SAMS

Linux® Cluster Architecture

Alex Vrenios

Alex Vrenios

Linux Cluster Architecture

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Linux Cluster Architecture

Copyright © 2002 by Sams Publishing

International Standard Book Number: 0-672-32368-0

Library of Congress Catalog Card Number: 752063323687

Printed in the United States of America

First Printing: June 2002

05 04 03 02 4 3 2 1

Acquisitions Editor
Kathryn Purdum

Development Editor
Steve Rowe

Managing Editor
Charlotte Clapp

Project Editor
George E. Nedeff

Copy Editor
Barbara Hacha

Indexer
Sharon Shock

Proofreader
Linda Seifert

Technical Editor
Kurt Wall

Team Coordinator
Amy Patton

Multimedia Developer
Dan Scherf

Interior Designer
Gary Adair

Cover Designer
Alan Clements

Page Layout
Brad Lenser

Contents at a Glance

Table of Contents

About the Author

Alex Vrenios is Chief Scientist at the Distributed Systems Research Lab, a technical consulting group specializing in the performance measurement and analysis of distributed computing architecture. He holds a B.S. and M.S. in Computer Science, is the author of numerous articles, and is a Professional member of the ACM and a Senior member of the IEEE. Alex started his career working with communication and data transmission software on mainframe computers. When he made the transition to the networked C/Unix environment and found out about interprocess communication, distributed algorithms, and cluster computing, he never looked back. After nearly 30 years in industry, he believes it is time to organize his experiences and passions for publication in an effort to leave a contribution for others to build on. He married his lovely wife, Diane, 13 years ago. They have two Miniature Schnauzers named Brutus and Cleo. His other interests include amateur radio, astronomy, photography, bicycling, hiking, and riding a bright red Honda PC800 motorcycle.

Dedication

for Diane

Acknowledgments

I would like to thank my lovely wife, Diane, for her patience in allowing me to undertake this long-term project when we have plenty of other demands on our time, for her diligence in reading, rereading, and gently suggesting improvements to the quality of this work, and for her confidence in my ability to present these complex technical issues.

Let me also thank the editors at Sams Publishing, without whose assistance and direction I would never have completed this work.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book, and that because of the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	`opensource@samspublishing.com`
Mail:	Mark Taber Associate Publisher Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

For more information about this book or another Sams title, visit our Web site at `www.samspublishing.com`. Type the ISBN (excluding hyphens) or the title of a book into the Search field to find the page you're looking for.

Introduction

You may have opened this particular book because you heard or read something recently about cluster computing. It is definitely a hot topic! You might be a hobbyist who enjoys the hands-on experience of working with computers or an engineer or engineering manager who needs to understand the trade-offs and the pitfalls in cluster computer development. I hope that when you finish reading this book, you will either have or know how to design and build a working cluster computer. This may help you understand what kinds of problems your development project is likely to encounter. It may allow you to test and tweak distributed algorithms that are destined to run on hardware that is still under development, in parallel with your software. Or your cluster might be the first parallel Web server in your local computer club. Whatever your motivation, I want you to know how much I enjoyed learning and employing the skills necessary to build such systems; I consider it an honor and a privilege to be the one to pass them along to you.

Alex Vrenios
Phoenix, AZ

1 Linux Cluster Computer Fundamentals

IN THIS CHAPTER

This book is about cluster computers. It explains what they are, how they can be used, what kind of hardware is needed to build one, and what kinds of software architectures bring out the best in them.

Clusters are a hot topic. What is unique about them is that the personality of their architecture—how they appear to behave to their users—is so dependent on their software. The same collection of computers interconnected via some network can become a supercomputer, a fault-tolerant server, or something completely different—something you design!

These architectural varieties are all a part of *distributed computing*, a network of individual computers working together at solving a problem or providing a service. A cluster computer is a distributed system; this concept will become more familiar in the near future as the world's problems become more geographically dispersed. Examples include international financial transactions, global weather forecasting, and rapid military deployment. The farthest reaches of the world are getting closer together through the proliferation of distributed computer and communication systems. Exposure to their underlying principles might help you get a better job some day or move up the ranks in one you already have.

Why a Cluster?

Let's begin our journey with a couple of loose definitions. First, just what is a *cluster computer*? A cluster computer is a set of individual computers that are connected through specialized hardware and software, presenting a single-system image to its users.

What is a cluster computer? A bunch of PCs connected on a network does *not* qualify as a cluster computer. To be a cluster computer, each of these PCs must be running software that takes advantage of the fact that other PCs are available. They must be made to act like a team, working together and supporting one another toward a common goal, presenting a single-system image to their users.

What is a *single-system image* then? A single-system image is one that is indistinguishable from that of a single computer. Its users are unaware that more than one computer exists.

Consider the PC on your desk connected to some distant Web site over the Internet. With each click of your mouse, you are downloading that Web page's text and image file data into your browser for display. The Web server could be dozens of PCs in a network—a cluster server. Each time your browser asks for new data, its request packet could be broken up and passed along to several of the other computers, which read the files from disk and send them back to your browser. You are unlikely to be aware that these computers were working together to service your requests. Such a cluster server presents a single-system image to your browser, to the client, and to you, the user.

Why bother? Why not just start with a single high-speed server and replace it with a faster one each time your client resource requirement grows? The answers to these questions encompass some complex issues that are the subject of much of today's computer science research: cost-effectiveness, scalability, high availability, and fault tolerance. You'll learn more about these issues later in the chapter.

Computer science brought home the concept of how a *stepwise process* leads to a desired goal. Further, it showed that any process toward a desired goal can be broken down into a set of steps, and that each step can be further broken down into smaller steps, and so on. If all the achievements to date and all the motivations that continue to drive computer science research could be condensed into a single word, that word might be *overlap*.

Overlap refers to the capability to perform two or more functions simultaneously. Early computers were required to stop all their other processing just to print a line of characters. In contrast, modern computers routinely execute instructions while redrawing a screen image, reading from or writing to the hard drive, printing, and so on. Exploiting even the smallest potential for overlap can yield a tremendous improvement in performance.

A cluster computer is a type of parallel processor called a *multicomputer* to distinguish it from a true multiprocessor, which has a single physical memory that is shared among its processors. Later chapters discuss some different kinds of parallel processors. For now, let's look toward exploiting overlap at the task execution level. You can do this in a cluster by finding tasks or portions of tasks that can be executed in parallel and by scheduling these chunks on different processors in your cluster. As each chunk finishes, its results are returned to the scheduling processor. Individual results are combined into what serial processing of these chunks might produce. This combined result is fed into the next high-level step, and so on, until all the task's processing is complete.

Examining the interdependencies among the steps of any process, you can see that some steps must be executed in strict sequence, the outcome of one feeding the requirements of another. Other steps are independent of one another and may be executed in parallel, allowing you to exploit any parallelism inherent in that process. The benefit of executing steps in parallel is a shorter process-completion time. The cost is in the extra resources required to perform this parallel execution and the time required to schedule their parallel execution. Cluster computing is one kind of parallel processing. You'll see some other forms in later chapters.

Gene Amdahl put the benefits of parallel processing in perspective back in the 1960s. He pointed out that of all the time spent by a parallel processor executing your application, it is only in that part of it where parallelism occurs, that any time can be saved, and any time savings will be in proportion to the amount of inherent parallelism, minus the overhead in managing parallel processes. This concept later came to be known as Amdahl's Law.

If the inherent parallelism in your application is so small that most of the steps have to execute sequentially, you probably do not want to consider using a parallel architecture. There are enough common processes with inherent parallelism, however, to generate a lot of interest in parallel processors. The cluster architecture, in particular, gives you the benefits of a scalable parallel processor at a very low cost and a potential for high availability through software fault tolerance.

Architectural Design Features

Cost-effectiveness, in terms of performance and reliability, is the single most important design issue. The goal is to assemble several individual computers and to sacrifice some resources in managing them as a cluster system. This should result in a higher level of performance and reliability than what you can get from a single computer in the same price range.

Scalability, high availability, and fault tolerance are the three features necessary to achieve high performance and reliability. If the support software overhead for any of these is too high, the cluster may not expand beyond a trivial number of nodes, or it may fail to sustain even a modest query load. The following sections look at each of these features.

Scalability

As you add more node computers to the cluster, the number of simultaneous users it can service should increase proportionally. Reality, however, shows that the performance increase is limited to some upper number of individual computers because of the overhead of interprocessor (computer-to-computer) communication. As the number of individual computers increases, the performance improves up to a point and then drops off; this is called the *point of diminishing returns*.

Much of computer science research on scalability is focused on moving that point further and further out, providing a corresponding increase in performance over a greater range as more node computers are added to the cluster. You learn how to determine the exact value of that point for your cluster in Chapter 8, "External Performance Measurement/Analysis." Performance measurement is an iterative process, which means that you will take a baseline measurement, tweak your system, and then measure again to see what effect your efforts had on system scalability.

High Availability

In a single-computer server implementation, a hardware failure can bring all its services to an unpleasant halt. In a cluster server implementation, such a loss can mean that one or more computers go down, suffering only a degraded performance level proportional to that loss. The advantage is that you have several other computers still running, which are able to keep your critical applications running. You are not down completely, as in the single-server architecture.

Exactly how one computer in a cluster senses the failure of another, how it chooses another computer to assume the responsibilities and the resources of the failing system, and exactly how all this can be handled seamlessly with the least impact to the user is the subject of much research. This is the concept of a highly available server.

Fault Tolerance

A system becomes fault tolerant when individual computer systems in a cluster begin to share the responsibility of running many processes. Some processes are dependent on others; some are totally independent. Either a local monitoring process or perhaps some feature of the operating system detects the failure of a process and restarts it, either from its initial state or from its most recent checkpoint. A global monitor may try to balance the resource load on each computer by stopping a process on one system and restarting it on another. Cluster-management software can grow very complex, commandeering resources that would ordinarily be used by the mainline service processes. The trade-off is between the reliability and the capacity of the server's primary function.

Feature Interdependence

The astute reader will notice that all these features are, to a great extent, interdependent. The reason that the performance of a high-availability system degrades beyond the extent of its loss, for example, has a lot to do with scalability. A considerable amount of overhead is needed just to monitor for an internal failure, let alone determining and reassigning resources and responsibilities to some other system in the cluster. In league with all these evils is the ever-present threat of failure of a monitoring process while corrective actions are being negotiated—a potential cause of recursive recovery operations! Also, there is a trade-off when applying the cluster's resources to managing the soundness of the cluster, rather than toward its primary goal of serving the user.

Cluster Applications

Keep in mind that when you install Linux onto a bunch of PCs and configure them into a network, you have a network of PCs running Linux—nothing more. To achieve *cluster* status, the PCs must work as a team. You must either make changes to the operating system, which is beyond the scope of this book, or design a cluster-aware application, which is exactly what you'll do here.

Cluster computing applications generally fall into two broad categories: *supercomputing* and *transaction processing*.

Supercomputers

Supercomputers execute complex numerical algorithms in parallel to solve equations. Solutions might generate a graphical projection of weather data patterns, such as software simulations predicting what the weather might be tomorrow. Supercomputers require special programming skills in numerical methods. You must know how to solve a complex equation by a digital approximation algorithm, how to validate that algorithm, and how to detect and exploit any parallelism in it.

Supercomputing applications on cluster architectures are historically successful because a large number of computers will often outperform a single processor of equivalent cost, making a cluster computer the optimum choice. NASA's Beowulf Project, for example, began in 1994 with a 16-node supercomputer, which was assembled from high-end PCs and specialized communication gear for one-tenth the cost of a single computer of equivalent power. (See the section titled "Further Reading" later in this chapter.) Later researchers expanded the number of nodes and worked out the resulting bottlenecks, pushing the point of diminishing returns higher and higher.

Scalability is less important to a supercomputer when the inherent parallelism in an application is limited. Parallelism is usually exploited by scheduling blocks of identical sets of instructions on several nodes, with each block operating on a different subset of the data. If the maximum number of blocks never exceeds seven, the system architect need not bother extending the point of diminishing returns beyond that. Similarly, if the optimum number of available nodes in the system exceeds the maximum number of blocks, the system has a built-in margin of fault tolerance and high availability.

Transaction Processors

The word *transaction* is used here in the general sense to include database queries, financial instructions, and the familiar Web browser requests for file and image data. In this realm, you can exploit parallelism at the transaction level to achieve performance improvement, in contrast to careful examination of parallelism inherent in some complex numerical algorithm, assuming that each transaction is independent of all the others. The transaction contains the block of action codes and data to be processed.

When you direct your browser at a complex Web page, you might have to download a dozen images and text files before that page is completely displayed on your screen. These requests are often independent of one another and can be read from disk simultaneously by each server node, which sends its data to your browser. Your request for a Web page resulted in a list of subrequests to be processed in parallel. Alternatively, a cluster server might support your browser along with many others, on the same simple Web page.

Scalability, fault tolerance, and high availability in the case of an ATM (automatic teller machine), for example, translate into customer satisfaction. Imagine that every one of your bank's branch offices has an ATM installed both inside and out. A few months later, the bank signs a contract with a local supermarket chain, tripling the number of its ATMs in the city. If every transaction on those machines is sent to a central server that was operating at near capacity before the contract, a faster system is in order. Adding nodes to a cluster seems inherently less risky than replacing the single working machine.

The Sample System: The Master-Slave Interface

The focus of this book is to help you design, build, and tune a cluster computer. In Chapter 7, "The Master-Slave Interface Software Architecture," you will see a simple server undergo design changes that morph it into the final design. A master process, which may be started on any of the cluster's nodes, starts slave processes on each of the other nodes. These slaves contact the master, which establishes a set of subtasks that are dedicated to communicating with each of the slaves. The master accepts incoming queries from the clients. It passes the query to an available subtask, which sends it to a remote node's slave for processing. This is how parallelism is exploited in the master-slave interface.

You will discover several ways to measure the performance of the sample system, examine some possible results, and learn how to detect and surmount the bottlenecks. Your personal version will likely present different technical challenges, but the techniques and the concepts you'll learn will probably still help you tune it.

Reader Skills and OS Information

It takes certain skills to program cluster application software. This book assumes that you are a strong C programmer with an exposure to Linux, interprocess communication, and networking. What you need to know about these three areas to build a working cluster system is presented in some detail in this book. The C language cluster software source code is available to you, but for you to customize it confidently, we will present these skills along with short programming examples in C, which are also available on this book's Web page at `www.samspublishing.com`. You are encouraged to compile and run these examples and to experiment with them. This book and all its associated software are intended to provide you with a safe learning environment and a solid learning experience. Enjoy it!

Some of the techniques and software solutions presented here depend on details that can change from one operating system release to another. The examples and cluster software were designed for and tested under Linux kernel 2.0.30, on both i386 and i486. Later releases have slightly different *proc* file structures, and the most recent versions of Linux have improved security mechanisms.

The Remainder of This Book

Now that you have been introduced to cluster computing and some of the fundamental ideas and skills related to it, take a brief look at what lies ahead. The next few paragraphs give you a bit of a road map for your learning journey. Take a moment to see where you will be going in the next few chapters!

Chapter 2, "Multiprocessor Architecture," presents you with a historical perspective on multiprocessing architectures. You will discover some high-level hardware and software alternatives that you might face, and you'll learn about the choices made for the sample architecture used in this book.

Chapter 3, "Inter-Process Communication," presents multitasking, and when you have a multiplicity of tasks, shows how you use interprocess communication (IPC) techniques to make them work together. Following are some of the specific skills

Chapter 3 covers:

- Multitasking with `fork` and the `exec` family of calls
- Signaling other tasks with `signal` and custom signal handlers
- Passing data to other tasks with shared memory and socket connections
- Protecting shared data areas with semaphores
- Differences between TCP and UDP
- Internetworking considerations
- Detecting and handling failures

Chapter 4, "Assembling the Hardware for Your Cluster," focuses on the hardware in a cluster system. You will discover what choices are available and, given the choices made in Chapter 2, how you can put it all together. Because the mechanical assemblage of networking hardware doth not a network make, Chapter 5, "Configuring the Relevant Operating System Files," presents the operating system configuration files of interest and describes what you must do to them to make use of all that networking hardware. You'll also see what benefit networking brings in the form of the remote access r-commands.

By this point of the book, you should have a bunch of networked PCs running Linux. Chapter 6, "Configuring a User Environment for Software Development," presents a user-friendly environment in which to work. In addition to the root user, you will add an application-level user ID and provide access to it and its files from everywhere in the network. You also will see how to organize the cluster software in a convenient structure and how to build it using the make utility.

Chapter 7, "The Master-Slave Interface Software Architecture," presents the cluster software architecture, the inner workings of the sample concurrent server, the interface, and the peer module. You will also uncover the foundations of distributed system support and learn about some limitations of your design.

Chapter 8, "External Performance Measurement/Analysis," opens up some performance issues: What kinds of performance should you be interested in measuring? How can you take these measurements? How can you minimize your measurement's effect on the performance results? You will want your system to meet certain goals, and Chapter 8 presents ways to measure how effectively you are meeting them. When your system might be experiencing performance failures, there are ways to detect where the bottlenecks could lie. You will learn some of these techniques and learn how to interpret and use the results they yield to improve the overall system performance.

Chapter 11, “Further Explorations,” presents the sample system in context and discusses what you might do to change its personality. You saw some different types of multicomputer architectures in Chapter 2. In Chapter 11 you will see what you might do to change the sample system into something else again. Keep in mind that these are experimental ideas, designed to give you a flavor for such systems. In this chapter you will also take a journey into the future of distributed computing. You will explore some computing architectures that may not exist beyond this author’s imagination, and you will see how you might convert the example in this book into one.

Chapter 12, “Conclusions,” wraps up what you learned in earlier chapters, describes the results you achieved from this text—assuming that you build a working model—and reviews what you learned from the process.

This chapter discusses how the working model might be put to productive use in a laboratory. You will also explore distributed algorithms to directly measure the cost/benefits of a particular software, fault tolerance technique. You will also see how to use these methods to turn your server into a test bed for software that is designed to run on some other, perhaps embedded, cluster system that is currently being designed and built in parallel with your software.

Finally, learning doesn’t have to stop with the building of a working example. The system you build has some afterlife and I hope you take it to its limits, beyond this author’s imagination.

Summary

A cluster computer architecture presents a low-cost alternative to a sophisticated multiprocessor system for many applications that are generally associated with supercomputers and transaction processing servers. Furthermore, building a cluster computer is well within the reach of even a computer hobbyist who has solid C language programming skills and a knowledge of computer operating systems, hardware, and networking. This book leads you through the design and assembly of such a system and shows you how to measure and tune its overall performance.

A cluster computer is a multicomputer, a network of node computers running specialized software that allows them to work together as a team. The specialized software turns the simple collection of networked computers into a distributed system. It presents the user with a single-system image and gives the system its personality. It can turn this network of computers into a transaction processor, as described in this book, or it can make it become a supercomputer. Other possibilities exist, too, and these are the subject of speculation near the end of the book.

Some of the techniques used in our distributed algorithms may be new to many readers, so several of the chapters that follow are dedicated to such topics. You will learn about the hardware needed to network a bunch of PCs, the operating system files you must change to support that network, and the multitasking and the interprocess communication skills needed to put the network to good use. Finally, there is a sample application for your experimental use.

The outcome of your following the steps toward actually building and tuning such a system might lead to your advancement, in formal education or at work. This author hopes that you won't let it end here, that you will create some steps of your own, taking these simple concepts into some uncharted territory. Enjoy the journey!

Further Reading

To complement what you are going to learn in this book, a list of recommended titles is given as a means to expand your knowledge of this area even further. Look for a list like this at the end of each chapter:

- See `http://www.linux.org` for further information on the Linux operating system.
- See `http://www.beowulf.org` for more details on NASA's Beowulf Project.
- *Linux Journal*, January 1998, issue 45, is a special issue on Network Clustering.
- Jalote, Pankaj. *Fault Tolerance in Distributed Systems.* Englewood Cliffs, NJ: PTR Prentice Hall, 1994.
- Lyu, Michael R., ed. *Software Fault Tolerance*. West Sussex, England: J. Wiley & Sons, 1995.
- Tanenbaum, Andrew S. *Distributed Operating Systems*. Englewood Cliffs, NJ: PTR Prentice Hall, 1995.

2 Multiprocessor Architecture

Where did the cluster architecture come from? How did it evolve? This chapter describes several kinds of multiprocessors, starting with the classical multiprocessor naming convention and its extensions and ending on a unique and pivotal design that focuses on the system's software rather than on its hardware. The cluster architecture is an outgrowth of that change in thinking—a new paradigm. This chapter discusses many of the hardware and software options a cluster architect must assess; it also describes some transaction-processing system-performance issues. The chapter ends with a discussion about why a cluster computer is best suited to our purposes and an introduction to our sample system.

The cluster computer presented in this book is called a *multicomputer.* It gets its name from the fact that it is built up from many individual computer systems that are interconnected through a communication network. Each of these computers runs processes that must communicate with others across the network, working together to present the single-system image described in Chapter 1, "Linux Cluster Computer Fundamentals."

Most multiprocessors can be built only by computer manufacturers. The multicomputer, however, is one that you can build from readily available components and some custom software. When properly tuned, it can outperform a single computer of equivalent power by exploiting *overlap*, the parallelism inherent in many transaction-processing and supercomputer applications. Let's start with an overview of the classical multiprocessor architectures and follow their trend to the cluster system you will design and build in the later chapters of this book.

Alternative Computer Architectures

The *architecture* of a computer is, literally, "how it is built." To see how computer architectures can differ, it is necessary to understand the building blocks inside a computer, such as the typical desktop computer shown in Figure 2.1.

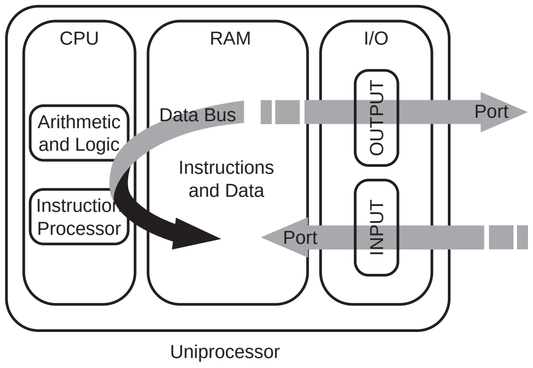

FIGURE 2.1 Uniprocessor—a typical computer and its three functional components.

The three functional units of a typical computer (see Figure 2.1) are instruction processing, memory, and input-output, as described next:

- The *processing* function includes instruction execution, arithmetic/logic operations, and memory data access. These functions are grouped into the processing function because the central processing unit, or CPU, usually performs them all.
- The *memory* function, which *remembers* the data values, is performed by the physical memory and its associated hardware. Memory is where data may be stored after bringing it in from the outside world, where data may be created and altered by a running process, and where it may be picked up for delivery to the outside world.
- The *input-output* function is the process of transferring data between the physical memory and the outside world through an I/O port to a device external to the computer—to a floppy disk, for example.

Different kinds of computer architectures can be created by designing a computer with more than one of these basic components. *Overlap*, or *parallelism*, is the capability to do two or more things at the same time. Instruction-execution overlap can be achieved, not too surprisingly, by having more than one CPU. Memory-access overlap is achieved by having more than one memory data bus, and I/O overlap comes with more than one I/O port. Let's focus on multiple CPUs first, because this is the essence of a multiprocessor.

Overlap with Multiple CPUs

The number of CPUs inside the computer and how they are interconnected form the basis for one multiprocessor naming convention. Four basic kinds of multiprocessors exist, representing the four ways that CPUs and physical memories have been successfully combined into a working computer.

Uniprocessors

A *uniprocessor* (UP) is literally a single-processor computer that contains just one CPU, such as the one shown in Figure 2.1's system. The computer stores instructions and data in physical memory, also called random access memory, or RAM. Provisions exist for getting information into and out of the computer's memory through input and output (I/O) devices, supported by software inside the operating system, or OS. (Don't you just love this alphabet soup?)

Multiprocessors

A *multiprocessor* (MP) has two or more processors, but it has only a single, shared memory that may be accessed by any of its CPUs, as shown in Figure 2.2. With multiple CPUs, more than one program can run at the same time. This allows for parallel execution, provided that software in the OS supports this feature.

This is a double-edged sword, however. You must program these machines carefully, taking into account that two of your instructions might set one of your variables to different values at the same time! (Technically, only one of them will actually happen first; which one of them depends on the wiring and logic associated with the physical memory, but that level of detail is beyond the scope of this book.) You can and should protect shared data on an MP by using *semaphores*, which are discussed in the next chapter.

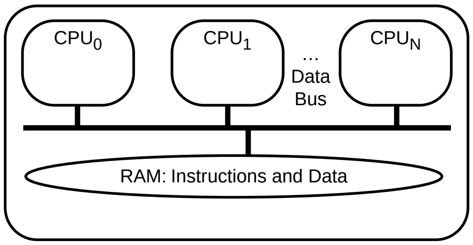

FIGURE 2.2 Multiprocessor—many CPUs interact with a single memory.

Array Processors

An *array* processor is a special-purpose computer with many individual processors, each with its own physical memory area, as shown in Figure 2.3. The benefit of such a machine is its capability to execute the same instruction on all its processors at once. For example, consider an `ADD` instruction. The two numbers to be added together are specified as address offsets into each of its processor's memory areas. Your program gives the same `ADD A to B` instruction to each processor in the array, but each memory area has a different set of numbers at its A and B offsets.

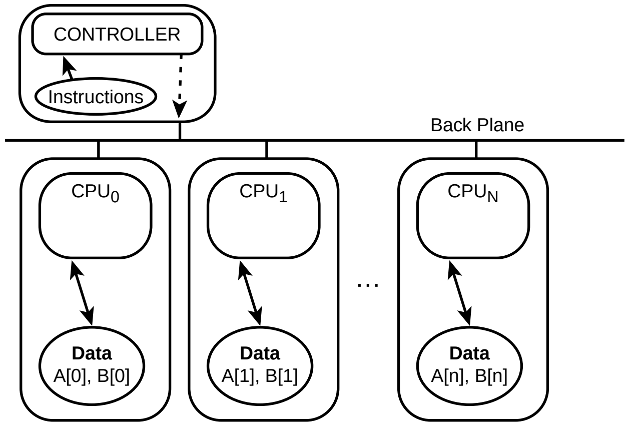

FIGURE 2.3 Array processor—many CPUs interact with special-purpose memories.

Note that a list of numbers is called an *array* in most programming languages; hence the name *array processor*. Mathematicians call a list of related numbers a vector, so you may also see this kind of machine called a vector processor.

If you have two lists of numbers to be added together, A[0] to B[0], A[1] to B[1], and so on, you can give the same `ADD` instruction to every processor. Each value in list A is added to each corresponding value in list B, all in a single instruction cycle! If your application requires many of the same operations on lists of values, this is the kind of computer for you.

Pipeline Processors

A *pipeline* processor is a highly specialized machine that can exploit overlap at the instruction level with only one CPU. The `ADD A to B` instruction, for example, can be viewed as a multistep process:

1. Get the `ADD` instruction from memory.
2. Get the address of variable A from the instruction, and then the value of variable A from memory.
3. Get the address of B from the instruction, and then the value of B from memory.
4. Add the two values, putting the result into the memory location of variable B.

Overlap can be achieved by a pipeline processor by getting the address of variable B from the instruction at the same time it gets the value of A from memory. Pipeline computers fill their pipe with instructions and begin processing all of them at once to maximize the overlap effect. They are often used as supercomputers, in which the speed of instruction processing is paramount. Debugging is especially challenging when you don't know exactly which instruction was being executed when the failure occurred!

NOTE

This author worked with one of the early pipeline processors, an IBM 370 model 195. Execution errors, called program interrupts, were coded so the programmer could tell the kind of error that occurred. Errors often resulted in a listing of memory at the time of the error, called a *dump*, a hexadecimal listing of your entire program space along with its instructions, data, and the address of the next instruction to be executed. Most of the time it was a matter of finding the failing instruction in the dump and maybe looking at the data that was being manipulated by that instruction. When an imprecise interrupt occurred, however, you had to look at perhaps a dozen instructions prior to the one indicated, because they were *all* being executed as far as the system was concerned.

NOTE

The *back plane* gets its name from its location in the back of a cabinet. Computer boards, each with a CPU and memory, slide on rails into the cabinet and plug into that plane in the back, where they can access power and board-to-board data communication lines.

A Taxonomy for Multiprocessors

A *multicomputer* is a multiprocessor that is constructed from many individual computers, which are interconnected via a network or *back plane* (see the following Note for more on back plane). From a hardware perspective, considerable differences exist between a multiprocessor and a multicomputer. They are sometimes referred to as *tightly* and *loosely* coupled multiprocessors, respectively, in reference to the CPU-to-RAM data coupling. From a software perspective, however, they look almost alike; they form a special class, with its own subclass taxonomy, described in more detail next.

In the late 1960s, M. J. Flynn created a simple taxonomy for computing systems to help distinguish among their many architectural designs. He focused on instruction and data access from the perspective of the CPU, noting how many different pathways were available. He asked, "How many instructions can the CPU's instruction processor(s) access at the same time, and how many blocks of data can the CPU's arithmetic and logic unit(s) access at the same time?" The answer was either one or many, in both cases, for a total of four distinct architectures.

The uniprocessor has one pathway to an instruction and a single pathway to data, so he named it the *single instruction, single data* machine, or *SISD*. The MP has multiple pathways both to instructions and data; therefore, it was designated *multiple instruction, multiple data,* or *MIMD*. The array processor has multiple paths to data but only one path to instructions, so it is a *single instruction*, *multiple data* machine, or *SIMD*. Controversy exists about the *multiple instruction, single data*, or *MISD*, designation. Trying to imagine such a construct is a good exercise: multiple instruction pathways with only one data pathway? Some cite the instruction pipeline processor as an MISD. One vintage mainframe, the IBM 360 model 95, is a good example of a pipeline processor.

The MIMD In Depth

The MIMD turns out to be a very useful and interesting type of machine and is the subject of much speculation and experimentation, so much so that the two variants of this basic structure have their own subcategories:

- Uniform Memory Access (UMA)—In tightly coupled MPs, every CPU has direct access to central memory through the same memory data register; each memory access takes approximately the same amount of time (see Figure 2.4).

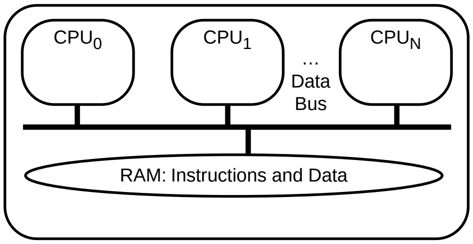

FIGURE 2.4 UMA—uniform memory access times (same as tightly coupled).

- Non-Uniform Memory Access (NUMA)—In loosely coupled MPs, the CPUs have direct access through their memory data registers to their own local memory but not directly to those of the other CPUs (see Figure 2.5). Access to data in their own memory is very fast compared to access to data in another CPU's memory, often over an external data network. Access times are nonuniform across CPUs.

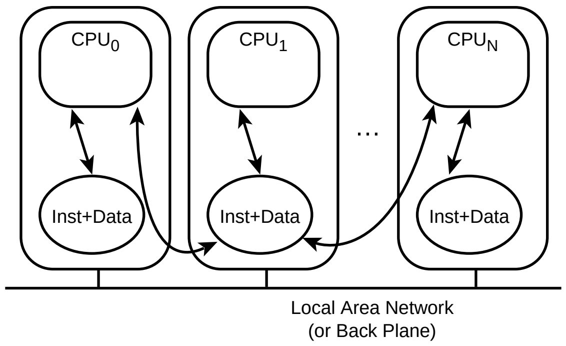

FIGURE 2.5 NUMA—non-uniform memory access times (same as loosely coupled).

Note that in some references, the NUMA is further separated into NUMA and NORMA (No Remote Memory Access through hardware). NORMA systems, these texts explain, require software intervention to catch a remote memory access and then send a message to that system, requesting the required memory data from it. A NUMA differs from a NORMA in that it has its individual systems interconnected with a high-speed back plane, and when a memory address is outside of the local system, the back plane hardware electronically gates the requested data into the memory data register of the requesting CPU. Remote memory access in a NUMA is still much slower than memory access speeds, but the hardware handles everything, eliminating the need for specialized memory access software. Technically, you are going to build a NORMA system. If you do any further reading, it will be more pleasant if you are aware of this minor technical disagreement.

Tightly Versus Loosely Coupled Multiprocessors

Tightly coupled MP systems are very efficient. All their processes can communicate with one another at internal data bus speeds, usually in a nanosecond time frame. Their disadvantage is their high cost and their single point of failure—the central memory and its data bus. Some designs incorporate redundant bus structures and the like, but this chapter considers such classifications in general terms.

A network of computers, such as the multicomputer you are going to design in this book, is called a *loosely coupled* MP because if the MP processes want to communicate with one another, they must do so over their external interconnection media at I/O bus speeds, usually in a millisecond time frame. Interprocessor communication over a network is thousands of times slower than that of a tightly coupled multiprocessor (MP), but it has the advantage of relatively low cost. High availability through fault tolerance may be achieved in software.

Loosely coupled MP systems are difficult to tune because when a process on one computer communicates with a process on another computer, it is likely to be sending or receiving a packet containing control information and data. Control information needs to be examined and data needs to be processed. The ratio of the size of the information packet one process sends to another is a primary measure of efficiency in such systems. If the chosen packet size is too small, many packets may have to be sent to complete a transfer; if the packet size is too large, the receiving process might spend too much of its time being idle, waiting for another packet to process. When the packet is tuned to just the right size, according to relative processor and bus speeds, the next packet arrives just as the previous one is finished being processed—just in time. The trouble is, the correct packet size can vary from application to application and may vary from one point in the same application to another—a function of processor or network loading. This nontrivial problem is the subject of ongoing research.

Tightly coupled MPs are very efficient because the time it takes to transfer an instruction to be executed or for data to be processed is the same for all its CPUs; they all share the same memory. Loosely coupled MPs have a memory transfer bottleneck by design: their CPUs spend a lot of time sending messages to one another over a slow communication network medium. Although switched gigabit ethernet compares well against all but the fastest back plane interconnections, it's still slow when compared to direct memory access speeds. Successful loosely coupled MP systems have their software tuned to the needs of a specific application.

In general, MP systems that have one central memory shared among its processors perform better than those built from many independent node computers on a network, each with its own physical memory. The central memory MP system is far more expensive, however, and a network of computers can compete, performance-wise, if it is carefully tuned to the application for which it is designed.

Remember, too, that when a CPU fails inside a tightly coupled MP, the whole system is sent back to the manufacturer. When one of your network's PCs fails, only that one needs to be repaired or replaced. The system is still there, minus just one PC. Such a benefit can far outweigh the many technical challenges faced by the cluster architect.

You will design, build, and tune a loosely coupled MP in the course of this book. Evidence supports the claim that an 8-to-16-node loosely coupled MP can provide the same or better performance as a tightly coupled MP costing 10 times as much. (See the description of NASA's Beowulf Project at `www.beowulf.org`.) The deciding factor, however, is that designing, building, and tuning a loosely coupled MP is within our grasp; a tightly coupled MP may not be. Consider too, that you can buy the necessary individual processors at a very reasonable price in today's salvage PC market.

Distributed Shared Memory Systems

As part of his doctoral work in 1986, K. Li designed a unique system using PCs on a network. He suggested mapping the concept of a tightly coupled multiprocessor running virtual memory onto the collective physical memory space of a network of workstations.

Each time data was accessed in local memory, it came directly from local memory at memory data bus access speeds. But when data was accessed in a remote PC's memory, however, memory data request and response messages would have to be exchanged before processing could resume. The data address would be unique within the address space of the distributed process. Any process designed and compiled for a tightly coupled MP system could be run without modification on this relatively inexpensive collection of PCs on a simple local area network.

Li called this his IVY system. This class of systems later became known as distributed shared memory, or DSM, systems. DSMs were painfully slow compared to their tightly coupled MP counterparts, but they were very cheap, and they had some very interesting properties worth further exploration. Researchers flooded computer architecture and operating system conferences with DSM papers for many years, and many of those papers have been collected into a book that is cited in the "Further Reading" section at the end of this chapter.

Cluster Architectures

The software running on Li's networked PCs was unique in that it defined the architecture. Subsequent research capitalized on this feature, giving us what we call the cluster system today. Their software gave these otherwise independent PCs unity, defining the system's personality; the software was the soul of the system!

The plethora of research into DSM systems led to the maxim regarding the ratio of packet size versus packet processing time mentioned earlier. It also brought to light many ways to achieve high availability through software fault tolerance. This was certainly a welcome feature considering the dramatic growth in online banking, computerized stock trading, and distributed database servers. Ten years after Li's initial work, when all this computerization was getting to be a very hot topic, the World Wide Web took the online computing community by storm.

Today, a popular commercial Web site's server absolutely must be up and operational 24 hours a day, 7 days a week, or a good deal of money may be lost. A big enough market exists for high availability servers that the software running on each node and the hardware that connects them has become very sophisticated indeed. (And that sophistication is reflected in their cost!)

Our goal is to design and build a cluster system. Let’s look at some of your choices in deciding what hardware to assemble and what software architecture to choose. The following sections of this chapter discuss the hardware decisions, the software features, and wrap up with a discussion about performance and some hardware and software choices made for our target system.

Hardware Options

You need to consider two hardware decisions: the node computer architecture and the interconnection network.

Node Computers

Big name workstations are available for a hefty price. They run sophisticated operating system software, have humungous physical memories and disk drives, and are highly reliable. The principles of designing a cluster system either out of workstations or PCs are pretty much the same. Therefore, this discussion focuses on the typical, Intel-based PC.

The node computers you choose should all be very much alike, for simplicity of replacement, load balancing, and performance analysis. The cluster system used to test the software described in this book has eight node computers with nearly identical processor speeds, memory sizes, and so on.

How powerful should these node computers be? The quick answer might be, “As powerful as you can afford.” However, consider this for a moment. This author’s system grew out of two home computers that were overdue for replacement. Throwing them out seemed rude considering all the years of good service they gave. I still had the original user’s manual, the diagnostics guide, and initialization floppy disks, not to mention all the experience gained working inside the machines, upgrading the memory and hard drives, and so on. When old machines like these hit the salvage market, I bought several more of them at $20 apiece, including a spare machine for parts.

I built a four-node system recently for less than $300. These each had network interface hardware built in to their system boards. They were almost 10 times the speed of the original system’s nodes, and their salvage price was only slightly higher, probably because of inflation. I attached a monitor, a mouse, and a keyboard to only one of them because the network allows remote access to the others, and these newer PCs would successfully boot without one. (Some older machines will fail their power-on self-test if a keyboard is not attached.)

You can get details about computer salvage outlets from large corporations and educational institutions. (Some of these organizations are renting computers now, so their old models go back to the rental agencies. Contact the rental agency directly in that case.) One large university has a warehouse dedicated to selling its obsolete systems to the public and a test bench where you can plug it in to see if it works before you buy. Watch out, however, for missing components; memory and hard drives are often removed prior to salvaging old computers. Be aware, too, that most companies wipe the hard drives clean for security reasons, so bring a boot disk. Call and ask their public information office what, if anything, they offer.

You can buy the latest machines off the shelf, over the telephone, or through the Internet. Before you pay top dollar for anything, you may want to inspect the less-than-latest systems that are two or three tiers down from the top-of-the-line items. These older (some times only six months older) machines are often on sale because the stores want to clear the older machines from its inventories.

Finally, adequate systems are often in the closets and garages of your friends, neighbors, and relatives; these systems also can be found in quantity at swap meets, garage sales, and so on. They might not work when you get them home, and they might not form a matching set, but the original documentation and software is often included in the bargain-basement price, and documentation is invaluable when all you hear are some seemingly random beeps when you plug it in. Should you need it, you can sometimes get documentation, such as PDF files and driver software, from larger manufacturers who have links from their Web sites to their free download areas. One of my test system computers came free from someone who read my computer newsletter ad. Many people have an old computer that runs perfectly. They won't throw it away but would gladly give it away to someone who had a good use for it. Ask, and ye may receive!

Interconnection Networks

Again, you are presented with a wide range of selections, complicated by a large quantity of sophisticated networking equipment, even in modest computer stores. There are firewalls, routers, switches, hubs, and media access cards for speeds of up to a gigabit per second, including the cabling and support software. Equipment that was the pride of enterprise data center managers a few years ago is finding its way into the home network as this book is being written.

A *firewall* can be thought of as a router that lets you control what kind of information passes through it. A router is usually employed as a traffic-isolating connection between two networks. A *switch* is a vastly improved hub, allowing multiple, simultaneous point-to-point communication pathways. A *hub* is a device that employs simple cables (called *twisted pairs*, although they aren't really twisted anymore) in a network that behaves as if a coaxial cable (coax) is being used for the communication medium. And coax got its start by fooling the media access cards into believing that a single radio frequency channel was being used to communicate.

The original implication of the term *ethernet* was a reference to the ether, or atmosphere, through which the data packets were transmitted when the protocol was invented. About 30 years ago, people with several computers on separate Hawaiian islands wanted to communicate with one another. The interisland telephone system was too unreliable to support dial-up communication, so these clever people used a radio frequency (RF) link and designed a new communication protocol to support this new bus topology, called the ALOHA system. Each computer sent its data as a *packet*, a string of bytes with header information that included the sender and receiver's addresses and the number of data bytes; trailer information held information used to verify that the bytes were received as sent. A corrupted packet was said to be the result of a *collision*—two or more data packets transmitted at the same time.

This new protocol was so simple and yet so powerful that computer scientists took notice. They thought about the possibilities that this new computer-to-computer communication technique might open up. Following is some more history.

A radio transceiver (transmitter-receiver) is usually connected to its antenna over a convenient length of transmission line, often a coaxial cable designed to match the transceiver's output characteristics to the antenna's, hopefully with minimum loss of signal strength. The radio sits on a desk in the radio room, but the antenna is up high on the roof or maybe even higher up on a tower next to the radio room. The transmission line makes this separation possible. Every node in the original experiment had its own transceiver, transmission line, and antenna system attached to its computer and had some custom software to control sending and receiving data packets.

A later group of scientists decided to eliminate the antennas, connecting all the transceivers to a single coax; if the distance between computers was not an issue, the same protocol should work for a *local area network*, or LAN. Ethernet was born.

Coax got smaller and cheaper and the same concepts worked almost as well using much simpler and cheaper hardware connectors. (It was dubbed thinnet, or cheapernet, and the original-size coax networks became thicknet.) It was still coax, however, and it was costly to "pull cable" through buildings built before any computers existed. (This author has spent more than a few hours in ceilings and elevator shafts in support of such efforts!) Most office buildings had miles of telephone voice-grade wiring in them already, usually many more "pairs" than were necessary to support the internal telephone network. (Bundles of pairs of wires, twisted together, were often available at each wall plate. One pair of color-coded wires supported a single telephone set; hence, the phrase *twisted pair*.) If only someone could figure out a way of using all this existing wire!

It wasn't hard to redesign the media access card to match its characteristics to a pair of wires. It worked fine as a point-to-point link, but it seemed impossible to use this in a bus network in which each of the computers could communicate with any of the others, as long as the rules of the protocol were preserved. Enter the concentrator device, later called a network hub.

A *hub* is a small device that supports a fixed number of connections to its ports, one per node on the network (see Figure 2.6). Although the physical appearance of the network's connections looks like a star (all nodes radiating from some central point), the behavior is that of ethernet, a bus topology that allows direct communication between any two nodes. The electronics inside a hub listen to each of its ports, retransmitting any signal it hears on each of the other connections. This behavior simulates the behavior of a bus network.

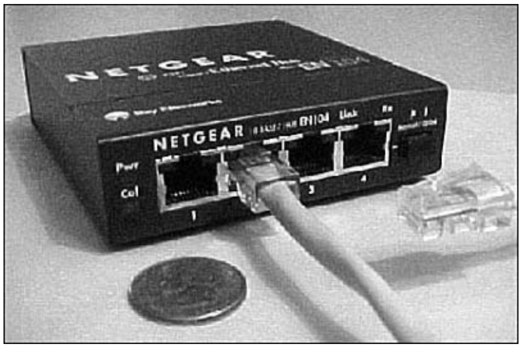

FIGURE 2.6 A 10BASE-T hub connects many network interface cards together over ordinary wires.

A naming convention exists for networks and their connectors. The original half-inch thick coax networks supported a single piece of coax up to about 500 meters and operated first at a 1-, then a 2-, and finally a 10-megabit data rate. These were designated 1BASE-5, 2BASE-5, and 10BASE-5, respectively. The first number represents the data rate in megabits per second (Mbps). Base refers to base band communication, in which only one carrier frequency is used. (Contrast this to broadband communication.) A final number represents the approximate maximum coax length in hundreds of meters. Coax connectors are commonly called UHF connectors because they are rated for frequencies into the UHF (300MHz and up) range before they exhibit noticeable losses. (Technically these are called PL-259 connectors: PL for plug, which matches to the SO-239 socket.) Thinnet uses BNC connectors that are easy to connect or disconnect. They both require a terminator at each end to make them appear electrically infinite in length. The terminator is a resistor that absorbs the transmitted energy, avoiding interference-causing reflections.

The phrase *twisted pair* comes from the telephone company, so the common household four-wire RJ-11 telephone-to-wall connector was the model for a larger eight-wire RJ-45. My test system's network uses RJ-45 connectors into a 10BASE-T hub. The final T is a reference to twisted pair. 100BASE-T is used to describe 100Mbps, and 1000BASE-T refers to gigabit hubs and networks. You may see some less obvious designations for fiber optic support and the like in your further readings.

In summary, the best solution is the fastest solution—hottest node processors interconnected via the speediest network. This book isn't about building a solution to any specific problem as much as it is about getting into the thick of cluster computer architectures, however. Six-month-old technology is good enough for serious developers; hobbyists will be happy with a 100Base-T hub connecting four or more salvaged PCs, one of which has a monitor, a keyboard, and a mouse attached. This is the philosophy behind the test system, and it developed into quite an adequate system, as you will see.

Software Options

The software choices are simpler. You want an operating system that is sophisticated enough to support a local area network, remote command execution (remote shell) access from one machine to any other on the network, and a reliable C compiler.

Unix has all the required features, and several versions are available for the PC. The cost of the OS software goes up with features, not quality, so you have an advantage because you will not need many. A sophisticated Linux user can download, configure, build, and install the OS for no cost beyond the download time on an Internet connection. For the rest of us, a simple version of Linux is available for less than $50, including basic documentation and a user-friendly installation program. The test system runs the Linux 2.0.30 kernel in character mode (no X Window support). If you are going to attach only one monitor, keyboard, and mouse, you may want to include the X Window System, which will enable you to open an individual terminal window to each of the machines on that single monitor.

Linux comes with a good C compiler, unlike many other operating systems. Linux is certainly the best choice. The security is one Linux feature that gets more sophisticated with time, so older versions are preferred over newer ones. Security within your local area network is not an issue—you are not going to attach anything to the Internet or elsewhere outside your direct control.

Performance Issues

You can look at a cluster's performance in two ways: externally and internally. The external viewpoint focuses on the rate at which the cluster can handle incoming transactions for a server. It looks at how quickly high-level calculations can be performed for a supercomputer. The internal view gives the view of how long it takes to do whatever the cluster is designed to do, in detail. For example, an incoming transaction is accepted by the master and then passed along for further processing by one of the slaves. The response is built and returned to the master, where it is sent back to the requestor. Many steps are associated with this process. You can (and will in a later chapter) time the execution of each of its phases and each of the steps in each phase if you are so moved. You can plot these times as a bar graph to see where most of the time is being spent and then take a close look at that code, optimizing it to reduce the overall response time. This is a direct application of principles that are often applied to software optimization: time each function, and then time each high-level instruction of the long-running functions to get an idea where the execution time is being spent; then you can optimize that code if possible. Distributed systems can be analyzed as if they were an application, but they are more complex because of the variance in their interprocess communication times on heavily loaded networks.

Following are two maxims of system optimization: time is almost always being wasted somewhere other than where you expect, and (the good news) initial efforts are often rewarded with huge gains. As with newly written large application programs, a close look at the exact way in which calculations are made, for example, and a little effort toward optimizing those methods can reduce the overall execution time dramatically. You should find that the same rewards await you when you optimize your cluster, but it will take a much greater effort on your part to get the timing data because these tools are not generally available at this writing. Fret not, however, because later chapters will guide you through a custom development of tools specific to your needs.

Our Cluster System's Architecture

The system presented as the example for this book went through several evolutionary phases. The goal was to keep it simple and keep it cheap, without sacrificing any potential learning experiences. It has solid system architecture, both in hardware and in software, and so will yours if you build your system as you follow along in this book. Advanced readers, or those with special objectives, may want to use these examples as guidelines and embellish as they go along.

The hardware chosen consisted of salvaged PC base units, a simple SVGA color monitor, and an ordinary keyboard and mouse. The four-node processors were identical at 166MHz; each had 64MB RAM, a 1GB hard drive, a 1.44MB floppy, a CD-ROM, and a $30 price tag. They had 10Base-T built into their system boards, so I purchased a four-port matching hub and cabling for less than $70. The monitor, keyboard, and mouse came in at about $50. The commercial Linux OS installation package, the only purchased software, brought the total to just under $300, including sales tax.

The cluster system software was designed with simplicity in mind. We expect the reader to learn by doing what we describe, but it is our sincerest hope that you reach beyond what we offer, trying alternatives that spring to mind and comparing your performance results to ours.

The master-slave concept is a common approach to distributed processing architecture design. One node is chosen as the master, receiving incoming queries from some external client base. Each transaction is passed along to a slave node for processing, freeing up the master to continue receiving input. The basic concept is diagrammed in Figure 2.7.

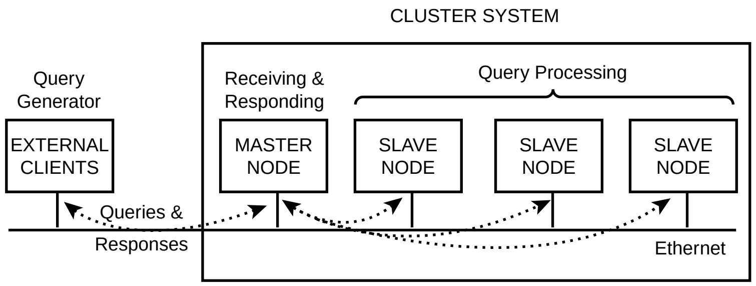

FIGURE 2.7 Cluster System—master-slave software architecture is accessed by external clients.

Summary

This chapter began by examining the components of a typical computer and then looked at some of the other kinds of computer architectures built with multiple CPUs, multiple memories, or some specialized hardware. The introduction of the desktop computer and the LAN, or local area network, led to the design of the multicomputer, a network of computers running custom software. You saw how the nature of this software defined the properties of such machines, making them suitable for transaction processing, supercomputing applications, and the like.

The simplest multicomputer construct is the master-slave architecture, in which one of the network machines uses the others according to their availability to exploit any parallelism inherent in the data processing algorithm. The design of such a system is the focus of this book.

Further Reading

- Hennessey, John, and David Patterson. *Computer Organization and Design*. San Francisco, CA: Morgan Kaufmann, 1998.
- Patterson, David, and John Hennessey. *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.
- Protic, Jelica, Milo Tomasevic, and Veljko Milutinovic. *Distributed Shared Memory: Concepts and Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1998.
- Tanenbaum, Andrew. *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall PTR, 1996.

3 Inter-Process Communication

IN THIS CHAPTER

Even seasoned software developers—solid C programmers with expert knowledge of Unix internals—may not feel completely comfortable using some of the more esoteric techniques presented in this chapter. This chapter is a review of the C-Unix interface skills used in the development of our cluster software. Please feel free to skip any one or all of the following sections if you feel confident about this material. These examples are presented for your convenience, along with a complete set of sample C programs that you can compile and run. The source code can be found on this book's Web page at `www.samspublishing.com`. We hope you will then modify and rerun them to satisfy your own curiosity about details that may not have been mentioned!

Note that detection and handling of error returns from many of the system calls in these examples is intentionally omitted to keep the program listings short and therefore more readable. Each of these sample programs was tested under the Linux 2.0.30 kernel in Red Hat version 4.2.

Subtasking with `fork` and `execl`

In the C-Unix system interface, subtasking requires that you make a system call to create an identical copy of your original (parent) process. When it returns, your subtask (child) is running in its own address space, with its own process ID, or PID. Now two identical processes exist that both experience a return from that same system call; they must discover for themselves whether they are to play the role of parent or child, and the return code is the only way to tell the difference. The parent process receives the process ID of its child subtask and the child receives zero. The following program (see Listing 3.1) demonstrates this technique:

LISTING 3.1 Single Module Subtasking Using `fork`

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

main(void)
/*
**   Listing3.1.c - single module subtasking using fork()
*/
{
   int pid;   /* child task's process id */

   /* Parent task sends a process id message to the screen */
   printf("\nParent task active as process %d.\n", getpid());

   /* fork() creates an identical copy of the parent task */
   if((pid = fork()) == 0)
   {
      /* This block is ONLY executed by the subtask */

      /* Child task sends a process id message to the screen */
      printf("Child task active as process %d.\n", getpid());

      /* Child task exits normally */
      exit(0);
   }
   /* Parent awaits child task exit */
   waitpid(pid);
   return;
}
```

Generally, only one program is written, with support for both roles. Another paradigm exists, however. The child process (subtask) support code is kept simple, issuing a system call that specifies the disk location of a separate executable. This other executable contains the complexity intended for the child subtask. It begins execution, completely replacing the original child subtask's image in memory and assumes its process ID. Note these items in Listings 3.2a and 3.2b.

NOTE

The execl function requires a full path name of an executable. Replace this string with one describing your test environment.

LISTING 3.2A Parent: Subtasking Using fork and execl

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

main(void)
/*
**   Listing3.2a.c - parent: subtasking using fork() and execl()
*/
{
   int pid;   /* child task's process id */

   /* Parent task sends a process id message to the screen */
   printf("\nParent task active as process %d.\n", getpid());

   /* fork() creates an identical copy of the parent task */
   if((pid = fork()) == 0)
   {
      /* This block is ONLY executed by the subtask */

      /* Child task replaces itself with disk image of child */
      execl("/home/user/bin/Listing3.2b", "Listing3.2b", NULL);
      printf("A successful execl() call will never return!\n");
      exit(-1);
   }
   /* Parent awaits child task exit */
   waitpid(pid);
   return;
}
```

LISTING 3.2B Child: Subtasking Using fork() and execl()

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

main(void)
/*
```

LISTING 3.2B Continued

```
**   Listing3.2b.c - child: subtasking using fork() and execl()
*/
{
   /* Child task sends a process id message to the screen */
   printf("Child task active as process %d.\n", getpid());

   /* Child task exits normally */
   exit(0);
}
```

The two system calls are to `fork` and `execl`. (Note that `execl` is actually one of a family of similarly named functions. This book focuses on the *`execl`* variant, which is similar to the familiar command line syntax used to start a process.) A call to `fork` creates the second identical process, a clone of the parent. A call to `execl` causes the calling process to be completely replaced, overwritten in memory by the specified executable. Keep in mind that there is no return to the calling program from a successful `execl` call!

You may want to develop a single module that includes support for both the parent and the child subtask. In that case, the return value from the `fork` call is used to control which section of the module will execute next. A `fork` call always returns the child's process ID to the parent and a zero to the child. (See Listing 3.1. If you compile and execute this program, you will notice that parent and child print different process IDs.)

Alternatively, you may want to develop two separate modules. The return value from the `fork` call is used by the child process in deciding to make a call to `execl`. The first module, Listing 3.2a, looks very much like the single-module version in Listing 3.1, but there is a simple system call to `execl` where the subtask code would be. That subtask code, simple as it is, has been encapsulated into a module of its own, as shown in Listing 3.2b. You should notice two different process IDs from this version, too. In fact, the behavior should be indistinguishable from the single module, `fork`-only version. Generally, if the child code is very complex or if the same child is intended as a subtask of more than one parent, separate them into two modules and use the `fork` and `execl` call paradigm.

In either case, these two processes are your responsibility. The parent process knows about the child process because the fork call returns the subtask's process ID to the parent. It should wait for its child subtask to exit before it does, via a call to `waitpid`, as in the example.

NOTE

Parent and child processes often communicate with one another by design, and there are many ways for them to do so. When one active process communicates with another, it is called *interprocess communication*, or IPC. IPC is the subject of much of the rest of this chapter.

Sending Signals and Handling Received Signals

One process may communicate with another in several ways. You will learn about some of these ways, in detail, later in this chapter. Certain events happen so regularly that they have been condensed into a set of numerically coded messages, called *signals*. A signal can be described as an indication of the occurrence of one of these common events. A signal may come from an ordinary process, the operating system, the command line (via the `kill` command), or directly from the keyboard as a special combination keystroke (sometimes called a *chord*), as in our example. Use the `kill -l` Unix command (the process-kill command with the list option) to display a complete listing of all 32 signals.

One signal that is easily described is `SIGINT` (signal an interrupt, event number 2), which may be sent directly from the keyboard by an interactive user. When you run the program in Listing 3.3, a message is printed on the screen instructing you to `Type a ^C when ready`. (Type the letter C while holding down the Ctrl key to effect a ^C.) Watch what happens when you do. After three such events, the demonstration terminates.

LISTING 3.3 Handles the Signal from a Ctrl+C Typed at the Keyboard

```
#include <sys/signal.h>
#include <stdio.h>
#include <unistd.h>

int count = 0; /* counter for the number of ^C events handled */

main(void)
/*
**   Listing3.3.c - handles signal from ^C typed on keyboard
*/
{
   void control_C();

   /* initialize ^C handler */
   signal(SIGINT, control_C);

   while(count < 3) {
```

LISTING 3.3 Continued

```
      /* tell interactive user what to do */
      printf("\n\tType a ^C when ready...");
      fflush(stdout);
      pause();
      printf("handled event #%d.\n", count);
   }
   printf("\nThree events were handled - bye!\n\n");
}

void control_C()
{
   /* reestablish handler */
   signal(SIGINT, control_C);

   /* count ^Cs */
   count++;
}
```

After the program is running, you can send the signal via the `kill` command and the event code for `SIGINT`:

```
kill   -2   3721
```

in which 2 is the `SIGINT` event code and 3721 is used here in place of the actual process ID. You will have to use another session window (per the Linux documentation) or remotely log in from some other networked machine to try this. You can also send a signal from another process, using the `kill` system call:

```
kill(2, 3721);
```

in which 3721 is assumed to be the PID again.

A header file contains definitions for SIGINT (2), SIGUSR1 (10), SIGUSR2 (12), and so on, so you can code these easier-to-remember definitions instead of their numeric code values. Coded events numbered 10 (SIGUSR1) and 12 (SIGUSR2) are generic and are never sent from the OS. Use one or both of these when you want to define the application-specific meanings of your own signals.

Using Shared Memory Areas

Remember that a subtask created by a call to `fork` runs in its own address space, which is an exact copy of that of the parent process. Every variable is set for the child process to whatever it was when the parent process called `fork`. It also means those values do not change in the child's address space when the parent changes them, or vice versa.

This is an important concept: If the child process changes the value of a variable, the parent cannot see it because they are now two different values, in two different variables, within two different address spaces. When the child executes, it sees only its address space. A change in one has absolutely no bearing on the state of the other; they are not related in any way.

If the child process needs to communicate with the parent by changing the value of a variable, for example, how can it do that? Shared memory is one answer. The OS provides a set of system calls to allow one task to allocate shared memory. The parent needs only to identify it and set appropriate permissions to allow the child to attach it, mapping it into its own address space. (A shared memory area may be attached and modified by any process that has permission to do so, by the way, so set the permissions on your new memory area carefully. Shared memory is not just for subtasks, but may be accessed by any other process running on the same computer.) Let's look at Listing 3.4 as an example.

LISTING 3.4 Child Changes Data in Parent's Shared Memory

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>

main(void)
/*
**   Listing3.4.c - child changes data in parent's shared memory
*/
{
   struct area {  /* shared memory area */
      int value;  /* (a simple example) */
   } area, *ap;

   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = sizeof(struct area);
   key_t key = 0x01020304; /* example */
   int shmid; /* shared memory area id */
   int pid;   /* child task process id */

   /* get a shared memory area */
   shmid = shmget(key, size, flag);

   /* attach the shared memory area */
```

LISTING 3.4 Continued

```
   ap = (struct area *) shmat(shmid, 0, 0);

   /* set the value of the shared memory data */
   ap->value = 0; /* initialize value to zero */
   printf("\nParent initial value set to zero\n");

   if((pid = fork()) == 0)   /* SUBTASK */
   {
      /* attach parent's shared memory */
      ap = (struct area *) shmat(shmid, 0, 0);
      /* note: shmid is still set correctly */

      sleep(2); /* let parent brag a while */

      /* change the value of the shared memory data */
      ap->value = 1; /* something other than zero */

      /* child task exits normally */
      exit(0);
   }

   /* watch for value to change */
   while(1)
   {
      printf("Parent: value = %d,", ap->value);
      fflush(stdout);
      if(ap->value != 0)
         break;
      printf(" continuing...\n");
      sleep(1);
   }
   printf(" Bye!\n\n");

   /* parent awaits child task exit */
   waitpid(pid);

   /* release the shared memory area */
   shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);

   return;
}
```

NOTE

Note that calls to sleep are added just to slow down everything for effect. Without the calls to sleep, everything happens so fast that the sequence of events might not be so obvious. Try running it without the sleep calls. The more you experiment, the more you will learn and the longer you will retain the material.

Using Semaphores with Shared Data

The shared data example in Listing 3.4 doesn't quite illustrate the risk associated with shared variables. Consider a similar example in which the parent bank account manager creates two subtasks: one child process responsible for deposits and the other for withdrawals, each calculating the new balance as part of its processing. Can you see the risk?

If two transactions arrive at approximately the same time—one a deposit, the other a withdrawal—a possibility exists, albeit a small one, that one of them will be nullified. Consider the case in which the withdrawal process gets the balance from shared memory, subtracts the withdrawal amount from it, and is then interrupted by the deposit process. The deposit process picks up the balance from shared memory (the same one that the withdrawal process still has), adds in the deposit, puts the balance back into shared memory, and then waits for another deposit transaction. The withdrawal process continues processing, putting its recalculated balance into shared memory, over the value that the deposit process set. The deposit transaction is lost because all the actions of the deposit process were nullified by that interrupt.

The research concept of interest here is the *critical section* of a process. How can you be certain that after a process starts its processing by picking up a balance from shared memory, it will not be nullifying another transaction when it puts a recalculated value back? You can protect the integrity of shared data by using *semaphores*.

What's a semaphore? First, consider what it is not: A semaphore does not stop another process from modifying shared data, no more so than a red traffic light prevents you from entering an intersection. We go on green, slow on amber, and stop on red by agreement. This holds true for semaphores, too.

Each semaphore is associated with a shared resource, not directly, but indirectly through the source code comments and other documentation. A semaphore is either set or not, and that's all the operating system cares about. If you try to set a semaphore and it's not set, the OS will set it. If you try to set a semaphore that is set, the OS will cause your process to wait until the last process to set it clears it.

To avoid any difficulties with the bank transaction example, both processes (deposit and withdrawal) must set a semaphore prior to getting the balance from shared memory and then clear it after the recalculated balance is put back (see Listing 3.5).

This is sometimes called *serializing* on a resource, meaning that any actions performed on a resource must happen one at a time, in sequence. (You can find more on this in texts included in the "Further Reading" section at the end of this chapter.)

LISTING 3.5 Simultaneous Handling of Deposits and Withdrawals

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <unistd.h>

main(void)
/*
**   Listing3.5.c - simultaneous handling of deposits and withdrawals
*/
{
   struct area {  /* shared memory area */
      int balance;
      int deposit;
      int withdrawal;
   } area, *ap;
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)\
   /* union semun is usually defined by including <sys/sem.h> */
#else
   /* otherwise we have to define it ourselves, per <bits/sem.h> */
union semun
{
   int val;                    /* value for SETVAL */
   struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
   unsigned short int *array;  /* array for GETALL, SETALL */
   struct seminfo *__buf;      /* buffer for IPC_INFO */
};
#endif
   union semun un; /* semaphore union and buffer */
   struct sembuf buf;

   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = sizeof(struct area);
   key_t skey = 0x04030201; /* sample keys */
   key_t mkey = 0x01020304;
```

LISTING 3.5 Continued

```
   int semid; /* semaphore id */
   int shmid; /* shared memory id */
   int pidD;  /* deposit child task pid */
   int pidW;  /* withdrawal child task pid */
   int amount;
   char inbuf[80];

   /* get a shared memory area */
   shmid = shmget(mkey, size, flag);

   /* attach the shared memory area */
   ap = (struct area *) shmat(shmid, 0, 0);

   /* initialize account info in shared memory */
   ap->balance = 1000; /* initialize balance to $1000 */
   ap->withdrawal = 0;
   ap->deposit = 0;

   /* get and initialize a semaphore */
   semid = semget(skey, 1, flag);
   /* set semaphore available */
   un.val = 1;
   semctl(semid, 0, SETVAL, un);
   /* first and only semaphore */
   buf.sem_num = 0;
   /* wait if set */
   buf.sem_flg = 0;

   if((pidD = fork()) == 0)   /* DEPOSITS */
   {
      /* attach parent's shared memory */
      ap = (struct area *) shmat(shmid, 0, 0);
      /* note: shmid is still set correctly */

      /* handle deposit */
      while(1)
      {
         sleep(1);

         /* set semaphore */
         buf.sem_op = -1;
         semop(semid, (struct sembuf *) &buf, 1);
```

LISTING 3.5 Continued

```
         if(ap->deposit < 0)
            break; /* exit req */

         if(ap->deposit > 0)
         {
            /* handle the deposit */
            ap->balance += ap->deposit;
            ap->deposit = 0; /* clear it */
         }

         /* clear semaphore */
         buf.sem_op = 1;
         semop(semid, (struct sembuf *) &buf, 1);
      }

      /* child task exits normally */
      exit(0);
   }

   if((pidD = fork()) == 0)   /* WITHDRAWALS */
   {
      /* attach parent's shared memory */
      ap = (struct area *) shmat(shmid, 0, 0);
      /* note: shmid is still set correctly */

      /* handle withdrawal */
      while(1)
      {
         sleep(1);

         /* set semaphore */
         buf.sem_op = -1;
         semop(semid, (struct sembuf *) &buf, 1);

         if(ap->withdrawal < 0)
            break; /* exit signal */

         if(ap->withdrawal > 0)
         {
            /* handle the withdrawal */
            ap->balance -= ap->withdrawal;
            ap->withdrawal = 0; /* clear it */
         }
```

LISTING 3.5 Continued

```
         /* clear semaphore */
         buf.sem_op = 1;
         semop(semid, (struct sembuf *) &buf, 1);
      }

      /* child task exits normally */
      exit(0);
   }

   /* parent: handle deposit and withdrawal transactions */
   printf("\n\n\n\tWELCOME TO THE FIRST INTERACTIVE BANK\n\n");
   while(1)
   {
      printf("\nEnter D for deposit, W for withdrawal: ");
      fflush(stdout);
      fgets(inbuf, sizeof(inbuf), stdin);
      if(inbuf[0] == 'D' || inbuf[0] == 'd')
      {
         printf("\tCurrent account balance is $%d\n", ap->balance);
         printf("\tEnter deposit amount (0 to exit): ");
         fflush(stdout);
         fgets(inbuf, sizeof(inbuf), stdin);

         /* set the semaphore */
         buf.sem_op = -1;
         semop(semid, (struct sembuf *) &buf, 1);

         amount = atoi(inbuf);
         if(amount <= 0) /* exit requested */
         {
            /* signal subtasks */
            ap->deposit = -1;
            ap->withdrawal = -1;
            break; /* exit infinite loop */
         }
         ap->deposit = amount; /* deposit it */

         /* clear semaphore */
         buf.sem_op = 1;
         semop(semid, (struct sembuf *) &buf, 1);
      }
      else if(inbuf[0] == 'W' || inbuf[0] == 'w')
      {
```

LISTING 3.5 Continued

```
        printf("\tCurrent account balance is $%d\n", ap->balance);
        printf("\tEnter withdrawal amount (0 to exit): ");
        fflush(stdout);
        fgets(inbuf, sizeof(inbuf), stdin);

        /* set the semaphore */
        buf.sem_op = -1;
        semop(semid, (struct sembuf *) &buf, 1);

        amount = atoi(inbuf);
        if(amount <= 0) /* exit requested */
        {
           /* signal subtasks */
           ap->deposit = -1;
           ap->withdrawal = -1;
           break; /* exit infinite loop */
        }
        else if(amount <= ap->balance)
        {
           ap->withdrawal = amount; /* withdraw it */
        }
        else
        {
           printf("ERROR: Insufficient funds!\n");
        }

        /* clear semaphore */
        buf.sem_op = 1;
        semop(semid, (struct sembuf *) &buf, 1);
     }
     else
     {
        printf("Invalid transaction code '%c'\n", inbuf[0]);
     }
  }

  /* await child exits */
  waitpid(pidD);
  waitpid(pidW);

  printf("\nYour final account balance is %d\nBye!\n\n", ap->balance);
```

```
    /* remove shared memory, semaphore, and exit */
    shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
    semctl(semid, 0, IPC_RMID, un);
    return;
}
```

Messaging: UDP Versus TCP

Signals allowed you to send basic information from one process to another, assuming you already had or knew how to obtain the target process ID. What if the *information* you want to send is more complex—a database query, for example? A more general way to send information is called *interprocess communication*, or IPC, or often just *messaging*. Two popular message protocols exist, and they mimic how we communicate as humans. They are called *user datagram protocol*, or UDP, and *transmission control protocol,* or TCP. Collectively, they are referred to as *socket* programming; you can find more information listed in the "Further Reading" section at the end of this chapter.

Both protocols require a message sending and receiving address, called the *name* of the socket. Remember that a signal was associated with a process, so the PID was a sufficient signal target address. You can have several communication sessions per process, so a PID is too vague. The same socket name can be used by a process that terminates and then restarts with a new PID, so the PID can be misleading. Finally, the OS picks the PID, but a socket name must be predictable if a process on one machine is to send a message to its colleague on another. The well-known socket name solves all these problems nicely.

IPC with UDP

UDP takes the concept of a signal to a higher level. When one process sends a signal to another, it has no feedback and no indication that the signal was received, or if it was received, that it was handled, and so on. UDP is a one-way messaging protocol that can be used for simplex transmission, in which one process sends information to another without regard or concern for reply. This is how we humans might send a postcard home while on vacation.

It can be used for half-duplex transmission, too, in which one process sends a request for information to another, and that process responds accordingly. Half-duplex communication is more complex, involving allowances for lost requests and lost or duplicate responses, similar to how publishers might send magazine subscribers a request for renewal.

Allowance for error conditions, also called *fault tolerance*, was anticipated and included in a more sophisticated, connection-oriented protocol—TCP. UDP is said to be connectionless to distinguish it from TCP, which opens a point-to-point connection before any communication can begin. TCP is presented next.

IPC with TCP/IP

TCP is often likened to a telephone conversation. The initiating process connects to the recipient, which is awaiting connection. (This is similar to when a caller dials the number of a telephone that is on-hook, ready to receive a call.) After the connection is established, messages can be sent in either direction, much like two people talking to each other. It's called full-duplex communication and because computers are doing the communicating, the back-and-forth protocols are necessarily more formal than what humans might use.

The good news about TCP is the strong, two-way connection. Any packets sent are tracked, timed, and re-sent, if necessary, to ensure delivery. The bad news about TCP is that the strong, two-way connection isn't always appropriate in a cluster architecture, in which the communication is often one-to-many. A lot of overhead occurs in all that packet delivery management software as well. Real-time and high-bandwidth applications often avoid TCP because there is too much overhead, which prevents work from being done on time. A good compromise in these circumstances is to use UDP with custom fault tolerance that adds only a small amount of overhead—only what is necessary to meet your system's requirements.

To illustrate these two protocols, two nearly identical programs are shown in Listings 3.6 and 3.7. Note that some of these features may not be supported in later releases.

LISTING 3.6 UDP Communication with a Forked Subtask

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char path[] = {"/tmp/socket3.3.6"};   /* socket name */

main(void)
/*
**   Listing3.6.c - UDP communication with a forked subtask
*/
{
   struct sockaddr_un sock;
   int len=sizeof(sock);
```

LISTING 3.6 Continued

```
   int pid;   /* child task's process id */
   int fd;    /* socket file descriptor */
   char buffer[80];

   /* establish and initialize UDP socket struct */
   fd = socket(AF_UNIX, SOCK_DGRAM, 0);
   memset((char *) &sock, 0, sizeof(sock));
   strcpy(sock.sun_path, path);
   sock.sun_family = AF_UNIX;

   /* create child subtask */
   if((pid = fork()) == 0) >
   {
      /* publish the port number we are listening to */
      bind(fd, (struct sockaddr *) &sock, len);

      /* echo messages received from parent back to user */
      while(1) {
         recvfrom(fd, buffer, sizeof(buffer), 0,
          (struct sockaddr *) &sock, &len);
         printf("Child: %s", buffer);
         if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
         {
            unlink(path);
            puts("Bye!");
            close(fd);
            break;
         }
      }

      /* Child exit */
      exit(0);
   }

   /* send a message typed from interactive user to child subtask */
   while(1)
   {
      sleep(1); /* response time */
      printf("\nEnter a message: ");
      fflush(stdout); >
      fgets(buffer, sizeof(buffer), stdin);
      sendto(fd, buffer, strlen(buffer)+1, 0,
       (struct sockaddr *) &sock, sizeof(sock));
```

LISTING 3.6 Continued

```
      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         close(fd);
         break;
      }
   }

   /* await Child exit */
   waitpid(pid);
   return; >
}
```

LISTING 3.7 TCP Communication with a Forked Subtask

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char path[] = {"/tmp/socket3.3.7"};   /* socket name */

main(void)
/*
**   Listing3.7.c - TCP communication with a forked subtask
*/
{
   struct sockaddr_un sock;
   int len=sizeof(sock);
   int pid;   /* child task's process id */
   int soc;   /* socket file descriptor */
   char buffer[80];

   /* establish and initialize TCP socket struct */
   if((soc = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
   {
      perror("Listing3.7: socket()");
      exit(-1);
   }
```

LISTING 3.7 Continued

```
memset((char *) &sock, 0, sizeof(sock));
strcpy(sock.sun_path, path);
sock.sun_family = AF_UNIX;

/* create child subtask */
if((pid = fork()) == 0)
{
   /* publish the socket name we listen to */
   bind(soc, (struct sockaddr *) &sock, len);

   /* accept connection */
   if(listen(soc, 5) < 0)
   {
      perror("Child: listen()");
      exit(-1);
   }
   if((soc = accept(soc, (struct sockaddr *) &sock, &len)) < 0) {
      perror("child: accept()");
      exit(-1);
   }

   /* echo messages received from parent back to user */
   while(1)
   {
      if(recv(soc, buffer, sizeof(buffer), 0) < 0)
      {
         perror("child: recv()");
         break;
      }
      printf("child: %s", buffer);
      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         unlink(path);
         puts("Bye!");
         close(soc);
         break;
      }
   }

   /* Child exit */
   exit(0);
}
```

LISTING 3.7 Continued

```
   /* connect to server socket created by our child subtask */
   sleep(2); /* wait for setup */
   if(connect(soc, (struct sockaddr *) &sock, sizeof(sock)) < 0)
   {
     perror("parent: connect()");
     exit(-1);
   }

   /* send a message typed from interactive user to child subtask */
   while(1)
   {
      sleep(1); /* response time */
      printf("\nEnter a message: ");
      fflush(stdout);
      fgets(buffer, sizeof(buffer), stdin);

      if(send(soc, buffer, strlen(buffer)+1, 0) < 0)
      {
         perror("parent: send()");
         exit(-1);
      }

      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         close(soc);
         break;
      }
   }

   /* await Child exit */
   waitpid(pid);
   return;
}
```

Both listings interact with the user through the keyboard and screen; both use `fork` to create a child subtask, and each parent communicates with its child subtask. One program uses UDP (Listing 3.6) and the other uses TCP (Listing 3.7). The TCP version ensures receipt by the child, but the distinction is not obvious at the level of an application program's code. TCP does a lot of work for you to keep the communication link available and to make sure that the messages the parent sends to the child arrive as sent, and in the same order.

Simple as they are, they can serve as patterns for some of your future applications. Enter the message `EXIT` at the prompt to exit these programs cleanly.

Internetworking Protocol Addresses

Up to now, you have been concerned with subtasks and other processes that are all running on the same computer. However, you need to communicate with processes on other machines on the network if you are going to manage a cluster.

The Internetworking Protocol, or *IP address*, is a 32-bit number that uniquely defines each machine on a network. This addressing scheme is used by interprocess communication, the subject of this chapter.

NOTE

When the IP address was designed, the more than four billion addresses it offered seemed like a number that would never be approached, let alone exceeded. IP version 6 and other schemes, however, are recommending that this value be drastically expanded to allow for estimates based on modern day experience. For purposes of this discussion, an IP address is 32 bits long.

The IP address is kept in memory as part of a data structure used by the communication support software. When you send a message, that software looks there to find out where to send it. The IP address is included as part of the packet header information. As your message moves through the communication infrastructure, the IP address is used first to get it into the correct network and then to get it to the right host. (After it is in the correct network, the IP address is used to determine the media access, or MAC, level address of the destination host. See the "Further Reading" section for more information on this subject.)

The IP address is a binary string of bits that are often printed out in hexadecimal for debugging purposes. When humans need to look at IP addresses, they are often converted to dotted decimal notation first, for convenience. Convert each byte from hex to decimal, left to right, separating each of the four decimal values with periods, or dots, as shown in Figure 3.1.

Your network will be represented by the first three decimal numbers, and the fourth number will distinguish each of your individual host machines. In this book, IP addresses 10.0.0.1, 10.0.0.2, and so on represent hosts numbered 1, 2,and so on, on network number 10.0.0.

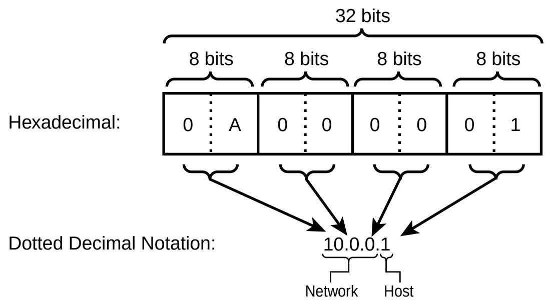

FIGURE 3.1 The Internetworking Protocol (IP) address format and its contents.

Messaging Across the Network

Now that you have an internetworking address scheme, how do you use it to send a message from one process on one machine to another process running on a different machine on the network? Listing 3.8a and Listing 3.8b represent a pair of programs that communicate across the network.

LISTING 3.8A Sends UDP Datagrams to a Remote Server

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 5678

main(int argc, char *argv[])
char *argv[];
int argc;
/*
**   Listing3.8a.c - sends UDP datagrams to a remote server
*/
{
   struct sockaddr_in sock;
   struct hostent *hp;
   int port = PORT;
   int fd;
```

LISTING 3.8A Continued

```
char buffer[80];

/* server */
if(argc < 2)
{
   printf("\n\tUsage: %s <IP_Addr>\n\n", argv[0]);
   exit(-1);
}

/* get vitals regarding remote server */
strncpy(buffer, argv[1], sizeof(buffer));
if((hp = gethostbyname(buffer)) == NULL)
{
   perror("client: gethostbyname()");
   exit(-1);
}

/* establish and initialize the UDP socket */
if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
   perror("client: socket()");
   exit(-1);
}
memset((char *) &sock, 0, sizeof(sock));
memcpy(&sock.sin_addr, hp->h_addr,   hp->h_length);
sock.sin_family = hp->h_addrtype;
sock.sin_port = htons(port);

/* send a message typed from interactive user to remote server */
while(1)
{
   printf("\nEnter a message: ");
   fflush(stdout);
   fgets(buffer, sizeof(buffer), stdin);

   sendto(fd, buffer, strlen(buffer)+1, 0,
    (struct sockaddr *) &sock, sizeof(sock));

   if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
   {
      close(fd);
```

LISTING 3.8A Continued

```
         break;
      }
   }
   puts("Bye!");
   return(0);
}
```

LISTING 3.8B Prints UDP Datagrams from a Remote Client

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 5678

main()
/*
**   Listing3.8b.c - prints UDP datagrams from a remote client
*/
{
   struct sockaddr_in sock;
   int len=sizeof(sock);
   struct hostent *hp;
   int port = PORT;
   int fd;
   char buffer[80];

   /* establish and initialize UDP socket struct */
   if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
      perror("server: socket()");
      exit(-1);
   }
   memset((char *) &sock, 0, sizeof(sock));
   sock.sin_addr.s_addr = htonl(INADDR_ANY);
   sock.sin_port = htons(port);
   sock.sin_family = AF_INET;
   if(bind(fd, (struct sockaddr *) &sock, len) < 0)
   {
```

LISTING 3.8B Continued

```
      perror("server: bind()");
      close(fd);
      exit(-1);
   }

   /* echo messages received from client back to user */
   while(1)
   {
      recvfrom(fd, buffer, sizeof(buffer), 0,
       (struct sockaddr *) &sock, &len);
      printf("Server: %s", buffer);
      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         puts("Bye!");
         close(fd);
         break;
      }
   }
   return(0);
}
```

Start the server (see Listing 3.8b) first, and then start the client (see Listing 3.8a) on some other machine. To tell the client where the server is running, include the server's IP address in the command line when you start it.

On the server machine at 10.0.0.2:

```
> server            [Note that the ">" symbol represents a command line prompt.]
```

On the client machine at 10.0.0.1:

```
> client 10.0.0.2
```

Interact with the client the same way you did with Listing 3.6 or Listing 3.7. Look for responses from the server on the server machine. Here's what you might see. Perform steps 1 through 5 on the server or the client columns, as shown:

```
      10.0.0.2                  10.0.0.1


1)      > server


2)                          > client 10.0.0.2
```

```
3)                          Enter a message: Hello there, server!

4)      Server: Hello there, server!

5)                          Enter a message: (and so on. Enter "EXIT" to end the
                            session)
```

The programs in Listing 3.8a and Listing 3.8b are separate client and server programs. They use UDP to communicate just like the combined form in Listing 3.6, but they communicate from one machine to another across the network. (Similarly, Listings 3.9a and 3.9b are the internetworking counterparts of Listing 3.7 using TCP.) Examine the differences between these two sets and the overall differences in the requirements when communicating outside your own machine. As always, you are encouraged to experiment. Sometimes the fastest answer to "What happens if I do this?" is "Try it and see!"

LISTING 3.9A Sends TCP Packets to a Remote Server

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4321

main(argc, argv)
char *argv[];
int argc;
/*
**   Listing3.9a.c - sends TCP packets to a remote server
*/
{
   struct sockaddr_in sock;
   struct hostent *hp;
   int port = PORT;
   int fd;
   char buffer[80];

   /* server */
   if(argc < 2)
```

LISTING 3.9A Continued

```
   {
      printf("\n\tUsage: %s <IP_Addr>\n\n", argv[0]);
      exit(-1);
   }

   /* get vitals regarding remote server */
   strncpy(buffer, argv[1], sizeof(buffer));
   if((hp = gethostbyname(buffer)) == NULL)
   {
      perror("client: gethostbyname()");
      exit(-1);
   }

   /* initialize the TCP stream socket structure */
   if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      perror("client: socket()");
      exit(-1);
   }
   memset((char *) &sock, 0, sizeof(sock));
   memcpy(&sock.sin_addr, hp->h_addr,   hp->h_length);
   sock.sin_family = hp->h_addrtype;
   sock.sin_port = htons(port);

   /* connect to the remote server */
   if(connect(fd, (struct sockaddr *) &sock, sizeof(sock)) < 0)
   {
      perror("client: connect()");
      exit(-1);
   }

   /* send a message typed from interactive user to child subtask */
   while(1)
   {
      printf("\nEnter a message: ");
      fflush(stdout);
      fgets(buffer, sizeof(buffer), stdin);

      if(send(fd, buffer, strlen(buffer)+1, 0) < 0)
      {
         perror("parent: send()");
```

LISTING 3.9A Continued

```
         exit(-1);
      }

      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         close(fd);
         break;
      }
   }
   close(fd);
   return(0);
}
```

LISTING 3.9B Prints TCP Packets from a Remote Client

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4321

main(void)
/*
**   Listing3.9b.c - prints TCP packets from a remote client
*/
{
   struct sockaddr_in sock;
   int len=sizeof(sock);
   struct hostent *hp;
   int port = PORT;
   int acc, soc;
   char buffer[80];

   /* create server socket to accept a connection */
   if((acc = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      perror("server: socket()");
      exit(-1);
   }
```

LISTING 3.9B Continued

```
   memset((char *) &sock, 0, sizeof(sock));
   sock.sin_addr.s_addr = htonl(INADDR_ANY);
   sock.sin_port = htons(port);
   sock.sin_family = AF_INET;
   if(bind(acc, (struct sockaddr *) &sock, len) < 0)
   {
      perror("server: bind()");
      close(acc);
      exit(-1);
   }

   /* await connection */
   if(listen(acc, 5) < 0)
   {
      perror("server: listen()");
      close(acc);
      exit(-1);
   }
   if((soc = accept(acc, (struct sockaddr *) &sock, &len)) < 0)
   {
      perror("server: accept()");
      close(acc);
      exit(-1);
   }
   close(acc);

   /* echo messages received from client back to user */
   while(1)
   {
      recv(soc, buffer, sizeof(buffer), 0);
      printf("Server: %s", buffer);
      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         puts("Bye!");
         close(soc);
         break;
      }
   }
   return(0);
}
```

TIP

You can test these program pairs using an alternative way, in case you don't have a network built yet. Start a second login session (by opening a new X Window or by pressing Alt+F2 on Linux) and consider it a server machine. Run the client application from your original login session with a destination IP address of 127.0.0.1, which is the universally recognized "local loop-back" IP address. The term *loop-back* means "the same machine." Do everything else the same as you did in the preceding example.

Automatically Starting Remote Servers

In the client/server example, you started the server manually, but a server will usually be running before you try to connect to it. A database server machine, for example, usually starts a database server application at boot and restarts it in the event of failure. Your client interacts with the database server and expects it to always be available. A more familiar example might be your Web browser as a client that accesses your favorite Web sites, expecting them to be up 24 hours a day, 7 days a week.

You don't want to have to start your experimental server every time, and it isn't practical to have it running all the time either! How can you have a server start when you want to use it and have it go away again after you're done with it? Unix allows you to specify your own services, but setting this up requires root access. Because this subject is closer to communication than to administration, it is covered here. If you're not comfortable being a root operator yet, skip this section for now and come back to it after you've had a look at Chapter 5, "Configuring the Relevant Operating System Files," later on in this book.

The Unix internetworking daemon, named inetd, is started at boot and is referred to as the super server. This is because it is a generic service, awaiting a connection to any one of its supported services. That is, instead of having your service up and running all day, every day, you configure inetd to respond to a connection attempt by actually starting your service process and then giving it control, after reassigning standard input and output streams that would be a socket connection in an ordinary server. That's right—an inetd service simply reads from standard input (file descriptor 0) and optionally writes back to the client via standard output (file descriptor 1). Compare Listing 3.8c and Listing 3.9c to their original server counterparts in Listing 3.8b and Listing 3.9b. These are much simpler indeed!

LISTING 3.8C Service: Prints UDP Datagrams from a Remote Client

```
#include <stdio.h>

main(void)
```

LISTING 3.8C Continued

```
/*
**   Listing3.8c.c - service: prints UDP datagrams from a remote client
*/
{
   FILE *log;
   char buffer[80];

   /* open a message log file */
   log = fopen("/tmp/log", "w");

   /* echo messages received from client into log file */
   while(1)
   {
      read(0, buffer, sizeof(buffer));

      fprintf(log, "Service: %s", buffer);

      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         puts("Bye!");
         break;
      }
   }
   fclose(log);
   return(0);
}
```

LISTING 3.9C Service: Prints TCP Packets from a Remote Client

```
#include <stdio.h>

main(void)
/*
**   Listing3.9b.c - service: prints TCP packets from a remote client
*/
{
   FILE *log;
   char buffer[80];

   /* open a message log file */
```

LISTING 3.9C Continued

```
   log = fopen("/tmp/log", "w");

   /* echo messages received from client into log file */
   while(1)
   {
      read(0, buffer, sizeof(buffer));

      fprintf(log, "Service: %s", buffer);

      if(strncmp(buffer, "EXIT", 4) == 0)   /* exit request */
      {
         puts("Bye!");
         break;
      }
   }
   fclose(log);
   return(0);
}
```

Of course, there is a cost to such a convenience: you must change two of the inetd configuration files. These files are owned by root, so you must be root to change them.

Add the following two lines to file /etc/services near the bottom, in order of their port numbers:

```
tcpsvc      4321/tcp      # Linux Cluster Architecture TCP example service
udpsvc      5678/tcp      # Linux Cluster Architecture UDP example service
```

When you examine the contents of the /etc/services file, you may notice that the lower port numbers are associated with commonly used remote services. The `Telnet` and `FTP` services, for example, would be impossible for the public to use if there were no agreed-upon port numbers. These are referred to as *well-known ports*. Some of the higher numbers, used by commercial server packages, Web servers and database servers, for example, publish their port numbers to reserve them for their use. It is generally safe for you to use an unused port number over 4096.

Similarly, add the following section to file /etc/inetd.conf:

```
      #
      # Linux Cluster Architecture TCP and UDP service examples
      #
      tcpsvc      stream      tcp      nowait      nobody
➥/home/user/bin/Listing3.9c Listing3.9c
```

```
    udpsvc      stream      udp      nowait      nobody
➥/home/user/bin/Listing3.8c Listing3.8c
```

Now enter

```
> killall -HUP inetd
```

as root, and, issue the same client commands you used to test the programs in Listings 3.8a and 3.8b and Listings 3.9a and 3.9b.

You will not see your messages echoed to the screen on the remote server system, however, because the standard output stream, was reset by inetd to represent the communication pathway back to the client. The messages you sent are kept in a log file now, in /tmp/log, which may be listed out to make sure everything works as expected.

```
    > cat   /tmp/log
```

The `cat` command, short for concatenate (to the output stream, in this case), is used to list file contents to the screen.

TIP

If you do not find the /etc/inetd.conf files, your version of Linux may be using `xinetd` instead of the classic `inetd`.

In this case, instead of the two lines entered in each file, create two small files (as root) under /etc/rc.d/init.d/ named for the services Listing3.8c and Listing3.9c, respectively.

For example, in file /etc/rc.d/init.d/Listing3.9c:

```
service Listing3.9c
{
       port            = 4321
      socket_type      = stream
       protocol      = tcp
      wait            = no
      user            = nobody
      server            = /home/user/bin/Listing3.9c
      only_from      = 10.0.0.0
      disable            = no
}
```

Create a similar file for the UDP server. You can get more details about `xinetd` service file entries from the xinetd.conf manual page. Enter `man xinetd.conf` at the command line.

When both files are created, enter

```
  > /etc/rc.d/init.d/xinetd restart
```

as root, and experiment with this client-server pair as before.

Summary

You started with a simple subtasking example using `fork`. Splitting this program into its parent and child parts and compiling them separately, you saw how the parent could create a subtask from a separate executable using `execl`. You learned about the use, capture, and handling of signals and how to create and manage a shared memory area between a parent and child process, including how to prevent corrupted data with semaphores. Finally, you learned about inter-process communication using two kinds of sockets.

Moving from one machine into a pair of processors on a network, the parent child concept expanded into the more common client and server paradigm. You set up client and server processes on two separate machines using both kinds of sockets again, and then configured the system to automatically start the server process on the remote machine when the client requested service.

Inter-process communication is a complex subject. If you compile each of these demonstration programs and experiment with them, they will lead you into the general case where many clients communicate with a single-server system. This is the mechanism in use by database servers and Web servers alike. It is the foundation of network programming.

Further Reading

- Brown, Chris. *Unix Distributed Programming*. Hertfordshire, England: Prentice Hall International (UK), 1994.
- Hawkins, Scott. *Linux Desk Reference*. Englewood Cliffs, NJ: Prentice-Hall PTR, 2000.
- Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Reading, MA: Addison-Wesley, 1992.

4 Assembling the Hardware for Your Cluster

IN THIS CHAPTER

Processor speeds increased dramatically over the last three years. Newer, faster computer models kept coming out until the threat of a recession put the damper on their sales. Individuals and families thought, quite rightly, that they could make do with the computers they already owned. Office managers decided that cuts to their budgets could be partially met by extending computer lease agreements another two years. Lately, the computer manufacturing industry has experienced hard times.

The only good news to come out of all this is for the consumer: major computer stores have their hottest systems on sale at about 15% below last year's prices. Second-tier models are even cheaper, and the economy models are selling for just under a $1,000. Computer salvage houses have $30 price tags on computers that will easily run all but the very latest operating systems and their productivity suites. Add a monitor, keyboard, mouse, modem, and a sound card, and you've built an adequate home system for about $100. Considering the computer configuration you need for the nodes of your cluster system, a four-node cluster can be had for very little money. Let's get started!

Node Processors and Accessories

If "money is no object," as they say, you can buy four or more of the best systems available. For the fairly well heeled, a set of second-tier machines or perhaps a larger number of economy machines is the cluster of choice. We frugal designers, however, must plan things more carefully.

Even if you can afford better, maybe your first cluster ought to be built from salvage PCs. Someone was proud to have these systems on their desk not so long ago. They earned their keep and have nothing more to prove; they can probably still cut the mustard with an efficient OS such as Linux. Besides, old machines will teach you a lot more about computers than the new ones will. You will swap cards and upgrade memory, get to know a PCI slot from an ISA, and learn to spot the notch end of memory strip without a thought. You will have to figure out memory part numbers, determine its network interface speed, and so forth. In so doing, you will gain a sense of confidence and a working knowledge of your systems. Let's look deeper into the cost-effective components that can be used to make a frugal, but effective, cluster.

NOTE

Large corporations, computer discount houses, and institutions such as hospitals and universities have a central information telephone number that you can call for details about their salvage policies. Many have employees who volunteer to rebuild the company's old computers for use at local schools, so they don't sell them to the public. Some do, however, and it's worth a local call to find out. A university near me has a warehouse dedicated to such sales and is open to the public during daytime business hours. They have a test bench in the back, and their prices are quite reasonable. A bottle of general purpose cleaner and a roll of paper towels can bring these systems back to very nice, if not showroom, condition.

Hardware Accessories

Assuming you are going to build a four-node cluster, try to find five PC base units that are identical in manufacturer and model number. Look inside them to make sure they all have a hard disk and CD-ROM drives and that the memory slots are at least partially populated. Check the rear port male connectors for bent pins and don't forget the power cords. If the store will let you try it before you buy it, do so. Bring a boot disk to see if it will boot without errors. An extra machine is nice to have for parts when something goes wrong, but buying five dead PCs can really dampen your enthusiasm!

I/O Requirements

You need only one monitor, keyboard, and mouse for this cluster because Linux will let you log in to remote systems from the main computer. Attach them directly to each machine as you check it out and load the OS, but after they are up and running on a network, they will be attached to the network file system (NFS) server. You will learn more about NFS in the next chapter.

Don't forget to look inside keyboard, mouse, and monitor cable ends and check all those tiny, delicate pins carefully. Be sure the monitor has both a power cord and a video interface cable. Do not remove the cover from the monitor unless TV repair is your specialty. The video screen, a cathode ray tube, or CRT, is a lot like a TV screen and requires a very high operating voltage that stays high even after it has been unplugged for long periods of time.

Networking Hardware

Remember, too, that these machines are going to be networked, so look for the RJ45 jack on the back of the PC on the system board along with the other ports, as shown in Figure 4.1, or on one of the accessory cards, as shown in Figure 4.2. Find out what speeds the network interface supports before you buy a hub. Many of the salvage systems at this writing are 10Mbps, and computers with 10/100Mbps support are just starting to become available. The speed of the hub must be compatible with the speed of each computer's network interface. A 10/100Mbps hub will work with both 10BASE-T and 100BASE-T, but if one 10BASE-T interface is on the network, everything runs at 10Mbps. If the hub is 10Mbps and all your machines are at 100Mbps, nothing will work, so be careful about what you buy. You can always add a 100Mbps card to a computer and ignore its 10Mbps system board interface, but that drives up the overall cost of the cluster.

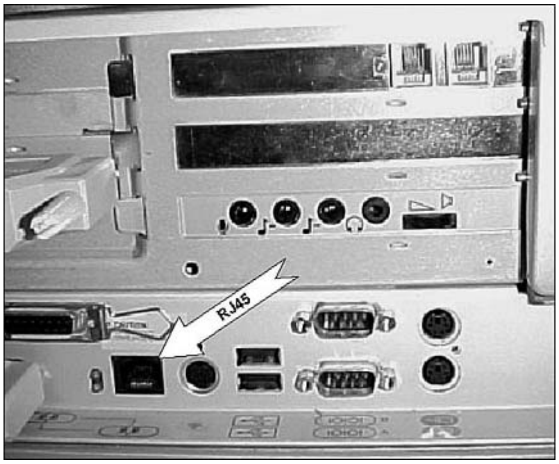

FIGURE 4.1 A network interface RJ45 connector on the computer's system board.

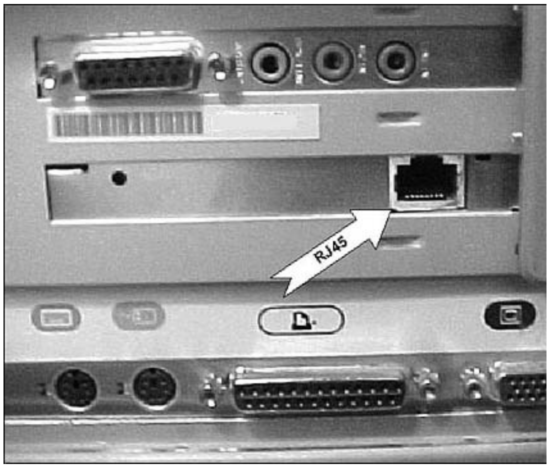

FIGURE 4.2 A network interface RJ45 connector on an accessory card.

NOTE

Most computer manufacturers have specification information for their discontinued models on their Web sites, usually under a Support section. See the "Further Reading" section at the end of this chapter for a few reference Web sites.

Network Media and Interfaces

Results from the Beowulf Project (see `www.beowulf.org`) emphasized the need for a fast external network if a network of PCs was going to be used as a competitive replacement for a more expensive multiprocessor in supercomputing applications. The custom external network and device drivers they built are well beyond our needs, but you should be aware of the choices available to you, even on a home network. Radio frequency links, fiber optics, and even high speed (gigabit plus) ethernet are available to ordinary consumers in many of the larger computer stores today. So many firewalls, routers, switches, and hubs are available that it can be hard to find the simple networking products you need here.

Switches or Hubs?

If money is not an object, you might consider switched, gigabit ethernet. A hub emulates the coaxial cable network by letting each of the other machines on the network hear what the sender is transmitting. A switch makes a connection between the sender and receiver, reducing the potential for collisions. The benefit is better performance and the cost is, well, the cost; switches are expensive and are probably a lot more sophisticated than what you will need for this project. If your application regularly drives network utilization up beyond saturation, a higher-speed network is definitely indicated, and a switch might be necessary. A 10BASE-T or 100BASE-T hub will suit your networking needs.

It is hard to find a 10Mbps hub on computer store shelves today because everything is moving to the 100BASE-T standard and higher. Be sure the number of ports on whichever hub you buy exceeds the size of your cluster. You will want an extra machine on the same network for performance monitoring, transaction query generation, and so on, and you should leave room for cluster expansion. Note that many of the "home network" hubs allow stacking. You can plug one stacking hub into another, at the cost of only one port on each, to make a single hub with the sum of their individual ports, minus two. Stacking a four-port and a six-port hub gives you an eight-port hub, and so on. It's cheaper just to buy an eight-port hub, of course, so plan ahead.

Network Cabling

Network cabling comes two ways: in bulk rolls so you can cut it to custom lengths and attach your own RJ45 plug with a special pair of crimping pliers, or as commercially prepared cables with the plugs already connected. Bulk rolls can be a real cost saver for companies installing hundreds of PCs on their networks, but the prepared cables at 6-foot lengths have served this author well over the years. If your space is tight, you can consider the 3-foot lengths, but the price difference may not justify the inconvenience if you expand your cluster later.

The ordinary, low-cost gray cables are fine for simple networks. (They look like larger versions of ordinary telephone cords.) There are even cables with special jackets that meet building code requirements, should they have to run through air conditioning or heating system ventilation ducts. Our system is simple; you are not likely to be running cables through your air-conditioning ducts, and inexpensive cable should suit your needs perfectly.

Cabling with Crossover Cables

One more thing about networking cables: If you have only two machines, you can avoid the cost of a hub by buying a specially wired cable called a *crossover* cable. The hub, you may recall, connects to each PC through a single cable (see Figure 4.3). The hub's job is to retransmit everything it gets in from any one cable out to every other cable. An ethernet network connects all its machines through the same single coaxial cable described in Chapter 2, "Multiprocessor Architecture." When one PC sends a packet, it is received simultaneously by all the others. Firmware in the network interface card inspects the destination address and ignores the packet if it doesn't match.

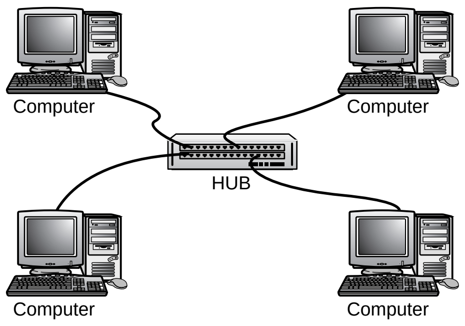

FIGURE 4.3 A hub and four networked computers.

A crossover cable eliminates the hub through crossed wiring; the transmit pin of one machine's network interface is connected to the receive pin of the other. (This is similar to the null modem cable approach to RS-232 serial port interfacing of two machines in close proximity.) These RJ45 crossover cables work only for a two-node network and will not work with a hub. If you buy one of these, and it is not already clearly marked as a crossover cable, mark it as such. It can save you a lot of troubleshooting time at some future date.

Implementing an OS

Salvage PCs usually come with a reformatted hard drive, wiped clean of its previous owner's e-mail, favorite Web sites, and sensitive documents and data. You must supply an operating system if you want these machines to boot. The Linux OS is used for all the examples in this book. It is available for free at several Internet download sites. If you are an expert capable of configuring and building an OS and installing it on a remote system, it really is free. For the rest of us, the $30 or so for an off-the-shelf Linux installation package is worth the price.

Some installation packages include a boot disk—a 3 1/2-inch disk. Some packages provide for one to be created during the installation process. Follow the instructions in the installation guide to create a simple system. Remember to include support for the C programming compiler and libraries. You may want to include support for the X Window System. You certainly want support for a fixed address, local area network.

NOTE

My first cluster did not include the X Window option because the hard disks were too small to support anything beyond a text-based interface. My second cluster included X support, however, making it possible to monitor activity on all the remote systems at once. X support isn't necessary for what you are going to build, but it's nice to have because you can open a separate window for each node on the monitor connected to the main node.

Adding Network Support to the OS Installation

When you get to the networking information part of the installation process, select the local area network option, without any dynamic name server (DNS) address support. Use fixed IP addresses from 10.0.0.1 through 10.0.0.x, in which x is the number of machines in your cluster. For example, a four-node cluster will require IP addresses 10.0.0.1, 10.0.0.2, 10.0.0.3, and 10.0.0.4.

You will want to have friendly names for each machine in your cluster. Your host-names can be formal, such as system1, system2, and so on, but nothing prevents you from having a little fun with them. You can use the names of the planets: Mercury, Venus, Earth, and Mars, for example. You can use the names of presidents, automobiles, four of your favorite cartoon characters, gemstones, and the like; the choice is yours. The OS changes a friendly name into an IP address when you send a message or log in to a remote machine, based on entries in the system's hosts file, which is discussed in the next chapter.

TIP

An inexpensive labeling machine is available in most discount department stores. Print a strip with each name and one with each IP address and stick them on the front of each machine for easy recognition.

Our Cluster System's Network Topology

Our four-node cluster is built from four 166MHz PCs, each with a 64MB memory, a 3 1/2-inch floppy drive, a CD-ROM drive, a 1GB hard drive, and a 10Mbps ethernet RJ45 connector built in to the system board. We have a single monitor, keyboard, and mouse, five commercially made network cables with RJ45 plugs on each end, and a six-port, 10BASE-T hub to complete the package. Our cost was $30 for each PC base unit, $50 for the monitor, keyboard, and mouse, and $70 for the hub and cables—for less than $300, including a commercial Linux package and sales tax. The hub's network topology is diagrammed in Figure 4.4.

Note that each of the four cluster nodes in Figure 4.4 is connected to the hub by a single cable. Only the local node has a monitor and keyboard attached to it. (If all four nodes are essentially the same, any one can be the local node. If one has a larger hard drive, choose it. This is not a critical choice.) You will be accessing each of the remote notes from the local node through that single keyboard and monitor. These techniques are discussed in a later chapter.

The cluster nodes are going to work together as a server. You will need at least one external computer on this same network to run the client software that sends queries to your server. It will also run performance monitoring and display software that shows you how stressed each server node is. These functions are discussed in greater detail in later chapters.

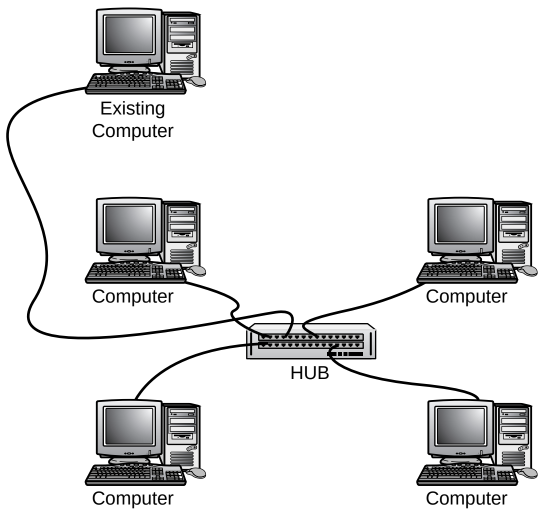

FIGURE 4.4 A hub and four networked computers connected to a fifth, existing computer.

Summary

In this chapter, four salvage PCs were purchased, Linux was loaded on each of them, and they were interconnected with a 10Mbps hub. Only one of the PCs has a monitor, keyboard, and mouse attached. It will become the network file server.

The ease with which multiple computer systems can be networked under the Linux operating system will become apparent when you create a user work area in the file system. This single user can log in to any machine on the network and use the same home directory. All its files look the same from any machine because they *are* the same files, made available through the NFS service. You will discover file service in the next chapter.

Further Reading

- `http://www.compaq.com` has a support section that contains details about its older and even some of its obsolete computer lines. See also `http://www.dell.com`, `http://www.ibm.com`, and others.
- `http://www.crucial.com` is another good source of computer memory information.
- `http://www.cisco.com`, `http://www.baynetworks.com`, and other networking sites can teach you about how the more complex approaches work.
- `http://www.google.com` and other search engines can help you find whatever else you might need to know about building these kinds of systems.

5 Configuring the Relevant Operating System Files

IN THIS CHAPTER

Network interface cards, a hub, and a few cables are just not enough to meet networking needs. Each machine on the network must know that a network exists, know its own network name, and know the names of at least some of the remote machines. Until you can access a remote machine and all its resources from your local machine, the network isn't a network.

Making all this hardware work together requires that you change some Linux system files. The Linux network support you need to influence comes in the form of active processes, called *daemons*, started at boot or as needed in response to an inter-process communication. The remote service is discussed in Chapter 3, "Inter-Process Communicatios," and we build on that discussion later in this chapter in the section titled "Remote Reference Commands." These processes look at certain system files every time you try to access a remote machine by name. These files are well defined and can be found in predictable locations. (Remember that later versions of an OS might change the format or location of these system files.)

A Brief Review of the Cluster Configuration

During the Linux installation process, you are asked to identify hostnames, IP addresses, the domain name, and so on. The configuration examples that follow use invented machine names and addresses to show you what needs to be added or changed in each case. Remember that you can choose whatever names you like, provided you stay within some minor constraints.

Our four-node cluster's system names are alpha, beta, gamma and delta, at IP addresses 10.0.0.1 through 10.0.0.4, respectively. They are all on a domain named cluster.net. Finally, you will need a user ID for nonsystem configuration and application development. We use *chief* for our general-purpose user ID. We assume you established a user ID named *chief* on each of these systems during the Linux installation process.

The machine named alpha is our primary system; always boot it first. It will have the monitor, keyboard, and mouse (if any) directly attached to it. It will be our file server, which means user files in the home directory will be physically located on alpha's hard drive. When we log in to one of the other remote machines, those files, through the magic of the Network File System, or NFS, will appear to be there. We'll discuss NFS when we make changes to the /etc/fstab and /etc/exports files in a later section of this chapter.

The Linux root User

If dozens of user accounts are on a particular machine, then dozens of home directories exist, each containing its user's personal files. Which user, then, is responsible for the system files?

A special user ID, named *root*, is on all UNIX systems. This user is also called the *super user* because of its special powers, as you will see. The root user owns and is responsible for all the system files. If any changes must be made to any of them, you must become the root user; you may do so with the `su` (switch user ID) command, as follows:

```
> su  -  root
```

The `su` command enables you to switch to some other user ID. That dangling dash (-) shows your intent to assume the personal profile of that user. (When you change to the root user ID, you don't even have to specify root; just su—is sufficient.) You will be asked to type in a password before you are allowed to proceed. Use the root password that you established during system installation.

When you are finished doing work as root, type the **`logout`** command to end the root login session and resume the earlier session under your personal user ID. You may also end a session by typing an **`exit`** command, or simply a Ctrl+D keystroke, which is by far the quickest and easiest method.

Logging in as `root` at the System Prompt

Another way to become the root user is to log in as root at the system prompt. Some security-conscious systems, however, do not allow the root user to log in directly, forcing the use of the `su` command so the system administrator can restrict its use to certain trusted users. You will find that we lean toward lax security in this book because the cluster is assumed to be under your full control and completely unavailable to anyone who might want to do it harm. This assumption drastically reduces system administration requirements.

Altering Linux System Files as `root`

Now that you are the root user, let's look at some changes you might make to the system's configuration.

The Linux OS may be configured to support certain kinds of devices, for example, before the OS is built as part of the installation process. These configuration parameters are semipermanent, requiring you to rebuild the OS if you want to change them. There are less-permanent configuration parameters, which may require you to reboot the system before they take effect. And some configuration parameters require only that you restart their associated process or send that process a signal telling it to reread its configuration file. Again, the aim is to keep it simple.

The parameters you must change are inside a text file, making them easy to edit. If you change the contents of a system configuration file and then restart or signal the associated process, you've changed the system configuration. That's one big reason why you don't want others to know the root password. Another reason is that when you are logged in as root, you can use the `su` command to become any other user, without having to know that user's password. In fact, you can reset other users' passwords while you are the root user, and you can modify or even remove their files. The purpose of becoming root is to do system administration, which includes changes to user accounts. The most important thing to remember about root is that you can do so much harm to the integrity of a system as root that you might have to wipe everything clean and reinstall the OS. *Be careful as root.*

Use the root user ID when you need it, but do your personal software development and personal account administration under an ordinary user ID, such as the account chief used in our examples.

Changes to the /etc/hosts File

How do all these name parts fit together? If you want to send an email message to the root user on the machine named gamma, for example, the address is `root@gamma.cluster.net`, which shows the user ID first, the hostname next, then the host's network (domain) name—from most specific to least specific. Getting that message to the root user's mailbox requires a look at the domain name first, then the hostname, and then the user ID.

NOTE

A mailbox, by the way, is a text file with the same name as your user ID; it gets larger with each new e-mail and shrinks as you read and delete them. The mailbox for chief, for example, is found at /var/spool/mail/chief. This is not too different from the address you write on the front of an envelope if you generalize syntax requirements.

Every time a remote host's name is used by a remote reference command, it must be translated into an IP address. The /etc/hosts file contains all the names that a node might need, so you must add all your node names and IP addresses to this file. Log in as `root` and use your favorite editor to change the contents of /etc/hosts, as follows:

```
10.0.0.1      alpha.cluster.net      alpha
10.0.0.2      beta.cluster.net       beta
10.0.0.3      gamma.cluster.net      gamma
10.0.0.4      delta.cluster.net      delta
127.0.0.1     localhost.localdomain  localhost
```

Left-justify the IP addresses and use tabs for readability between the long, fully qualified hostname and the shorter alias name. The last entry is typed as shown and is in reference to the local host on every system. Whatever the hostname, localhost can always be used as its host name. Make all the hosts files on each of your nodes look the same.

The machine name is used by a remote reference command to discover an IP address, which is used to communicate with the corresponding process on that remote machine. (You will design a sample remote reference command later in this chapter.) You may be unaware of the many messages sent back and forth between the two machines. You will experience a strong feeling that you are actually logged in to a remote machine, provided the network is configured correctly.

Almost anything you can do while actually logged in to a machine can be done while you are remotely logged in to it. This is the power of networking, but it requires more than just hardware. It requires changes to the OS configuration files such as the /etc/hosts file to make the OS aware of the other machines on the network.

Changes to the /etc/fstab and /etc/exports Files

One thing you might expect when you use a remote login feature is that your home directory, along with all your files, is available to you on the remote machine. This is accomplished by changing two files: the "file system table" (/etc/fstab) on all the remote machines and the "file exports table" (/etc/exports) on the local file server machine. The local file server is named alpha in our examples.

At boot, the presence of a file system reference in the local fstab file triggers interprocess communication to initialize the service, provided that the reference is to a matching file system in the server's exports file. Use the exports file to announce the availability of a local file system, such as the home directory; use the fstab file entry to tell each client machine to request that these file systems be mounted from the file server at boot.

Using NFS with the /etc/fstab and /etc/exports Files

NFS uses a team of active system processes to support the strong illusion that you are working with your personal files on one machine when they are physically located on another machine. That is, the file data resides on the server's disk (alpha's hard drive), but you are working with them as if they were located on a remote node's local disk.

Note that we are referring to the personal files in your home directory—your C programs, for example. System files on your cluster exist on each system's hard drive because they represent the local machine's configuration parameters. There are remote systems, called diskless workstations, that get all their files from a remote file server. These kinds of systems are outside the scope of this book. They do not have a hard drive. They boot from a floppy disk that contains just enough data to get them up and running to a point where they request subsequent file data from a designated server.

Add the following line to the /etc/exports file on the NFS server machine named alpha:

```
/home (rw)
```

This says that you want the /home directory to be available for NFS mount on any remote system with read and write privileges.

Add the following line to the /etc/fstabs file on each of the other (NFS client) nodes:

```
alpha:/home        /home   nfs   rw   0 0
```

This says that you want the NFS to mount alpha's /home directory locally, as /home, with read and write privileges, or, more precisely, over your local /home directory because you are required to have a /home directory defined locally for this mount to succeed. This process is called mounting a remote file system locally.

The NFS server boots first, and its NFS processes discover that the local /home subdirectory is offered for export to any other node that requests it. As the other nodes boot, NFS processes each request that the /home directory be made available to it from alpha. That file system now appears to be on the local node's hard drive. When you log in to the node named gamma (or beta or delta), your home directory will be available to you, along with all your files.

Remote Access Security

The simple cluster system in this book is assumed to enjoy full physical access security. That is, you and your trusted associates are the only ones who will be allowed to get near enough to any of the nodes to log in to them, and no node is accessible from any network outside this physical perimeter. Configuration gets a lot more complicated if you want to keep anyone out! The focus of this book is to help you get your cluster up and running. If you ever allow public access to it—through the Internet, for example—you will have to make changes to tighten up the security. Here, we are interested in loosening security to make remote access easy on ourselves, to save time.

After you are logged in to alpha, you can access a remote machine through the `telnet` command:

```
> telnet beta
```

This command presents a login prompt to you, asking for a user ID and password, like the process you followed when you logged in to alpha. It will log you in to the remote machine named beta. This assumes, of course, that the user ID and password you use match the user accounting you established during the Linux installation process on beta.

Similarly, you can issue the `rlogin` (remote login) command, and it will assume that you want to log in with your current user ID, asking only for your password:

```
> rlogin beta
```

The `rlogin` command will do the same as `telnet`, but it can assume your current user ID to save you even more time.

The `rsh` (remote shell) command is the same as `rlogin`, but it is shorter and faster to type:

```
> rsh delta
```

It will log you in to the remote machine named delta, just as the `rlogin` command does. This is the fastest and easiest way to remotely log in to a machine, but it gets even faster if you don't have to supply your password. Remember: you are not creating a secure system here!

Optional Addition of a /home/chief/.rhosts File

Even on a physically secure system, OS software security checks can prevent you from simply logging in to a remote system—even one that is right next to you. To take advantage of perhaps the ultimate time-saver, you can create a file in your home directory that will let the system assume your password as well as your user ID. Create a remote host access file named .rhosts (the leading dot is required) located in /home/chief, your home directory.

NOTE

Directory listings can get cluttered with lots of set-and-forget configuration files. By convention, they are named with a leading dot. Operands on the `ls` (directory listing) command ignore these files unless you specifically request to see them. (In the DOS operating system, there are hidden files, but this isn't quite that strong a statement.) Use `ls -l` to list all the files except those beginning with a dot, and `ls -al` when you want to see these dot-files along with all the rest.

```
alpha.cluster.net     chief
beta.cluster.net      chief
gamma.cluster.net     chief
delta.cluster.net     chief
```

Each hostname is followed by the trusted user ID. In this case, you have only the one user ID, so the presence of the .rhosts file in your home directory will facilitate your remotely logging in from any of your cluster nodes to any other. Although you can establish a remote login chain from alpha to gamma to beta to delta, you should form a habit of logging out of one remote machine before logging into another, just to keep it simple.

This file will be checked by the remote login process every time you try to log in from some other node on the network. Note that every node on the network will have this file because every file in your home directory is made available through the network file system, as configured previously. NFS is a very nice feature.

By specifying all the nodes on the network, along with your user ID, you allow remote access to any node from any other node, without any password checking.

Optional Changes to the /etc/passwd File

If you are doing a lot of logging in and out as user chief and a lot of system configuration work as root, you might want to consider removing the need for either of these passwords. This is certainly a serious security risk, but it will save a lot of time. If your system is in a secure area and no one but you (and those you trust with the root password) will ever get close to these systems, you can physically remove the password from your user ID and the root entries in file */etc/passwd*:

```
root:c6752ffa:0:0:root:/root:/bin/bash
chief:4456aab:500:500:Chief User:/home/chief:/bin/csh
```

Each line entry in the password file contains several fields, each field separated by a colon (:) for readability. In Unix systems, readability usually implies *machine* readability, and it can sometimes present a challenge to humans. You can edit this file, as root, and remove the contents of the second field:

```
root::0:0:root:/root:/bin/bash
chief::500:500:Chief User:/home/chief:/bin/csh
```

Note that `::` represents an empty, or null, field in the entry. Passwords are no longer required for either the root user or your personal user ID. Do not do this unless you implicitly trust others who might have access to these systems! You will have to make these changes in the /etc/passwd file on each of your nodes.

NOTE

You may find that later releases of Linux have changed the significance of, or may have even replaced, the password file. There are other ways to eliminate the necessity of typing in a password in those cases. Refer to the Linux documentation included with your installation package.

Remote Reference Commands

In Chapter 3, you examined several variations of fork/execl subtasking and ended up with a pair of processes: one a local client and the other a remote service. The TCP sample pair in Listing 3.9a and Listing 3.9c can be converted into a remote reference command and a remote service daemon process pair. A remote file listing service is implemented in Listing 5.1a and Listing 5.1b.

NOTE

Readers who are not familiar with C program compilation under Linux may want to skip ahead to Chapter 6, "Configuring a User Environment for Software Development."

LISTING 5.1A Remote-`cat` Command: Lists the Contents of a Remotely Located File

```
/*
**   Listing5.1a.c - rcat: lists the contents of a remote file
*/
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
```

LISTING 5.1A Continued

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

#define BUFLEN 8192
#define PORT 6789

main(int argc, char *argv[])
char *argv[];
int argc;
{
   struct sockaddr_in sock;
   struct hostent *hp;
   int port = PORT;
   int fd, ii;
   char buffer[BUFLEN];
   char *p;

   /* argv[] */
   if(argc < 2)
   {
      printf("\n\tUsage: %s <IP_Addr>:<file_name>\n\n", argv[0]);
      exit(-1);
   }

   /* isolate remote host */
   p = strchr(argv[1], ':');
   *p = '\0';
   p++; /* reset to file */

   /* get vitals regarding remote server */
   strncpy(buffer, argv[1], sizeof(buffer));
   if((hp = gethostbyname(buffer)) == NULL)
   {
      perror("Listing5.1a: gethostbyname()");
      exit(-1);
   }

   /* initialize the TCP stream socket structure */
   if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      perror("Listing5.1a: socket()");
```

LISTING 5.1A Continued

```
      exit(-1);
   }
   memset((char *) &sock, 0, sizeof(sock));
   memcpy(&sock.sin_addr, hp->h_addr,   hp->h_length);
   sock.sin_family = hp->h_addrtype;
   sock.sin_port = htons(port);

   /* connect to the remote server */
   if(connect(fd, (struct sockaddr *) &sock, sizeof(sock)) < 0)
   {
      perror("Listing5.1a: connect()");
      exit(-1);
   }

   /* send the file name to the service */
   strncpy(buffer, p, sizeof(buffer));
   if(send(fd, buffer, strlen(buffer)+1, 0) < 0)
   {
      perror("Listing5.1a: send()");
      exit(-1);
   }

   /*
   **   Print remote file contents to screen
   */
   while(1)
   {
      if((ii = recv(fd, buffer, sizeof(buffer), 0)) < 0)
      {
         perror("Listing5.1a: recv()");
         close(fd);
         exit(-1);
      }
      else if(ii > 0) /* good packet received */
      {
         p = buffer;
         while(ii) /* process all input lines */
         {
            /* print each new line till DONE */
            if(strncmp(p, "DONE", 4) == 0)
               break;
```

LISTING 5.1A Continued

```
            else
               printf("Listing5.1a: '%s'\n", p);

            /* subsequent lines */
            ii -= (strlen(p) + 1);
            p += strlen(p) + 1;
         }
      }
      else /* socket connection was closed */
         break;
   }
   close(fd);
   return(0);
}
```

LISTING 5.1B Remote-`cat` Daemon: Sends File Contents Back to the Client

```
/*
**   Listing5.1b.c - rcatd: sends file contents back to client
*/
{
#include <stdio.h>
#define FALSE 0
#define TRUE 1

main(void)
   FILE *fp;
   int notDone;
   char file[64];
   char buffer[80];

   /* client sends file name */
   read(0, file, sizeof(file));
   if((fp = fopen(file, "r")) == NULL)
   {
      sprintf(buffer, "Listing5.1b: file '%s' not found.\n", file);
      fclose(fp);
      exit(-1);
   }

   /*
```

LISTING 5.1B Continued

```
   **   Send file contents back to client
   */
   notDone = TRUE;
   while(notDone)
   {
      if(fgets(buffer, sizeof(buffer), fp) == NULL)
      {
         strcpy(buffer, "DONE\n");
         notDone = FALSE;
      }
      /* zero the newline character */
      buffer[strlen(buffer)-1] = '\0';

      /* send back each line from file */
      write(1, buffer, strlen(buffer)+1);
   }
   close(0);
   fclose(fp);
   return(0);
}
```

As `root`, add the following lines to the /etc/services and /etc/inetd.conf files on the remote machine delta.

In the remote machine's /etc/services file:

```
rcatd 6789/tcp   # Linux Cluster Architecture remote cat service
```

In the remote machine's /etc/inetd.conf file:

```
#
# Linux Cluster Architecture remote cat service
#
rcatd stream tcp nowait nobody /home/chief/bin/Listing5.1b Listing5.1b
```

Reboot the delta to force the internetworking daemon (inetd) to reconfigure itself. Alternatively, you may issue the `killall  -HUP  inetd` command on delta, as root. The HUP signal, short for hangup, is understood by the internetworking daemon to mean a request to reinitialize itself.

Create a text file on delta's /tmp file area and execute the remote-cat command:

```
> Listing5.1a delta:/tmp/sample.txt
```

from alpha, what you see on your screen is actually the text from that file as it exists on the machine named delta (see Figures 5.1 and 5.2).

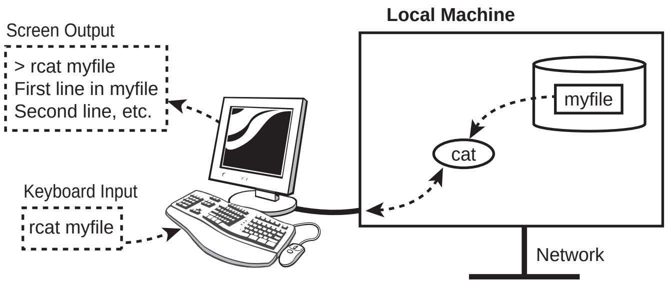

FIGURE 5.1 Local execution of a `cat` command to display a local file's contents.

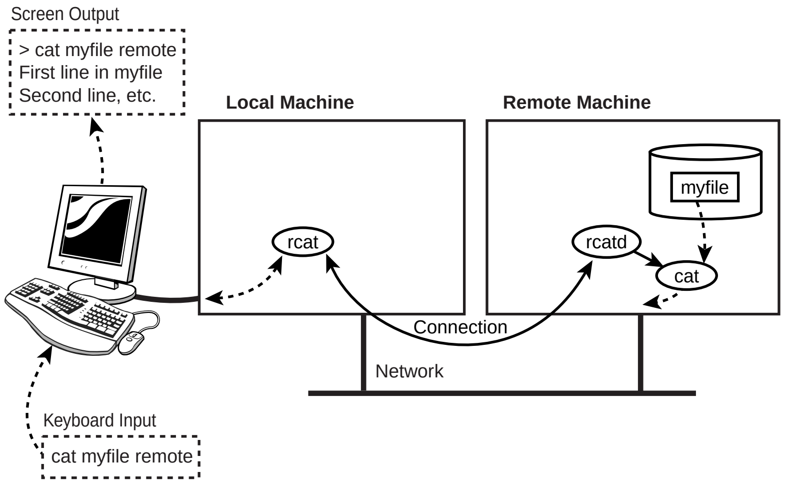

FIGURE 5.2 Remote execution of a `cat` command to display a remote file's contents.

Summary

Successful networking goes beyond the hardware, requiring the reconfiguration of at least a few system files. Each machine on the network needs to know the names and addresses of each of the other machines before it can interact with them. You configured the primary node, named alpha, as the NFS file server. You removed the system's need for password checking for convenience. A network that allows a user to log in to any machine from any other machine without a password check can be a serious security risk, but it makes working with the cluster a lot easier. Finally, you implemented a remote file access command pair: a remote reference command and a file access daemon implemented as an inetd service, described in Chapter 3.

In the next chapter, we will switch topics from the administration of system files to the management of the files you will use to develop the software. The system file configuration you did in this chapter is all aimed toward creating an application development environment, and that's what is discussed next.

Further Reading

- Bach Maurice J. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice Hall, 1986.
- Lamb, Linda, and Arnold Robbins. *Learning the vi Editor*. and Sebastopol, CA: O'Reilly and Associates, Inc., 1998.

6 Configuring a User Environment for Software Development

IN THIS CHAPTER

The work you do for yourself—software development and testing, for example—should not be done as root. The root user has so much authority that a simple slip up can cost you weeks of work, potentially requiring you to reinstall everything from scratch. You'll learn more about disaster mitigation in the "Backup and Recovery" section later in this chapter. One very good habit that you can form early on is to set up one or more user IDs for yourself, perhaps one for each of your major projects, and use them to do the development for that project. The root account is best used for system administration work that requires root access to files or for executing certain privileged commands.

If you do a lot of software development in a Unix environment, and you are already familiar with the C compiler and make utility, feel free to use this chapter as a quick review or to skip it altogether. Although a lot of this is basic, not everyone knows everything, so even a seasoned, professional software developer might gain something here.

An Overview of the Linux File System

The Linux file system is organized in a treelike structure, having a root directory (/) and several subdirectories underneath it (see Figure 6.1). Notice that the branches of the tree spread downward, and the root (no relation to the user ID) is more of a base—a starting point in the tree. Note that /home/chief is the home directory, the starting point or working directory when you log in as chief.

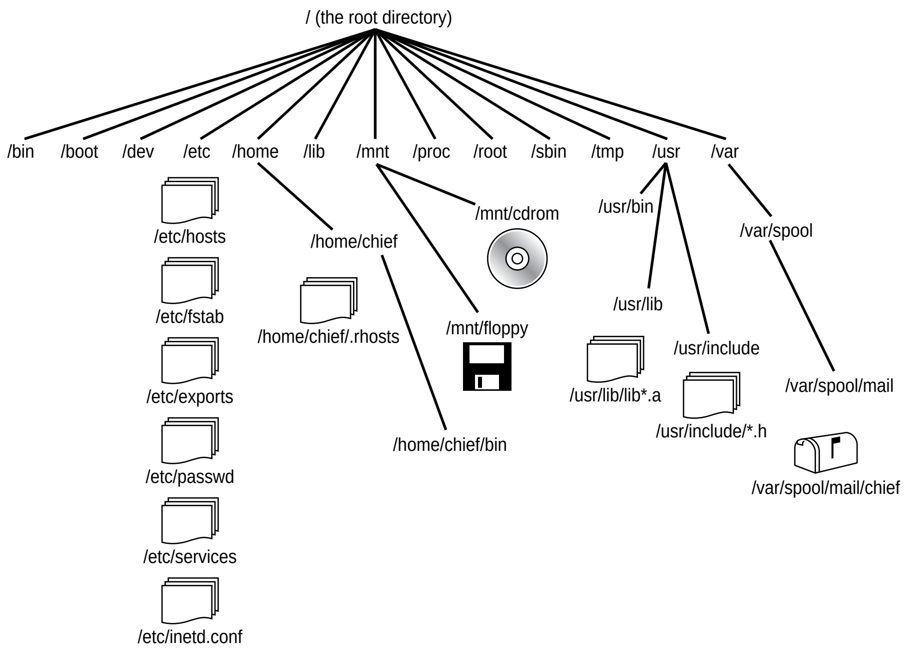

FIGURE 6.1 Linux file system overview.

Following are some of the directory names discussed in this book, followed by a description of what you might find under them. You should recognize some of them as the locations of files discussed in the previous chapter.

- /bin—Executables composing the basic Unix command set
- /dev—Devices accessed through these file-like I/O ports
- /etc—System configuration files
- /home—Home directories for (nonroot) system users
- /mnt—Mount points for removable file systems: floppy disk, CD-ROM, and so on
- /proc—Process information pseudo files, discussed in Chapter 8, “External Performance Measurement and Analysis”
- /root—The root user’s home directory
- /tmp—Temporary files
- /usr—System and application executables and their associated files
- /var—Logs, email, and so on

The directories of most interest in this chapter are /home and /usr. The files under our home directory reside on alpha's hard drive; alpha is the NFS file server. (NFS is discussed in Chapter 5, "Configuring the Relevant Operating System Files.") You will add subdirectories under your /home/chief user's home directory and create some files to facilitate compiling C language programs. The C programming language function libraries and header files are under /usr/lib and /usr/include, respectively.

Your /home/chief Home Directory

When you created the new user ID named chief during installation, a new subdirectory named /home/chief was created for it, along with some environmental control files that you can change to suit your needs. We created a file named .rhosts (see Chapter 5) under /home/chief to make our cluster system a little easier to use. You will also find a bin subdirectory (short for binary) created to contain your local executables.

You need some other subdirectories under your home directory—places to put all your C language source programs and their associated header files. The executables that you make from them will go into /home/chief/bin. Source files are kept in /home/chief/src and headers under /home/chief/inc. Create these subdirectories by using the `mkdir` command, a contraction of "make directory."

```
> cd                    [change to the home directory if not already there]
> mkdir src             [create /home/chief/src for C language source files]
> mkdir inc             [create /home/chief/inc for included C header files]
```

Using the C Compiler

The GNU C Compiler (the `gcc` command) creates executables in processing steps: precompile, compile, assemble, and link. The precompiler processes any directives it finds (the `#include`, for example, inserts header files); the compiler creates object files from C source, the assembler creates object files from assembly language source, and the linker creates executables from object files. Each time you use the `gcc` command, the compiler and the linker need to be told which source programs to compile or assemble and which object files need to be linked. It also needs to know where to find any header files referenced in precompiler directives, which object files compose the resultant executable, and so on. This is a complex process, requiring correspondingly complex instructions.

Listing 6.1 is a C program that starts out simple and gradually grows in complexity to demonstrate how to control what `gcc` does.

LISTING 6.1 Linux File System Overview

```
#include <stdio.h>

main(void)
/*
**   Listing6.1.c – prints a line on the screen
*/
{
    puts("Prints on your screen.");
    return (0);
}
```

This simple program has a precompiler directive to include `stdio.h`, the standard I/O header file. It is then compiled and linked into an executable under /home/chief/bin. Instructions are passed to `gcc` through its line command operands:

```
> gcc -o ../bin/Listing6.1 Listing6.1.c
```

The `-o` tells gcc to put the output (executable) in /home/chief/bin and to name it Listing6.1. It also tells gcc to start by compiling the C program named Listing6.1.c. Note that gcc is not told where to find this C program, where the header is located, nor where the function named `puts` may be found. These defaults are described in detail in its man (short for manual) page.

```
> man gcc
```

or

```
> info gcc
```

After it is preprocessed, compiled, and linked (from now on I'll call the aggregate process *compiled*) the executable can be tested as follows:

```
> Listing6.1
Prints on your screen.
```

Note that you put the executable under /home/chief/bin because it is in your path, the set of subdirectories that are searched when you type a command at the command prompt. Unix users commonly create a bin subdirectory under their home directory. Linux automatically creates one for each new user and adds support for it in the default environmental control files.

You will define values that are global to more than one program. The maximum length of a buffer one program sends to the other, for example, is common to both.

You can define the length in both, but you run the risk of it being changed in one but not the other; it is best to define it in a header file and then include that header file in both programs that use it. This example uses only one program; it should help you understand how to create and include your own C header file.

Change directories into your `include` subdirectory and create a header named Listing 6.2.h that looks like the following:

```
/*
**   Listing6.2.h – a simple C language header file
*/
#define   BUFLEN   64
```

Just as the `#include` acts as a preprocessor directive to insert a named header file, the `#define` tells the precompiler to create a local variable named `BUFLEN` and to assign it the value `64`. Note that this is a string-level substitution. Every time the preprocessor finds the string `BUFLEN`, it substitutes the string `64`. Note that even though this is treated as a string, no substitution is made to the contents of quoted strings within the C program. The string

```
char str[] = {"A buffer of length BUFLEN is used as a parameter list."};
```

will not be changed by the preprocessorr, but

```
int   x   =   BUFLEN;
```

will be changed to

```
int   x   =   64;
```

Even though 64 is treated as a string of text at precompile time, it is interpreted at compile time as a numeric value to be assigned to the variable `x` at execution time. Listing 6.2 is an example that may help you understand these subtle differences.

LISTING 6.2 Your /home Directory

```
#include <stdio.h>
#include <Listing6.2.h>

main(void)
/*
**   Listing6.2.c – prints a different line on the screen
*/
{
    printf("Maximum buffer length BUFLEN = %d bytes\n", BUFLEN);
}
```

Because the precompiler is not likely to find your new header file, you must specify its location as a place to start looking for it:

```
> gcc -I../inc -o../bin/Listing6.2 Listing6.2.c
```

The `-I` (for include) tells the preprocessor to look in the local include directory /home/chief/inc first. Figure 6.2 shows what your home directory looks like and where each file is located at this point.

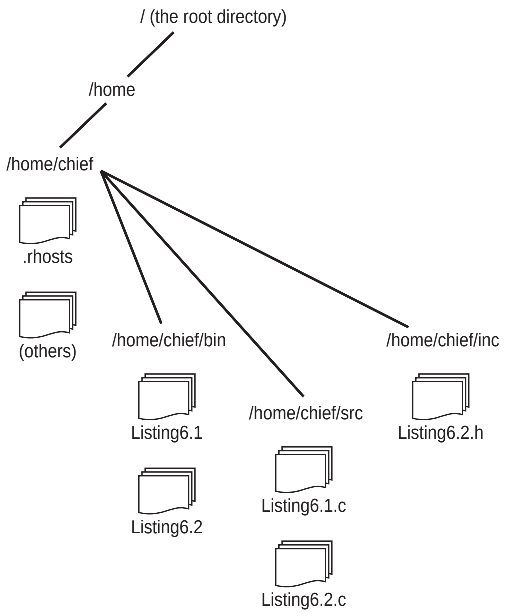

FIGURE 6.2 Your /home directory.

Note that stdio.h was found and inserted into the text of the file named Listing6.1.c without any help. Standard C header files are located under /usr/include and under its many subdirectories. Take a closer look at some of the examples in Chapter 3, "Inter-Process Communications," to see how header socket.h is included from the /usr/include/sys subdirectory, for example.

The function calls to printf and puts, you may have noticed, were resolved without mention. Standard functions are similarly organized in well-known library directories. Sometimes what appears to be a function call is really a reference to a macro. These are beyond the scope of this book, but interested readers are directed to the more advanced uses of the `#define` directive previously introduced.

At times, you will want to create your own functions and then direct the compiler to resolve any references to it. Listing 6.3 demonstrates the use of a local function named screen, which uses the standard function to print a string to the screen.

LISTING 6.3 Using a Local Function to Print on the Screen

```
#include <stdio.h>

main(void)
/*
**   Listing6.3.c – uses a local function to print on the screen
*/
{
    void screen(char *);

    screen("Indirect print to screen.");

}

void screen(char * str)
{
    puts(str);
    return;
}
```

At times, you also will want to let more than one main program have access to your function named screen. You can use the compiler to create only the object file from a C file that contains only the function, as in Listing 6.4a.

LISTING 6.4A Using a `puts` Call Directly to Print on the Screen

```
#include <stdio.h>

void screen(char * str)
/*
**   Listing6.4a.c – function uses puts to print to screen
*/
{
    puts(str);
    return;
}
```

You compile this using the `-c` operand, without specifying any output information.

```
> gcc -c Listing6.4a.c
```

This creates a file named Listing6.4a.o, the object file with only the screen function in it. For the main program, see Listing 6.4b.

LISTING 6.4B Using the Screen Function's Object File in a Main Program

```
main(void)
/*
**   Listing6.4b.c – indirectly prints a line to your screen
*/
{
void screen(char *);

    screen("Prints to your screen.");
}
```

This time, compile the main program with a reference to the object file, as well as the C program.

```
> gcc -o../bin/Listing6.4b Listing6.4a.o Listing6.4b.c
```

This allows the function named `screen` to be found inside the object file named Listing6.4a.o, while resolving the reference to it in Listing6.4b.c.

```
> Listing6.4b
Prints to your screen.
```

Note that this function can be referenced in several C programs, each one compiled with a request for Listing6.4a.o in the line command operands for gcc.

Remembering the names of all these functions becomes easier if you name them screen.c and screen.o, for example. It gets even easier if you put such detail into a separate file of instructions on how to compile these programs. This instruction file is called a makefile because it is used by the make utility, described in the next section.

Using the make Utility

The `make` command is a very powerful utility used to contain the complexity of compiling large sets of programs. Large software development projects use make to its fullest advantage, with both local and global variables and dependencies, to build a package of executables from perhaps hundreds of C programs, header files, and function libraries. In this book, we merely scratch the surface of its power, creating only the simplest kind of instruction file. Listing 6.5 is such a file that can be used to compile every program in this chapter. It must be named makefile, or Makefile, and reside in /home/chief/src.

LISTING 6.5 Contents of the File Named makefile

```
all:   Listing6.1 Listing6.2 Listing6.3 Listing6.4
Listing6.1:
   gcc -o../bin/Listing6.1 Listing6.1.c
Listing6.2:
   gcc -I../inc -o../bin/Listing6.2 Listing6.2.c
Listing6.3:
   gcc -o../bin/Listing6.3 Listing6.3.c
Listing6.4: screen
   gcc -o../bin/Listing6.4b Listing6.4a.o Listing6.4b.c
screen:
   gcc -c Listing6.4a.c
```

Each program is represented by a pair of lines. The first entry on the first line is the rule name. After the colon comes dependencies, if any. The second line is the command string to execute. (It must begin with a tab.) Using the make utility, you can compile the program in Listing 6.1 as follows:

```
> make Listing6.1
gcc –o ../bin/Listing6.1 Listing6.1.c
```

The command line associated with the rule labeled Listing6.1 is executed. When you issue

```
> make Listing6.4
gcc -c Listing6.4a.c
gcc –o ../bin/Listing6.4b Listing6.4a.o Listing6.4b.c
```

Listing6.4b depends upon Listing6.4a, where the screen function is defined. Listing6.4a must be compiled into Listing6.4a.o before Listing64b can be compiled and linked into an executable.

The Holy Grail of build masters everywhere is to create a makefile set that corresponds to their hundreds of interdependent source and header files so their job can be reduced to issuing

```
> make
```

Then they can go home for the night, arriving the next morning ready to pore over the resulting output stream to verify that everything worked as planned.

When you issue the `make` command without anything else, the first rule in the makefile is selected. In this case, this rule is named all, which has all four of the other listing rules listed as dependencies, so everything is recompiled.

Backup and Recovery

A small number of C programs can represent a lot of time and effort on your part. Creating a simple backup and recovery scheme is highly recommended. A common 3 1/2-inch disk holds almost a megabyte and a half of uncompressed data. Regularly using the Unix `tar` (archive) command, or even a simple copy operation, can be a significant defense against the loss of a hard drive.

A large number of files can be backed up to the hard drive of another machine in the cluster by archiving these files to the local temporary file area (/tmp) and then copying that archive file to a subdirectory (/backups, for example) on some remote machine's hard drive.

For a straight file copy operation, you can use a DOS-formatted disk, but you must mount it (as root) so that Linux knows what it is.

Under /mnt there will probably be a subdirectory named floppy. If not, create one:

```
> cd /mnt
> mkdir floppy
```

Now mount the disk, copy the source files directly to it, and dismount it again:

```
> mount -t msdos /dev/fd0 /mnt/floppy
> cd /home/chief/src
> cp *.c /mnt/floppy/
> cd ../inc
> cp *.h /mnt/floppy/
> umount /mnt/floppy
```

Finally, remove the disk and check its contents on a DOS (or Windows) machine. Similarly, you can restore these files as follows:

```
> mount -t msdos /dev/fd0 /mnt/floppy
> cd /mnt/floppy
> cp * /tmp/
> umount /mnt/floppy
```

All your C programs and header files should be under the /tmp directory. Many people do not take the time to actually try to restore their files from a backup medium. It is a good idea to make sure that backup and restore procedures both work to your satisfaction before placing your confidence in them. It's your time being saved here.

Another backup and restore to disk process uses the `tar` (archive) command. You do not need to use a command to mount the disk in this case, but you still have to be root to write to that device. (And you must, of course, put the disk into the drive!)

```
> cd /home/chief
> tar -zcvf /dev/fd0 .          [Note the final dot]
```

This copies everything under chief's home directory, in compressed archive file format, onto the disk. You can restore the data as follows:

```
> cd /tmp
> tar -zxvf /dev/fd0
```

You can check the contents of such a disk with yet another minor difference in operands.

```
> tar -ztvf /dev/fd0
```

The differences in the operands of these commands are so subtle that you can easily destroy the contents of an existing archive disk by specifying `c` instead of `x` on the command line. Remember, `c` is for create, `x` is for extract, and `t` is for tell me what's in it.

Summary

This chapter started with an introduction to the Linux file system and focused on the home directory of your nonroot user ID, named chief, which created a subdirectory for C programs and header files. You noted that your executables will be under /home/chief/bin, which was created by the Linux OS when you asked that chief be created as a user ID.

The C compiler came next. You learned what it does and how these processing steps related to the software you will be developing in later chapters of this book. Some programming examples were offered to clarify the relationships between precompiler directives, header files, and C language source files. Each of the compiler commands necessary to compile the sample programs was presented and discussed in detail. The make utility was presented as a remedy for having to remember all those compiler command-line options.

Finally, you learned two ways to back up files to disk to mitigate the possibility of a lost hard drive—an ever-present possibility if you are building your cluster from salvage PCs.

Further Reading

- Oram, A., and S. Talbott. *Managing Projects with Make*. Sebastopol, CA: O'Reilly and Associates, Inc., 1992.

7 The Master-Slave Interface Software Architecture

IN THIS CHAPTER

This chapter marks a turning point in the book. In earlier chapters, you selected the cluster system's hardware, installed and configured its operating system, and learned several techniques used to develop cluster system software. In this chapter, you focus on developing a *concurrent* server, one that handles incoming queries concurrently. It begins as a simple, serialized processing loop and evolves into a master-slave configuration, a software architecture that is well suited to your cluster system's hardware.

Each change to a server's architecture ought to show some improvement over the earlier version. This chapter starts by introducing a client that drives the server and gives you an indication of overall system performance. You take a much closer look at performance measurement and analysis in Chapter 8, "External Performance Measurement and Analysis," when you fine-tune the system.

Each version of the server in this chapter is driven by the same client. Let's take a look at that first.

The Client Process

The client requests a random line of text from a file that is located on the server's disk. It sends the server a query containing a filename and a line number. The server opens the file, retrieves the line of text, and sends it back to the client in its response.

As simple as this scenario is, it involves the acceptance of a client's socket connection, access to a shared resource, and a response back to the client. That's pretty much what all servers do, and you will continue using this scenario as the sample server grows in complexity. The source code for the client is shown in Listing 7.1. Much of the checking and error-reporting code you might normally add has been omitted for clarity and simplicity.

You can specify how many packets are in the test stream and how much time separates them. The time between each packet is called the inter-arrival time, abbreviated IAT in the source listing. If you gradually decrease the IAT, you will find a point where the server fails because it cannot process packets in that short amount of time. This value is a good estimate of a server's overall performance, and you can expect improved versions of the server to handle comparatively shorter IAT values.

LISTING 7.1 Client Requests a Specific Line of Text from a Remote File

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 9876
#define NUM 50

main(int argc, char *argv[])

{
   /*
   **   Listing7.1.c - Client: request a line from a remote file
   */
   struct timeval bef, aft;   /* timer value before/after */
   struct sockaddr_in sock;   /* structure for socket */
   struct hostent *hp;        /* structure for IP address */
   double random;             /* random reals: 0.0->1.0 */
   long timeleft;             /* time remaining in IAT */
   long secs=0L;              /* inter-arrival time seconds */
   long usec=0L;              /* inter-arrival microseconds */
   int pids[NUM];             /* subtask process ids */
   int opt=1;                 /* setsockopt parameter */
```

LISTING 7.1 Continued

```
int fd;                    /* socket file descriptor */
int ii, jj, kk=0;
char buf[80], *p;

srand((unsigned int) getpid()); /* seed rand() */

/*
**   Operands are remote HOST name and IAT (Inter-Arrival Time)
*/
if(argc != 3)
{
   printf("\n\tUsage: %s <HOST> <IAT>\n\n", argv[0]);
   exit(-1);
}

/* uSeconds part of inter-arrival */
if((p=strchr(argv[2], '.')) != NULL)
{
   *p = '\0'; /* end whole number at decimal */
   p++; /* bump pointer to start of mantissa */
   while(strlen(p) < 6)
      strcat(p, "0"); /* pad out to 6 digits */
   p[6] = '\0';  /* truncate to 6 digits max */
   usec = atol(p);
}
secs = atol(argv[2]); /* seconds part of IAT */

/*
**   LOOP: send and receive NUM packets
*/
for(ii=0; ii<NUM; ii++)
{
   /* get time before send */
   gettimeofday(&bef, NULL);

   /* random integer from 1 through 5 */
   random = rand() / (double) RAND_MAX;
   jj = (int) ((double) (5.0) * random) + 1;
   if(jj == 6)
      jj = 5;
```

LISTING 7.1 Continued

```
      sprintf(buf, "/tmp/sample.txt %d", jj);

      if((pids[kk++]=fork()) == 0)
      {
         /* set up socket info for connect */
         fd = socket(AF_INET, SOCK_STREAM, 0);
         memset((char *) &sock, 0, sizeof(sock));
         hp = gethostbyname(argv[1]);
         memcpy(&sock.sin_addr, hp->h_addr,  hp->h_length);
         sock.sin_family = hp->h_addrtype;
         sock.sin_port = htons(PORT);
         setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
            (char *) &opt, sizeof(opt));

         /* connect to server */
         connect(fd, (struct sockaddr *) &sock, sizeof(sock));
         send(fd, buf, strlen(buf)+1, 0);
         buf[0] = 0; /* clear buffer */
         recv(fd, buf, sizeof(buf), 0);

         /* print out response to our query */
         printf("\tLine %d: '%s' ", jj, buf);
         /* check for correct line */
         p = strrchr(buf, ' ') + 2;
         if(jj != atoi(p))
            printf("*");
         printf("\n");
         close(fd);
         exit(0);
      }

      /*
      **   Sleep for remainder of IAT
      */
      gettimeofday(&aft, NULL);
      aft.tv_sec -= bef.tv_sec;
      aft.tv_usec -= bef.tv_usec;
      if(aft.tv_usec < 0L)
      {
         aft.tv_usec += 1000000L;
         aft.tv_sec -= 1;
```

LISTING 7.1 Continued

```
      }
      bef.tv_sec = secs;
      bef.tv_usec = usec;
      bef.tv_sec -= aft.tv_sec;
      bef.tv_usec -= aft.tv_usec;
      if(bef.tv_usec < 0L)
      {
         bef.tv_usec += 1000000L;
         bef.tv_sec -= 1;
      }
      timeleft = (bef.tv_sec * 1000000L ) + bef.tv_usec;
      if(timeleft < 0)
      {
         printf("\tERROR: A higher IAT value is required - exiting.\n");
         break;
      }
      usleep(timeleft);
   }
   for(ii=0; ii<kk; ii++)
      wait(pids[ii]);
   puts("Done.");
   return(0);
}
```

The number of packets in a stream is defined in the source as NUM, which must be large enough to get beyond any time delays that occur because of the initial socket connection, but not so large as to make testing tedious. The inter-arrival time is a command line argument, for convenience. To test the system, start a server on one of your remote machines, named gamma in this example:

```
> Listing7.1   gamma   0.025
```

The preceding command starts a client that sends queries to a server process running on gamma 25 milliseconds. All queries are sent 25 milliseconds apart, equally separated in time. The next chapter introduces some more realistic inter-arrival time distributions when you expand the client's capabilities.

The Serial Server Process

A server process can handle incoming transactions in several ways; the most straightforward is a simple receive-process-respond-repeatloop, as shown in Listing 7.2.

LISTING 7.2 The Serial Server Approach

```
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 9876

main(void)
{
   /*
   **   Listing7.2.c - Server: Return one line from a file
   */
   struct sockaddr_in sock; /* structure for socket() */
   int socklen=sizeof(sock);/* struct len for accept() */
   FILE *fp;                /* sample.txt file pointer */
   int acc;                 /* accept file descriptor */
   int cli;                 /* client file descriptor */
   int line;                /* requested line number */
   int opt=1;               /* setsockopt parameter */
   char buf[80];            /* socket receive buffer */
   char file[32];           /* text file from client */

   /* create a (server) socket to accept a client */
   acc = socket(AF_INET, SOCK_STREAM, 0);

   /* bind socket to port */
   sock.sin_family = AF_INET;
   sock.sin_port = htons(PORT);
   sock.sin_addr.s_addr = htonl(INADDR_ANY);
   bind(acc, (struct sockaddr *) &sock, socklen);
   setsockopt(acc, SOL_SOCKET, SO_REUSEADDR,
      (char *) &opt, sizeof(opt));

   while(1)
   {
      listen(acc, 5);

      /* get a random line of data requested by the client */
      cli = accept(acc, (struct sockaddr *) &sock, &socklen);
```

LISTING 7.2 Continued

```
      recv(cli, buf, sizeof(buf), 0);

      /* open the requested file */
      sscanf(buf, "%s %d", file, &line);
      fp = fopen(file, "r");
      while(line—)
         fgets(buf, sizeof(buf), fp);
      fclose(fp);

      buf[strlen(buf)-1] = '\0';
      send(cli, buf, strlen(buf)+1, 0);
      close(cli);
   }
}
```

This server's query contains the name of a text file and a line number. The server opens the file, reads the required number of lines of text, sends the last line read as its response, and closes the file.

Each line in this sample file differs only by the line number it contains. A five-line file named sample.txt might appear as follows:

sample.txt

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
```

The response to the query

```
/tmp/sample.txt 3
```

would be

```
This is line 3
```

If you actually run tests on this client/server pair, start the server first on the target machine and specify its name on the command line when you start the client. Begin with a high inter-arrival time, as in the example, and reduce it until the server fails to return a complete list of responses. Note that you should not have to stop and restart the server each time you rerun the client process. Leave the server process running while you rerun the client with different IAT values.

This simple server has its limitations, handling only one incoming query at a time, which can be overcome through exploitation of parallelism. In this case, exploitation of parallelism means handling more than one query at a time. Our first improvement to the server has it creating a subtask to handle each incoming query. This will let each subtask read the text file in parallel with others, reducing the overall system-response time.

The Concurrent Server Process

You can use several approaches to converting a serial server to a parallel, or *concurrent,* server. The first method establishes a subtask to handle each incoming query, exploiting parallelism. Listing 7.3 is an example of such a server program.

LISTING 7.3 Concurrent Server Spawns a Subtask for Each Incoming Query

```
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 9876

main(void)
{
   /*
   **   Listing7.3.c - Concurrent Server: subtask per query
   */
   struct sockaddr_in sock; /* structure for socket() */
   int socklen=sizeof(sock);/* struct len for accept() */
   FILE *fp;                /* sample.txt file pointer */
   int acc;                 /* accept file descriptor */
   int cli;                 /* client file descriptor */
   int line;                /* requested line number */
   int opt=1;               /* setsockopt parameter */
   char buf[80];            /* socket receive buffer */
   char file[32];           /* text file from client */

   /* create a (server) socket to accept a client */
   acc = socket(AF_INET, SOCK_STREAM, 0);

   /* bind socket to port */
   sock.sin_family = AF_INET;
```

LISTING 7.3 Continued

```
   sock.sin_port = htons(PORT);
   sock.sin_addr.s_addr = htonl(INADDR_ANY);
   bind(acc, (struct sockaddr *) &sock, socklen);
   setsockopt(acc, SOL_SOCKET, SO_REUSEADDR,
      (char *) &opt, sizeof(opt));
   listen(acc, 5);

   while(1)
   {
      /* await connection from remote client */
      cli = accept(acc, (struct sockaddr *) &sock, &socklen);

      /* get a random line of data requested by the client */
      if(fork() == 0)
      {
         recv(cli, buf, sizeof(buf), 0);

         /* open the requested file */
         sscanf(buf, "%s %d", file, &line);
         fp = fopen(file, "r");
         while(line—)
            fgets(buf, sizeof(buf), fp);
         fclose(fp);

         buf[strlen(buf)-1] = '\0';
         send(cli, buf, strlen(buf)+1, 0);
         close(cli);
         exit(0);
      }
      close(cli);
   }
}
```

The benefit to creating a subtask for each incoming packet is the exploitation of any parallelism in packet processing. In this case, the shared resource is read-only; many subtasks can read the text file simultaneously, improving the performance by reducing the response time. A cost is associated with the creation of separate subtasks for every incoming packet, however. You can reduce this overhead by creating enough subtasks in advance to handle your expected peak load (shortest inter-arrival time), at the cost of adding some interprocess communication overhead to manage the parallel query processing, as in Listing 7.4.

LISTING 7.4 Server Creates a Pool of Subtasks in Advance to Handle Incoming Query Stream

```
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/signal.h>

#define PORT 9876
#define FALSE 0
#define TRUE 1
#define MAX 5

int Running = TRUE; /* boolean: reset in control_C() */

main(void)
{
   /*
   **   Listing7.4.c - Concurrent Server: Subtask Pool
   */
   struct area {            /* flag for every subtask */
      int flag[MAX];        /* 0:idle, 1:good, -:exit */
   } area, *aptr;
   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = sizeof(struct area);
   key_t key = 0x5a5a5a5a;
   int shmid;               /* shared memory area id */

   void control_C();        /* ctrl-C signal handler */
   struct sockaddr_in sock; /* structure for socket() */
   int socklen=sizeof(sock);/* struct len for accept() */
   FILE *fp;                /* query text file pointer */
   fd_set fds;              /* select waits for client */
   int acc;                 /* accept file descriptor */
   int cli;                 /* client file descriptor */
   int line;                /* requested line number */
   int opt=1;               /* setsockopt parameter */
```

LISTING 7.4 Continued

```
   int ident;               /* internal subtask id */
   int pids[MAX];           /* subtask process ids */
   char file[32];           /* text file from client */
   char buf[80];            /* socket receive buffer */

   /* initialize ^C handler */
   signal(SIGINT, control_C);

   /* create a (server) socket to accept a client */
   acc = socket(AF_INET, SOCK_STREAM, 0);

   /* bind socket to port */
   sock.sin_family = AF_INET;
   sock.sin_port = htons(PORT);
   sock.sin_addr.s_addr = htonl(INADDR_ANY);
   bind(acc, (struct sockaddr *) &sock, socklen);
   setsockopt(acc, SOL_SOCKET, SO_REUSEADDR,
      (char *) &opt, sizeof(opt));
   listen(acc, 5);

   /* get the shared memory area */
   shmid = shmget(key, size, flag);
   /* attach subtask's shared memory array */
   aptr = (struct area *) shmat(shmid, 0, 0);

   /* pre-establish subtask pool */
   for(ident=0; ident<MAX; ident++)
   {
      aptr->flag[ident] = 0; /* set flags idle */

      if((pids[ident]=fork()) == 0) /* SUBTASK */
      {
         /* attach parent's shared memory arrray */
         aptr = (struct area *) shmat(shmid, 0, 0);
         /* notice: shmid is still set correctly */

         /* nullify ^C handler */
         signal(SIGINT, SIG_DFL);

         /*
         **   Poll memory array for a non-zero flag
```

LISTING 7.4 Continued

```
      */
      while(1)
      {
         if(aptr->flag[ident] == 0)
         {
            usleep(1);   /* release processor */
            continue;    /* stay in poll loop */
         }
         else if(aptr->flag[ident] < 0)
         {
            exit(0); /* subtask's normal exit */
         }
         else /* action flag must be positive */
         {
            cli = accept(acc, (struct sockaddr *) &sock, &socklen);
            /* receive an incoming query */
            recv(cli, buf, sizeof(buf), 0);

            /* open the requested text file */
            sscanf(buf, "%s %d", file, &line);
            fp = fopen(file, "r");
            while(line—)
               fgets(buf, sizeof(buf), fp);
            fclose(fp);

            /* send back a response */
            buf[strlen(buf)-1] = '\0';
            send(cli, buf, strlen(buf)+1, 0);
            close(cli);

            /* set to available */
            aptr->flag[ident] = 0;
         }
      }
   }
}

/*
**   Parent task passes incoming connections to a subtask
*/
```

LISTING 7.4 Continued

```
   while(Running) /* set FALSE in control_C signal handler */
   {
      FD_ZERO(&fds);
      FD_SET(acc, &fds);
      /* block until client is ready for connect */
      if(select(acc+1, &fds, NULL, NULL, NULL) < 0)
         break;
      /*
      **   Assign incoming query to first available subtask
      */
      for(ident=0; ident<MAX; ident++)
      {
         if(aptr->flag[ident] == 0)
         {
            aptr->flag[ident] = 1;
            break;
         }
      }
   }

   /* wait for each subtask exit */
   for(ident=0; ident<MAX; ident++)
   {
      aptr->flag[ident] = -1;
      waitpid(pids[ident]);
   }

   /* remove unused shared memory area and exit */
   shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
   return(0);
}
/*
**   ^C signal handler
*/
void control_C()
{
   Running = FALSE; /* reset flag to allow exit */
   return;
}
```

So far, you have seen three different approaches to server process software architecture. These were each tested over a 10BASE-2 (thinnet) network, between a beefed-up 486-based machine, and on an older 386. You sent streams of twenty packets, randomly querying a 99-line text file. The fastest reliable inter-arrival times were 8, 7, and 4 milliseconds on Listings 7.2, 7.3, and 7.4, respectively. A lot of overhead occurs in creating a subtask to process every incoming query, as shown by the less-than-stellar improvement from adding that feature to a serial server. The big return came, as expected, from moving the overhead into the process initialization phase, by establishing a fixed number of processes up front, and hoping the incoming queries don't swamp them. You might be able to improve this server's performance even further by finding an optimal number of preestablished subtasks for your processor and network speed and amount of available memory.

Note that an interrupt handler is added to Listing 7.4 so that you can use Ctrl+C to exit the server during testing. When that keystroke is detected, the subtasks and the parent exit gracefully.

The Distributed Server Process

Notice that the inter-process communication between the master process and its slave subtasks in Listing 7.4 is through a shared memory space (see Figure 7.1a). This low-overhead communication link is possible because the master and all its slaves are on the same machine, running in the same physical memory. (A memory bus transfers data much faster than an external network.) The benefit from parallel processing is limited, however, because this single machine has only one CPU. If you spread the slaves across your cluster system, as in Figure 7.1b, you can gain in parallelism through the simultaneous use of multiple processors. Performance will improve if query processing required high CPU utilization. Performance will suffer, however, because of the added cost of interprocess communication across the network.

The client will access the server (the master process, in this case) the same way it did in the earlier examples:

```
> Listing7.2   gamma   0.25
```

The server's master process will create its subtasks in advance, as the server did in Listing 7.4. It can do so by treating its slaves as second-tier servers, as shown in Figure 7.1b. The master process must establish communication with each slave process, so it should be aware of other machines on the network in advance. Instead of accessing a list of these other processors, you can generalize this process by having the master process issue a broadcast message to all possible slave ports.

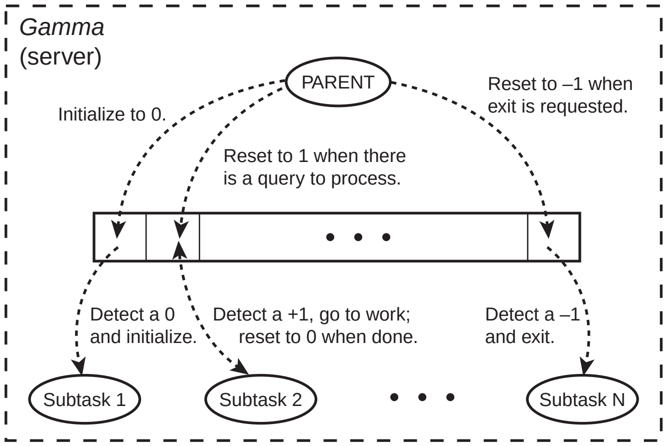

FIGURE 7.1A Inter-process communication via shared memory.

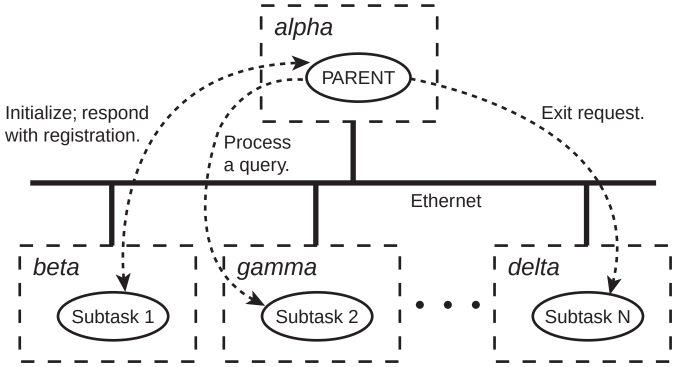

FIGURE 7.1B Inter-process communication over a network.

Recall in Chapter 3, "Inter-Process Communication," that you caused a remote service to start by sending a UDP datagram to its well-known port address on a remote machine (Listings 3.8a and 3.8c). You can generalize this concept to create all possible remote slave services in advance by defining their well-known ports on every remote machine on your network and then sending a broadcast UDP datagram, as shown in the short example of Listing 7.5. Every machine on the network will receive the datagram, and every machine on the network will start the remote slave processes in the same manner as the examples in Chapter 3. (Be sure to add definitions for Listing 7.6 in every remote system's /etc/services and /etc/inetd.conf files, a process that is also described in Chapter 3.)

A broadcast message is different from an ordinary message in two ways. The socket (UDP datagram in this case) must be prepared via a call to `setsockopt` to accept these special messages, and you must use a special wildcard IP address. Recall from Chapter 3 that the first part of an IP address defines the network, whereas the last part defines the node, usually a machine on the network. The special node value of 255 (or FF in hex) is the wildcard node address that is interpreted as all nodes; all nodes accept such messages, regardless of their particular node address. Our network has IP addresses of 10.0.0.1 through 10.0.0.4; therefore, 10.0.0.255 would be the address used in a broadcast message.

LISTING 7.5 Master Starts All Possible Slave Processes via UDP Broadcast Datagram

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

#define INIT 3579
#define WORK 9753

unsigned char broadcast[4] = {0x0A, 0x00, 0x00, 0xFF}; /* 10.0.0.255 */

main(int argc, char *argv[])
char *argv[];
int argc;
/*
**   Listing7.5.c - MASTER: UDP datagram activates SLAVE services
*/
{
   int len = sizeof(struct sockaddr_in);
   struct sockaddr_in init;
   struct sockaddr_in work;
   struct hostent *hp;
   int ifd, wfd;
   int opt=1;
   char host[32];
   char buf[80], *p;

   /* establish initialization socket */
   ifd = socket(AF_INET, SOCK_DGRAM, 0);
   memset((char *) &init, 0, sizeof(init));
```

LISTING 7.5 Continued

```
   init.sin_family = AF_INET;
   init.sin_port = htons(INIT);
   memcpy(&init.sin_addr, broadcast, 4);
   setsockopt(ifd, SOL_SOCKET, SO_BROADCAST,
             (char *) &opt, sizeof(opt));

   /* establish the work-to-do UDP socket struct */
   wfd = socket(AF_INET, SOCK_DGRAM, 0);
   memset((char *) &work, 0, sizeof(work));
   work.sin_addr.s_addr = htonl(INADDR_ANY);
   work.sin_port = htons(WORK);
   work.sin_family = AF_INET;
   bind(wfd, (struct sockaddr *) &work, len);

   /* msg starts remote SLAVES */
   gethostname(host, sizeof(host));
   if((p=strchr(host, '.')) != NULL)
      *p = '\0'; /* trunc .domain */
   sprintf(buf, "%s %d", host, WORK);
   sendto(ifd, buf, strlen(buf)+1, 0,
         (struct sockaddr *) &init, sizeof(init));
   close(ifd);

   /*
   **   Display any registration response info
   */
   while(1)
   {
      buf[0] = '\0'; /* clear buffer */
      recvfrom(wfd, buf, sizeof(buf), 0,
        (struct sockaddr *) &work, &len);
      printf("\tReceived: '%s'\n", buf);
   }
   close(wfd);
   return(0);
}
```

In this example, the slave processes are configured as internetworking services, which are started automatically by the internetworking daemon, named inetd (covered in Chapter 3). Because the master process has no knowledge of which slaves might respond to this general call, each slave process must register with the master process, making it aware of its presence. They do so by sending a registration message (a UDP datagram) back to the master, over the network, as shown in Listing 7.6. As each registration message is received, the master process adds a record of the active slave process to its internal table of available workers.

LISTING 7.6 Slave Activated by UDP Datagram to Its Defined Service Port

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

main(void)
/*
**   Listing7.6.c - SLAVE: activated by a UDP datagram to port 3579
*/
{
   struct sockaddr_in sock;
   int len=sizeof(sock);
   struct hostent *hp;
   int port;
   int fd;
   char host[16];
   char buf[80];

   /*
   **   Get host and port from MASTER
   */
   read(0, buf, sizeof(buf));
   sscanf(buf, "%s %d", host, &port);

   /* initialize a UDP socket struct */
   fd = socket(AF_INET, SOCK_DGRAM, 0);
   memset((char *) &sock, 0, sizeof(sock));
   hp = gethostbyname(host); /* MASTER */
   memcpy(&sock.sin_addr, hp->h_addr,   hp->h_length);
   sock.sin_port = htons(port);
   sock.sin_family = AF_INET;

   /*
   **   Register with MASTER
```

LISTING 7.6 Continued

```
   */
   gethostname(buf, sizeof(buf));
   sendto(fd, buf, strlen(buf)+1, 0,
     (struct sockaddr *) &sock, sizeof(sock));

   /* . . . further processing */

   close(fd);
   return(0);
}
```

The source listings are intended to be suggestive of how a master process can remotely start a set of slave processes with no foreknowledge of how many there might be or where they may be located. After the broadcast message is out, the registration responses come in, allowing the master process to build a dynamic membership availability list.

How the Master-Slave Interface Works

The master has two phases: initialization and processing handoff. In the initialization phase, described previously, the remote slaves are activated and registered. The result is a utilization table, with one entry for each registered slave process. Each entry contains a machine name, an IP address, and a projected utilization index, or UI. (The UI is an integer from zero to one hundred: zero being idle and 100 being saturated.) In the processing handoff phase, each query is received and passed on to the slave process on the least busy machine in the utilization table.

Local performance data are gathered at each remote machine by subtasks of the slave processes and passed back to the master process inside each acknowledgement packet—the receipt of each query. A danger exists in this technique, however. If a machine gets so busy in relation to all the others that it is no longer chosen for work, the table may continue to show it as busy, although it dropped down to idle long ago. If it is never chosen for work, it doesn't acknowledge receipt, and the master's table isn't updated. The local data get stale and unreliable, yet decisions are still being made based on its content.

One way to circumvent this problem is to have each slave send regular reports of its utilization, but the potentially excessive network traffic associated with these reports can adversely affect the overall system performance. In another method, slaves report only significant changes in utilization, and only when such changes occur. A 5% change in the calculated index, for example, might require the slave to send an update to the master. Under a highly irregular query load, however, the utilizations can also fluctuate to an extent that causes excessive network traffic.

It is quite reasonable for a slave to acknowledge receipt of each query, and piggybacking utilization data onto such packets is a low-cost solution. The next chapter describes an experimental UI calculation that should overcome the stale data problem without the added cost or concern.

You may have noticed that the preestablished subtasks example in Listing 7.4 detected the readiness of an incoming query before giving control to one of its subtasks. The chosen subtask accepted and received the query and responded to it independently of the parent task. This technique exploits a high degree of parallelism and frees the parent task to focus on the incoming query stream. We will use an over-the-network equivalent technique so that the master process can sustain the highest possible incoming query load. See the diagram in Figure 7.2.

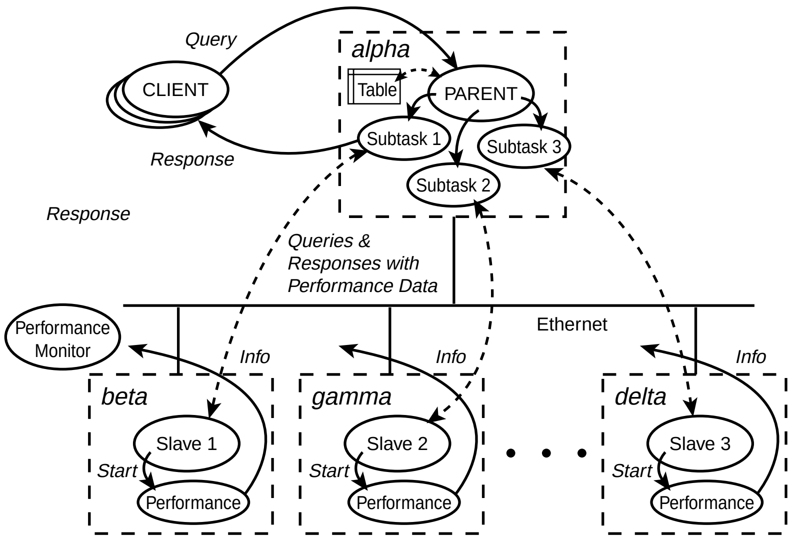

FIGURE 7.2 The master-slave interface.

When the number of remote slave processes is known, the master will preestablish a local subtask dedicated to each. The parent will again detect the readiness of a query, choose a remote slave based on its local utilization table data, and assign the query to its counterpart subtask. This gives you the benefit of parallelism that you want and moves the burden of utilization table updates away from the parent and onto the associated subtask.

There is the further benefit of fault detection: the communication link from local subtask to remote slave process can indicate a process failure and can serve as a link to confirm or deny a failure, should that suspicion arise. Similarly, any of the slave processes can detect a failure in the master—or at least in its associated subtask. The local subtask can restart a failed remote slave process by sending a UDP datagram, just as the parent did at initialization time, without the parent's involvement.

Complete source listings of these processes are in the Appendix at the back of this book. The source code is also available at `www.samspublishing.com` and may be downloaded at no cost. You are encouraged to experiment with solutions to these design issues, and having this code as a starting point should help you do so.

System Limitations

The most obvious limitation is the master processor as a single point of failure. If the master fails, so does the entire server. A fault-tolerant solution would detect failure of the master and hold an election. One of the slave systems would assume the role of master and assume the IP address of the potentially failed machine. When queries come from an external client pool, there is often only one published IP address and port number where all the queries are sent. One solution publishes an IP address that the master processor assumes as an *alias*, accepting packets addressed to this IP. When a master processor fails, another processor can assume that alias by sending a gratuitous ARP (address resolution protocol) message, which is beyond the scope of this book. (See the "Further Reading" section later in this chapter.)

Summary

The focus of this chapter is the server. You start simply and build up to the master-slave interface server concept that adapts well to cluster architectures.

This chapter started with a client and a simple application. The simplest server version was introduced and converted to a concurrent server—one that exploited the parallelism of incoming queries by spawning a subtask to handle each of them as they arrived. Finally, the more sophisticated approach of creating a pool of subtasks in advance was used. Finally, this subtask pool was extended into the realm of the cluster computer architecture by creating the same number of subtasks as there were remote processors. This architecture exploits the parallelism afforded by a multicomputer, or cluster architecture.

In the next chapter, you will find out just how well the system performs, at a level of detail that will help you fine-tune this system to meet the expected needs of your clientele.

Further Reading

- Stevens, W. Richard. *TCP/IP Illustrated*. Vol. 1. Reading, MA: Addison-Wesley, 1994.

8 External Performance Measurement and Analysis

IN THIS CHAPTER

Why measure the external performance of a server? The short answer is because that is what matters to your clients: the overall system-response time.

The server is your cluster system—its hardware and its software. The clients are represented by one or more processes running on some external machines that you know little or nothing about. You want to give your clients the best possible service, and clients usually measure their quality of service in terms of the *response time*, the amount of time they have to wait for a server to respond to a query.

You can measure the response time from the client's perspective by viewing the server from the outside, as if it were a black box, the only discernable features being its inputs and outputs. Looking at a server's external performance means measuring the time it takes to get a suitable response to your query, on average, and under a variety of loading conditions.

This chapter shows you how to design a query-generating client that will simultaneously stress your server and report on its performance. You saw a simple one in the previous chapter (refer to Listing 7.1) that sent queries at a constant rate. In this chapter, three more inter-arrival time (IAT) distributions are used to stress the server and update the client in Listing 7.1 accordingly.

When external performance is less than stellar, you'll need an alternative approach. In the next chapter, you'll look inside the server, viewing it as a white box, and discover where and how all the response time is being spent, reducing it wherever possible.

Query Generation

Recall the client program in Listing 7.1. It records the time of day and then creates a subtask to send each query. When the parent regains control, it calculates the amount of time remaining in the IAT and waits before sending another query. This client implements a *constant* IAT—each query follows the next after some fixed amount of time. If only our world were so orderly!

Arrival rates in cases so mundane as a bank teller's window were of such interest to early time-and-efficiency experts that a whole branch of applied mathematics, called *queueing theory*, was a direct result of it. Queueing theory allows bank managers to predict how many tellers will be needed at a particular time of day for a given day of the week. Telephone companies can determine how many central offices it can connect to a trunk line, and airlines can decide how many flights to schedule, all based on records of their customers' historical behavior. Computer scientists use queueing theory to predict the bandwidth needed between nodes of a network and to determine how many users can be supported by a particular server. This is powerful stuff, but it's beyond the scope of this book. A classic text on the subject is cited in the "Further Reading" section at the end of this chapter.

Those early efficiency experts sat in view of a teller's window and wrote down the exact time that each bank customer arrived. A modern-day observer is more likely to be a small metal box at the side of the road. It has a long black rubber sensor stretching out across a lane of traffic; as each car's tires roll over the sensor, the time of day is recorded. The box is picked up at day's end, and traffic flow is analyzed based on the observed data. But the analysis is the same.

Taking the differences between successive arrival times produces a list of IATs that can be ordered, categorized, and tabulated. Categories represent the number of times that an IAT between zero and ten seconds was observed, the number of times that an IAT between ten and twenty seconds was observed, and so on, as shown in the first two columns of Table 8.1.

TABLE 8.1 A Discrete Inter-Arrival Time Distribution

IAT (seconds)	Number of Observations	Percentage of Observations	Probability of Observation	Cumulative Probabilities
1-10	52	26	0.26	0.26
11-20	42	21	0.21	0.47
21-30	32	16	0.16	0.63
31-40	26	13	0.13	0.76
41-50	16	8	0.08	0.84
51-60	12	6	0.06	0.90
61-70	8	4	0.04	0.94
71-80	6	3	0.03	0.97
81-90	4	2	0.02	0.99
91-100	2	1	0.01	1.00
	Total = 200	Total = 100%	Total = 1.00	

To analyze the data in this table, total the number of observations recorded in the second column. In a third column, calculate the percentages of that total each category represents; they should add up to 100 percent. You can treat each category's percentage as a probability by dividing it by 100, which yields a fourth column of values, each between zero and one. In the rightmost column, list cumulative probabilities—the running totals from the category probabilities in the fourth column.

NOTE

This analysis technique is from *System Simulation*, a book by G. Gordon, which is referenced in the "Further Reading" section at the end of this chapter. Any text on the subject should cover this material in a similar amount of detail.

The cumulative probabilities in this final column are important because you can use these values to simulate the stream of queries you observed earlier. Here's how.

Review Listing 7.1 to see how you used the standard `rand()` function to calculate a random number between 1 and 99. You can use a similar technique in a client process to calculate a random number of seconds to delay before sending the next query: a *distributed* (rather than constant) set of IATs. Furthermore, if you processed these generated queries as you did the ones represented in Table 8.1, you would get almost the same distribution; the more numbers you generate and tabulate, the closer its distribution will come to that of the originally observed data.

Listing 8.1 is a program that demonstrates how to generate delay times using the information you gathered from Table 8.1. Note especially how the cumulative probabilities are used to choose a time delay category.

LISTING 8.1 Generate Random Delay Times from IAT Observations

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

#define MAX 200
#define NUM 10

double probs[NUM] = {.26, .47, .63, .76, .84, .90, .94, .97, .99, 1.0};
int    slots[NUM] = { 10,  20,  30,  40,  50,  60,  70,  80,  90, 100};

main(void)
{
   /*
   **   Listing8.1.c - generate random delay times from IAT observations
   */
   double random, delay;
   double dtab[NUM][2]; /* probabilities and cumulative probabilities */
```

LISTING 8.1 Continued

```
   int itab[NUM];       /* counts of observations within range slots */
   int ii, jj;

   srand((unsigned int) getpid()); /* seed rand() */

   /*
   **   Generate delay times
   */
   for(ii=0; ii<MAX; ii++)
   {
      /* random values from .0 thru 1.0 */
      random = rand() / (double) RAND_MAX;

      /* match a time slot */
      for(jj=0; jj<NUM; jj++)
      {
         if(random <= probs[jj])
            break;
      }
      /* random values within each slot */
      random = rand() / (double) RAND_MAX;
      delay = (double) slots[jj] - (random * (double) slots[0]);

      /*******************************************************/
      /*** Send each query after the calculated delay times ***/
      /*******************************************************/

   }
}
```

The first random number is used to search the cumulative probabilities distribution, previously tabulated, to derive a time delay category. A second random number is used to choose (uniformly) a value within that category. The result will be 200 (defined by MAX) delay times with the same distribution as your earlier observations.

Inter-Arrival Time Distributions

Our *derivation* technique is extremely powerful and is a perfect solution when observations don't fall into any obvious pattern. Actual IATs, however, often do fall into a pattern. If you graph the data in the cumulative probabilities column of Table 8.1, you may recognize this as the plot of a decaying exponential. Instead of tabulating observation data, you can generate query IATs with an exponential distribution or perhaps with some other function that fits your data. You will discover how to generate exponential IATs and learn about a few other, simpler distributions.

Uniformly distributed means each possible choice of a value from some range is chosen with equal likelihood; each possible value has the same probability of being chosen. The C language has the `srand()` and the `rand()` functions. You can use

```
random = rand()/(double)RAND_MAX;
```

to get a random number that is chosen uniformly between 0.0 and 1.0.

Note how `srand()` is used in Listing 8.1 to seed the random number generator function, `rand()`. In this example, you used `getpid()` which returns the process ID. You can also use the microseconds portion returned by the `gettimeofday()` function.

NOTE

The seed value doesn't have to be a random number, just different each time you run, assuming you want a different set of results each time.

To get random numbers distributed exponentially or otherwise, you can use uniformly distributed random numbers as the starting point. After a uniform value is chosen, you can use it to calculate a value in any distribution you want, as long as you know the formula. If the data suggest that query IATs are exponentially distributed, you can use the formula

$y = 1 - e^{-Ax}$ [Equation 8.1]

representing a decaying exponential, in which A is the desired mean, or average IAT. Generating exponentially distributed random numbers from a set of uniformly distributed random numbers requires the use of an *inverse* function, in which y is expressed as a function of x. (See the books in the section "Further Reading" for more details.) The inverse of Equation 8.1 is Equation 8.2:

$x = -A * \log_e(y)$ [Equation 8.2]

which can be implemented in C as

```
#include <math.h>

double x, y, a = 1.0;
random = rand()/(double)RAND_MAX;
random = -a * log(random);
```

This gives you rapidly decreasing IAT values ranging from infinity through 0.0, with about half of its values below 1.0. You may want to limit the maximum value with

```
if(random > 10.0)
    random = 10.0;
```

or adjust the mean value (A) to suit your needs. Results from a recent exponential distribution analysis can be seen in Figure 8.1.

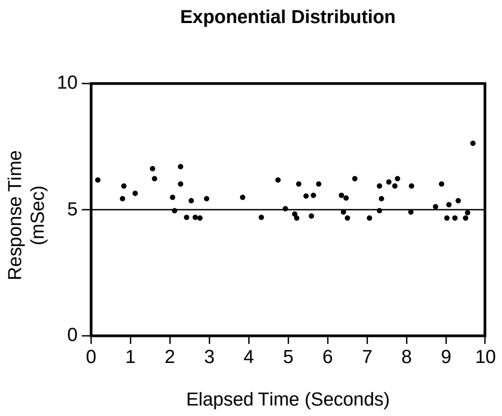

FIGURE 8.1 The exponential distribution.

Other kinds of IAT distributions are much easier to implement. The two discussed next are called *sweep* and *pulse*.

The sweep distribution starts with some high IAT value and gradually reduces it (increases the send rate) down to some low IAT value of your choosing. A program segment that implements the sweep distribution follows:

```
double ht=.25L, lt=.0001, dt, iat;
int ii, steps = 100;

dt = (ht - lt) / (double) steps;
for(ii=0; ii<steps; ii++)
{
       iat = ht - ((double) steps * dt);

       /*** delay for calculated iat ***/
       /*** then send a query packet ***/


}
```

Note that the values used for ht (high time value), lt (low time value) and steps are your choice. You can use this distribution along with the send-and-wait scheme in Listing 7.1, for example. Results from a recent sweep distribution analysis can be seen in Figure 8.2.

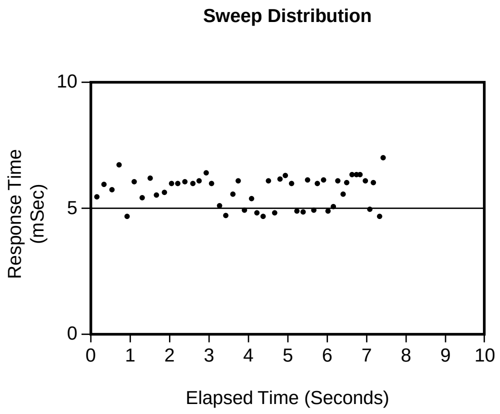

FIGURE 8.2 The sweep distribution.

Another IAT distribution you may want to use is called the *pulse*, or pulse train distribution. A stream of equally spaced, high IAT (slow) queries are followed by a burst of low IAT (fast) queries. After the burst, the stream reverts to the high IAT values until the last query is sent.

```
double ht=.25L, lt=.0001, iat;
int start = 60; stop = 80;
int ii, total = 100;

for(ii=0; ii<total; ii++)
{
       if((ii < start) || (ii >= stop))
        iat = ht;
       else
        iat = lt;

       /*** delay for calculated iat ***/
       /*** then send a query packet ***/

}
```

Again, the choice of variables is yours, and the send-and-wait scheme from Listing 7.1 may be used if desired. You can see results from a recent pulse distribution analysis in Figure 8.3.

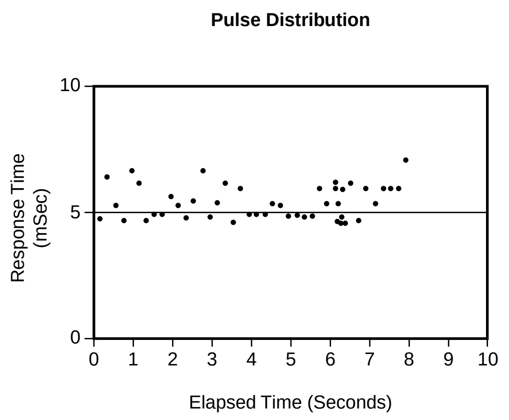

FIGURE 8.3 The pulse distribution.

Checking for an Accurate Response

The protocols used to send and receive query and response packets implement length fields to ensure receipt of the expected amount of data, and they use checksum fields to ensure that the data received is the same as the data sent. There is still a possibility, albeit a small one, that your client may receive a corrupted response from your server under heavy network and system loading conditions.

Listing 7.1 uses a simple check of the line number that is contained, by design, on each line of text. It reads the line number from the text and compares it to the line number it asked for, as follows:

```
/* print out response to our query */
printf("\tLine %d: '%s' ", jj, buf);
/* check for correct line */
p = strrchr(buf, ' ') + 2;
if(jj != atoi(p))
   printf("*");
          printf("\n");
```

Note the asterisk (*) appended to each report line when a mismatch is detected.

You can make this more sophisticated by counting the number of correct and incorrect responses and by keeping track of the percentage of correct packets. The client program you develop later in this chapter implements such a feature and includes these statistics in its display.

Estimating and Displaying Network Utilization

Network utilization may be displayed by the `netstat` line command:

```
> netstat   -i
```

It gives you the number of packets sent and received on each of the network interfaces, along with any packets in error, dropped, overflowed, collided, and so on. Note that this information is associated only with the machine that executes this command. What you really want to know is the overall network loading, and that requires an estimate.

Assuming your cluster is on a closed network, with perhaps a single client machine that also monitors and displays the near real-time performance, you can estimate the network loading by adding all the sent and received packet counts from each node and then dividing by two. If machine A sends a packet to machine B, for example, A will show one packet sent and B will show one packet received. The number of packets on the network, however, is one.

One way to get real-time netstat data is to actually issue the command, via a `system()` function call, from a process that runs on every node. It must read the report from netstat (see also the `popen()` function call), cull out the interesting values, and send them to a central performance monitoring process on the client machine for analysis and display. Each parent server process (master or slave) can start such a subtask as part of its initialization. The client can start the centralized performance subtask. Each performance reporter must know the IP address of the client and the port where performance packets are received. You learn more about the implementation of these features later in this chapter.

The `netstat` command is an inefficient way to get these numbers. Luckily, Linux provides an application-level user interface—a window into the inner workings of the kernel that is very efficient and easy to use. Several kernel modules update what are called /proc pseudofiles. Your process can open and read them as if they were text. You will learn about the contents of some of them as they were defined in an early version of the Linux OS. Note that their contents change without warning from release to release. The best way to understand them is to list them from the command line and then investigate their exact meaning by looking at the kernel source code that updates each file of interest. (See the man proc page and the Linux programming book cited in the “Further Reading” section at the end of this chapter for more details.)

Listing 8.2 is a program that reads the /proc files, gathering real-time statistics. This program will run on each node of your cluster. It will be started via fork-execl (explained in Chapter 3, “Interprocess Communication”) with an operand that tells it the hostname where the network utilization display process runs. These processes read and process text once each second, so they represent a very low impact on the node's resources. They may be running even when the display process is not. The UDP packets sent to the display process will be ignored. You will start and stop the display process at your convenience while the automatically started statistical information gatherers continue to run.

LISTING 8.2 Gathering and Sending Performance Statistics to the Centralized Display Program in Listing 8.3

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include "Listing8.2.h"

main(int argc, char *argv[])

{
/*
**   Listing8.2.c - gathers and sends performance stats to Listing8.3.c
*/
   struct sockaddr_in soc;
   struct cprStats stats;
   struct hostent *hp;
   long usec0, usec1; /* user */
   long nsec0, nsec1; /* nice */
   long ssec0, ssec1; /* system */
   long isec0, isec1; /* idle */
   long rpkts0, rpkts1; /* rcvd */
   long spkts0, spkts1; /* sent */
   long tmem, umem;
   long rio0, rio1;
   long wio0, wio1;
   FILE *fptr;
   float fcpu;
   float fmem;
   int ii, jj;
   int icpu;
   int imem;
   int sum;
   int fd;
   char local[16];
```

LISTING 8.2 Continued

```
char remote[16];
char buf[256], *p;

if(argc != 2)
{
   printf("\n\tUsage: %s <hostname>\n\n", argv[0]);
   exit(-1);
}
/* get remote hostname */
strcpy(remote, argv[1]);
/* and the local hostname, also */
gethostname(local, sizeof(local));
if((p = strchr(local, '.')) != NULL)
   *p = '\0'; /* trunc domain.nam */

/*
**   Central Performance Reporting socket
*/
fd = socket(AF_INET, SOCK_DGRAM, 0);
memset((char *) &soc, 0, sizeof(soc));
hp = gethostbyname(remote);
memcpy(&soc.sin_addr, hp->h_addr, hp->h_length);
soc.sin_port = htons(PORT);
soc.sin_family = AF_INET;

/*
**   Initial CPU utilization
*/
fptr = fopen("/proc/stat", "r");
fgets(buf, sizeof(buf), fptr);
buf[strlen(buf)-1] = 0;
sscanf(buf, "%*s %ld %ld %ld %ld",
   &usec0, &nsec0, &ssec0, &isec0);
/*
**   Initial disk IO rates (also in /proc/stat)
*/
while((fgets(buf, sizeof(buf), fptr)) != NULL)
{
   if(strncmp(buf, "disk_", 5) == 0)
      break;
}
```

LISTING 8.2 Continued

```
   sscanf(buf, "%*s %ld", &rio0);
   fgets(buf, sizeof(buf), fptr);
   sscanf(buf, "%*s %ld", &wio0);
   fclose(fptr);

   /*
   **   Initial network I/O rates
   */
   fptr = fopen("/proc/net/dev", "r");
   /* skip column header strings in net/dev */
   fgets(buf, sizeof(buf), fptr); /* skip */
   fgets(buf, sizeof(buf), fptr); /* skip */
   fgets(buf, sizeof(buf), fptr); /* skip */
   fgets(buf, sizeof(buf), fptr); /* eth0 */
   buf[strlen(buf)-1] = 0; /* over '\n' */
   buf[6] = ' ';           /* over  ':' */
   sscanf(buf, "%*s %ld %*d %*d %*d %*d %ld", &rpkts0, &spkts0);
   fclose(fptr);

   /*
   **   Statistics-Gathering Loop
   */
   while(1)
   {
      /*
      **   Real MEM utilization
      */
      fptr = fopen("/proc/meminfo", "r");
      fgets(buf, sizeof(buf), fptr); /* skip hdr */
      fgets(buf, sizeof(buf), fptr);
      buf[strlen(buf)-1] = 0;
      sscanf(buf, "%*s %ld %ld", &tmem, &umem);
      fmem = (float) umem / (float) tmem;
      stats.rm = (int) (100 * (fmem + .005));
      fclose(fptr);

      /*
      **   Subsequent CPU utilization
      */
      fptr = fopen("/proc/stat", "r");
```

LISTING 8.2 Continued

```
      fgets(buf, sizeof(buf), fptr);
      buf[strlen(buf)-1] = 0;
      sscanf(buf, "%*s %ld %ld %ld %ld",
         &usec1, &nsec1, &ssec1, &isec1);
      /* calculate %CPU utilization */
      usec1 -= usec0; /* 1-second user CPU counter */
      nsec1 -= nsec0; /*          nice             */
      ssec1 -= ssec0; /*          system           */
      isec1 -= isec0; /*          idle             */
      fcpu = (float) (usec1 + nsec1 + ssec1)
           / (float) (usec1 + nsec1 + ssec1 + isec1);
      stats.cp = (int) (100 * (fcpu + .005));

      /*
      **   Subsequent disk I/O rates (also in /proc/stat)
      */
      while((fgets(buf, sizeof(buf), fptr)) != NULL)
      {
         if(strncmp(buf, "disk_rio", 8) == 0) /* skip */
            break;
      }
      sscanf(buf, "%*s %ld", &rio1);
      /* next string after disk_rio is disk_wio */
      fgets(buf, sizeof(buf), fptr);
      sscanf(buf, "%*s %ld", &wio1);
      fclose(fptr);
      /* calculate disk I/O rates */
      stats.dr = (rio1 - rio0) / 5;
      stats.dw = (wio1 - wio0) / 5;
      fclose(fptr);

      /*
      **   Subsequent network I/O rates
      */
      fptr = fopen("/proc/net/dev", "r");
      /* skip column header strings in net/dev */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* eth0 */
```

LISTING 8.2 Continued

```
      buf[strlen(buf)-1] = 0; /* over '\n' */
      buf[6] = ' ';           /* over  ':' */
      sscanf(buf, "%*s %ld %*d %*d %*d %*d %ld", &rpkts1, &spkts1);
      /* calc network I/O rates */
      stats.pr = rpkts1 - rpkts0;
      stats.ps = spkts1 - spkts0;
      fclose(fptr);

      /*
      **   Reset values for next iteration
      */
      rpkts0 = rpkts1;
      spkts0 = spkts1;
      usec0 += usec1;
      nsec0 += nsec1;
      ssec0 += ssec1;
      isec0 += isec1;
      rio0 = rio1;
      wio0 = wio1;

      /*
      **   Send statistics at 1-second intervals
      */
      sendto(fd, &stats, sizeof(stats), 0,
         (struct sockaddr *) &soc, sizeof(soc));
      sleep(1);
   }
}
```

Three of the /proc pseudofiles contain enough data to represent the general condition of each processor. This information is sent to a process that runs on the client or some other network machine that is not one of the cluster nodes. It may be started from the command line by

```
> Listing8.3
```

and stopped via the Ctrl+C keystroke. A sample of the character-based network utilization report is in Figure 8.4. CPU and memory utilization, disk and network I/O rates, and overall network loading give a good indication of performance for now. The program that produces it is shown in Listing 8.3.

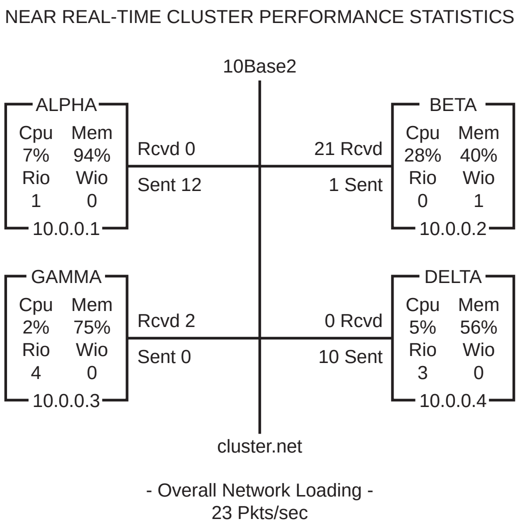

FIGURE 8.4 Near real-time network utilization display.

LISTING 8.3 Centralized Performance Reporting

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdio.h>
#include <netdb.h>
#include "Listing8.2.h"

main(void)
{
   /*
   **   Listing8.3.c - Centralized Performance Reporting
   */
   int cp[4], rm[4]; /* central processor, real memory */
   int ps[4], pr[4]; /* packets sent, packets received */
   int dr[4], dw[4]; /* disk reads, disk writes */
   union {
      unsigned long addr;
      char bytes[4];
   } u;
   struct sockaddr_in soc;
   struct cprStats stats;
```

LISTING 8.3 Continued

```
int len=sizeof(soc);
struct hostent *hp;
int fd, ii, jj;
int netLoad;
char buf[256];

/* establish and initialize UDP socket struct */
fd = socket(AF_INET, SOCK_DGRAM, 0);
bzero((char *) &soc, sizeof(soc));
soc.sin_addr.s_addr = htonl(INADDR_ANY);
soc.sin_port = htons(PORT);
soc.sin_family = AF_INET;
bind(fd, (struct sockaddr *) &soc, len);

/* initialize stats */
for(ii=0; ii<4; ii++)
{
   cp[ii] = 0;
   rm[ii] = 0;
   ps[ii] = 0;
   pr[ii] = 0;
   dr[ii] = 0;
   dw[ii] = 0;
}

/*
**   Centralized Performance Reporting Loop
*/
while(1)
{
   for(ii=0; ii<4; ii++)
   {
      recvfrom(fd, &stats, sizeof(stats), 0,
        (struct sockaddr *) &soc, &len);
      /* index is last IP addr digit-1 */
      memcpy(&u.addr, &soc.sin_addr, 4);
      jj = (unsigned int) u.bytes[3];
      cp[jj-1] = stats.cp;
      rm[jj-1] = stats.rm;
      ps[jj-1] = stats.ps;
      pr[jj-1] = stats.pr;
      dr[jj-1] = stats.dr;
```

LISTING 8.3 Continued

```
         dw[jj-1] = stats.dw;
      }

      /*
      **   Calculate network load
      */
      netLoad = 0;
      for(ii=0; ii<4; ii++)
         netLoad += (ps[ii] + pr[ii]);
      netLoad /= 2; /* average of each node's number sent and received */

      /*
      **   Draw a network diagram showing cluster performance statistics
      */
      system("clear");
      printf("\n      NEAR REAL-TIME CLUSTER PERFORMANCE STATISTICS     \n");
      printf("                                                        \n");
      printf("                         10Base2                        \n");
      printf(" +----ALPHA-----+           |           +-----BETA-----+\n");
      printf(" |   Cpu    Mem |           |           |   Cpu    Mem |\n");
      printf(" |  %3d%%   %3d%% |Rcvd %-5d | %5d Rcvd|  %3d%%   %3d%% |\n",
         cp[0], rm[0], pr[0], pr[1], cp[1], rm[1]);
      printf(" |   Rio    Wio +-----------+----------+   Rio    Wio |\n");
      printf(" |%6d %6d |Sent %-5d | %5d Sent|%6d %6d |\n",
         dr[0], dw[0], ps[0], ps[1], dr[1], dw[1]);
      printf(" +---10.0.0.1---+           |           +---10.0.0.2---+\n");
      printf("                            |                           \n");
      printf(" +----GAMMA-----+           |           +----DELTA-----+\n");
      printf(" |   Cpu    Mem |           |           |   Cpu    Mem |\n");
      printf(" |  %3d%%   %3d%% |Rcvd %-5d | %5d Rcvd|  %3d%%   %3d%% |\n",
         cp[2], rm[2], pr[2], pr[3], cp[3], rm[3]);
      printf(" |   Rio    Wio +-----------+----------+   Rio    Wio |\n");
      printf(" |%6d %6d |Sent %-5d | %5d Sent|%6d %6d |\n",
         dr[2], dw[2], ps[2], ps[3], dr[3], dw[3]);
      printf(" +---10.0.0.3---+           |           +---10.0.0.4---+\n");
      printf("                        cluster.net\n\n");
      printf("               - Overall Network Loading -\n");
      printf("                %9d Pkts/sec\n", netLoad);
      printf("\n");
   }
}
```

Note the requirement for common knowledge of the performance information packet structure as well as the destination port address. The header file referenced by Listings 8.2 and 8.4 follows:

```
#define PORT 9090
struct cprStats
{
   int cp; /* cpu utilization */
   int rm; /* mem utilization */
   int ps; /* packets sent */
   int pr; /* packets rcvd */
   int dr; /* disk reads */
   int dw; /* disk writes */
} cprStats;
```

The method of reading four incoming packets prior to each redrawing of the display area may seem inelegant. It lacks robustness by assuming everything is working correctly. You can ignore that for now; you will learn more about software robustness issues in Chapter 10, “Robust Software.”

Displaying Response Time Statistics

The subtasks in Listing 7.1 can record the time a query was sent, when the corresponding response was received, and can easily calculate each of their response times. It can also keep this information in a shared memory space set up by the client’s parent task for that purpose. You want the subtasks to record the individual response times in a shared memory space and the parent to report the external performance as it changes.

The parent task in Listing 7.1 waits for any remainder of its IAT, and it issues error messages in cases where the amount of time necessary to prepare and send a query exceeds that value. Assuming that it can work within the IAT limitation and must wait before continuing, it can certainly take a tiny moment to update an array of response time categories. It can display these response times in a bar chart when the packet stream is finished, as shown in Figure 8.5. A character-based presentation is used for simplicity.

Listing 8.4 shows a client process that supports exponential, pulse, and sweep distributions and that presents a bar chart of overall response time statistics.

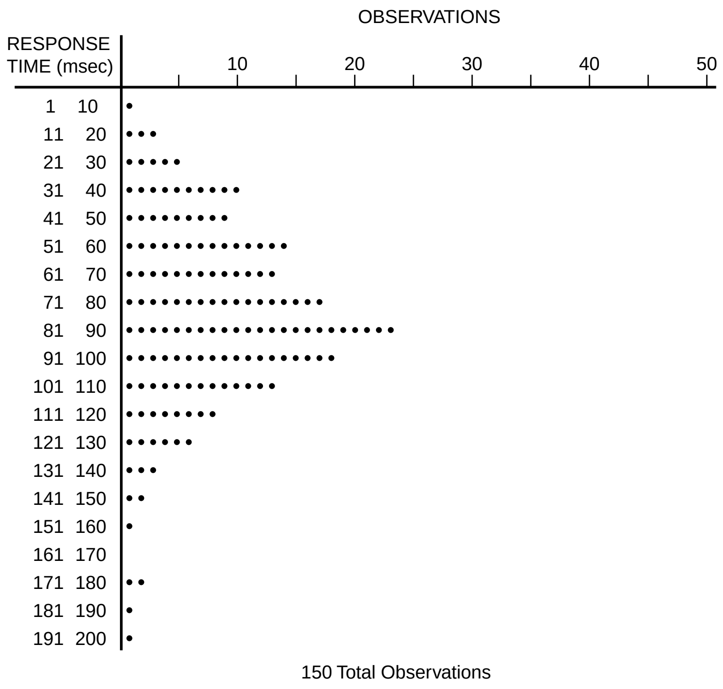

FIGURE 8.5 Response time displayed as a histogram.

LISTING 8.4 Client Reports Response Time Statistics

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "msi.h"

#define FALSE 0
#define TRUE 1
```

LISTING 8.4 Continued

```
#define NUM 50

#define CATS 20 /* response time category values */
double cats[CATS] = {.010, .020, .030, .040, .050,
                     .060, .070, .080, .090, .100,
                     .110, .120, .130, .140, .150,
                     .160, .170, .180, .190, .200};
int bars[CATS] = {0}; /* counters for bar chart */

main(int argc, char *argv[])

{
   /*
   **   Listing8.4.c - Client: reports response time stats
   */
   struct timeval bef, aft;   /* timer value before/after */
   struct sockaddr_in sock;   /* structure for socket */
   struct hostent *hp;        /* structure for IP address */
   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = NUM * sizeof(double);
   key_t key = 0x01234567;    /* example shared memory key */
   int shmid;                 /* shared memory area id */
   int Expon = FALSE;         /* boolean: exponential dist */
   int Pulse = FALSE;         /* boolean: pulse distribution */
   int Sweep = FALSE;         /* boolean: sweep distribution */
   double mean=0.047;         /* mean value for exponential */
   double slow=0.250;         /* slow value for pulse & sweep */
   double fast=0.050;         /* fast value for pulse burst */
   double random;             /* random reals: 0.0->1.0 */
   double delay;              /* calculated delay time */
   double *rt;                /* response times from subtasks */
   long timeleft;             /* time remaining in IAT */
   long usec=0L;              /* inter-arrival microseconds */
   long secs=0L;              /* time remaining in seconds */
   int pids[NUM];             /* subtask process ids */
   int opt=1;                 /* setsockopt parameter */
   int fd;                    /* socket file descriptor */
   int ii, jj, kk=0;
   char buf[BUFSIZE], *p;

   srand((unsigned int) getpid()); /* seed rand() */

   /*
```

LISTING 8.4 Continued

```
**   Operands are remote HOST name and distribution code
*/
if(argc != 3)
{
   printf("\n\tUsage: %s <HOST> <DISTRIBUTION>\n\n", argv[0]);
   exit(-1);
}
hp = gethostbyname(argv[1]);

if((argv[2][0] == 'e') || (argv[2][0] == 'E'))
{
   Expon = TRUE;
   srand((unsigned int) getpid()); /* rand() seed */
}
else if((argv[2][0] == 'p') || (argv[2][0] == 'P'))
{
   Pulse = TRUE;
}
else if((argv[2][0] == 's') || (argv[2][0] == 'S'))
{
   Sweep = TRUE;
}
else
{
   printf("\n\tUsage: %s <HOST> <DISTRIBUTION>\n", argv[0]);
   printf("\t       DISTRIBUTION = { 'e' | 'p' | 's' }\n");
   printf("\t       for exponential, pulse and sweep\n\n");
   exit(-1);
}

/* attach shared memory array */
shmid = shmget(key, size, flag);
/* attach the shared memory area */
rt = (double *) shmat(shmid, 0, 0);

/*
**   LOOP: send and receive NUM packets
*/
for(ii=0; ii<NUM; ii++)
{
   rt[ii]= 0.0L; /* zero shared array values */

   if(Expon)
```

LISTING 8.4 Continued

```
      {
         random = rand() / (double) RAND_MAX;
         delay = -mean * log(random);
         if(delay > 0.999999L)
            delay = 0.999999L; /* limit maximum */
         usec = delay * 1000000L;
      }
      else if(Pulse)
      {
         if((ii > (NUM * 0.3)) && (ii < (NUM * 0.4)))
            delay = fast;
         else
            delay = slow;
      }
      else if(Sweep)
      {
         delay = slow - (ii * (slow / NUM));
      }
      else
      {
         printf("\n\tError in logic?\n\n");
         exit(-1);
      }
      secs = (long) delay;
      usec = (long) (1000000.0L * (delay - ((double) secs)));

      /* random line numbers: 1 thru 99 */
      random = rand() / (double) RAND_MAX;
      jj = (int) ((double) (99.0) * random) + 1;
      if(jj == 100)
         jj = 99;
      sprintf(buf, "/home/chief/sample.txt %d", jj);

      /* record sending time */
      gettimeofday(&bef, NULL);

      /*
      **   A subtask for each query/response
      */
      if((pids[kk++]=fork()) == 0)
      {
         /* attach parent's shared memory */
```

LISTING 8.4 Continued

```
        rt = (double *) shmat(shmid, 0, 0);
        /* shmid is still set correctly */

        /* Set up TCP socket to the server */
        fd = socket(AF_INET, SOCK_STREAM, 0);
        bzero((char *) &sock, sizeof(sock));
        bcopy(hp->h_addr, &sock.sin_addr, hp->h_length);
        sock.sin_family = hp->h_addrtype;
        sock.sin_port = htons(QUERYLINK);
        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
           (char *) &opt, sizeof(opt));
        connect(fd, (struct sockaddr *) &sock, sizeof(sock));
        send(fd, buf, strlen(buf)+1, 0);
        buf[0] = 0; /* clear buffer */
        if(recv(fd, buf, BUFSIZE, 0) > 0)
        {
           printf("\t%d. Line %2d: %s ", kk, jj, buf);
           p = strrchr(buf, ' ') + 2;
           if(jj != atoi(p))
              printf("***"); /* incorrect response */
           printf("\n");
        }

        /* record response time */
        gettimeofday(&aft, NULL);
        aft.tv_sec -= bef.tv_sec;
        aft.tv_usec -= bef.tv_usec;
        if(aft.tv_usec < 0L)
        {
           aft.tv_usec += 1000000L;
           aft.tv_sec -= 1;
        }
        rt[ii] = (double) aft.tv_sec + ((double) aft.tv_usec / 1000000.0L);

        close(fd);
        exit(0);
     }

     /*
     **   Sleep for remainder of IAT
     */
     gettimeofday(&aft, NULL);
     aft.tv_sec -= bef.tv_sec;
```

LISTING 8.4 Continued

```
      aft.tv_usec -= bef.tv_usec;
      if(aft.tv_usec < 0L)
      {
         aft.tv_usec += 1000000L;
         aft.tv_sec -= 1;
      }
      bef.tv_sec = secs;
      bef.tv_usec = usec;
      bef.tv_sec -= aft.tv_sec;
      bef.tv_usec -= aft.tv_usec;
      if(bef.tv_usec < 0L)
      {
         bef.tv_usec += 1000000L;
         bef.tv_sec -= 1;
      }
      timeleft = (bef.tv_sec * 1000000L ) + bef.tv_usec;
      if(timeleft < 0)
      {
         printf("\tERROR: A higher IAT value is required - exiting.\n");
         break;
      }
      usleep(timeleft);
   }
   for(ii=0; ii<kk; ii++)
      wait(pids[ii]);

   /*
   **   Report the response time statistics
   */
   for(ii=0; ii<NUM; ii++)
   {
      for(jj=0; jj<CATS; jj++)
      {
         if(rt[ii] < cats[jj])
         {
            bars[jj]++;
            break;
         }
      }
   }
   printf("RESPONSE    |                  OBSERVATIONS\n");
   printf("TIME (msec) |        10        20        30        40        50\n");
```

LISTING 8.4 Continued

```
   printf("-----------+----+----+----+----+----+----+----+----+----+----+\n");
   ii = 0;
   for(jj=0; jj<CATS; jj++)
   {
      ii += bars[jj];
      kk = 1000 * cats[jj];
      printf(" %5d%5d |", kk-9, kk);
      for(kk=0; kk<bars[jj]; kk++)
         printf("*");
      printf("\n\n");
   }
   printf("                          %d Total Observations\n", ii);

   /* remove unused shared memory area and exit */
   shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
   return(0);
}
```

Here too, a need exists for common knowledge between the client in Listing 8.4 and the master and slave processes in Appendix A "The Source Code." The short header file, named msi.h, follows.

```
#define QUERYLINK 9000
#define BROADCAST 5000
#define REGISTERS 5001
#define SLAVELINK 5002
#define BUFSIZE 256
```

The maximum size of the query packets (`BUFSIZE`) and the `MASTER` process query port (`QUERYLINK`) are of interest to the client, as well as the master and its slave processes.

The External Performance of Your MSI Server

Now that the client (see Listing 8.4) is sophisticated enough to present a stream of queries to your MSI cluster server, it's time to run some preliminary tests to get some baseline statistics. When you examine the internal performance and make improvements, you can compare these graphs to the new ones you get, to see if what you did had any beneficial effect.

Figures 8.6, 8.7, and 8.8 show the client's bar chart displays of overall response times for an exponential, pulse, and sweep distribution, respectively. Each of these runs sent a 50-packet stream, as indicated at the bottom of each chart.

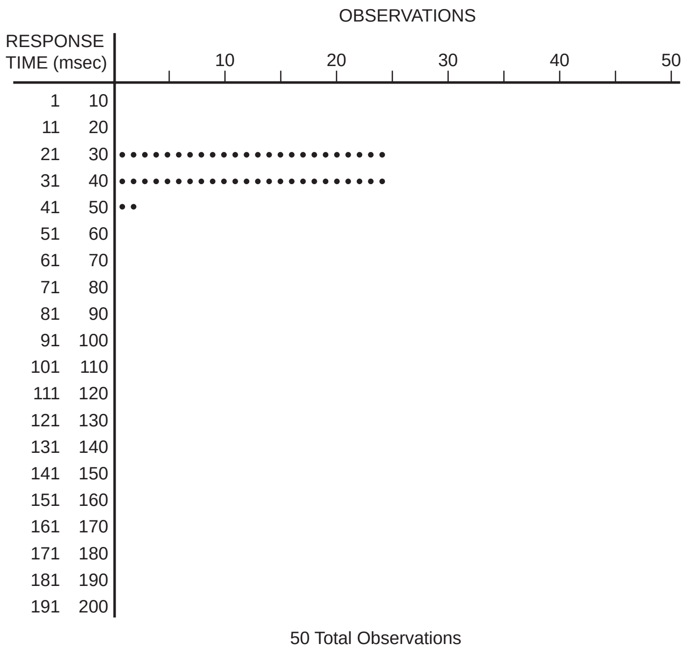

FIGURE 8.6 Exponential distribution—live test results.

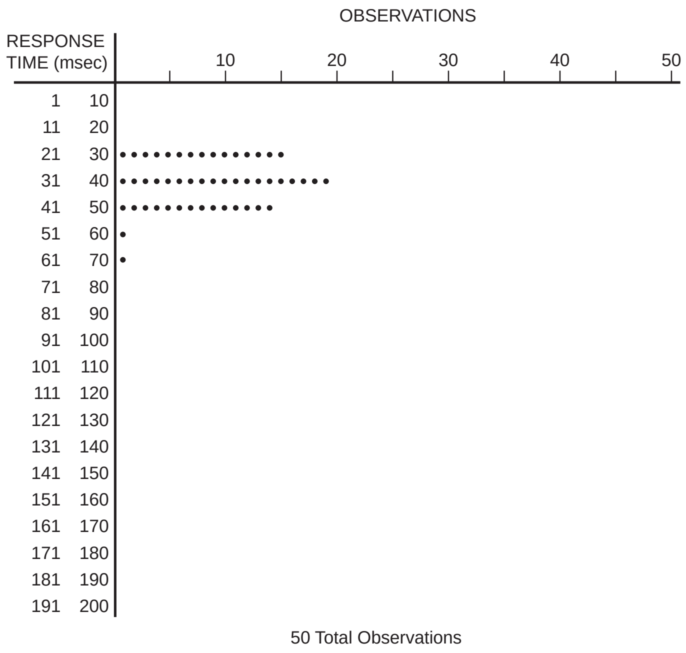

FIGURE 8.7 Pulse distribution—live test results.

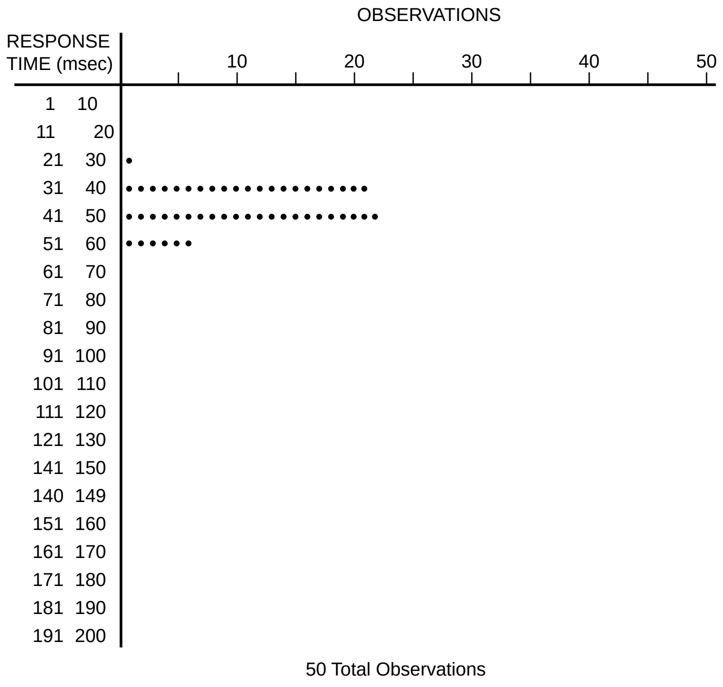

FIGURE 8.8 Sweep distribution—live test results.

Figure 8.9 shows a snapshot of the near real-time display as the overall network load peaked during a test of the sweep distribution.

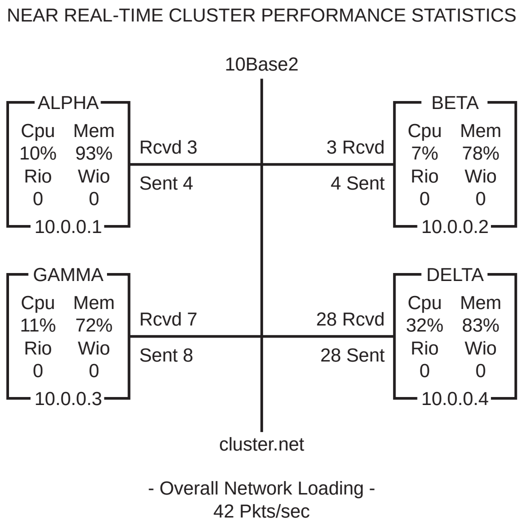

FIGURE 8.9 Sweep distribution—live performance display.

Summary

The chapter began with a look at what it means to generate and send queries to a server and how you might calculate an IAT distribution from observed data. You looked at how to generate queries in an exponential distribution, in which the data can be described by a formula. Finally, you saw two artificial distributions: sweep and pulse.

You learned how to calculate a node's system resource loading (CPU, real memory, network and disk I/O) and how to gather those statistics from all four of your nodes and display them, in near real-time, for quick visual reference.

You updated the client program from the previous chapter in support of three new distributions, and you learned how to collect and display the overall response times in a bar chart.

Lastly, the bar charts from live test runs are presented, along with a snapshot of the overall system loading display during the sweep distribution test.

Your master-slave interface server runs on a cluster that you built from four PCs and some networking hardware. You have a client process that can stress it, a performance monitor that can measure the level of stress at each node, and a display program that can show you what they are all experiencing at once. You have the first set of live data representing the overall external system performance of your cluster, and that's exactly what you set out to do at the beginning of this chapter.

In Chapter 9, "Internal Performance Measurement and Tuning," you will learn about timing and how to get an accurate image of internal performance. You will break the query processing into its individual phases, time them, and analyze them. The goal is to find parts of the process that can be sped up, thereby reducing the overall response time and improving your system's overall external performance—exactly what the clients want.

Further Reading

- Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. New York, NY: John Wiley and Sons, 1993.
- Gordon, Goeffrey. *System Simulation*. Englewood Cliffs, NJ: Prentice-Hall, 1969.
- Kleinrock, Leonard. *Queueing Systems, Volume I: Theory, and Volume II: Computer Applications*. New York, NY: John Wiley and Sons, 1975.
- Rusling, David, et al. *Linux Programming White Papers*. Scottsdale, AZ: Coriolis Press, 1999.

9 Internal Performance Measurement and Timing

IN THIS CHAPTER

Why measure the internal performance of a cluster server? You learned how to measure external performance in Chapter 8, “External Performance Measurement and Analysis.” If you are ever unhappy with your external performance results and want to see some improvement, you will need to know where the server spends its time as it processes each incoming query. In this chapter, you will divide query processing into phases, measure the time it takes each phase to complete, and present the average phase completion times in a bar graph for analysis. An example of such a graph, called a *phase execution profile*, is shown in Figure 9.1. Each bar represents a single phase of execution in order of execution, from left to right. The height of each bar represents the average execution time for that phase.

Some phases will be more complex than others and will take correspondingly longer to complete. Having designed the processing architecture, you should have a good feel for how long one phase takes compared to others. Sending data over the network will take much longer than transferring it across the internal data bus. Generally speaking, TCP/IP links have a greater overhead and take longer to transfer data than UDP/IP links do. Some general rules exist, but don’t be surprised if you guess wrong about which phases are longer.

If two or more phases are thought to take equal amounts of time but do not, look for time-saving improvements in the ones with the unexpectedly long execution times. Finally, rerun the internal performance analysis to see whether the changes you made caused any performance improvements.

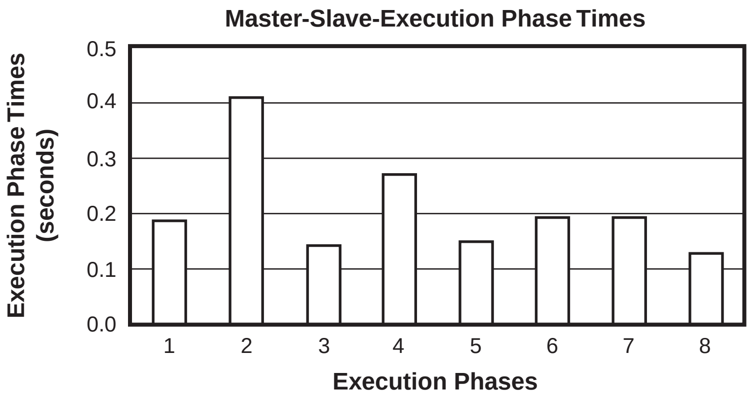

FIGURE 9.1 A sample execution phase profile.

In this chapter, you will see the master-slave interface (MSI) processing broken down into execution phases. Source code used to time these phases is presented and discussed in some detail. The results of actual timing runs are graphed and analyzed. Unexpectedly long phases are investigated, and you will see what was done to reduce their execution times. A final set of timing data will help you quantify any performance improvement over the baseline data presented in Chapter 8.

Profiling Software Execution

Thirty years ago, excessive application execution times (measured in days in extreme cases!) were of great concern to scientific data center managers. They measured internal performance by recompiling these programs along with a then state-of-the-art mainframe profiler. The profiler started the program under test as a subtask and paused it at regular intervals to record the address of its next executable instruction. After a short representative run, the profiler printed a histogram of real memory address ranges, along with the number of times the program under test was found to be executing in each of the address ranges. An analyst converted each memory address range to a range of Fortran language statements by hand, with the aid of a linkage editor map. Finally, the analyst and the programmer recoded these sections in a more efficient manner. This effort often took days, sometimes weeks, but initial efforts provided generous rewards, and computer time was expensive.

Ten years ago, you could buy a C language program execution profiler for your PC for less than $30. It reported execution times at the main and function level and, optionally, at the statement level within any one of your functions. You could optimize your C code for faster execution time in a matter of hours.

NOTE

Code profiling is easier because the tools are better today than they ever were. Code optimization still requires a lot of talent, however, because the developer must understand that an index reference into an array, for example, is slow compared to an incremental pointer reference. Developers must also understand that an optimizing compiler might generate the same machine-level code for either of those approaches and that the array index might be a better choice for documentation purposes. (Don't forget the person who might have to debug your code later!)

Knowledge of the target platform's assembler, or at least the general hardware requirements for a given set of instructions, can save a lot of time during an optimization effort.

If you sample a program's point of execution at regular intervals, the percentage of times you observed it executing a particular phase is a good representation of the percentage of time it spends in that phase. This is a statistical result, of course, so the larger your sample size, the better the representation. Because it doesn't require knowledge of the process, sampling is perhaps the best way to obtain an execution profile.

Direct measurement—the technique used here—gives more accurate results, but you have to know where to do the measuring.

The MSI is a distributed system—several tasks running on a network of nodes, working together to respond to incoming queries. Profiling each program separately may prove helpful, but using the phase timing and reporting techniques outlined in this chapter should give you insight into broader performance issues. The overall processing time of a distributed system can be greatly influenced by process interaction—interprocess communication, for example, which cannot be measured with a simple profiler. Take another look at Figure 7.2, the MSI architecture diagram, to refresh your memory of how these processes interact.

Distributed System Execution Profiling

The MSI server responds to incoming queries, and each query is processed in the same way. That means the processing phases can be ordered linearly in time. Keep in mind that phases of a more complex system—one that processes queries differently based on their content, for example—may have a treelike ordering in time. Treat each processing branch as a separate service and analyze them separately in such cases.

The first question to ask is, "What are the execution phases required to process a single incoming query?" Remember that you must not limit your thinking to the response-building phases, but to all the cluster management in support of it, as well. Let's look at what steps the server performs to process a query.

1. The master chooses a local subtask (from its pool) to receive an incoming query:

   In Chapter 7, "The Master-Slave Interface Software Architecture," you saw how a concurrent server outperformed a serial server and how a resident pool of subtasks reduced the overhead of spawning a new subtask to handle every incoming query. In the subtask pool architecture, the parent task must choose an available subtask from the pool. This is the first phase of execution. This phase, and most others, can be timed directly by subtracting the phase-start time from the phase-end time.

2. The chosen subtask receives its query and passes it to a remote slave process:

   Each subtask is paired with a slave process running on a remote node of the cluster (refer to Figure 7.2). Each query is sent to a remote slave for processing. This phase includes the preparation of the packet but not the actual transmission time, which is included in the next phase.

3. Calculated master-to-slave network transit time:

   Each packet is sent from one node to another. In an inactive network, these transmission times are predictable, but when several nodes on the network compete for its use, the network access time can become excessive.

4. The slave process receives the incoming query and extracts a request from it:

   A filename and a line number are inside each query in the example. These details are extracted from the incoming query before a response can be built.

5. The slave process retrieves the requested line of text from the shared text file:

   The slave opens the specified file, reads the requested line of text, and closes the file.

6. The slave process builds and returns the response to the master subtask:

   The requested line of text is encapsulated into a response packet.

7. Calculated slave-to-master network transit time (same transit time as step 3):

   This is the exact opposite route as the preceding one taken in phase 3, so you can assume that it takes approximately the same amount of time under similar network utilization. The network load is probably the same during both trips because the time between them is very short, as you will learn from the analysis later in this chapter. The analysis is based on a chart of the averages times taken to process many queries, so unexpected differences in trip times will become insignificant.

8. The master's chosen subtask responds to the query and declares itself idle:

   The subtask is waiting for the slave to process the request. When the response arrives, the subtask extracts the timing data from the slave's execution phases and sends it back to the client. The subtask resets its condition flag to show the master that it is idle.

Let's assume Figure 9.1 represents these eight execution phases, although it does not. During the analysis, you might wonder why phases 3 and 7, the network transit times, are not equal. You might also wonder why phase 2 takes so long when there is so little effort involved. These are exactly the kind of questions that lead to better internal performance. The answers to these questions, and the analysis that follows, lead to better internal performance. It leads to better external performance, too, in direct support of your ultimate goal of client satisfaction.

Event Timing Techniques

Most event phases will begin and end in the same process and have no intervening wait times. Some phases, however, will include I/O, and some will span processes on several of the network nodes. Some of these cases will prove very interesting, leading to valuable insights into the inner workings of a distributed system.

The simple events are easy to time using the `gettimeofday` function call. It returns the time of day, in microseconds, since a point of time in the distant past. (As of this writing, that distant point in time was midnight, January 1, 1970, Universal Coordinated Time.) The exact number of seconds and microseconds is returned as two long integers (type `time_t`). You may use two time-of-day values to calculate the elapsed time, as follows:

LISTING 9.1 A Sample Program That Times an Execution Phase

```
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

main(void)
{
   /*
   **   Listing9.1.c - sample execution phase timing
   */
   struct timeval time1, time2;

   gettimeofday(&time1, NULL); /* START time */
```

LISTING 9.1 Continued

```
      /* (execution phase of interest) */

   gettimeofday(&time2, NULL); /* END time */
   time2.tv_sec -= time1.tv_sec;
   time2.tv_usec -= time1.tv_usec;
   if(time2.tv_usec < 0L)
   {
      time2.tv_usec += 1000000L;
      time2.tv_sec -= 1;
   }
   printf("Execution time = %ld.%06ld seconds\n",
      time2.tv_sec, time2.tv_usec);
   return(0);
}
```

Note that two time-of-day values are requested, and the difference between them is the time it takes to execute the event phase, plus overhead in the `gettimeofday` function call. This overhead is very small and usually can be ignored. If your application cannot ignore even such small timing overheads, however, you can time this whole timed event and subtract out the overhead, assuming it will be the same for both sets of time-of-day calls.

Most of the events in the MSI are simple and can be timed using the technique in Listing 9.1. Phases 4 through 6 are timed in the slave process, which means that these measurements must be brought back to the master's subtask for consolidation with the local data. These results can be sent along with the packet each time the slave sends back a response. The network transit times, however, require special consideration.

At first glance, it might seem reasonable to get the packet-send time on the sending machine and the packet-receive time on the receiving machine. Passing the latter time back and subtracting the two calculates the transit time—au contraire!

The call to `gettimeofday` returns a value that represents the exact number of microseconds since January 1, 1970, according to documentation. Practically speaking, however, timers on a PC are driven by a crystal oscillator that is subject both to manufacturer intolerances and temperature variations. Each crystal is slightly off its design frequencies, so any two may drift apart over time. Although months may go by before you must reset the time of day on your PC, the differences between any two nodes on a network prevents your relying on them as being perfectly synchronized. This difference is especially apparent when calculating the short time it takes to send a packet between nodes. Not to worry—a clever way exists to handle this situation. See the timing diagram in Figure 9.2.

Request the query-send time on the sending node, as shown previously, and then request the time that the response is received on that same sending node. You can subtract two time-of-day values from the same clock reliably, in this case getting the round-trip transit time plus the processing time on the remote slave node. (This is shown as A in Figure 9.2.) Phases 4 through 6 are measured on the remote node and sent back to the master, and adding just one more event time to that list doesn't cost very much.

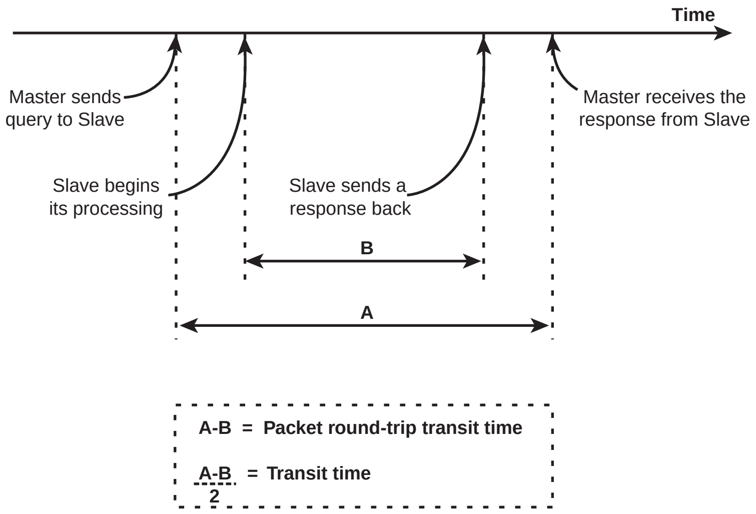
Time
Master sends query to Slave
Slave begins its processing
Slave sends a response back
Master receives the response from Slave
B
A
A-B = Packet round-trip transit time
$\frac{A-B}{2}$ = Transit time

FIGURE 9.2 Calculating the network transit time.

One solution is to request the query-receive time and the response-send time at the slave node. Both times are from the slave's clock, so their difference reliably represents the entire slave node processing time. (This is shown as B in Figure 9.2.) Subtract B from A to get the round-trip transit time; divide that result by two to get the one-way transit time. Assume that these two times are equal and record the same value in phases 3 and 7. Any network loading-induced discrepancies will become insignificant because the analysis deals with the average of a large number of such estimates.

Execution Phase Timing Plots

The master has a subtask for each registered slave node, as described in earlier chapters. Each subtask establishes a communication link with a dedicated process on a remote slave node. The master subtask sends a query to the slave process, and the slave sends back a response. Included in the response packet are the results of event timing on the slave node.

When the subtask receives this packet, it extracts the timing data, adding each event phase time into an accumulator, and increments a counter, all of which are in memory that is shared with the parent task.

The response packet has the following structure:

```
struct packet
{
   /*
   **   Information request packet
   */
#ifdef TIMING
   /* slave phase times */
   struct timeval remote;
   struct timeval phase4;
   struct timeval phase5;
   struct timeval phase6;
#endif
   char buffer[QUERYSIZE]; /* request */
} packet;
```

The buffer contains the query when the master sends it to the slave; it contains the response when the slave sends it back. Note that the timing results (the three slave's phases and the overall slave processing time described previously) are present in the packet only when timing is of interest. The packet is smaller when the server is not undergoing timing performance measurements.

```
/*
**   Shared Memory Space
*/
struct area                   /* flag for every subtask */
{
   int flag[MAX];             /* 0:idle, 1:good, -:exit */
#ifdef TIMING
   int count[MAX];            /* number of queries handled */
   struct timeval phase2[MAX];
   struct timeval phase3[MAX];
   struct timeval phase4[MAX];
   struct timeval phase5[MAX];
   struct timeval phase6[MAX];
   struct timeval phase7[MAX];
   struct timeval phase8[MAX];
#endif
} area, *aptr;
```

Each subtask sets its flag, indicating its state to the parent task. Each subtask adds in timing results from the local node and from the remote node, via the extra space in the packet when timing is being measured.

The master process is started manually and is designed to trap the Ctrl+C interrupt, sending termination messages to each of its subtasks, which in turn send them to their slave processes, resulting (it is hoped) in a graceful distributed system shutdown. (Complete source listings are available in Appendix A, “The Source Code,” and source may be downloaded from the Sams Web site. Enter the ISBN number at `http://www.samspublishing.com` for further information.) The master prints a report of the phase timing results from each slave node, as well as the overall system processing. Figure 9.3 shows an actual report.

```
SWEEP    SLAVENAME    PH    NUM       TOTAL      AVERAGE
         ---------    --    ---     -------    ---------
              beta    28      1    0.391384     0.013978
                              2    0.017312     0.000618
                              3    0.206328     0.007369
                              4    0.021074     0.000753
                              5    0.208529     0.007447
                              6    0.004056     0.000145
                              7    0.206328     0.007369
                              8    0.125714     0.004490
             gamma    21      1    0.312508     0.014881
                              2    0.012343     0.000588
                              3    0.193668     0.009222
                              4    0.020774     0.000989
                              5    0.230004     0.010953
                              6    0.004376     0.000208
                              7    0.193668     0.009222
                              8    0.090228     0.004297
             delta     1      1    0.019779     0.019779
                              2    0.000947     0.000947
                              3    0.018193     0.018193
                              4    0.002140     0.002140
                              5    0.027182     0.027182
                              6    0.000276     0.000276
                              7    0.018193     0.018193
                              8    0.002402     0.002402
           OVERALL     1     50    0.723671     0.014473
                       2     50    0.030602     0.000612
                       3     50    0.418189     0.008364
                       4     50    0.043988     0.000880
                       5     50    0.465715     0.009314
                       6     50    0.008708     0.000174
                       7     50    0.418189     0.008364
                       8     50    0.218344     0.004367
```

FIGURE 9.3 Timing report from the master process.

After you start the master process, you may run many streams of test packets in any of the inter-arrival time (IAT) distributions supported by the client. When timing results are of interest, however, you must terminate the master (via a Ctrl+C keystroke) after each test stream to get a report that is specific to that test run. Execution phase timing plots come from these reports.

Figure 9.4 is an execution phase timing plot in the form of a bar chart that was constructed from three such timing runs—one for each of the three IAT distributions.

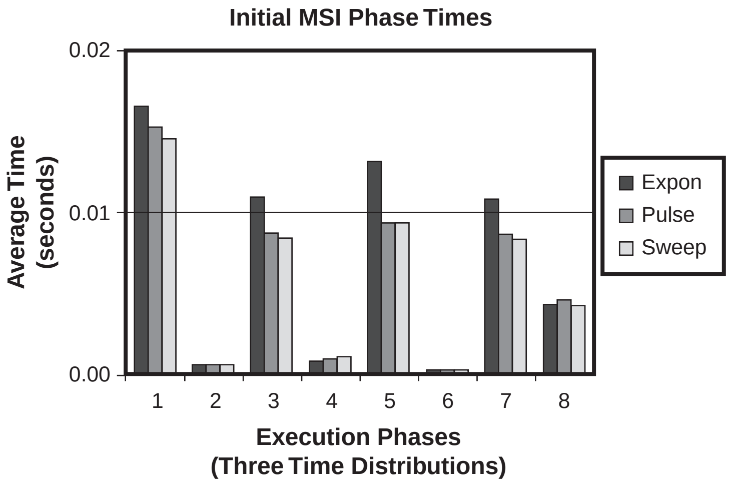

FIGURE 9.4 The baseline MSI execution phase profile.

The next step, the analysis, compares actual timing results from the plot to some common sense and some original expectations.

System Performance Improvements

External system performance (overall response time) is approximately the sum of these individual phase times. If you look back to Chapter 8, Figure 8.8 shows that a response time of approximately 50 milliseconds can be expected during a sweep IAT distribution, and the sum of the bar heights that represent the sweep timing data add up to 47 milliseconds—a good check of the results.

Assuming that the data in Figure 9.4 is correct, consider what can be gleaned from it that is both interesting and useful in the quest for faster response times. Performance measurement is a careful and laborious process, but performance analysis is often subjective, and sometimes fun—a combination of mathematics and gut feelings. Pause for a few seconds and just look at the data.

Phases 3 and 7 are equal by design. They are quite reasonable considering the packet size, the network speed, and the generally dated hardware.

Phases 2, 4, and 6 do not contribute much to the overall response time, and they may be ignored in this analysis, leaving phases 1, 5, and 8 for closer examination.

In phase 1, the master chooses a local subtask (from its pool) to receive an incoming query. The original code for this phase of the processing follows:

```
for(ident=0; ident<count; ident++)
{
   if(aptr->flag[ident] == 0) /* available? */
```

```
        {
           aptr->flag[ident] = 1; /* work to do! */
           usleep(1);
           break;
        }
     }
```

The call to `usleep` requests a one-microsecond pause, effectively relinquishing control of the CPU and allowing another of the ready processes to execute. This is a simple search for a subtask-idle flag, which is then reset to subtask-busy, instructing the subtask to receive an incoming query.

NOTE

The master process blocks on a `select` call until the presence of an incoming packet is detected. The subtask actually receives it after noticing that its subtask-idle flag is reset to subtask-busy.

With three subtasks running, it is hard to imagine how such a little bit of code can take such a long time. It is very likely that the master's `usleep` call is effectively blocking it, forcing it to compete on equal terms with each of its subtasks.

One way to increase overall performance is to increase the master's priority in this phase. Because the master is responsible for assigning work to its subtasks and spends most of its time blocked inside the `select` call, giving it a higher priority over its subtasks should speed up its notification of an eminent query arrival.

Our performance improvement to phase 1 is to add a call to `nice` (specifically, a `nice(1)` call to slightly lower its priority with respect to that of its parent) at the beginning of each subtask as part of its initialization.

In phase 5, the slave process retrieves the requested line of text from the shared text file. The source code from the slave that retrieves the line of text follows:

```
/*
**   Extract the requested line of text
*/
fp = fopen(file, "r");
while(line--)
   fgets(packet.buffer, sizeof(packet.buffer), fp);
fclose(fp);
```

There isn't very much going on in this phase either, but I/O timing can be misleading when there is contention for a shared device. Given the timing consistency across the three IAT distributions, however, this is probably not the case. What can we do to reduce the time it takes to extract a line of text from a file?

One solution is to access the file as if it were direct access, but that runs counter to the simplicity of the example.

Another solution is to maintain a cache, a local pool of earlier requests and their responses. If a duplicate request arrives, the master subtask can look up the response in its cache entry and respond immediately, saving a tremendous amount of time by avoiding the use of its remote slave process entirely. Typical test streams contain fewer packets than there are possible queries, however, so the probability that a duplicate will arrive is small. We can certainly change this ratio, but this solution also seems to be overkill for such a simple server.

Here is a compromise solution that is simple, yet shows the promise of dramatic performance improvement. Originally, every time a query arrives, the slave opens, reads up to the requested line, and then closes the file until next time. Given that the requested line numbers are random, the probability that the next line number will be higher than the previous one is approximately 50%.

If the file was already open when the next query arrives, it would have to be closed and reopened in only half the cases—where the requested line was lower than the one most recently retrieved. Also, in cases where the requested line number is higher than the previous one, not all the lines prior to the one requested need be read. Following is the performance improvement solution for phase 5:

```
/*
**   Extract the requested line of text
*/
if(line > curr) /* false first time */
{
   line -= curr; /* adjust to offset */
   curr += line; /* save for next time */
}
else
{
   fclose(fp); /* ignored first time */
   fp = fopen(file, "r");
   curr = line; /* save for next time */
}
while(line--)
   fgets(packet.buffer, sizeof(packet.buffer), fp);
```

In phase 8, the master's chosen subtask responds to the query and declares itself idle.

```
/* result back to client */
strcpy(buf, packet.buffer);
send(cli, buf, strlen(buf)+1, 0);
close(cli);

/* check for exit request before continuing */
```

```
if(aptr->flag[ident] < 0) /* exit request? */
{
   sendto(s_fd, "Terminate 0", 12, 0,
      (struct sockaddr *) &slavelink, len);
   break;
}
aptr->flag[ident] = 0; /* no: flag as idle */
```

There doesn't seem to be any possible improvement in this phase. Leave this code just the way it is for now.

Final MSI Performance Results

The "before" data in Figure 9.5 can be compared to the improved results shown in Figure 9.5.

```
SWEEP   SLAVENAME     PH    NUM         TOTAL      AVERAGE
        ---------    ----   -----     -------    ---------
             beta     47       1     0.007628     0.000162
                               2     0.026164     0.000557
                               3     0.340557     0.007246
                               4     0.047095     0.001002
                               5     0.255919     0.005445
                               6     0.010009     0.000213
                               7     0.340557     0.007246
                               8     0.208840     0.004443
            gamma      3       1     0.000485     0.000162
                               2     0.001841     0.000614
                               3     0.033318     0.011106
                               4     0.002692     0.000897
                               5     0.034676     0.011559
                               6     0.000423     0.000141
                               7     0.033318     0.011106
                               8     0.009087     0.003029
            delta      0       1     0.000163     0.000000
                               2     0.000000     0.000000
                               3     0.000000     0.000000
                               4     0.000000     0.000000
                               5     0.000000     0.000000
                               6     0.000000     0.000000
                               7     0.000000     0.000000
                               8     0.000000     0.000000
          OVERALL      1      50     0.008276     0.000166
                       2      50     0.028005     0.000560
                       3      50     0.373875     0.007478
                       4      50     0.049787     0.000996
                       5      50     0.290595     0.005812
                       6      50     0.010432     0.000209
                       7      50     0.373875     0.007478
                       8      50     0.217927     0.004359
```

FIGURE 9.5 Post-improvement timing report from the master process.

Similarly, the "before" execution profile in Figure 9.4 may be compared to the improved results shown in Figure 9.6.

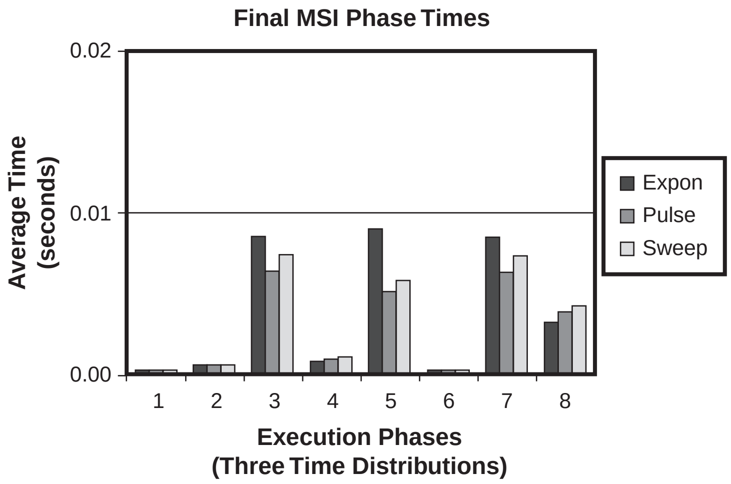

FIGURE 9.6 Post-improvement MSI execution phase profile.

Look at the individual improvements, by phase.

Phase 1—The addition of a call to the `nice` function was a complete disaster, nearly doubling phase 1 execution time. Closer inspection revealed the `usleep(1)` call to be the culprit; it was inside the section of timed code. When the processor is released to the next ready process, the amount of time before control is regained (to establish the phase-end time) is unpredictable. After the call to `usleep` was moved outside the timed section of code, phase 1 execution time was too low to be of further interest.

Phase 5—The decision to leave the file open in case a subsequent line of text was requested was a resounding success, cutting the phase 5 execution time by one third.

Phases 3 and 7 (network transit times)—These showed a slight reduction for no apparent reason. All the other phase execution times reported were similar to phase execution times before the performance improvements were implemented.

Finally, some overall performance comparisons follow that were promised in Chapter 8. The response-time profiles for the exponential, pulse, and sweep IAT distributions are shown in Figures 9.7, 9.8, and 9.9, respectively.

```
                                   OBSERVATIONS
RESPONSE   |
TIME (mesc)|    10         20         30         40         50
-----------+-----+-----+-----+-----+-----+-----+-----+-----+-----+
    1   10 |
   11   20 |***
   21   30 |********************************
   31   40 |********
   41   50 |**
   51   60 |**
   61   70 |
   71   80 |*
   81   90 |
   91  100 |
 OVER  100 |*
           |
                              50 Total Observations
```

FIGURE 9.7 Exponential distribution—improved response times.

```
                                   OBSERVATIONS
RESPONSE   |
TIME (mesc)|    10         20         30         40         50
-----------+-----+-----+-----+-----+-----+-----+-----+-----+-----+
    1   10 |
   11   20 |*
   21   30 |********************************
   31   40 |**************
   41   50 |***
   51   60 |*
   61   70 |*
   71   80 |*
   81   90 |
   91  100 |
 OVER  100 |
           |
                              50 Total Observations
```

FIGURE 9.8 Pulse distribution—improved response times.

```
                  |                    OBSERVATIONS
RESPONSE          |
TIME (mesc)       |       10        20        30        40        50
------------------+---------+----+----+----+----+----+----+----+----+-
       1    10    | **
      11    20    |
      20    30    | **********************
      31    40    | ********************
      41    50    | ***
      50    60    |
      61    70    | **
      71    80    |
      80    90    | *
      91   100    |
    OVER   100    |
                  |
                          50 Total Observations
```

FIGURE 9.9 Sweep distribution—improved response times.

Compare them to their Chapter 8 counterparts in Figures 8.6, 8.7, and 8.8. Improvement is shown in all cases.

Summary

Chapter 9 begins by introducing execution profiling, a common performance measurement technique. Profilers have gotten cheaper and more sophisticated over the years, but they are still not well suited to distributed systems analysis.

Direct measurement is perhaps the most demanding performance measurement technique, but it lends itself well to a distributed environment. In the case of a cluster server, query processing can be broken into discrete execution phases. Execution times for each phase either can be measured directly or estimated from a higher-level direct measurement. The results, plotted on a time-ordered bar chart, are used to analyze the internal system performance. Where execution times are seen to be unexpectedly long, the source code associated with that phase can be optimized for speed. When all the suspect phases are examined and their code optimized, another performance run produces a subsequent bar chart, which can be compared to the earlier baseline and examined for any performance improvement.

Finally, the overall (external) system performance is measured and compared to earlier results. The goal is, as always, client satisfaction, and clients are most satisfied when response times are low.

Further Reading

- Tanenbaum, Andrew S. *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

10 Robust Software

IN THIS CHAPTER

Robust software is software that keeps running correctly in spite of error conditions. Things can go wrong during the execution of the examples presented in earlier chapters. In this chapter, you will learn what kinds of errors are likely to occur in the MSI and what you can add to your software to prevent its failure in the presence of such errors.

If you've worked through the text this far and followed along with your own cluster system development, you have probably discovered how unpredictable distributed software systems can be. The master process, for example, has an interrupt handler that catches a Ctrl+C keystroke. On receipt, the parent sets flags for each of its subtasks, telling them to end gracefully. They send a datagram to each of their slave counterparts, telling them to exit gracefully, too, before cleaning up and exiting themselves. The parent waits for each subtask to exit before removing the shared memory area and exiting.

This is a good design, yet there are times when some of the processes terminate in a manner that is less than graceful, or not graceful at all.

Most of the time, even after several streams of packets, a Ctrl+C keystroke produces the expected result: the master, all its subtasks, and all the remote slave processes terminate normally. A look at each node via

```
> ps aux
```

shows all these processes terminated. A look at shared memory areas via

```
> ipcs -m
```

shows that both the client and the master removed their areas before exiting. Every so often, however, you will find a subtask running without its parent or a remote slave process still running along with its performance subtask on one of the remote nodes.

You kill these processes by their process IDs:

```
> kill <pid> … <pid>
```

remove the shared memory area by its ID:

```
> ipcrm shm <shmid>
```

and recheck everything before beginning another test session. Humans seem to recover from these errors without any difficulty.

Datagrams can get lost, which might explain the remote slave problems, but after many years of working with distributed systems, you come to believe that things happen—things that you will never fully understand. You know exactly what to do if a process is left running or a shared memory area still exits before resuming testing, but what can your software do to avoid failing in the presence of such anomalies?

Alarm Exits

The central performance monitoring process (refer to Listing 8.3) runs on a machine that is external to the cluster. It gets regular performance statistics from each node and displays the changing state of the overall system. All the cluster nodes are shown, along with their CPU and memory utilization, their disk and network I/O rates, and an estimate of the overall network load. You might want to have a timer inside the monitor, causing it to terminate after an hour of inactivity. You can do this with a call to `alarm`:

```
#include <unistd.h>
/* . . . */
main(void)
{
    while(1)
    {
    alarm(3600); /* alarm in one hour */
    waitForSomethigToHappen();
    alarm(0); /* cancel the alarm */

    /* respond accordingly */
    }
}
```

The `alarm` call requests that a signal be sent to your process after 3,600 seconds, or one hour. If no activity occurs in that amount of time, the signal will be sent, and your process will be terminated. This happens because you did not tell the OS to ignore that particular signal, nor did you establish a signal handler for it. The OS takes the default action; *it terminates the process*.

But this is a good thing. The monitor should be getting several packets each second. If there has been an hour of inactivity, you may not want that process to remain in the system indefinitely. Adding this simple device ensures that it is terminated after a configurable amount of inactivity. If it does get a packet from a performance statistics gatherer, the alarm signal request is canceled via `alarm(0)`. It responds to whatever condition it finds, resets the inactivity alarm, and waits for another packet—normal operation.

Timeouts

If you don't want your process to be rudely interrupted, so to speak, you can add an alarm signal handler—an internal function that tells the process when an alarm signal is received so that it can exit gracefully.

Listing 7.5 is an example of the master process using a broadcast UDP datagram to start the remote slave services. This simple program reports the hostnames where slave processes sent back a registration packet as they become active. In the production version of the MSI, however, the master will not know how many slave registration packets to expect. Without the luxury of waiting indefinitely for all possible registration packets, it uses an alarm signal as a timeout indicator. If no more packets arrive within the set time limit, it assumes that all possible remote slaves have registered.

Two headers are required by the `alarm` and `signal`, and you will need a Boolean variable that can be set by the signal handler when an alarm goes off and that can be tested by the main program each time it is unblocked from the `recvfrom` call. The Boolean variable is `FALSE` for a packet arrival and `TRUE` for an alarm signal, indicating the timeout condition.

NOTE

The `recvfrom` call unblocks when either an expected packet or some signal, including alarm, is received.

```
#include <unistd.h>
#include <signal.h>
/* . . . */
#define FALSE 0
#define TRUE 1

int Alarm = FALSE;
```

The main program must declare the `alarm` signal handler at compile and link time and establish it at runtime.

```
void sig_alarm();
signal(SIGALRM, sig_alarm);
```

The `while` loop at the bottom of Listing 7.5 can be changed to the following:

```
/*
**   Display registration response info
*/
    while(1) /* alarm signal causes exit */
    {
        alarm(1); /* set a 1-second alarm */
        buf[0] = '\0'; /* clear input buf */
        /* wait for registration or alarm */
        recvfrom(r_fd, buf, sizeof(buf), 0,
                 (struct sockaddr *) &registers, &len);
        if(Alarm)
            break;
        alarm(0); /* cancel alarm timer */
        if((p=strchr(buf, '.')) != NULL)
            *p = '\0'; /* trunc .domain */
        printf("\tSlave on host '%s' registered\n", buf);
        strcpy(hosts[count++], buf);
    }
    close(r_fd);
    printf("\tRegistered %d remote slave task(s)", count);
```

This detects the end of incoming registration packets. If no packet is received after a full second, all the slave processes are assumed to have registered.

Finally, the signal handler itself is included as a function at the end of the program.

```
void sig_alarm()
{
    /* reestablish handler */
    signal(SIGALRM, sig_alarm);
    Alarm = TRUE; /* set flag */
}
```

A complete listing of the master source code is in the Appendix A, “The Source Code.”

Subtask Restarts

The master process starts one local subtask for each registered remote slave process. Each local subtask establishes a permanent communication link with its associated remote slave process. The master assigns incoming queries to the first available subtask, which hands off the work to the remote slave for processing. How can the master discover that a local subtask has failed, and what can it do in such an event?

Similar to the `alarm` call sending a `SIGALRM` signal to the requesting process when the time runs out, the termination of a subtask causes a `SIGCHLD` signal to be sent to that child's parent. If you establish a signal handler and keep track of all your subtasks, not only can you detect the termination of one of them, you can tell which one ended and then restart it (see Listing 10.1).

LISTING 10.1 Recovering from the Unexpected Termination of a Subtask

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#define FALSE 0
#define TRUE 1
#define NUM 3

int Okay; /* Boolean */

main(void)
{
    /*
    **   Listing10.1.c - Recovery from the termination of a subtask
    */
    void sig_child();
    int pids[NUM];
    int ii, jj;
    int pid;

    /* setup SIGCHLD handler */
    signal(SIGCHLD, sig_child);

    /*
    **   Start NUM subtasks
    */
    for(ii=0; ii<NUM; ii++) {
        if((pids[ii]=fork()) == 0) {
            printf("\tSubtask #%d started, pid = %d\n", ii, getpid());
```

LISTING 10.1 Continued

```
            while(1)
                usleep(1);
        }
    }

    /*
    **   Wait for a subtask to end
    */
    while(1) {
        Okay = TRUE;
        while(Okay) /* pause */
            usleep(1);

        pid = wait(NULL); /* exiting pid */
        for(ii=0; ii<NUM; ii++)
            if(pid == pids[ii])
                break;
        printf("\tSubtask #%d ended, pid = %d\n", ii, pids[ii]);
        if((pids[ii]=fork()) == 0) {
            printf("\tSubtask #%d restarted, pid = %d\n", ii, getpid());
            while(1)
                usleep(1);
        }
    }
    return(0);
}
void sig_child() {
   signal(SIGCHLD, sig_child);
   Okay = FALSE;
   return;
}
```

Start this program in one window (or login session) and note the process IDs reported by the subtasks. Terminate one or more of these subtasks in another window and watch the recovery happen in real-time. Terminating the parent process ends the demonstration.

You may want to enhance this program by including a list of counters, one per subtask. Each time a particular subtask terminates unexpectedly, decrement its counter and restart it, but only if this restart limit hasn't been reached.

Main Task Restarts

When a subtask terminates, a signal is sent to its parent task, but how do you detect the termination of a parent task? There are a few ways to handle this case, depending on how the parent is started.

One simple approach is to develop a high-level startup process. Its job is to start all the processes that would normally run at the highest level and watch them, restarting any that might fail.

Another strategy is to let the OS start the process and specify that it be restarted on failure. The system file /etc/inittab allows you (as root) to add your own executable to its list of processes to be started (and restarted, if necessary) when a particular run level is entered. See `man inittab` for further details. The downside of using this technique is that a process can become difficult to get rid of—it works too well!

One final strategy involves detecting what fault-tolerant software designers call a "heartbeat." The master's parent process can detect the termination of a subtask, but consider how it might detect (and recover from) the termination of a slave process running on a remote node.

The slave registers with the master as soon as it becomes active, so an unused UDP communication link exists between the two. If the slave sent a UDP datagram to the master every second (its heartbeat), and the master kept track of the time that the last such message was received for each slave, it could scan these last-heard-from times each time a query arrives. Any slave that had not been heard from in some configurable amount of time could be restarted (via its UDP service port) along with its local subtask, for total recovery. (This is a good enhancement project for those readers who are actually developing a cluster system.)

Finally, consider that there are levels of recovery, and at some point, a clever process might decide that a remote machine needs to be rebooted. You might establish a UDP service port that starts a short program, running as root, that issues the shutdown command (with a restart option) via a system call:

```
#include <stdlib.h>

main(void)
{
    system("/sbin/shutdown -r now");
}
```

If it is not possible to run a process on a failed machine, a reboot might still be accomplished through custom hardware. Processor chips have a RESET pin that, when grounded, will cause a system reset, or reboot. Such a drastic measure, of course, should be considered only under highly unusual circumstances. This kind of reboot is akin to cycling the power switch, a very cold restart indeed.

Reattaching Shared Memory

Even if you decide to implement one or more of the preceding schemes, you still may not be completely out of the woods. Plenty of residual conditions are waiting to ensure that this latest instantiation of your process fails as well. The most likely suspect in the MSI architecture is its shared memory area.

If the master fails, terminating before removing its shared memory area, that area will still be present when the master restarts. You must decide, based on your knowledge of how your system works, whether it's best to have the master reattach that same area or to remove the old shared memory area and start afresh.

The master can attempt to reattach its shared memory area during its initialization. If the return value from the `shmat` call is unsuccessful, either the failing master removed the memory area, or this master is the first one active since a reboot. (A reboot covers a multitude of sins.)

Reliable UDP Communication

Software developers love TCP because it guarantees successful delivery of all packets sent, in the order that they were sent. Software developers hate TCP because it guarantees successful delivery of all packets sent, in the order that they were sent. It seems that the overhead associated with such a strong guarantee is intolerable in cases where processing time is of the essence.

UDP is often called an *unreliable* protocol, but that may be an overindictment. You can improve its reliability with the timeout feature described earlier in this chapter, resending any datagrams that are not acknowledged within set limits. Because you are setting these limits, you can control just how much overhead you are willing to suffer for a given level of reliability.

This simple send and wait for an acknowledgement protocol is referred to as the stop-and-wait flow control. One variation on this scheme allows a set number of outstanding packets (called the buffer window) before waiting for an acknowledgement. That is, you continue to send packets until you've reached the set limit, and then you wait for acknowledgements, improving your throughput. Further improvement can be had by allowing the receiver to acknowledge several packets (up to the window size) in a single response. These point-to-point communication protocols have been refined over the years and can be found in any high-quality text on the subject. See the references in the "Further Reading" section at the end of this chapter.

Summary

Designing robust software up front—or finding and removing vulnerabilities in an experimental system—should lead to a reliable cluster server. You find out what can go wrong through hands-on testing of your system and use this knowledge to add detection and recovery software to your system, minimizing the probability of seeing those kinds of problems again. (You have better things to do with your time than finding and terminating stray processes and removing leftover, shared memory areas.)

Distributed software systems can be complex. Several subtasks communicate across the network to their remote node counterparts, working together toward the common goal: servicing clients' needs. Strong process interdependence should not result in a system failure because of an unexpected termination in one of them. You can detect and recover from process failures and reestablish disconnected communication links. You can design a system in which process failure results in another stepping in to take its place.

The result of your efforts, the focus of this chapter, is robust software: software that does exactly what you want it to do, when you want it to do it.

Further Reading

- Jalote, P. *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.
- Lyu, Michael R. (editor). *Software Fault Tolerance*. West Sussex, England: J. Wiley and Sons, 1995.
- Stallings, W. *Data and Computer Communications*. 4th ed. New York: Macmillan, 1994.

11 Further Explorations

IN THIS CHAPTER

- Beowulf-like Supercomputers
- Supercomputer Applications
- Ad Hoc Peer-to-Peer Networks
- Future Applications
- Further Reading

The hardware for your cluster—the PCs, the monitor and keyboard, and the network hub and cabling—are generic. You can use this same hardware to develop any kind of multicomputer system. Most of the OS configuration changes you made to whatever Linux distribution and version you run are generic, too. The software for the MSI, however, is specific to this project, which highlights an earlier point about how the software defines the "personality" of a cluster system. This chapter is about some other kinds of cluster systems you might want to design and build now that you're an experienced systems architect. Get ready to broaden your horizons, to face bigger challenges.

The MSI is a distributed transaction server system. It receives a query, processes it in parallel with other queries, and sends the response back to an external client. It initializes at startup and runs forever after. Not all distributed applications work like this.

As an application's requirements increase, so does its complexity. The amount of effort necessary to upgrade a complex application eventually becomes cost prohibitive for two reasons. First, the amount of work necessary to support a new feature can require additional infrastructure software to support it. Second, the complexity of a distributed system's infrastructure software can make it brittle, more prone to failure because of the increased complexity.

The classic solution to these infrastructure-related problems was to encapsulate the infrastructure software in a package—a function library that the application programmer can use to reduce the cost of development and enhancement. Two of the more popular packages can be downloaded for free from the Internet. The PVM (parallel virtual machine) and the MPI (message passing interface) package download sites are referenced in the "Further Reading" section at the end of this chapter.

As a result, modern clusters have a two-tiered software layer: the application and the distributed system infrastructure. Think about how this impacts a multicomputer system's personality as you learn about some variations in their architecture.

Beowulf-like Supercomputers

What makes a supercomputer Beowulf-like? A Beowulf-class machine has two constraints: it must be built from commercially available PCs and support hardware, and it must cost less overall than a high-performance scientific workstation. This cost was approximately $50,000 in 1994, when definitions like this one were being refined. The cost of the test bed described in this book was less than $2,000 in 1998, and it can be built for even less than that today. The cost of your system depends on how much emphasis you place on the "high" in high performance. Cluster performance depends heavily on the speed of the external I/O bus, which is the interconnection network or Ethernet in the case of the test bed and most other clusters. Beowulf reverted to custom internetworking hardware to achieve some spectacular cost/performance ratios in the later stages of its development.

Scientific researchers pit their creativity against an inadequate budget, squeezing every ounce of utility out of existing equipment. The Beowulf Project was motivated by recent developments in parallel computing on Pile-of-PC (POPC) clusters, where impressive benchmarks were achieved with COTS, or commercial off-the-shelf products. (See also references to COW, for cluster of workstations, and NOW, for network of workstations, in the "Further Reading" section at the end of this chapter.) Beowulf is a NASA initiative headed by Thomas Sterling and Donald Becker at the Center of Excellence in Space and Data Information Sciences, a NASA contractor. Scientists work with huge amounts of data generated by the simulation of large-scale physical phenomena (interacting tectonic plates, for example) or collected by a network of sensors, such as weather satellites. Beowulf was chartered to solve the problems associated with this kind of research, at a bare-bones cost.

Caltech and Los Alamos National Laboratories run Beowulf clusters, as do large and small universities alike. Large schools (Princeton and Oregon State, for example) focus on how high-speed networking hardware affects the limitations of message passing and virtual memory mapping systems. Smaller universities, even some high schools, build and maintain such systems to give students hands-on experience with parallel computing concepts and challenges. They are ideal test beds for experiments, class projects, and post-graduate research. You can read more about these Beowulf projects at the Web sites referenced in the "Further Reading" section at the end of this chapter. There may also be some interesting material at `http://www.redhat.com` regarding their collaboration in 1998 with the Beowulf Project in releasing NASA's official software as an Extreme Linux CD-ROM.

A solidly written, free OS such as Linux, a glut of outdated but reasonably high-performance PCs, and some bright young experimenters who are willing to share their ideas have made all this possible. Their enthusiasm is so evident as you read their goals, accomplishments, and results that you can't help but at least consider building a system of your own.

Supercomputer Applications

The MSI is a transaction processor in the Web or database server sense. It receives incoming queries and sends a response back to some external client. A real-time scientific application exhibits similar behavior in that it receives acquired or simulated sensory data and develops a solution. Most scientific applications present a static solution, but some systems reacquire data at regular intervals and display a continuously updated solution that changes with the environment.

For example, consider a major city that, interested in its air quality, installs a dozen or more air pollution particulate sensors. Each sensor measures wind speed, wind direction, temperature, humidity, and levels of ozone, carbon monoxide, sulfur dioxide, and other air particles that contribute to known respiratory health problems. It averages these counts over a 10-minute period and transmits the readings to a central computing site. This central computer's job is to receive all the latest data and update its real-time display, a false-color image depicting the location, shape, and density of each particulate. Not too surprisingly, around rush hour, the shape of the carbon monoxide pattern resembles a crude map of the highways and arterial surface streets of that city.

On a larger scale, the National Center for Atmospheric Research (NCAR) has weather satellites that feed sensory data into computers, which generate real-time global weather patterns. Software simulation data is also used to generate predictions of future weather patterns. Hopefully, the software is efficient enough, and the computers are fast enough to generate tomorrow's weather before tomorrow arrives. More information can be found at the Universities Corporation for Atmospheric Research Web site, referenced in the "Further Reading" section at the end of this chapter.

Artificial intelligence (AI) researchers take their games very seriously. An IBM RS/6000 SP computer, nicknamed Deep Blue, played against Garry Kasparov, a chess grand master, and won. In this kind of application, the current board arrangement is the acquired data, and the solution is a single legal move that is judged best by looking at its consequences several moves ahead. The software builds an inverted tree structure beginning with the current board. Each lower branch represents a legal move, and each branch below those represents a possible response from the opponent, and so on. This tree structure is called a problem space and the number of branches below each possible move is called its depth. Searching for solutions and evaluating each in terms of its long-term ramifications can take a tremendous amount of execution power. Because each sub-branch is independent of the others, however, a parallel computing architecture can reduce the overall problem space search time by having each processor working on a separate branch until the entire space is solved. Clever ways exist to reduce the number of branches, called "tree pruning" algorithms. See the "Further Reading" section for a reference to one of the better books on AI.

Engineering problems, a design of the heat radiating fin structure on a modern high-speed processor chip, for example, might use finite element analysis to judge the effectiveness of a given fin structure. Instead of looking at the fins as a solid piece of metal, it models it as being constructed of tiny volumes of metal called finite elements. A thin layer of elements makes up the face that is in contact with the top of the chip. As the chip heats up, this face layer absorbs heat from it. The next layer of elements above the face absorbs heat from them, and so on, until the outer fin elements transfer their heat to the surrounding (perhaps moving) air. The tinier these elements are, the longer the analysis takes, but the closer the model reflects reality. Simulated chip processing requirements cause the simulation to change the temperature of the chip's surface, which causes the heat to transfer throughout every one of the elements in the fin structure and finally into the air. The amount of heat that a particular element can absorb is a function of how hot it is now and how hot its surrounding elements are. An ideal fin structure has outer fins that transfer heat to the air so efficiently that the elements cool quickly, absorbing heat from the next layer down; then they can absorb heat, and so on, right down to the face layer, so that the chip doesn't melt while you're working on some programming of your own.

Software simulations can compress billions of years of geological time into a few seconds or expand a few nanoseconds at the beginning of time (the Big Bang) into several minutes so that geologists and astrophysicists can study the behavior of their phenomena of interest in a reasonable amount of real-time. Software simulations can derive great benefit from distributed simulation, and this is a very hot research topic.

Each of these kinds of applications, and the list is not exhaustive, has its inherent parallelism that can be exploited by a parallel processor. Parallel compilers can detect such parallelism by analyzing your source code. (This process itself is a candidate for a supercomputer application!) These compilers are very expensive, however, and alternatives exist. The scientific application developer is perhaps the best person to determine where parallelism exists in the process. He or she can break these sections into blocks of code that work on subsections of the problem and arrange them into a scheduling tree, as shown in Figures 11.1a and 11.1b.

```
int  costs [3] [10];
int  sums [3];
int  total;
int  i, j;

main (void)
{

     /* initialization . . .*/

     total = 0;
     /* sum the costs */
     for (i=0; i<3; i++)
     {
          sums[i] = 0;
          for(j=0; j<10; j++)
               sums[i] += costs [i] [j];
          total += sums [i];
     }
     /* . . . cleanup */
}
```

FIGURE 11.1A An algorithm with inherent parallelism.

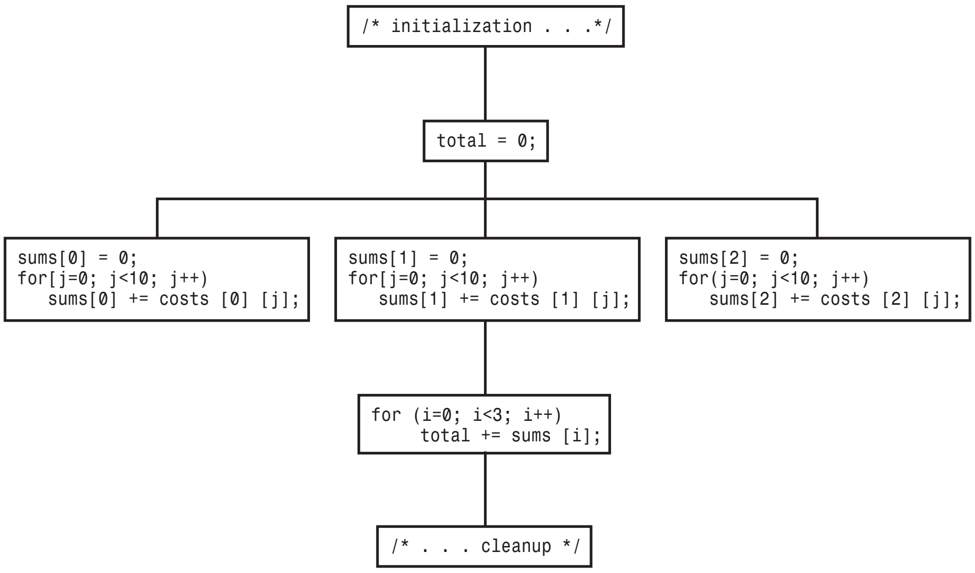
/* initialization . . .*/
total = 0;
sums[0] = 0; for[j=0; j<10; j++) sums[0] += costs [0] [j];
sums[1] = 0; for[j=0; j<10; j++) sums[1] += costs [1] [j];
sums[2] = 0; for(j=0; j<10; j++) sums[2] += costs [2] [j];
for (i=0; i<3; i++) total += sums [i];
/* . . . cleanup */

FIGURE 11.1B A parallel process-block scheduling tree.

The master node executes the initial thread until it detects the three-way split. It updates the three process blocks with the latest values for any variables they might depend on, sends them to three available slave nodes, and then waits for the slaves to complete their parallel operations. As each slave node finishes its process block, it sends the results back to the master node, which continues waiting until all three slave nodes are complete. It updates the local single-thread process block variables with the results returned from the slaves and resumes the processing of that single thread to completion or until another split occurs.

The parallel parts of this program would take approximately three times longer to execute on a uniprocessor. On a multicomputer with at least three CPUs, it would take one third that time, plus the overhead of transferring each block to a separate computer, transferring the results back, and marshaling the individual results for continued processing along the single thread of processing. You must consider the time-saving benefits of parallel processing in light of the time consumed managing the parallel implementation. The whole `sums` array must be broken into three parts before the three-way split can occur. The three results must be combined back into that array before the next steps in single-threaded processing can continue. Remember that the only part where any savings can be realized is during the parallel sections of the process-block scheduling tree.

Ad Hoc Peer-to-Peer Networks

The MSI is a hierarchical architecture—it is a tree structure with the master node at the base and slave nodes on each branch. Not all clusters are organized like this.

Peer-to-peer networks do not have a master and slave relationship. Rather, they each can take on either role, as needed. These nodes are programmed identically, with the same processes running. Each will accept incoming data and ask for assistance as if it were a master, and each will accept a request for assistance from another node as if it were a slave. Some of these distributed systems can form ad hoc (temporary, as needed) networks on-the-fly and disassemble them just as quickly when they are no longer needed.

As an example, consider an automobile on a four-lane highway, in medium traffic. Today you activate the cruise control, settle into the right lane, and let others go by you on the left. As you approach an exit ramp, a car comes past you on the left, puts on a right turn signal, and crosses over your lane to make that exit. You see it coming, touch your brakes, and move into the left lane in case the car brakes before exiting, adding injury to insult. You relied upon your instinct, your vigilance, and the key visual clue of the other car's turn signal. You did not look behind you before you touched the brakes, however, nor did you check too carefully before moving into the left lane. Not a problem this time, but this could have ended differently.

Fast-forward about six years. You are in the same situation, but most cars are equipped with an onboard driver-awareness system, a node computer with a bidirectional radio frequency link, and a display on the dashboard. As soon as the car comes by you on your left and begins to drift into your lane, a warning tone sounds both in your car and in his. As the first car came up behind you in the left lane, its beacon was picked up by your system, or perhaps its system picked up yours. In any event, your two systems became aware of each other's relative position and speed.

When the car went by you and drifted into your lane, the car's system sensed the change in relative position, turned on the tone to warn its driver, and sent a message to your car, which also sounded a tone to warn you. When you touched your brakes, your system knew there were no cars behind you and did not send a message. When you drifted left, it knew this too was not a problem and remained neutral.

You still saw the other car's turn signal. If the driver forgot to turn it on, the system, sensing the drift, would have done so. When you touched the brakes, your brake lights came on, and when you drifted left, your turn signal came on, just in case there was an older vehicle back there without this kind of gear. This scenario was intended as an example of a peer-to-peer network. If you can see how each node is completely autonomous, willing to introduce as well as respond to issues, you should have a feel for the potential of such architecture. It is good exercise to think about how complex the node-to-node interaction might get with one extra car behind you or a fourth car in the left lane, behind the one that cut you off. Ask yourself some questions: If the car behind you didn't see your brake lights, should its system automatically slow down that car? Should the car on your left slow itself down when it senses that you are about to cut it off? Should you trust such systems to make the right decision, considering your experience with commercial software to date?

As futuristic as this sounds, the technology to implement such a system is perhaps 10 years old. The next section discusses applications that get a bit closer to the technological edge.

Future Applications

You are probably familiar with the PDA, or personal digital assistant. You probably know that it is becoming more like a cellular telephone every day and that many PDAs can access the Internet. What follows is not too much of a leap from what we have today. It accepts the promise of electronic commerce, takes its lead from today's corporate intranets, and adds an extension to the shared virtual memory concepts of the DSM.

The problem with the handheld PDA device is the size of the screen. When the screen is of acceptable size, the device is inconvenient to carry. If there is no objection, let's assume that there is a folding-screen version that fits into your pocket, purse, or clips on your belt when not in use (see Figure 11.2). It's an ordinary PDA unless you want it to interact with whatever environment you happen to be in—more on that aspect later.

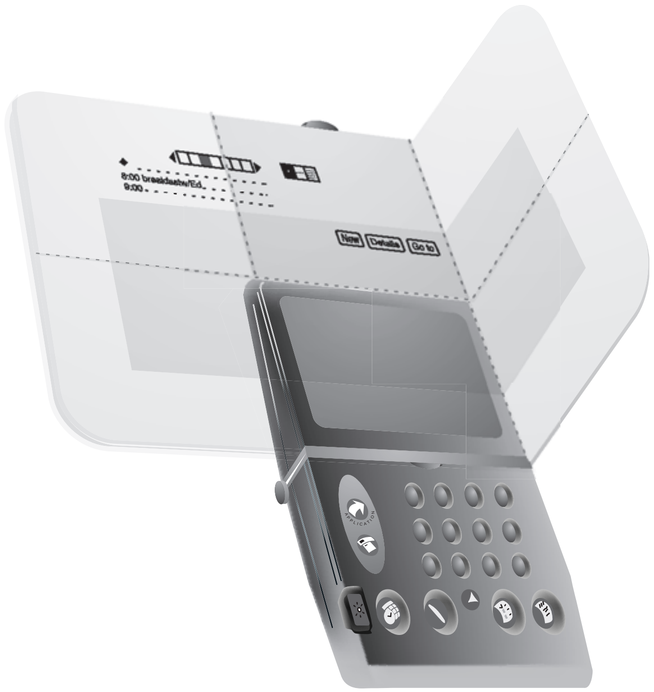

FIGURE 11.2 Personal digital assistant with a fold-out display screen.

Now apply the corporate intranet concept to your school campus, your bank, a department store, an office building, or some other institution. Public access intranets allow anyone to interact. These might be found in a public museum or library, a historic site, a city skyline tour, an auto dealership, an airport terminal, and so on.

Semiprivate intranets allow you authorization to interact. Your system is temporarily authorized when you pay admission to a theme park, a private art gallery, or a theater. A private intranet requires stronger authorization. Your bank must know you are a depositor, your school campus must recognize your student ID, and your workplace must recognize your employee status.

On campus, you can check your midterm exam grade, download the paper you brought from your desktop computer, and schedule a meeting with your advisor for later that afternoon. At the local snack shop, you can order a sandwich while you're finding a place to sit and then pay for it in advance, if you like. You can get a look at traffic when you get into your car and be presented with some alternate routes home if the usual one is congested for any reason.

And if you want to look for a library book while you're eating lunch, you can always do that the old-fashioned way: dial your ISP and connect through the Internet.

Finally, perhaps the ultimate cluster software architecture is one in which the structure is defined by the processing requirements instead of by the available hardware. New divisions of a corporation and brand-new companies are represented by a corporate chart of positions first, people second. The distributed software architect should be designing the process chart that will get the job done, without regard for the hardware at first. Each of these processes, with their interprocess communication links defined in advance, should never be burdened with any concerns related to where they execute relative to their peers. If a process senses that it isn't capable of meeting its goals from where it happens to be executing (too many processes are running there already, for example, slowing down access to the local resources), it should be able to send itself to some other machine for the betterment of the overall system performance.

The concept of a process reestablishing itself on some other machine is described under autonomous agents, but that area of computer science has the vision of your personal agent roaming throughout the Internet, seeking information for you. It may be looking for a good restaurant in Seattle, for example, knowing you will be there on business this week. It carries a matrix describing your preferences in food and price range and knows where you will stay. When it finds an optimal solution, it reports to your desktop, where the information waits until your PDA is in range for a download.

Ubiquitous computing will soon be the production, not the demonstration package. The software and hardware will be designed by fresh minds that are unburdened by the way things have always been done and by flexible minds willing and anxious to shed old ways of thinking in the light of new problems. Join the fun.

Summary

The MSI is but a toy compared to commercial systems, but it is an excellent learning tool nonetheless. Modern systems employ two-tiered software architectures, encapsulating the interprocess communication software to remove its influence on the design and enhancement of the high-level service application.

Beowulf-class architectures are the classic distributed systems design driving much of the cluster architecture development today. Scientific applications provide the motivation for such systems, perhaps more so than information servers do today, but this is changing.

Parallel processing of scientific applications requires a preprocessor of some sort to break up an algorithm into parallel processing blocks and an execution-time manager to schedule these blocks on available processors, consolidating their results for continued processing of the single, central thread.

Networks don't have to be fixed. Ad hoc networking—among automobiles on the highway or soldiers on a battlefield—allows communication where none would be possible without tremendous effort, filling in vast areas with cellular-style communication base stations wherever highways are built or battles might be fought.

Finally, thinking outside the box, consider a fixed network where the processes move from node to node. The future may hold a world where computational and communicational power is as ubiquitous as AC power is to the appliances we use today. And there will be no shortage of technical challenges.

Further Reading

- Hill, J., M. Warren, and P. Goda. "I'm Not Going to Pay a Lot for This Supercomputer!" *Linux Journal*, no. 45 (January 1998): 56-60.
- `http://www.netlib.org/pvm3` for the parallel virtual machine (PVM) download.
- `http://www.netlib.org/mpi` for the message passing interface (MPI) downloads.
- `http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html` for Pile of PCs information.
- `http://www.beowulf.org` for general Beowulf information.
- `http://www.beowulf-underground.org` for the more adventurous Beowulf fans.
- `http://www.cs.orst.edu/swarm` for information on the SWARM system.
- `http://www.cs.princton.edu/shrimp` for information on the SHRIMP system.

- `http://ess.jpl.nasa.gov/subpages/reports/97report/hyglac.html` for Hyglac information.
- `http://www.cacr.caltech.edu/beowulf` for information on Naegling.
- `http://loki-www.lanl.gov` for information on the Loki system.
- `http://www.ucar.edu/ucar` or the NCAR site at `http://www.ncar.ucar.edu/ncar` for information on the National Center for Atmospheric Research.
- Rich, E. and K. Knight. *Artificial Intelligence*. 2nd ed. New York, NY: McGraw-Hill, 1991.
- Schwarz, H. R., *Finite Element Methods*. San Diego, CA: Academic Press, 1988.

12 Conclusions

IN THIS CHAPTER

- Multiprocessor Architectures
- Cluster Configuration
- Distributed Applications
- Final Comments

You've learned a lot about cluster hardware. In Chapter 1, "Linux Cluster Computer Fundamentals," you learned the difference between a network of PCs running Linux and a cluster computer. You discovered the difference between a loosely coupled multiprocessor and a tightly coupled one in Chapter 2, "Multiprocessor Architecture."

You've learned even more about cluster software. You have a feel for how complex or how general a subtask has to be before you make it an independent module, using `fork` and `execl` in the parent instead of just a `fork` call. You know at least three ways to pass information between any two processes and know why some of them don't work when the processes are running on different nodes of the network. Finally, from Chapter 3, "Inter-process Communication," you understand the differences between UDP and TCP and know when to choose one over the other.

That's the groundwork[md]the basic knowledge you need to develop a distributed system. Only a small percentage of software developers understand this, and you should feel good about being one of them.

Next comes the hands-on experience. If you actually built and configured a network during the course of reading this book, you have something else to brag about: You can look at the back of a PC and tell if it has a network interface and what kind of cabling it requires. You can go into a large computer store, look at hubs, switches, routers, and firewalls and choose what you need with confidence. You can install a Linux OS, log in as root, and administer that system to work along with all the others. You can perform some rudimentary tests on this new addition and be sure that it works as planned. You can create a nonroot user and customize its home directory. You can create C programs that are all part of a project and a makefile that ties them together. You can measure the external performance of a transaction server and time the execution phases of any kind of system. You can tune that system and test it again to see if your tuning changes had any benefit. You have at least a few ideas about fault-tolerant software and can add code to avoid problems that might cause your system to fail.

You have an overview of what others have done with their clusters, and you know the difference between transaction processors and supercomputers. You understand the difference between real-time and simulated environmental data and how a supercomputer can transform such data into a display of the environment it represents. You understand how the distribution of inter-arrival times can affect the performance of a server. This is a broad based but critical level of understanding. Solutions applicable to one kind of system may have a detrimental effect on another.

Finally, you can see at least a little bit into the future. You know, as most others do, that computers will continue to become faster, cheaper, and smaller. You know, too, what many do not[md]that there is a trend away from hierarchical, client/server constructs toward the more flexible, more complex, more general peer-to-peer networks; the move toward ubiquitous computing can provide anyone with useful information about anything and can let anyone communicate with anyone else, provided the infrastructure is working properly. Most important, you understand that an infrastructure necessary to support such lofty goals begins with a bunch of computers on a network, working together toward that common goal.

Multiprocessor Architectures

Each time an algorithm can do two or more things in parallel, a multiprocessor can improve its performance. If the same actions are performed in parallel but on different data, the multiprocessor can be a vector processor. If not, a more general multiprocessor is indicated.

If the execution time necessary to process data in parallel is much less than the time to acquire that data, a tightly coupled multiprocessor is required. If not, a multicomputer will suffice. A multicomputer is harder to program than a tightly coupled multiprocessor, but it is a lot less expensive and cheaper to maintain and easier to upgrade. Remember that a multicomputer that performs half as well at a tenth the price is still quite a bargain, assuming half the performance suits your needs.

Cluster Configuration

You can get on a mail-order Web site or walk into the nearest large computer store and buy everything you will ever need to build a cluster computer. The cost, however, can be kept to an absolute minimum by getting the computers from a salvage center, finding them in flea markets or garage sales, or getting them for free from co-workers, friends, or neighbors. One of the test bed nodes came free from a co-worker who responded to an ad in a newsletter.

Configuring the OS files, creating a home directory for the software development work, and actually writing, testing, and debugging distributed software takes quite a bit of skill, but this is a learned skill, developed over time. You'll make mistakes, as we all do, insert some print statements, troubleshoot the problem, and solve it. That's where such skills really come from; none of us is born with them. Getting your hands dirty can be rewarding as you learn and enjoyable when the system starts to work.

Distributed Applications

You may be a manager reading this book to find out what kind of problems your people are facing on some new product. You may be a seasoned developer wanting to shed some light on some of the gray areas surrounding such systems. You may be a hobbyist wanting to stay near the edge of systems development. If you actually build such a system, as I hope you will, the choice of application can become a complete second half of the journey. Digital signal processing (noise reduction, filtering, and so on) can apply equally to CCD images from a telescope or to audio signals from a shortwave radio. Maybe the sample system, a data server, is just what you had in mind. The choice is yours, and all you need to do is change the application software to convert a supercomputer into a transaction server, and back again.

Final Comments

A few years ago, this author was seduced by the macho appeal of the dual-processor PC systems being offered as personal workstations by a few manufacturers. A short conversation revealed that a dual 100MHz PC cost twice as much as a single one and wouldn't quite match the speed of the single 200MHz PC that would be announced in a few months. This fact cleared up the thinking process, but the allure of a multiprocessor persisted.

There came a time when that old '386 processor just wouldn't run the latest version of the OS that was required to run the latest version of the word processor that was in use by everyone now, at work. If you wanted to work late, you would have to stay late or buy a faster PC. A year later, two dated but adequate PCs sat in a closet until several others just like them became available for a very small amount of money. Another year later saw an eight-node Linux cluster come to life.

This book would have made the process a whole lot easier. Here's to hoping you'll find your way to that point and take it at least one step further. Enjoy the journey!

APPENDIX A

The Source Code

IN THIS APPENDIX

- Query-Generating Client
- Master-Slave Interface
- Near Real-time Performance
- Makefile

Here are the source code listings for the query-generating client, the entire master-slave interface, including the near real-time performance display modules, their header files, and a simple `make` utility file. The C files should be located under /home/chief/src and the header files should be located under /home/chief/inc, as described in Chapter 6, "Configuring a User Environment for Software Development." Remember that you can download all these files from the publisher's companion Web site, `www.samspublishing.com`. Type in this book's ISBN number and select the Downloads link for further information.

Figure A.1 shows a diagram of the hardware we use for our test bed and where each of these processes runs.

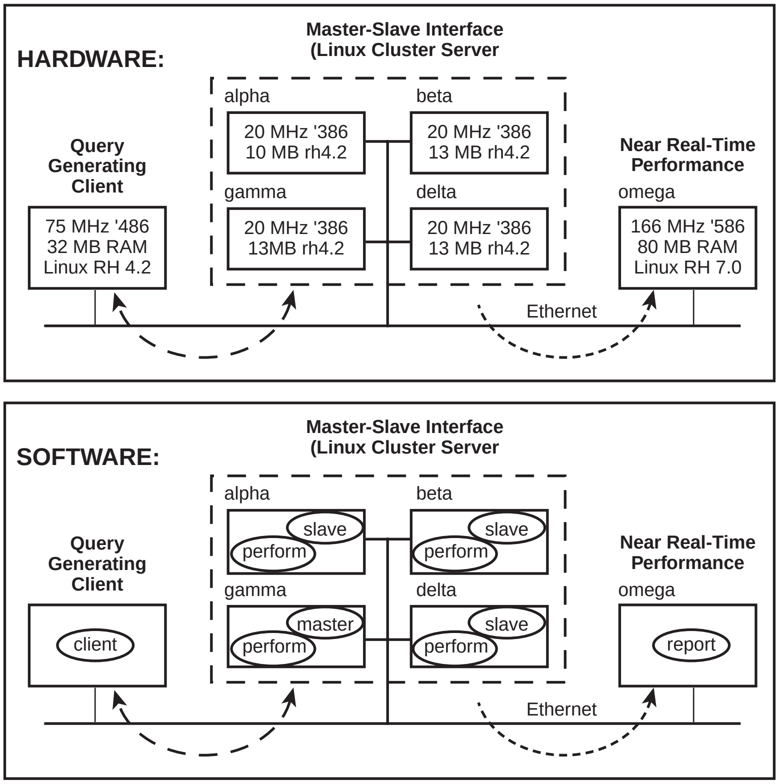

FIGURE A.1 The client, cluster, and performance display.

Query-Generating Client

The client runs on a machine that has access to the cluster network but that is not itself a cluster node. Following is a sample line command, assuming the server is already running.

```
> client alpha p
```

This assumes that the master task was started on alpha and requests a stream of queries in a pulse distribution (see Chapter 8, “External Performance Measurement and Analysis”).

LISTING A.1 The Client Process

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "msi.h"

#define FALSE 0
#define TRUE 1
#define NUM 50

#define CATS 10 /* # of response time categories */
double cats[CATS] = {.01, .02, .03, .04, .05,
                     .06, .07, .08, .09, .10};
int bars[CATS] = {0}; /* counters for bar chart */

main(int argc, char *argv[])
{
   /*
   **   client.c: reports response time stats
   */
   struct timeval bef, aft;   /* timer value before/after */
   struct sockaddr_in sock;   /* structure for socket */
   struct hostent *hp;        /* structure for IP address */
   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = NUM * sizeof(double);
   key_t key = 0x01234567;    /* example shared memory key */
   int shmid;                 /* shared memory area id */
   int Expon = FALSE;         /* boolean: exponential dist */
   int Pulse = FALSE;         /* boolean: pulse distribution */
   int Sweep = FALSE;         /* boolean: sweep distribution */
   double mean=0.50;          /* mean value for exponential */
   double slow=0.25;          /* slow value for pulse & sweep */
   double fast=0.10;          /* fast value for pulse burst */
```

LISTING A.1 Continued

```
   double random;             /* random reals: 0.0->1.0 */
   double delay;              /* calculated delay time */
   double *rt;                /* response times from subtasks */
   long timeleft;             /* time remaining in IAT */
   long usec=0L;              /* inter-arrival microseconds */
   long secs=0L;              /* time remaining in seconds */
   int pids[NUM];             /* subtask process ids */
   int opt=1;                 /* setsockopt parameter */
   int fd;                    /* socket file descriptor */
   int ii, jj, kk=0;
   char distribution[4];
   char buf[QUERYSIZE], *p;

   srand((unsigned int) getpid()); /* seed rand() */

   /*
   **   Operands are remote HOST name and distribution code
   */
   if(argc != 3)
   {
      printf("\n\tUsage: %s <HOST> <DISTRIBUTION>\n\n", argv[0]);
      exit(-1);
   }
   hp = gethostbyname(argv[1]);

   if((argv[2][0] == 'e') || (argv[2][0] == 'E'))
   {
      Expon = TRUE;
      strcpy(distribution, "EXP");
      srand((unsigned int) getpid()); /* rand() seed */
   }
   else if((argv[2][0] == 'p') || (argv[2][0] == 'P'))
   {
      Pulse = TRUE;
      strcpy(distribution, "PLS");
   }
   else if((argv[2][0] == 's') || (argv[2][0] == 'S'))
   {
      Sweep = TRUE;
      strcpy(distribution, "SWP");
   }
```

LISTING A.1 Continued

```
else
{
   printf("\n\tUsage: %s <HOST> <DISTRIBUTION>\n", argv[0]);
   printf("\t       DISTRIBUTION = { 'e' | 'p' | 's' }\n");
   printf("\t       for exponential, pulse and sweep\n\n");
   exit(-1);
}

/* attach shared memory array */
shmid = shmget(key, size, flag);
/* attach the shared memory area */
rt = (double *) shmat(shmid, 0, 0);

/*
**   LOOP: send and receive NUM packets
*/
for(ii=0; ii<NUM; ii++)
{
   rt[ii]= 0.0; /* zero shared array values */

   if(Expon)
   {
      random = rand() / (double) RAND_MAX;
      delay = -mean * log(random);
      if(delay > 0.999999)
         delay = 0.999999; /* limit maximum */
   }
   else if(Pulse)
   {
      if((ii > (NUM * 0.3)) && (ii < (NUM * 0.4)))
         delay = fast;
      else
         delay = slow;
   }
   else if(Sweep)
   {
      delay = slow - (ii * (slow / NUM));
   }
   else
   {
      printf("\n\tError in logic?\n\n");
```

LISTING A.1 Continued

```
            exit(-1);
        }
        secs = (long) delay;
        usec = (long) (1000000.0 * (delay - ((double) secs)));

        /* random line numbers: 1 thru 99 */
        random = rand() / (double) RAND_MAX;
        jj = (int) ((double) (99.0) * random) + 1;
        if(jj == 20)
            jj = 99;
        sprintf(buf, "/home/chief/sample.txt %d", jj);

        /* record sending time */
        gettimeofday(&bef, NULL);

        /*
        **   A subtask for each query/response
        */
        if((pids[kk++]=fork()) == 0)
        {
            /* attach parent's shared memory */
            rt = (double *) shmat(shmid, 0, 0);
            /* shmid is still set correctly */

            /* Set up TCP socket to the server */
            fd = socket(AF_INET, SOCK_STREAM, 0);
            memset((char *) &sock, 0, sizeof(sock));
            memcpy(&sock.sin_addr, hp->h_addr, hp->h_length);
            sock.sin_family = hp->h_addrtype;
            sock.sin_port = htons(QUERYPORT);
            setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
                (char *) &opt, sizeof(opt));
            connect(fd, (struct sockaddr *) &sock, sizeof(sock));
            send(fd, buf, strlen(buf)+1, 0);

            /* await result from server */
            buf[0] = 0; /* clear buffer */
            if(recv(fd, buf, QUERYSIZE, 0) > 0)
            {
/*              printf("\t%d. Line %2d: %s ", kk, jj, buf);   */
/*              p = strrchr(buf, ' ') + 2;   */
```

LISTING A.1 Continued

```
/*          if(jj != atoi(p))   */
/*             printf("***"); /* incorrect result! */
/*          printf("\n");   */
         }

         /* record response time */
         gettimeofday(&aft, NULL);
         aft.tv_sec -= bef.tv_sec;
         aft.tv_usec -= bef.tv_usec;
         if(aft.tv_usec < 0L)
         {
            aft.tv_usec += 1000000L;
            aft.tv_sec -= 1;
         }
         rt[ii] = (double) aft.tv_sec + ((double) aft.tv_usec / 1000000.0);

         close(fd);
         exit(0);
      }

      /*
      **   Sleep for remainder of IAT
      */
      gettimeofday(&aft, NULL);
      aft.tv_sec -= bef.tv_sec;
      aft.tv_usec -= bef.tv_usec;
      if(aft.tv_usec < 0L)
      {
         aft.tv_usec += 1000000L;
         aft.tv_sec -= 1;
      }
      bef.tv_sec = secs;
      bef.tv_usec = usec;
      bef.tv_sec -= aft.tv_sec;
      bef.tv_usec -= aft.tv_usec;
      if(bef.tv_usec < 0L)
      {
         bef.tv_usec += 1000000L;
         bef.tv_sec -= 1;
      }
      timeleft = (bef.tv_sec * 1000000L ) + bef.tv_usec;
```

LISTING A.1 Continued

```
      if(timeleft < 0)
      {
         printf("\tERROR: A higher IAT value is required - exiting.\n");
         break;
      }
      usleep(timeleft);
   }
   for(ii=0; ii<kk; ii++)
      wait(pids[ii]);

   /*
   **   Report the response time statistics
   */
   for(ii=0; ii<NUM; ii++)
   {
      for(jj=0; jj<CATS; jj++)
      {
         if(rt[ii] < cats[jj])
         {
            bars[jj]++;
            break;
         }
      }
   }
   sleep(2);
   printf("RESPONSE    |              %3d QUERY PACKETS                %s\n",
      NUM, distribution);
   printf("TIME (msec) |        10        20        30        40        50\n");
   printf("------------+----+----+----+----+----+----+----+----+----+----+\n");
   ii = 0; /* total */
   for(jj=0; jj<CATS; jj++)
   {
      ii += bars[jj];
      kk = (1000.0 * cats[jj]) + .5;
      printf(" %5d%5d |", kk-9, kk);
      for(kk=0; kk<bars[jj]; kk++)
         printf("*");
      printf("\n");
   }
   printf("  OVER%5d |", (int) (1000 * cats[CATS-1]));
   jj = NUM - ii;
```

LISTING A.1 Continued

```
   for(kk=0; kk<jj; kk++)
      printf("*");
   printf("\n");

   /* remove unused shared memory area and exit */
   shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
   return(0);
}
```

Master-Slave Interface

This is the distributed server module pair. The master process may be started on any of the cluster nodes. It will automatically start a slave process on each of the other nodes. Both master and slave start performance subtasks on their respective nodes, targeted toward the display node, named *omega* in the source listings (see Figure 7.2).

Following is a line command used to start the master task.

```
> master
```

Note that there are no line command operands. The Ctrl+C keystroke ends it gracefully, along with its subtasks, all the remote slave tasks, and their subtasks.

LISTING A.2 The Master Process

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <netdb.h>
#include <stdio.h>
#include "msi.h"

#define MAX 16 /* maximum remote slaves supported */
#define LEN 80 /* maximum hostname length allowed */

#define FALSE 0
```

LISTING A.2 Continued

```
#define TRUE 1

/* Boolean flags */
int Running = TRUE;
int Alarm = FALSE;

unsigned char bcaddr[4] = {0x0A, 0x00, 0x00, 0xFF}; /* 10.0.0.255 */

main(void)
/*
**   master.c - Master feeds queries to Slave services
*/
{
   int len = sizeof(struct sockaddr_in);
   struct sockaddr_in broadcast;
   struct sockaddr_in registers;
   struct sockaddr_in querylink;
   struct sockaddr_in slavelink;
   struct hostent *hp;
#ifdef TIMING
 struct timeval workval, phase1[MAX]; /* phase 1 happens in Parent */
 struct timeval ph01wrk;    /* choose an IDLE subtask from the pool */
 struct timeval ph02wrk;    /* subtask sends incoming query to remote */
 struct timeval ph03wrk;    /* calculated Master-to-Slave transit time */
 struct timeval ph04wrk;    /* Slave gets query and extracts request */
 struct timeval ph05wrk;    /* Slave retrieves line of text from file */
 struct timeval ph06wrk;    /* Slave builds response and sends it back */
 struct timeval ph07wrk;    /* calculated Slave-to-Master transit time */
 struct timeval ph08wrk;    /* subtask responds to query and sets IDLE */
 double tot, avg;
 int pkts=0;                /* incoming packet counter */
 FILE *tfp;
#endif
   void sig_inter();
   void sig_alarm();

   /*
   **   Shared Memory Space
   */
   struct area {            /* flag for every subtask */
      int flag[MAX];        /* 0:idle, 1:good, -:exit */
```

LISTING A.2 Continued

```
#ifdef TIMING
 int count[MAX];            /* number of queries handled */
 struct timeval phase2[MAX];
 struct timeval phase3[MAX];
 struct timeval phase4[MAX];
 struct timeval phase5[MAX];
 struct timeval phase6[MAX];
 struct timeval phase7[MAX];
 struct timeval phase8[MAX];
#endif
   } area, *aptr;

   int flag = (IPC_CREAT | IPC_EXCL | 0660);
   int size = sizeof(struct area);
   key_t key = 0x01234567;  /* shared memory area key */
   int shmid;               /* shared memory area id */
   int acc;                 /* accept file descriptor */
   int cli;                 /* query file descriptor */
   int pid;                 /* performance monitor */
   int b_fd;                /* broadcast file desc */
   int r_fd;                /* registration file desc */
   int s_fd;                /* slave link file desc */
   int ident;               /* subtask id and offset */
   int opt=1;
   fd_set fds;              /* select list for queries */
   int count=0;             /* registered slave count */
   int pids[MAX];           /* subtask process idents */
   char hostname[LEN];      /* host where Master runs */
   char hosts[MAX][LEN];    /* registered slave hosts */
   char buf[QUERYSIZE], *p; /* client query/response */

   /* init signal handlers */
   signal(SIGINT, sig_inter);
   signal(SIGALRM, sig_alarm);

   /* establish a UDP broadcast socket */
   b_fd = socket(AF_INET, SOCK_DGRAM, 0);
   memset((char *) &broadcast, 0, len);
   broadcast.sin_family = AF_INET;
   broadcast.sin_port = htons(BROADCAST);
   memcpy(&broadcast.sin_addr, bcaddr, 4);
```

LISTING A.2 Continued

```
setsockopt(b_fd, SOL_SOCKET, SO_BROADCAST,
   (char *) &opt, sizeof(opt));

/* establish a registrations socket */
r_fd = socket(AF_INET, SOCK_DGRAM, 0);
memset((char *) &registers, 0, len);
registers.sin_addr.s_addr = htonl(INADDR_ANY);
registers.sin_port = htons(REGISTERS);
registers.sin_family = AF_INET;
bind(r_fd, (struct sockaddr *) &registers, len);

/* start up all remote slave services */
gethostname(hostname, sizeof(hostname));
if((p=strchr(hostname, '.')) != NULL)
   *p = '\0'; /* trunc .domainname */
sendto(b_fd, hostname, strlen(hostname)+1, 0,
   (struct sockaddr *) &broadcast, len);
close(b_fd);

/*
**   Display registration response info
*/
while(1) /* alarm signal causes exit */
{
   alarm(1); /* set a 1-second alarm */
   buf[0] = '\0'; /* clear input buf */
   /* wait for registration or alarm */
   recvfrom(r_fd, buf, sizeof(buf), 0,
     (struct sockaddr *) &registers, &len);
   if(Alarm)
      break;
   alarm(0); /* cancel alarm timer */
   if((p=strchr(buf, '.')) != NULL)
      *p = '\0'; /* trunc .domain */
   printf("\tSlave on host '%s' registered\n", buf);
   strcpy(hosts[count++], buf);
}
close(r_fd);
printf("\tRegistered %d remote slave task(s)", count);

/*
```

LISTING A.2 Continued

```
**   ERROR if unsupported #slaves
*/
if((count <= 0) || (count > MAX))
{
   printf(" - Unsupported!\n");
   exit(-1);
}
printf(".\n\n");

/* start "perform" */
if((pid=fork()) == 0)
{
   execl("/home/chief/bin/perform", "perform", "cardinal", NULL);
   exit(-1); /* a successful execl never returns to the caller */
}

/*
**   Shared memory area for subtask pool
*/
size *= count;
shmid = shmget(key, size, flag);
aptr = (struct area *) shmat(shmid, 0, 0);

/*
**   TCP socket for client's queries
*/
acc = socket(AF_INET, SOCK_STREAM, 0);
querylink.sin_family = AF_INET;
querylink.sin_port = htons(QUERYPORT);
querylink.sin_addr.s_addr = htonl(INADDR_ANY);
bind(acc, (struct sockaddr *) &querylink, len);
setsockopt(acc, SOL_SOCKET, SO_REUSEADDR,
   (char *) &opt, sizeof(opt));
listen(acc, 5);

/*
**   Pre-establish a subtask pool
*/
for(ident=0; ident<count; ident++)
{
   aptr->flag[ident] = 0; /* idle */
```

LISTING A.2 Continued

```
#ifdef TIMING
 aptr->count[ident] = 0; /* zero cumulative statistics */
 phase1[ident].tv_sec = 0L;
 phase1[ident].tv_usec = 0L;
 aptr->phase2[ident].tv_sec = 0L;
 aptr->phase2[ident].tv_usec = 0L;
 aptr->phase3[ident].tv_sec = 0L;
 aptr->phase3[ident].tv_usec = 0L;
 aptr->phase4[ident].tv_sec = 0L;
 aptr->phase4[ident].tv_usec = 0L;
 aptr->phase5[ident].tv_sec = 0L;
 aptr->phase5[ident].tv_usec = 0L;
 aptr->phase6[ident].tv_sec = 0L;
 aptr->phase6[ident].tv_usec = 0L;
 aptr->phase7[ident].tv_sec = 0L;
 aptr->phase7[ident].tv_usec = 0L;
 aptr->phase8[ident].tv_sec = 0L;
 aptr->phase8[ident].tv_usec = 0L;
#endif

      if((pids[ident]=fork()) == 0)                   /* SUBTASK */
      {
         /* nullify ^C handler */
         signal(SIGINT, SIG_IGN);

         /* attach the shared memory flags array */
         aptr = (struct area *) shmat(shmid, 0, 0);
         /* notice: shmid is still set correctly */

         /*
         **   Establish slave comm link socket
         */
         s_fd = socket(AF_INET, SOCK_DGRAM, 0);
         memset((char *) &slavelink, 0, len);
         hp = gethostbyname(hosts[ident]);
         memcpy(&slavelink.sin_addr, hp->h_addr, hp->h_length);
         slavelink.sin_port = htons(SLAVEPORT);
         slavelink.sin_family = AF_INET;
         bind(s_fd, (struct sockaddr *) &slavelink, len);

         /*
```

LISTING A.2 Continued

```
         **   Poll on our flag until it becomes non-zero
         */
         while(1)
         {
            if(aptr->flag[ident] == 0) /* do we stay idle? */
            {
               usleep(1);   /* release processor */
               continue;    /* stay in poll loop */
            }
            else if(aptr->flag[ident] < 0) /* exit request? */
            {
               /* signal remote slave to exit */
               strcpy(packet.buffer, "Terminate 0");
               sendto(s_fd, &packet, sizeof(packet),
                  0, (struct sockaddr *) &slavelink, len);
               break; /* SUBTASKs normal exit */
            }
            else /* action flag must be positive */
            {
               cli = accept(acc, (struct sockaddr *) &querylink, &len);
#ifdef TIMING
 gettimeofday(&ph02wrk, NULL); /* start Ph2 */
#endif
               /* recv incoming query from client */
               packet.buffer[0] = 0; /* clear buf */
               recv(cli, packet.buffer, QUERYSIZE, 0);

               /* give packet to slave task for processing */
#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph2 */
 workval.tv_sec -= ph02wrk.tv_sec;
 workval.tv_usec -= ph02wrk.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 aptr->phase2[ident].tv_sec += workval.tv_sec;
 aptr->phase2[ident].tv_usec += workval.tv_usec;
 /* start transmission time */
 gettimeofday(&ph03wrk, NULL); /* start Ph3 through Ph7 */
```

LISTING A.2 Continued

```
#endif
               sendto(s_fd, &packet, sizeof(struct packet),
                  0, (struct sockaddr *) &slavelink, len);
               packet.buffer[0] = 0; /* empty the buffer */
               recvfrom(s_fd, &packet, sizeof(struct packet),
                  0, (struct sockaddr *) &slavelink, &len);
#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph3 through Ph7 */
 /* calulate send+slave+recv time */
 workval.tv_sec -= ph03wrk.tv_sec;
 workval.tv_usec -= ph03wrk.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 /* subtract off slave processing time */
 workval.tv_sec -= packet.remote.tv_sec;
 workval.tv_usec -= packet.remote.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 /* divide round-trip time by two */
 ph03wrk.tv_sec = workval.tv_sec / 2;
 if((ph03wrk.tv_usec * 2) < workval.tv_sec)
    workval.tv_usec += 500000L;
 aptr->phase3[ident].tv_sec += workval.tv_sec / 2;
 aptr->phase3[ident].tv_usec += workval.tv_usec / 2;
 /* put the other half into Ph7 */
 aptr->phase7[ident].tv_sec += workval.tv_sec / 2;
 aptr->phase7[ident].tv_usec += workval.tv_usec / 2;
 /* capture Slave's timing statistics from packet */
 aptr->phase4[ident].tv_sec += packet.phase4.tv_sec;
 aptr->phase4[ident].tv_usec += packet.phase4.tv_usec;
 aptr->phase5[ident].tv_sec += packet.phase5.tv_sec;
 aptr->phase5[ident].tv_usec += packet.phase5.tv_usec;
 aptr->phase6[ident].tv_sec += packet.phase6.tv_sec;
 aptr->phase6[ident].tv_usec += packet.phase6.tv_usec;
 gettimeofday(&ph08wrk, NULL); /* start Ph8 */
```

LISTING A.2 Continued

```
#endif
               /* result back to client */
               strcpy(buf, packet.buffer);
               send(cli, buf, strlen(buf)+1, 0);
               close(cli);

               /* check for exit request before continuing */
               if(aptr->flag[ident] < 0) /* exit request? */
               {
                  sendto(s_fd, "Terminate 0", 12, 0,
                     (struct sockaddr *) &slavelink, len);
                  break;
               }
               aptr->flag[ident] = 0; /* no: flag as idle */
#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph8 */
 workval.tv_sec -= ph08wrk.tv_sec;
 workval.tv_usec -= ph08wrk.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 aptr->phase8[ident].tv_sec += workval.tv_sec;
 aptr->phase8[ident].tv_usec += workval.tv_usec;

 aptr->count[ident]++; /* increment counter */
#endif
            }
         }
         close(s_fd);
         exit(0);                                 /* end SUBTASK */
      }
   } /* for count */

   /*
   **   Pass incoming queries to an available subtask
   */
   ident = 0;
   while(Running)                               /* start PARENT */
   {
```

LISTING A.2 Continued

```
      FD_ZERO(&fds);
      FD_SET(acc, &fds);
      /* block until client is ready for connect */
      if(select(acc+1, &fds, NULL, NULL, NULL) < 0)
         break;
      /*
      **   Find an idle subtask to process a query
      */
#ifdef TIMING
 gettimeofday(&ph01wrk, NULL); /* start Ph1 */
#endif
      for(ident=0; ident<count; ident++)
      {
         if(aptr->flag[ident] == 0) /* available? */
         {
            aptr->flag[ident] = 1; /* work to do! */
            break;
         }
      }
      if(ident == count) /* no available subtasks */
      {
         cli = accept(acc, (struct sockaddr *) &querylink, &len);
         send(cli, "DROPPED", 8, 0);
         close(cli);
      }
#ifdef TIMING
 /* Ph1 is the only Parent processing time */
 gettimeofday(&workval, NULL); /* end Ph1 */
 workval.tv_sec -= ph01wrk.tv_sec;
 workval.tv_usec -= ph01wrk.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 if(ident < count) /* not DROPPED queries */
 {
    phase1[ident].tv_sec += workval.tv_sec;
    phase1[ident].tv_usec += workval.tv_usec;
 }
#endif
```

LISTING A.2 Continued

```
      usleep(1);
   } /* while Running */

   /*
   **   Tell each subtask to exit gracefully
   */
   for(ident=0; ident<count; ident++)
   {
      aptr->flag[ident] = -1;
      waitpid(pids[ident]);
   }
   kill(pid, 9);
   waitpid(pid);

#ifdef TIMING
   tfp = fopen("master.txt", "w");
   /* print final timing statistics */
   fprintf(tfp, "\n\tSLAVENAME   PH   NUM       TOTAL     AVERAGE\n");
   fprintf(tfp, "\t---------   --   ---       -----     -------\n");
   ph01wrk.tv_sec = 0L;
   ph01wrk.tv_usec = 0L;
   ph02wrk.tv_sec = 0L;
   ph02wrk.tv_usec = 0L;
   ph03wrk.tv_sec = 0L;
   ph03wrk.tv_usec = 0L;
   ph04wrk.tv_sec = 0L;
   ph04wrk.tv_usec = 0L;
   ph05wrk.tv_sec = 0L;
   ph05wrk.tv_usec = 0L;
   ph05wrk.tv_sec = 0L;
   ph05wrk.tv_usec = 0L;
   ph06wrk.tv_sec = 0L;
   ph06wrk.tv_usec = 0L;
   ph07wrk.tv_sec = 0L;
   ph07wrk.tv_usec = 0L;
   ph08wrk.tv_sec = 0L;
   ph08wrk.tv_usec = 0L;
   for(ident=0; ident<count; ident++)
   {
      pkts += aptr->count[ident];
      tot = ((double) phase1[ident].tv_sec +
```

LISTING A.2 Continued

```
          ((double) phase1[ident].tv_usec / 1000000.0));
    ph01wrk.tv_sec += phase1[ident].tv_sec;
    ph01wrk.tv_usec += phase1[ident].tv_usec;
    if(aptr->count[ident] > 0)
       avg = tot / (double) aptr->count[ident];
    else
       avg = 0.0;
    fprintf(tfp, "\t %8s  %3d     1  %10.6lf  %10.6lf\n", hosts[ident],
           aptr->count[ident], tot, avg);
    tot = ((double) aptr->phase2[ident].tv_sec +
          ((double) aptr->phase2[ident].tv_usec / 1000000.0));
    ph02wrk.tv_sec += aptr->phase2[ident].tv_sec;
    ph02wrk.tv_usec += aptr->phase2[ident].tv_usec;
    if(aptr->count[ident] > 0)
       avg = tot / (double) aptr->count[ident];
    else
       avg = 0.0;
    fprintf(tfp, "\t                   2  %10.6lf  %10.6lf\n", tot, avg);
    tot = ((double) aptr->phase3[ident].tv_sec +
          ((double) aptr->phase3[ident].tv_usec / 1000000.0));
    ph03wrk.tv_sec += aptr->phase3[ident].tv_sec;
    ph03wrk.tv_usec += aptr->phase3[ident].tv_usec;
    if(aptr->count[ident] > 0)
       avg = tot / (double) aptr->count[ident];
    else
       avg = 0.0;
    fprintf(tfp, "\t                   3  %10.6lf  %10.6lf\n", tot, avg);
    tot = ((double) aptr->phase4[ident].tv_sec +
          ((double) aptr->phase4[ident].tv_usec / 1000000.0));
    ph04wrk.tv_sec += aptr->phase4[ident].tv_sec;
    ph04wrk.tv_usec += aptr->phase4[ident].tv_usec;
    if(aptr->count[ident] > 0)
       avg = tot / (double) aptr->count[ident];
    else
       avg = 0.0;
    fprintf(tfp, "\t                   4  %10.6lf  %10.6lf\n", tot, avg);
    tot = ((double) aptr->phase5[ident].tv_sec +
          ((double) aptr->phase5[ident].tv_usec / 1000000.0));
    ph05wrk.tv_sec += aptr->phase5[ident].tv_sec;
    ph05wrk.tv_usec += aptr->phase5[ident].tv_usec;
    if(aptr->count[ident] > 0)
```

LISTING A.2 Continued

```
         avg = tot / (double) aptr->count[ident];
      else
         avg = 0.0;
      fprintf(tfp, "\t                   5  %10.6lf  %10.6lf\n", tot, avg);
      tot = ((double) aptr->phase6[ident].tv_sec +
            ((double) aptr->phase6[ident].tv_usec / 1000000.0));
      ph06wrk.tv_sec += aptr->phase6[ident].tv_sec;
      ph06wrk.tv_usec += aptr->phase6[ident].tv_usec;
      if(aptr->count[ident] > 0)
         avg = tot / (double) aptr->count[ident];
      else
         avg = 0.0;
      fprintf(tfp, "\t                   6  %10.6lf  %10.6lf\n", tot, avg);
      tot = ((double) aptr->phase7[ident].tv_sec +
            ((double) aptr->phase7[ident].tv_usec / 1000000.0));
      ph07wrk.tv_sec += aptr->phase7[ident].tv_sec;
      ph07wrk.tv_usec += aptr->phase7[ident].tv_usec;
      if(aptr->count[ident] > 0)
         avg = tot / (double) aptr->count[ident];
      else
         avg = 0.0;
      fprintf(tfp, "\t                   7  %10.6lf  %10.6lf\n", tot, avg);
      tot = ((double) aptr->phase8[ident].tv_sec +
            ((double) aptr->phase8[ident].tv_usec / 1000000.0));
      ph08wrk.tv_sec += aptr->phase8[ident].tv_sec;
      ph08wrk.tv_usec += aptr->phase8[ident].tv_usec;
      if(aptr->count[ident] > 0)
         avg = tot / (double) aptr->count[ident];
      else
         avg = 0.0;
      fprintf(tfp, "\t                   8  %10.6lf  %10.6lf\n", tot, avg);
   }
   tot = ((double) ph01wrk.tv_sec +
         ((double) ph01wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t  OVERALL    1   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph02wrk.tv_sec +
         ((double) ph02wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             2   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph03wrk.tv_sec +
```

LISTING A.2 Continued

```
        ((double) ph03wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             3   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph04wrk.tv_sec +
        ((double) ph04wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             4   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph05wrk.tv_sec +
        ((double) ph05wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             5   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph06wrk.tv_sec +
        ((double) ph06wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             6   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph07wrk.tv_sec +
        ((double) ph07wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             7   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   tot = ((double) ph08wrk.tv_sec +
        ((double) ph08wrk.tv_usec / 1000000.0));
   avg = tot / (double) pkts;
   fprintf(tfp, "\t             8   %3d  %10.6lf  %10.6lf\n", pkts, tot, avg);
   fclose(tfp);
#endif

   /* remove unused shared memory area and exit */
   shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);
   printf("\tDone.\n");
   return(0);
}                                                   /* end PARENT */

/**************/
/*** alarms ***/
/**************/
void sig_alarm()
{
   /* reestablish handler */
   signal(SIGALRM, sig_alarm);
   Alarm = TRUE; /* set flag */
```

LISTING A.2 Continued

```
}

/**************/
/*** ^C key ***/
/**************/
void sig_inter()
{
   Running = FALSE; /* set flag */
   return;
}
```

There is no line command to start the slave task because it is an inetd service (see Chapter 3, "Inter-process Communication"). The master task starts each remote slave task by broadcasting a UDP datagram to its service port.

LISTING A.3 The Slave Process

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include "msi.h"

main(void)
/*
**   slave.c - inetd service activated by a UDP datagram to port 5000
**
**   inetd configuration (as root - man inetd, or info inetd):
**     1. /etc/services:   slaves 5000/udp     # Master-Slave Interface
**     2. /etc/inetd.conf: slaves dgram udp wait chief /home/chief/slave
**     3. Reconfiguration: killall -HUP inetd
**
**   xinetd configuration (as root - man xinetd.conf, or info xinetd):
**     1. Install xinetd package if necessary
**     2. /etc/xinetd.d/rcatd:
**           service rcatd
**           {
```

LISTING A.3 Continued

```
**              port        = 5000
**              socket_type = dgram
**              protocol    = udp
**              wait        = yes
**              user        = chief
**              server      = /home/chief/bin/slave
**              only_from   = 10.0.0.0
**              disable     = no
**           }
**     3. Reconfiguration: /etc/rc.d/init.d/xinetd restart
**
**   In either case:
**     4. Start the Master process first, on any of the cluster's nodes
**        and it will send a broadcast UDP datagram to port 5000, starting
**        a Slave process on every node other than where the Master runs.
*/
{
   int len=sizeof(struct sockaddr_in);
   struct sockaddr_in registers;
   struct sockaddr_in slavelink;
   struct hostent *hp;
#ifdef TIMING
 struct timeval workval;
#endif
   FILE *fp;                      /* query text file pointer */
   int pid;                       /* performance task's pid */
   int line;                      /* requested line number */
   int r_fd;
   int s_fd;
   int opt=1;
   int curr=999;                  /* init current line number */
   char file[80];
   char host[32], *p;
   char buf[QUERYSIZE];

#ifdef TIMING
 /* start remote processing timer */
 gettimeofday(&packet.remote, NULL);
#endif

   /*
```

LISTING A.3 Continued

```
**   Get MASTER's hostname
*/
read(0, buf, sizeof(buf));
sscanf(buf, "%s", host);

/*
**   Register with the MASTER task
*/
r_fd = socket(AF_INET, SOCK_DGRAM, 0);
memset((char *) &registers, 0, len);
hp = gethostbyname(host); /* get MASTER IP Address */
memcpy(&registers.sin_addr, hp->h_addr, hp->h_length);
registers.sin_port = htons(REGISTERS);
registers.sin_family = AF_INET;
/* check the local hostname */
gethostname(buf, sizeof(buf));
if((p = strchr(buf, '.')) != NULL)
   *p = 0; /* trunc .domainname */
if(strcmp(host, buf) == 0)
   exit(0); /* no local slaves! */
sendto(r_fd, buf, strlen(buf)+1, 0,
  (struct sockaddr *) &registers, len);
close(r_fd);

/* start "perform" */
if((pid=fork()) == 0)
{
   execl("/home/chief/bin/perform", "perform", "cardinal", NULL);
   exit(-1); /* a successful execl never returns to the caller */
}

/*
**   Setup a socket to receive queries
**   from one of the MASTER's subtasks
*/
s_fd = socket(AF_INET, SOCK_DGRAM, 0);
memset((char *) &slavelink, 0, len);
slavelink.sin_family = AF_INET;
slavelink.sin_port = htons(SLAVEPORT);
slavelink.sin_addr.s_addr = htonl(INADDR_ANY);
```

LISTING A.3 Continued

```
   bind(s_fd, (struct sockaddr *) &slavelink, len);

   /*
   **   Respond to each query with
   **   the requested line of text
   */
   while(1)
   {
      /* process all incoming queries from Master */
      recvfrom(s_fd, &packet, sizeof(packet),
        0, (struct sockaddr *) &slavelink, &len);

      /*
      **   Get file name & line number from query
      */
#ifdef TIMING
 gettimeofday(&packet.phase4, NULL); /* start Ph4 4 */
#endif
      sscanf(packet.buffer, "%s %d", file, &line);
      if(strncmp(file, "Terminate", 9) == 0)
         break;                                 /* EXIT requested! */
#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph4 */
 workval.tv_sec -= packet.phase4.tv_sec;
 workval.tv_usec -= packet.phase4.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 packet.phase4.tv_sec = workval.tv_sec;
 packet.phase4.tv_usec = workval.tv_usec;
 gettimeofday(&packet.phase5, NULL); /* start Ph5 */
#endif

      /*
      **   Extract the requested line of text
      */
      if(line > curr) /* false first time */
      {
         line -= curr; /* adjust to offset */
```

LISTING A.3 Continued

```
         curr += line; /* save for next time */
      }
      else
      {
         fclose(fp); /* ignored first time */
         fp = fopen(file, "r");
         curr = line; /* save for next time */
      }
      while(line--)
         fgets(packet.buffer, sizeof(packet.buffer), fp);

#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph5 */
 workval.tv_sec -= packet.phase5.tv_sec;
 workval.tv_usec -= packet.phase5.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 packet.phase5.tv_sec = workval.tv_sec;
 packet.phase5.tv_usec = workval.tv_usec;
 gettimeofday(&packet.phase6, NULL); /* start Ph6 */
#endif

      /*
      **   Respond back to Master
      */
      packet.buffer[strlen(packet.buffer)-1] = '\0';
#ifdef TIMING
 gettimeofday(&workval, NULL); /* end Ph6 */
 workval.tv_sec -= packet.phase6.tv_sec;
 workval.tv_usec -= packet.phase6.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 packet.phase6.tv_sec = workval.tv_sec;
 packet.phase6.tv_usec = workval.tv_usec;
#endif
```

LISTING A.3 Continued

```
      sendto(s_fd, &packet, sizeof(packet), 0,
        (struct sockaddr *) &slavelink, len);
   } /* while(1) */

   /*
   **   Cleanup and exit
   */
   kill(pid, 9); /* stop performance task */
   waitpid(pid);
   close(s_fd);
   fclose(fp);

#ifdef TIMING
 gettimeofday(&workval, NULL); /* end remote */
 workval.tv_sec -= packet.remote.tv_sec;
 workval.tv_usec -= packet.remote.tv_usec;
 if(workval.tv_usec < 0L)
 {
    workval.tv_usec += 1000000L;
    workval.tv_sec -= 1;
 }
 packet.remote.tv_sec = workval.tv_sec;
 packet.remote.tv_usec = workval.tv_usec;
#endif
   return(0);
}
```

This C header file is referenced in the master and the client source.

LISTING A.4 The Master-Slave Interface Header File

```
/*
**   msi.h - master-slave interface header
*/
#define QUERYPORT 9000
#define BROADCAST 5000
#define REGISTERS 5001
#define SLAVEPORT 5002
#define QUERYSIZE 256

struct packet
```

LISTING A.4 Continued

```
{
   /*
   **   Information request packet
   */
#ifdef TIMING
   /* slave phase times */
   struct timeval remote;
   struct timeval phase4;
   struct timeval phase5;
   struct timeval phase6;
#endif
   char buffer[QUERYSIZE]; /* request */
} packet;
```

Near Real-Time Performance

The performance-gathering process is started automatically by the master and each of the slave processes; every cluster node has a copy running, sending local performance information to the central performance report display task's UDP port. (A UDP packet will be discarded if the report process is not running.) No line command operand is required to start the perform task.

LISTING A.5 The Central Performance-Gathering Process

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include "cpm.h"

main(int argc, char *argv[])
{
/*
**   perform.c - Performance gatherer sends node utilization to monitor.c
*/
   struct sockaddr_in soc;
   struct cprStats stats;
   struct hostent *hp;
   long usec0, usec1; /* user */
```

LISTING A.5 Continued

```
long nsec0, nsec1; /* nice */
long ssec0, ssec1; /* system */
long isec0, isec1; /* idle */
long rpkts0, rpkts1; /* rcvd */
long spkts0, spkts1; /* sent */
long tmem, umem;
long rio0, rio1;
long wio0, wio1;
FILE *fptr;
float fcpu;
float fmem;
int ii, jj;
int icpu;
int imem;
int sum;
int fd;
char local[16];
char remote[16];
char buf[256], *p;

if(argc != 2)
{
   printf("\n\tUsage: %s <hostname>\n\n", argv[0]);
   exit(-1);
}
/* get remote hostname */
strcpy(remote, argv[1]);
/* and the local hostname, also */
gethostname(local, sizeof(local));
if((p = strchr(local, '.')) != NULL)
   *p = '\0'; /* trunc domain.nam */

/*
**   Central Performance Reporting socket
*/
fd = socket(AF_INET, SOCK_DGRAM, 0);
memset((char *) &soc, 0, sizeof(soc));
hp = gethostbyname(remote);
memcpy(&soc.sin_addr, hp->h_addr, hp->h_length);
soc.sin_port = htons(PORT);
```

LISTING A.5 Continued

```
soc.sin_family = AF_INET;

/*
**   Initial CPU utilization
*/
fptr = fopen("/proc/stat", "r");
fgets(buf, sizeof(buf), fptr);
buf[strlen(buf)-1] = 0;
sscanf(buf, "%*s %ld %ld %ld %ld",
   &usec0, &nsec0, &ssec0, &isec0);
/*
**   Initial disk IO rates (also in /proc/stat)
*/
while((fgets(buf, sizeof(buf), fptr)) != NULL)
{
   if(strncmp(buf, "disk_", 5) == 0)
      break;
}
sscanf(buf, "%*s %ld", &rio0);
fgets(buf, sizeof(buf), fptr);
sscanf(buf, "%*s %ld", &wio0);
fclose(fptr);

/*
**   Initial network I/O rates
*/
fptr = fopen("/proc/net/dev", "r");
/* skip column header strings in net/dev */
fgets(buf, sizeof(buf), fptr); /* skip */
fgets(buf, sizeof(buf), fptr); /* skip */
fgets(buf, sizeof(buf), fptr); /* skip */
fgets(buf, sizeof(buf), fptr); /* eth0 */
buf[strlen(buf)-1] = 0; /* over '\n' */
buf[6] = ' ';           /* over  ':' */
sscanf(buf, "%*s %ld %*d %*d %*d %*d %ld", &rpkts0, &spkts0);
fclose(fptr);

/*
**   Statistics-Gathering Loop
*/
while(1)
{
   /*
```

LISTING A.5 Continued

```
**   Real MEM utilization
*/
fptr = fopen("/proc/meminfo", "r");
fgets(buf, sizeof(buf), fptr); /* skip hdr */
fgets(buf, sizeof(buf), fptr);
buf[strlen(buf)-1] = 0;
sscanf(buf, "%*s %ld %ld", &tmem, &umem);
fmem = (float) umem / (float) tmem;
stats.rm = (int) (100 * (fmem + .005));
fclose(fptr);

/*
**   Subsequent CPU utilization
*/
fptr = fopen("/proc/stat", "r");
fgets(buf, sizeof(buf), fptr);
buf[strlen(buf)-1] = 0;
sscanf(buf, "%*s %ld %ld %ld %ld",
   &usec1, &nsec1, &ssec1, &isec1);
/* calculate %CPU utilization */
usec1 -= usec0; /* 1-second user CPU counter */
nsec1 -= nsec0; /*          nice             */
ssec1 -= ssec0; /*          system           */
isec1 -= isec0; /*          idle             */
fcpu = (float) (usec1 + nsec1 + ssec1)
     / (float) (usec1 + nsec1 + ssec1 + isec1);
stats.cp = (int) (100 * (fcpu + .005));

/*
**   Subsequent disk I/O rates (also in /proc/stat)
*/
while((fgets(buf, sizeof(buf), fptr)) != NULL)
{
   if(strncmp(buf, "disk_rio", 8) == 0) /* skip */
      break;
}
sscanf(buf, "%*s %ld", &rio1);
/* next string after disk_rio is disk_wio */
fgets(buf, sizeof(buf), fptr);
sscanf(buf, "%*s %ld", &wio1);
fclose(fptr);
```

LISTING A.5 Continued

```
      /* calculate disk I/O rates */
      stats.dr = (rio1 - rio0) / 5;
      stats.dw = (wio1 - wio0) / 5;
      fclose(fptr);

      /*
      **   Subsequent network I/O rates
      */
      fptr = fopen("/proc/net/dev", "r");
      /* skip column header strings in net/dev */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* skip */
      fgets(buf, sizeof(buf), fptr); /* eth0 */
      buf[strlen(buf)-1] = 0; /* over '\n' */
      buf[6] = ' ';           /* over  ':' */
      sscanf(buf, "%*s %ld %*d %*d %*d %*d %ld", &rpkts1, &spkts1);
      /* calc network I/O rates */
      stats.pr = rpkts1 - rpkts0;
      stats.ps = spkts1 - spkts0;
      fclose(fptr);

      /*
      **   Reset values for next iteration
      */
      rpkts0 = rpkts1;
      spkts0 = spkts1;
      usec0 += usec1;
      nsec0 += nsec1;
      ssec0 += ssec1;
      isec0 += isec1;
      rio0 = rio1;
      wio0 = wio1;

      /*
      **   Send statistics at 1-second intervals
      */
      sendto(fd, &stats, sizeof(stats), 0,
         (struct sockaddr *) &soc, sizeof(soc));
      sleep(1);
   }
}
```

The monitor process runs on a noncluster node, named *omega* in our examples. It receives info from each of the perform tasks running on the cluster nodes, displaying overall system performance information for quick visual reference. See Chapter 8 for further details. Following is the line command to start the monitor task:

```
> monitor
```

Note that there are no line command operands. Note also that you may start and stop this task at your convenience because any UDP packets sent to it while it is not running will be discarded.

LISTING A.6 The Centralized Performance-Monitoring Process

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdio.h>
#include <netdb.h>
#include "cpm.h"

main(void)
{
   /*
   **   monitor.c - Centralized Performance Monitoring
   */
   int cp[4], rm[4]; /* central processor, real memory */
   int ps[4], pr[4]; /* packets sent, packets received */
   int dr[4], dw[4]; /* disk reads, disk writes */
   union {
      unsigned long addr;
      char bytes[4];
   } u;
   struct sockaddr_in soc;
   struct cprStats stats;
   int len=sizeof(soc);
   struct hostent *hp;
   int fd, ii, jj;
   int netLoad;
   char buf[256];

   /* establish and initialize UDP socket struct */
   fd = socket(AF_INET, SOCK_DGRAM, 0);
   memset((char *) &soc, 0, sizeof(soc));
```

LISTING A.6 Continued

```
   soc.sin_addr.s_addr = htonl(INADDR_ANY);
   soc.sin_port = htons(PORT);
   soc.sin_family = AF_INET;
   bind(fd, (struct sockaddr *) &soc, len);

   /* initialize stats */
   for(ii=0; ii<4; ii++)
   {
      cp[ii] = 0;
      rm[ii] = 0;
      ps[ii] = 0;
      pr[ii] = 0;
      dr[ii] = 0;
      dw[ii] = 0;
   }

   /*
   **   Centralized Performance Reporting Loop
   */
   while(1)
   {
      for(ii=0; ii<4; ii++)
      {
         recvfrom(fd, &stats, sizeof(stats), 0,
           (struct sockaddr *) &soc, &len);
         /* index is last IP addr digit-1 */
         memcpy(&u.addr, &soc.sin_addr, 4);
         jj = (unsigned int) u.bytes[3];
         cp[jj-1] = stats.cp;
         rm[jj-1] = stats.rm;
         ps[jj-1] = stats.ps;
         pr[jj-1] = stats.pr;
         dr[jj-1] = stats.dr;
         dw[jj-1] = stats.dw;
      }

      /*
      **   Calculate network load
      */
      netLoad = 0;
      for(ii=0; ii<4; ii++)
```

LISTING A.6 Continued

```
        netLoad += (ps[ii] + pr[ii]);
      netLoad /= 2; /* average of each node's number sent and received */

      /*
      **   Draw a network diagram showing cluster performance statistics
      */
      system("clear");
      printf("\n      NEAR REAL-TIME CLUSTER PERFORMANCE STATISTICS     \n");
      printf("                                                        \n");
      printf("                        10Base2                         \n");
      printf(" +----ALPHA-----+           |           +-----BETA-----+\n");
      printf(" |   Cpu    Mem |           |           |   Cpu    Mem |\n");
      printf(" |  %3d%%   %3d%% |Rcvd %-5d | %5d Rcvd|  %3d%%   %3d%% |\n",
        cp[0], rm[0], pr[0], pr[1], cp[1], rm[1]);
      printf(" |   Rio    Wio +-----------+----------+   Rio    Wio |\n");
      printf(" |%6d %6d |Sent %-5d | %5d Sent|%6d %6d |\n",
        dr[0], dw[0], ps[0], ps[1], dr[1], dw[1]);
      printf(" +---10.0.0.1---+           |           +---10.0.0.2---+\n");
      printf("                            |                           \n");
      printf(" +----GAMMA-----+           |           +----DELTA-----+\n");
      printf(" |   Cpu    Mem |           |           |   Cpu    Mem |\n");
      printf(" |  %3d%%   %3d%% |Rcvd %-5d | %5d Rcvd|  %3d%%   %3d%% |\n",
        cp[2], rm[2], pr[2], pr[3], cp[3], rm[3]);
      printf(" |   Rio    Wio +-----------+----------+   Rio    Wio |\n");
      printf(" |%6d %6d |Sent %-5d | %5d Sent|%6d %6d |\n",
        dr[2], dw[2], ps[2], ps[3], dr[3], dw[3]);
      printf(" +---10.0.0.3---+           |           +---10.0.0.4---+\n");
      printf("                        cluster.net\n\n");
      printf("               - Overall Network Loading -\n");
      printf("                %9d Pkts/sec\n", netLoad);
      printf("\n");
   }
}
```

This C header file is referenced in the perform and in the monitor task source code. It describes the performance information packet contents.

LISTING A.7 The Central Performance-Reporting Header File

```
/*
**   report.h - referenced by report.c and perform.c
*/
#define PORT 9090
struct cprStats
{
   int cp; /* cpu utilization */
   int rm; /* mem utilization */
   int ps; /* packets sent */
   int pr; /* packets rcvd */
   int dr; /* disk reads */
   int dw; /* disk writes */
} cprStats;
```

Makefile

This file directs the make utility in building each of the executables needed to run the client, the master-slave interface, and the near real-time performance monitor and display processes. This file is intended to reside under /home/chief/src, where you may issue the following commands:

```
> make client
```

To build just the client executable, issue

```
> make
```

or issue

> make all

to build all the master-slave interface executables.

LISTING A.8 The makefile

```
#
#   makefile - Builds the Master-Slave Interface
#
all:    client master slave monitor perform

client:
```

LISTING A.8 Continued

```
    gcc -O -I../inc -o../bin/client -lm client.c

master:
    gcc -O -I../inc -o../bin/master master.c

slave:
    gcc -O -I../inc -o../bin/slave slave.c

monitor:
    gcc -O -I../inc -o../bin/monitor monitor.c

perform:
    gcc -O -I../inc -o../bin/perform perform.c

# Add execution phase timing code
#
timing:       client masterT slaveT monitor perform

masterT:
    gcc -O -DTIMING -I../inc -o../bin/master master.c

slaveT:
    gcc -O -DTIMING -I../inc -o../bin/slave slave.c

clean:
    rm -f ../bin/client
    rm -f ../bin/master
    rm -f ../bin/slave
    rm -f ../bin/monitor
    rm -f ../bin/perform
```

Index