CHAPTER

Enforcing Modularity with Clients and Services

4

CHAPTER CONTENTS

Principles of Computer System Design: An Introduction

DOI: 10.1016/B978-0-12-374957-4.00013-X

OVERVIEW

The previous chapters established that dividing a system into modules is good and showed how to connect modules using names. If all of the modules were correctly implemented, the job would be finished. In practice, however, programmers make errors, and without extra thought, errors in implementation may too easily propagate from one module to another. To avoid that problem, we need to strengthen the modularity. This chapter introduces a stronger form of modularity, called *enforced modularity*, that helps limit the propagation of errors from one module to another. In this chapter we focus on software modules. In Chapter 8 [on-line] we develop techniques to handle hardware modules.

One way to limit interactions between software modules is to organize systems as clients and services. In the client/service organization, modules interact only by sending messages. This organization has three main benefits:

- Messages are the only way for a programmer to request that a module provide a service. Limiting interactions to messages makes it more difficult for programmers to violate the modularity conventions.
- Messages are the only way for errors to propagate between modules. If clients and services fail independently and if the client and the service check messages, they may be able to limit the propagation of errors.
- Messages are the only way for an attacker to penetrate a module. If clients and services carefully check the messages before they act on them, they can block attacks.

Because of these three benefits, system designers use the client/service organization as a starting point for building modular, fault tolerant, and secure systems.

Designers use the client/service model to separate larger software modules, rather than, say, individual procedures. For example, a database system might be organized as clients that send messages with queries to a service that implements a complete database management system. As another example, an e-mail application might be organized into readers—the clients—that collect e-mail from a service that stores mailboxes.

One effective way to implement the client/service model is to run each client and service module in its own computer and set up a communication path over a wire between the computers. If each module has its own computer, then if one computer (module) fails, the other computer (module) can continue to operate. Since the only communication path is that wire, that is also the only path by which errors can propagate.

Section 4.1 of this chapter shows how the client/service model can enforce modularity between modules. Section 4.2 presents two styles of sending and receiving messages: remote procedure call and publish/subscribe. Section 4.3 summarizes the major issues identified in this chapter but not addressed, and presents a road map for addressing them. Finally, there are detailed case studies of two widely used client/service applications, the Internet Domain Name System and the Network File System.

4.1 CLIENT/SERVICE ORGANIZATION

A standard way to create modularity in a large program is to divide it up into named procedures that call one another. Although the resulting structure can be called modular, implementation errors can propagate from caller to callee and vice versa, and not just through their specified interfaces. For example, if a programmer makes a mistake and introduces an infinite loop in a called procedure and the procedure never returns, then the callee will never receive control again. Or since the caller and callee are in the same address space and use the same stack, either one can accidentally store something in a space allocated to the other. For this reason, we identify this kind of modularity as *soft*. Soft modularity limits interactions of correctly implemented modules to their specified interfaces, but implementation errors can cause interactions that go outside the specified interfaces.

To enforce modularity, we need hard boundaries between modules so that errors cannot easily propagate from one module to another. Just as buildings have firewalls to contain fires within one section of the building and keep them from propagating to other sections, so we need an organization that limits the interaction between modules to their defined interfaces.

This section introduces the client/service organization as one approach to structuring systems that limit the interfaces through which errors can propagate to the specified messages. This organization has two benefits: first, errors can propagate only with messages. Second, clients can check for certain errors by just considering the messages. Although this approach doesn't limit the propagation of all errors, it provides a ***sweeping simplification*** in terms of reasoning about the interactions between modules.

4.1.1 From Soft Modularity to Enforced Modularity

As a more concrete example of how modules interact, suppose we are writing a simple program that measures how long a function runs. We might want to split it into two modules: (1) one system module that provides an interface to obtain the time in units specified by the caller and (2) one application module that measures the running time of a function by asking for time from the clock device, running the function, and requesting the time from the clock device after the function completes. The purpose of this split is to separate the measurement program from the details of the clock device:

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```

The procedure MEASURE takes a function *func* as argument and measures its running time. The procedure GET_TIME returns the time measured in the units specified by the caller. We may desire this clear separation in modules because, for example, we don't want every function that needs the time to know the physical address of the clock (CLOCK in line *2* of GETTIME) in all application programs, such as MEASURE, that use the clock. On one computer, the clock is at physical address $17E5_{hex}$, but on the next computer it is at $24FFF2_{hex}$. Or some clocks return microseconds, and others return sixtieths of a second. By putting the clock's specific properties into GET_TIME, the callers of GET_TIME do not have to be changed when a program is moved to another computer; only GET_TIME must be changed.

This boundary between GET_TIME and its caller, is soft, however. Although procedure call is a primary tool for modularity, errors can still leak too easily from one module to another. It is obvious that if GET_TIME returns a wrong answer, the caller has a problem. It is less obvious that programming errors in GET_TIME can cause trouble for the caller even if GET_TIME returns a correct answer. This section explains why procedure call allows propagation of a wide variety of errors and will introduce an alternative that resembles procedure call but that more strongly limits propagation of errors.

To see why procedure calls allow propagation of many kinds of errors, one must look at the detail of how procedure calls work and at the processor instructions that implement procedure calls. There are many ways to compile the procedures and the call from MEASURE to GET_TIME into processor instructions. For concreteness we pick one procedure call convention. Others differ in the details but exhibit the same issues that we want to explore.

We implement the call to GET_TIME with a stack, so that GET_TIME could call other procedures (although in this example it does not do so). In general, a called procedure may call another procedure or even call itself recursively. To allow for calls to other procedures, the implementation must adhere to the *stack discipline*: each invocation of a procedure must leave the stack as it found it.

To adhere to this discipline, there must be a convention for who saves what registers, who puts the arguments on the stack, who removes them, and who allocates space on the stack for temporary variables. The particular convention used by a system is called the *procedure calling convention*. We use the convention shown in Figure 4.1. Each procedure call results in a new stack frame, which has space for saved registers, the arguments for the callee, the address where the callee should return, and local variables of the callee.

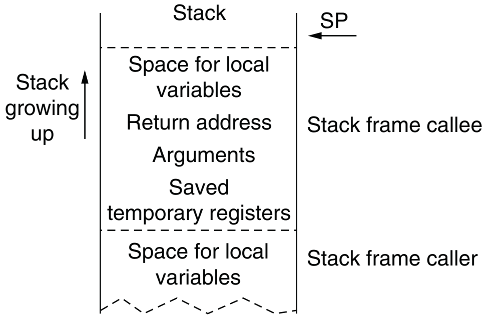


FIGURE 4.1

Procedure call convention.

Given this calling convention, the processor instructions for these two modules are shown in Figure 4.2. In this example, the instructions of the caller (MEASURE) start at address 100, the instructions of the callee (GET_TIME) start at address 200. The stack grows up, from a low address to a high address. The return value of a procedure is passed through register R0. For simplicity, assume that instructions, memory locations, and addresses are all 4 bytes wide. For our example, MEASURE invokes GETTIME as follows:

1. The caller saves content of temporary registers (R1 and R2) at addresses 100 through 112.
2. The caller stores the arguments on the stack (address 116 through 124) so that the callee can find them. (GET_TIME takes one argument: *unit*.)
3. The caller stores a return address on the stack (address 128 through 136) so that the callee can know where the caller should resume execution. (The return address is 148.)

Machine code for MEASURE:

```
100: STORE R1, SP            // save content of R1
104: ADD 4, SP               // adjust stack
108: STORE R2, SP            // save content of R2
112: ADD 4, SP               // adjust stack
116: MOV SECONDS, R1         // move argument to GET_TIME in R1
120: STORE R1, SP            // store argument in R1 on stack
124: ADD 4, SP               // adjust stack
128: MOV 148, R1             // place return address in R1
132: STORE R1, SP            // store return address in R1 on stack
136: ADD 4, SP               // adjust stack
140: STORE 200, R1           // load address of GET_TIME into R1
144: JMP R1                  // jump to it
148: SUB 8, SP               // adjust top of stack
152: MOV SP, R2              // restore R2's content
156: SUB 4, SP               // adjust stack
160: MOV SP, R1              // restore R1's content
164: SUB 4, SP               // adjust stack
168: MOV R0, start           // store result in local stack variable start
172: .....                   // invoke func and GET_TIME again
```

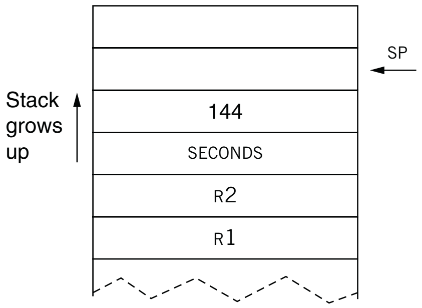
Machine code for GET_TIME:

```
200: MOV SP, R1              // move stack pointer into R1
204: SUB 8, R1               // subtract 8 from SP in R1
208: LOAD R1, R2             // load argument from stack into R2
212: ....                    // instructions for body of GET_TIME
220: MOV time, R0            // move return value in R0
224: MOV SP, R1              // move stack pointer in R1
228: SUB 4, R1               // subtract 4 from SP in R1
232: LOAD R1, PC             // load return address from stack into PC
```

FIGURE 4.2

The procedure MEASURE (located at address 100) calls GET_TIME (located at address 200).

4. The caller transfers control to the callee by jumping to the address of its first instruction (address 140 and 144). (The callee, GET_TIME, is located at address 200.) The stack for our example looks now as in the following figure.



5. The callee loads its argument from the stack into R2 (address 200 through 208).
6. The callee computes with the arguments, perhaps calling other functions (address 212).
7. The callee loads the return value of GET_TIME into R0, the register the implementation reserves for returning values (address 220).
8. The callee loads the return address from the stack into PC (address 224 through 232), which causes the caller to resume control at address 148.
9. The caller adjusts the stack (address 148).
10. The caller restores content of R1 and R2 (addresses 152 through 164).

We use the low-level instructions of the processor for the specific example in Figure 4.2 because it exposes the fine print of the contract between the caller and the callee, and shows how errors can propagate. In the MEASURE example, the contract specifies that the callee returns the current time in some agreed-upon representation to the caller. If we look under the covers, however, we see that this functional specification is not the full contract and that the contract doesn't have a good way of limiting the propagation of errors. To uncover the fine print of the contract between modules, we need to inspect how the stack from Figure 4.2 is used to transfer control from one module to another. The contract between caller and callee contains several subtle potential problems:

- By contract, the caller and callee modify only shared arguments and their own variables in the stack. The callee leaves the stack pointer and the stack the way the caller has set it up. If there is a problem in the callee that corrupts the caller's area of the stack, then the caller might later compute incorrect results or fail.

- By contract, the callee returns where the caller told it to. If by mistake the callee returns somewhere else, then the caller probably performs an incorrect computation or loses control completely and fails.

- By contract, the callee stores return values in register `R0`. If by mistake the callee stores the return value somewhere else, then the caller will read whatever value is in register `R0` and probably perform an incorrect computation.
- By contract, the caller saves the values in the temporary registers (`R1`, `R2`, etc.) on the stack before the call to the callee and restores them when it receives control back. If the caller doesn't follow the contract, the callee may have changed the content of the temporary registers when the caller receives control back, and the caller probably performs an incorrect computation.
- Disasters in the callee can have side effects in the caller. For example, if the callee divides by zero and, as a result, terminates, the caller may terminate too. This effect is known colloquially as *fate sharing*.
- If the caller and callee share global variables, then by contract, the caller and callee modify only those global variables that are shared between them. Again, if the caller or callee modifies some other global variable, they (or other modules) might compute incorrectly or fail altogether.

Thus, the procedure call contract provides us with what might be labeled *soft modularity*. If a programmer makes an error or there is an error in the implementation of the procedure call convention, these errors can easily propagate from the callee to the caller. Soft modularity is usually attained through specifications, but nothing forces the interactions among modules to their defined interfaces. If the callee doesn't adhere (intentionally or unintentionally) to the contract, the caller has a serious problem. We have modularity that is not enforced.

There are also other possibilities for propagation of errors. The procedures share the same address space, and, if a defective procedure incorrectly smashes a global variable, even a procedure that did not call the defective one may be affected. Any procedure that doesn't adhere, either intentionally or unintentionally, to the contract may cause trouble for other modules.

Using a constrained and type-safe implementation language such as Java can beef up soft modularity to a certain extent (see Sidebar 4.1) but is insufficient for complete systems. For one, it is uncommon that all modules in a system are implemented in type-safe language. Often some modules of a system are for performance reasons written in a programming language that doesn't enforce modularity, such C, C++, or processor instructions. But even if the whole system is developed in a type-safe language like Java, we have a need for stronger modularity. If any of the Java modules raises an error (because the interpreter raises a type violation, the module allocated more memory than available, the module couldn't open a file, etc.) or has a programming error (e.g., an infinite loop), we would like to ensure that other modules don't immediately fail too. Even if a called procedure doesn't return, we would like to ensure that the caller has a controlled problem.

What we desire in systems is *enforced modularity*: modularity that is enforced by some external mechanism. This external mechanism limits the interaction among modules to the ones we desire. Such a limit on interactions reduces the number of

Sidebar 4.1 Enforcing Modularity with a High-Level Languages A high-level language is helpful in enforcing modularity because its compiler and runtime system perform all stack and register manipulation, presumably accurately and in accordance with the procedure calling convention. Furthermore, if the programming language enforces a restriction that programs write only to memory locations that correspond to variables managed by the language and in accordance with their type, then programs cannot overwrite arbitrary memory locations and, for example, corrupt the stack. That is, a program cannot use the value of a variable of type integer as an address of a memory location and then store to that memory location. Such languages are called *strongly typed* and, if a program cannot avoid the type system in any way, *type safe*. Modern examples of strongly typed languages include Java and C#.

But even with strongly typed languages, modularity through procedure calls doesn't limit the interactions between modules to their defined interfaces. For example, if the callee has a programming error and allocates all of the available memory space, then the caller may be unable to proceed. Also, strongly typed languages allow the programmer to escape the type system of the language to obtain direct access to memory or to processor registers and to exercise system features that the language does not support (e.g., reading and writing memory locations that correspond to the control registers and state of a device). But this access opens a path for the programmer to make mistakes that violate the procedure call contract.

Another concern is that in many computer systems different modules are written in different programming languages, perhaps because an existing, older module is being reused, even though its implementation language does not provide the type-safety features, or because a lower-level language fragment is essential for achieving maximum performance. Even when the caller and callee are written in two different, strongly typed languages, unexpected interactions can occur at their interface because their conventions do not match.

Another source of errors, which in practice seem to occur much less often, is an implementation error in the interpreter of the application (though with increasing complexity of compilers, runtime support systems, and processor designs, this source may yet become significant). The compiler may have a programming error, the runtime support system may have set up the stack incorrectly, the processor or operating system may save and restore registers incorrectly on an interrupt, a memory error causes a LOAD instruction to return an incorrect value, and so on. Although these sources are less likely to occur than programming errors, it is good to contain the resulting errors so that they don't propagate to other modules.

For all these reasons, designers use the client/service organization. Combining the client/service organization with writing a system in a strongly typed language offers additional opportunities for enforcing modularity; see, for example, the design of the Singularity operating system [Suggestions for Further Reading 5.2.3].

opportunities for propagation of errors. It also allows verification that a user uses a module correctly, and it helps prevent an attacker from penetrating the security of a module.

4.1.2 Client/Service Organization

One good way to enforce modularity is to limit the interactions among modules to explicit messages. It is convenient to impose some structure on this organization by identifying participants in a communication as clients or services.

Figure 4.3 shows a common interaction between client and service. The *client* is the module that initiates a request: it builds a message containing all the data necessary for the service to carry out its job and sends it to a service. The *service* is the module that responds: it extracts the arguments from the request message, executes the requested operations, builds a response message, sends the response message back to the client, and waits for the next request. The client extracts the results from the response message. For convenience, the message from the client to the service is called the *request*, and the message is called the *response* or *reply*.

Figure 4.3 shows one common way in which a client and a service interact: a request is always followed by a response. Since a client and a service can interact using many other sequences of messages, designers often represent the interactions using *message timing diagrams* (see Sidebar 4.2). Figure 4.3 is an instance of a simple timing diagram.

Conceptually, the client/service model runs client and services on separate computers, connected by a wire. This implementation also allows client and service to be separated geographically (which can be good because it reduces the risk that both fail owing to a common fault such as a power outage) and restricts all interactions to well-defined messages sent across a wire.

The disadvantage of this implementation is that it requires one computer per module, which may be costly in equipment. It may also have a performance cost because it may take a substantial amount of time to send a message from one computer to another, in particular if the computers are far away geographically. In some cases these disadvantages are unimportant; for cases in which it does matter, Chapter 5 will explain how to implement the client/service model within a single computer using

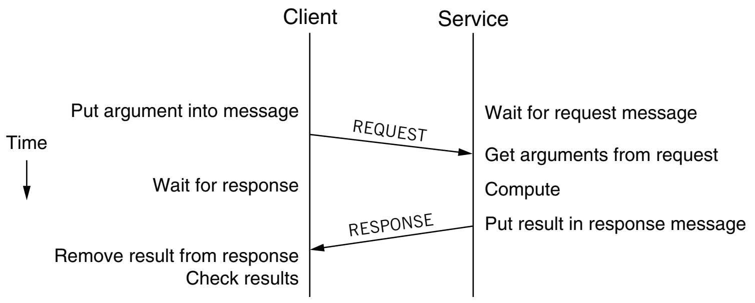

Client
Service
Put argument into message
Wait for request message
REQUEST
Time
Get arguments from request
Wait for response
Compute
RESPONSE
Put result in response message
Remove result from response
Check results

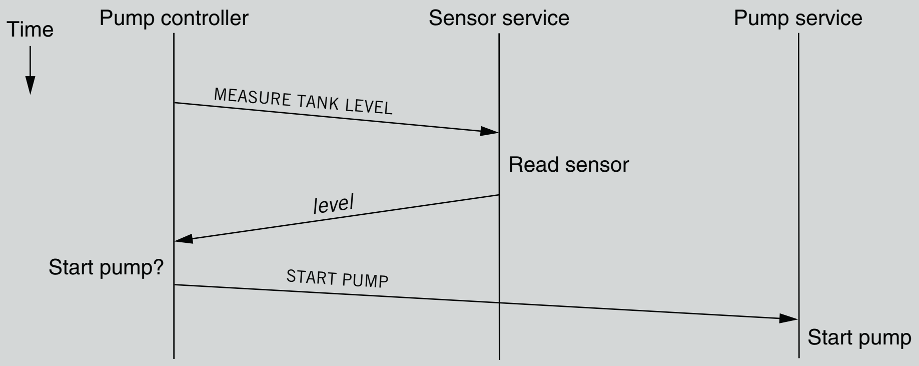
FIGURE 4.3

Communication between client and service.

Sidebar 4.2 Representation: Timing Diagrams A *timing diagram* is a convenient representation of the interaction between modules. When the system is organized in a client/service style, this presentation is particularly convenient, because the interactions between modules are limited to messages. In a timing diagram, the lifetime of a module is represented by a vertical line, with time increasing down the vertical axis. The following example illustrates the use of a timing diagram for a sewage pumping system. The label at the top of a timeline names the module (pump controller, sensor service, and pump service). The physical separation between modules is represented horizontally. Since it takes time for a message to get from one point to another, a message going from the pump controller to the pump service is represented by an arrow that slopes downward to the right.

The modules perform actions, and send and receive messages. The labels next to the time indicate actions taken by the module at a certain time. Modules can take actions at the same time, for example, if they are running on different processors.

The arrows indicate messages. The start of the arrow indicates the time the message is sent by the sending module, and the point of an arrow indicates the time the message is received at the destination module. The content of a message is described by the label associated with the arrow. In some examples, messages can be reordered (arrows cross) or lost (arrows terminate midflight before reaching a module).



The simple timing diagram shown in this sidebar describes the interaction between a pump controller and two services: a sensor service and a pump service. There is a request containing the message "measure tank level" from the client to the sensor service, and a response reports the level read by the sensor. There is a third message, "start pump", which the client sends to the pump service when the level is too high. The second message has no response. The diagram shows three actions: reading the sensor, deciding whether the pump must be started, and starting the pump. Figure 7.7 [on-line] shows a timing diagram with a lost message, and Figure 7.9 [on-line] shows one with a delayed message.

an operating system. For the rest of this chapter we will assume that the client and the service each have their own computer.

To achieve high availability or handle big workloads, a designer may choose to implement a service using multiple computers. For instance, a file service might use several computers to achieve a high degree of fault tolerance; if one computer fails, another one can take over its role. An instance of a service running on a single computer is called a *server*.

To make the client/service model more concrete, let's rearrange our MEASURE program into a simple client/service organization (see Figure 4.4). To get a time from the service, the client procedure builds a request message that names the service and specifies the requested operation and arguments (lines *2* and *6*). The requested operation and arguments must be converted to a representation that is suitable for transmission. For example, the client computer may be a big-endian computer (see Sidebar 4.3), while the service computer may be a little-endian computer. Thus, the client must convert arguments into a canonical representation so that the service can interpret the arguments.

This conversion is called *marshaling*. We use the notation {*a*, *b*} to denote a marshaled message that contains the fields *a* and *b*. Marshaling typically involves converting an object into an array of bytes with enough annotation so that the

```
Client program

1    procedure MEASURE (func)
2        SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
3        response ← RECEIVE_MESSAGE (NameForClient)
4        start ← CONVERT2INTERNAL (response)
5        func ()      // invoke the function
6        SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
7        response ← RECEIVE_MESSAGE (NameForClient)
8        end ← CONVERT2INTERNAL (response)
9        return end – start

Service program

10   procedure TIME_SERVICE ()
11       do forever
12           request ← RECEIVE_MESSAGE (NameForTimeService)
13           opcode ← GET_OPCODE (request)
14           unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
15           if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
16               time ← CONVERT_TO_UNITS (CLOCK, unit)
17               response ← {"OK", CONVERT2EXTERNAL (time)}
18           else
19               response ←{"Bad request"}
20           SEND_MESSAGE (NameForClient, response)
```

FIGURE 4.4

Example client/service application: time service.

Sidebar 4.3 Representation: Big-Endian or Little-Endian? Two common conventions exist for numbering bits within a byte, bytes within a word, words within a page, and the like. One convention is called *big-endian*, and the other *little-endian*.* In big-endian the most significant bit, byte, or word is numbered 0, and the significance of bits *decreases* as the address of the bit increases:

Words	0				1			
Bytes	0	1	…	7	0	1	…	7
Bits	$2^0\,2^1\,2^2\ldots$			$\ldots 2^{63}$	$2^0\,2^1\,2^2\ldots$			$\ldots 2^{63}$

In big-endian the hex number $ABCD_{hex}$ would be stored in memory, so that if you read from memory in increasing memory address order, you would see A-B-C-D. The string "john" would be stored in memory as john.

In little-endian, the other convention, the least significant bit, byte, or word is numbered 0, and the significance of bits *increases* as the address of the bit increases:

Words	n				$n-1$			
Bytes	7	…	1	0	7	…	1	0
Bits	$2^{63}\ldots$			$\ldots 2^2\,2^1\,2^0$	$2^{63}\ldots$			$\ldots 2^2\,2^1\,2^0$

In little-endian, the hex number $ABCD_{hex}$ would be stored in memory, so that if you read from memory in increasing memory address order, you see D-C-B-A. The string "john" would still be stored in memory as john. Thus, code that extracts bytes from character strings transports between architectures, but code that extracts bytes from integers does not transport.

Some processors, such as the Intel x86 family, use the little-endian convention, but others, such as the IBM PowerPC family, use the big-endian convention. As Danny Cohen pointed out in a frequently cited article "*On holy wars and a plea for peace*" [Suggestions for Further Reading 7.2.4], it doesn't matter which convention a designer uses as long as it is the *same* one when communicating between two processors. The processors must agree on the convention for numbering the bits sent over the wire (that is, send the most significant bit first or send the least significant bit first). Thus, if the communication standard is big-endian (as it is in the Internet protocols), then a client running on a little-endian processor must marshal data in big-endian order. This book uses the big-endian convention.

This book also follows the convention that bit numbers start with zero. This choice is independent of the big-endian convention; we could have chosen to use 1 instead, as some processors do.

*The labels "big-endian" and "little-endian" were coined by Jonathan Swift in Chapter 4 of *Gulliver's Travels*, to identify two quarreling factions that differed over which end of an egg it was best to open.

unmarshal procedure can convert it back into a language object. In this example, we show the marshal and unmarshal operations explicitly (e.g., the procedure calls starting with CONVERT), but in many future examples these operations will be implicit to avoid clutter.

After constructing the request, the client sends it (*2* and *6*), waits for a response (line *3* and *7*), and unmarshals the time (*4* and *8*).

The service procedure waits for a request (line *12*) and unmarshals the request (lines *13* and *14*). Then, it checks the request (line *15*), processes it (lines *16* through *19*), and sends back a marshaled response (line *20*).

The *client/service organization* not only separates functions (abstraction), it also enforces that separation (enforced modularity). Compared to modularity using procedure calls, the client/service organization has the following advantages:

- The client and service don't rely on shared state other than the messages. Therefore, errors can propagate from the client to the service, and vice versa, in only one way. If the services (as in line *15*) and the clients check the validity of the request and response messages, then they can control the ways in which errors propagate. Since the client and service don't rely on global, shared data structures such as a stack, a failure in the client cannot directly corrupt data in the service, and vice versa.

- The transaction between a client and a service is an arm's-length transaction. Many errors cannot propagate from one to the other. For instance, the client does not have to trust the service to return to the appropriate return address, as it does using procedure calls. As another example, arguments and results are marshaled and unmarshaled, allowing the client and service to check them.

- The client can protect itself even against a service that fails to return because the client can put an upper limit on the time it waits for a response. As a result, if the service gets into an infinite loop, or fails and forgets about the request, the client can detect that something has gone wrong and undertake some recovery procedure, such as trying a different service. On the other hand, setting timers can create new problems because it can be difficult to predict how long a wait is reasonable. The problem of setting timers for service requests is discussed in detail in Section 7.5.2 [on-line]. In our example, the client isn't defensive against service errors; providing these defenses will make the program slightly more complex but can help eliminate fate sharing.

- Client/Service organization encourages explicit, well-defined interfaces. Because the client and service can interact only through messages, the messages that a service is willing to receive provide a well-defined interface for the service. If those messages are well specified and their specification is public, a programmer can implement a new client or service without having to understand the internals of another client or the service. Clear specification allows clients and service to be implemented by different programmers, and can encourage competition for the best implementation.

Separating state and passing well-defined messages reduce the number of potential interactions, which helps contain errors. If the programmer who developed the service introduces an error and the service has a disaster, the client has only a *controlled* problem. The client's only concern is that the service didn't deliver its part of the contract; apart from this wrong or missing value, the client has no concern for its own integrity. The client is less vulnerable from faults in the service, or, in slightly different words, fate sharing can be reduced. Clients can be mostly independent of service failures, and vice versa.

The client/service organization is an example of a *sweeping simplification* because the model eliminates all forms of interaction other than messages. By separating the client and the service from each other using message passing, we have created a firewall between them. As with firewalls in buildings, if there is a fire in the service, it will be contained in the service, and, assuming the client can check for flames in the response, it will not propagate to the client. If the client and service are well implemented, then the only way to go from the client to the service and back is through well-defined messages.

Of course, the client/service organization is not a panacea. If a service returns an incorrect result, then the client has a problem. This client can check for certain problems (e.g., syntactic ones) but not all semantic errors. The client/service organization reduces fate sharing but doesn't eliminate it. The degree to which the client/service organization reduces fate sharing is also dependent on the interface between the client and service. As an extreme example, if the client/service interface has a message that allows a client to write any value to any address in the service's address space, then it is easy for errors to propagate from the client to the service. It is the job of the system designer to define a good interface between client and service so that errors cannot propagate easily. In this chapter and later chapters, we will see examples of good message interfaces.

For ease of understanding, most of the examples in this chapter exhibit modules consisting of a single procedure. In the real world, designers usually apply the client/service organization between software modules of a larger granularity. The tendency toward larger granularity arises because the procedures within an application typically need to be tightly coupled for some practical reason, such as they all operate on the same shared data structure. Placing every procedure in a separate client or service would make it difficult to manipulate the shared data. The designer thus faces a trade-off between ease of accessing the data that a module needs and ease of error propagation within a module. A designer makes this trade-off by deciding which data and procedures to group into a coherent unit with the data that they manipulate. That coherent unit then becomes a separate service, and errors are contained within the unit. The client and service units are often complete application programs or similarly large subsystems.

Another factor in whether or not to apply the client/service organization to two modules is the plan for recovery when the service module fails. For example, in a simulator program that uses a function to compute the square root of its argument, it makes little sense to put that function into a separate service because it doesn't reduce fate sharing. If the square-root function fails, the simulator program cannot proceed. Furthermore, a good recovery plan is for the programmer to reimplement the function correctly, as opposed to running two square-root servers, and failing over to the second one when the first one fails. In this example, the square-root function might as well be part of the simulator program because the client/service organization doesn't reduce fate sharing for the simulator program and thus there is no reason use it.

A nice example of a widely used system that is organized in a client/service style, with the client and service typically running on separate computers, is the World Wide Web. The Web browser is a client, and a Web site is a service. The browser and the site communicate through well-defined messages and are typically geographically separated. As long as the client and service check the validity of messages, a failure of a service results in a controlled problem for the browser, and vice versa. The World Wide Web provides enforced modularity.

In Figures 4.3 and 4.4, the service always responds with a reply, but that is not a requirement. Figure 4.5 shows the pseudocode for a pump controller for the sewage pumping system in Sidebar 4.2. In this example, there is no need for the pump service to send a reply acknowledging that the pump was turned off. What the client cares about is a confirmation from an independent sensor service that the level in the tank is going down. Waiting for a reply from the pump service, even for a short time, would just delay sounding the alarm if the pump failed.

Client program: pump controller

```
procedure PUMP_CONTROLLER ()
    do forever
        SEND_MESSAGE (NameForSensor, "measure tank level")
        level ← RECEIVE_MESSAGE (NameForClient)
        if level > UpperPumpLimit then SEND_MESSAGE (NameForPump, "turn on pump")
        if level < LowerPumpLimit then SEND_MESSAGE (NameForPump, "turn pump off")
        if level > OverflowLimit then SOUND_ALARM ()
```

Pump service

```
procedure PUMP_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForPump)
        if request = "Turn on pump" then SET_PUMP (on)
        else if request = "Turn off pump" then SET_PUMP (off)
```

Sensor service

```
procedure SENSOR_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForSensor)
        if request = "Measure tank level" then
            response ← READ_SENSOR ()
            SEND_MESSAGE (NameForClient, response)
```

FIGURE 4.5

Example client/service application: controller for a sewage pump.

Sidebar 4.4 The X Window System The X Window System [Suggestions for Further Reading 4.2.2] is the window system of choice on practically every engineering workstation and many personal computers. It provides a good example of using the client/service organization to achieve modularity. One of the main contributions of the X Window System is that it remedied a defect that had crept into the UNIX system when displays replaced typewriters: the display and keyboard were the only hardware-dependent parts of the UNIX application programming interface. The X Window System allowed display-oriented UNIX applications to be completely independent of the underlying hardware.

The X Window System achieved this property by separating the service program that manipulates the display device from the client programs that use the display. The service module provides an interface to manage windows, fonts, mouse cursors, and images. Clients can request services for these resources through high-level operations; for example, clients perform graphics operations in terms of lines, rectangles, curves, and the like. The advantage of this split is that the client programs are device independent. The addition of a new display type may require a new service implementation, but no application changes are required.

Another advantage of a client/service organization is that an application running on one machine can use the display on some other machine. This organization allows, for example, a computing-intensive program to run on a high-performance supercomputer, while displaying the results on a user's personal computer.

It is important that the service be robust to client failures because otherwise a buggy client could cause the entire display to freeze. The X Window system achieves this property by having client and service communicate through carefully designed remote procedure calls, a mechanism described in Section 4.2. The remote procedure calls have the property that the service never has to trust the clients to provide correct data and that the service can process other client requests if it has to wait for a client.

The service allows clients to send multiple requests back to back without waiting for individual responses because the rate at which data can be displayed on a local display is often higher than the network data rate between a client and service. If the client had to wait for a response on each request, then the user-perceived performance would be unacceptable. For example, at 80 characters per request (one line of text on a typical display) and a 5-millisecond round-trip time between client and service, only 16,000 characters per second can be drawn, while typical hardware devices are capable of displaying an order of magnitude faster.

Other systems avoid response messages for performance reasons. For example, the popular X Window System (see Sidebar 4.4) sends a series of requests that ask the service to draw something on a screen and that individually have no need for a response.

4.1.3 Multiple Clients and Services

In the examples so far, we have seen one client and one service, but the client/service model is much more flexible:

- One service can work for multiple clients. A printer service might work for many clients so that the cost of maintaining the printer can be shared. A file service might store files for many clients so that the information in the files can be shared.
- One client can use several services, as in the sewage pump controller (see Figure 4.5), which uses both a pump service and a sensor service.
- A single module can take on the roles of both client and service. A printer service might temporarily store documents on a file service until the printer is ready to print. In this case, the print service functions as a service for printing requests, but it is also a client of the file service.

4.1.4 Trusted Intermediaries

A single service that has multiple clients brings up another technique for enforcing modularity: the *trusted intermediary*, a service that functions as the trusted third party among multiple, perhaps mutually suspicious, clients. The trusted intermediary can control shared resources in a careful manner. For example, a file service might store files for multiple clients, some of which are mutually suspicious; the clients, however, trust the service to keep their affairs distinct. The file service could ensure that a client cannot have access to files not owned by that client, or it could, based on instructions from the clients, allow certain clients to share files.

The trusted intermediary enforces modularity among multiple clients and ensures that a fault in one client has limited (or perhaps no) effect on another client. If the trusted intermediary provides sharing of resources among multiple clients, then it has to be carefully designed and implemented to ensure that the failures of one client don't affect another client. For example, an incorrect update made by one client to its private files shouldn't affect the private files of another client.

A file service is only one example of a trusted intermediary. Many services in client/service applications are trusted intermediaries. E-mail services store mailboxes for many users so that individual users don't have to worry about losing their e-mail. As another example, instant message services provide private buddy lists. It is usually the clients that need some form of controlled sharing, and trusted intermediaries can provide that.

There are also situations in which intermediaries that do not have to be trusted are useful. For example, Section 4.2.3 describes how an untrusted intermediary can be used to buffer and deliver messages to multiple recipients. This use allows communication patterns other than request/response.

Another common use of trusted intermediaries is to simplify clients by having the trusted intermediary provide most functions. The buzzword in trade magazines for

Sidebar 4.5 Peer-to-peer: Computing without Trusted Intermediaries Peer-to-peer is a decentralized design that lacks trusted intermediaries. It is one of the oldest designs and has been used by, for example, the Internet e-mail system, the Internet news bulletin service, Internet service providers to route Internet packets, and IBM's Systems Network Architecture. Recently, it has received much attention in the popular press because file-sharing applications have rediscovered some of its advantages.

In a peer-to-peer application, every computer participating in the application is a peer and is equal in function (but perhaps not in capacity) to any other computer. That is, no peer is more important than any other peer; if one peer fails, then this failure may degrade the performance of the application, but it won't fail the application. The client/service organization doesn't have this property: if the service fails, the application fails, even if all client computers are operational.

UsenetNews is a good example of an older peer-to-peer application. UsenetNews, an on-line news bulletin, is one of the first peer-to-peer applications and has been operational since the 1980s. Users post to a newsgroup, from which other users read articles and respond. Nodes in UsenetNews propagate newsgroups to peers and serve articles to clients. An administrator of a node decides with which nodes the administrator's node peers. Because most nodes interconnect with several other nodes, the system is fault tolerant, and the failure of one node leads at most to a performance degradation rather than to a complete failure. Because the nodes are spread across the world in different jurisdictions, it is difficult for any one central authority to censor content (but an administrator of a node can decide not to carry a group). Because of these properties, designers have proposed organizing other applications in a peer-to-peer style. For example, LOCKSS [Suggestions for Further Reading 10.2.3] has built a robust digital library in that style.

Recently, music-sharing applications and improvements in technology have brought peer-to-peer designs into the spotlight. Today, client computers are as powerful as yesterday's computers for services and are connected with high data-rate links to the Internet. In music-sharing applications the clients are peers, and they serve and store music for one another. This organization aggregates the disk space and network links of all clients to provide a tremendous amount of storage and network capacity, allowing many songs to be stored and served. As often happens in the history of computer systems, the first version of this application was developed not by a computer scientist but by an 18-year-old, Shawn Fanning, who developed Napster. It (and its successors) has changed the characteristics of network traffic on the Internet and has raised legal questions as well.

In Napster, clients serve and store songs, but a trusted intermediary stores the location of a song. Because Napster was used for illegal music sharing, the Recording Industry Association of America (RIAA) sued the operators of the intermediary and was able to shut it down. In more recent peer-to-peer designs, developers adopted the design of censor-resistant applications and avoided the use of a trusted intermediary to locate

(*Sidebar continues*)

songs. In these successors to Napster, the peers locate music by querying other peers; if any individual node is shut down, it will not render the service unavailable. The RIAA must now sue individual users.

Accurately and quickly finding information in a large network of peers without a trusted intermediary is a difficult problem. Without an intermediary there is no central, well-known computer to track the locations of songs. A distributed algorithm is necessary to find a song. A simple algorithm is to send a query for a song to all neighbor peers; if they don't have a copy, the peers forward the query to their neighbors, and so on. This algorithm works, but it is inefficient because it sends a query to every node in the network. To avoid flooding the network of peers on each query, one can stop forwarding the query after it has been forwarded a number of times. Bounding a search in this way may cause some queries to return no answer, even though the song is somewhere in the network.

This problem has sparked interest in the research community, leading to the invention of better algorithms for decentralized search services and resulting in a range of new peer-to-peer applications. Some of these topics are covered in problem sets; see, for example, problem sets *20* [on-line] and *23* [on-line].

this use is "thin-client computing". In this use, only the trusted intermediary must run on a powerful computer (or a collection of computers connected by a high-speed network) because the clients don't run complex functions. Because in most applications there are a few trusted intermediaries (compared to the number of clients), they can be managed by a professional staff and located in a secure machine room. Trusted intermediaries of this type may be expensive to design, build, and run because they may need many resources to support the clients. If one isn't careful, the trusted intermediary can become a choke point during a flash crowd when many clients ask for the service at the same time. At the cost of additional complexity, this problem can be avoided by carefully dividing the work between clients and the trusted intermediary and replicating services using the techniques described in Chapters 8 [on-line] through 10 [on-line].

Designs that have trusted intermediaries also have some general downsides. The trusted intermediary may be vulnerable to failures or attacks that knock out the service. Users must trust the intermediary, but what if it is compromised or is subjected to censorship? Fortunately, there are alternative architectures; see Sidebar 4.5.

4.1.5 A Simple Example Service

Figure 4.6 shows the file system example of Figure 2.18 organized in a client/service style, along with the messages between the clients and services. The editor is a client of the file service, which in turn is a client of the block-storage service. The figure shows the message interaction among these three modules using a message timing diagram.

In the depicted example, the client constructs an OPEN message, specifying the name of the file to be opened. The service checks permissions and, if the user is allowed access, sends a response back indicating success (OK) and the value of the file pointer (0) (see Section 2.3.2 for an explanation of a file pointer). The client writes text to the file, which results in a WRITE request that specifies the text and the number of bytes to be written. The service writes the file by allocating blocks on the block service, copies the specified bytes into them, and returns a message stating the number of bytes written (16). After receiving a response from the block service, it constructs a response for the client, indicating success and informing the client of the new value of the file pointer. When the client is done editing, the client sends a CLOSE message, telling the service that the client is finished with this file.

This message sequence is too simple for use in practice because it doesn't deal with failures (e.g., what happens if the service fails while processing a write request), concurrency (e.g., what happens if multiple clients update a shared file), or security (e.g., how to ensure that a malicious person cannot write the business plan). A file service that is almost this simple is the Woodstock File System (WFS), designed by researchers at the Xerox Palo Alto Research Center [Suggestions for Further Reading 4.2.1]. Section 4.5 is a case study of a widely used successor, the Network File System (NFS), which is organized as a client/service application, and summarizes how NFS handles failures and concurrency. Handling concurrency, failures, and security in general are topics we explore in a systematic way in later chapters.

The file service is a trusted intermediary because it protects the content of files. It must check whether the messages came from a legitimate client (and not from an attacker), it decides whether the client has permission to perform the requested operation, and, if so, it performs the operation. Thus, as long as the file service does its job correctly, clients can share files (and thus also the block-storage service) in a protected manner.

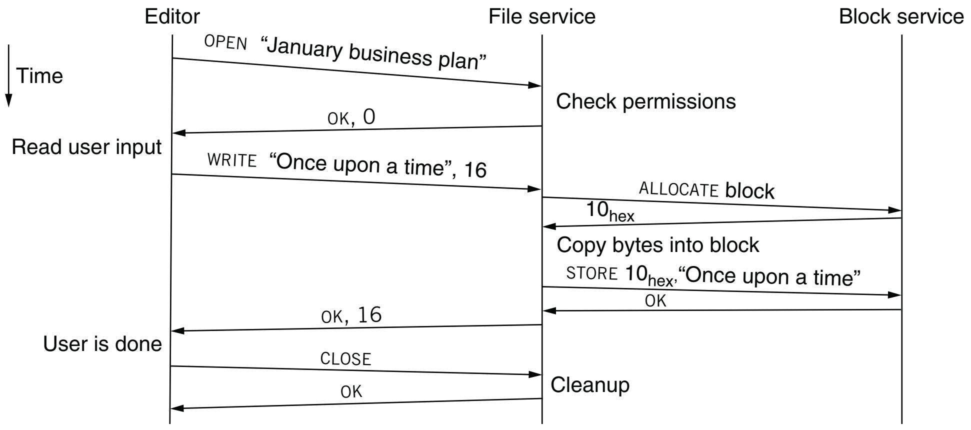


FIGURE 4.6

File service using message timing diagram.

4.2 COMMUNICATION BETWEEN CLIENT AND SERVICE

This section describes two extensions to sending and receiving messages. First, it introduces *remote procedure call (RPC),* a stylized form of client/service interaction in which each request is followed by a response. The goal of RPC systems is to make a remote procedure call look like an ordinary procedure call. Because a service fails independently from a client, however, a remote procedure call can generally not offer identical semantics to procedure calls. As explained in the next subsection, some RPC systems provide various alternative semantics and the programmer must be aware of the details.

Second, in some applications it is desirable to be able to send messages to a recipient that is not on-line and to receive messages from a sender that is not on-line. For example, electronic mail allows users to send e-mail without requiring the recipient to be on-line. Using an intermediary for communication, we can implement these applications.

4.2.1 Remote Procedure Call (RPC)

In many of the examples in the previous section, the client and service interact in a stylized fashion: the client sends a request, and the service replies with a response after processing the client's request. This style is so common that it has received its own name: *remote procedure call*, or RPC for short.

RPCs come in many varieties, adding features to the basic request/response style of interaction. Some RPC systems, for example, simplify the programming of clients and services by hiding many the details of constructing and formatting messages. In the time service example above, the programmer must call SEND_MESSAGE and RECEIVE_MESSAGE, and convert results into numbers, and so on. Similarly, in the file service example, the client and service have to construct messages and convert numbers into bit strings and the like. Programming these conversions is tedious and error prone.

Stubs remove this burden from the programmer (see Figure 4.7). A stub is a procedure that hides the marshaling and communication details from the caller and callee. An RPC system can use stubs as follows. The client module invokes a remote procedure, say GET_TIME, in the same way that it would call any other procedure. However, GET_TIME is actually just the name of a stub procedure that runs inside the client module (see Figure 4.8). The stub marshals the arguments of a call into a message, sends the message, and waits for a response. On arrival of the response, the client stub unmarshals the response and returns to the caller.

Similarly, a service stub waits for a message, unmarshals the arguments, and calls the procedure that the client requests (GET_TIME in the example). After the procedure returns, the service stub marshals the results of the procedure call into a message and sends it in a response to the client stub.

Writing stubs that convert more complex objects into an appropriate on-wire representation becomes quite tedious. Some high-level programming languages

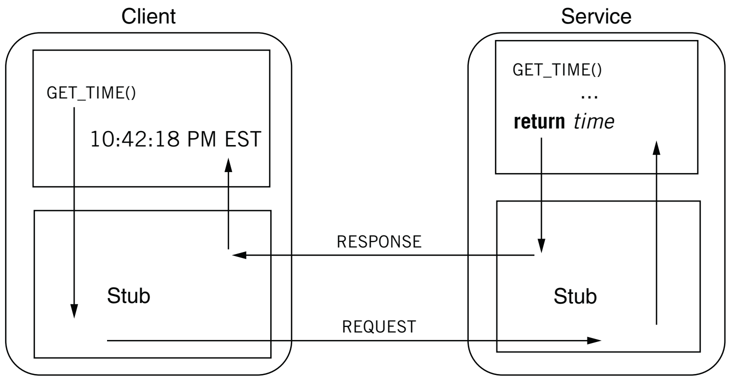


FIGURE 4.7

Implementation of a remote procedure call using stubs. The stubs hide all remote communication from the caller and callee.

```
Client program

procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func ()        // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start

procedure GET_TIME (unit)        // the client stub for GET_TIME
    SEND_MESSAGE (NameForTimeService, {"Get time", unit})
    response ← RECEIVE_MESSAGE (NameForClient)
    return CONVERT2INTERNAL (response)

Service program

procedure TIME_SERVICE ()        // the service stub for GET_TIME
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← GET_ARGUMENT (request)
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            response ← {"ok", GET_TIME (unit)}
        else
            response ← {"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

FIGURE 4.8

GET_TIME client and service using stubs.

such as Java can generate these stubs automatically from an interface specification [Suggestions for Further Reading 4.1.3], simplifying client/service programming even further. Figure 4.9 shows the client for such an RPC system. The RPC system would generate a procedure similar to the GET_TIME stub in Figure 4.8. The client program of Figure 4.9 looks almost identical to the one using a local procedure call on page 149,

The client program

```
1    procedure MEASURE (func)
2        try
3            start ← GET_TIME (SECONDS)
4        catch (signal servicefailed)
5            return servicefailed
6        func ()      // invoke the function
7        try
8            end ← GET_TIME (SECONDS)
9        catch (signal servicefailed)
10           return servicefailed
11       return end – start
```

FIGURE 4.9

GET_TIME client using a system that generates RPC stubs automatically.

except that it handles an additional error because remote procedure calls are not identical to procedure calls (as discussed below).The procedure that the service calls on line 7 is just the original procedure GET_TIME on page 149.

Whether a system uses RPC with automatic stub generation is up to the implementers. For example, some implementations of Sun's Network File System (see Section 4.5) use automatic stub generation, but others do not.

4.2.2 RPCs are not Identical to Procedure Calls

It is tempting to think that by using stubs one can make a remote procedure call behave exactly the same as an ordinary procedure call, so that a programmer doesn't have to think about whether the procedure runs locally or remotely. In fact, this goal was a primary one when RPC was originally proposed—hence the name remote "procedure call". However, RPCs are different from ordinary procedure calls in three important ways: First, RPCs can reduce fate sharing between caller and callee by exposing the failures of the callee to the caller so that the caller can recover. Second, RPCs introduce new failures that don't appear in procedure calls. These two differences change the semantics of remote procedure calls as compared with ordinary procedure calls, and the changes usually require the programmer to make adjustments to the surrounding code.Third, remote procedure calls take more time than procedure calls; the number of instructions to invoke a procedure (see Figure 4.2) is much less than the cost of invoking a stub, marshaling arguments, sending a request over a network, invoking a service stub, unmarshaling arguments, marshaling the response, receiving the response over the network, and unmarshaling the response.

To illustrate the first difference, consider writing a procedure call to the library program SQRT, which computes the square root of its argument x. A careful programmer would plan for the case that SQRT (x) will fail when x is negative by providing an explicit exception handler for that case. However, the programmer using ordinary procedure calls almost certainly doesn't go to the trouble of planning for certain possible failures because they have negligible probability. For example, the programmer probably would not think of setting an interval timer when invoking SQRT (*x*), even though SQRT internally has a successive-approximation loop that, if programmed wrong, might not terminate.

But now consider calling SQRT with an RPC. An interval timer suddenly becomes essential because the network between client and service can lose a message, or the other computer can crash independently. To avoid fate sharing, the RPC programmer must adjust the code to prepare for and handle this failure. When the client receives a "service failure" signal, the client may be able to recover by, for example, trying a different service or choosing an alternative algorithm that doesn't use a remote service.

The second difference between ordinary procedure calls and RPCs is that RPCs introduce a new failure mode, the "no response" failure. When there is no response from a service, the client cannot tell which of two things went wrong: (1) some failure occurred before the service had a chance to perform the requested action, or (2) the service performed the action and then a failure occurred, causing just the response to be lost.

Most RPC designs handle the no-response case by choosing one of three implementation strategies:

- *At-least-once* RPC. If the client stub doesn't receive a response within some specific time, the stub resends the request as many times as necessary until it receives a response from the service. This implementation may cause the service to execute a request more than once. For applications that call SQRT, executing the request more than once is harmless because with the same argument SQRT should always produce the same answer. In programming language terms, the SQRT service has no side effects. Such side-effect-free operations are also ***idempotent***: repeating the same request or sequence of requests several times has the same effect as doing it just once. An at-least-once implementation does not provide the guarantee implied by its name. For example, if the service was located in a building that has been blown away by a hurricane, retrying doesn't help. To handle such cases, an at-least-once RPC implementation will give up after some number of retries. When that happens, the request may have been executed more than once or not at all.

- *At-most-once* RPC. If the client stub doesn't receive a response within some specific time, then the client stub returns an error to the caller, indicating that the service may or may not have processed the request. At-most-once semantics may be more appropriate for requests that do have side effects. For example, in a banking application, using at-least-once semantics for a request to transfer $100 from one account to another could result in multiple $100 transfers. Using at-most-once semantics assures that either zero or one transfers take place, a somewhat more controlled outcome. Implementing at-most-once RPC is harder than it sounds because the underlying network may duplicate the request message without the client stub's knowledge. Chapter 7 [on-line] describes an at-most-once implementation, and Birrell and Nelson's paper gives a nice, complete description of an RPC system that implements at-most-once [Suggestions for Further Reading 4.1.1].

- *Exactly-once* RPC. These semantics are the ideal, but because the client and service are independent it is in principle impossible to guarantee. As in the case of at-least-once, if the service is in a building that was blown away by a hurricane, the best the client stub can do is return error status. On the other hand, by adding the complexity of extra message exchanges and careful record-keeping, one can approach exactly-once semantics closely enough to satisfy some applications. The general idea is that, if the RPC requesting transfer of $100 from account A to B produces a "no response" failure, the client stub sends a separate RPC request to the service to ask about the status of the request that got no response. This solution requires that both the client and the service stubs keep careful records of each remote procedure call request and response. These records must be fault tolerant because the computer running the service might fail and lose its state between the original RPC and the inquiry to check on the RPC's status. Chapters 8 [on-line] through 10 [on-line] introduce the necessary techniques.

The programmer must be aware that RPC semantics differ from those of ordinary procedure calls, and because different RPC systems handle the no-response case in different ways, it is important to understand just which semantics any particular RPC system tries to provide. Even if the name of the implementation implies a guarantee (e.g., at-least-once), we have seen that there are cases in which the implementation cannot deliver it. One cannot simply take a collection of legacy programs and arbitrarily separate the modules with RPC. Some thought and reprogramming is inevitably required. Problem set *2* explores the effects of different RPC semantics in the context of a simple client/service application.

The third difference is that calling a local procedure takes typically much less time than calling a remote procedure call. For example, invoking a remote SQRT is likely to be more expensive than the computation for SQRT itself because the overhead of a remote procedure call is much higher than the overhead of following the procedure calling conventions. To hide the cost of a remote procedure call, a client stub may deploy various performance-enhancing techniques (see Chapter 6), such as caching results and pipelining requests (as is done in the X Window System of Sidebar 4.4). These techniques increase complexity and can introduce new problems (e.g., how to ensure that the cache at the client stays consistent with the one at the service). The performance difference between procedure calls and remote procedure calls requires the designer to consider carefully what procedure calls should be remote ones and which ones should be ordinary, local procedure calls.

A final difference between procedure calls and RPCs is that some programming language features don't combine well with RPC. For example, a procedure that communicates with another procedure through global variables cannot typically be executed remotely because separate computers usually have separate address spaces. Similarly, other language constructs that use explicit addresses won't work. Arguments consisting of data structures that contain pointers, for example, are a problem because pointers to objects in the client computer are local addresses that have different bindings when resolved in the service computer. It is possible to design systems that use global references for objects that are passed by reference to remote procedure calls but require significant additional machinery and introduce new problems. For example, a new plan is needed for determining whether an object can be deleted locally because a remote computer might still have a reference to the object. Solutions exist, however; see, for example, the article on Network Objects [Suggestions for Further Reading 4.1.2].

Since RPCs don't provide the same semantics as procedure calls, the word "procedure" in "remote procedure call" can be misleading. Over the years the concept of RPC has evolved from its original interpretation as an exact simulation of an ordinary procedure call to instead mean any client/service interaction in which the request is followed by a response. This text uses this modern interpretation.

4.2.3 Communicating through an Intermediary

Sending a message from a sender to a receiver requires that both parties be available at the same time. In many applications this requirement is too strict. For example, in electronic mail we desire that a user be able to send an e-mail to a recipient even if the recipient is not on-line at the time. The sender sends the message and the recipient receives the message some time later, perhaps when the sender is not on-line. We can implement such applications using an intermediary. In the case of communication, this intermediary doesn't have to be trusted because communication applications often consider the intermediary to be part of an untrusted network and have a separate plan for securing messages (as we will see in Chapter 11 [on-line]).

The primary purpose of the e-mail intermediary is to implement *buffered* communication. Buffered communication provides the SEND/RECEIVE abstraction but avoids the requirement that the sender and receiver be present simultaneously. It allows the delivery of a message to be shifted in time. The intermediary can hold messages until the recipient comes on-line. The intermediary might buffer messages in volatile memory or in non-volatile memory, such as a file system. The latter design allows the intermediary to buffer messages across power failures.

Once we have an intermediary, three interesting design opportunities arise. First, the sender and receiver may make different choices of whether to *push* or *pull* messages. Push is when the initiator of a data movement sends the data. Pull is when the initiator of a data movement asks the other end to send it the data. These definitions are independent of whether or not the system uses an intermediary, but in systems with intermediaries it is not uncommon to find both in a single system. For example, the sender in the Internet's e-mail system, Simple Mail Transfer Protocol (SMTP), pushes the mail to the service that holds the recipient's mailbox. On the other hand, the receiving client pulls messages to fetch mail from a mailbox: the user hits the

"Get new mail" button, which causes the mail client to contact the mailbox service and ask it for any new mail.

Second, the existence of an intermediary opens an opportunity to apply the design principle ***decouple modules with indirection*** by having the intermediary, rather than the originator, determine to whom a message is delivered. For example, an Internet user can send a message to `president@whitehouse.gov`. The intermediary that forwards the message will deliver it to whoever happens to be the President. As another example, users should be able to send an e-mail to a mailing list or to post a message to a bulletin board without knowing exactly who is on the mailing list or subscribed to the bulletin board.

Third, when indirection through an intermediary is available, the designer has a choice of when and where to duplicate messages. In the mailing list example, the intermediary sends a copy of the e-mail to each member of the list. In the bulletin board example, an intermediary may group messages and send them as a group to other intermediaries. When a user fetches the bulletin article from its local intermediary, the local intermediary makes a final copy for delivery to the user.

Publish/subscribe is a general style of communication that takes advantage of the three design opportunities of communication through an intermediary. In this communication model, the sender is called the publisher and notifies an event service that it has produced a new message on a certain topic. Recipients subscribe to the event service and express their interest in certain topics. If multiple recipients are interested in the same topic, all of them receive a copy of the message. Popular usages of publish/subscribe are electronic mailing lists and instant messaging services that provide chat rooms. A user might join a chat room on a certain topic. When another user publishes a message in the room, all the members of that room receive it. Another publish/subscribe application is Usenet News, a bulletin board service (described in Sidebar 4.5 on peer-to-peer computing).

4.3 SUMMARY AND THE ROAD AHEAD

The client/service model enforces modularity and is the basic approach to organizing complex computer systems. The rest of the book works out major issues in building computer systems that this chapter has identified but has not addressed:

- *Enforcing modularity within a computer* (Chapter 5). Restricting the implementation of client/service systems to one computer per module can be too expensive. Chapter 5 shows how an operating system can use a technique called virtualization to create many virtual computers out of one physical computer. The operating system can enforce modularity between each client and each service by giving each client and each service a separate virtual computer.
- *Performance* (Chapter 6). Computer systems have implicit or explicit performance goals. If services are not carefully designed, it is possible that the slowest

service in the system becomes a performance bottleneck, which causes the complete system to operate at the performance of the slowest service. Identifying performance bottlenecks and avoiding them is a challenge that a designer faces in most computer systems.

- *Networking* (Chapter 7 [on-line]). The client/service model must have a way to send the request message from the client to the service, and the response message back. Implementing SEND_MESSAGE and RECEIVE_MESSAGE is a challenging problem, since networks may lose, reorder, or duplicate messages while routing them between the client and the service. Furthermore, networks exhibit a wide range of performance properties, making straightforward solutions inadequate.
- *Fault tolerance* (Chapter 8 [on-line]). We may need for a service to continue to operate even if some of the hardware and software modules fail. For example, we may want to construct a fault tolerant date-and-time service that runs on several computers so that if one of the computers fails, another computer can still deliver a response to requests for the date and time. In systems that harness a large number of computers to deliver a single service, it is unavoidable that at any instant of time some of the computers will have failed. For example, Google, which indexes the Web, reportedly uses more than 100,000 computers to deliver the service. (A description of the systems Google has designed can be found in Suggestions for Further Reading 3.2.4 and 10.1.10.) With so many computers, some of them are certain to be unavailable. Techniques for fault tolerance allow designers to implement reliable services out of unreliable components. These techniques involve detecting failures, containing them, and recovering from them.
- *Atomicity* (Chapter 9 [on-line]). The file service described in this chapter (Figure 4.6 in Section 4.1.6) must work correctly in the face of concurrent access and failures, and use OPEN and CLOSE calls to mark related READ and WRITE operations. Chapter 9 [on-line] introduces a single framework called atomicity that addresses both issues. This framework allows the operations between an OPEN and CLOSE call to be executed as an atomic, indivisible action. As we saw in Section 4.2.2, exactly-once RPC is ideal for implementing a banking application. Chapter 9 [on-line] introduces the necessary tools for exactly-once RPC and building such applications.
- *Consistency* (Chapter 10 [on-line]). This chapter uses messages to implement various protocols to ensure consistency of data stores on different computers.
- *Security* (Chapter 11 [on-line]). The client/service model protects against accidental errors propagating from one module to another module. Some services may need to protect against malicious attacks. This requirement arises, for example, when a file service is storing sensitive data and needs to ensure that malicious users cannot read the sensitive data. Such protection requires that the service reliably identify users so that it can make an authorization decision. The design of systems in the face of malicious users is a topic known as *security*.

The subsystems that address these topics are interesting systems in their own right and are case studies of managing complexity. Typically, these subsystems are internally structured as client/service systems, applying the concept of this chapter recursively. The next two sections provide two case studies of real-world client/service systems and also illustrate the need for the topics addressed in the subsequent chapters.

4.4 CASE STUDY: THE INTERNET DOMAIN NAME SYSTEM (DNS)

The Internet Domain Name System (DNS) provides an excellent case study of both a client/service application and a successful implementation of a naming scheme, in this case for naming of Internet computers and services. Although designed for that specific application, DNS is actually a general-purpose name management and name resolution system that hierarchically distributes the management of names among different naming authorities and also hierarchically distributes the job of resolving names to different name servers. Its design allows it to respond rapidly to requests for name resolution and to scale up to extremely large numbers of stored records and numbers of requests. It is also quite resilient, in the sense that it provides continued, accurate responses in the face of many kinds of network and server failures.

The primary use for DNS is to associate user-friendly character-string names, called *domain names*, with machine-oriented binary identifiers for network attachment points, called *Internet addresses*. Domain names are hierarchically structured, the term *domain* being used in a general way in DNS: it is simply a set of one or more names that have the same hierarchical ancestor. This convention means that hierarchical regions can be domains, but it also means that the personal computer on your desk is a domain with just one member. In consequence, although the phrase "domain name" suggests the name of a hierarchical region, every name resolved by DNS is called a domain name, whether it is the name of a hierarchical region or the name of a single attachment point. Because domains typically correspond to administrative organizations, they also are the unit of delegation of name assignment, using exactly the hierarchical naming scheme described in Section 3.1.4.

For our purposes, the basic interface to DNS is quite simple:

```
value ← DNS_RESOLVE (domain_name)
```

This interface omits the context argument from the standard name-resolving interface of the naming model of Section 2.2.1 because there is just a single, universal, default context for resolving all Internet domain names, and the reference to that one context is built into DNS_RESOLVE as a configuration parameter.

In the usual DNS implementation, binding is not accomplished by invoking BIND and UNBIND procedures as suggested by our naming model, but rather by using a text editor or database generator to create and manage tables of bindings. These tables are then loaded into DNS servers by some behind-the-scenes method as often as their managers deem necessary. One consequence of this design is that changes to DNS bindings don't often occur within seconds of the time you request them; instead, they typically take hours.

Domain names are path names, with components separated by periods (called *dots,* particularly when reading domain names aloud) and with the least significant component coming first. Three typical domain names are

```
ginger.cse.pedantic.edu    ginger.scholarly.edu    ginger.com
```

DNS allows both relative and absolute path names. Absolute path names are supposed to be distinguished by the presence of a trailing dot. In human interfaces the trailing dot rarely appears; instead, DNS_RESOLVE applies a simple form of multiple lookup. When presented with a relative path name, DNS_RESOLVE first tries appending a default context, supplied by a locally set configuration parameter. If the resulting extended name fails to resolve, DNS_RESOLVE tries again, this time appending just a trailing dot to the originally presented name. Thus, for example, if one presents DNS_RESOLVE with the apparently relative path name "`ginger.com`", and the default context is "`pedantic.edu.`", DNS_RESOLVE will first try to resolve the absolute path name "`ginger.com.pedantic.edu.`". If that attempt leads to a NOT-FOUND result, it will then try to resolve the absolute path name "`ginger.com.`"

4.4.1 Name Resolution in DNS

DNS name resolution might have been designed in at least three ways:

1. *The telephone book model:* Give each network user a copy of a file that contains a list of every domain name and its associated Internet address. This scheme has a severe problem: to cover the entire Internet, the size of the file would be proportional to the number of network users, and updating it would require delivering a new copy to every user. Because the frequency of update tends to be proportional to the number of domain names listed in the file, the volume of network traffic required to keep it up to date would grow with the cube of the number of domain names. This scheme was used for nearly 20 years in the Internet, was found wanting, and was replaced with DNS in the late 1980s.

2. *The central directory service model:* Place the file on a single well-connected server somewhere in the network and provide a protocol to ask it to resolve names. This scheme would make update easy, but with growth in the number of users its designer would have to adopt increasingly complex strategies to keep it from becoming both a performance bottleneck and a potential source of massive failure. There is yet another problem: whoever controls the central server is by default in charge of all name assignment. This design does not cater well to delegation of responsibility in assignment of domain names.

3. *The distributed directory service model.* The idea is to have many servers, each of which is responsible for resolving some subset of domain names, and a protocol for finding a server that can resolve any particular name. As we shall see in the following descriptions, this model can provide delegation and respond

to increases in scale while maintaining reliability and performance. For those reasons, DNS uses this model.

With the distributed directory service model, the operation of every name server is the same: a server maintains a set of name records, each of which binds a domain name to an Internet address. When a client sends a request for a name resolution, the name server looks through the collection of domain names for which it is responsible, and if it finds a name record, it returns that record as its response. If it does not find the requested name, it looks through a separate set of referral records. Each referral record binds a hierarchical region of the DNS name space to some other name server that can help resolve names in that region of the naming hierarchy. Starting with the most significant component of the requested domain name, the server searches through referral records for the one that matches the most components, and it returns that referral record. If nothing matches, DNS cannot resolve the original name, so it returns a "no such domain" response.

The referral architecture of DNS, though conceptually simple, has a number of elaborations that enhance its performance, scalability, and robustness. We begin with an example of its operation in a simple case, and we later add some of the enhancements. The dashed lines in Figure 4.10 illustrate the operation of DNS when the client computer named `ginger.cse.pedantic.edu`, in the lower left corner, tries to resolve the domain name `ginger.Scholarly.edu`. The first step, shown as request #1, is that DNS_RESOLVE sends that domain name to a *root name server*, whose Internet address it somehow knows. Section 4.4.4 explains how DNS_RESOLVE discovers that address.

The root name server matches the name in the request with the subset of domain names it knows about, starting with the most significant component of the requested domain name (in this example, `edu`). In this example, the root name server discovers that it has a referral record for the domain `edu`, so it responds with a referral, saying, in this example, "There is a name server for a domain named `edu`. The name record for that name server binds the name `names.edu.` to Internet address 192.14.71.191." This response illustrates that name servers, like any other servers, have both domain names and Internet addresses. Usually, the domain name of a name server gives some clue about what domain names it serves, but there is no necessary correspondence. Responding with a complete name record provides more information than the client really needs (the client usually doesn't care about the name of the name server), but it allows all responses from a name server to be uniform. Because the name server's domain name isn't significant and to reduce clutter in Figure 4.10, that figure omits it in the illustrated response.

When the client's DNS_RESOLVE receives this response, it immediately resends the same name resolution request, but this time it directs the request (request 2 in the figure) to the name server located at the Internet address mentioned in response number 1. That name server matches the requested path name with the set of domain names it knows about, again starting with the most significant component. In this case, it finds a match for the name `Scholarly.edu.` in a referral record. It thus sends back a response saying, "There is a name server for a domain named `Scholarly.edu`. The

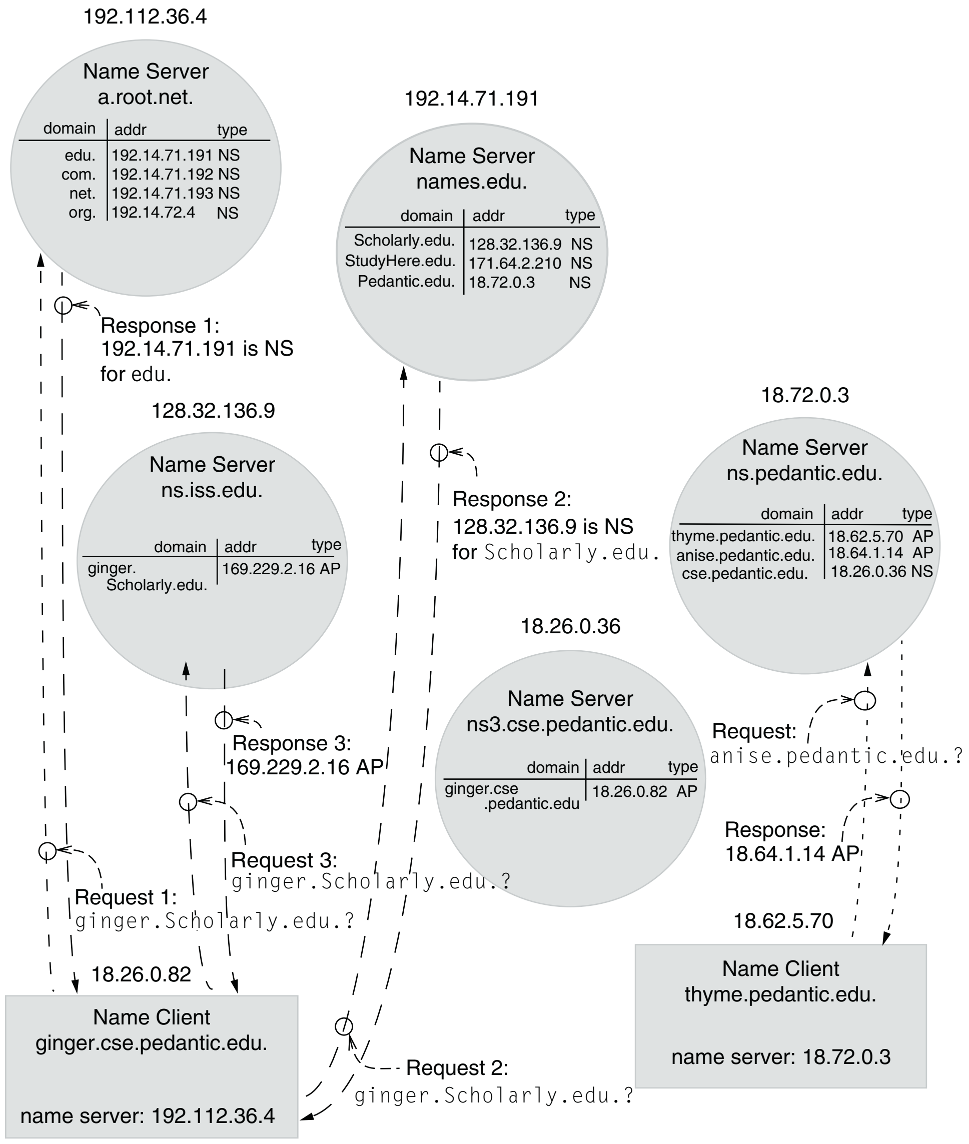

192.112.36.4
Name Server
a.root.net.
domain | addr | type
edu. | 192.14.71.191 NS
com. | 192.14.71.192 NS
net. | 192.14.71.193 NS
org. | 192.14.72.4 NS
192.14.71.191
Name Server
names.edu.
domain | addr | type
Scholarly.edu. | 128.32.136.9 NS
StudyHere.edu. | 171.64.2.210 NS
Pedantic.edu. | 18.72.0.3 NS
Response 1: 192.14.71.191 is NS for edu.
128.32.136.9
Name Server
ns.iss.edu.
domain | addr | type
ginger.Scholarly.edu. | 169.229.2.16 AP
Response 2: 128.32.136.9 is NS for Scholarly.edu.
18.72.0.3
Name Server
ns.pedantic.edu.
domain | addr | type
thyme.pedantic.edu. | 18.62.5.70 AP
anise.pedantic.edu. | 18.64.1.14 AP
cse.pedantic.edu. | 18.26.0.36 NS
18.26.0.36
Name Server
ns3.cse.pedantic.edu.
domain | addr | type
ginger.cse.pedantic.edu | 18.26.0.82 AP
Response 3: 169.229.2.16 AP
Request 3: ginger.Scholarly.edu.?
Request 1: ginger.Scholarly.edu.?
Request: anise.pedantic.edu.?
Response: 18.64.1.14 AP
18.62.5.70
18.26.0.82
Name Client
ginger.cse.pedantic.edu.
name server: 192.112.36.4
Request 2: ginger.Scholarly.edu.?
Name Client
thyme.pedantic.edu.
name server: 18.72.0.3

FIGURE 4.10

Structure and operation of the Internet Domain Name System. In this figure, each circle represents a name server, and each rectangle is a name client. The type NS in a table or in a response means that this is a referral to another name server, while the type AP in a table or a response means that this is an Internet address. The dashed lines show the paths of the three requests made by the name client in the lower left corner to resolve the name `ginger.Scholarly.edu`, starting with the root name server. The dotted lines show resolution of a request of the name client in the lower right corner to resolve `anise.pedantic.edu` starting with a local name server.

name record for that name server binds the name `ns.iss.edu.` to Internet address 128.32.136.9." The illustration again omits the domain name of the name server.

This sequence repeats for each component of the original path name, until DNS_RESOLVE finally reaches a name server that has the name record for `ginger.Scholarly.edu`. That name server sends back a response saying, "The name record for `ginger.Scholarly.edu.` binds that name to Internet address 169.229.2.16." This being the answer to the original query, DNS_RESOLVE returns this result to its caller, which can go on to initiate an exchange of messages with its intended target.

The server that holds either a name record or a referral record for a domain name is known as the *authoritative name server* for that domain name. In our example, the name server `ns3.cse.pedantic.edu.` is authoritative for the `ginger.cse.pedantic.edu.` domain, as well as all other domain names that end with `cse.pedantic.edu.`, and `ns.iss.edu.` is authoritative for the `Scholarly.edu.` domain. Since a name server does not hold the name record for its own name, a name server cannot be the authoritative name server for its own name. Instead, for example, the root name server is authoritative for the domain name `edu.`, while the `names.edu.` name server is authoritative for all domain names that end in `edu.`.

That is the basic model of DNS operation. Here are some elaborations in its operation, each of which helps make the system fast-responding, robust, and capable of growing to a large scale.

1. It is not actually necessary to send the initial request to the root name server. DNS_RESOLVE can send the request to *any* convenient name server whose Internet address it knows. The name server doesn't care where the request came from; it simply compares the requested domain name with the list of domain names for which it is responsible in order to see if it holds a record that can help. If it does, it answers the request. If it doesn't, it answers by returning a referral to a root name server. The ability to send any request to a local name server means that the common case in which the client, the name server, and the target domain name are all three in the same domain (e.g., `pedantic.edu`) can be handled swiftly with a single request/response interaction. (The dotted lines in the lower right corner of Figure 4.10 show an example, in which `thyme.pedantic.edu.` asks the name server for the `pedantic.edu` domain for the address of `anise.pedantic.edu`.) This feature also simplifies name discovery because all a client needs to know is the Internet address of any nearby name server. The first request to that nearby server for a distant name (in the current example, `ginger.scholarly.edu`) will return a referral to the Internet address of a root name server.

2. Some domain name servers offer what is (perhaps misleadingly) called *recursive* name service. If the name server does not hold a record for the requested name, rather than sending a referral response, the name server takes on the responsibility for resolving the name itself. It forwards the initial request to a root name server, then continues to follow the chain of responses to resolve the complete path name, and finally returns the desired name record to its client. By

itself, this feature seems merely to simplify life for the client, but in conjunction with the next feature it provides a major performance enhancement.

3. Every name server is expected to maintain, in addition to its authoritative records, a cache of all name records it has heard about from other name servers. A server that provides recursive name service thus collects records that can greatly speed up future name resolution requests. If, for example, the name server for cse.pedantic.edu offers recursive service and it is asked to resolve the name flower.cs.scholarly.edu, in the course of doing so (assuming that it does not in turn request recursive service), its cache might acquire the following records:

```
edu                  refer to names.edu at 198.41.0.4
Scholarly.edu        refer to ns.iss.edu at 128.32.25.19
cs.Scholarly.edu     refer to cs.Scholarly.edu at 128.32.247.24
flower.cs.Scholarly.edu       Internet address is 128.32.247.29
```

   Now, when this name server receives, for example, the request to resolve the name psych.Scholarly.edu, it will discover the record for the domain Scholarly.edu in the cache and it will be able to quickly resolve the name by forwarding the initial request directly to the corresponding name server.

   A cache holds a duplicate copy, which may go out of date if someone changes the authoritative name record. On the basis that changes of existing name bindings are relatively infrequent in the Domain Name System and that it is hard to keep track of all the caches to which a domain name record may have propagated, the DNS design does not call for explicit invalidation of changed entries. Instead, it uses expiration. That is, the naming authority for a DNS record marks each record that it sends out with an expiration period, which may range from seconds to months. A DNS cache manager is expected to discard entries that have passed their expiration period. The DNS cache manager provides a memory model that is called *eventual consistency,* a topic taken up in Chapter 10 [on-line].

4.4.2 Hierarchical Name Management

Domain names form a hierarchy, and the arrangement of name servers described above matches that hierarchy, thereby distributing the job of name resolution. The same hierarchy also distributes the job of managing the handing out of names, by distributing the responsibility of operating name servers. Distributing responsibility is one of the main virtues of the distributed directory service model.

The way this works is actually quite simple: whoever operates a name server can be a *naming authority,* which means that he or she may add authoritative records to that name server. Thus, at some point early in the evolution of the Internet, some Pedantic University network administrator deployed a name server for the domain pedantic.edu and convinced the administrator of the edu domain to install a binding for the domain name pedantic.edu. associated with the name and Internet address of the `pedantic.edu` name server. Now, if Pedantic University wants to add a record, for example, for an Internet address that it wishes to name `archimedes.pedantic.edu`, its administrator can do so without asking permission of anyone else. A request to resolve the name `archimedes.pedantic.edu` can arrive at any domain name server in the Internet; that request will eventually arrive at the name server for the `pedantic.edu` domain, where it can be answered correctly. Similarly, a network administrator at the Institute for Scholarly Studies can install a name record for an Internet address named `archimedes.Scholarly.edu` on its own authority. Although both institutions have chosen the name `archimedes` for one of their computers, because the path names of the domains are distinct there was no need for their administrators to coordinate their name assignments. Put another way, their naming authorities can act independently.

Continuing this method of decentralization, any organization that manages a name server can create lower-level naming domains. For example, the Computer Science and Engineering Department of Pedantic University may have so many computers that it is convenient for the department to manage the names of those computers itself. All that is necessary is for the department to deploy a name server for a lower-level domain (named, for example, `cse.pedantic.edu`) and convince the administrator of the `pedantic.edu` domain to install a referral record for that name in its name server.

4.4.3 Other Features of DNS

To ensure high availability of name service, the DNS specification calls on every organization that runs a name service to arrange that there be at least two identical replica servers. This specification is important, especially at higher levels of the domain naming hierarchy, because most Internet activity uses domain names and inability to resolve a name component blocks reachability to all sites below that name component. Many organizations have three or four replicas of their name servers, and as of 2008 there were about 80 replicas of the root name server. Ideally, replicas should be attached to the network at places that are widely separated, so that there is some protection against local network and electric power outages. Again, the importance of separated attachment increases at higher levels of the naming hierarchy. Thus, the 80 replicas of the root name server are scattered around the world, but the three or four replicas of a typical organization's name server are more likely to be located within the campus of that organization. This arrangement ensures that, even if the campus is disconnected from the outside world, communication by name within the organization can still work. On the other hand, during such a disconnection, correspondents outside the organization cannot even verify that a name exists, for example, to validate an e-mail address. Therefore, a better arrangement might be to attach at least one of the organization's multiple replica name servers to another part of the Internet.

For the same reason that name servers need to be replicated, many network services also need to be replicated, so DNS allows the same name to be bound to several Internet addresses. In consequence, the *value* returned by DNS_RESOLVE can be a list of (presumably) equivalent Internet addresses. The client can choose which

Internet address to contact, based on order in the list, previous response times, a guess as to the distance to the attachment point, or any other criterion it might have available.

The design of DNS allows name service to be quite robust. In principle, the job of a DNS server is extremely simple: accept a request packet, search a table, and send a response packet. Its interface specification does not require it to maintain any connection state, or any other durable, changeable state; its only public interface is idempotent. The consequence is that a small, inexpensive personal computer can provide name service for a large organization, which encourages dedicating a computer to this service. A dedicated computer, in turn, tends to be more robust than one that supplies several diverse and unrelated network services. In addition a server with small, read-only tables can be designed so that when something such as a power failure happens, it can return to service quickly, perhaps even automatically. (Chapters 8 [on-line] and 9 [on-line] discuss how to design such a system.)

DNS also allows synonyms, in the form of indirect names. Synonyms are used conventionally to solve two distinct problems. For an example of the first problem, suppose that the Pedantic University Computer Science and Engineering Department has a computer whose Internet address is named `minehaha.cse.pedantic.edu`. This is a somewhat older and slower machine, but it is known to be very reliable. The department runs a World Wide Web server on this computer, but as its load increases the department knows that it will someday be necessary to move the Web server to a faster machine named `mississippi.cse.pedantic.edu`. Without synonyms, when the server moves, it would be necessary to inform everyone that there is a new name for the department's World Wide Web service. With synonyms, the laboratory can bind the indirect name `www.cse.pedantic.edu` to `minehaha.cse.pedantic.edu` and publicize the indirect name as the name of its Web site. When the time comes for `mississippi.cse.pedantic.edu` to take over the service, it can do so by simply having the manager of the `cse.pedantic.edu` domain change the binding of the indirect name. All those customers who have been using the name `www.cse.pedantic.edu` to get to the Web site will find that name continues to work correctly; they don't care that a different computer is now handling the job. As a general rule, the names of services can be expected to outlive their bindings to particular Internet addresses, and synonyms cater to this difference in lifetimes.

The second problem that synonyms can handle is to allow a single computer to appear to be in two widely different naming domains. For example, suppose that a geophysics group at the Institute of Scholarly Studies has developed a service to predict volcano eruptions but that organization doesn't actually have a computer suitable for running that service. It could arrange with a commercial vendor to run the service on a machine named, perhaps, `service-bureau.com` and then ask the manager of the Institute's name server to bind the indirect name `volcano.iss.edu` to `service-bureau.com`. The Institute could then advertise its service under the indirect name. If the commercial vendor raises its prices, it would be possible to move the service to a different vendor by simply rebinding the indirect name.

Because resolving a synonym requires an extra round-trip through DNS, and the basic name-to-Internet-address binding of DNS already provides a level of indirection, some network specialists recommend just manipulating name-to-Internet-address bindings to get the effect of synonyms.

4.4.4 Name Discovery in DNS

Name discovery comes up in at least three places in the Domain Name System: a client must discover the name of a nearby name server, a user must discover the domain name of a desired service, and the resolving system must discover an extension for unqualified domain names.

First, in order for DNS_RESOLVE to send a request to a name server, it needs to know the Internet address of that name server. DNS_RESOLVE finds this address in a configuration table. The real name-discovery question is how this address gets into the configuration table. In principle, this address would be the address of a root server, but as we have seen it can be the address of any existing name server. The most widely used approach is that when a computer first connects to a network it performs a name discovery broadcast to which the Internet service provider (ISP) responds by assigning the attacher an Internet address and also telling the attacher the Internet address of one or more name servers operated by or for the ISP. Another way to terminate name discovery is by direct communication with a local network manager, to obtain the address of a suitable name server, followed by configuring the answer into DNS_RESOLVE.

The second form of name discovery involves domain names themselves. If you wish to use the volcano prediction service at the Institute for Scholarly Studies, you need to know its name. Some chain of events that began with direct communication must occur. Typically, people learn of domain names via other network services, such as by e-mail, querying a search engine, reading postings in newsgroups or while surfing the Web, so the original direct communication may be long forgotten. But using each of those services requires knowing a domain name, so there must have been a direct communication at some earlier time. The purchaser of a personal computer is likely to find that it comes with a Web browser that has been preconfigured with domain names of the manufacturer's suggested World Wide Web query and directory services (as well as domain names of the manufacturer's support sites and other advertisers). Similarly, a new customer of an Internet service provider typically may, upon registering for service, be told the domain name of that ISP's Web site, which can then be used to discover names for many other services.

The third instance of name discovery concerns the extension that is used for unqualified domain names. Recall that the Domain Name System uses absolute path names, so if DNS_RESOLVE is presented with an unqualified name such as `library` it must somehow extend it, for example, to `library.pedantic.edu`. The default context used for extension is usually a configuration parameter of DNS_RESOLVE. The value of this parameter is typically chosen by the human user when initially setting up a computer, with an eye to minimizing typing for the most frequently used domain names.

4.4.5 Trustworthiness of DNS responses

A shortcoming of DNS is that, although it purports to provide authoritative name resolutions in its responses, it does not use protocols that allow authentication of those responses. As a result, it is possible (and, unfortunately, relatively easy) for an intruder to masquerade as a DNS server and send out mischievous or malevolent responses to name resolution requests.

Currently, the primary way of dealing with this problem is for the user of DNS to treat all of its responses as potentially unreliable hints and independently verify (using the terminology of Chapters 7 [on-line] and 11 [on-line] we would say "perform end-to-end authentication of") the identity of any system with which that user communicates. An alternative would be for DNS servers to use authentication protocols in communication with their clients. However, even if a DNS response is assuredly authentic, it still might not be accurate (for example, a DNS cache may hold out-of-date information, or a DNS administrator may have configured an incorrect name-to-address binding), so a careful user would still want to independently authenticate the identity of its correspondents.

Chapter 11 [on-line] describes protocols that can be used for authentication; there is an ongoing debate among network experts as to whether or how DNS should be upgraded to use such protocols.

The reader interested in learning more about DNS should explore the documents in the readings for DNS [Suggestions for Further Reading 4.3].

4.5 CASE STUDY: THE NETWORK FILE SYSTEM (NFS)

The Network File System (NFS), designed by Sun Microsystems, Inc. in the 1980s, is a client/service application that provides shared file storage for clients across a network. An NFS client grafts a remote file system onto the client's local file system name space and makes it behave like a local UNIX file system (see Section 2.5). Multiple clients can mount the same remote file system so that users can share files.

The need for NFS arose because of technology improvements. Before the 1980s, computers were so expensive that each one had to be shared among multiple users and each computer had a single file system. But a benefit of the economic pressure was that it allowed for easy collaboration because users could share files easily. In the early 1980s, it became economically feasible to build workstations, which allowed each engineer to have a private computer. But users still desired to have a shared file system for ease of collaboration. NFS provides exactly that: it allows a user at any workstation to use files stored on a shared server, a powerful workstation with local disks but often without a graphical display.

NFS also simplifies the management of a collection of workstations. Without NFS, a system administrator must manage each workstation and, for example, arrange for backups of each workstation's local disk. NFS allows for centralized management; for example, a system administrator needs to back up only the server's disks to archive the file system. In the 1980s, the setup also had a cost benefit: NFS allowed organizations to buy workstations without disks, saving the cost of a disk interface on every workstation and the cost of unused disk space on each workstation.

The designers of NFS had four major goals. NFS should work with existing applications, which means NFS should provide the same semantics as a local UNIX file system. NFS should be deployable easily, which means its implementation should be able to retrofit into existing UNIX systems. The client should be implementable in other operating systems such as Microsoft's DOS, so that a user on a personal computer can have access to the files on an NFS server; this goal implies that the client/service messages cannot be too UNIX system-specific. Finally, NFS should be efficient enough to be tolerable to users, but it doesn't have to provide as high performance as local file systems. NFS only partially achieves these goals because achieving them all is difficult. The designers made a trade-off: simplify the design and lose some of the UNIX semantics.

This section describes version 2 of NFS. Version 1 was never deployed outside of Sun Microsystems, while version 2 has been in use since 1987. The case study concludes with a brief summary of the changes in versions 3 (1990s) and 4 (early 2000s), which address weaknesses in version 2. Problem set *3* explores an NFS-inspired design to reinforce the ideas in NFS.

4.5.1 Naming Remote Files and Directories

To programs, NFS appears as a UNIX file system providing the file interface presented in Section 2.5. User programs can name remote files in the same way as local files. When a user program invokes, say, OPEN ("/users/alice/.profile", READONLY), it cannot tell from the path name whether "users" or "alice" are local or remote directories.

To make naming remote files transparent to users and their programs, the NFS client must mount a remote file system on the local name space. NFS performs this operation by using a separate program, called the *mounter*. This program serves a similar function as the MOUNT call (described in Section 2.5.10); it grafts the remote file system—named by *host*:*path*, where *host* is a DNS name and *path* a path name—onto the local file name space. The mounter sends a remote procedure call to the file server *host* and asks for a *file handle*, a 32-byte name for the object identified by *path*. On receiving the reply, the NFS client marks the mount point in the local file system as a remote file system. It also remembers the file handle for *path* and the network address for the server.

To the NFS client a file handle is a 32-byte opaque name that identifies an object on a remote NFS server. An NFS client obtains file handles from the server when the client mounts a remote file system, or it looks up a file in a directory on the NFS server. In all subsequent remote procedure calls to the NFS server for that file, the NFS client includes the file handle. In many ways the file handle is similar to an inode number; it is not visible to applications, but it used as a name internal to NFS to name files.

To the NFS server a file handle is a structured name—containing a *file system identifier,* an *inode number*, and *a generation number*—which the server can use to locate the file. The file system identifier allows the server to identify the file system responsible for the file. The inode number (see page 58) allows the identified file system to locate the file on the disk.

One might wonder why the NFS designers didn't choose to put path names in file handles. To see why, consider the following scenario with two user programs running on different clients:

```
      Program 1 on client 1              Program 2 on client 2      |
                                                                    |
1         CHDIR ("dir1")                                            |
2         fd ← OPEN ("f", READONLY)                                 |  Time
3                                          RENAME ("dir1", "dir2")  |
4                                          RENAME ("dir3", "dir1")  ↓
5         READ (fd, buf, n)
```

RENAME (*source*, *destination*) changes the name of *source* to *destination*. The first rename operation (on line *3*) in program 2 renames "dir1" to "dir2", and the second one (on line *4*) renames "dir3" to "dir1". This scenario raises the following question: when program 1 invokes READ (line *5*) after the two rename operations have completed, does program 1 read data from "dir1/f", or "dir2/f"?

If the two programs were running on the same computer and sharing a local UNIX file system, program 1 would read "dir2/f", according to the UNIX specification. The goal is that NFS should provide the same behavior. If the NFS server were to put path names inside handles, then the READ call would result in a remote procedure call for the file "dir1/f". By putting the inode number in the handle the specification is met.

The file handle includes a generation number to handle scenarios such as the following almost correctly:

```
      Program 1 on client 1              Program 2 on client 2      Time
                                                                     |
1                                          fd ← OPEN ("f", READONLY) |
2         UNLINK ("f")                                               |
3         fd ← OPEN ("f", CREATE)                                    |
4                                          READ (fd, buf, n)         ↓
```

A program on a client 1 deletes a file "f" (line *2*) and creates a new file with the same name (line *3*), while another program on a client 2 already has opened the original file (on line *1*). If the two programs were running on the same computer and sharing a local UNIX file system, program 2 would read the old file on line *4*.

If the server should happen to reuse the inode of the old file for the new file, remote procedure calls from client 2 will get the new file, the one created by client 1, instead of the old file. The generation number allows NFS to avoid this incorrect behavior. When the server reuses an inode, it increases the generation number by one. In the example, client 1 and client 2 would receive different file handles, and client 2 will use the old handle. Increasing the generation number makes it always safe for the NFS server to recycle inodes immediately.

For this scenario, NFS does not provide identical semantics to a local UNIX file system because that would require that the server know which files are in use. With NFS, when client 2 uses the file handle, it will receive an error message: "stale file handle".

This case is one example of the NFS designers trading some UNIX semantics to obtain a simpler implementation.

File handles are usable across server failures, so that even if the server computer fails and restarts between a client program opening a file and then reading from the file, the server can identify the file using the information in the file handle. Making file handles (which include a file system identifier and a generation number) usable across server failures requires small changes to the server's on-disk file system: the NFS designers modified the super block to record the file system identifier and modified inodes to record the generation number for the inode. With this information recorded, after a reboot the NFS server will be able to process NFS requests that the server handed out before it failed.

4.5.2 The NFS Remote Procedure Calls

Table 4.1 shows the remote procedure calls used by NFS. The remote procedure calls are best explained by example. Suppose we have the following fragment of a user program:

```
fd ← OPEN ("f", READONLY)
READ (fd, buf, n)
CLOSE (fd)
```

Table 4.1 NFS Remote Procedure Calls

Remote Procedure Call	Returns
NULL ()	Do nothing.
LOOKUP (*dirfh*, *name*)	fh and file attributes
CREATE (*dirfh*, *name*, *attr*)	fh and file attributes
REMOVE (*dirfh*, *name*)	status
GETATTR (*fh*)	file attributes
SETATTR (*fh*, *attr*)	file attributes
READ (*fh*, *offset*, *count*)	file attributes and data
WRITE (*fh*, *offset*, *count*, *data*)	file attributes
RENAME (*dirfh*, *name*, *tofh*, *toname*)	status
LINK (*dirfh*, *name*, *tofh*, *toname*)	status
SYMLINK (*dirfh*, *name*, *string*)	status
READLINK (*fh*)	string
MKDIR (*dirfh*, *name*, *attr*)	fh and file attributes
RMDIR (*dirfh*, *name*)	status
READDIR (*dirfh*, *offset*, *count*)	directory entries
STATFS (*fh*)	file system information

Figure 4.11 shows the corresponding timing diagram where "f" is a remote file. The NFS client implements each file system operation using one or more remote procedure calls.

In response to the program's call to OPEN, the NFS client sends the following remote procedure call to the server:

LOOKUP (*dirfh*, "f")

From before the program runs, the client has a file handle for the current working directory's (*dirfh*). It obtained this handle as a result of a previous lookup or as a result of mounting the remote file system.

On receiving the LOOKUP request, the NFS server extracts the file system identifier and inode number from *dirfh*, and asks the identified file system to look up the inode number in *dirfh*. The identified file system uses the inode number in *dirfh* to locate the directory's inode. Now the NFS server searches the directory identified by the inode number for "f". If present, the server creates a handle for "f". The handle contains the file system identifier of the local file system, the inode number for "f", and the generation number stored in the inode of "f". The NFS server sends this file handle to the client.

On receiving the response, the client allocates the first unused entry in the program's file descriptor table, stores a reference to f's file handle in that entry, and returns the index for the entry (*fd*) to the user program.

Next, the program calls READ (*fd, buf, n*). The client sends the following remote procedure call to the NFS server:

READ (*fh*, 0, *n*)

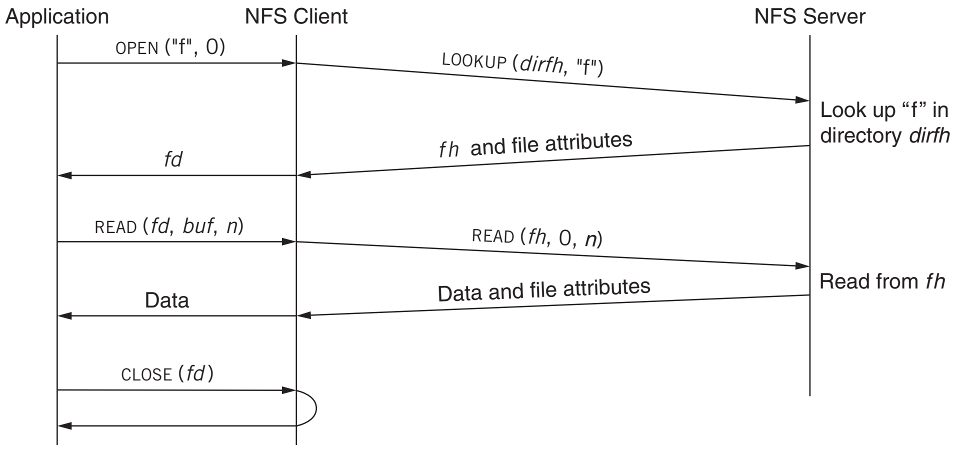


FIGURE 4.11

Example interaction between an NFS client and service. Since the NFS service is stateless, the client does not need to inform the service when the application calls CLOSE. Instead, it just deallocates *fd* and returns.

As with the directory file handle, the NFS server looks up the inode for *fh*. Then, the server reads the data and sends the data in a reply message to the client.

When the program calls CLOSE to tell the local file system that it is done with the file descriptor *fd*, NFS doesn't issue a CLOSE remote procedure call; the protocol doesn't have a CLOSE remote procedure call. Because the application didn't modify the file, the NFS client doesn't have to issue any remote procedure calls. As we shall see in Section 4.5.4, if a program modifies a file, the NFS client will issue remote procedure calls on a CLOSE system call to provide coherence for the file.

The NFS remote procedure calls are designed so that the server can be ***stateless***, that is, the server doesn't need to maintain any other state than the on-disk files. NFS achieves this property by making each remote procedure call contain all the information necessary to carry out that request. The server does not maintain any state about past remote procedure calls to process a new request. For example, the client, not the server, must keep track of the file cursor (see Section 2.3.2), and the client includes it as an argument in the READ remote procedure call. As another example, the file handle contains all information to find the inode on the server, as explained above.

This stateless property simplifies recovery from server failures: a client can just repeat a request until it receives a reply. In fact, the client cannot tell the difference between a server that failed and recovered, and a server that is slow. Because a client repeats a request until it receives a response, it can happen that the server executes a request twice. That is, NFS implements at-least-once semantics for remote procedure calls.

Since many requests are idempotent (e.g., LOOKUP, READ, etc.), that is not a problem, but for some requests it results in surprising behavior. Consider a user program that calls UNLINK on an existing file that is stored on a remote file system. The NFS client would send a REMOVE remote procedure call and the server would execute it, but it could happen that the network lost the reply. In that case, the client would resend the REMOVE request, the server would execute the request again, and the user program would receive an error saying that the file didn't exist!

Later implementations of NFS minimize this surprising behavior by avoiding executing remote procedure calls more than once when there are no server failures. In these implementations, each remote procedure call is tagged with a transaction number and the server maintains some "soft" state (it is lost if the server fails), namely, a reply cache. The reply cache is indexed by transaction identifier and records the response for the transaction identifier. When the server receives a request, it looks up the transaction identifier (ID) in the reply cache. If the ID is in the cache, the server returns the reply from the cache, without reexecuting the request. If the ID is not in the cache, the server processes the request.

If the server doesn't fail, a retry of a REMOVE request will receive the same response as the first attempt. If, however, the server fails and restarts between the first attempt and a retry, the request is executed twice. The designers opted to maintain the reply cache as soft state because storing it in non-volatile storage is expensive. Doing so would require that the reply cache be stored, for example, on a disk and would require a disk write for each remote procedure call to record the response. As explained in Section 6.1.8, disk writes are often a performance bottleneck and much more expensive than a remote procedure call.

Although the stateless property of NFS simplifies recovery, it makes it impossible to implement the UNIX file interface correctly because the UNIX specification requires maintaining state. Consider again the case where one program deletes a file that another program has open. The UNIX specification is that the file exists until the second program closes the file.

If the programs run on different clients, NFS cannot adhere to this specification because it would require that the server keep state. It would have to maintain a reference count per file, which would be incremented on an OPEN system call and decremented on a CLOSE system call, and persist across server failures. In addition, if a client would not respond to messages, the server would have to wait until the client becomes reachable again to decrement the reference count. Instead, NFS just does the easy but slightly wrong thing: remote procedure calls return an error "stale file handle" if a program on another client deletes a file that the first client has open.

NFS does not implement the UNIX specification faithfully because that simplifies the design of NFS. NFS preserves most of the UNIX semantics, and only in rarely encountered situations may users see different behavior. In practice, these situations are not a serious problem, and in return NFS gets by with simple recovery.

4.5.3 Extending the UNIX File System to Support NFS

To implement NFS as an extension of the UNIX file system while minimizing the number of changes required to the UNIX file system, the NFS designers split the file system program by introducing an interface that provides *vnodes*, virtual nodes (see Figure 4.12). A vnode is a structure in volatile memory that abstracts whether a file or directory is implemented by a local file system or a remote file system. This design allows many functions in the file system call layer to be implemented in terms of vnodes, without having to worry about whether a file or directory is local or remote. The interface has an additional advantage: a computer can easily support several, different local file systems.

When a file system call must perform an operation on a file (e.g., reading data), it invokes the corresponding procedure through the vnode interface. The vnode interface has procedures for looking up a file name in the contents of a directory vnode, reading from a vnode, writing to a vnode, closing a vnode, and so on. The local file system and NFS support their own implementation of these procedures.

By using the vnode interface, most of the code for file descriptor tables, current directory, name lookup, and the like, can be moved from the local file system module into the file system call layer with minimal effort. For example, with a few changes, the procedure PATHNAME_TO_INODE from Section 2.5 can be modified to be PATHNAME_TO_VNODE and be provided by the file system call layer.

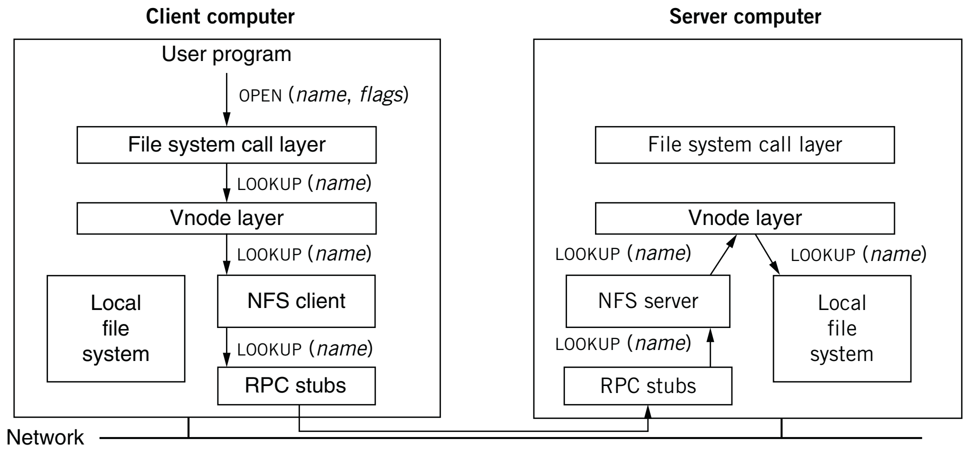


FIGURE 4.12

NFS implementation for the UNIX system

To illustrate the vnode design, we consider a user program that invokes OPEN for a file (see Figure 4.12). To open the file, the file system call layer invokes PATHNAME_TO_VNODE, passing the vnode for the current working directory and the path name for the file as arguments. PATHNAME_TO_VNODE will parse the path name, invoking LOOKUP in the vnode layer for each component in the path name. If the directory is a local directory, the vnode-layer LOOKUP invokes the LOOKUP procedure implemented by the local file system to obtain a vnode for the path name component. If the directory is a remote directory, LOOKUP invokes the LOOKUP procedure implemented by the NFS client.

The NFS client invokes the LOOKUP remote procedure call on the NFS server, passing as arguments the file handle of the directory and the path name's component. On receiving the lookup request, the NFS server extracts the file system identifier and inode number from the file handle for the directory to look up the directory's vnode and then invokes LOOKUP in the vnode layer, passing the path name's component as an argument. If the directory is implemented by the server's local file system, the vnode layer invokes the procedure LOOKUP implemented by the server's local file system, passing the path name's component as an argument. The local file system looks up the name and, if present, creates a vnode and returns the vnode to the NFS server. The NFS server sends a reply containing the vnode's file handle and some metadata for the vnode to the NFS client.

On receiving the reply, the NFS client creates a vnode, which contains the file handle, on the client computer and returns it to the file system call layer on the client machine. When the file system call layer has resolved the complete path name, it returns a file descriptor for the file to the user program.

To achieve usable performance, a typical NFS client maintains various caches. A client stores the vnode for every open file so that the client knows the file handles for open files. A client also caches recently used vnodes, their attributes, recently used blocks of those cached vnodes, and the mapping from path name to vnode. Caching reduces the latency of file system operations on remote files because for cached files a client can avoid the cost of remote procedure calls. In addition, because clients make fewer remote procedure calls, a single server can support more clients. If multiple clients cache the same file, however, NFS must ensure read/write coherence in some way.

4.5.4 Coherence

When programs share a local file in a UNIX system, the program calling READ observes the data from the most recent WRITE, even if this WRITE was performed by another program. This property is called read/write coherence (see Section 2.1.1.1). If the programs are running on different clients, caching complicates implementing these semantics correctly.

To illustrate the problem, consider a user program on one computer that writes a block of a file. The file system call layer on that computer might perform the update to the block in the cache, delaying the write to the server, just like the local UNIX file system delays a write to disk. If a program on another computer then reads the file from the server, it may not observe the change made on the first computer because that change may not have been propagated to the server yet. Because this behavior would be incorrect, NFS implements a form of read/write coherence.

NFS could guarantee read/write coherence for every operation, or just for certain operations. One option is to provide read/write coherence for only OPEN and CLOSE. That is, if an application OPENs a file, WRITEs, and CLOSEs the file on one client, and if later an application on a second client opens the same file, then the second application will observe the results of the writes by the first application. This option is called *close-to-open consistency.* Another option is to provide read/write coherence for every read and write. That is, if two applications on different clients have the same file open concurrently, then a READ of one observes the results of WRITEs of the other.

Many NFS implementations provide close-to-open consistency because it allows for higher data rates for reading or writing a big file; a client can send several reads or write requests without having to wait for a response after each request. Figure 4.13 illustrates close-to-open semantics in more detail. If, as in case 1, a program on one client calls WRITE and then CLOSE, and then another client calls OPEN and READ, the NFS implementation will ensure that the READ will include the results of the WRITEs by the first client. But, as in case 2, if two clients have the same file open, one client writes a block of the file, and then the other client invokes READ, READ may return the data either from before or after the last WRITE; the NFS implementation makes no guarantees in that case.

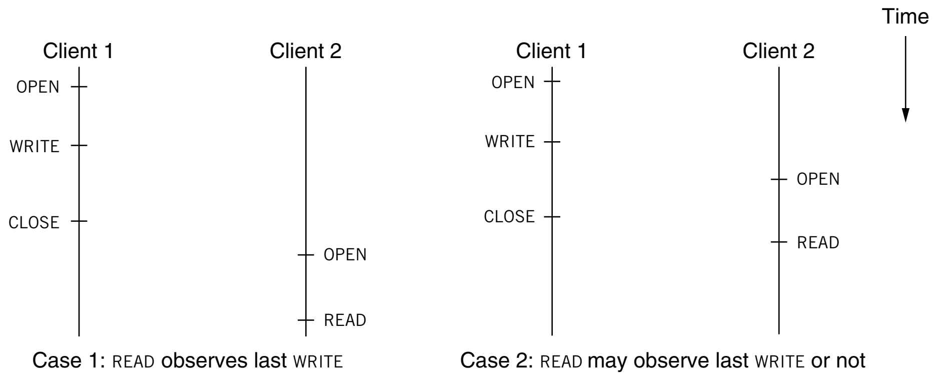


FIGURE 4.13

Two cases illustrating close-to-open consistency

NFS implementations provide close-to-open semantics as follows. The client stores with each data block in its cache the modification of the block's vnode at the time the client reads the block from the server. When a user program opens a file, the client sends a GETATTR request to fetch the last modification time of the file. The client reads a cached data block only if the block's modification time is the same as its vnode's modification time. If the modification times are not the same, the client removes the data block from its cache and fetches it from the server.

The client implements WRITE by modifying its local cached version, without incurring the overhead of remote procedure calls. Then, in the CLOSE call of Figure 4.11, the client, rather than simply returning, would first send any cached writes to the server and wait for an acknowledgment. This implementation is simple and provides decent performance. The client can perform READs and WRITEs at local memory speeds. By delaying sending the modified blocks until CLOSE, the client absorbs modifications that are overwritten (e.g., the application writes the same block multiple times) and aggregates WRITEs to the same block (e.g., WRITEs that modify different parts of the block).

By providing close-to-open semantics, most user programs written for a local UNIX file system will work correctly when their files are stored on NFS. For example, if a user edits a program on a personal workstation but prefers to compile on a faster compute machine, then NFS with close-to-open consistency works well, requiring no modifications to the editor and the compiler. After the editor has written out the modified file and the user starts the compiler on the compute machine, the compiler will observe the edits.

On the other hand, certain programs will not work correctly using NFS implementations that provide close-to-open consistency. For example, a multiclient database program that reads and writes records stored in a file over NFS will not work correctly because, as the second case in Figure 4.13 illustrates, close-to-open semantics doesn't specify the semantics when clients execute operations concurrently—for example, if client 2 opens the database file before client 1 closes it and client 3 opens the database file after client 1 closes it. If client 2 and 3 then read data from the file, client 2 may not see the data written by client 1, while client 3 will see the data written by client 1.

Furthermore, because NFS caches blocks (instead of whole files), the file may have blocks from different versions of the file intermixed. When a client fetches a file, it fetches only the inode and perhaps prefetches a few blocks. Subsequent READ RPCs may fetch blocks from a newer version of the file because another client may have written those blocks after this client opened the file.

To provide the correct semantics in this case requires more sophisticated machinery, which NFS implementations don't provide, because databases often have their own special-purpose solutions anyway, as we discuss in Chapters 9 [on-line] and 10 [on-line]. If the database program doesn't provide a special-purpose solution, then tough luck—one cannot run it over NFS.

4.5.5 NFS Version 3 and Beyond

NFS version 2 is being replaced by NFS version 3. Version 3 addresses a number of limitations in version 2, but the extensions do not significantly change the preceding description. For example, version 3 supports 64-bit numbers for recording file sizes and adds an asynchronous write (i.e., the server may acknowledge an asynchronous WRITE request as soon as it receives the request, before it has written the data to disk).

NFS version 4, which took a number of lessons from the Andrew File System [Suggestions for Further Reading 4.2.3], is a bigger change than version 3; in version 4 the server maintains some state. Version 4 also protects against intruders who can snoop and modify network traffic using techniques discussed in Chapter 11 [on-line]. Furthermore, it provides a more efficient scheme for providing close-to-open consistency, and it works well across the Internet, where the client and server may be connected using low-speed links.

The following references provide more details on NFS:

1. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and implementation of the Sun network file system", *Proceedings of the 1985 Summer Usenix Technical Conference*, June 1985, El Cerrito, CA, pages 119–130.

2. Chet Juszezak, "Improving the performance and correctness of an NFS server", *Proceedings of the 1989 Winter Usenix Technical Conference*, January 1989, Berkeley, CA, pages 53–63.

3. Brian Pawlowski, Chet Juszezak, Peter Staubach, Carl Smith, Diana Lebel, and David Hitz, "NFS Version 3 design and implementation", *Proceedings of the 1990 Summer Usenix Technical Conference*, June 1994, Boston, MA.

4. Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Turlow, "The NFS Version 4 protocol", *Proceedings of Second International SANE Conference*, May 2000, Maastricht, The Netherlands.

EXERCISES

4.1 When modularity between a client and a service is enforced, there is no way for errors in the implementation of the service to propagate to its clients. True or False? Explain.

1995–1–1d

4.2 Chapter 1 discussed four general methods for coping with complexity: modularity, abstraction, hierarchy, and layering.

4.2a Which of those four methods does client/service use as its primary organizing scheme?

4.2b Which does remote procedure call use? Explain.

1996–1–1b,d

4.3 To client software, a notable difference between remote procedure call and ordinary local procedure call is:

A. None. That's the whole point of RPC!
B. There may be multiple returns from one RPC call.
C. There may be multiple calls for one RPC return.
D. Recursion doesn't work in RPC.
E. The runtime system may report a new type of error as a result of an RPC.
F. Arguments to RPCs must be scalars.

1998–2–4

4.4 Which of the following statements is true of the X Window System (see Sidebar 4.4)?

A. The X server is a trusted intermediary and attempts to enforce modularity between X clients in their use of the display resource.
B. An X client always waits for a response to a request before sending the next request to the X server.
C. When a program running on another computer displays its window on your local workstation, that *remote* computer is considered an X server.

2005–1–6

4.5 While browsing the Web, you click on a link that identifies an Internet host named `www.cslab.scholarly.edu`. Your browser asks your Domain Name System (DNS) name server, *M*, to find an Internet address for this domain name. Under what conditions is each of the following statements true of the name resolution process?

A. To answer your query, *M* must contact one of the root name servers.

B. If *M* answered a query for `www.cslab.scholarly.edu` in the past, then it can answer your query without asking any other name server.

C. M must contact one of the name servers for `cslab.scholarly.edu` to resolve the domain name.

D. If *M* has the current Internet address of a working name server for `scholarly.edu` cached, then that name server will be able to directly provide an answer.

E. If *M* has the current Internet address of a working name server for `cslab.scholarly.edu` cached, then that name server will be able to directly provide an answer.

4.6 For the same situation as in Exercise 4.5, which of the following is always true of the name resolution process, assuming that all name servers are configured correctly and no messages are lost?

A. If M had answered a query for the IP address corresponding to `www.cslab.scholarly.edu` at some time in the past, then it can respond to the current query without contacting any other name server.

B. If M has a valid IP address of a functioning name server for `cslab.scholarly.edu` in its cache, then M will get a response from that name server without any other name servers being contacted.

2000–2–5 and 2005–2–4

4.7 The Network File System (NFS) described in Section 4.5 allows a client machine to run operations on files that are stored at a remote server. For the version of NFS described there, decide if each of these assertions is true or false:

A. When the server responds to a client's WRITE call, all modifications required by that WRITE will have made it to the server's disk.

B. An NFS client might send multiple requests for the same operation to the NFS server.

C. When an NFS server crashes, after the operating system restarts and recovers the disk contents, the server must also run its own recovery procedure to make its state consistent with that of its clients.

2005–1–2

4.8 Assume that an NFS (described in Section 4.5) server contains a file `/a/b` and that an NFS client mounts the NFS server's root directory in the location `/x`, so that the client can now name the file as `/x/a/b`. Further assume that this is the only client and that the client executes the following two commands:

```
chdir /x/a
rm b
```

The REMOVE message from the client to the server gets through, and the server removes the file. Unfortunately, the response from the server to the client is lost and the client resends the message to remove the (now non-existent) file. The server receives the resent message. What happens next depends on the server implementation. Which of the following are correct statements?

A. If the server maintains an in-memory reply cache in which it records all operations it previously executed, and there are no server failures, the server will return "OK".

B. If the server maintains an in-memory reply cache but the server has failed, restarted, and its reply cache is empty, both of the following responses are possible: the server may return "file not found" or "OK".

C. If the server is stateless, it will return "file not found".

D. Because `REMOVE` is an idempotent operation, any server implementation will return "OK".

2006-2-2

Additional exercises relating to Chapter 4 can be found in the problem sets beginning on page 425.