `fd[1]` is the write end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary `read()` and `write()` system calls.

An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`. Recall from Section 3.3.1 that a child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process. Figure 3.20 illustrates

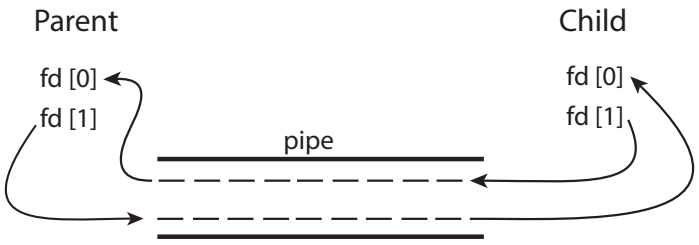

Figure 3.20 File descriptors for an ordinary pipe.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

      /* Program continues in  Figure 3.22 */
```

Figure 3.21 Ordinary pipe in UNIX.

the relationship of the file descriptors in the `fd` array to the parent and child processes. As this illustrates, any writes by the parent to its write end of the pipe—`fd[1]`—can be read by the child from its read end—`fd[0]`—of the pipe.

In the UNIX program shown in Figure 3.21, the parent process creates a pipe and then sends a `fork()` call creating the child process. What occurs after the `fork()` call depends on how the data are to flow through the pipe. In this instance, the parent writes to the pipe, and the child reads from it. It is important to notice that both the parent process and the child process initially close their unused ends of the pipe. Although the program shown in Figure 3.21 does not require this action, it is an important step to ensure that a process reading from the pipe can detect end-of-file (`read()` returns 0) when the writer has closed its end of the pipe.

Ordinary pipes on Windows systems are termed **anonymous pipes**, and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent–child relationships between the communicating processes. In addition, reading and writing to the pipe can be accomplished with the ordinary `ReadFile()` and `WriteFile()` functions. The Windows API for creating pipes is the `CreatePipe()` function, which is passed four parameters. The parameters provide separate handles for (1) reading and (2) writing to the pipe, as well as (3) an instance of the STARTUPINFO structure, which is used to specify that the child process is to inherit the handles of the pipe. Furthermore, (4) the size of the pipe (in bytes) may be specified.

Figure 3.23 illustrates a parent process creating an anonymous pipe for communicating with its child. Unlike UNIX systems, in which a child process automatically inherits a pipe created by its parent, Windows requires the programmer to specify which attributes the child process will inherit. This is

```
    /* create the pipe */
    if (pipe(fd) == -1) {
       fprintf(stderr,"Pipe failed");
       return 1;
    }

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }

    if (pid > 0) { /* parent process */
       /* close the unused end of the pipe */
       close(fd[READ_END]);

       /* write to the pipe */
       write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

       /* close the write end of the pipe */
       close(fd[WRITE_END]);
    }
    else { /* child process */
       /* close the unused end of the pipe */
       close(fd[WRITE_END]);

       /* read from the pipe */
       read(fd[READ_END], read_msg, BUFFER_SIZE);
       printf("read %s",read_msg);

       /* close the read end of the pipe */
       close(fd[READ_END]);
    }

    return 0;
}
```

Figure 3.22 Figure 3.21, continued.

accomplished by first initializing the SECURITY_ATTRIBUTES structure to allow handles to be inherited and then redirecting the child process's handles for standard input or standard output to the read or write handle of the pipe. Since the child will be reading from the pipe, the parent must redirect the child's standard input to the read handle of the pipe. Furthermore, as the pipes are half duplex, it is necessary to prohibit the child from inheriting the write end of the

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
   HANDLE ReadHandle, WriteHandle;
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
   char message[BUFFER_SIZE] = "Greetings";
   DWORD written;

      /* Program continues in  Figure 3.24 */
```

Figure 3.23 Windows anonymous pipe — parent process.

pipe. The program to create the child process is similar to the program in Figure 3.10, except that the fifth parameter is set to TRUE, indicating that the child process is to inherit designated handles from its parent. Before writing to the pipe, the parent first closes its unused read end of the pipe. The child process that reads from the pipe is shown in Figure 3.25. Before reading from the pipe, this program obtains the read handle to the pipe by invoking `GetStdHandle()`.

Note that ordinary pipes require a parent–child relationship between the communicating processes on both UNIX and Windows systems. This means that these pipes can be used only for communication between processes on the same machine.

3.7.4.2 Named Pipes

Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Next, we explore named pipes in each of these systems.

Named pipes are referred to as FIFOs in UNIX systems. Once created, they appear as typical files in the file system. A FIFO is created with the `mkfifo()` system call and manipulated with the ordinary `open()`, `read()`, `write()`, and `close()` system calls. It will continue to exist until it is explicitly deleted

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
  fprintf(stderr, "Create Pipe Failed");
  return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
  fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.24 Figure 3.23, continued.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

Figure 3.25 Windows anonymous pipes — child process.

from the file system. Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If intermachine communication is required, sockets (Section 3.8.1) must be used.

Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts. Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Additionally, only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data. Named pipes are created with the `CreateNamedPipe()` function, and a client can connect to a named pipe using `ConnectNamedPipe()`. Communication over the named pipe can be accomplished using the `ReadFile()` and `WriteFile()` functions.

3.8 Communication in Client–Server Systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems (Section 1.10.3) as well. In this section, we explore two other strategies for communication in client–server systems: sockets and

PIPES IN PRACTICE

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another. For example, the UNIX `ls` command produces a directory listing. For especially long directory listings, the output may scroll through several screens. The command `less` manages output by displaying only one screen of output at a time where the user may use certain keys to move forward or backward in the file. Setting up a pipe between the `ls` and `less` commands (which are running as individual processes) allows the output of `ls` to be delivered as the input to `less`, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the | character. The complete command is

```
ls | less
```

In this scenario, the `ls` command serves as the producer, and its output is consumed by the `less` command.

Windows systems provide a `more` command for the DOS shell with functionality similar to that of its UNIX counterpart `less`. (UNIX systems also provide a `more` command, but in the tongue-in-cheek style common in UNIX, the `less` command in fact provides *more* functionality than `more`!) The DOS shell also uses the | character for establishing a pipe. The only difference is that to get a directory listing, DOS uses the `dir` command rather than `ls`, as shown below:

```
dir | more
```

remote procedure calls (RPCs). As we shall see in our coverage of RPCs, not only are they useful for client–server computing, but Android also uses remote procedures as a form of IPC between processes running on the same system.

3.8.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as SSH, FTP, and HTTP) listen to ***well-known*** ports (an SSH server listens to port 22; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered ***well known*** and are used to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024. For example, if a client on host `X` with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at

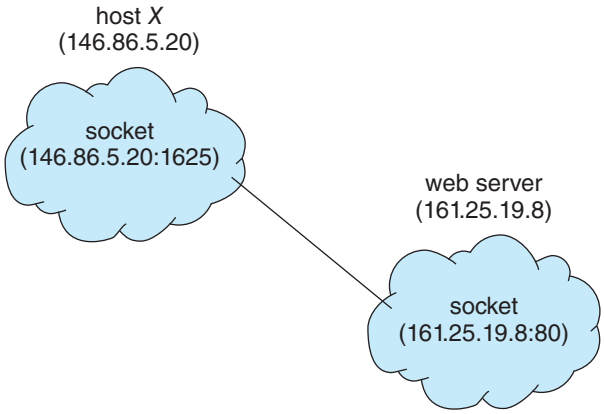

Figure 3.26 Communication using sockets.

address 161.25.19.8, host `X` may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host `X` and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.26. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

All connections must be unique. Therefore, if another process also on host `X` wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

Java provides three different types of sockets. **Connection-oriented** (**TCP**) sockets are implemented with the `Socket` class. **Connectionless** (**UDP**) sockets use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary, unused number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.27. The server creates a `ServerSocket` that specifies that it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.27 Date server.

server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Figure 3.28. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as `www.westminstercollege.edu`, can be used as well.

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.28 Date client.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next subsection, we look a higher-level method of communication: remote procedure calls (RPCs).

3.8.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, which was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.

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

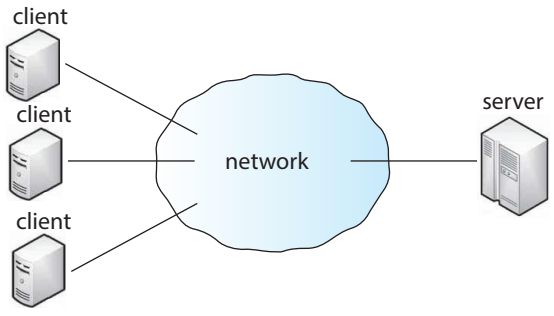

Figure 19.12 Client–server DFS model.

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**

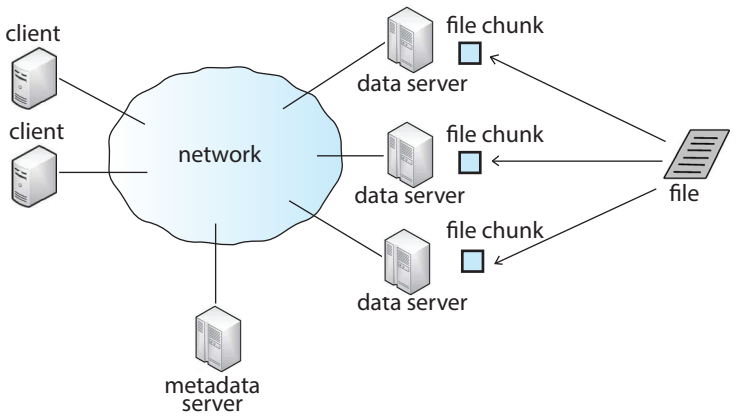

Figure 19.13 An example of a cluster-based DFS model

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.